LUKAS MAX WEBER

NOVEL ARCHITECTURES FOR OFFLOADING AND ACCELERATING COMPUTATIONS IN ARTIFICIAL INTELLIGENCE AND BIG DATA

TECHNISCHE
UNIVERSITÄT
DARMSTADT

NOVEL ARCHITECTURES FOR OFFLOADING AND ACCELERATING COMPUTATIONS IN ARTIFICIAL INTELLIGENCE AND BIG DATA

Doctoral thesis
by Lukas Max Weber, M.Sc.
from Speyer, Germany

submitted in fulfilment of the requirements for the
degree of Doctor of Engineering (Dr.-Ing.)

Computer Science Department
Technische Universität Darmstadt

Reviewers
Prof. Dr.-Ing. Andreas Koch
Assoc. Prof. Oliver Sinnen

Date of the oral exam
15th of September, 2023

Date of submission: 21st of July, 2023

Darmstadt, 2023
D 17

Lukas Max Weber: *Novel Architectures for Offloading and Accelerating Computations in Artificial Intelligence and Big Data*
Dissertation (Doctoral Thesis)
Darmstadt, Technische Universität Darmstadt
Date of the oral exam: 15th of September, 2023

Please cite this work as:
URN: urn:nbn:de:tuda-tuprints-243490
URL: https://tuprints.ulb.tu-darmstadt.de/24349

This document is provided by TUprints,
The Publication Service of the Technische Universität Darmstadt
https://tuprints.ulb.tu-darmstadt.de
Year published in TUprints: 2023

Urheberrechtlich geschützt / In copyright
https://rightsstatements.org/page/InC/1.0/

ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, July 2023

Lukas Max Weber

ABSTRACT

Due to the end of Moore's Law and Dennard Scaling, performance gains in general-purpose architectures have significantly slowed in recent years. While raising the number of cores has been a viable approach for further performance increases, Amdahl's Law and its implications on parallelization also limit further performance gains. Consequently, research has shifted towards different approaches, including domain-specific custom architectures tailored to specific workloads.

This has led to a new golden age for computer architecture, as noted in the Turing Award Lecture by Hennessy and Patterson, which has spawned several new architectures and architectural advances specifically targeted at highly current workloads, including Machine Learning. This thesis introduces a hierarchy of architectural improvements ranging from minor incremental changes, such as High-Bandwidth Memory, to more complex architectural extensions that offload workloads from the general-purpose CPU towards more specialized accelerators. Finally, we introduce novel architectural paradigms, namely Near-Data or In-Network Processing, as the most complex architectural improvements.

This cumulative dissertation then investigates several architectural improvements to accelerate Sum-Product Networks, a novel Machine Learning approach from the class of Probabilistic Graphical Models. Furthermore, we use these improvements as case studies to discuss the impact of novel architectures, showing that minor and major architectural changes can significantly increase performance in Machine Learning applications.

In addition, this thesis presents recent works on Near-Data Processing, which introduces Smart Storage Devices as a novel architectural paradigm that is especially interesting in the context of Big Data. We discuss how Near-Data Processing can be applied to improve performance in different database settings by offloading database operations to smart storage devices. Offloading data-reductive operations, such as selections, reduces the amount of data transferred, thus improving performance and alleviating bandwidth-related bottlenecks.

Using Near-Data Processing as a use-case, we also discuss how Machine Learning approaches, like Sum-Product Networks, can improve novel architectures. Specifically, we introduce an approach for offloading Cardinality Estimation using Sum-Product Networks that could enable more intelligent decision-making in smart storage devices. Overall, we show that Machine Learning can benefit from developing novel architectures while also showing that Machine Learning can be applied to improve the applications of novel architectures.

Aufgrund des Endes von Moore's Law und Dennard Scaling haben Leistungszuwächse in typischen Architekturen signifikant abgenommen. Während das Hinzufügen zusätzlicher Kerne anfangs ein sinnvoller Ansatz für zusätzliches Leistungswachstum war, so sind die entsprechenden Zuwächse aufgrund von Amdahl's Law und dessen Auswirkungen auf Parallelisierung ebenfalls begrenzt. Als Konsequenz musste sich die Forschung in Richtung neuer Ansätze, wie beispielsweise domänen-spezifische Architekturen, orientieren.

Die entsprechende Umorientierung wird in der Turing-Award-Vorlesung von Hennessy und Patterson als ein neues goldenes Zeitalter der Computerarchitektur beschrieben, welches schon damals einige neuartige und hochspezialisierte Architekturen für künstliche Intelligenz hervorgebracht hatte. Diese Thesis stellt eine Hierarchie entsprechender architektonischer Anpassungen auf, welche von kleinen inkrementellen Verbesserungen, wie der Verwendung von High-Bandwidth Memory, zu deutlich komplexeren architektonischen Erweiterungen reicht, die das Offloading bestimmter komplexer Aufgaben ermöglichen. Zum Schluss wird auf neue architektonische Paradigmen, wie Near-Data oder In-Network Processing, eingegangen, die die komplexeste Form von architektonischen Veränderungen repräsentieren.

Diese kumulative Dissertation untersucht dann eine Reihe entsprechender architektonischer Verbesserungen zur Beschleunigung von Sum-Product Networks, die ein Beispiel für neuartige Ansätze im Bereich des maschinellen Lernens sind. Darüber hinaus nutzen wir die Verbesserungen zur Diskussion entsprechender Ansätze und wie kleine und große architektonische Änderungen die Leistung in Anwendungen des maschinellen Lernens erhöhen können.

Zusätzlich präsentiert diese Thesis neuartige Forschungsarbeiten aus dem Bereich des Near-Data Processing, welche sich mit intelligenten Speichermedien befassen, die insbesondere in Big-Data Anwendungen interessant sind. Wir diskutieren, wie Near-Data Processing angewendet werden kann, um die Leistung in verschiedenen Datenbankszenarien zu erhöhen, indem Datenbankoperationen auf intelligente Speichermedien ausgelagert werden. Insbesondere die Auslagerung datenreduzierender Operationen, wie Selektionen, verringert die Menge an zu übertragenden Daten, wodurch die Leistung erhöht werden kann.

In der Umkehrung betrachten wir auch, wie Ansätze des maschinellen Lernens verwendet werden können, um neuartige Architekturen, wie beispielsweise intelligente Speichermedien, zu verbessern. Speziell betrachten wir die Verwendung von Sum-Product Networks für das

Offloading der Cardinality Estimation, die in intelligenten Speichermedien verwendet werden könnte, um bessere Entscheidungen zu treffen. Insgesamt zeigen wir, dass sowohl maschinelles Lernen von neuartigen Architekturen profitieren kann, aber auch, dass maschinelles Lernen eingesetz werden kann, um die Anwendung von neuartigen Architekturen zu verbessern.

PUBLICATIONS

The following publications are part of this cumulative dissertation and can be found in Chapter 8 to Chapter 15.

[1] Lukas Weber, Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting, and Andreas Koch. "Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs." In: *International Conference on Field-Programmable Technology (FPT)*. 2019, pp. 251–254. DOI: 10.1109/ICFPT47387.2019.00040.

[2] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. "Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs." In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020, pp. 75–83. DOI: 10.1109/FCCM48280.2020.00020.

[3] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. "NKV: Near-Data Processing with KV-Stores on Native Computational Storage." In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 2020. DOI: 10.1145/3399666.3399934.

[4] Tobias Vinçon, Lukas Weber, Arthur Bernhardt, Christian Riegger, Sergey Hardock, Christian Knoedler, Florian Stock, Leonardo Solis-Vasquez, Sajjad Tamimi, and Andreas Koch. "nKV in Action: Accelerating KV-Stores on Native Computation Storage with Near-Data Processing." In: *Proceedings of the VLDB Endowment, Volume 13*. 2020. DOI: 10.14778/3415478.3415524.

[5] Lukas Weber, Lukas Sommer, Leonardo Solis-Vasquez, Tobias Vinçon, Christian Knödler, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. "A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems." In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 136–143. DOI: 10.1109/IPDPSW52791.2021.00028.

[6] Marco Hartmann, Lukas Weber, Johannes Wirth, Lukas Sommer, and Andreas Koch. "Optimizing a Hardware Network Stack to Realize an In-Network ML Inference Application." In: *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2021, pp. 21–32. DOI: 10.1109/H2RC54759.2021.00008.

[7] Lukas Weber, Johannes Wirth, Lukas Sommer, and Andreas Koch. “Exploiting High-Bandwidth Memory for FPGA- Acceleration of Inference on Sum-Product Networks.” In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 112–119. DOI: 10.1109/IPDPSW55747.2022.00028.

[8] Lukas Weber, Yannick Lavan, Torben Kalkhof, Carsten Heinz, and Andreas Koch. “Exploiting Sum-Product Network Inference to Offload Database Query Cardinality Estimation to FPGA-based Computational Storage Devices.” In: Submitted for Publication in ASPLOS 2024: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2024.

In addition to the publications that are part of this thesis, the author has contributed to the following peer-reviewed publications:

[1] Micha Ober, Jaco Hofmann, Lukas Sommer, Lukas Weber, and Andreas Koch. "High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud." In: *Fifth International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2019, pp. 26–33. doi: 10.1109/H2RC49586.2019.00009.

[2] Julian Oppermann, Lukas Sommer, Lukas Weber, Melanie Reuter-Oppermann, Andreas Koch, and Oliver Sinnen. "SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis." In: *International Conference on Field-Programmable Technology (FPT)*. 2019, pp. 36–44. doi: 10.1109/ICFPT47387.2019.00013.

[3] T. Vinçon, A. Bernhardt, L. Weber, A. Koch, and I. Petrov. "On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems." In: *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. 2020, pp. 109–114. doi: 10.1109/ICDEW49219.2020.00009.

[4] Lukas Weber, Tobias Vincon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. "On the necessity of explicit cross-layer data formats in near-data processing systems." In: *Distributed and Parallel Databases* (2021). doi: 10.1007/s10619-021-07328-z.

[5] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. "The TaPaSCo Open-Source Toolflow." In: *Journal of Signal Processing Systems* (2021). doi: 10.1007/s11265-021-01640-8.

[6] Hanna Kruppe, Lukas Sommer, Lukas Weber, Julian Oppermann, Cristian Axenie, and Andreas Koch. "Efficient Operator Sharing Modulo Scheduling for Sum-Product Network Inference on FPGAs." In: *Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*. 2021, pp. 242–258. doi: 10.1007/978-3-031-04580-6_16.

[7] Christian Knödler, Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Leonardo Solis-Vasquez, Lukas Weber, and Andreas Koch. "A Cost Model for NDP-Aware Query Optimization for KV-Stores." In: *17th International Workshop on Data Management on New Hardware (DaMoN)*. 2021. doi: 10.1145/3465998.3466013.

[8] Tobias Vinçon, Christian Knödler, Arthur Bernhardt, Leonardo Solis-Vasquez, Lukas Weber, Andreas Koch, and Ilia Petrov. "Result-Set Management for NDP Operations on Smart Storage." In: *18th International Workshop on Data Management on New Hardware (DaMoN)*. 2022. doi: 10.1145/3533737.3535097.

[9] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. "Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads." In: *Proceedings of the VLDB Endowment, Volume 15*. 2022. DOI: 10.14778/3547305.3547307.

ACKNOWLEDGMENTS

I would like to express my gratitude to my parents and my siblings for their unwavering support and encouragement throughout my academic journey. Without their love and support, this work would not have been possible.

I would also like to extend my sincere gratitude to Prof. Koch, who has guided and mentored me throughout my journey. His guidance and insight have inspired my academic work.

In addition, I would like to thank Oliver Sinnen for taking the time to serve as second reviewer on the following thesis.

Additionally, I would like to thank all my colleagues at the Embedded Systems and Applications Group at TU Darmstadt for their support and for extensive technical discussions on my and their research. Specifically, I would like to point out the great mentorship and collaboration of Lukas Sommer and Julian Oppermann that originally got me interested in Compilers, Hardware Engineering and corresponding research. Additionally, Carsten Heinz, Torben Kalkhof, Christoph Spang and Yannick Lavan for the technical and mathematical support.

Lastly, I would like to thank Lea Döpp and all my other friends that have supported me in this endeavor.

CONTENTS

I Synopsis
1 Introduction 3
2 Artificial Intelligence and Novel Architectures 7
2.1 Moore's Law, Dennard Scaling and the End of the Line 7
2.2 Domain-specific Architectures 9
2.3 Artificial Intelligence Architectures 10
3 Architectural Optimizations and Adaptations 13
3.1 General-Purpose Architectures 13
3.2 Incremental Improvements 14
3.3 Architectural Extensions and Offloading 15
3.4 Novel Architectural Paradigms 17
3.4.1 Data-Flow Processing 18
3.4.2 In-Network Processing 19
3.4.3 Near-Data Processing 20
4 Accelerating Sum-Product Network Inference 25
4.1 Introduction to Sum-Product Networks 25
4.1.1 Inference 26
4.1.2 Learning and Training 27
4.2 Accelerated Inference 28
4.2.1 CPU- and GPU-based Acceleration 28
4.2.2 FPGA-based Acceleration 30
4.2.3 In-Network Acceleration 33
5 Near-Data Processing and Smart Storage 37
5.1 Introduction to nKV 37
5.2 Prior and Related Work 38
5.3 Offloading and Acceleration in Smart Storage Systems 39
5.4 Discussion 41
6 Artificial Intelligence in Smart Storage Systems 43
6.1 Result Set Handling 43
6.2 Cardinality Estimation and Sum-Product Networks . . 44
7 Conclusion 47

Bibliography 48

II Sum-Product Networks
8 Logarithmic Arithmetic for SPN Inference on FPGAs 59
8.1 Introduction 60
8.2 Sum-Product Networks 60
8.2.1 Model Representation 61
8.2.2 Inference 61
8.3 Prior Work 62

8.3.1 Logarithmic Number Scale on FPGA 62
8.3.2 SPN Inference on FPGA 62
8.4 Approach 62
8.4.1 LNS Multiplication 63
8.4.2 LNS Addition 63
8.5 Evaluation 64
8.5.1 Benchmarks 64
8.5.2 Parameters 64
8.5.3 FPGA Resource Consumption 65
8.5.4 Performance Evaluation 65
8.6 Conclusion & Outlook 68
9 Comparison of Arithmetic Number Formats for SPN Inference 71
9.1 Introduction 72
9.2 SPN Background 73
9.2.1 Model Representation 74
9.2.2 Inference 74
9.3 Arithmetic Number Formats 75
9.3.1 Fixed Point 75
9.3.2 Floating Point 76
9.3.3 Posit 76
9.3.4 Logarithmic Number System 77
9.4 Design-Space Exploration using Software Emulation . 78
9.4.1 Implementation 78
9.4.2 Accuracy Results 79
9.5 Implementation of Hardware Arithmetic Operators . . 81
9.5.1 Floating Point 82
9.5.2 Logarithmic Number System 84
9.5.3 Posit 84
9.6 Evaluation 84
9.6.1 Benchmarks 84
9.6.2 FPGA Implementation Results 85
9.6.3 Power Evaluation 86
9.6.4 Performance Evaluation 88
9.7 Conclusion & Outlook 90
10 In-Network ML Inference Application 95
10.1 Introduction 96
10.2 Related Work 97
10.3 Background 98
10.3.1 Hardware TCP/IP Stack 98
10.3.2 The TaPaSCo Framework 99
10.4 Implementation 100
10.4.1 Modification and Extension of the Network Stack 100
10.4.2 Integration with the TaPaSCo Framework 101
10.4.3 Design Primitives for Network Access 102
10.4.4 Creating a TCP/IP-capable Design 103

10.4.5 Fast Hardware/Software-Co-Simulation of TCP/IP-capable Applications . . . 104
10.4.6 FSM-based Simulator Architecture . . . 107
10.4.7 "In Circuit" Emulation . . . 107
10.5 Evaluation . . . 108
10.5.1 Experimental Setup . . . 108
10.5.2 Throughput . . . 109
10.5.3 Latency . . . 111
10.5.4 RX Bypass . . . 111
10.5.5 Resource Utilization . . . 113
10.6 Case Study: In-Network Acceleration of Sum-Product Network Inference . . . 114
10.6.1 Sum-Product Network Background . . . 115
10.6.2 Streaming-based Accelerator . . . 116
10.6.3 Network Integration . . . 116
10.6.4 Experimental Evaluation . . . 117
10.7 Conclusion & Outlook . . . 118
11 Using HBM for FPGA-Acceleration of SPNs 125
11.1 Introduction . . . 126
11.2 Background . . . 127
11.2.1 Sum-Product Networks . . . 127
11.2.2 High-Bandwidth Memory . . . 128
11.3 Approach . . . 130
11.3.1 Motivation . . . 130
11.3.2 SPN-Accelerator . . . 131
11.4 Implementation . . . 132
11.4.1 On-Device Architecture . . . 132
11.4.2 Parallel Runtime . . . 132
11.5 Evaluation . . . 133
11.5.1 Resource Utilization . . . 134
11.5.2 Performance Scaling . . . 136
11.5.3 Scaling Limitations . . . 138
11.5.4 End-to-End Performance . . . 140
11.6 Related Work . . . 142
11.7 Conclusion . . . 143

III Near-Data Processing
12 nKV: NDP with KV-Stores on Native Computational Storage 149
12.1 Introduction . . . 150
12.2 Background . . . 152
12.3 Architecture of nKV . . . 154
12.3.1 NDP Interface Extensions . . . 155
12.3.2 In-situ Data Access and Interpretation . . . 156
12.3.3 Operations and Algorithms . . . 157
12.3.4 Data Consistency, Database Maintenance and NDP 158
12.3.5 Result Set Handling . . . 158

12.4 Hardware-Architecture . . . 159
12.5 Hardware-Acceleration . . . 160
12.6 Evaluation . . . 164
12.6.1 Low-level Flash Properties . . . 164
12.6.2 Experiment 1: Lean Native Stack . . . 165
12.6.3 Experiment 2: Data Transfer Reduction . . . 166
12.6.4 Experiment 3: Native Computational Storage . . . 166
12.6.5 Experiment 4: Execution Parallelism . . . 167
12.7 Related Work . . . 167
12.8 Conclusion . . . 168
13 nKV in Action 173
13.1 Introduction . . . 173
13.2 Architecture of nKV . . . 175
13.3 Demonstration Walk-through . . . 177
13.3.1 Demonstration Walk-Through . . . 178
13.4 Related Work . . . 181
13.5 Conclusion . . . 181
14 Auto-Gen of FPGA-based NDP Accelerators 185
14.1 Introduction . . . 186
14.2 Motivation . . . 187
14.3 Near-Data Processing Background . . . 188
14.3.1 Background: Key-Value Stores . . . 188
14.3.2 nKV: Near-Data Processing Architecture . . . 189
14.4 Near-Data Processing Accelerator Generation . . . 190
14.4.1 NDP Accelerator Architecture Template . . . 191
14.4.2 Automatic Generation of NDP Accelerators . . . 192
14.4.3 Automatic Generation of the Software Interface 196
14.5 Evaluation . . . 197
14.6 Related Work . . . 200
14.7 Conclusion & Outlook . . . 202

IV Sum-Product Networks in Near-Data Processing
15 SPNs for Cardinality Estimation 207
15.1 Introduction . . . 208
15.2 Background . . . 210
15.2.1 Sum-Product Networks . . . 210
15.2.2 Cardinality Estimation . . . 211
15.3 Related Work . . . 211
15.4 SPNs and Cardinality Estimation . . . 212
15.4.1 Estimating Query Cardinalities . . . 212
15.4.2 Hardware-Specific Model Optimization . . . 214
15.4.3 Empirical Analysis . . . 214
15.5 SPN Accelerators . . . 217
15.5.1 System Integration . . . 219
15.6 Evaluation . . . 220
15.6.1 Benchmarks . . . 220

15.6.2 Fixed-Point Encoding 222
15.6.3 Performance 226
15.7 Conclusion 227

LIST OF FIGURES

Figure 1.1 Number of parameters for well-knwon Large Language Models. 4
Figure 2.1 Maximum possible speedup as dictated by Amdahl's Law, depending on the number of processor cores and the sequential portion of the code. Replotted based on [22]. 9
Figure 3.1 Simplified general-purpose Architecture of a typical contemporary computer. 14
Figure 3.2 Examples for incremental improvements to the baseline system architecture. 16
Figure 3.3 Baseline architecture with an additional Tensor Processing Unit-extension for Machine Learning (ML) workloads. 17
Figure 3.4 Baseline architecture that is connected to a network with access to In-Network Processing (INP) compute capabilities. 21
Figure 4.1 Example SPN, taken from Chapter 10 26
Figure 4.2 Inference example in an SPN, representing the joint probability distribution $P(A, B)$. In *joint* inference, all histograms output a corresponding value (a), while in *marginal* inference some histograms are marginalized and always output the value 1.0 (b). Taken from Chapter 15. . . . 27
Figure 4.3 Floating Point binary format, simplified from Chapter 9. 31
Figure 4.4 Logarithmic Number Scale binary format. . . . 31
Figure 4.5 Inference Throughput of Field-Programmable Gate Array (FPGA)-, Central Processing Unit (CPU)- and Graphics Processing Unit (GPU)-based implementations, taken from Chapter 9. 34
Figure 5.1 Architecture of the nKV system, taken from Chapter 12. 40
Figure 5.2 Baseline architecture with a smart Solid State Drive (SSD), containing on-board Dynamic Random-Access Memory (DRAM) and flash memory, corresponding memory controllers and additional compute capabilities in the form of accelerators. 41
Figure 8.1 Example of a valid SPN, capturing the joint probability distribution of the variables x_1, x_2 and x_3. 61

Figure 8.2 Throughput of the CPU-, GPU- and both FPGA-implementations in samples/μs. Each group represents an example SPN. The single outlier is the CPU-Throughput for example NIPS5 which is cut off from more than 2x its size to fit. 67
Figure 9.1 Example of a valid SPN. 74
Figure 9.2 Posit binary format. 77
Figure 9.3 Development of maximum error depending on arithmetic format configuration. 80
Figure 9.4 FP Adder dual path mantissa processing . . . 82
Figure 9.5 Comparison of FP operators with prior work. . 86
Figure 9.6 Throughput comparison of the CPU, GPU and FPGA-implementations. 89
Figure 10.1 Architecture of a TCP/IP-capable PE with connections to TaPaSCo subsystems. 104
Figure 10.2 Simulator architecture for TCP/IP-capable TaPaSCo PEs 106
Figure 10.3 Throughput of TCP, UDP, and Ethernet, for different payload sizes per packet. Note that the y-axis does not start at zero. 109
Figure 10.4 Measured throughput and theoretical maximum for different protocols. Except for Aurora, all protocols use a payload size of 4 KiB 111
Figure 10.5 RTT latencies using different protocols. For TCP, the duration of the handshake is not included. 112
Figure 10.6 TCP throughput with and without receive buffer bypass, for HBM and BRAM based buffer implementations. The MSS is 4 KiB, and the corresponding theoretical maximum throughput is marked. 113
Figure 10.7 Latency results of a TCP ping test, with and without buffer bypass, for HBM and BRAM based buffer implementations. The figure shows the duration of the TCP handshake (HS) and the subsequent RTT of a ping packet. 114
Figure 10.8 Example for the graph structure of a Sum-Product Network, capturing the joint probability distribution over a set of variables. 115
Figure 10.9 Throughput of different SPN variants using different network protocols for input and output data transfer. 118
Figure 11.1 Different types of SPNs. 128

Figure 11.2 Maximum throughput when issuing linear read and write accesses in parallel to one HBM memory channel for two different configurations and different request sizes. The first configuration runs the block generating the accesses with the 450 MHz clock used by the HBM and natively connects both. The second configuration runs the PE at *half* the clock frequency but the interface width is *doubled*. An AXI Smart Connect is used to perform clock- and data-width-conversion. 129
Figure 11.3 Architecture of the SPN-Accelerator 131
Figure 11.4 Comparison of peak performance in samples per second. On the left, host-to-device data-transfers are excluded to disregard a PCIe-based bottleneck. On the right, actual end-to-end performance is measured. Note that the inclusion of the data-transfer time leads to severely skewed scaling. 137
Figure 11.5 Scaling potential of the presented architecture under the assumption that logic resources are sufficient and host-to-device bandwidth is available. For each benchmark, we depict the required memory throughput depending on the number of instantiated SPN-cores. The throughput is compared against the maximum throughputs of a single HBM channel as measured in Fig. 11.2. Additionally, we compare against the practical maximum throughput scaled from our single-channel benchmarks (HBM $\max_p$), and the theoretical limit quoted by the vendor (HBM $\max_t$). 139
Figure 11.6 Peak performance measurements for the different benchmark SPNs on different target platforms. The number of samples per second is calculated from the end-to-end execution time. For AWS F1, V100 and HBM, host-to-device data transfers are *included* in the runtime. . . . 141
Figure 12.1 KV-Store transferring data along a traditional I/O stack (a); and (b) nKV executing operations in-situ on native computational storage. 150
Figure 12.2 Conceptual organization of the multi-level LSM-Trees in RocksDB/LevelDB. 153
Figure 12.3 Architecture of nKV 154
Figure 12.4 In-situ access and data interpretation in nKV, based on layout accessors and format parsers. 157

Figure 12.5 A simplified view of the architecture of the *COSMOS+* OpenSSD [7], including the proposed extension. 160
Figure 12.6 The overall Microarchitecture of the proposed Parser Processing Elements. 161
Figure 12.7 Break-Down of Execution Times within the NDP Stack with HW support. 163
Figure 12.8 GET execution times for Blk, NDP:SW and NDP:SW+HW. 165
Figure 12.9 SCAN execution times for Blk, NDP:SW and NDP:SW+HW. 166
Figure 12.10 Betweenness centrality (BC) execution times for Blk, NDP:SW and NDP:SW+HW. 167
Figure 12.11 Betweenness centrality (BC) execution times for for NDP:SW+HW using 3, 5 and 7 instances of the ref-PE hardware parser. 168
Figure 13.1 KV-Store transferring data along a traditional I/O stack (a); and (b) nKV executing operations in-situ on native computational storage. 174
Figure 13.2 Architecture of nKV 175
Figure 13.3 In-situ access and data interpretation in nKV, based on layout accessors and format parsers. 176
Figure 13.4 COSMOS+ and the Demonstration Setup . . . 177
Figure 13.5 Interactive GUI. 178
Figure 13.6 Betweenness Centrality: (A) BC on different stacks; (B) BC with different levels of parallelism; (C) BC execution time vs number of relevant edges (complexity). 179
Figure 13.7 GET Latencies on different stacks. 179
Figure 13.8 SCAN performance: (A) SCAN on different stacks 180
Figure 13.9 SCAN performance: (B) Data Transfer Volume 180
Figure 14.1 Comparison of traditional KV-store and the nKV-architecture with native computational storage and Near-Data Processing. 190
Figure 14.2 Overall system architecture based on the Cosmos+ OpenSSD platform, extend with FPGA-based NDP accelerators. 191
Figure 14.3 Architectural template used by the generated NDP accelerators. 192
Figure 14.4 Example Code showing how a PE is defined for automatic generation. The generated PE will automatically transform data from the `Point3D`-type to `Point2D`-type, discarding the field `x`. Additionally, the `Point3D`-structs can be filtered using predicates on all of the present fields (`x`, `y` and `z`). 193

Figure 14.5 Internal structure of the Filtering Unit. 195
Figure 14.6 Snippet from the generated software-interface that can be used to interact with the PEs. . . . 197
Figure 14.7 Execution times of the GET and SCAN operations, comparing our work to the work provided in [18]. For both Operations execution is executed with HW-acceleration (HW) and without (SW). 198
Figure 14.8 Out-of-Context Slice Utilization of generated PEs in correlation to the size of the processed tuples. Half refers to accelerators using the prefixing, whereas Full refers to the ones using all data. 200
Figure 14.9 Out-of-Context Slice Utilization (in percent) of generated PEs in correlation to the number of filtering stages. Additional stages increase resource requirement in a linear fashion, but provide more flexibility. The use of string-prefixing (Half) has only minor impact. 201
Figure 15.1 Inference example in an SPN, representing the joint probability distribution $P(A, B)$. In *joint* inference, all histograms output a corresponding value (a), while in *marginal* inference some histograms are marginalized and always output the value 1.0 (b). 210
Figure 15.2 Observed estimation error for range queries on all NIPS datasets. 215
Figure 15.3 `FxHistogram` Module. Dotted Lines indicate pipeline stages. The number of pipeline stages for `FxSub` and `FxAdder` depends on the used encoding. 218
Figure 15.4 Arithmetic errors of the encoding depending on the combination of SPN and encoding. The darker bar in front is the maximum *measured* error, while the lighter bar behind is the maximum *theoretical* error. 221
Figure 15.5 Changes in resource utilization of using a 24 bit fixed-point encoding compared against the closest matching custom floating-point (CFP) encoding from [14]. DSP utilization is identical for both encodings. 221
Figure 15.6 FPGA Resource Utilizations of the different accelerators in a 24 bit low-latency configuration on the different devices. 224

Figure 15.7 Throughput of the throughput-optimized accelerator variants on VC709 and Alveo U280 in comparison to single-core prior work on VC709 published at FCCM [14]. 224
Figure 15.8 End-to-End inference latencies (less is better) of TaPaSCo-based architectures in comparison against baremetal NIPS40-accelerators on Ultra96 and COSMOS+ and the best reference CPU implementation of an RSPN from prior work [7]. 225

LIST OF TABLES

Table 8.1 FPGA implementation results using double-precision floating-point (FP) and logarithmic number scale (LNS) arithmetic operators, respectively. For brevity those numbers are given as usage relative to the resources available on the FPGA in percent. 66
Table 9.1 Arithmetic format configuration for each benchmark. 79
Table 9.2 Comparison of the per-operator resource requirements and pipeline depth. 81
Table 9.3 Comparison of the FP adder before and after optimization. 83
Table 9.4 FPGA implementation results for all benchmarks. 87
Table 9.5 Power consumption of the datapath. 88
Table 10.1 Overview of resource utilization of different TCP/IP stack configurations. 114
Table 10.2 Degree of replication of different NIPS-SPNs used in the experimental evaluation. 117
Table 11.1 Resource Utilization of the comparable NIPS-based benchmarks. "New" columns show the result of this work, while [11] refers to our prior work. 135
Table 12.1 FPGA-Resource Utilization of the Baseline and extended Architectures, including hierarchical utilizations of relevant sub-modules. 162
Table 12.2 Flash Latencies and Bandwidth (BW) of the *COSMOS+ OpenSSD* for different levels of parallelism. 165

Table 14.1 FPGA Resource Utilization of the PEs used in [18] and our work. The design contains the complete COSMOS+ OpenSSD platform as well as 1 paper-PE and 7 ref-PEs. 199
Table 15.1 Datasets and the corresponding number of examined queries 216
Table 15.2 Functionality of `FxHistogram` and its submodules. *b* denotes the memory storing the accumulated histogram buckets. 218
Table 15.3 Hardware Platforms used in Evaluation. Latency measured for NIPS40. 220
Table 15.4 SPN and Data Characteristics of the used benchmarks 221
Table 15.5 FPGA Resources of the different platforms used in Evaluation 226
Table 15.6 Absolute FPGA Resource Utilizations on different platforms 228

ACRONYMS

AI	Artificial Intelligence
AIE	Artificial Intelligence Engine
APU	Accelerated Processing Unit
AQP	Approximate Query Processing
BRAM	Block Random-Access Memory
CFG	Control-Flow Graph
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DL	Deep Learning
DMA	Direct Memory Access
DFG	Data-Flow Graph
DFP	Dataflow Processing
DRAM	Dynamic Random-Access Memory
DSA	Domain-specific Architecture
EM	Expectation Maximization
FPGA	Field-Programmable Gate Array
FTL	Flash Transaction Layer
GPU	Graphics Processing Unit
HBM	High-Bandwidth Memory
HDD	Hard Disk Drive
IC	Integrated Circuit
INP	In-Network Processing
IPU	Intelligence Processing Unit
ISA	Instruction Set Architecture
LLM	Large Language Model
LLVM	Low-Level Virtual Machine
LUN	Logical Unit
MIMD	Multiple Instruction, Multiple Data
ML	Machine Learning
MLIR	Multi-Level Intermediate Representation
MMU	Matrix Multiplication Unit
NDP	Near-Data Processing

NIC Network Interface Card

NN Neural Network

OS Operating System

PGM Probabilistic Graphical Model

RAT-SPN Random Tensorized Sum-Product Network

RDMA Remote Direct Memory Access

RSPN Relational Sum-Product Network

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SoC System-on-a-Chip

SPN Sum-Product Network

SPNC SPN Compiler

SSD Solid State Drive

TPU Tensor Processing Unit

Part I

SYNOPSIS

1 INTRODUCTION

Over the last years, ML and Big Data have become two of the most active fields of research within computer science. Especially recent developments like ChatGPT and similar models have sparked a strong and growing interest in the fields of ML and Artificial Intelligence (AI), also by the general public. A significant driver for the success of ML are the continuously increasing sizes of corresponding models and training data sets. In addition, specific improvements to existing and the development of novel hardware architectures have also contributed to the success. Interestingly, this trend goes both ways: Specific novel architectures tailored towards ML are built, and ML-based methods are used to improve existing architectures and their applications. Examples of this development can be found almost everywhere: From many big companies like Google developing custom chips for their ML applications, such as Google's Tensor Processing Unit (TPU) [26, 27], over the inclusion of AI Engines in AMD's newest Ryzen 7000 chips [2], to ML-based branch prediction in regular CPUs [46]. As a result, ML has become an integral part of computer science and impacts us all on a regular basis.

While many parts of regular systems have already been improved by and for ML, one part that is still a lot less tailored towards ML is persistent storage. The only significant improvement to persistent storage in recent years was the development of SSDs, which effectively replace Hard Disk Drives (HDDs) in high-performance systems due to the lower latencies and increased throughput. In addition, eliminating moving parts also means that SSDs typically have a longer lifetime or mean time to failure. In contrast, ML-related improvements to other components are much more common. This is especially interesting when we consider the growing relevance of persistent storage. Simply considering Large Language Models: The relatively early ELMo (released 2018, [41]) has 94 million parameters. Only about a year later, GPT-2 was released and already had 1.5 billion parameters [44]. The currently largest known model is Megatron-Turing NLG [48] with 530 billion parameters, which was released in 2022. Please note that it is suspected that GPT-4 is already using at least a trillion parameters. While a technical report on GPT-4 does not include a specific number [39], it would fit the current trend, which is also shown in Fig. 1.1.

While Large Language Models (LLMs) are still relatively new, we currently see an exponential growth in the number of parameters. Accordingly, the size of models and their corresponding training data is also increasing. For example, GPT-3 was trained on 45 TB of data

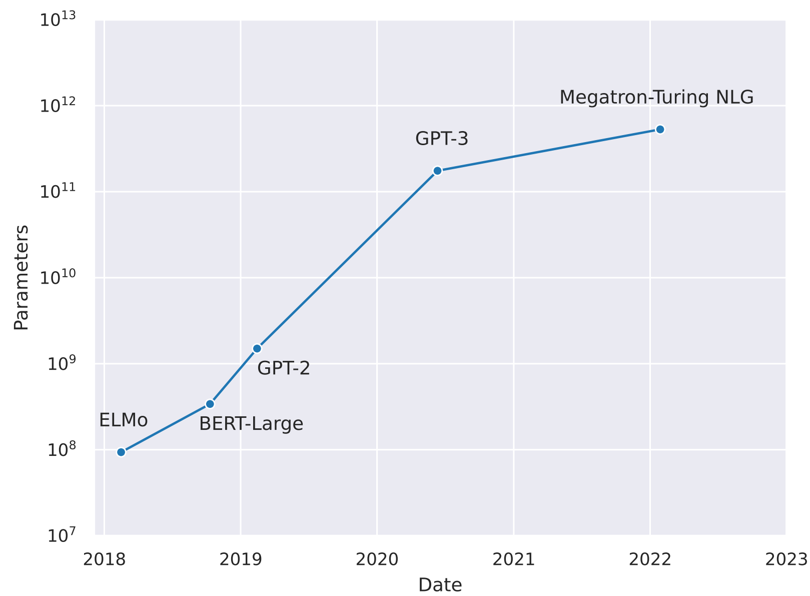


Figure 1.1: Number of parameters for well-knwon Large Language Models.

and requires about 800 GB of storage capacity [5]. The sheer size of these models already motivates improvements in persistent storage. However, in addition to simply storing LLMs, intelligent storage systems might also become more relevant in the future, as they could help pre-process the vast amount of data required for training in AI applications. Moreover, matching improvements could also carry over into different Big Data applications.

One possible improvement to make storage systems smarter is Near-Data Processing (NDP), which aims to move computation closer to persistent storage to reduce unnecessary data movement, improving storage performance. In corresponding devices, this is possible because NDP can reduce the required bandwidth of different applications, as data-reductive operations like selections and projections can be applied before data is transported using the slower system buses. As a result, reduced pressure allows these buses to be used for other tasks. Moreover, moving computational load away from the CPU can also increase responsiveness in applications like databases.

Overall, it is clear that advances in computer architecture can significantly improve applications from the fields of Big Data and AI. Therefore, this thesis investigates several recent advances in computer architecture and how they have improved specific applications, like probabilistic inference. To this end, this work introduces the concept of novel computer architectures and why they have become relevant in the following Chapter 2. Based on this concept, the subsequent Chapter 3 discusses different levels of granularity or scopes at which

computer architectures can be optimized or tailored towards specific workloads.

Chapter 4 gives an overview of how novel computer architectures can be used to offload and accelerate an application from the field of AI. Specifically, we introduce Sum-Product Networks (SPNs), a modern model from the class of Probabilistic Graphical Models (PGMs), and how corresponding inference tasks can be offloaded and accelerated using novel architectures. A more detailed look at this topic can be found in Part ii.

In Chapter 5, we shift the focus from AI towards Big Data and take a more in-depth look at the new architectural paradigm of NDP. In addition, the chapter serves as an overview of the publications included in Part iii that discuss using smart storage devices for offloading and accelerating database-related workloads.

While Chapter 4 and Chapter 5 focus on the application of novel architectures to improve AI or Big Data applications, the focus of Chapter 6 inverts this and instead shows that AI can also be applied to solve issues in novel architectures. Specifically, we discuss using SPNs for Cardinality Estimation, which could help improve potential applications of smart storage devices.

Finally, Chapter 7 will conclude this thesis. It summarizes the contributions and an outlook on the potential future of novel architectures, AI and Big Data.

2 ARTIFICIAL INTELLIGENCE AND NOVEL ARCHITECTURES

Two of the most formative minds in computer architecture are John L. Hennessy and David A. Patterson. Both were awarded the 2017 Turing Award for their work in computer architecture. In their Turing Award Lecture "A New Golden Age for Computer Architecture" [22], they give a detailed history of how computer architecture has advanced over the years. For completeness, the following chapter will reiterate the advances described in [22]. From this baseline, we discuss how their predictions became true in the field of AI, focusing on new architectural paradigms and Domain-specific Architectures (DSAs).

2.1 MOORE'S LAW, DENNARD SCALING AND THE END OF THE LINE

Moore's Law is among the most well-known laws of computer architecture. Named after Gordon Moore's observation in 1965 that the number of components in an Integrated Circuit (IC) are doubling every year [37]. Based on that observation, Moore's Law was derived. About ten years later, in 1975, it was updated since the increase in components had slowed down. The newer version predicted a doubling of components every second year. While Moore's Law did persist for many years, it started to slow down around 2004.

In conjunction with Moore's Law, a less well-known prediction was made by Robert H. Dennard [7]. The so-called Dennard Scaling stated that power density would stay constant with smaller transistors. The main observation of Dennard was that voltage and current are proportional to the linear dimensions of a transistor. At the same time, the power of an IC is proportional to the capacitance. Since capacitance depends on the area of an IC instead of the linear dimensions, a reduction in transistor area allows a corresponding increase in clock frequency without impact on the power density.

Through the conjunction of Moore's Law and Dennard Scaling, computer chips could increase their performance simply by shrinking down transistors and packing more of them into the same area. Unfortunately, Dennard Scaling abstracts two essential details, which only became relevant with the increasingly tiny transistors of recent years: Leakage current and threshold voltage. While the dynamic switching power of a transistor or IC adheres to Dennard Scaling, leakage current and threshold voltage do not scale with the area or linear dimensions of the transistors. In contrast, leakage currents increase when transistor

size is decreased due to the proportional reduction in insulation. The resulting problem is called Power Wall and limits the clock frequency of modern CPUs to roughly 4 GHz.

Due to the end of Dennard Scaling and Moore's Law around 2004, developments in computer architecture moved more towards adding additional cores. Multi-core systems are particularly interesting in consumer-facing electronics like regular workstations or smartphones since they allow for the concurrent execution of different programs or tasks. Given a higher number of cores, execution of background tasks can occur concurrently without context switching, which improves system responsiveness and eliminates corresponding lags and freezes. While this can also be achieved by using fast context switches, using multiple slower cores can achieve the same effect using less power.

While adding cores is helpful in systems with lots of multi-tasking, it is also limited in its scalability. The main problem can be attributed to situations where the performance of a single task is the determining factor. While many tasks can be parallelized to a certain degree, almost all have at least a small portion that has to be executed sequentially, even if it is only synchronization or communication between concurrent subtasks. Unfortunately, the sequential portion of a program will hence always limit the speedup that can be achieved using multiple cores. This limitation was first described by Gene Amdahl and is hence called Amdahl's Law [3]. The corresponding correlation between speedup, sequential code, and the number of cores is illustrated in Fig. 2.1.

To further elaborate on this correlation, let us consider the following Eq. (2.1).

$$S = \frac{1}{r_s + \frac{r_p}{n}} \tag{2.1}$$

The speedup S is determined by the number of cores n used to execute the parallel part r_p and the sequential part $r_s = 1 - r_p$, executed by a single core. We can see that the sequential part dominates the overall maximum speedup, as we cannot reduce the corresponding execution time by adding cores. Assuming that the parallel part is infinitely parallelizable, reducing its runtime to almost nothing, $\frac{r_p}{n}$ would equal 0. Thus the upper bound of the speedup can be described by Eq. (2.2).

$$S_{max} = \frac{1}{r_s} \tag{2.2}$$

The critical issue hidden in Amdahl's Law is that no matter how many cores we package into a single CPU, performance improvements will degrade, as sequential portions will limit overall performance. Thus, adding more cores will yield diminishing returns sooner or later.

According to [22], this point was reached in 2015, and we have reached the so-called *End of the Line*. While the current outlook for

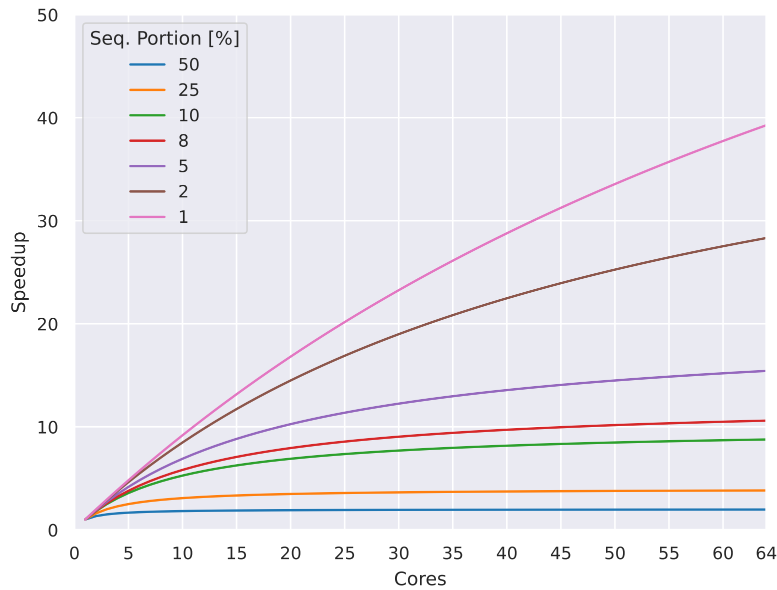


Figure 2.1: Maximum possible speedup as dictated by Amdahl's Law, depending on the number of processor cores and the sequential portion of the code. Replotted based on [22].

computer architecture seems dire, the *End of the Line* only applies to general purpose ICs, such as CPUs and GPUs.

2.2 DOMAIN-SPECIFIC ARCHITECTURES

Hennessy and Patterson end their Turing Award Lecture on the topic of Domain-specific Architectures (DSAs). While the *End of the Line* will be a significant issue in further increasing the performance of general-purpose processors, there is still massive potential for novel architectures. Specifically, DSAs offer huge optimization potential since they can be tailored towards a single application instead of many. One example described by [22] is the use of caches. While caches are generally crucial for performance in general-purpose processors, they have two disadvantages. First, if the computation requires a huge dataset, caches do not work, as they are continuously overwritten. Additionally, if caches do work, this is primarily due to locality, which means that most of the cache is idle. While caches most definitely have a significant impact on general-purpose applications, there are domains where caches are more or less useless.

Considering AI as an example domain, caches do not significantly impact the performance since the datasets are typically way beyond the capacity of the caches. This is also prevalent in Google's TPU, also used as an example in [22]. The original TPU v1 came without any caches and is one of the first examples of a DSA, targeted explicitly

at tensor processing in the context of ML. Presented by Google in 2016, the TPU v1 is a PCIe-based accelerator card tailored towards the acceleration of Neural Network (NN) inference. To this end, it comes with a Matrix Multiplication Unit (MMU) capable of executing a 256×256 matrix multiplication of 8-bit integers.

While Google's TPU v1 is an exciting piece of hardware and an excellent example for the predicted *Golden Age of Computer Architecture*, it is basically just a harbinger of what was about to come. While there is probably no better source for the history of computer architecture than Hennessy and Patterson, the following sections will now discuss advances in computer architecture that happened after their outstanding Turing Award Lecture.

2.3 ARTIFICIAL INTELLIGENCE ARCHITECTURES

The prior section already introduced the Google TPU v1, which was among the first DSAs in the field of AI. The main goal of the TPU was accelerating NN inference. More specifically, the goal was the offloading and acceleration of the TensorFlow library [34]. Now and then (2016), TensorFlow is among the most important libraries in ML. One advantage of TensorFlow is its support for Nvidia GPUs, which helped improve training times by exploiting the available parallelism. While GPUs already offer tremendous training performance, specifically tailored hardware like the TPU allow for even better performance. Additionally, tailored hardware is often more energy efficient as well.

Since 2016, Google has developed three improvements over the original TPU v1. The most recent v4 was released in 2021. Interestingly enough, the improvements to the TPU are similar to how general-purpose hardware evolved. For example, on-chip memory has increased from 28 to 144 MiB, and clock frequency has gone from 700 to 1050 MHz. While these improvements are valuable for overall performance, they are not that interesting from an architectural perspective. Simply put, these improvements can be attributed to using a more modern technology node (7 nm in the v4 vs. 28 nm in the v1).

In contrast, one significant improvement is the switch from DDR3 memory to High-Bandwidth Memory (HBM), which AMD, Samsung, and SK Hynix originally developed. HBM stacks a number of DRAM dies. While typical DRAM implementations use relatively narrow interfaces (64 or 128 bit for CPUs and up to 512 bit for GPUs), HBM natively uses 1024 bit per stack. In some GPUs, four corresponding stacks (of four DRAM-dies each) are used, which yields a 4096 bit memory interface. Due to this, throughput in the TPU v4 is 1200 GBps, whereas the TPU v1 only achieved 34 GBps. This represents a 35x increase in memory throughput, achieved by implementing a significant architectural improvement.

Switching from DRAM to HBM is a relatively simple architectural improvement that can be used to increase performance. We will discuss a similar case in more detail in Chapter 11. At a higher level of granularity, the use of a TPU could also be considered an architectural extension, as less optimized processing units such as CPUs and GPUs are replaced by a new processing unit more specific to the relevant application. Generally speaking, the resulting systems are still typical general-purpose systems, but they feature a co-processor that can be used for specific tasks. Systems that use multiple types of processors are typically called heterogeneous systems.

A step further from application-specific co-processors or accelerator cards are complete systems designed around specific applications. Examples of this can be found in the more data-flow-oriented systems developed by Graphcore, Cerebras, and SambaNova. The overall system is entirely targeted at a specific application in those cases. Considering the Graphcore Intelligence Processing Unit (IPU) as an example: The IPU comes in so-called pods, each featuring a high-performance general-purpose compute server, as well as several IPU machines. The resulting overall system is typically a complete server rack instead of a single workstation or server. Each IPU machine features four IPU processors, which are made up of 1472 independent cores. Overall, an IPU machine delivers 1.4 PFLOPS, and pods typically contain up to 64 IPU machines.

In comparison, these new data-flow architectures represent a much more significant architectural change than the development of the TPU. Instead of adapting an existing architecture, it is rebuilt from the ground up, allowing for much more optimization toward a particular goal. While all of the aforementioned architectures are relatively new, most of them have already been used in academic or industrial applications. For example, the survey by Emani et al. [14] compares these architectures with the Nvidia A100 GPUs in the context of ML and more specifically on LLMs. While the paper does not offer a clear winner, it does offer some interesting insights on the device selection. Additionally, it also highlights that porting effort is required to fully utilize the capabilities of novel architectures. More recently, there are also a number of publications, such as [54] and [13], that investigate the use of novel dataflow architectures in other scientific applications, including SARS-CoV-2 simulations.

3 ARCHITECTURAL OPTIMIZATIONS AND ADAPTATIONS

In the prior Chapter 2, we have given an overview of different novel architectures that have become relevant within the field of AI. This chapter introduces different scopes at which architectural optimizations and adaptions can be applied. To this end, we define a baseline architecture in Section 3.1. Then, using this as a baseline, we will discuss incremental changes and extensions in Sections 3.2 and 3.3. The final Section 3.4 introduces even more advanced adaptions that go beyond the original baseline.

3.1 GENERAL-PURPOSE ARCHITECTURES

Computer architecture generally describes the structure of computers using their components. Due to the increasing complexity of computers, the term architecture can generally be used at very different scopes. For example, the Instruction Set Architecture (ISA) defines how an abstract processor can be programmed. It defines a memory model, the number of registers, their size, and similar system characteristics. A corresponding microarchitecture then describes a concrete implementation of a given abstract ISA. For the scope of this work, we use the term architecture at a much broader scope. Instead of looking at the components of a CPU, we look at the components of a typical computer *system*. Accordingly, the general-purpose baseline architecture is a simplified model, which is shown in Fig. 3.1. Note that the shown architecture already includes a GPU and a Network Interface Card (NIC), as both are commonly included in regular desktop or mobile computers.

Compared to the previously described ISAs, the abstraction level of this model is much higher, which allows us to focus on the relevant concepts without requiring extensive knowledge about the intricacies of different system buses or other low-level details. In addition, this model corresponds to the perspective of most users and a significant portion of developers since most programming models and languages are still centered around this perspective.

The baseline architecture is mainly defined by the CPU and its main memory. This perspective is suitable since it allows the model to capture very different systems without including superfluous low-level details. For example, the model can depict servers, workstations, and embedded systems. While the system bus will typically be PCIe in servers and workstations, corresponding on-chip buses can be used in

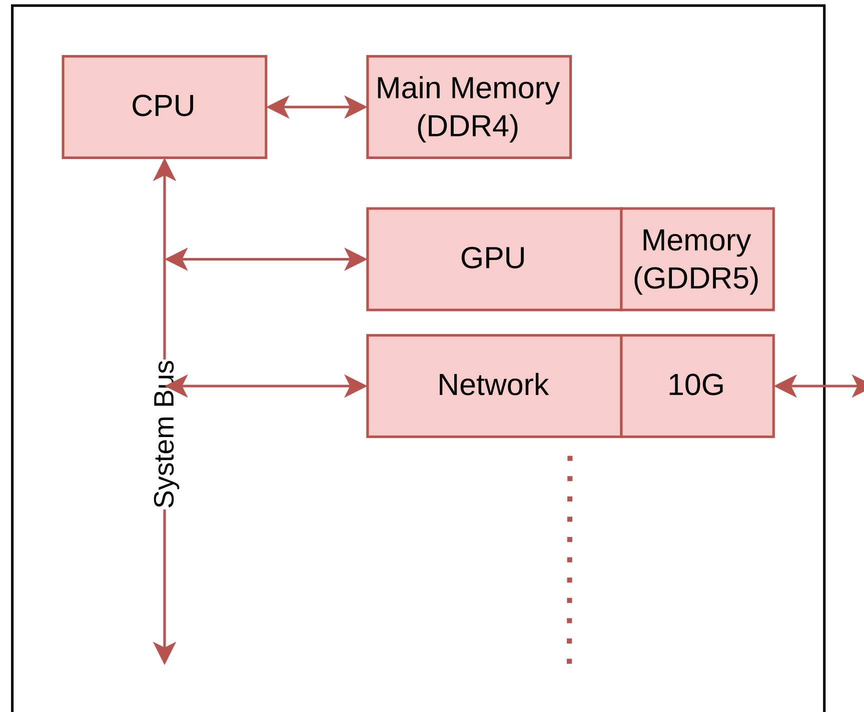


Figure 3.1: Simplified general-purpose Architecture of a typical contemporary computer.

embedded System-on-a-Chips (SoCs). While this certainly impacts the system performance, it does not impact the following considerations regarding the adaptation and optimization of the architecture towards particular use cases, such as AI.

3.2 INCREMENTAL IMPROVEMENTS

Incremental improvements are the first kind of architectural changes we can apply to a given system architecture. In general, incremental changes have been the driver for increasing performance in general-purpose systems for quite some time. For example, let us consider the PCIe standard. PCIe is used in many general-purpose computers to allow extension using accelerator cards. Initially released in 2003, PCIe Gen 1 can transfer 0.25 GBps per lane using up to 16 lanes. While this was in line with the required transfer bandwidths at the time, more recent workloads also require more bandwidth, which is why the more recent PCIe Gen 5 supports up to 15.059 GBps per lane. The evolving PCIe standards are one example of incremental changes to an existing architecture.

Generally speaking, incremental improvements are changes to the system that do not require programmers to rethink their programs. For example, moving from regular DRAM to HBM will typically not break existing hardware accelerators or software, as memory accesses

are still processed in a similar, if not the same, way. Furthermore, considering existing applications, incremental improvements are typically backward compatible so that existing applications do not have to be adapted. Depending on the scope of the change, adaptations to software might help exploit the novel capabilities. However, they are often not required since compilers or interpreters and the Operating System (OS) typically hide these low-level architectural details.

The simplicity and backward compatibility of minor incremental improvements are the main reasons general-purpose architectures evolve step-by-step using many different incremental changes. In Domain-specific Architectures (DSAs), the impact of incremental changes is typically predictable. For example, considering the TPU, it was clear that in the original v1, memory bandwidth was an issue. Accordingly, the following v2 immediately switched to HBM, which increased memory bandwidth by almost 20x. The predictable nature of minor architectural changes makes them comparably cheap and low-risk. In addition, due to them being more manageable in their implementation overhead, they can be used reactively. Specifically, if the relevancy of a workload increases, an incremental improvement can be applied to adapt to the changed overall workload. Examples of incremental improvements are illustrated in Fig. 3.2 and we will discuss a number of matching publications as part of this cumulative dissertation in the following Chapters 8, 9 and 11. These chapters will specifically look at optimized digital arithmetic and the use of HBM in the context of ML inference acceleration.

3.3 ARCHITECTURAL EXTENSIONS AND OFFLOADING

While incremental improvements are a reasonable way of consistently evolving an existing architecture, they are unsuitable for tackling more significant shifts in the typical workloads. Extending the architecture to ensure that the desired performance goals are met can be a better approach in those cases. However, due to the more complex nature of developing architectural extensions, they are much more uncommon than incremental improvements. While they are less common, most general-purpose systems still feature at least one specific extension, specifically GPUs. In addition, several common extensions are typically included in the mainboards of general-purpose machines.

To understand the rise of a specific architectural extension, consider how GPUs became relevant: The typical general-purpose workload suddenly shifted when operating systems moved from pure textual IO to more visual user interfaces. As a result, graphics-related computations became much more common, especially since they have to be performed continuously at a relatively high frequency to enable fluid interaction with the system. With the corresponding increase in graphics- and geometry-related computations, it made sense to

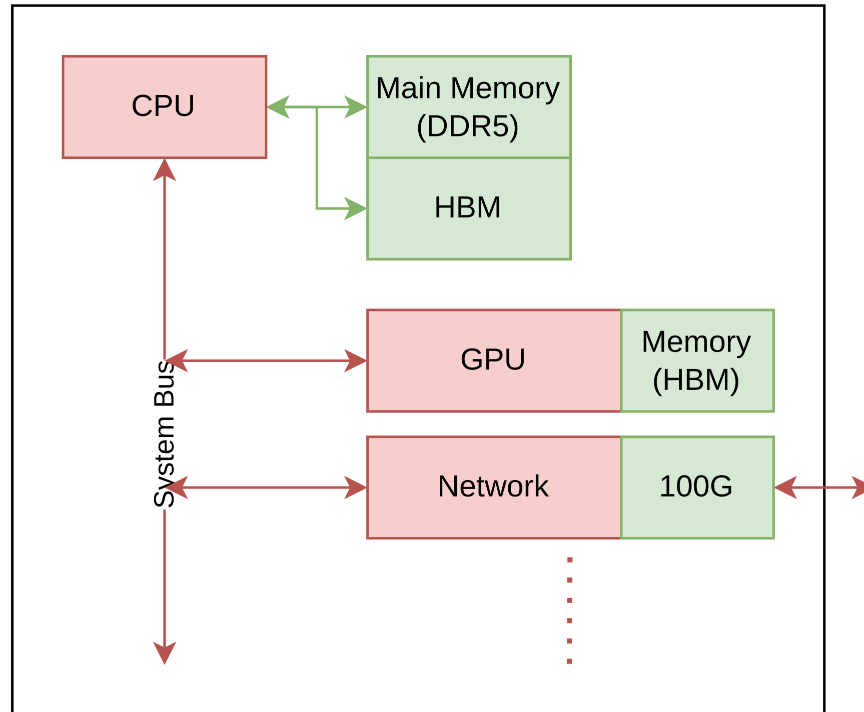


Figure 3.2: Examples for incremental improvements to the baseline system architecture.

introduce a specific co-processor to the system, which was optimized toward these workloads.

GPUs, as a common example of an extension to an existing architecture, are often integrated into the system using PCIe-based accelerator cards that can be plugged into the system's mainboard. Sometimes, the GPU is integrated into the same chip or die as the CPU. In embedded systems, this is often the case with System-on-a-Chips, which can also incorporate additional extensions. In desktops and workstations, corresponding chips are sold as CPU with integrated graphics (Intel) or as Accelerated Processing Units (AMD). In contrast, servers are commonly configured without a GPU, as they do not offer a regular graphical user interface. The GPU is such a common example that it is intentionally included in the baseline architecture from Section 3.1.

A more recent example of an architectural extension is Google's Tensor Processing Unit, also shown in Fig. 3.3. With the recent shift towards ML workloads (at least at Google), it became reasonable to develop a corresponding accelerator card that could be used to offload these workloads. Interestingly enough, ML workloads were already being offloaded to GPUs since their massively parallel architecture was well suited for NN training and inference. However, the development of the TPU allowed Google to optimize even more toward their specific use cases.

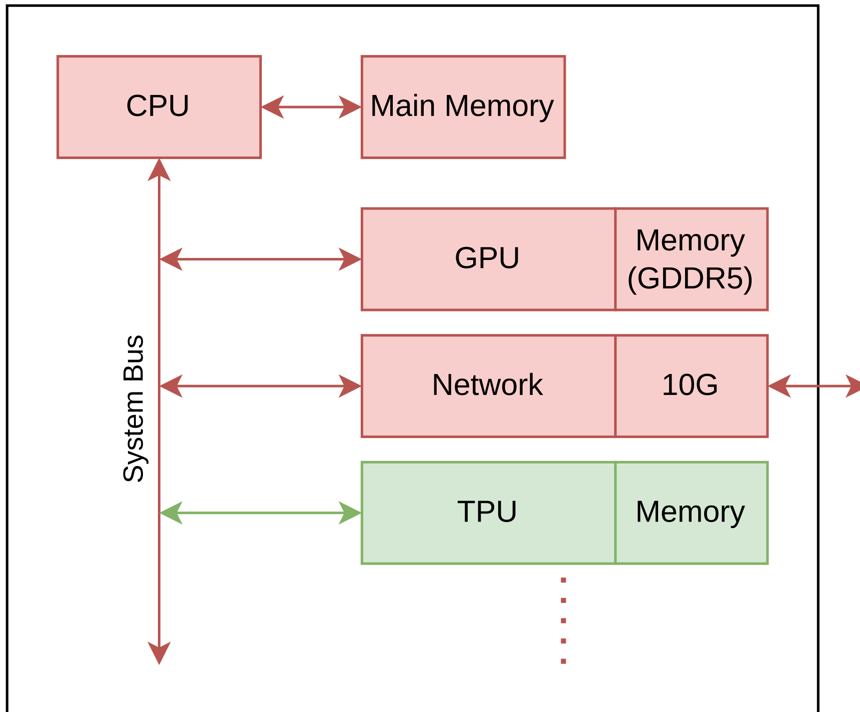


Figure 3.3: Baseline architecture with an additional Tensor Processing Unit-extension for ML workloads.

Designing and implementing a novel architectural extension takes time and effort. The number of different architectural extensions is quite limited. Apart from widespread use cases, such as graphics, audio, or networking, there are very few applications for which specific high-end extension cards exist. Interestingly enough, FPGA-based accelerator cards offer a compromise since they can be configured to represent arbitrary hardware designs. Accordingly, they can be configured to realize novel architectural extensions without the cost of developing a matching chip and board, which makes them extremely interesting in research. Looking specifically at FPGAs and NNs, many recent works are investigating the use of FPGAs for NN-acceleration, which is highlighted by Guo et al. in [19]. The work surveys more than 20 approaches from the last ten years and only includes NN-based implementations. Beyond that Parts ii and iv feature several FPGA-based approaches for an architectural extension of the acceleration of SPN inference.

3.4 NOVEL ARCHITECTURAL PARADIGMS

While the architectural optimizations discussed in the two prior sections already offer great potential to improve existing architectures, there is still an even more complex approach. Instead of adapting an existing architecture like the baseline architecture introduced in

Section 3.1, architectures can also be created from scratch. While developing these novel architectures is much more complex, it also offers much more potential for optimization. This potential is also one of the reasons why these novel architectures are typically oriented toward a particular paradigm or, instead, a corresponding paradigm shift. Since these architectures are tailored toward concrete goals and requirements, comparing or generalizing them is challenging. Thus, the following sections introduce three examples of novel architectural paradigms: Data-Flow, In-Network, and Near-Data Processing.

3.4.1 *Data-Flow Processing*

With the recent developments in AI, computations are often based on data flow instead of control flow. However, in typical programming languages, commands are executed in a sequence occasionally broken by control-flow statements, such as if-else commands or loops. Thus, Control-Flow Graphs (CFGs) describe programs using unbroken sequences of commands called basic blocks as nodes and control flow as the edges connecting them. This structure does not apply in Dataflow Processing (DFP) as computations are typically represented using graphs, or more specifically, Directed Acyclic Graph (DAG). Instead of describing control flow, the edges in those Data-Flow Graphs (DFGs) represent the data flow. While DFGs typically encode a single operation per node, DFP breaks this convention. Instead, nodes may contain more complex computations, including local control flow. Each node can be executed as soon as its predecessors have finished and the corresponding output data is available. In addition, approaches like Streaming and Pipelining can increase throughput by overlaying the execution of multiple instances.

Data-flow-based models have become increasingly attractive, as recent models from the field of AI and ML can easily be captured by them. Examples of such models are Sum-Product Networks and many kinds of NNs, which are based on DAGs without any control flow. Since DFP-based models are well suited to represent complex and relevant ML problems, there has been a lot of research and development towards hardware that can execute corresponding compute graphs. In this section, we want to introduce two different architectures based on this novel paradigm. Graphcore's IPU architecture, as well as the AMD Xilinx Artificial Intelligence Engines (AIEs), were initially targeted at data centers to accelerate matching workloads in the cloud. Interestingly, through the acquisition of Xilinx by AMD, AIEs are now also being marketed towards desktop and mobile markets.

Both architectures are based on a similar programming model: Instead of writing a single program representing the overall problem, the developers write several smaller programs (called codelets) defined by their inputs and outputs and a function containing the actual

computation. Using these codelets, the programmers then define a graph of computations by defining the data flow between them. The resulting compute graph is then compiled for the specific architectures, both based on two-dimensional arrays of compute cores. Additionally, the cores are interconnected to allow some form of data exchange between cores. Accordingly, the data flow defined by the compute graph is mapped to corresponding data movement, whereas each codelet is compiled to machine code for one of the cores. The compiler also infers how to perform the required data movements defined by the compute graph's edges.

Considering the hundreds or even thousands of cores of these architectures, they are examples of truly massively parallel Multiple Instruction, Multiple Data (MIMD) architectures, which are especially interesting in contrast to GPUs. While they are also massively parallel, they instead use a Single Instruction, Multiple Data (SIMD) or Single Instruction, Multiple Threads (SIMT) approach, where multiple cores are grouped, executing one stream of instructions on different streams of data, which is a lot less flexible in comparison to the MIMD offered by DFP architectures.

3.4.2 *In-Network Processing*

A different approach also targeted at data centers is In-Network Processing (INP). The main goal of INP is moving workloads away from local machines towards compute resources available on the local or global network, which is primarily achieved using Smart Network Interface Cards (NICs). Like regular NICs, the smart versions are PCIe-based extension cards with high-speed network interfaces. In some cases, INP is also realized using smart switches, which unify the functionality of regular switches with additional compute capabilities. Generally speaking, most commercial INP approaches focus primarily on networking-based applications or algorithms, such as packet inspection. While this is reasonable from a commercial perspective, INP offers much more potential when considering new workloads, such as ML inference. Especially interesting are corresponding FPGA-based data-center accelerator cards, which unify 100G networking interfaces with the compute capabilities of modern FPGA.

One significant advantage of those devices is their sheer amount of bandwidth. For example, some off-the-shelf devices offer up to four external 100G interfaces in addition to internal interfaces like PCIe. This bandwidth is especially interesting since FPGA-based accelerators are frequently optimized towards throughput using hardware pipelining. The functionality of those accelerators can then be exposed to the local network using corresponding networking stacks that enable clients to send their workloads to the network-attached FPGAs. Furthermore, since the workload can reach the FPGA via its dedicated network

interfaces, no additional intervention from the server hosting these cards is required, which additionally decreases latency. Accordingly, network-based offloading using FPGAs has been of increasing interest as it democratizes the provided functionality. A conceptual setup is shown in Fig. 3.4.

Another major advantage of INP is the shared nature of the functionality. Instead of fitting many different workstations with FPGA-based accelerator cards to perform an offloadable workload, it can be hosted in a single machine, which acts as a server, decreasing cost since fewer accelerator cards are required. Additionally, due to the reconfigurable nature of FPGAs, they can be reused for different workloads. Corresponding setups often make sense when the offloaded workloads increase without reaching the critical mass to warrant specific accelerators. This is also the case for accelerators that are not FPGA-based. Since they are not limited to a single system, the workloads from multiple clients can be pooled together to increase utilization. One such example of INP will be discussed in the included publication in Chapter 10.

3.4.3 *Near-Data Processing*

The last novel paradigm we introduce is Near-Data Processing (NDP). The focus of NDP is mostly on smart storage devices that can be used to offload typical Big Data workloads. To understand the objective of NDP, we must first consider general-purpose databases or key-value stores. They are typically used to store big datasets and enable quick and easy access to the hosted data. Additionally, a big concern in such systems is scalability. While small databases and stores might be hosted on regular machines, production databases of companies will typically reside in specific machines that offer multiple terabytes of working memory in addition to tens or even hundreds of terabytes of persistent memory, such as flash-based SSDs or HDDs. Due to the vast working memory, corresponding databases can typically hold most of the relevant data in their working memory, so access is generally fast. Problems typically only arise if more extensive parts of the less frequently accessed data are inspected since this will require vast data transfers to the host CPU. In addition to the bandwidth required for the data transfers, the CPU typically has to loop over many or even all dataset entries to locate the relevant ones. Considering the increasing size of modern databases and datasets, this can lead to situations where the system responsiveness can suffer.

Considering the typical database server, they will likely employ regular SSDs, which use flash memory to store the data. Interestingly, the bottleneck in modern SSDs is often not the actual flash memory's bandwidth or latency but the underlying system bus (PCIe/NVMe). In smart storage devices, scanning and filtering operations could be

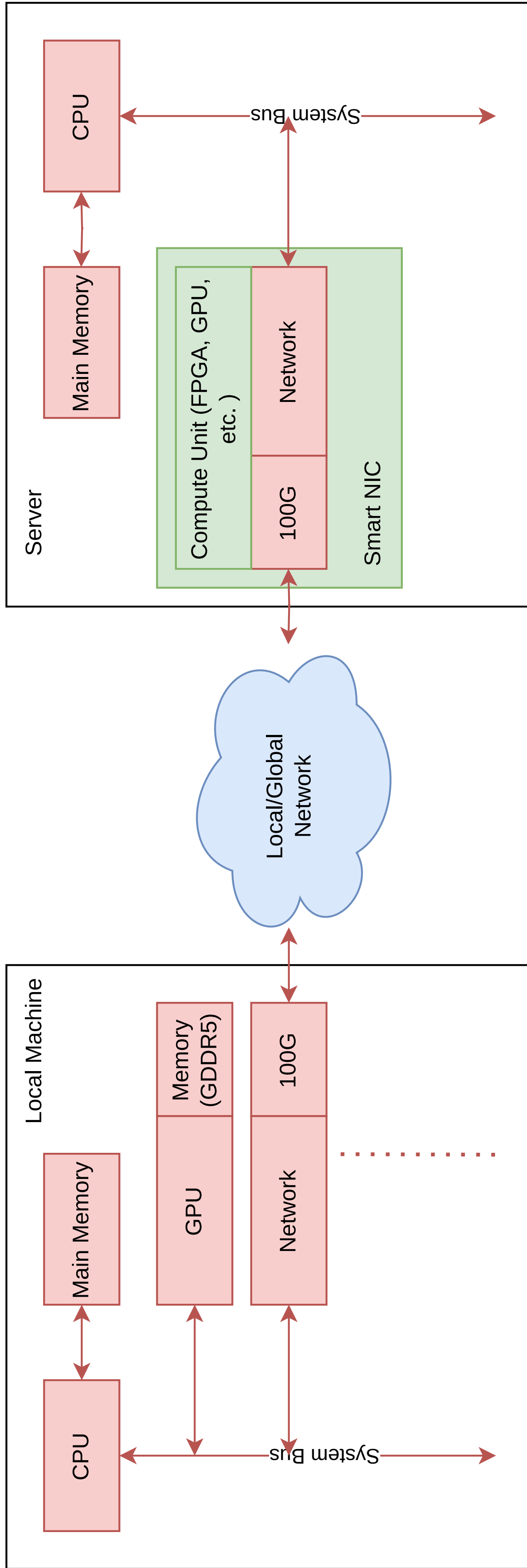


Figure 3.4: Baseline architecture that is connected to a network with access to INP compute capabilities.

performed before the data is transferred via system buses, which in turn has the potential to reduce the required bandwidth. To this end, smart storage devices combine persistent flash memory with additional computational capabilities that can be used to perform some computations on the data before it is transferred to the host CPU. Instead of the typical *data-to-code* approach, this *code-to-data* approach could alleviate bandwidth-related bottlenecks. Additionally, due to the offloading of corresponding workloads, overall system responsiveness would not necessarily be reduced since the computation can now be performed asynchronously by the storage device.

The lack of suitable devices is a central issue in investigating NDP. At the time of writing, only four devices could generally be defined as smart storage devices that unify reasonable capacities of persistent flash storage with the required computational capabilities. Additionally, all of those devices come with certain caveats. For example, the Fidus Sidewinder is an FPGA-based accelerator card, which can hold up to two NVMe-based SSDs. While this could be considered smart storage, access to the flash storage must be propagated through the flash controllers on the SSDs. This indirect access is problematic, as the controllers typically rely on intermediate abstractions like the Flash Transaction Layer (FTL), which takes care of wear-leveling and similar flash-related issues. While this makes sense in consumer-facing SSDs, it also reduces the available bandwidth, thus limiting the potential of NDP.

Similarly, the Samsung SmartSSD unifies a Samsung SSD with an FPGA-based accelerator card in a single case. Within the case, an additional switch connects both devices to the host and each other. While this does enable high-speed peer-to-peer communication between the SSD and the FPGA, it does not actually unify the functionality, as the accelerator card only comes with 4 GB of on-board DRAM, which is most likely not enough to perform significant operations on terabytes of data. Additionally, due to the peer-to-peer nature of the interconnection, bandwidth is also shared when communication is supposed to include storage, compute, and host. The remaining two devices are the COSMOS+ OpenSSD and the prodesign HAWK/FALCON. Both devices unify persistent flash memory with FPGA-based compute capabilities. Due to the flash controllers being realized using the FPGA, all requested data travels through the FPGA, which enables *bump-in-the-wire* NDP, where data is automatically pre-processed while it is transported to the host. Unfortunately, corresponding approaches are still theoretical. Additionally, the COMOS+ relies on relatively old hardware, specifically a Xilinx Zynq-7000, which only offers limited resources for implementing flash controllers and additional accelerators. While the prodesign HAWK might solve this problem with its state-of-the-art Xilinx Versal FPGA, it is unfortunately not yet available with the required flash extension. The included publications in

Chapters 12 to 14 discuss NDP using the COSMOS+ in more depth. Specifically, these chapters discuss the offloading of key-value store operations using custom and generated FPGA-based accelerators.

4 ACCELERATING SUM-PRODUCT NETWORK INFERENCE

The previous Chapter 2 already showed the relevance of novel architectures to AI. In this chapter, we are going to introduce SPNs as one potential AI workload that profits from the use of novel architectures. To this end, the following sections will first discuss SPNs in general and then describe and discuss several approaches of using modern architectures to accelerate SPN inference.

4.1 INTRODUCTION TO SUM-PRODUCT NETWORKS

NNs and Deep Learning (DL) have been responsible for most of the recent success in AI and ML. However, due to the sustained advances in ML, the research focus has broadened, and other models are also investigated more. One class of models also receiving increased interest are PGMs. The general idea behind PGMs is using graphs to represent relations between different probabilistic variables. Bayesian Networks and Markov Random Fields are generally the most well-known examples for PGMs. A more recent example are SPNs, proposed by Poon et al. [42]. While the original publication only introduces the general idea, a later version includes initial training approaches for SPNs [43].

Exact inference is one significant advantage of SPNs over other DL techniques. In this context, exact inference refers to the fact that SPN inference computes a probability using the given partial or complete evidence, which is one of the main features that differentiates PGMs from DL approaches. The resulting probability is comparable, inherently enabling uncertainty quantification. For example, we can implement classification using SPNs by querying the probability of a given sample belonging to a specific class. This approach will yield different probabilities for each class, and comparing them will allow us to classify the sample. In this situation, the comparison between the probabilities tells us whether a decision was close or not. Additionally, if a sample significantly differs from the samples of the training data, the probability will be extremely low, indicating the corresponding uncertainty. In contrast, NNs are typically more of a black box and output only the resulting classification, which does not indicate how certain that decision has been. Lastly, the inference is not only exact but also tractable w.r.t to its size, as the nodes can simply be evaluated bottom-up yielding a result. For other PGMs, evaluation requires multiple passes, as the nodes and edges encode more complex probabilistic relationships.

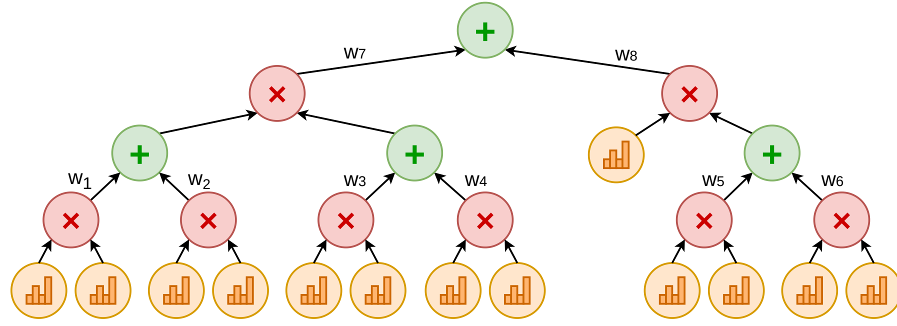


Figure 4.1: Example SPN, taken from Chapter 10

More formally, SPNs capture joint probabilities over a set of random variables using a rooted DAG. An example is depicted in Fig. 4.1. Note that the graph is made up of three distinct types of nodes:

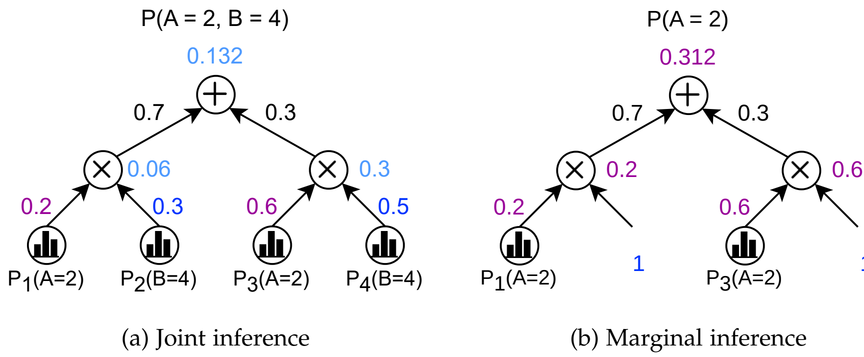
- The terminal leaf nodes are univariate distributions over a single random variable. Different distributions, such as Gaussian or Poisson distributions [35], can be used. Additionally, histograms can be used to approximate more complex distributions or as a replacement in discrete cases [36].
- Product nodes are used to represent factorizations over independent distributions. For *consistent* SPNs, it is additionally required that all children of a product node depend on either the non-negated or the negated random variable, but not both at the same time.
- Weighted Sum nodes represent mixture models over distributions. For *complete* SPNs, the child nodes of weighted sums must use the same set of random variables.

An SPN is considered to be valid if and only if it is *complete* and *consistent*.

4.1.1 *Inference*

Due to the structure of SPNs, the inference is a simple bottom-up pass. Using the given evidence, the leaf nodes are evaluated and their results are forwarded performing the operations indicated by the nodes. The root of the SPN will then yield the inference result. In addition, SPNs allow different kinds of queries. Specifically, we want to introduce joint and marginal queries relevant to this work.

A query is considered a joint query if it uses complete evidence. For example, in a valid SPN, this implies that the value of each random variable is known. Thus the joint query can be answered by evaluating the terminal leaf nodes using the random values. The results of the terminal nodes are then propagated upwards while corresponding multiplications and additions are evaluated. Finally, after evaluating



(a) Joint inference

(b) Marginal inference

Figure 4.2: Inference example in an SPN, representing the joint probability distribution $P(A, B)$. In *joint* inference, all histograms output a corresponding value (a), while in *marginal* inference some histograms are marginalized and always output the value 1.0 (b). Taken from Chapter 15.

the root node, the resulting probability is the result of the joint query, as shown in Fig. 4.2 (a).

In general, marginal queries can be evaluated in the same manner. The terminal leaf nodes of marginalized random variables are set to the probability value 1. The remaining bottom-up evaluation of the DAG stays the same. The process is also shown in Fig. 4.2 (b).

4.1.2 *Learning and Training*

Due to their structure, building SPNs by hand is relatively simple, which is especially interesting, if they are used to model less complex probability distributions. However, while this approach is interesting, it does not scale. Accordingly, different approaches aim at the automatic learning and training of SPNs. In general, these approaches fall into two categories. First, some approaches assume that the general structure of the SPN is already known. Instead of creating a new SPN from scratch, the weights of the weighted sums are adjusted to learn an underlying probability distribution. Such approaches are generally called Weight Learning. In contrast, Structure Learning tries to generate the overall structure, including weights, from scratch.

The general approach in weight learning is typically based on the Expectation Maximization (EM) algorithm, a cornerstone of statistical ML. In the original work by Poon et al. [42], they generate a dense structure and then use EM to adjust the weights. Depending on the initial structure and the underlying dataset, the resulting SPN may contain branches with weights very close to 0. Then, these branches can be pruned to reduce the size of the SPN. The corresponding algorithms are discussed in [15] and [43]. Additionally, later approaches extend on the idea and enable learning of arbitrary leaf node distributions [9]. An even more exciting approach are Random Tensorized Sum-Product

Networks (RAT-SPNs) [40], which use randomly generated structures and Weight Learning for training. The most significant advantage of this approach is its simplicity. Interestingly, the resulting SPNs can compete with NNs[40].

In contrast to Weight Learning, Structure Learning is more complex. Initial approaches like [8] and [16] use clustering to create the structure of the SPN. These approaches exploit *completeness* and *consistency* to derive nodes recursively by trying to find independent subsets of random variables. If that is possible, corresponding product nodes are generated. If not, weighted sums are generated. Finally, the approach is applied recursively until the subsets contain only a single random variable. Concurrently, the dataset is clustered and divided depending on the independence of the random variables.

4.2 ACCELERATED INFERENCE

In the following Sections 4.2.1 to 4.2.3, we are going to discuss several different approaches to accelerate SPN inference. First, we are going to discuss more traditional acceleration approaches, such as CPU- and GPU-based acceleration in Section 4.2.1. Then, the following Sections 4.2.2 and 4.2.3 will introduce different FPGA-based acceleration approaches using four recent publications that are also part of this cumulative dissertation (cf. Chapters 8 to 11). We will also compare existing prior and related work. The corresponding sections will also discuss how different approaches exploit traditional and novel architectures and how they fit into the hierarchy of architectural improvements presented in Chapter 3.

4.2.1 *CPU- and GPU-based Acceleration*

While SPNs research has gained much traction in the ML community, corresponding research focuses mainly on general training and inference instead of acceleration. While the two primary SPN libraries, LibSPN and SPFlow, generally support acceleration, it is typically achieved using approaches based on TensorFlow, a well-known framework in the ML space. Its aim is mainly to enable acceleration of ML tasks, like inference and training using GPUs. Originally, TensorFlow was developed by Google to accelerate inference and training of NNs. While it does offer a relatively simple way of interacting with GPUs from Python, its bias towards NNs is also one of its weaknesses. Due to their structure, SPN inference is not as easily transformed into tensor-based operations. Thus, acceleration using the library-provided TensorFlow support is only a minor improvement.

In contrast, the works by Sommer et al. [49–51] offer a more promising approach. In [51], they first introduced the SPN Compiler (SPNC), which builds on the popular compiler frameworks Low-Level Virtual

Machine (LLVM) and Multi-Level Intermediate Representation (MLIR) to implement a tool flow that enables the compilation of SPNs to CPUs and GPUs. The general way to use SPNC is via a provided Python interface that wraps the whole SPNC tool flow and integrates well with the SPFlow library. Then, using serialization and protocol buffers, the SPNs can be transferred from SPFlow to SPNC, translating the models into its target-agnostic HiSPN dialect. Dialects are MLIR's way of defining domain-specific intermediate representations. The main goal of the HiSPN dialect is to represent a given model. Therefore, as part of the compilation flow, the model is lowered into the LoSPN dialect, representing the corresponding model more functionally. Using the LoSPN representation of the SPN, it is then possible to compile for CPU and GPU. In both cases, platform-specific optimizations, such as vectorization using AVX and AVX2, are applied to ensure that the capabilities of the target architecture are used.

In their initial publication [51], they compare their compilation-based approach against a Python baseline, achieving a speedup of 814.8x using an AMD Ryzen 9 3900XT with 32 GB RAM. For their evaluation, they use a benchmark set from [38], where SPNs are used to identify speakers in different clean and noisy speech samples. Note that the benchmarks are performed throughput-oriented: The model is transformed or compiled using either TensorFlow or SPNC once. Afterward, the corresponding inferences are performed as a batch. Execution time is measured, including compilation time. While it is evident that the Python baseline is not competitive against a compilation-based approach, it is still interesting since it could be assumed that compilation time may be an issue. In their work, they show that TensorFlow-based acceleration only yields speedups of 1.5x and 1.4x using a Ryzen 9 3900XT and an Nvidia RTX2070 Super, respectively. In contrast, the compilation-based approach achieves the speedup mentioned above of 814.8x using AVX2 vectorization. Interestingly enough, the GPU cannot compete with a speedup of only 524.7x. This is because of the additional data transfer overhead required to get the input and output data to and from the GPU.

The original tool flow was further evaluated in a range of different CPUs in a subsequent publication [49]. Specifically, the evaluation includes an embedded device to show that the approach applies to cheaper and smaller devices in addition to the high-end devices from the earlier work. Moreover, the functionality is extended to support vectorization using AVX-512. While prior work relied on the Libmvec library for AVX and AVX2 support, Intel SVML support is used to allow the mapping of elementary functions to AVX-512. Exploiting AVX-512 in an Intel Xeon Platinum 9242 CPU with 384 GB RAM further increases the speedup over the Python baseline to 976x.

The most recent paper [50] on the SPNC elaborates more on the underlying compilation flow using MLIR. In addition, a novel graph

partitioning approach is presented, which aims to enable the compilation of even larger SPNs, which became relevant due to novel training approaches and using SPNs. In comparison, the speaker identification benchmark consists of 628 different SPNs, each able to identify a unique speaker; more recent work on RAT-SPNs [40] used only ten SPNs, which in turn were much larger. Compared to 2,500 operations per SPN in the original benchmarks, RAT-SPNs typically contain more than 300,000 operations. The vastly increased number of operations made compilation as a singular graph impractical. Using the novel graph partitioning approach makes compilation of very large RAT-SPNs tractable and compilation times are similar to the TensorFlow-based approach introduced in [40]. Due to the tensorized nature of RAT-SPNs, TensorFlow-based performance is much better than for generic SPNs, especially on GPUs. Comparing on a GPU, TensorFlow achieves a speedup of 3x over SPNC, while SPNC achieves a speedup of 3.8x on a CPU. Interestingly, SPNC performance on a CPU is on par with TensorFlow-based execution on a GPU, which is especially interesting considering the cost of high-performance GPUs.

4.2.2 *FPGA-based Acceleration*

The general idea of FPGA-based acceleration of SPN inference was initially presented by Sommer et al. [52]. Using textual representations of SPNs, a Scala-based compilation flow reads in and pre-processes the model. Most pre-processing focuses on making the model as regular as possible. For example, product nodes with more than two children are split up to ensure that all arithmetic operations are binary. In addition, analysis passes determine the number of random variables used in the SPN, as that information is required to generate a hardware design. After the transformations and analyses, the tool flow then generates a fully spatial, fully pipelined Chisel3 module, where each operation in the DAG of the model directly corresponds to a hardware operator. The hardware operators were generated using the FloPoCo [1] library in the original implementation [11]. FloPoCo provides a floating-point encoding and matching arithmetic operators. While FloPoCo generally allows custom configuration of the number of exponent and mantissa bits for the encoding, the authors use a typical 64 bit encoding. The resulting datapath was the centerpiece of an accelerator, which also provided corresponding control and data interfaces. The authors synthesize bitstreams for the Xilinx VC709 evaluation board using a set of typical benchmarks from several datasets. The bitstreams are evaluated for their end-to-end inference throughput compared to a Source-to-Source compiler that transforms SPN models into TensorFlow- or C-code.

1 https://www.flopoco.org/

Figure 4.3: Floating Point binary format, simplified from Chapter 9.

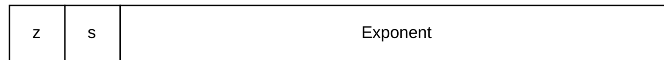


Figure 4.4: Logarithmic Number Scale binary format.

This approach was not nearly as sophisticated as the compilation flow discussed in Section 4.2.1. However, it was still a much more reasonable baseline than the Python-based implementation used in SPFlow. For all sixteen benchmarks, the FPGA-based implementation outperforms GPU by quite a considerable margin, achieving speedups of up to 100x. For ten of the sixteen benchmarks, the FPGA also outperforms a CPU by up to 6x. For smaller benchmarks, the CPU-implementation offers higher throughput since there are no additional overheads for data transfer. In those six cases, the CPUs is up to 13.5x faster.

Offloading SPN, inference like this is a typical example of an architectural extension. Assuming that SPN inference is a recurring task, it would make sense to try and offload it to more specialized hardware. Since developing novel accelerator cards is quite expensive, FPGA-based accelerator cards like the VC709 offer a more reasonable way of evaluating potential architectural extensions. In addition, due to their reconfigurability, they can also be used to test out different approaches. While the original work presents an architectural extension, the corresponding work was also incrementally improved in subsequent publications.

An optimized logarithmic number system was introduced in the publication included in Chapter 8 to replace the floating-point encoding. The main difference between logarithmic number systems and floating point numbers should become clear, when we consider Fig. 4.3 and Fig. 4.4, which depict how both encodings represent numbers in binary. Note that these specific formats are taken from Chapters 8 and 9 and already optimized towards SPN inference. Floating Point numbers are encoded using a fixed-point mantissa and a biased exponent, which is used to change the order of magnitude of the number (cf. Eq. (4.1)).

$$A = (-1)^s \times (1.0 + Mantissa) \times 2^{Exponent-Bias} \tag{4.1}$$

In contrast, logarithmic number systems only use an fixed-point exponent, as shown in Eq. (4.2). To encode 0, an additional bit (z in Fig. 4.4) is required.

$$B = 2^{(-1)^s \times Exponent} \tag{4.2}$$

Since log-likelihoods are commonly used in ML to represent very small probabilities better, it was reasonable to assume that a number system with logarithmic scaling would also achieve similar error margins while using less of the logic resources on the FPGA. Therefore, in addition to the changed encoding, the encoding was optimized towards the FPGA-based implementation, and the number of bits was fine-tuned towards each specific benchmark. Using this approach, the resulting accelerators could uphold a maximum relative error margin of 10^{-6} while saving up to 50% of Slices and DSP Slices on a VC709 FPGA accelerator card. The contributed logarithmic number system is one instance of an incremental improvement. While the prior work [52] was already able to achieve similar performance, switching to a new encoding significantly increased the resource efficiency.

Since the switch to a logarithmic number system had improved resource utilization significantly, the following publication (cf. Chapter 9) applied an even more sophisticated software simulation to fine-tune the encoding to given error margins. In addition, a novel custom floating-point format and a format based on Posit numbers were included in the framework. All three encodings were additionally optimized, and the Posit implementation was pipelined. Using the software simulation, all encodings were fine-tuned to match certain error margins. The Source-to-Source compiler used for the CPU- and GPU-baseline was also improved, and GPU compilation moved from TensorFlow to native CUDA. Similar to the prior work, FPGA-based acceleration was able to significantly outperform CPU and GPU for all but the smallest three benchmarks. While performance was not significantly increased over prior work, the corresponding publication offers insight into the suitability of different number systems: Specifically, the custom floating-point and the logarithmic format are reasonable choices, depending on the ratio of sums and products within a model. In cases with very few additions, the logarithmic number system is more resource- and energy-efficient. Interestingly, the Posit-based encoding could not keep up with the other two due to inefficient run-length encoding for an additional scaling factor, which is not present in the other two formats. While the Posit-based encoding was not as efficient, all three encodings were able to outperform CPU- and GPU-based execution for the bigger benchmarks, as shown in Fig. 4.5. Only for smaller benchmarks, the CPU achieved higher performance, because expensive data-transfers could be omitted.

The resulting tool flow offers increased flexibility, since the encoding can be fine-tuned and matched to the application. The evaluation of the work included in Chapter 9, it also became clear that the Posit number system is not optimal for probability-based applications on an FPGA. Additionally, we were able to find a deciding factor that determines, whether the logarithmic or the custom floating point number system were better suited. Specifically, the ratio between

adders and multipliers is important, since multiplications become much more simple in a logarithmic number system, while additions become more complex. If an SPN has a multiplier-to-adder ratio of about 8 : 1 or more, a logarithmic number system will be more resource and energy efficient.

The architecture's scalability was evaluated in a more recent publication in Chapter 11. Specifically, the architecture was adapted to use HBM, which had become more common on FPGA-based accelerator cards. Since the throughput of the accelerators had already reached line rate in prior work, the use of HBM was expected to allow further performance scaling. In addition, due to improvements in the architecture in a separate earlier publication (discussed in the following Section 4.2.3) and newer FPGAs with more hardware resources, the accelerators could now be replicated for parallel execution of multiple inferences. While the achieved performance gains were less than expected due to the limitations of the PCIe bus, the throughput was still increased by 1.5x over prior work. The switch from regular DRAM to HBM is also a fitting example of an incremental improvement to the architecture.

While the original work on offloading of SPN inference is an excellent example of an architectural extension, the three subsequent publications are all examples of incremental improvements. Exploiting different approaches like HBM or specialized data types can significantly impact an architecture's performance and efficiency. This was especially highlighted in Chapter 9, which also performs a power evaluation and optimizes multiple data types for the application. While this saves hardware resources on the FPGA, it also translates into a corresponding reduction in power consumption.

4.2.3 *In-Network Acceleration*

The prior sections have introduced different approaches to accelerate SPN inference. Apart from the CPU-based implementations, all are offloading-based approaches, where a PCIe-based accelerator card is used to offload a recurring workload. Additionally, some of the presented works also improved the general offloading approach by exploiting additional changes to the architecture, such as HBM.

We further investigate a novel architectural paradigm in Chapter 10. Instead of locally offloading SPN inference, this functionality is integrated into an INP stack. To this end, the architecture of the SPN accelerators was reworked to support streaming-based inference processing. Additionally, the accelerators were extended to enable the pass-through of additional meta-data. Using the meta-data and an adapted networking stack, the SPN inference can be performed from a remote machine by sending a packet or frame with the inference input to the accelerator using 100G networks. Furthermore, since the

Figure 4.5: Inference Throughput of FPGA-, CPU- and GPU-based implementations, taken from Chapter 9.

networking stack is flexible, inference can be realized using different networking protocols, depending on how the functionality is supposed to be exposed.

This INP approach is especially promising for end users. While AI and ML workloads are becoming increasingly common, it is still impractical for most users to have specific ML hardware in their private systems. Even workstations in academic or industrial institutions would typically not feature ML acceleration due to the high cost of high-end FPGAs and GPU. Moving the functionality to the network effectively democratizes it and enables many clients to request and use the inference without the need for costly hardware.

One downside of INP is the additional latency for network communication. While the increase in latency can be relatively small in compact, local setups, it will typically be much higher in remote or cloud setups, as the data has to be transported using the regular internet infrastructure. Thus, corresponding approaches are typically limited to applications that do not require extremely low latencies.

5 NEAR-DATA PROCESSING AND SMART STORAGE

While the prior chapter primarily focused on architectural extensions, this chapter looks at the novel architectural paradigm NDP. Specifically, we want to look at the nKV system, which investigates NDP using a RocksDB-based key-value store on the COSMOS+ OpenSSD to accelerate database operations, such as the GET and SCAN operations. In addition, the nKV system can perform more complex algorithms, like the Betweenness Centrality algorithm, using the accelerated GET and SCAN operation.

5.1 INTRODUCTION TO NKV

We have already introduced the general concept of NDP in Section 3.4.3. In this section, we are going to introduce nKV, which is a key-value store that relies on native storage to improve performance and to enable efficient NDP on the COSMOS+ OpenSSD. In contrast to traditional approaches, native storage exposes the actual physical flash memory to the key-value store. Instead of intermediate layers like the FTL, the key-value store can directly access the flash memory by its physical addresses. This approach's main advantage is removing intermediate layers that typically increase overheads. Additionally, nKV can exploit the underlying architecture of the flash memory, which is typically organized in multiple channels and Logical Units (LUNs). Finally, moving the control towards the key-value store allows the exploitation of the internal bandwidth of the flash memory, which is typically not possible if the operating system or intermediate layers control the flash memory.

In addition, the direct control of the flash memory also allows the key-value store to control where intermediate data is stored during the execution of requests, allowing nKV to manipulate intermediate data. This can be used to implement NDP functionality in the absence of more elaborate systems. One potential future avenue for research would be *bump-in-the-wire* NDP, which manipulates or transforms requested data without materializing intermediate results. While this is not yet possible in nKV, the advantages of native storage already allow for increased performance.

By allowing interaction with the requested data, nKV also enables NDP. Unfortunately, the ARM cores used in the Zynq-7000 chip on the COSMOS+ are already used to run the firmware. Due to this, the best option for offloading computational load like GET or SCAN operations was using FPGA-based hardware accelerators. We will go into more

depth on this topic in one of the following sections (Section 5.3) and the corresponding publications in Chapters 12 to 14. Section 5.2 will show how the concept of smart storage and NDP has evolved historically. Lastly, Section 5.4 will classify NDP within the hierarchy of architectural improvements.

5.2 PRIOR AND RELATED WORK

The following section will introduce related approaches in NDP. This section is loosely based on Section 12.7 of the previously published Chapter 12.

The concept of NDP was initially developed in the 1970s. Back then, the concept was known under the term *database machine*, which referred to architectures explicitly targeted at hosting databases. The term originated in [10], but there were other approaches, like Active Disks [1, 45] and IDISK [30], that were quite similar. However, a significant issue of these approaches was the proprietary nature, which made these machines expensive. As a result, corresponding approaches did not gain any significant relevance, which was often attributed to the limited IO bandwidths and the limitations in storage parallelism, which limited overall system performance.

Recent advances in storage technology, especially moving away from mechanical storage mediums, have significantly changed typical storage devices' characteristics. Most notably, mechanical mediums always suffered from limited parallelism, as a read-write head is required to move to the correct position on the tape or disk to read or write data. Specifically, the rise of flash memory alleviates this problem by using electric charges to store information.

Relying on flash memory, SSDs have become the typical persistent storage in most systems. HDDs are typically only used when cost-efficiency is problematic. Due to the much higher memory parallelism in typical SSDs, they have also become more attractive to researchers. Starting in 2013, the term Smart SSD was introduced and generally referred to devices that unified storage and compute capabilities in a single device. A matching prototype from Samsung is first discussed in [12], where an ARM core inside the SSD is used to perform queries from the TPC-H benchmark. A similar approach is presented in [29]. Both approaches are similar because they target offloading and acceleration of relational databases and the corresponding queries. Soon after, a more general interface for extending the functionality of a smart SSD was presented by Seshadri et al. in [47]. In contrast to the previous examples, the presented Willow SSD is a more advanced NVMe-based SSD, while the prior examples were SATA-based.

A similar approach using FPGA-based acceleration was presented by Woods et al. in [57]. Their IBEX system uses a SATA-based SSD connected to an FPGA. Requests are forwarded from a MySQL server

to the FPGA, which accesses the SSD to retrieve the data. Simple query processing for some data types allowed the processing of selections. Additionally, the framework supported an FPGA-based approach for string matching from prior work [53].

Based on these initial works, several different systems were developed, all targeted at different flavors of NDP using either simple embedded processors or FPGAs to perform typical query workloads. The most relevant examples are Biscuit [18] and JAFAR [4, 58], which both consider selection as a candidate for NDP. In contrast, YourSQL [25] and a similar approach [32] were aimed at the acceleration of join operations. Overall, all of these approaches are mainly focused on relational databases.

While most of the previously mentioned work focused on relational databases, some approaches are more focused on key-value stores. The earliest approaches were PapyrusKV [31] and Minerva [6]. With the advance of distributed systems in data centers, key-value stores also gained more relevance, which is one of the reasons why more recent work is typically focused on key-value-based approaches. In addition, key-value stores can be used as storage engines for relational databases, and thus acceleration and offloading in key-value stores can be transferred to relational databases. The more distributed approach in many commercial key-value stores is also why Remote Direct Memory Access (RDMA)-based research has increased. Corresponding academic implementations are Caribou [24], BlueDBM [28], and Kanzi [21]. While Kanzi and Caribou use DRAM for storage, BlueDBM also incorporates persistent flash memory. All three are focused around key-value stores that are distributed over multiple nodes. In those cases, the scope of NDP is generally broader, as operations are pushed to nodes instead of performing them on the overarching distributed key-value store.

5.3 offloading and acceleration in smart storage systems

A more recent approach at NDP is the nKV system, which is shown in Fig. 5.1. The basic idea behind nKV is using native storage in combination with NDP. To this end, the corresponding work relies on the COSMOS+ OpenSSD, which offers a completely open SSD architecture. The hardware and firmware of the COSMOS+ are publicly available and can be adapted for specific use-cases. The hardware, consisting of a flash controller and the NVMe-endpoint, are realized using the programmable logic of the Xilinx Zynq-7000 SoC. The ARM cores of the Zynq are used to implement the firmware. With the overall system described in [33], a regular consumer SSD can be approximated. While the performance of the COSMOS+ is not competitive with consumer devices, it is still a valuable resource for evaluating novel approaches.

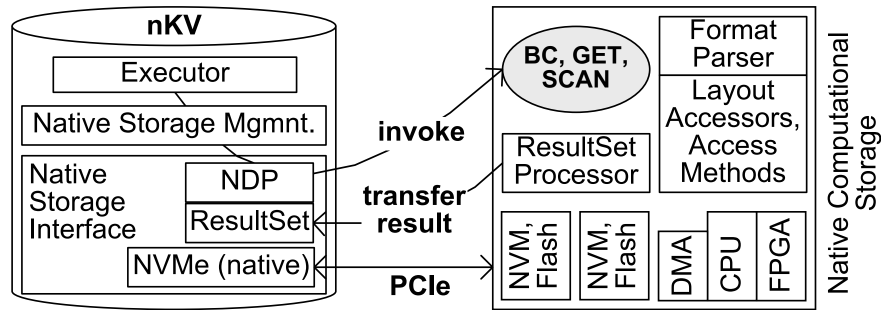


Figure 5.1: Architecture of the nKV system, taken from Chapter 12.

The nKV system relies heavily on the prior work presented in [55], which already exploited the open nature of the COSMOS+ to remove unnecessary intermediate layers. In addition, they adapted the RocksDB key-value store, originally developed at Facebook, to take direct control over the physical flash memory. Direct control's main advantage is removing intermediate layers, which eliminates write amplification. Write amplification occurs in SSDs, as flash memory has to be erased before it can be rewritten. While write operations typically manipulate single flash pages, erase operations are mostly implemented on a more coarse-grained level and erase multiple pages at the same time. Thus, writing a single flash page will typically incur matching re-writes, due to the erase operation. Additionally, intermediate abstraction layers form the operating system oftentimes also incur additional write and erase operations. In contrast, direct control of the physical memory reduces the number of unnecessary writes incurred by intermediate layers and from operations performed by the operating system.

Based on the prior work, nKV was implemented and initially presented in the publications included in Chapters 12 and 13. The corresponding publications establish a baseline for an FPGA-based smart storage device (based on the COSMOS+ OpenSSD) that can be used as a regular SSD while enabling additional NDP operations to be executed. Furthermore, the chapters go into more depth on how typical key-value operations like GET and SCAN can be accelerated by exploiting parallelism using the available logic resources of the Zynq-7000. In addition, both works demonstrate and discuss the impact of software-hardware co-design since more complex algorithms profit from hardware acceleration and intelligent orchestration from the firmware. The works also demonstrate that a significant advantage can be gained from moving to native storage. For example, we achieved a speedup of 1.4x for the SCAN operation by exploiting native storage. If we also deploy multiple hardware accelerators, this speedup can be increased to about 2.1x.

While big data applications can be optimized and improved using hardware acceleration, it typically requires a lot of time to design and

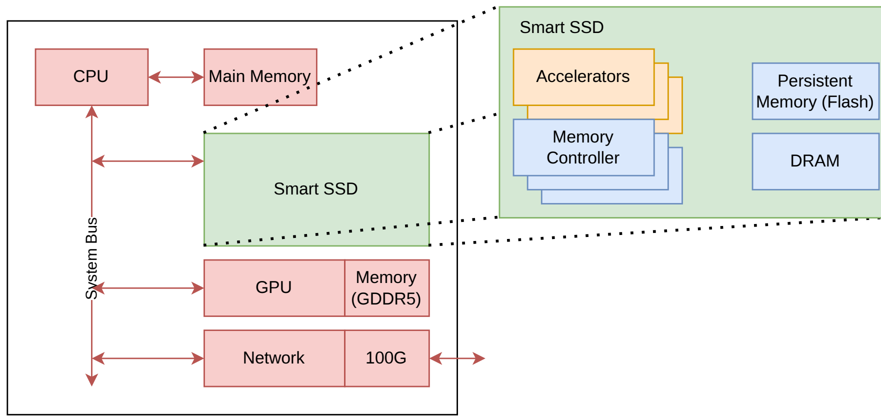


Figure 5.2: Baseline architecture with a smart SSD, containing on-board DRAM and flash memory, corresponding memory controllers and additional compute capabilities in the form of accelerators.

implement the required accelerators. In addition, hardware and software have to be co-designed and integrated, which further increases development times. To circumvent this problem, Chapter 14 introduces a framework that can generate hardware accelerators for common operations automatically based on a model of the used data structures. The work exploits that most key-value stores used application-specific data structures instead of entirely unstructured data. Extracting the underlying data structures from C-code enables the automatic generation of hardware accelerators similar to the custom ones used in prior work. In addition, the corresponding interface code can also be generated, which reduces the overhead of designing and testing custom accelerators. While the automatically generated accelerators perform slightly worse for the GET operation, their performance is on par for the more complex SCAN operation. This is especially interesting, since the tool flow presented in Chapter 14 is fully automatic and requires almost no intervention or optimization and still offers similar performance to the custom accelerators built in Chapters 12 and 13.

5.4 DISCUSSION

Generally speaking, a smart SSD could be classified as a typical architectural extension. After all, SSDs are typically devices that can simply be plugged into the mainboard of a regular desktop or server machine. The capabilities provided by an SSD are also relatively concise, and on the first glance, NDP might not be a significant architectural change. This is also obvious in Fig. 5.2, which depicts how a smart SSD would be integrated into a the baseline architecture from Section 3.1.

The reason, we still consider NDP a novel architectural paradigm lies in the programmability issue. Due to the novelty of smart storage devices, there is currently no framework or compute model in place

that simplifies the programming of those devices. Accordingly, actually using smart storage for any offloading tasks is still complex and tedious. In addition, it requires knowledge about the application domain, persistent storage, low level firmware and potentially hardware design, which further increases the complexity of working with smart storage.

Considering the development of general-purpose GPU computing or the more recent TPUs, we can see that novel paradigms can become increasingly important. With research, development and adoption, the novel paradigms become more well-known, which leads to the development of compute models and programming abstractions that make these novel architectures accessible to regular programmers. Something similar is already happening for INP, with increasing numbers of hardware vendors developing and selling corresponding smart NICs. With regards to programmability, languages like P4 are already in place to make smart NICs accessible, with a recent survey discussing over 75 publications [17].

Based on the prior paragraphs, it is clear that a major dividing factor between common architectural *extensions* and novel architectural *paradigms* is their adoption, which is typically driven by matching workloads. GPUs only became common, because most users prefer the simplicity of a graphical interface over text-based interfaces. Similarly, TPUs became important, because Google had to tackle ML workloads. Due to their importance, TPUs moved from being a completely novel paradigm to a relatively common (but still proprietary) architectural extension in less than ten years. While TPUs and GPUs are more recent examples, there is a significant number of similar examples, from floating-point and vector extensions in the ISAs of almost all general-purpose CPUs to network interfaces. If the functionality or performance increase is beneficial to a significant number of users, the adoption of matching architectural extensions increases, making the costly research and development worthwhile.

For NDP, adoption is still low and it is not yet clear, whether NDP-based smart SSDs will be as common as GPUs or TPUs in the future. Recent work has definitely shown the potential of NDP and how it can impact Big Data applications. With the increasing importance of Big Data and AI, it seems likely that smart storage will be one of the keys to continuous improvements in corresponding fields.

6 ARTIFICIAL INTELLIGENCE IN SMART STORAGE SYSTEMS

The prior Chapters 4 and 5 have already introduced SPNs and NDP. Specifically, the corresponding chapters have discussed using novel architectures to accelerate SPN inference and how smart storage devices are examples of novel architectures that could help improve database systems. Both chapters show how computer architectures can benefit AI and Big Data. In this chapter, we investigate a different approach, where SPNs are applied to solve a problem that arises in one of the potential applications of a novel architecture, namely a NDP-based smart storage device. To this end, we give a theoretical introduction to the issue of result set handling in smart storage and why it is an essential issue in Section 6.1. Using this background, we discuss prior work on Cardinality Estimation using SPNs in Section 6.2 and how it could be applied to smart storage devices. This section also serves as an overview of the publication in Chapter 15, which investigates the use of FPGA-based SPN inference for Cardinality Estimation.

6.1 RESULT SET HANDLING

In the prior Chapter 5, we have already introduced and discussed nKV, a key-value store that uses NDP and native storage on the COSMOS+ to offload and accelerate typical key-value store operations, such as GET and SCAN. The corresponding Section 12.3.5 discusses the handling of result sets as one of the issues that arise in smart storage devices. In typical storage devices, requests read or write a certain amount of data. Since the data is only read or written, its size is known and fixed throughout this process. Thus, the data transfers are pre-determined and relatively easy to perform using a Direct Memory Access (DMA) engine.

In smart storage devices, this is more complex since operations may interact with the data. For example, while the amount of data loaded from the flash memory is pre-determined, pre-processing, like selections, may reduce the volume of data. Therefore, the data transfers must be adjusted to achieve performance increases. In the original publication on nKV, this problem is circumvented by passing a result size with the NDP request that indicates an expected *maximum* transfer size. This size is then used to allocate the given space in the DRAM of the COSMOS+, and the result of the selection is stored there before it is transferred back using DMA.

For some operations, this is a good approach. For example, considering the GET operation, a single key-value pair is retrieved. The maximum transfer size is typically known or can be easily approximated by simply tracking the maximum size of key-value pairs. Since key-value stores are often used to store application-specific data, the key-value pairs typically have a known structure that can also be used to determine the result size. However, the number of results might not be known for more complex operations. While this already shows the importance of this issue, it is also likely to get even more complex with further advances in NDP approaches.

In addition, there is also a performance-related side to this issue. The COSMOS+ features a specific memory hierarchy, including persistent flash memory, onboard DRAM, on-chip Block Random-Access Memory (BRAM), and the corresponding L2/L1 caches in the ARM Cortex-A9 of the Zynq-7000 SoC. Since DMA requires that the result set is stored at least temporarily, choosing a reasonable layer of the memory hierarchy becomes an issue. If a more reductive operation is used, the result may fit into the caches or BRAM. Since BRAM and L1 caches can typically be accessed in a single cycle, they offer low latency and high throughput. Unfortunately, they are also relatively costly, so their capacity is typically small. Specifically on the COSMOS+, BRAM capacity is 19.1 Mb, and the L1 cache is 32 KB. Moving to DRAM, the COSMOS+ features 1 GB, albeit part of that is required to run the firmware. Thus, intermediate results bigger than a few hundred MB have to be stored in flash memory. While this still allows offloading of workloads, it also requires the device to split its available flash bandwidth, likely reducing the performance of the NDP request.

While there is some prior work on this issue by Vinçon et al. [56], there has yet to be a consensus on how this issue can be solved. One problem in this context is the shift in how those workloads are handled. In non-NDP approaches, the database is an application running on a regular OS. The OS will automatically handle memory allocations and ensure the data can be stored somehow. Even in worst-case scenarios, the system will typically raise exceptions and inform the user of those issues. In smart storage systems, the firmware is much less elaborate and such issues would typically result in data loss or even undefined behavior.

6.2 CARDINALITY ESTIMATION AND SUM-PRODUCT NETWORKS

A possible solution to the issue of result set handling might be cardinality estimation, a process already being researched extensively in typical relational databases. In the context of relational databases, the cardinality of a table is the number of rows in that table. Using different approaches, it is possible to estimate the number of rows in a relational database table. This information is especially valuable in

query optimization, as it can be used to order the atomic operations in more complex queries in a way that reduces the overall runtime, which is typically achieved by estimating the result size of different operations and performing those first that reduce the amount of data the most.

Most relational database systems include some cardinality estimation in their query optimization process, and several such approaches are discussed in a recent survey by Harmouch et al. [20]. Cardinality estimation approaches generally are based on *sampling* or *sketching* the dataset. For *sampling*, a reduced dataset is used to estimate the cardinality of the entire dataset. In *sketching*-based approaches, a sketch of the dataset is created by applying a hash function. The sketch is then used for estimating cardinalities. While the survey is limited to sampling- and sketch-based approaches, more recently, work has been done on using ML techniques for cardinality estimation. Specifically, SPNs can encode the distributions of underlying datasets. Hilprecht et al. first introduced a corresponding approach in [23].

The work by Hilprecht et al. introduces so-called Relational Sum-Product Networks (RSPNs), which can be used to estimate cardinalities. Additionally, they can be used to perform Approximate Query Processing (AQP). To this end, RSPNs introduce extensions specifically targeted at relational databases. Training RSPNs on singular tables yields models that can predict the cardinalities of queries. To predict more complex queries, multiple RSPNs are trained on the different tables of a relational database. The authors use this approach to target the TPC-H benchmarks and show that the SPN-based approach generally yields competitive results. In addition, due to the tractable inference on SPNs, cardinality estimation using SPNs is also relatively fast, with the fastest queries only taking about 260 µs using automatically generated and compiled C++ code on a CPU.

Interestingly, the extensions introduced by RSPNs are interesting for general-purpose databases, but are not required to achieve better results. Thus, the publication included in Chapter 15 shows that regular SPNs can estimate selectivity, which can be used to determine cardinality by multiplying with the dataset size. The work also investigates how different kinds of queries can be estimated using SPNs and determine several classes of queries and how they have to be handled. The two dividing factors are the number of queried columns, and whether we query based on equality. Generally speaking, queries operating on single columns and queries using only equality operations are relatively accurate with estimations only deviating from the correct result by up to 0.075. For more complex queries using ranges and operating on multiple columns, the estimations deviate up to 0.1. Applying these results and combining them with the prior works from Part ii, the work presents an approach for using regular SPNs for cardinality estimation using FPGA-based accelerators that could

be integrated into smart storage devices. In addition, the work also investigates the use of corresponding accelerators in regular databases, showing speedups of *almost 40x* over the prior work by Hilprecht et al.

While the integration into an actual smart storage device was beyond the scope of the publication, the evaluation highlights that even the limited resources of the COSMOS+ are sufficient to implement cardinality estimation.

7 CONCLUSION

In Chapters 1 and 2, we have discussed the new golden era for computer architecture and the corresponding increase in interest in novel computer architectures. Especially the recent hype for ML, AI and Big Data has been an important driver for the research and development of architectural improvements. Since this trend has been in progress for a few years now, there are already some interesting new architectures, such as Google's TPU or Graphcore's IPU. Based on this trend, we discuss different scopes of architectural improvements in Chapter 3, introducing a hierarchy for the classification of architectural improvements ranging from minor incremental changes to complex and costly redesigns based on new paradigms like NDP, INP and DFP.

Chapter 4 then introduces SPNs as a modern ML workload that can be accelerated using FPGA-based accelerators. The chapter introduces four publications, which are also included in Part ii, that highlight how architectural changes can impact performance. Specifically, the publications investigate custom data types and how they can increase resource and energy efficiency, as well as the impact of HBM and replication, showing that a significant issue is data movement and the limitations imposed by PCIe. One solution to this problem is the use of 100G networking, which is also discussed in Chapter 10.

Chapter 5 discusses the novel architectural paradigm NDP and corresponding smart storage devices. The chapter introduces nKV, a novel key-value store that is based around the COSMOS+ OpenSSD, which is used as a smart SSD allowing the offloading of database-specific operations to the storage device. A main advantage of NDP is the reduction in bandwidth requirements, as unnecessary data transfers can be omitted by applying data-reductive operations before data transfers occur.

While Chapters 4 and 5 show the impact of novel, tailored and optimized computer architectures on modern ML and Big Data applications, Chapter 6 investigates the use of an ML-based approach for improving the applications of smart storage devices. The corresponding publication explores the use of FPGA-based SPN accelerators for Cardinality Estimation and how this could be used in the future to solve problems arising in NDP scenarios.

Overall, we show the potential offered by novel architectures, especially in the context of Big Data and ML. In addition, we also show that ML also offers great potential in improving the applications of novel architectures. In conjunction, this thesis confirms the *new golden age for computer architecture* declared by Hennessy and Patterson.

While recent advances in computer architecture already contribute significantly to the success and progress of AI, examples like smart storage also show that there is still much more potential. Especially novel paradigms like NDP and INP have the prospect to shape and impact the computer systems of tomorrow.

BIBLIOGRAPHY

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VIII. San Jose, California, USA: Association for Computing Machinery, 1998, pp. 81–91. ISBN: 1581131070. DOI: 10.1145/291069.291026. URL: https://doi.org/10.1145/291069.291026.

[2] *AMD Extends its Leadership with the Introduction of its Broadest Portfolio of High-Performance PC Products for Mobile and Desktop*. https://www.amd.com/en/newsroom/press-releases/2023-1-4-amd-extends-its-leadership-with-the-introduction-o.html. [Online; accessed 14.05.2023]. 2023.

[3] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[4] Aurelia Augusta and Stratos Idreos. "JAFAR: Near-Data Processing for Databases." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 2069–2070. ISBN: 9781450327589. DOI: 10.1145/2723372.2764942. URL: https://doi.org/10.1145/2723372.2764942.

[5] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[6] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. "Minerva: Accelerating Data Analysis in Next-Generation SSDs." In: *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. 2013, pp. 9–16. DOI: 10.1109/FCCM.2013.46.

[7] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[8] Aaron Dennis and Dan Ventura. "Learning the Architecture of Sum-Product Networks Using Clustering on Variables." In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf.

[9] Mattia Desana and Christoph Schnörr. *Learning Arbitrary Sum-Product Network Leaves with Expectation-Maximization*. 2017. arXiv: 1604.07243 [cs.LG].

[10] David DeWitt and Jim Gray. "Parallel Database Systems: The Future of High Performance Database Systems." In: *Commun. ACM* 35.6 (June 1992), pp. 85–98. ISSN: 0001-0782. DOI: 10.1145/129888.129894. URL: https://doi.org/10.1145/129888.129894.

[11] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design and Test of Computers* 28.4 (2011), pp. 18–27. DOI: 10.1109/MDT.2011.44.

[12] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 1221–1230. ISBN: 9781450320375. DOI: 10.1145/2463676.2465295. URL: https://doi.org/10.1145/2463676.2465295.

[13] Murali Emani, Venkatram Vishwanath, Corey Adams, Michael E. Papka, Rick Stevens, Laura Florescu, Sumti Jairath, William Liu, Tejas Nama, and Arvind Sujeeth. "Accelerating Scientific Applications With SambaNova Reconfigurable Dataflow Architecture." In: *Computing in Science and Engineering* 23.2 (2021), pp. 114–119. DOI: 10.1109/MCSE.2021.3057203.

[14] Murali Emani et al. "A Comprehensive Evaluation of Novel AI Accelerators for Deep Learning Workloads." In: *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2022, pp. 13–25. DOI: 10.1109/PMBS56514.2022.00007.

[15] Robert Gens and Pedro Domingos. "Discriminative Learning of Sum-Product Networks." In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 3239–3247.

[16] Robert Gens and Domingos Pedro. "Learning the Structure of Sum-Product Networks." In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 873–880. URL: https://proceedings.mlr.press/v28/gens13.html.

[17] Bhargavi Goswami, Manasa Kulkarni, and Joy Paulose. "A Survey on P4 Challenges in Software Defined Networks: P4 Programming." In: *IEEE Access* 11 (2023), pp. 54373–54387. DOI: 10.1109/ACCESS.2023.3275756.

[18] Boncheol Gu et al. "Biscuit: A Framework for Near-Data Processing of Big Data Workloads." In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 153–165. DOI: 10.1109/ISCA.2016.23.

[19] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, et al. "[DL] A Survey of FPGA-Based Neural Network Inference Accelerators." In: *ACM Trans. Reconfigurable Technol. Syst.* 12.1 (Mar. 2019). ISSN: 1936-7406. DOI: 10.1145/3289185. URL: https://doi.org/10.1145/3289185.

[20] Hazar Harmouch and Felix Naumann. "Cardinality Estimation: An Experimental Survey." In: *Proc. VLDB Endow.* 11.4 (Dec. 2017), pp. 499–512. ISSN: 2150-8097.

[21] Masoud Hemmatpour, Bartolomeo Montrucchio, Maurizio Rebaudengo, and Mohammad Sadoghi. "Kanzi: A Distributed, In-Memory Key-Value Store." In: *Proceedings of the Posters and Demos Session of the 17th International Middleware Conference*. Middleware Posters and Demos '16. Trento, Italy: Association for Computing Machinery, 2016, pp. 3–4. ISBN: 9781450346665. DOI: 10.1145/3007592.3007594. URL: https://doi.org/10.1145/3007592.3007594.

[22] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture." In: *Commun. ACM* 62.2 (Jan. 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307.

[23] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, Not from Queries!" In: *Proc. VLDB Endow.* 13.7 (Mar. 2020), pp. 992–1005. ISSN: 2150-8097. DOI: 10.14778/3384345.3384349. URL: https://doi.org/10.14778/3384345.3384349.

[24] Zsolt István, David Sidler, and Gustavo Alonso. "Caribou: Intelligent Distributed Storage." In: *Proc. VLDB Endow.* 10.11 (Aug. 2017), pp. 1202–1213. ISSN: 2150-8097. DOI: 10.14778/3137628.3137632. URL: https://doi.org/10.14778/3137628.3137632.

[25] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, et al. “YourSQL: A High-Performance Database System Leveraging in-Storage Computing.” In: *Proc. VLDB Endow.* 9.12 (Aug. 2016), pp. 924–935. ISSN: 2150-8097. DOI: 10.14778/2994509.2994512. URL: https://doi.org/10.14778/2994509.2994512.

[26] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. “A Domain-Specific Architecture for Deep Neural Networks.” In: *Commun. ACM* 61.9 (Aug. 2018), pp. 50–59. ISSN: 0001-0782. DOI: 10.1145/3154484. URL: https://doi.org/10.1145/3154484.

[27] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit.” In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. URL: https://doi.org/10.1145/3140659.3080246.

[28] Sang-Woo Jun, Ming Liu, Sungjin Lee, et al. “BlueDBM: An Appliance for Big Data Analytics.” In: *SIGARCH Comput. Archit. News* 43.3S (June 2015), pp. 1–13. ISSN: 0163-5964. DOI: 10.1145/2872887.2750412. URL: https://doi.org/10.1145/2872887.2750412.

[29] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. “Enabling cost-effective data processing with smart SSD.” In: *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. 2013, pp. 1–12. DOI: 10.1109/MSST.2013.6558444.

[30] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. “A Case for Intelligent Disks (IDISKs).” In: *SIGMOD Rec.* 27.3 (Sept. 1998), pp. 42–52. ISSN: 0163-5808. DOI: 10.1145/290593.290602. URL: https://doi.org/10.1145/290593.290602.

[31] Jungwon Kim, Seyong Lee, and Jeffrey S. Vetter. “PapyrusKV: A High-Performance Parallel Key-Value Store for Distributed NVM Architectures.” In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126943. URL: https://doi.org/10.1145/3126908.3126943.

[32] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. “In-Storage Processing of Database Scans and Joins.” In: *Inf. Sci.* 327.C (Jan. 2016), pp. 183–200. ISSN: 0020-0255. DOI: 10.1016/j.ins.2015.07.056. URL: https://doi.org/10.1016/j.ins.2015.07.056.

[33] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. “Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems.” In: *ACM Trans. Storage* 16.3 (July 2020). ISSN:

1553-3077. DOI: 10.1145/3385073. URL: https://doi.org/10.1145/3385073.

[34] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[35] Alejandro Molina, Sriraam Natarajan, et al. "Poisson Sum-Product Networks: A Deep Architecture for Tractable Multivariate Poisson Distributions." In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI'17. San Francisco, California, USA: AAAI Press, 2017, pp. 2357–2363.

[36] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'18/IAAI'18/EAAI'18. New Orleans, Louisiana, USA: AAAI Press, 2018. ISBN: 978-1-57735-800-8.

[37] Gordon E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[38] Aaron Nicolson and Kuldip K. Paliwal. "Sum-Product Networks for Robust Automatic Speaker Identification." In: *Proc. Interspeech 2020*. 2020, pp. 1516–1520. DOI: 10.21437/Interspeech.2020-1501.

[39] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[40] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. *Probabilistic Deep Learning using Random Sum-Product Networks*. 2018. arXiv: 1806.01910 [cs.LG].

[41] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL].

[42] Hoifung Poon and Pedro Domingos. "Sum-product networks: A new deep architecture." In: *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. 2011, pp. 689–690. DOI: 10.1109/ICCVW.2011.6130310.

[43] Hoifung Poon and Pedro Domingos. *Sum-Product Networks: A New Deep Architecture*. 2012. arXiv: 1202.3732 [cs.LG].

[44] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language Models are Unsupervised Multitask Learners." In: (2019).

[45] E. Riedel, C. Faloutsos, G.A. Gibson, and D. Nagle. "Active disks for large-scale data processing." In: *Computer* 34.6 (2001), pp. 68–74. DOI: 10.1109/2.928624.

[46] Yeongeun Seo, Jaehyun Park, Jung Ho Ahn, and Taesup Moon. "Exploring Deep Learning-based Branch Prediction for Computer Devices." In: *2019 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*. 2019, pp. 85–87. DOI: 10.1109/ICCE-Asia46551.2019.8942202.

[47] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, et al. "Willow: A User-Programmable SSD." In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 67–80. ISBN: 9781931971164.

[48] Shaden Smith et al. *Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model*. 2022. arXiv: 2201.11990 [cs.CL].

[49] Lukas Sommer, Cristian Axenie, and Andreas Koch. "SPNC: Fast Sum-Product Network Inference." In: *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Cham: Springer International Publishing, 2021. ISBN: 978-3-030-93736-2. DOI: 10.1007/978-3-030-93736-2_31.

[50] Lukas Sommer, Cristian Axenie, and Andreas Koch. "SPNC: An Open-Source MLIR-Based Compiler for Fast Sum-Product Network Inference on CPUs and GPUs." In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 1–11. DOI: 10.1109/CGO53902.2022.9741277.

[51] Lukas Sommer, Michael Halkenhäuser, Cristian Axenie, and Andreas Koch. "SPNC: Accelerating Sum-Product Network Inference on CPUs and GPUs." In: *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2021, pp. 53–56. DOI: 10.1109/ASAP52443.2021.00015.

[52] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-based Accelerators." In: *IEEE International Conference on Computer Design (ICCD)*. IEEE. 2018.

[53] Jens Teubner, Louis Woods, and Chongling Nie. "Skeleton Automata for FPGAs: Reconfiguring without Reconstructing." In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 229–240. ISBN: 9781450312479. DOI: 10.1145/2213836.2213863. URL: https://doi.org/10.1145/2213836.2213863.

[54] Anda Trifan et al. "Intelligent Resolution: Integrating Cryo-EM with AI-driven Multi-resolution Simulations to Observe the SARS-CoV-2 Replication-Transcription Machinery in Action." In: *bioRxiv* (2021). DOI: 10.1101/2021.10.09.463779. eprint: https://www.biorxiv.org/content/early/2021/10/12/2021.10.09.463779.full.pdf. URL: https://www.biorxiv.org/content/early/2021/10/12/2021.10.09.463779.

[55] Tobias Vincon, Sergey Haddock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. "NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management." In: *Proc. of the 21st International Conference on Extending Database Technology (EDBT)*. 2018.

[56] Tobias Vincon, Christian Knoedler, Arthur Bernhardt, Leonardo Solis-Vasquez, Lukas Weber, Andreas Koch, and Ilia Petrov. "Result-Set Management for NDP Operations on Smart Storage." In: *Data Management on New Hardware*. DaMoN'22. Philadelphia, PA, USA: Association for Computing Machinery, 2022. ISBN: 9781450393782. DOI: 10.1145/3533737.3535097. URL: https://doi.org/10.1145/3533737.3535097.

[57] Louis Woods, Jens Teubner, and Gustavo Alonso. "Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 1073–1076. ISBN: 9781450320375. DOI: 10.1145/2463676.2463685. URL: https://doi.org/10.1145/2463676.2463685.

[58] Sam Likun Xi, Aurelia Augusta, Manos Athanassoulis, and Stratos Idreos. "Beyond the Wall: Near-Data Processing for Databases." In: *Proceedings of the 11th International Workshop on Data Management on New Hardware*. DaMoN'15. Melbourne, VIC, Australia: Association for Computing Machinery, 2015. ISBN: 9781450336383. DOI: 10.1145/2771937.2771945. URL: https://doi.org/10.1145/2771937.2771945.

Part II

SUM-PRODUCT NETWORKS

8 RESOURCE-EFFICIENT LOGARITHMIC NUMBER SCALE ARITHMETIC FOR SPN INFERENCE ON FPGAS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs* by *Lukas Weber, Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting, Andreas Koch* in *2019 International Conference on Field-Programmable Technology (ICFPT)*. The contribution of the author of this thesis is summarized as follows.

> » *As the corresponding and leading author, Lukas Max Weber was responsible for the initial design and implementation of the logarithmic number system operators and corresponding optimizations toward Sum-Product Network inference. He also contributed the necessary adaptions to the existing automatic tool flow. Alejandro Molina provided the set of benchmark SPNs. A custom code-to-code compilation flow for comparison against CPUs and GPUs and corresponding results were provided by Lukas Sommer. The resource and performance evaluation was also conducted by Lukas Max Weber. The manuscript was written by Lukas Max Weber, with feedback and support from all other co-authors.* «

ABSTRACT

FPGAs have been successfully used for the implementation of dedicated accelerators for a wide range of machine learning problems. The inference in so-called Sum-Product Networks can also be accelerated efficiently using a pipelined FPGA architecture.

However, as Sum-Product Networks compute exact probability values, the required arithmetic precision poses different challenges than those encountered with Neural Networks. In previous work, this precision was maintained by using double-precision floating-point number formats, which are expensive to implement in FPGAs.

In this work, we propose the use of a logarithmic number system format tailored specifically towards the inference in Sum-Product Networks. The evaluation of our optimized arithmetic hardware operators shows that the use of logarithmic number formats allows to save up to 50% hardware resources compared to double-precision floating point, while maintaining sufficient precision for SPN inference at almost identical performance.

8.1 INTRODUCTION

In the past, most of the work on FPGA-based accelerators for machine learning inference has focused on the acceleration of (artificial) neural networks [8], such as the very popular convolutional neural networks.

In contrast, in [10, 11] an automatic tool-flow for mapping the inference for so-called *Sum-Product Networks* (SPN) to an FPGA-based accelerator was developed. Sum-Product Networks are a very promising type of machine learning network, that belong to the class of tractable probabilistic models and share similarities with probabilistic graphical models (PGM).

Compared to "classical" neural networks, SPNs allow *exact* inference, thereby allowing them to explicitly deal with uncertainty over the inputs. However, the fact that Sum-Product Networks compute *exact* probabilities poses a number of unique challenges to the hardware implementation. Most of the optimization techniques employed for the hardware mapping of neural networks, such as quantization of weights, are not readily applicable to SPNs.

In [10], the use of double-precision allowed the preservation of sufficient accuracy at the cost of a relatively high resource requirement per operator. To circumvent this problem, we seek to use an optimization which is commonly used in ML. The use of logarithmic scaling is often used on CPUs and GPUs even though the scaling has to be emulated. Using the flexibility of FPGAs, we aim to implement the logarithmic scaling through a logarithmic number system (LNS). This allows us to maintain sufficient accurary with smaller bitwidths, leading to significant resource savings.

8.2 SUM-PRODUCT NETWORKS

Probabilistic models can be used to solve a range of machine learning problems. For example, the problem of multi-class classification can be solved with probabilistic queries on a PGM by determining the class with the highest probability, given some evidence, i.e., $arg\ max_c\ \mathcal{P}(class = c|evidence)$.

While PGMs are very versatile, they generally suffer from one disadvantage: In general, inference in unrestricted PGMs (e.g. Baysian Networks) is intractable. SPNs overcome this limitation and allow to compute *exact* probabilities in time *linear* wrt. to the network size. Additionally, SPNs inherit the universal approximation property from mixture models, as mixture models can easily be represented as SPNs using a single sum-node. This means that SPNs are able to represent any prediction function, similar to deep neural networks.

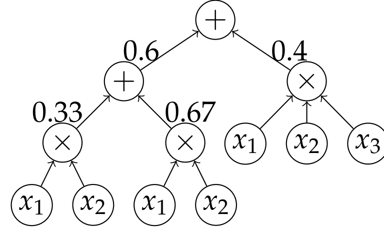


Figure 8.1: Example of a valid SPN, capturing the joint probability distribution of the variables x_1, x_2 and x_3.

8.2.1 *Model Representation*

A Sum-Product Network captures the joint probability $\mathcal{P}(X_1, X_2, ...)$ over a set of variables, called the scope of the SPN. The graph structure of the SPN, used to represent this joint probability distribution, is a rooted, directed acyclic graph with three different kinds of nodes: Sum, product and leaf nodes. With these three node types, an SPN can be defined recursively as follows:

1. A tractable, univariate distribution is an SPN. This corresponds to the leaf nodes in the network. 2. A product of SPNs over different scopes (i.e., random variables) is an SPN, represented by a product node in the network. Essentially, a product node corresponds to a factorization over independent distributions. 3. A convex combination (i.e., weighted sum) of SPNs over the same scope is an SPN. This is equivalent to a mixture of multiple distributions over the same variables and represented by a sum node and the weights associated with each of the child nodes. An example SPN is given in Fig. 8.1.

8.2.2 *Inference*

To answer probabilistic queries using SPNs the following scheme is used: Given (partial) evidence, histograms at the leaf nodes are evaluated, mapping input values to probabilities. As in [10], based on [9], all leave nodes use discrete input values. Using the probabilities of the leaves, nodes are evaluated bottom-up to calculate the resulting probability. The evaluation always results in a single probability value.

The basic case is the computation of the joint probability $\mathcal{P}(X_1, X_2, ...)$, which corresponds to a single evaluation with full evidence. In order to marginalize out one or multiple variables, it is sufficient to replace the leaf nodes for these variables with the value 1 and evaluate the SPN. Both cases can be combined to compute the conditional probability: $\mathcal{P}(Y|X) = \frac{\mathcal{P}(Y,X)}{\mathcal{P}(X)}$.

In this work, we focus on the joint computation, but the accelerator can be extended to also compute marginals and conditional probabilities.

8.3 PRIOR WORK

8.3.1 *Logarithmic Number Scale on FPGA*

Regarding the use of LNS, there has been extensive research. Especially for digital signal processing it gained traction through the works of Lewis, which employed a function interpolation scheme using interleaved memory to calculate the logarithmic addition [5]. This resulted in the development of an *arithmetic unit* (AU) which, at the time, outperformed all similar floating point-based AUs [6].

A very similar interpolation-based approach for calculating the logarithmic addition was later used [3]. The work additionally compared FP and LNS and determined, that for LNS to outperform FP in latency and required area, it was necessary that about 70% of operations were multiplicative.

8.3.2 *SPN Inference on FPGA*

To the best of our knowledge, the work presented in [10] is the first and to date only approach to accelerate SPN inference on FPGAs. In that work, an automatic toolflow that maps the inference in Sum-Product Networks to a fully pipelined FPGA-accelerator was developed. The toolflow uses a fully spatial mapping, i.e., for each arithmetic operator in the SPN, there is a corresponding hardware operator in the datapath. For the implementation of the hardware arithmetic operators, the FloPoCo framework [2] was used, with a numeric format similar to IEEE-754 double precision, achieving end-to-end speedups of 6x over an x86-CPU and 38x over a Tensorflow-based GPU-implementation.

In this work, we design LNS operators as drop-in replacement for the FloPoCo-operators in [10] and reuse the automatic toolflow to automatically map SPN inference to the FPGA.

8.4 APPROACH

Similar to Haselman et al. [3], we use a fixed-point number to encode the exponent E_A for a probability value A. In addition to the exponent, a zero-flag Z_A and a sign-flag S_A are used to denote special cases and the sign of the exponent. Since we only consider probabilities (values between 0 and 1), the sign-flag is inherently also a flag that indicates that the linear-scale value is 1, similar to the zero-flag which indicates a linear-scale value of 0.

In contrast to the encodings used by Haselman et al. [3] or Detrey et al. [1], this encoding removes the additional sign-bit, which is used to encode the overall sign for representing negative numbers. In addition, we chose to change from 2's complement encoding of the exponent to an encoding with an explicit sign-flag. Additionally, we do not encode

special cases such as NaN or $\pm\infty$. Thus we are able to save a bit, while the magnitude of the exponent gains an additional bit in comparison to [3].

8.4.1 *LNS Multiplication*

The multiplication of values in linear scaling corresponds to an addition in logarithmic scale. This is also visible in the corresponding logarithmic property: $log_2(x \times y) = log_2(x) + log_2(y)$. Assuming correct input values and neither input being 0 nor 1, the calculation is a simple addition of the exponents. If either Zero-flag is set, the result is zero (linear-scale multiplication by 0). Additionally, if the addition of exponents overflows, this results in a value that is too small and thus saturated towards 0.

While the resulting sign is $S_R = S_A \vee S_B$, the exponent is $E_R = E_A + E_B$. Additionally, the zero-flag is $Z_R = Z_A \vee Z_B$. The special case handling will override E_R and S_R to zero, if one of the operands zero-flag was set, or if the calculation of E_R overflowed.

In actual hardware, this is split into 3 pipeline stages: Decoding, calculation and special-case handling.

8.4.2 *LNS Addition*

In contrast to multiplication, addition is not simplified in the logarithmic scale. Instead, the calculation of an addition in logarithmic scale is *harder* than in linear scale. The main challenge with the logarithmic addition is obvious in the corresponding logarithmic property: $log_2(x + y) = log_2(x) + log_2(1 + 2^{(log_2(y) - log2(x))})$. Similar to [1, 3], we implement this using a simple piecewise polynomial of second degree.

The implementation of the interpolation poses an additional challenge, since it relies on binary arithmetic. The required bitwidth of these operations depends on the bitwidths of the exponents. For increased accuracy of these operations, the operations internal to the interpolator are up to *twice* the regular bitwidth. To achieve acceptable clock frequencies, we have to pipeline these operations and exploit the available special function slices.

For binary additions, this does not pose much of a challenge, since the carry-save chains on modern FPGAs generally allow additions of at least 40 bits without problems. Dividing the operands in chunks of corresponding size allows easy chaining of these adders using intermediary pipeline registers. The resulting adders are similar to pipelined Ripple-Carry Adders.

In contrast, the creation of the corresponding multipliers is much more complex. For FPGA applications, the de-facto standard for generating these multipliers is given by the work of Kumm et al. [4], which relies on *integer linear programming* (ILP) to calculate resource-optimal

multiplier compositions. Adapting their approach, it was possible to also avoid the high LUT utilizations of [10]. By solely using DSP-tiles, the resulting ILP solutions are perfectly tiled to map to DSPs only. Depending on the placement of the DSP-tile, Vivado then infers DSPs only if they are used efficiently or if a corresponding directive is used. Evaluating both approaches, we found the inference-option to be more resource efficient.

Using these binary additions and multiplications as primitives, we built the interpolation unit which calculates the interpolating polynomial $ax^2 + bx + c$. The coefficients a, b, c are pre-computed and stored in *read-only memory* (ROM). To reduce the size of these ROMs, we store only the fraction-bits and a single integer bit due to the observation by Vouzis et al. [13], that interpolation coefficients for this interpolated function are positive and in the range $[0, 1]$.

Using the interpolation unit we can now compose a unit for logarithmic addition by considering all possible cases under the assumptions that $|A| \geq |B|$, $x = \mathit{interpolate}(E_B - E_A)$ and $F_u = \mathit{underflow}(x)$. 1. Z_R is set only if Z_A and Z_B were set. 2. S_R is unset, unless Z_R or F_u. 3. If Z_R is set and S_R is unset, $E_R = 0$. In all other cases, $E_R = E_A - x$.

To actually implement this, a pipelined approach is used. After splitting the operands into exponents and flags, an additional stage ensures that $|A| \geq |B|$ holds, switching the operands if necessary. Then the difference of the exponents is calculated and pushed into the interpolation unit. Using the interpolation result and the flags we can detect the special cases and handle them accordingly.

8.5 EVALUATION

8.5.1 *Benchmarks*

For full comparability, we use the same set of benchmark datasets as in [10]. That set contains benchmarks of two different types, count-based and binary. While the count-based examples are taken from the well-known NIPS[1] corpus, the binary benchmarks are pre-processed and provided by [7] and [12]. A more detailed description of each of the datasets can be found in [10].

8.5.2 *Parameters*

We tuned the parameters of our operators (i.e. integer & fractional bit-width, interpolation error ceiling) using an iterative process. We identified a configuration with eight integer-bits, 32 fraction bits and an interpolation error of $2^{-21.5}$ as minimal configuration to maintain sufficient accuracy across all benchmarks. The acceptable interpolation error depends on the size of each benchmark, with smaller SPNs

1 archive.ics.uci.edu/ml/datasets/bag+of+words

tolerating higher interpolation error ($2^{-18.5}$ for *NIPS5*) and bigger SPNs requiring smaller interpolation errors ($2^{-21.5}$ for *Accidents*).

8.5.3 *FPGA Resource Consumption*

As target device for the FPGA evaluation, we select the Xilinx VC709 development board, containing a Virtex7-device (xc7vx690) and 4 GiB of RAM. With the improvements in TaPaSCo and Vivado we opted to reproduce the results from [10] using current tool versions TaPaSCo 2019.10 and Vivado 2019.1, again for better comparability.

To achieve the best possible results, we employed the design-space exploration feature of TaPaSCo, which automatically maximizes the design frequencies. The resulting frequencies and resource utilizations can be found in Table 8.1. For brevity, those numbers are given relative to the entire FPGA in percent[2].

Throughout the complete set of Benchmarks, the LNS-based implementations require *fewer* Slices and *fewer* DSPs than their FP-counterparts. Due to the use of ROMs for storing the coefficients for the interpolation, BRAM utilization is higher in LNS-implementations. The BRAM requirements are slightly more than doubled for the examples *Accidents* and *NIPS80*. However, that increase is almost irrelevant, since the original BRAM requirements were always below 5%, and thus the worst-case example (*NIPS80*) only requires 10% of BRAM.

In contrast, the resulting utilizations of Slices and DSPs are *always reduced*, depending on the size of the SPN and its adder-to-multiplier-ratio. For small examples such as *NIPS10*, the utilization is reduced by 1.8% and 2.2% for slices and DSPs, respectively. In the biggest example (*NIPS80*), the reduction grows to 44.8% *fewer* slices and 23.7% *fewer* DSPs.

8.5.4 *Performance Evaluation*

Similar to [10], we compare our accelerators to CPU- and GPU-implementations. For the CPU, we use the best results obtained in [10]. For the GPU, we implemented a custom CUDA-based compilation flow and evaluated it using the Nvidia CUDA compiler nvcc in version 10.0.130 and an Nvidia 1080Ti (11GB). Our new flow is up to 90x faster than the one used in [10].

Regarding throughput, the FPGA-implementations will generally outperform CPU and GPU, unless data transfer overhead exceeds a threshold. Examples for this are *NIPS5*, *NIPS10* and *NIPS20*. For these, the throughput of the CPU exceeds the corresponding throughput of all other implementations. As soon as the networks become larger, the FPGA-implementations will outperform CPU and GPU by

2 The absolute number of resources available are 433,200 (LUT), 866,400 (Register), 108,300 (Slices), 1,470 (BRAM) and 3,600 (DSP), respectively.

Table 8.1: FPGA implementation results using double-precision floating-point (FP) and logarithmic number scale (LNS) arithmetic operators, respectively. For brevity those numbers are given as usage relative to the resources available on the FPGA in percent.

	Slices [%]		BRAM [%]		DSP [%]		Freq. [MHz.]	
Benchmark	FP	LNS	FP	LNS	FP	LNS	FP	LNS
Accidents	69.4	42.0	3.5	7.3	36.1	17.2	205	**230**
Audio	85.2	34.8	3.5	4.8	45.8	7.6	205	**229**
Netflix	73.7	38.2	3.5	4.6	38.5	7.0	199	**250**
MSNBC200	59.8	42.7	3.5	6.6	27.5	19.1	**250**	225
MSNBC300	43.9	39.5	3.5	5.3	17.0	10.8	**250**	**250**
NLTCS	57.6	40.8	3.5	6.3	25.2	17.2	250	**265**
Plants	82.0	35.8	3.5	5.9	42.6	8.9	205	**233**
NIPS5	28.4	25.7	3.7	3.7	1.6	0.6	250	**255**
NIPS10	31.4	29.6	3.7	4.1	4.1	1.9	255	**270**
NIPS20	40.0	32.2	4.1	4.8	9.3	4.4	**255**	250
NIPS30	43.8	36.5	3.7	5.0	14.5	6.3	220	**230**
NIPS40	51.3	41.2	4.0	5.7	20.3	10.2	226	**255**
NIPS50	57.9	43.1	4.3	6.0	23.8	10.2	210	**250**
NIPS60	66.9	41.3	4.8	6.0	26.0	8.3	217	**240**
NIPS70	73.3	43.3	4.5	5.9	30.0	8.9	210	**215**
NIPS80	93.0	48.2	4.8	10.0	44.1	20.4	190	**240**

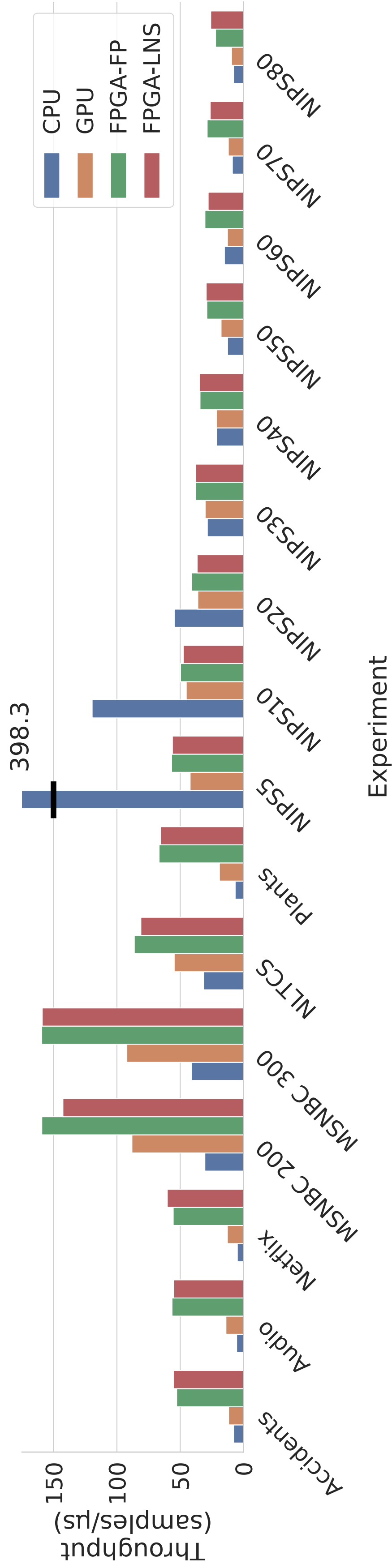


Figure 8.2: Throughput of the CPU-, GPU- and both FPGA-implementations in samples/μs. Each group represents an example SPN. The single outlier is the CPU-Throughput for example NIPS5 which is cut off from more than 2x its size to fit.

many times. For the example *Netflix*, the throughput of both FPGA-implementations is more than 11.4x of CPU and 4.7x of the GPU. Fig. 8.2 shows the throughput of all implementations.

Comparing the throughputs of both FPGA-implementations, only minor differences exist. On average, the more area-efficient LNS-variants have 1.1% reduced throughput. Note that GPU and FPGA throughput data *includes* PCIe data transfer overhead. Similar to [10], this can be up to 80% of overall compute time (*NIPS20*).

8.6 CONCLUSION & OUTLOOK

In this work, we have developed a specialized logarithmic number format for the use in Sum-Product Network inference and implemented highly efficient, pipelined hardware arithmetic operators for addition and multiplication. Our hardware operators seamlessly integrate with the existing framework developed in [10], which allows to automatically generate fully pipelined FPGA-accelerators for SPN inference.

We compared our implementation with the existing work [10] and CPU- and GPU-implementations of SPN inference. Our evaluation shows that we can maintain sufficient precision with just 42 bits for the LNS format, whereas the FloPoCo-operators in prior work use 66 bits, leading to reductions in logic resource consumption (Slices, DSPs) of up to 50%. At the same time, we are able to maintain *similar* performance to [10], significantly outperforming the GPU- and CPU-based implementations in thirteen out of sixteen examples.

In the future, we plan to extend the synthesis flow for other types of queries, such as marginalization.

ACKNOWLEDGEMENTS.

The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software. Additionally, calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt.

Finally, the authors would like to thank Martin Kumm, for much appreciated discussions of the subject.

REFERENCES

[1] J. Detrey and F. de Dinechin. "A VHDL library of LNS operators." In: *Asilomar Conference on Signals, Systems Computers*. 2003. DOI: 10.1109/ACSSC.2003.1292376.

[2] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo." In: *IEEE Design & Test of Computers* (2011).

[3] M. Haselman et al. "A comparison of floating point and logarithmic number systems for FPGAs." In: *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. 2005. DOI: 10.1109/FCCM.2005.6.

[4] M. Kumm et al. "Resource Optimal Design of Large Multipliers for FPGAs." In: *IEEE 24th Symposium on Computer Arithmetic*. 2017. DOI: 10.1109/ARITH.2017.35.

[5] D. M. Lewis. "An accurate LNS arithmetic unit using interleaved memory function interpolator." In: *11th Symp. on Computer Arith.c*. June 1993. DOI: 10.1109/ARITH.1993.378115.

[6] D. M. Lewis. "114 MFLOPS logarithmic number system arithmetic unit for DSP applications." In: *Proceedings ISSCC - International Solid-State Circuits Conference*. 1995. DOI: 10.1109/ISSCC.1995.535287.

[7] Daniel Lowd and Jesse Davis. "Learning Markov network structure with decision trees." In: *IEEE 10th Int. Conf. on Data Mining (ICDM)*. 2010.

[8] Janardan Misra and Indranil Saha. "Artificial neural networks in hardware: A survey of two decades of progress." In: *Neurocomputing*. Artificial Brains (2010). ISSN: 0925-2312.

[9] Alejandro Molina et al. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: *Proceedings of AAAI*. 2018.

[10] L. Sommer et al. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *IEEE 36th International Conference on Computer Design (ICCD)*. 2018.

[11] Lukas Sommer et al. "Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem." In: *ICML Workshop on Tractable Probabilistic Models (TPM)*. 2018.

[12] Jan Van Haaren and Jesse Davis. "Markov Network Structure Learning: A Randomized Feature Generation Approach." In: *AAAI*. 2012.

[13] P. D. Vouzis et al. "Cotransformation Provides Area and Accuracy Improvement in an HDL Library for LNS Subtraction." In: *DSD 2007*. 2007. DOI: 10.1109/DSD.2007.4341454.

Copyright Notice: © 2019 IEEE. Reprinted, with permission, from Lukas Weber et al. "Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs." In: *International Conference on Field-Programmable Technology (FPT)*. 2019, pp. 251–254. DOI: 10.1109/ICFPT47387.2019.00040.

9

COMPARISON OF ARITHMETIC NUMBER FORMATS FOR INFERENCE IN SUM-PRODUCT NETWORKS ON FPGAS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs* by *Lukas Sommer, Lukas Weber, Martin Kumm and Andreas Koch* in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. The contribution of the author of this thesis is summarized as follows.

» *As a co-leading author, Lukas Max Weber designed, implemented, and optimized logarithmic number system hardware operators for Sum-Product Network inference. Additionally, he pipelined and optimized existing Posit Operators. Martin Kumm contributed custom floating-point operators. Lukas Max Weber integrated all operators into the existing tool flow, contributed by Lukas Sommer. A source-to-source compiler for fast software based-based design-space exploration of arithmetic formats and fast inference on CPUs and GPUs was implemented by Lukas Sommer, who also carried out the corresponding experiments. FPGA performance measurements, as well as the design-space exploration, were conducted by Lukas Max Weber. The manuscript was joint work of all authors.* «

ABSTRACT

Probabilistic Graphical Models (PGM) have recently received increasing attention for various machine learning tasks and approaches for their acceleration on FPGAs have been presented.

In this work, we investigate three different arithmetic formats, namely customized floating-point, Posit and logarithmic number systems with regard to their suitability for the inference in PGMs, specifically so-called Sum-Product Networks (SPN). Based on results from an automatic design-space exploration developed in this work, we implement hardware arithmetic operators for each format, optimized for SPN inference.

Our evaluation shows that the choice of the most area-efficient solution depends on the relation between the numbers of adders to multipliers in the network. Up to 57% and 68% of Slice and DSP reductions, respectively, could be obtained compared to previous work. With regard to performance, all formats achieve similar results and

outperform CPU and GPU-based implementations of SPN inference by factors up to 12x and 4.6x, respectively.

9.1 INTRODUCTION

Next to GPUs and custom ASICs, such as Google's TPU, FPGAs have established themselves as a succesful implementation platform for the acceleration of machine learning (ML) tasks, in particular for inference. Besides numerous works on the acceleration of the inference in neural networks, for example convolutional neural networks (CNN) for computer vision applications, new approaches to accelerate inference in *probabilistic models* on FPGAs have recently been presented.

One such approach for the inference in so-called *Sum-Product Networks* (SPN) was developed in [26, 27, 29]. Compared to neural networks, Sum-Product Networks, which belong to the class of *tractable* Probabilistic Graphical Models (PGM), can better deal with missing input features and, as SPNs compute *exact* probability values, are also able to express uncertainty over their outputs.

However, this ability also poses new challenges to the implementation of such networks on FPGAs. In [26, 27], the authors used a double-precision floating-point format to preserve accuracy. Such an arithmetic format is expensive to implement on FPGAs. Therefore, in this work, we seek to optimize the hardware arithmetic operators to *reduce* resource usage, while *preserving* sufficient accuracy. To this end, we will investigate three different arithmetic formats, namely "traditional" but customized floating point, logarithmic number system (LNS) and Posit, with regard to their suitability for FPGA-based accelerators for SPN inference.

We exploit an automatic and efficient design-space exploration (DSE) flow, based on software-only emulation of the arithmetic formats for SPN inference, to determine the minimal bit-widths required to preserve accuracy with each of the formats prior to hardware generation.

Based on the findings from our DSE, we then implement hardware arithmetic operators for each of the three investigated arithmetic formats, optimized for the inference in Sum-Product Networks on FPGAs. The optimized arithmetic operators are used to generate fully pipelined datapaths, which are integrated into a SoC-design providing the host-CPU software interface. In our extensive evaluation, we investigate which arithmetic format is most suited for SPN inference on FPGAs and compare the performance of the generated datapaths with CPU and GPU-based implementations of SPN inference.

9.2 SPN BACKGROUND

Sum-Product Networks [22] belong to the class of *probabilistic models*, which can be used for a range of different machine learning tasks. As they are also able to take the statistical nature of the data into account, and deal well with uncertainty and missing features, this class of models has received increasing attention recently.

After a probabilistic model has been trained from data, different machine learning problems, such as classification and regression, can be solved by using probabilistic queries on the trained model. An example for such a query would be to determine which news-article a user is most likely interested in, based on information on whether or not he or she has looked at other articles before.

In comparison with other probabilistic models and other ML-techniques, such as deep neural networks, SPNs exhibit a number of interesting characteristics, that makes them attractive for use in a range of different applications. For example, SPNs have already been used succesfully for sequence labeling [24], i.e., classifying the characters in a handwritten sequence, or in path planning algorithms for mobile robots [23].

One very important property of SPNs for their practical usage is the *efficiency* of the inference: While in general, inference for unrestricted PGMs is *intractable*, the inference in SPNs is guaranteed to be *linear* w.r.t. the number of nodes [3, 22]. This *tractable* inference is key to efficiently answering probabilistic queries in practical applications.

Another interesting property of SPNs is their expressiveness: From mixture models, which can easily be represented by a shallow Sum-Product Network with a single sum-node, SPNs inherit the *universal approximation property* [20]. This means that Sum-Product Networks can represent *any prediction* function, similar to deep neural networks.

One of the most interesting properties about Sum-Product Networks, that also makes SPNs stand out from other ML-techniques such as deep neural networks, is the *precision* of the inference process. Whereas neural networks generally compute *approximate* values, Sum-Product Networks are instances of Arithmetic Circuits [30] and therefore facilitate the computation of *exact* probability values. Beyond more precise answers to queries, this also offers the advantage that the inference process can be combined with *anomaly detection* by comparing the respective probabilities from different SPNs, and also better account for the statistical nature of the data.

In this work, we focus on the *inference* process in a pre-trained SPN. In this case, the learning has taken place offline on a traditional CPU-based machine.

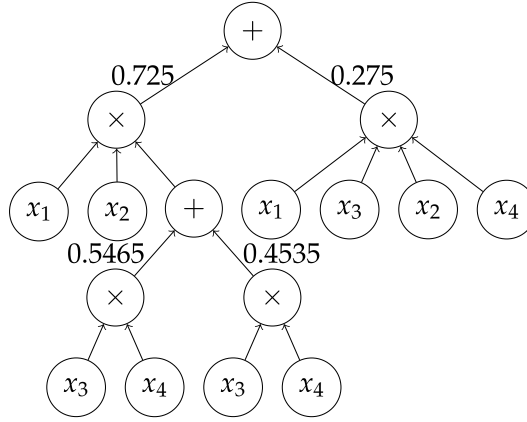


Figure 9.1: Example of a valid SPN representing the joint probability $\mathcal{P}(x_1, x_2, x_3, x_4)$.

9.2.1 *Model Representation*

A Sum-Product Network captures the joint probability $\mathcal{P}(X, Y, Z)$ over a set of variables $\{X, Y, Z\}$ in the form of a rooted, directed acyclic graph (DAG). An example for a valid SPN over the variables $\{x_1, x_2, x_3, x_4\}$ can be found in Fig. 9.1. The graph representation of SPNs is composed from three different kinds of nodes, with some additional restrictions to guarantee the validity of the SPN:

- Leaf nodes represent univariate distributions over a single variable. In this work, based on the approach proposed by Molina et al. [19], we represent these univariate distributions by histograms for an efficient mapping to the FPGA.

- Factorizations over independent distributions are represented by product nodes in the graph. The child nodes of a product node are defined over different scopes, i.e., each sub-tree uses a distinct set of variables.

- Mixtures over distributions defined over the same set of variables are represented by sum-nodes, where each child node is additionally associated with a weight. The child nodes of a sum node are defined over the same scope, i.e., the same set of variables appears in each subtree.

9.2.2 *Inference*

The inference process depends on the kind of probabilistic query that should be answered. Common to all kinds of inference is the bottom-up evaluation of the SPN graph, eventually yielding a probability value at the root of the graph.

The most basic kind of inference in an SPN is the joint computation, yielding the joint probability for given input values, i.e., full evidence.

In the first step, the leaf nodes are queried with the value of the associated input variable, yielding a probability value. In this work, the univariate distributions at leaf nodes associated with an input variable are modeled using histograms, which are simply indexed with the input value. The resulting probability values are then propagated upwards through the tree. At product nodes, the child node values are multiplied with each other. When a sum node is reached, the child node values are first multiplied with the corresponding weight and then summed up.

Marginalization [20] of variables is another possible kind of query that can be answered by inference. To this end, the leaf nodes associated with the marginalized input variables are replaced by the probability 1. The remaining leaf nodes are just queried with the associated input values from the partial evidence. The rest of the inference process is identical to the joint computation. Through the combination of joint computation and marginalization, it is also possible to compute conditional probabilities using the following equation, where the numerator of the fraction corresponds to the joint computation and the denominator can be computed by marginalization of Y: $\mathcal{P}(Y|X) = \frac{\mathcal{P}(Y,X)}{\mathcal{P}(X)}$.

In this work, we focus on joint computation, but the datapath architecture can easily be extended to support other kinds of inference, such as marginalization.

In prior work, accelerators for the inference in other Probabilistic Graphical Models such as Bayesian Networks (BN) [1] or Markov Random Fields (MRF) [5] were developed. However, as discussed in the previous section, the inference in these kinds of PGMs differs significantly from Sum-Product Networks and the techniques used in these works cannot be applied to SPNs without further ado.

To the best of our knowledge, the only approach to accelerate SPN inference on FPGAs was presented in [26, 27]. In this work, we seek to extend the automatic toolflow from this work with three different arithmetic formats.

9.3 ARITHMETIC NUMBER FORMATS

9.3.1 *Fixed Point*

Fixed-point arithmetic can be implemented very efficiently in FPGAs. Yet, we do not consider fixed-point further in this work, because with SPNs, very small numbers can still represent significant results. In [19], the authors reported on relevant *log-likelihoods* as small as -144 and a first analysis of the dynamic range of the results of our benchmark networks showed that the smallest numbers are as small as $1.85 \cdot 10^{-88}$.

As each number can also be as large as one, at least 292 bits would be necessary to encode this number. A binary multiplier of corresponding size would require over 200 DSP-slices and is thus not a viable option.

The fact that such small numbers can still be significant for the outcome of the SPN and the result of the ML-task is also the reason why we use comparisons in *log-space* to compute the deviation from reference results in the rest of this work.

9.3.2 *Floating Point*

As motivated above, for applications requiring a large dynamic range, the word length w of fixed-point numbers may get excessively large. *Floating point* (FP) numbers provide a much wider dynamic range, at the cost of a reduced precision, for the same number of bits. An FP number X according to the IEEE 754 standard is represented as $X = (-1)^s \times 1.f \times 2^{e-e_0}$, where s is the sign bit (0 for positive, 1 for negative), f is the fraction and e is the exponent field. The exponent field e is a w_e bit unsigned integer that represents the signed exponent $e - e_0$, where e_0 is called the bias defined as $e_0 = 2^{w_\text{e}-1} - 1$. As the FP format is normalized such that the leading bit of the significant is equal to '1', only the fractional bits of the mantissa are stored in w_m bits.

9.3.3 *Posit*

The Posit arithmetic format is a comparably young format, introduced in 2017 as an implementation of type-3 unum (universal number) arithmetic [12]. The Posit format is characterized by two parameters, the total number of bits in the format w and the number of bits used to represent the exponent w_es.

As shown in Fig. 9.2, the Posit number representation is composed of four parts.

Negative numbers are encoded as 2's complement where the most significant bit (s) indicates the sign of the number.

The next component, the so-called *regime*, distinguishes Posit from traditional floating point formats. The regime is represented using a variable run-length (or thermometer) encoding, i.e., a sequence of bits with identical value terminated by a bit of the opposite value, where the length of the sequence represents the encoded value. As an example, the sequence 0001 encodes the value -3, whereas the sequence 110 encodes the value 2.

The third component, the *exponent*, is encoded as a binary number using a fixed size of w_es bits. In contrast to IEEE754 floating point, the exponent only encodes *positive* numbers and no bias is used.

The last component is the *mantissa*, which is stored just as in IEEE754 floating point, with an implicit leading 1 omitted. The mantissa occu-

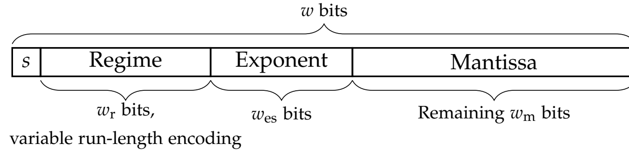


Figure 9.2: Posit binary format.

pies the remaining w_m bits, that are left after the run-length encoding of the regime and the fixed-size exponent.

Because the length of the regime is only limited by $w - 1$, the mantissa and also the exponent may not be present at all.

Given the sign bit s, a regime value r, the exponent e and the mantissa f, the number represented in Posit can be computed as follows: $(-1)^S \times \mathrm{useed}^r \times 2^e \times 1.f$, where $\mathrm{useed} = 2^{2^{w_\mathrm{es}}}$. As an example, with $w = 7$ and $w_\mathrm{es} = 2$, the bit-sequence 0␣01␣11␣10 encodes the decimal value $(-1)^0 \times (2^{2^2})^{-1} \times 2^3 \times 1.10_2 = 0.75$.

Multiple previous works have developed Posit arithmetic hardware operators for FPGAs. While [15] and [21] found that Posit incurred a significant area overhead over traditional floating point, the operators developed in [4] required resources comparable to FP implementations and for the particular application investigated in this work, floating point could be replaced with a smaller bit-width and more area-efficient Posit format. As only the operators from [15] are available open-source, we build on this library for the implementation of the Posit hardware operators in this work. We extend the operators to meet our requirements as described in Section 9.5.3.

9.3.4 *Logarithmic Number System*

Originally, Logarithmic Number Systems (LNS) were developed as an alternative to floating point numbers. The general idea behind LNS is that instead of storing a real number as a combination of an integer exponent and a fixed-point number, only the *logarithm* $\log_2(A) = E_A$ is stored as a fixed-point *exponent*. In general purpose applications, LNS-numbers are then encoded as follows: $A = -1^{S_A} \times 2^{E_A}$ and a flag is used for zero values [6, 14].

Due to the logarithmic nature of the encoding, all calculations are performed in a *logarithmic scale*. Thus, logarithmic properties apply and $\log_2(a \times b) = \log_2(a) + \log_2(b)$, greatly simplifying multiplicative calculations.

In contrast to this, additive arithmetic operations become more complex. Assuming that $x > y$ holds, addition and subtraction are given by $\log_2(x \pm y) = \log_2(x) + \log_2(1 \pm 2^{(\log_2(y) - \log_2(x))})$. The second part of the equation is usually implemented through a helper function h, and the allowed interpolation error determines how this function is implemented in hardware. In this work, we adapt the approach

from [29], which was optimized for SPNs and uses a quadratic spline interpolation for h.

9.4 DESIGN-SPACE EXPLORATION USING SOFTWARE EMULATION

A fair comparison of the three arithmetic formats considered requires that the individual parameters of the different formats (e.g. overall bitwidth) are *optimized* as much as possible.

To this end, prior work such as [25] has often used theoretical worst-case analyses based on error-models for fixed- and floating-point arithmetic operators. However, these analyses tend to *overestimate* the error that occurs during actual computation. Besides that, error analysis models for Posit and LNS are not readily available and many of the application-specific optimizations to the hardware operator implementations described in Section 9.5 cannot easily be modelled in such error models.

Therefore, we take a different approach: Using a C++-based software emulation of the individual SPN and the different arithmetic formats, the design-space is traversed to determine the best *viable* configuration for each arithmetic format on a per-benchmark basis. We use the available benchmark data to run the software emulation with each configuration and only accept a configuration, if it maintains a given error-threshold. As the representativeness of the training data is key to the ML training itself, the design-space exploration will yield configurations that work for all relevant input combinations. This approach is also common when quantizing neural networks [13].

9.4.1 *Implementation*

Using a graph-based intermediate representation and an abstract syntax tree (AST) infrastructure, we generate C++ code emulating the behavior of each of the different arithmetic formats in hardware as closely as possible.

The design-space of possible configurations is then automatically traversed. For each configuration, we generate and compile the C++ code and run the SPN inference on a CPU. If the maximum error does not exceed the configurable error threshold, we accept the configuration.

The performance of the DSE can be improved significantly by investigating multiple configurations in parallel and additionally parallelizing the CPU-based execution using OpenMP. This way, we could reduce the time required to determine the correct floating-point configuration for the largest benchmark instance *NIPS80* from 656 seconds to only 138 seconds.

Table 9.1: Configuration of the three arithmetic formats for each of the benchmarks maintaining an error of 1×10^{-6}.

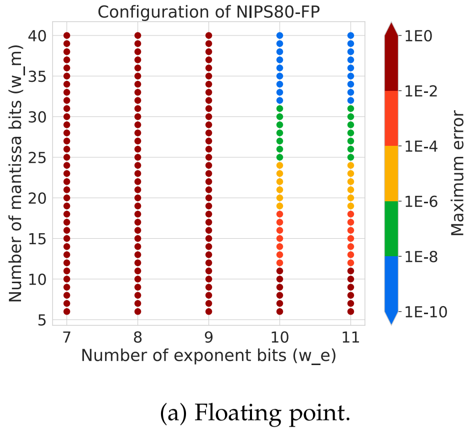
	FP		Posit		LNS		
Benchmark	w_e	w_m	w	w_{es}	w_I	w_F	h-Error
Accidents	8	26	36	4	7	32	21.5
Audio	9	28	36	4	8	30	20.5
MSNBC 200	8	26	32	4	7	31	19.5
MSNBC 300	8	24	32	4	7	31	20.5
Netflix	9	26	36	4	8	30	20.5
NLTCS	7	26	32	3	6	30	19.5
Plants	8	28	36	5	7	31	20.5
NIPS5	7	24	30	3	5	26	18.5
NIPS10	7	24	32	3	6	27	20.0
NIPS20	8	24	34	3	7	29	19.5
NIPS30	8	26	34	4	7	29	19.5
NIPS40	9	26	34	4	7	30	19.5
NIPS50	9	26	34	5	8	30	19.5
NIPS60	9	26	36	5	8	30	19.5
NIPS70	9	26	36	5	8	30	20.5
NIPS80	10	26	36	5	9	31	19.5

9.4.2 *Accuracy Results*

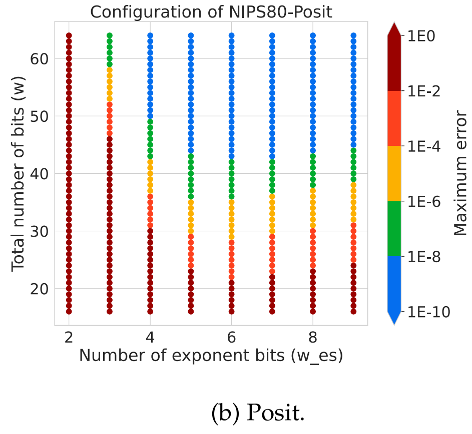
For the following accuracy evaluation, an error threshold of 1×10^{-6} was used. Note that we compute the error in log-space to determine the error independently from the magnitude of the values. For each of the arithmetic formats, different parameters can be chosen: For floating point, the number of bits in the mantissa (w_m) and the exponent (w_e) can be configured. The Posit format is parameterized by the total number of bits (w) and the number of bits used for the exponent (w_{es}). The LNS format can be configured by three parameters: The number of integer (w_I) and fraction (w_F) bits in the fixed-point format of the exponent and the maximum error allowed for the interpolation (Error) of the helper function h used in LNS-addition.

The configurations identified through our design-space exploration for each benchmark can be found in Table 9.1. The plots in Fig. 9.3 show how the maximum error develops across different configurations for each arithmetic format in the NIPS80 benchmark, the largest instance in our benchmark set.

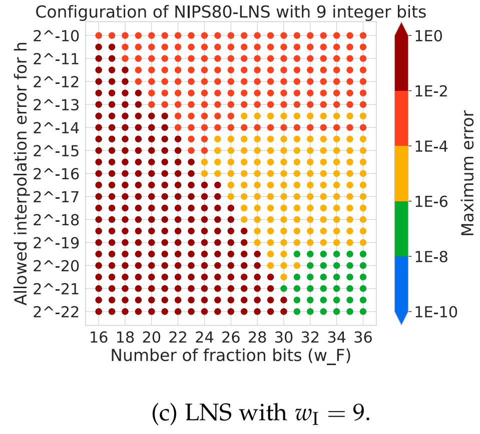
For floating point, a minimum number of exponent bits (w_e) is required to be able to represent small but significant values in the



(a) Floating point.



(b) Posit.



(c) LNS with $w_I = 9$.

Figure 9.3: Development of the maximum error depending on the configuration of the arithmetic formats. Best viewed in color.

Table 9.2: Comparison of the per-operator resource requirements and pipeline depth, using configurations for *NIPS80* (cf. Table 9.1).

Format	Op.	Slice	DSP	BRAM	pipeline depth
FP	Adder	106	0	0	5
	Mult.	86	2	0	5
Posit	Adder	374	0	0	7
	Mult.	340	4	0	12
LNS	Adder	757	20	1.5	64
	Mult.	36	0	0	3

first place. Beyond that, a certain number of mantissa bits (w_m) is required to represent numbers sufficiently accurate so the error will not accumulate beyond the error threshold.

With Posit, a minimum number of bits for the exponent (w_{es}) and the total size of the format (w) is required. However, if the size of the exponent is increased *beyond* that minimum number, the total number of bits *also* has to be increased, otherwise the number of bits remaining for the mantissa (max. $w - w_{es} - 3$) is no longer sufficient. So for Posit, the sweet spot is reached when w_{es} is just large enough to encode all relevant exponents.

The direct comparison of floating-point and Posit shows, that the total bitwidth of the formats is typically relatively close. This result aligns with the findings in [8]. The probabilistic values computed inside the SPN tree are very small, and lie outside of the *golden range* identified in [8]. In that range, relatively small Posit formats can be used to replace significantly larger floating-point formats.

The LNS format will only produce correct results, if the number of integer bits (w_I) is sufficiently large to represent all relevant exponents, therefore the plot in Fig. 9.3c shows the development of the error depending on the fraction bits (w_F) of the exponent and the interpolation error of the addition helper function h for $w_I = 9$. The number of fraction bits must be sufficiently large to represent numbers with a certain accuracy and, at the same time, the allowed interpolation error of h must be sufficiently small so the LNS addition does not introduce excessive error.

9.5 IMPLEMENTATION OF HARDWARE ARITHMETIC OPERATORS

Based on the findings from the automatic DSE presented in the previous section, specialized hardware arithmetic operators for SPN inference were developed. This section details the implementation for each arithmetic format. An overview of the resource requirements of the

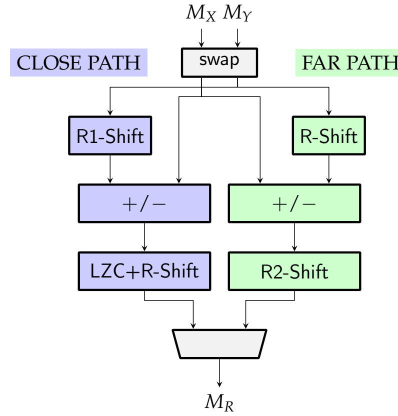


Figure 9.4: FP Adder dual path mantissa processing

individual operators can be found in Table 9.2. The operators were designed as drop-in replacement for the operators in [26] to enable reuse of the automatic toolflow in this work.

9.5.1 *Floating Point*

The floating point implementations used in this work are based on the FloPoCo tool [9] which was extended for the specifics of SPN. All of our extensions have been made publicly available in the FloPoCo git repository [7]. Note that subnormal numbers are not supported in FloPoCo as they are very costly to implement and the loss in dynamic range can be easily compensated by adding one additional mantissa bit.

9.5.1.1 *Floating Point Adder*

Addition in FP is a much more time and resource consuming operation compared to FP multiplication. The basic algorithm to perform a floating point addition requires the following computation steps: 1) computing the exponent difference, 2) alignment of the operands, 3) mantissa addition, 4) alignment and rounding of the result, and, 5) handling of special values. All these computations lie on the critical path where the large bit shifters required for the two alignment steps are among the most demanding. Also, faithful rounding does not help much for addition. However, a well-known technique to reduce the delay is the dual-path (DP) architecture [11]. The observation here is that two cases exist that can be treated separately: 1) when subtracting two numbers with similar magnitude, only a *small* operand shift is necessary while a *full* result shifter is required; 2) in all other cases, the operand shift has to be *large* while a *small* shift for the result is sufficient. In the DP adder, the computations for both cases are computed *in parallel*, and the correct result is selected at the end.

Table 9.3: Direct comparison of the FP adder before and after their optimization using the configurations for *NIPS80* (cf. Table 9.1).

Operator	Slice	DSP	PD	Freq. [MHz]
FP Adder Single Path	138	0	8	384
FP Adder Dual Path	184	0	7	274
FP Adder (only pos. args.)	106	0	5	389

Fig. 9.4 shows the data path for processing the mantissa, omitting the control signals for brevity. The first case is called the *close-path* (shown on the left in blue) and the second case the *far-path* (shown on the right in green). While for the *operand* alignment only a 1-bit right shift (R1-Shift) is necessary in the close-path, a full right shifter (R-Shift) is necessary in the far-path. In contrast, the *result* of the close-path requires a leading zero counter (LZC) and full right shifter, while the the far-path only requires a 2-bit shift (R2-Shift) for normalization and rounding.

To implement SPNs, we can make use of the dual-path idea by exploiting the fact that all values in SPNs are restricted to be *positive* and only *additions* occur. Hence, the close-path in a dual-path architecture will *never* be active in an SPN. To this end, we extended the dual-path implementation of the `FPAdd` operator in FloPoCo with an option to optimize the adder only for positive numbers, which omits all components from the close-path as well as the output multiplexer.

To gauge the effects of this optimization, we performed a synthesis experiment on the single operators (using the same setup later described in Section 9.6.2). The results are given in Table 9.3, showing the logic resources, the pipeline depth (PD) as well as the max. clock frequency. As there are two options for the FP adder in FloPoCo, a single path and a dual path, we synthesized both. As expected, the dual path has one pipeline stage less compared to the single path, but at the expense of a larger chip area. Remarkably, our optimization for only positive operands (listed as "only pos. args.") leads to a slice reduction of 23.2% and 42.4% compared to the single and dual path options, respectively, while reducing the pipeline depth by 3 and 2 cycles at the same time.

9.5.1.2 *Floating Point Multiplier*

The computation of an FP multiplication is much simpler compared to addition: 1) the mantissas are multiplied, 2) the exponents are added, and, finally 3) the result is normalized and rounded. This normalization requires only a small shift by one bit position and can usually be merged with the output MUX that is necessary for the special values. Besides this, the rounding mode has the most influence on the used

resources. In contrast to correct rounding, faithful rounding requires only about half the number of bits plus some guard bits of the mantissa multiplication result [2]. Hence, a truncated integer multiplier can be used for the mantissa which requires less chip area. Therefore, the FP multipliers in this work use faithful rounding based on the work in [2].

9.5.2 *Logarithmic Number System*

For the implementation of the LNS hardware operators, we employ the implementation of Weber et al., presented in [29]. They developed pipelined and parameterized LNS adders and multipliers targeted towards SPNs.

As discussed earlier, multiplication in the logarithmic space can be implemented as a simple binary addition, and consequently consumes less than half (36 vs. 86, cf. Table 9.2) of the slices compared to the floating-point multiplier, and no DSPs.

On the other hand, the much more complex calculation for addition in logarithmic space, for which a quadratic spline interpolation was used in [29], results in a larger chip area for the logarithmic adder, which consumes 757 slices and 20 DSPs, compared to 106 slices and no DSPs for FP.

9.5.3 *Posit*

For the implementation of the Posit hardware operators, we build upon PACoGen [15], an open-source project providing Posit basic arithmetic operators. These implementations are generally only realized as combinatorial circuits.

To ensure a fair comparison between the arithmetic formats regarding operating frequency, we introduced pipelining into the existing, parameterized implementation. The resulting multiplication operator requires almost five times the logic resources (340 vs. 86 slices), and, even though we adopted the optimal DSP allocation scheme from [17], twice the number of DSPs (4 vs. 2), as the floating-point multiplier. In case of the addition, the additional decoding logic for the regime and the higher internal precision cause the Posit adder to use significantly more resources than its floating-point counterpart (374 vs. 106 slices, cf. Table 9.2).

9.6 EVALUATION

9.6.1 *Benchmarks*

In order to be able to compare the performance and FPGA resource usage directly to [26], we use the same set of benchmarks. The set

contains two kinds of benchmarks: Count-based examples, which are taken from the NeurIPS corpus [10] and capture information about the frequency of words in texts, and examples with binary input variables, which were pre-processed by [18] and [28] and capture statistical data, such as usage statistics of services. More detailed information on the individual benchmarks can be found in [26].

9.6.2 *FPGA Implementation Results*

We first compare the resource usage of the three different arithmetic formats for the benchmark set, using the configurations from Table 9.1. Xilinx Vivado 2019.1 and TaPaSCo 2019.10 (pre-release) are used to generate bitstreams for a Xilinx Virtex 7 FPGA device (xc7vx690), all numbers given here are taken from the post-place&route reports. We use the automatic design-space exploration feature of TaPaSCo [16] to determine the best possible frequency. All bitstreams are tested in actual hardware on a Xilinx VC709 development board, verifying that the configurations determined by our DSE (cf. Section 9.4) maintain the given error bound of 1×10^{-6}.

The FPGA implementation results are given in Table 9.4. For brevity, numbers are given relative to the entire FPGA, the absolute number of resources available are 108,300 (Slices), 1,470 (BRAM) and 3,600 (DSP), respectively.

Through our automatic design-space exploration to determine the minimum viable configuration and the optimization to the floating point operators described in Section 9.5.1, the resource usage compared to the results reported in [26], decreases by up to 57% in logic slices (avg. 38.5%) and up to 68% in DSP (avg. 62.9%). Additionally, the clock frequency increases by up to 75 MHz (avg. 46.6 MHz). The decrease in resource consumption is also depicted in Fig. 9.5.

The comparison between customized floating-point (CFP) and Posit shows that the latter requires significantly more logic (avg. +53%) and, except for benchmarks *Audio* and *Plants*, which contain a low number of adders in comparison to the number of multipliers, also twice the number of DSPs. The BRAM utilization is almost identical, the frequency is typically lower for Posit (avg. 30 MHz less) and the pipelines are notably deeper. Overall, one can conclude that Posit is less suitable for SPN inference than floating-point, probably because the numbers involved in SPN inference lie outside of the *golden range* (cf. Section 9.4.2), where Posit could make up for the additional decoding logic by using much narrower bitwidths. However, the Posit-based arithmetic still outperforms the double-precision arithmetic used in [26] by up to 22.6% in slices (avg. 9.5%) and 36% in DSP (avg. 34.2%).

When compared with floating-point, LNS requires slightly more slices (avg. +7.57%) and significantly (avg. +56%) more BRAM, which, however, is not a critical resource in our case. The frequencies are

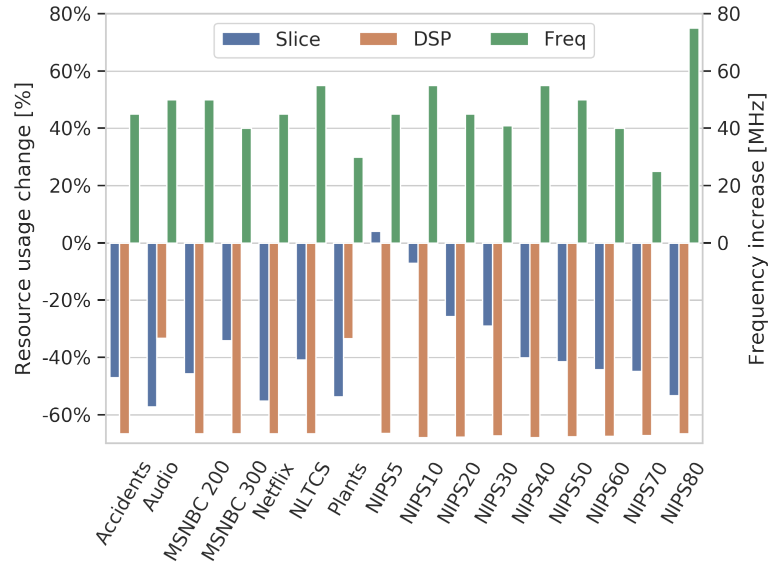


Figure 9.5: Improved resource and maximum frequency for floating-point arithmetic in comparison with prior work [26].

comparable, with winners in both formats. The pipelines are much deeper, mainly due to the long latency (64 cycles) of the LNS adder. The DSP usage comparison between floating-point and LNS is highly dependent on the multiplier/adder-ratio (given as *M/A* in Table 9.4) of the examples. Only if there are roughly nine times more multipliers than adders, LNS outperforms floating-point with regard to the DSP usage (NIPS10 is an outlier, probably due to the very low DSP usage in both formats). Overall, it seems that LNS is only suitable for such SPNs with a much higher number of multipliers than adders. Yet, the LNS-based arithmetic is able to outperform the FloPoCo double-arithmetic results from [26] by up to 57.5% in slices (avg. 33.7%) and 86% in DSP (avg. 66%), in particular for examples with only a few adders.

To further validate our results, we also tested relaxed error conditions, namely 1×10^{-4} and 1×10^{-2}, for benchmarks *Accidents* and *Audio*, which were chosen because of their very different adder/multiplier-ratio. We have to omit detailed results for brevity here, but overall, the relation between LNS- and floating-point format found in the evaluation for 1×10^{-6} *persists* for relaxed error conditions: LNS is only able to save resources in comparison to floating-point, if the SPN contains very few adders compared to the number of multipliers.

9.6.3 *Power Evaluation*

Next to the required chip area, we are also interested in the impact of the arithmetic format onto power consumption.

Table 9.4: FPGA implementation results for all benchmarks, using the configurations from Table 9.1. Best values bold.

		Slices [%]			DSP [%]			BRAM [%]			Frequency [MHz]			Pipeline Depth		
Benchmark	**M/A**	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS	CFP	Posit	LNS
Accidents	8.0	**40.19**	70.81	41.31	**12.06**	24.11	15.00	**3.71**	**3.71**	7.52	**245**	200	222	**73**	161	169
Audio	22.9	**38.22**	77.02	38.63	**30.56**	**30.56**	6.33	**3.71**	**3.71**	5.75	**250**	200	210	**73**	161	169
MSNBC 200	5.5	**35.82**	53.50	41.69	**9.17**	18.33	15.83	**3.71**	**3.71**	6.77	**250**	215	245	**143**	299	577
MSNBC 300	6.0	**30.61**	43.59	38.35	**5.67**	11.33	8.97	**3.71**	**3.71**	6.77	240	225	**255**	**89**	213	431
Netflix	21.0	**39.30**	67.98	37.33	12.83	25.67	**5.81**	**3.71**	**3.71**	5.75	**235**	215	220	**68**	149	166
NLTCS	5.6	**36.64**	51.38	40.51	**8.44**	16.89	13.50	**3.71**	**3.71**	6.46	255	220	**270**	**98**	206	342
Plants	18.3	**40.14**	74.78	41.17	28.44	28.44	**7.39**	**3.71**	**3.71**	6.09	230	205	**250**	**128**	278	385
NIPS5	10.0	**25.11**	26.59	26.17	0.56	1.11	**0.44**	3.74	**3.71**	3.84	**245**	240	240	**24**	58	60
NIPS10	8.3	**27.44**	30.71	28.09	1.39	2.78	**1.33**	3.74	**3.71**	4.18	**255**	240	**255**	**42**	96	182
NIPS20	8.0	**28.93**	37.88	31.28	**3.11**	6.22	3.50	**3.81**	4.01	4.69	245	200	**270**	**46**	108	203
NIPS30	8.7	**32.85**	44.87	35.43	**4.83**	9.67	5.00	**3.74**	4.01	4.93	240	220	**250**	**63**	132	209
NIPS40	7.6	**34.48**	50.35	39.42	**6.78**	13.56	7.56	**3.91**	4.08	5.95	255	215	**265**	**63**	132	224
NIPS50	8.9	**36.86**	54.99	42.42	**7.94**	15.89	8.44	**3.98**	4.15	6.09	**250**	215	245	**68**	144	227
NIPS60	12.0	**38.00**	59.91	40.34	8.67	17.33	**6.86**	**4.32**	4.46	6.19	**240**	215	235	**63**	132	224
NIPS70	12.9	**41.75**	67.86	43.12	10.00	20.00	**7.39**	**4.05**	4.35	6.97	**225**	203	210	**73**	156	230
NIPS80	8.3	**44.83**	82.84	47.52	**14.72**	29.44	20.44	**3.98**	4.63	7.72	**255**	195	230	**83**	211	306

Number of LUTs, FFs, BRAM slices and DSP blocks are given as percentage for brevity. The total number of available resources are 433200, 866400, 1470 and 3600 respectively.

Table 9.5: Power consumption of the datapath.

	Power Consumption [Watt]			
Benchmark	[26]	CFP	Posit	LNS
Accidents	12.493	3.069	5.267	3.721
Audio	18.427	5.518	8.358	2.818

In order to investigate the power consumption of the different arithmetic formats, we consider only the datapath itself, leaving out the memory infrastructure and TaPaSCo platform infrastructure. We again run synthesis and P&R for the Xilinx VC709 board using Vivado 2019.1. Afterwards, we use Mentor Questasim 2019.2 to run a *post-implementation timing simulation* to capture signal activity information from a run with actual inference input data. Using this activity information, we then use the Vivado 2019.1 power analysis for an estimate of the power consumption of the datapath.

As the post-implementation timing simulation can take several days for larger circuits, we again limit our investigation to the two benchmark instances *Accidents* and *Audio*, that we selected for the reasons described in the previous section. In addition to the three arithmetic formats investigated in this work, we also conduct the measurement for the double-precision FloPoCo-format from prior work [26].

The results from the power analysis (Table 9.5) align with our findings for the chip area in the previous sections: In the benchmark instance *Accidents*, where the customized floating-point was the most area-efficient format, it also requires the least power, followed by LNS. For the benchmark instance *Audio*, where LNS was the most area-efficient format due to the low number of adders in the SPN, LNS also requires the least power. Just as before, Posit is not able to keep up with the two other formats with regard to power usage.

Compared to the double-precision format from prior work, the SPN-optimized arithmetic formats developed in this work are able to save significant amounts of power.

9.6.4 *Performance Evaluation*

In this section, we evaluate the performance of three arithmetic formats implemented on the FPGA and compare it to a CPU and GPU-based implementation of SPN inference.

9.6.4.1 *CPU & GPU Baseline*

Based on the compiler infrastructure that we created for the design-space exploration (cf. Section 9.4), we additionally built a custom

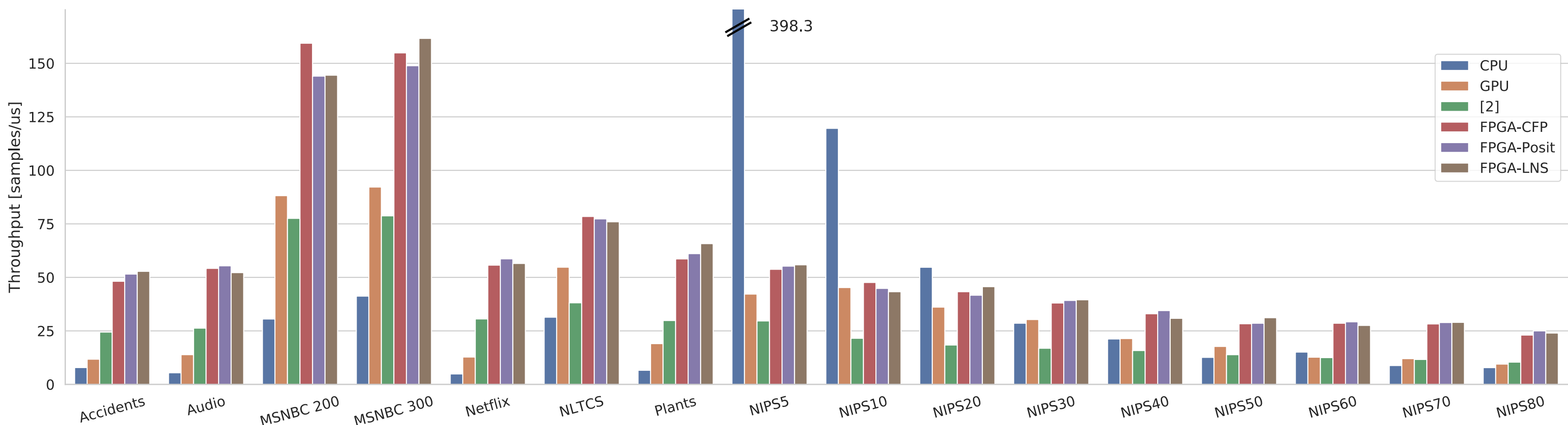


Figure 9.6: Throughput of the CPU, GPU and FPGA-implementations in samples/μs. Each group represents an example SPN. The single outlier is the CPU throughput for example NIPS5 which amounts to 398.8 samples/μs.

compilation flow mapping an SPN description to optimized C++ and CUDA-code, both using double-precision floating-point arithmetic. In both cases, we compiled using `-O3` and `-ffast-math` to enable aggressive compiler optimizations. Our C++ compilation flow on an AMD Ryzen 1600X performs on par with the CPU-baseline from [26], and our CUDA compilation flow is able to *outperform* the original Tensorflow-based GPU-mapping from [26] by a factor of up to 90x on a Nvidia 1080Ti GPU.

9.6.4.2 *Performance Comparison*

For the comparison, we run the inference on the VC709 development board, coupled with an AMD Ryzen 1600X. Our measurements of the throughput in Fig. 9.6 also include the time required to *transfer* the data between host and FPGA.

For the three smallest count-based samples (NIPS5-20), the CPU provides the best throughput. For these small networks the overhead for data-transfer to the accelerator (GPU or FPGA) clearly dominates the execution time. With our optimized CUDA compilation flow, the GPU provides better throughput than the CPU for the remaining benchmarks, in particular for the binary examples.

Despite the large differences in the pipeline-depth (cf. Table 9.4), the performance for the three arithmetic formats implemented on the FPGA varies only slightly. Overall, all three versions deliver very similar performance (with an overall difference of less than 2%). Compared to the previous FPGA implementation in [26], the new formats provide better throughput (geo.-mean. 2.1x speedup). This is partly due to the higher operator frequencies, but also caused by improvements to the underlying TaPaSCo framework.

All three formats significantly *outperform* the CPU. Except for the three benchmarks mentioned earlier, the speedup reaches as high as factor 12x (geo.-mean 2.5x). The three FPGA versions also provide significantly *higher throughput* than the GPU-based implementation, here, the speedups reach up to 4.6x (geo.-mean. 2.1x).

Again, note that our measurements include the PCIe data-transfer to the FPGA memory. On shared-memory systems such as Zynq MPSoC, the speedup over the CPU and the GPU would reach up to 37x and 14x, respectively.

9.7 CONCLUSION & OUTLOOK

In this work, we have investigated three different arithmetic formats with regard to their suitability for Sum-Product Network Inference on FPGAs. We have developed an automatic design-space exploration framework, which allows us to efficiently identify the minimum bitwidth required for each of the formats to maintain a given error margin. Based on the findings from the DSE, hardware arithmetic

operators, optimized for SPN inference, for each of the formats were implemented.

Our evaluation shows that customized floating-point is the most resource-efficient format for SPN inference, and is only outperformed by a logarithmic number system format for SPNs with *very few* adders compared to the number of multipliers. All three investigated arithmetic formats deliver almost identical performance and significantly outperform CPU and GPU-based implementations of SPN inference, by factors up to 12x and 4.6x, respectively.

In future work, we will investigate how the hardware arithmetic operators can be optimized further, e.g., by using fused operators.

ACKNOWLEDGEMENTS

The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software. Calculations for this research were conducted on the Lichtenberg high performance computer of TU Darmstadt.

Finally, the authors would like to thank Kristian Kersting and Alejandro Molina, for much appreciated discussions of the subject and insights into the matter of Sum-Product Networks.

Lukas Sommer and Lukas Weber contributed equally to this work.

REFERENCES

[1] JD Alves, JF Ferreira, J Lobo, and J Dias. "Brief survey on computational solutions for Bayesian inference." In: *Unconventional comp. for Bayesian inference*. 2015.

[2] Sebastian Banescu, Florent de Dinechin, Bogdan Pasca, and Radu Tudoran. "Multipliers for Floating-Point Double Precision and Beyond on FPGAs." In: *SIGARCH Computer Architecture News* 38.4 (Sept. 2010), pp. 73–79.

[3] Jessa Bekker, Jesse Davis, Arthur Choi, Adnan Darwiche, and Guy Van den Broeck. "Tractable Learning for Complex Probability Queries." In: *NIPS*. 2015.

[4] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers. "Parameterized Posit Arithmetic Hardware Generator." In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. Oct. 2018.

[5] Jungwook Choi and Rob A. Rutenbar. "Video-Rate Stereo Matching Using Markov Random Field TRW-S Inference on a Hybrid CPU+FPGA Computing Platform." In: *IEEE Trans. Circuits Syst. Video Techn.* (2016).

[6] J. Detrey and F. de Dinechin. “A VHDL library of LNS operators.” In: *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003*. Nov. 2003.

[7] Florent de Dinechin. *FloPoCo Project Website*. Website move from its orginal location at http://flopoco.gforge.inria.fr/. URL: http://flopoco.org (visited on 03/26/2023).

[8] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. “Posits: The Good, the Bad and the Ugly.” In: *Proceedings of the Conference for Next Generation Arithmetic 2019*. CoNGA’19. New York, NY, USA: ACM, 2019, 6:1–6:10. ISBN: 978-1-4503-7139-1. (Visited on 07/04/2019).

[9] Florent de Dinechin and Bogdan Pasca. “Designing Custom Arithmetic Data Paths with FloPoCo.” In: *IEEE Design & Test of Computers* 28.4 (2011), pp. 18–27.

[10] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[11] Paul Michael Farmwald. “On the design of high performance digital arithmetic units.” PhD thesis. Stanford University, 1981.

[12] John L. Gustafson and Isaac T. Yonemoto. “Beating Floating Point at its Own Game: Posit Arithmetic.” In: 4 (Apr. 2017). ISSN: 2313-8734.

[13] Song Han, Huizi Mao, and William J Dally. “Deep Compression: Compressing Neep Neural Networks with Pruning, Trained Quantization and Huffman coding.” In: (2015).

[14] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. S. Hemmert. “A comparison of floating point and logarithmic number systems for FPGAs.” In: *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’05)*. Apr. 2005, pp. 181–190. DOI: 10.1109/FCCM.2005.6.

[15] M. K. Jaiswal and H. K. H. So. “PACoGen: A Hardware Posit Arithmetic Core Generator.” In: *IEEE Access* 7 (2019), pp. 74586–74601. ISSN: 2169-3536.

[16] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. “The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems.” In: *Applied Reconfig. Comp.* 2019.

[17] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf. “Resource Optimal Design of Large Multipliers for FPGAs.” In: *2017 24th Symp. on Computer Arithmetic*. July 2017.

[18] Daniel Lowd and Jesse Davis. “Learning Markov network structure with decision trees.” In: *Data Mining (ICDM), 2010 IEEE 10th International Conf.* 2010.

[19] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Floriana Esposito, Siraam Natarajan, and Kristian Kersting. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2018.

[20] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. "On Theoretical Properties of Sum-Product Networks." In: *Proc. of AISTATS*. 2015.

[21] A. Podobas and S. Matsuoka. "Hardware Implementation of POSITs and Their Application in FPGAs." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018.

[22] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: a New Deep Architecture." In: *Proc. of UAI* (2011).

[23] Andrzej Pronobis, Francesco Riccio, and Rajesh PN Rao. "Deep spatial affordance hierarchy: Spatial knowledge representation for planning in large-scale environments." In: *ICAPS 2017 Workshop on Planning and Robotics*. 2017.

[24] Martin Ratajczak, Sebastian Tschiatschek, and Franz Pernkopf. "Sum-Product Networks for Sequence Labeling." In: (2018). arXiv: 1807.02324.

[25] Nimish Shah, Laura I. Galindez Olascoaga, Wannes Meert, and Marian Verhelst. "ProbLP: A Framework for Low-precision Probabilistic Inference." In: *56th Annual Design Automation Conference*. DAC '19. Las Vegas, NV, USA: ACM, 2019. ISBN: 978-1-4503-6725-7.

[26] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *36th Intl. Conf. on Computer Design (ICCD)*. Oct. 2018, pp. 350–357.

[27] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Synthesis of FPGA-based Accelerators for the Sum-Product Network Inference Problem." In: *ICML 2018 Workshop on Tractable Probabilistic Models (TPM)*. 2018.

[28] Jan Van Haaren and Jesse Davis. "Markov Network Structure Learning: A Randomized Feature Generation Approach." In: *AAAI*. 2012, pp. 1148–1154.

[29] Lukas Weber, Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting, and Andreas Koch. "Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs." In: *International Conference on Field-Programmable Technology (FPT)*. 2019.

[30] Han Zhao, Mazen Melibari, and Pascal Poupart. "On the Relationship between Sum-Product Networks and Bayesian Networks." In: *Proc. of ICML*. 2015.

Copyright Notice: © 2020 IEEE. Reprinted, with permission, from Lukas Sommer et al. "Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs." In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020, pp. 75–83. DOI: 10.1109/FCCM48280.2020.00020.

10 OPTIMIZING A HARDWARE NETWORK STACK TO REALIZE AN IN-NETWORK ML INFERENCE APPLICATION

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Optimizing a Hardware Network Stack to Realize an In-Network ML Inference Application* by *Marco Hartmann, Lukas Weber, Johannes Wirth, Lukas Sommer, Andreas Koch* in *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. The contribution of the author of this thesis is summarized as follows.

» *As a co-leading author, Lukas Max Weber contributed to the adapted tool flow for the automatic generation of Sum-Product Network inference accelerators. His contribution includes the changes to the architecture to enable streaming-based access and logic to enable correct addressing of result packages. The system architecture and integration with the network stack were contributed by Marco Hartmann, who also conducted the performance evaluation. A design-space exploration for the degree of replication was performed by Lukas Max Weber. The manuscript was joint work by Lukas Max Weber and Marco Hartmann, with support and feedback from the other co-authors.* «

ABSTRACT

FPGAs are an interesting platform for the implementation of network-attached accelerators, either in the form of smart network interface cards or as In-Network Processing accelerators.

Both application scenarios require a high-throughput hardware network stack. In this work, we integrate such a stack into the open-source TaPaSCo framework and implement a library of easy-to-use design primitives for network functionality in modern HDLs. To further facilitate the development of network-attached FPGA accelerators, the library is complemented by a handy simulation framework.

In our evaluation, we demonstrate that the integrated and extended stack can operate at or close to the theoretical maximum, both for the stack itself as well as an network-attached machine learning inference appliance.

10.1 INTRODUCTION

Numerous previous works have demonstrated the huge potential for acceleration that can result from attaching FPGAs directly to the network. In such scenarios, FPGAs can not only be used to implement smart network interface cards (SmartNIC) [13] and accelerate the network protocol stack, but they can also be employed to additionally offload parts of the application itself, e.g., machine learning inference in the case of Microsoft's Brainwave project [10], or network security [19].

While these approaches already demonstrate significant speedup, even more potential can be unlocked by moving the computation *into* the network, as so-called *In-Network Processing* (INP) [16, 26, 36]. This does not only allow moving computation closer to the origin of the data, but also facilitates distributed processing across the network.

However, with the network hardware available today, In-Network Processing is still severely limited. On the one hand, platforms such as Barefoot's Tofino provide high performance, but are limited with regard to programmability and the available memory on the device [41]. On the other hand, platforms providing full programmability, e.g., through Micro-C, provide only limited performance. FPGAs with high-speed network interfaces can provide both, high flexibility for the design *and* high performance.

Independent of whether FPGAs should be used as INP accelerator or SmartNIC, the availability of a high-throughput hardware network stack is crucial for successful deployment, and such a stack was presented by Ruiz et al. in [24].

In order to make this particular stack more accessible to researchers and designers, and to facilitate and automate the design of network-attached, FPGA-based accelerators, we integrate this stack with the open-source [32] TaPaSCo framework. This integration not only makes the network stack available in highly complex heterogeneous System-on-Chip (SoC) designs, but also allows using TaPaSCo's automatic design-space exploration for automatic and efficient traversal of large design spaces for network-attached accelerators.

After providing details on some modifications to the original stack (Section 10.4.1), making the stack more flexible, and describing the integration with TaPaSCo (Section 10.4.2), we present a library of easy-to-use design primitives (Section 10.4.3) that allow accelerators to communicate with the network stack on multiple protocol levels. As testing and verification through simulation are important steps of any design process, we also present a fast hardware/software-co-simulation for network-attached accelerators (Section 10.4.5).

To demonstrate how the design primitives and hardware/software-co-simulation facilitate the design of accelerators, we present a case study, transforming a host-attached FPGA-based accelerator for Sum-

Product Network inference [29] into a network-attached accelerator (Section 10.6). Before that, we evaluate the raw performance of the network stack and various configurations in Section 10.5.

We also provide related work in Section 10.2, necessary background information in Section 10.3, and a conclusion and outlook to future development in Section 10.7.

10.2 RELATED WORK

FPGA-based network stacks are well established in the academic and commercial domain with several implementations that provide differing feature sets, connection speeds, and latencies. Compared to traditional software stacks, most of them lack more complex features like IP segmentation and only support particular default configurations without optional protocol extensions.

The availability of commercial TCP Offload Engines (TOE) that are capable of 100 Gigabit (100G) is still limited. One of them is developed by the Fraunhofer Heinrich-Hertz-Institute and Missing Link Electronics [35], which supports a single TCP session and can typically implement send and receive buffers in BRAM due to their low memory space requirements.

Most commercial TOEs support sub-100G connection speeds and are optimized for low latency [1, 2, 12, 34]. A typical application field is High-Frequency Trading (HFT), where any reduction in latency may increase the profitability of a financial trading algorithm.

Notably, ultra-low latency stacks generally tend to support fewer concurrent sessions, because FPGAs only offer a limited amount of low-latency on-chip memory, which is required for the per-connection TCP buffers.

In addition, there are various FPGA-based network stacks pursuing different design goals in the academic space: For example, the authors in [42] describe a hybrid approach where only the most data-intensive parts of the TOE are implemented in hardware, while the more control-intensive parts are handled by firmware running on a CPU.

The work in [17] presents a TOE that can achieve 4 Gbps of throughput from up to 2048 receive sessions, and 40 Gbps of throughput to up to 20480 transmit sessions, targeting asymmetric workloads such as video on demand. It does not support jumbo frames, and the maximum segment size (MSS) is fixed to 1460 bytes. The memory architecture uses external SRAM for storing per-session state information and DRAM for the send and receive buffers.

The authors of a 100G-capable intrusion-prevention system in [43] observed that a per-session receive buffer with a fixed size is often unnecessary, since only 0.3% of network packets arrive out of order and require buffering for reordering. A more memory-dense buffer architecture based on linked lists and dynamic allocation that supports

as many as 100k concurrent sessions while still fitting into BRAM is developed. Packets that arrive in order are handled on a constant-time *fast path* without being buffered, whereas out-of-order packets are handled on a *slow path* that requires non-deterministic amounts of time.

A 10G-capable TCP/UDP network stack is presented in [27] which was designed to scale well in the number of sessions, verifiably achieving 10k open connections, however, at the cost of increased latencies. Unlike most published TOEs, this project handles the complexities of the TCP protocol through the use of high-level synthesis (HLS). By designing the stack in C++, the entire implementation comprises less than 8k lines of code and is thus significantly more compact than comparable implementations in HDL.

Several other research projects are based on this stack: For example IBM's *cloudFPGA* [40], which seeks to deploy large-scale datacenter applications on network-attached FPGAs, or the HLS-based HFT application in [7], which builds on top of the UDP stack and achieves a round-trip latency of 869 ns.

The *Limago* network stack published in [24] used the 10G stack from [27] as a basis, which was then upgraded for 100G support. The authors added several new features in order to achieve this higher link speed, such as TCP window scaling and a hash table based on cuckoo-hashing, which replaces the previous slower session lookup mechanism. During this upgrade process, methods to improve checksum calculation in hardware were investigated in [31]. The authors concluded that an HLS-based approach does not perform satisfactorily and consequently developed a solution in VHDL.

Most of *Limago's* upgrades are also part of the 100G TOE published by ETH Zurich in [38], which is the successor to the 10G stack in [27]. It is also the hardware network stack implementation used in this work.

10.3 BACKGROUND

This section presents necessary background information on the employed TCP/IP stack and the TaPaSCo framework.

10.3.1 *Hardware TCP/IP Stack*

The design of a 100G hardware TCP/IP stack is a significant undertaking that is off the scope of this paper. Instead, an existing TCP/IP stack published by the Systems Group at ETH Zurich is leveraged. An early 10G-capable variant of the stack is presented in [27], which has been upgraded for 100G support in [24]. The TCP/IP stack sets itself apart from other hardware TCP/IP stacks reviewed in Section 10.2 in several ways.

Except for few SystemVerilog-based wrapper modules, the entire project is implemented in C++ using Vivado HLS [39]. Designing the TCP/IP stack in a high-level language sacrifices some control over the resulting circuity but also increases productivity, in particular with regards to the highly control-intensive TCP protocol implementation. In addition, an HLS-based design methodology comes at the advantage of an easily maintainable and extensible codebase.

The TCP/IP stack is designed to support a large number of TCP sessions, which is demonstrated in experiments in [27] with 10,000 concurrent connections. Each session requires unique send and receive buffers, where the size of a single buffer ranges from 64 KiB to 256 KiB, depending on the TCP *window scaling* configuration. For 10,000 sessions, this yields a total required buffer size between 1.3 GB and 5.2 GB, which can only be implemented in off-chip memory like DRAM. This is a deliberate tradeoff that chooses a higher number of concurrent sessions at the cost of increased latency [27, p.42].

Per-session information, such as the connection status or timer values of the retransmission mechanism, is kept in BRAM-based tables indexed by a session ID. The session IDs are stored in a hash table, which handles collisions and realizes single-cycle lookups and deletions [24].

Hardware designs that use the TCP/IP stack are split into three AXI-Stream-interconnected Xilinx Vitis kernels: the *user kernel* containing the user logic of a network application, the *network kernel* containing the TCP/IP stack itself, and the *CMAC kernel* containing the Ethernet subsystem and physical layer implementation. The *network kernel* internally consists of multiple HLS-based IP cores that are wrapped by a SystemVerilog top module. The host software uses an OpenCL API provided by the Xilinx Runtime "XRT" to interact with the FPGA design.

10.3.2 *The TaPaSCo Framework*

The *Task Parallel System Composer* (TaPaSCo) [18] is an open-source framework providing a toolflow for the automated generation of System-on-Chip FPGA designs with a particular focus on task-parallel computation. TaPaSCo aims to increase the *portability* and *scalability* of FPGA designs.

TaPaSCo includes base FPGA designs, referred to as *platforms*, for multiple Xilinx FPGA families. The *platform* typically contains platform-specific implementations of the memory subsystem, interrupt subsystem, interface to the host CPU, distribution networks for clock and reset signals, and a *status core* containing descriptive information about the design.

The *platform* acts as a hardware abstraction layer and provides a standardized interface to the *architecture* component of TaPaSCo, which

itself is decoupled from the underlying FPGA technology. The *architecture* contains TaPaSCo *Processing Elements* (PE), which are application-specific compute kernels that can be implemented by the user either in an HDL or HLS based design flow. Due to the separation between *platform* and *architecture*, a PE needs to be designed only once and can be used across all FPGAs supported by TaPaSCo without any changes, which increases a design's portability.

Different types of PEs can be instantiated in different multiplicities, yielding what is called a *composition*. TaPaSCo supports automated *Design Space Exploration* (DSE), which can assist in finding a throughput-optimal *composition*. These mechanisms allow to easily scale a design without changing the actual user logic of the PEs.

Further to the hardware toolflow, TaPaSCo provides a runtime with a C/C++ API that allows host software to interact with the FPGA design. In particular, the API provides functionality to schedule jobs onto PEs, handle data transfers between host and FPGA, and monitor the execution state of individual PEs.

TaPaSCo contains several plugins, referred to as *features*, which add optional functionality that is not available across all supported FPGAs but specific to a particular *platform*. The *Network* feature adds an Ethernet subsystem to the FPGA design and selectively connects PEs to it, effectively providing them with link-level access to a network. The feature is available on multiple TaPaSCo platforms in a 10G variant. On the Xilinx Alveo U280 and the BittWare XUP-VVH platform, the Network feature additionally supports the instantiation of a 100G Ethernet backend. Three different operating modes are supported by the Network feature: In *singular* mode, a single PE is attached to the Ethernet subsystem, whereas in both *broadcast* and *round-robin* mode, the subsystem is shared by multiple PEs. We will use this feature in *singular* mode to implement the CMAC portion of the network stack architecture that was outlined in Section 10.3.1.

10.4 IMPLEMENTATION

This section describes the contributions with the goal of providing assisting tools and libraries for developing networked applications on FPGAs with the TaPaSCo framework.

10.4.1 *Modification and Extension of the Network Stack*

In preparation of using the TCP/IP stack withing the TaPaSCo ecosystem, we replaced the host software of [38], which uses OpenCL and the Xilinx Runtime XRT, with an implementation that makes use of the TaPaSCo runtime.

10.4.1.1 *Parameterizable Data Width*

While the TCP/IP stack's individual HLS cores are designed with parameterizable bit width, several of the HDL modules instantiate fixed-width IP cores.

The TaPaSCo Network feature supports both 10G and 100G Ethernet subsystems that have data interfaces of different bit widths. With the objective to attach to both subsystems natively, we reworked the TCP/IP stack to be more flexible and allow build-time configuration of the data bit width.

10.4.1.2 *Issues with Buffer Memory Addresses*

The TCP/IP stack can bypass the TCP receive buffer and directly deliver received data to the application layer. This is an optional configuration aimed at reducing latency. By default, the bypass optimization is enabled, such that the receive buffer is not used. In the original release [38], both logical and arithmetical errors existed in the calculation of buffer memory addresses, leading to data corruption when buffer bypassing is disabled. Using the simulation infrastructure presented in Section 10.4.5, we were able to track down the issues and fix them.

10.4.2 *Integration with the TaPaSCo Framework*

With its Network feature, the TaPaSCo framework already supports the automated instantiation of an Ethernet subsystem within an FPGA design. We use this subsystem to complement those parts implemented by the TCP/IP stack into a full implementation of the internet protocol suite.

In preparation for protocol comparisons during the experimental evaluation, we extended the Network feature with support for the Xilinx *Aurora 64B/66B* [3] point-to-point link-layer protocol, which can now optionally be used instead of the Ethernet link-layer protocol. Both Ethernet and Aurora can use the same physical layer implementation.

Since the AXI-Stream data interfaces of both Xilinx CMAC [37], and Xilinx Aurora [4] IP are clocked at higher frequencies (approx. 322 MHz and 403 MHz, respectively) than the TCP/IP stack within the PE (250 MHz), a clock domain crossing is required on the data path between them. This is implemented by an AXI-Stream interconnect that is optionally instantiated by the TaPaSCo Network feature. To prevent a continuous transaction coming from a slow clock domain from being broken into multiple partial transfers within a faster clock domain, the interconnect is configured to *packet mode*, which buffers the entirety of a frame and forwards it in one piece. As the Xilinx CMAC contains only minimal internal buffering and implements *cut*

through semantics on both RX and TX data paths [37, p.11], disabling the *packet mode* may result in a buffer-underrun in the CMAC and a corrupted packet on the wire.

The 100G network bandwidth places high demands on the bandwidth of the RX and TX buffer memories. In our case, even when using the RX buffer bypass, a DRAM-based memory subsystem would limit the achievable network throughput. To avoid being bound by memory performance, we instead make use of the *High-Bandwidth Memory* (HBM) feature of TaPaSCo, which allows a PE to attach to high-bandwidth on-chip memory modules. In the resulting FPGA design, the DRAM-based memory subsystem for TCP buffers is replaced by an HBM-based subsystem. With this optimization, we achieve a full saturation of the link bandwidth and are not limited by memory bandwidth, as long as RX buffer bypassing is enabled (a detailed evaluation follows in Section 10.5.4).

10.4.3 *Design Primitives for Network Access*

The interface between the TCP/IP stack and the user kernel comprises 16 AXI-Stream interfaces. Of those, 4 are used for UDP-related functions and 12 for TCP-related functions. The source code release of the TCP/IP stack in [38] contains a basic C++ library that simplifies the design of HLS-based user kernels by appropriately interacting with the 16 AXI-Stream interfaces. This HLS toolflow is also available for the TaPaSCo integration, but finer-grained control over the generated circuity may be required for more sophisticated user kernels. This is generally achieved by designing the user kernel in a dedicated HDL. Because of its good integration with TaPaSCo, the Bluespec [8] language is chosen as the target HDL.

To facilitate the development of Bluespec-based user kernels, we implemented a novel library that exposes all functionality of the TCP/IP stack via an idiomatic Bluespec API. Like the C++ library, the Bluespec library on the backend attaches to the 16 AXI-Stream interfaces of the TCP/IP stack and appropriately interacts with them.

The bit width of any data-carrying AXI-Stream interface changes depending on the configured data width of the TCP/IP stack. However, the user-visible bit width of the Bluespec library API is not determined by the TCP/IP stack configuration but can be freely chosen by the user. The library contains custom AXI-Stream width-converters that adapt the bit width of the TCP/IP stack to the user-configured API bit width. As a result, user kernels are portable across 10G and 100G subsystems when using our Bluespec library.

The library's TCP support is fully featured and supports data transferring, the opening and closing of connections, and putting a TCP port into *listen* state. Whenever a new packet shall be transmitted by the user kernel, it must first be announced to the TCP/IP stack.

Data transmission can only start once the TCP/IP stack confirms the announcement with a status message. According to the authors, the duration of this handshaking sequence varies between 10 and 30 clock cycles [38], during which the stack must e.g. verify that the TCP send buffer has enough free capacity for the new packet. To maximize link utilization, the announcement of new packets and the transfer of data words belonging to an already announced packet are automatically pipelined by the library. Furthermore, the interface exposed to the user kernel is transfer-based rather than packet-based, meaning that the user can simply supply a stream of payload data which is then split into MSS-sized packets by the library automatically.

The user-facing UDP Bluespec interface is less complex than the one for TCP. It consists of methods for getting and putting data words and metadata of a new packet, where metadata includes a packet's length, the source and destination port, and the destination IP address.

The Bluespec library is available as free and open-source software [6].

10.4.4 *Creating a TCP/IP-capable Design*

This section describes how the individual parts, described in previous sections, are combined into a complete design that is capable of network communication via TCP/IP. A mapping between the seven layers of the *OSI Reference Model* [44] and the individual parts that implement those layers is provided below.

- Layers 1, 2 (partially: MAC sublayer) are implemented by the Ethernet subsystem instantiated by the TaPaSCo Network feature that includes modifications as described in Section 10.4.2. Depending on the TaPaSCo configuration, a 10G or a 100G backend is used.
- Layers 2 (partially), 3, 4 are implemented by the TCP/IP stack that includes modifications as described in Section 10.4.1. The data width of the stack matches that of the Ethernet backend.
- Layers 5, 6, 7 are implemented by the user kernel using the Bluespec library introduced in Section 10.4.3. Alternatively, HLS-based user kernels are also supported.

Fig. 10.1 shows a visualization of how a TCP/IP-capable TaPaSCo PE, consisting of a user kernel using the Bluespec library and a TCP/IP stack, integrates with the subsystems of a TaPaSCo design. The figure also shows a set of SPN accelerators attaching to the user kernel which are used in the case study in Section 10.6. In addition to the AXI-Lite control port, the PE module has two AXI-Full interfaces to the memory subsystem, and an RX and TX AXI-Stream interfaces to the TaPaSCo Ethernet subsystem. The dedicated memory subsystem is needed for

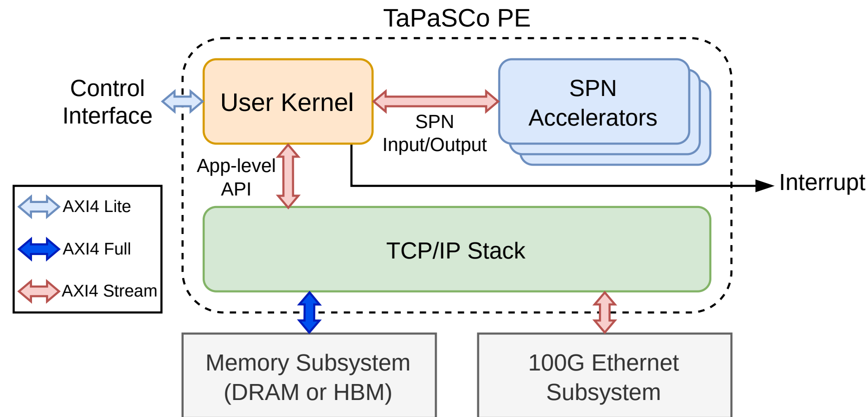


Figure 10.1: Architecture of a TCP/IP-capable PE with connections to TaPaSCo subsystems.

hosting TCP TX and RX buffers, which generally are too large to be implemented in internal memory such as BRAM (cf. Section 10.3.1).

When creating a TCP/IP-capable design for either the XUP-VVH or the AU280 board, which both use an FPGA of the Xilinx UltraScale+ family, special care has to be taken when placing the FPGA design. Internally, these particular FPGAs are not one monolithic chip, but are divided into three separate dies that are referred to as *super logic regions* (SLR). The SLRs are mounted on a silicon interposer and interconnected using a *Stacked Silicon Interconnect*. For designs where components are spread over different SLRs, timing closure may be hard to achieve because the connections between SLRs are limited and induce a higher-than-normal delay. For instance, the HBM ports are located in the bottommost SLR, whereas the GTY transceivers that attach to the boards's QSFP28 connector may be located in the uppermost SLR.

To remedy timing failures due to suboptimal placement onto SLRs, manual placement hints or SLR crossing register slices, which trade improvements in frequency for additional latency and area, can be used.

10.4.5 *Fast Hardware/Software-Co-Simulation of TCP/IP-capable Applications*

This section describes the approach of simulating the behavior of a user kernel and the TCP/IP stack in a way that is not purely static and testbench-driven but dynamic in the sense that the simulated model can interact with its environment by exchanging Ethernet packets with the host operating system running the simulator.

A TCP/IP-capable TaPaSCo PE consists of several components implemented in different languages, with each offering its own simulation infrastructure: HLS modules can be natively compiled and tested,

Bluespec modules can be simulated using *Bluesim*, and SystemVerilog modules can be simulated using an HDL-Simulator.

Seeing that both C++ and Bluespec can be compiled to Verilog, the fully integrated design can be simulated at the HDL level. While this HDL simulation is slower than a simulation of individual Bluespec or C++ components, it is the only way of assessing the behavior of the whole system.

The *Vivado Simulator* is a mixed-language (Verilog, SystemVerilog, VHDL), event-driven HDL simulator that is part of the Vivado software suite. It is chosen as the underlying HDL simulator since it integrates well with the remainder of the Vivado-based development flow of both TaPaSCo and the TCP/IP stack and since it supports the simulation of encrypted Xilinx IP cores. It contains the proprietary *Xilinx Simulator Interface*, a C API that allows a C/C++-based testbench to interact with a device under test (DUT) by reading and writing its top-level signals. A testbench implemented in C/C++ can use external libraries or operating system APIs, thus realizing complex interactions with a DUT that would be challenging to implement in an HDL-based testbench.

10.4.5.1 *Architecture*

To effectively simulate the behavior of a TCP/IP-capable TaPaSCo PE, the testbench must model the Ethernet layer to which the PE connects via its AXI-Stream ports. While hard-coding several test Ethernet frames into the testbench may be reasonable for stateless upper-layer protocols, this quickly becomes infeasible for the TCP protocol, where state information is attached to each TCP session e.g. in the form of sequence and ACK numbers. This implies that an effective testbench for TCP-based applications needs to be capable of processing network packets by, at least partially, implementing all involved network protocols.

Implementing any packet processing logic in Verilog seems unproductive, considering that even basic software implementations of a TCP/IP stack in high-level languages span several thousand lines of code (e.g., [21] and [33], both Linux userspace stacks with roughly 4k resp. 6k lines of C, or [28], a standalone embedded stack with roughly 24k lines of Rust).

Therefore, the complexity of the simulator architecture is reduced by (1) implementing the testbench in a higher-level language (C++) and (2) offloading packet processing itself to the Linux TCP/IP stack.

10.4.5.2 *High-level language for testbench*

The Vivado Simulator can be instructed to compile an HDL design into a C library that implements a behavioral simulation model of the design. A testbench written in C/C++ can be linked with this library

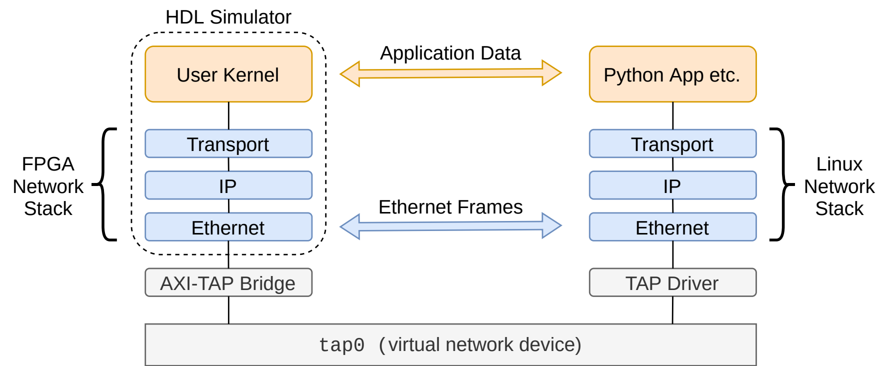


Figure 10.2: Simulator architecture for TCP/IP-capable TaPaSCo PEs

and use the Xilinx Simulator Interface (XSI) for communication with the top module. By appropriately calling XSI functions, the testbench can write values to top-level HDL ports, read current values from those ports, and advance the simulation time.

10.4.5.3 *Linux Stack Offloading*

Since the testbench is C++-based, it may leverage arbitrary APIs of the host operating system. In this particular case, it hands over any packet parsing and processing tasks to the fully-featured Linux TCP/IP stack, such that the simulator itself does not need to include dedicated logic for this.

The simulator interacts with the Linux TCP/IP stack via a *TAP* device, which is a type of virtual network interface. A TAP device behaves like a standard Linux network interface, but rather than attaching to a physical network interface controller that interacts with a transmission medium, the data stream on the data link layer is exposed via a file descriptor. This file descriptor is read and written by a simulator component called *AXI-TAP Bridge* that translates between HDL signals and Ethernet frames.

Like the TAP device, the FPGA TCP/IP stack implements all upper-layer protocols down to the link layer, thus it is possible to bridge the Linux TCP/IP stack and the FPGA TCP/IP stack on this layer, effectively emulating the physical layer. An overview of the simulator architecture in relation to protocol layers and the positioning of the *AXI-TAP Bridge* is shown in Fig. 10.2.

In more detail, after reassembling an Ethernet frame from AXI-Stream beats, which are transmitted by the FPGA stack via its TX AXI-Stream port, this frame is written to the TAP device. The frame is then parsed and processed according to its content by the Linux TCP/IP stack. Conversely, any application data sent from a userspace process via the TAP network interface is encoded by the Linux TCP/IP stack into Ethernet frames that are translated by the AXI-TAP Bridge

into AXI-Stream beats which are written to the FPGA stack via its RX AXI-Stream port.

10.4.6 *FSM-based Simulator Architecture*

A primary concern of the C++-based testbench is to interact with the AXI-Stream ports of the TaPaSCo PE simulation model for the exchange of Ethernet frames. In addition to these two AXI-Stream ports, the TaPaSCo PE contains an AXI-Lite control interface, which also needs to be driven by the testbench to set the PE arguments or control its execution state.

Seeing that AXI-Stream is unidirectional and AXI lite supports full-duplex, there are a total of four independent data streams to and from the TaPaSCo PE. While the AXI-Lite control interface is not strictly performance-critical, the AXI-Stream channels are. Waiting for a `TVALID` or `TREADY` signal in one channel must not block the other from sending or receiving data, as this would inaccurately model the underlying full-duplex connection of a real-world application.

As a result, the testbench must be able to handle four independent data streams to and from the PE simultaneously. This is achieved by an architecture of four FSMs that execute in parallel, where each FSM handles one data stream by interacting with its associated HDL signals in each clock cycle.

Read and write operations on the AXI-Lite interface are each implemented by an FSM that has a command queue for the addresses and data to be read or written. They are used e.g. to set an argument of the PE or to determine the return value of the PE.

The TX and RX AXI-Stream interfaces are each implemented by an FSM that is able to process one data word per clock cycle. Ethernet frames coming from the TAP device are split into individual AXI-Stream beats and transferred onto the RX AXI-Stream interface. AXI-Stream beats coming from the TX interface are buffered and reassembled into an Ethernet frame, which is then forwarded to the TAP device.

10.4.7 *"In Circuit" Emulation*

The proposed architecture enables arbitrary userspace software like *wireshark*, *netcat* or custom Python applications to interact with the live simulation model. Using these high-level tools significantly simplifies the development and debugging of TCP/IP-capable TaPaSCo PEs. The simulator supports two methods of interacting with the AXI-Lite control interface of the PE. First, a simple UNIX signal handler can trigger a predetermined AXI transaction, and second, arbitrary AXI transactions can be triggered via a UDP control socket.

The simulator is available as free and open-source software [20].

10.5 EVALUATION

In order to evaluate the achievable network performance of TaPaSCo designs, throughput and latency measurements using the network protocols TCP, UDP, plain Ethernet, and Aurora are performed and compared with each other.

10.5.1 *Experimental Setup*

Multiple experiments are conducted using a setup consisting of two FPGA boards, one BittWare XUP-VVH and one Xilinx Alveo U280, connected via a 100 Gigabit network link.

Early experiments have shown that in setups consisting of one FPGA and one commodity server, the server-side often severely limits network performance. Achieving throughput close to the line rate requires non-trivial optimizations on the server-side, as demonstrated in [15]. Therefore, we do not further consider this type of setup in the evaluation of this work. Instead, we use one of the FPGAs as traffic generator to test the true capabilities of the setup in a throughput test.

The FPGA designs for benchmarking different network protocols are generated using TaPaSCo and contain exactly one PE that operates at 250 MHz. At this frequency, the 512 bit AXI-Stream interface between PE and Ethernet or Aurora subsystem can provide a theoretical throughput of 128 Gbps. A simulation of the TCP/IP stack using an MSS of 4 KiB shows that a link utilization of over 97% is reached on this AXI-Stream interface. The resulting net data rate is sufficient to saturate the 100 Gbps line rate of the Ethernet backend.

Since a TCP buffer implementation in DRAM was found to bottleneck the system even when using RX buffer bypassing, if not mentioned otherwise, the buffers are implemented in HBM, which was found to not limit performance.

The benchmark PE implements both a throughput and a latency test, in both cases using a server-client-architecture. For the protocols TCP and UDP, it contains the TCP/IP stack and uses the Bluespec networking library introduced in Section 10.4.3 as foundation for the test implementation.

The throughput achievable on the link between the two FPGAs using a particular setup is measured by timing the duration it takes to transfer 100 GB of data. Network packets are sent from the client to the server PE as fast as possible. The amount of payload per packet is configurable and is varied between 1 KiB and 8 KiB, depending on the specific experiment.

For the case of TCP, splitting the 100 GB transfer into MSS-sized packets is handled by the Bluespec network library. For UDP and Ethernet, this is implemented within the user kernel itself. Depending on the specific network protocol, different amounts and different kinds

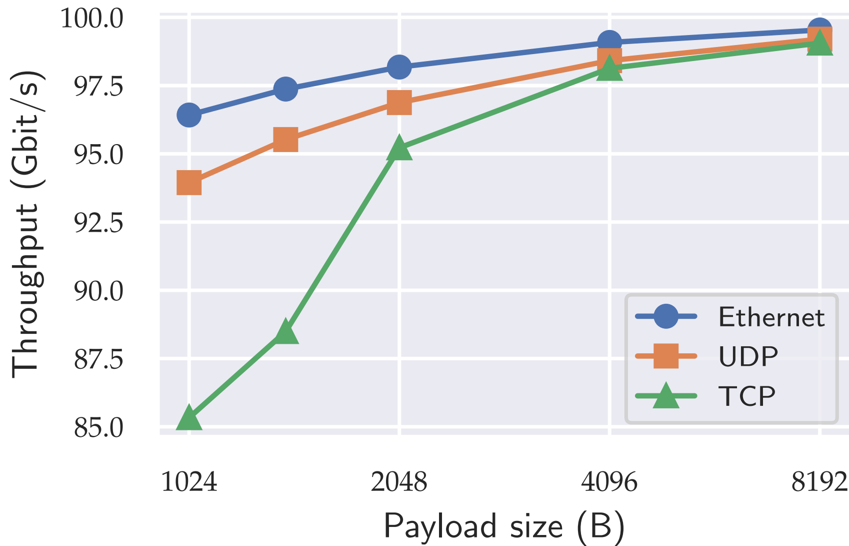


Figure 10.3: Throughput of TCP, UDP, and Ethernet, for different payload sizes per packet. Note that the y-axis does not start at zero.

of packet headers are prepended to the payload. While the payload size is kept constant across protocols, this results in overall packets of varying sizes. As Aurora is a packet-less protocol, splitting the data is not required in the Aurora-based test. Instead, all data is transmitted in a single headerless frame.

The latency of the link between the two FPGAs is measured as the round-trip time (RTT) of a 32 byte-sized ping packet. UDP, Ethernet, and Aurora are connectionless protocols where the ping packet can be sent to a server without any prerequisites. The same is not true for a TCP-based latency test, where a TCP connection has to be established prior to transmitting the ping packet.

10.5.2 *Throughput*

In this experiment, the relationship between the choice of network protocol, the payload size, and the measured throughput is evaluated. The payload size affects the fraction of protocol overhead, and thus places an upper bound on the achievable throughput. The experiment is conducted using receive buffer bypassing (see Section 10.5.4 for more details). Furthermore, a TCP window size of 256 KiB is used.

The usable throughput at the application layer ("goodput") is obviously lower than the bandwidth of the 100G network link, and depends on the ratio between payload size and total size of an Ether-

net frame. For plain Ethernet using an MTU of 4 KiB, the theoretical goodput is equal to

$$\frac{100Gbps \cdot 4KiB}{4KiB + 8B + 14B + 4B + 12B} = 99.081Gbps. \tag{10.1}$$

Using an MSS of 4 KiB and assuming an IPv4 header with no options (20 byte), a TCP header with no options (20 byte), and taking into consideration the overhead from Eq. (10.1), the theoretical goodput of a TCP connection is given by

$$\frac{100Gbps \cdot 4KiB}{4KiB + 38B + 20B + 20B} = 98.131Gbps. \tag{10.2}$$

Assuming equivalent constraints as with TCP, the UDP protocol achieves a theoretical goodput of

$$\frac{100Gbps \cdot 4KiB}{4KiB + 38B + 20B + 8B} = 98.414Gbps. \tag{10.3}$$

Since Aurora is not a packet-based protocol, it does not carry any packet header overhead. In its reference implementation, however, the *clock compensation* mechanism inhibits data transmission for a maximum of 8 clock cycles every 4992 clock cycles [4, p.20], resulting in a different kind of protocol overhead. Assuming an infinite data frame, the worst-case goodput of Aurora 64B/66B amounts to

$$\frac{100Gbps \cdot 4992B}{5000B} = 99.840Gbps. \tag{10.4}$$

Fig. 10.3 shows throughput measurements for payload sizes between 1 KiB and 8 KiB. The payload size of 1460 bytes is significant, as it is the largest MSS that fits into an Ethernet frame with default MTU of 1500 bytes. As both UDP and TCP operate on top of Ethernet, both carry a higher protocol overhead and achieve strictly lower throughput. For the same reason, UDP performs better than TCP, particularly for small payloads.

Fixing the packet based protocols to a payload size of 4 KiB, the measured throughput and calculated optimum for TCP, UDP, Ethernet, and Aurora are shown in Fig. 10.4. With this setup, the measured throughput of all protocols is virtually equivalent to the theoretical optimum derived in Eqs. (10.1) to (10.4).

The achievable throughput of a protocol expectedly is inversely proportional to the amount of overhead carried by it. From this perspective, Aurora is of particular relevance since it can be considered the limit case where packet header overhead is reduced to zero. However, due to the *clock compensation* overhead, Aurora cannot achieve perfect bandwidth utilization.

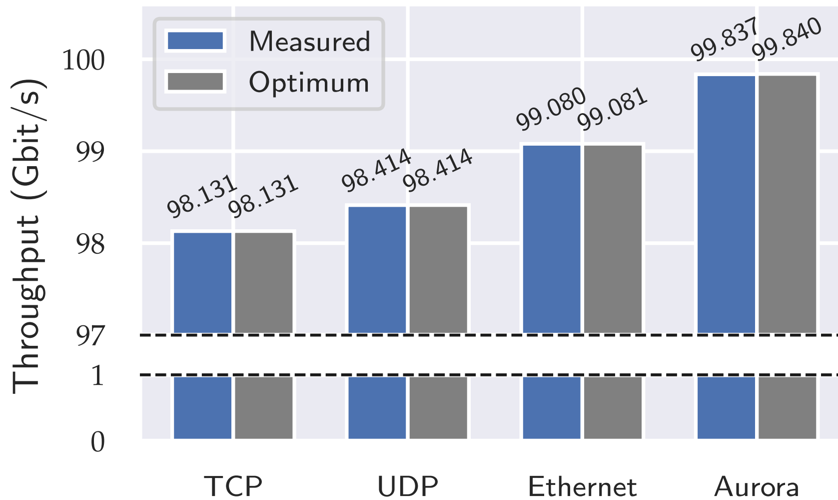


Figure 10.4: Measured throughput and theoretical maximum for different protocols. Except for Aurora, all protocols use a payload size of 4 KiB

10.5.3 *Latency*

A comparison of RTT latency measurements for different protocols is shown in Fig. 10.5. Naturally, both UDP and TCP have a higher latency than Ethernet since these protocols are constructed on top of Ethernet. Also expectedly, TCP has the highest latency of all since its implementation is by far the most control-intensive. All latency measurements for TCP are executed within an established TCP session.

The average duration of a TCP handshake, which is necessary for establishing a TCP connection, is 2579 ns, i.e. slightly faster than the RTT of a ping packet. This is plausible because the data-less handshake packets do not traverse the full TCP/IP stack, such that the handshaking sequence is processed faster than a data-carrying ping packet.

10.5.4 *RX Bypass*

This section presents the examination of the configurable receive buffer bypass of the TCP/IP stack for TCP connections. For this, two variables are considered: (1) whether or not the bypass is enabled in the FPGA design, (2) which memory technology is used to implement the buffers. Memory subsystems based on HBM and BRAM are evaluated, resulting in four possible configurations that are compared in terms of throughput and latency.

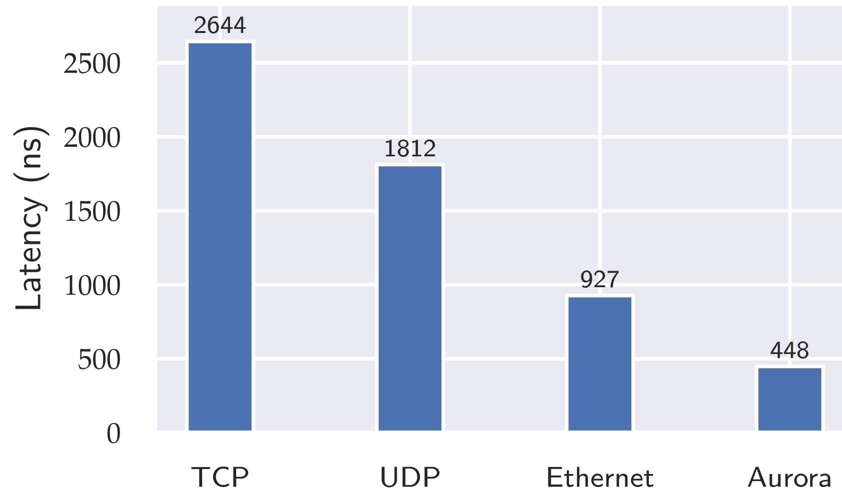


Figure 10.5: RTT latencies using different protocols. For TCP, the duration of the handshake is not included.

10.5.4.1 *Throughput*

Fig. 10.6 shows the results of a throughput test for all four configurations. The theoretical maximum is calculated according to Eq. (10.2) and is marked in the figure. The two configurations using receive buffer bypassing perform extremely close to the theoretical maximum, regardless of the memory technology used. The throughput of the HBM-based system approximately halves if the bypass is disabled, whereas the BRAM-based system achieves the same performance with or without bypassing.

Regardless of the configuration, all transmitted data is written to memory once. However, assuming there are no retransmissions, this data is never read back. Disabling the bypass further increases memory pressure, since all received data is written to memory and read back shortly after when the application requests new data from the receive buffer. For HBM specifically, this access pattern yields suboptimal performance due to bus turnaround times between write and read operations [5, p.23].

The throughput result of the HBM-based configuration without bypass is noticeably low, yet plausible, considering that data must be written to and read from memory at approximately 50 Gbps (6.25 GBps). In [9], a sequential combined read/write throughput of 12.9 GBps was measured using a single HBM channel on an Alveo U280, which aligns with the throughput result of the experiment in this work. It is thus concluded that memory performance can impede the achievable throughput performance, and that memory pressure can be relieved by employing RX buffer bypassing.

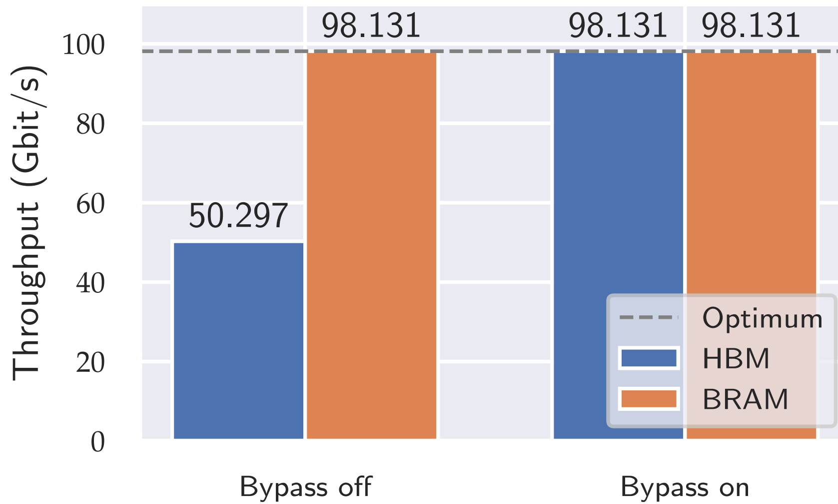


Figure 10.6: TCP throughput with and without receive buffer bypass, for HBM and BRAM based buffer implementations. The MSS is 4 KiB, and the corresponding theoretical maximum throughput is marked.

10.5.4.2 *Latency*

For each of the four possible configurations, Fig. 10.7 shows both the duration of a TCP handshake (HS) as well as the RTT of a ping packet that is sent immediately after the handshake completes. It is noticeable that the duration of the TCP handshake is largely unaffected across all different configurations. This is expected because data-less TCP control packets like SYN, SYN+ACK, and ACK are never buffered and thus independent of the buffer architecture.

When enabling the buffer bypass, the RTT decreases by approx. 27% for BRAM and by approx. 29% for HBM. This decrease is caused by the fact that a received ping packet is now directly delivered to the application, instead of being first written to memory and then read back before being delivered to the application. This bypassing takes place at the server, when receiving the initial ping packet, and at the client, when receiving the ping reply. Generally, the TCP receive buffer is essential if the application layer cannot process bursts of incoming packets at line rate or if packets regularly arrive out-of-order and require buffering for reordering.

10.5.5 *Resource Utilization*

In Table 10.1, the stack's resource utilization is summarized for a configuration that implements both the TCP and the UDP protocol, one that implements only UDP, and one that implements only TCP. As expected, the implementation of the TCP protocol proves to be the

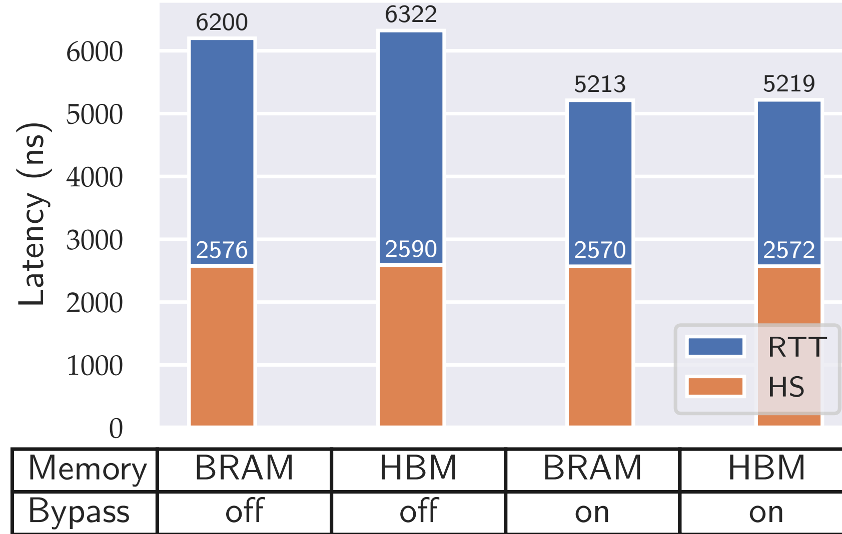


Figure 10.7: Latency results of a TCP ping test, with and without buffer bypass, for HBM and BRAM based buffer implementations. The figure shows the duration of the TCP handshake (HS) and the subsequent RTT of a ping packet.

Table 10.1: Overview of resource utilization of different TCP/IP stack configurations.

Component	CLB LUTS		Registers		Block RAMs	
	abs.	%	abs.	%	abs.	%
TCP & UDP	121491	9.3	212599	8.2	463.0	23.0
TCP only	114966	8.8	185244	7.1	439.5	21.8
UDP only	32395	2.5	93886	3.6	120.0	6.0

most resource-intensive. The TCP-only configuration requires almost four times the number of LUTs and BRAMs and almost twice the number of registers compared to the UDP-only version. As a result, the TCP-only configuration is similar in resource utilization to the combined TCP and UDP configuration.

10.6 CASE STUDY: IN-NETWORK ACCELERATION OF SUM-PRODUCT NETWORK INFERENCE

In order to demonstrate how the design primitives described in Section 10.4.3 can be used to realize a network-attached accelerator for an actual application and which handy role simulation (Section 10.4.5) can play in the design process, we will use an existing FPGA-based accelerator [29, 30] for the inference in so-called *Sum-Product Networks* (SPN) as an example.

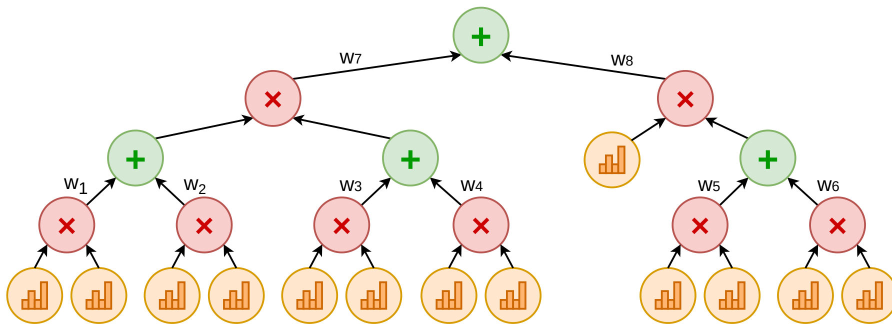


Figure 10.8: Example for the graph structure of a Sum-Product Network, capturing the joint probability distribution over a set of variables.

10.6.1 *Sum-Product Network Background*

Similar to other probabilistic (graphical) models, Sum-Product Networks [22] are recently receiving increasing attention from industry and academia alike. Due to their true probabilistic semantics, Sum-Product Networks can much better cope with the uncertainties found in real-world applications and are also able to quantify their uncertainty over their own output by means of probabilities, which makes them an appealing complement and alternative to currently more widely used machine learning techniques such as neural networks.

Sum-Product Networks capture the joint probability over a set of variables in the form of a directed acyclic graph (DAG), with three different types of nodes. Leaf nodes represent univariate distributions (e.g., Gaussian) over a single variable. Product nodes, on the other hand, represent a factorization of independent subsets of variables, while the weighted sum nodes indicate a mixture of multiple distributions. An example is shown in Fig. 10.8.

The graph structure of an SPN can either be handcrafted, complemented by weight learning, or automatically be learned from training data. An overview of the various available learning algorithms can be found in [25]. The survey also provides a nice overview of a wide range of usage examples for Sum-Product Networks, ranging from medical imaging [23] to approximate query processing for databases [14].

After the graph structure has been obtained, inference can be used to answer probabilistic queries. To this end, the SPN DAG is traversed in a bottom-up fashion starting at the leaf nodes, where the (partial) input evidence is used to obtain probabilities. After propagating these probabilities through the graph, performing the respective operations, the final probability value is obtained at the single root node of the SPN.

The existing SPN accelerator [30], which we use as an example here, is designed to accelerate this inference step. Training of the SPN is assumed to have taken place beforehand.

10.6.2 *Streaming-based Accelerator*

In our prior work, we have employed SPN-accelerators to accelerate batch-wise processing of SPN inference queries [29, 30]. For the work presented here, due to the streaming-based interfaces of the networking stack, we had to adapt the corresponding accelerators to make them compatible. In prior work, the accelerator uses four distinct sub-modules for the SPN inference. 1) A control register allows configuration of the accelerator. 2) A Load Unit is responsible for loading input data from on-device DRAM. 3) The SPN-Datapath performs the actual inference and is fed by the Load Unit. 4) Results from the SPN-Datapath are passed to a Store Unit, which will write back the data.

In this work, we have adapted the SPN-Datapath from prior work to run as a free-running kernel. This means that no configuration is necessary. Additionally, the Load- and Store Unit have been stripped from the accelerator. Instead of using AXI4 for loading and storing the input- and output-data, we now rely on AXI-Stream to feed data directly into the SPN-Datapath. The results are then passed on via a second AXI-Stream interface. Overall, this makes the accelerators a lot more light-weight and reduces the control- and configuration overhead to zero. Data is pushed into the accelerator via AXI-Stream and results can be received via a second AXI-Stream interface. The bitwidths of both of those interfaces may be varied depending on the underlying SPN. The AXI-Stream slave interface (used for receiving input-vectors) is sized according to the size of a single input-vector (in this work, we use up to 640 bit wide input-vectors). The AXI-Stream master interface (used for sending the inference results) is 64 bit wide, due to the fact that the output is a single IEEE 754 double precision float. The accelerator is able to accept a complete input-vector every cycle. Due to its deeply pipelined nature, the accelerator is able to processes a single input-vector every cycle, assuming the the pipeline is kept full. The latency of the accelerator depends on the underlying SPN, since the SPN datapath varies in depth with the corresponding SPN.

10.6.3 *Network Integration*

The data path of the Xilinx CMAC is implemented by a 64-byte-wide AXI-Stream interface. Since the AXI-Stream slave interface of an SPN can have an arbitrary byte-width, a width-conversion is necessary in the general case when connecting the RX-path of the CMAC and slave-side of the SPN. The same is required for connecting the master-side of the SPN to the TX-path of the CMAC. In both cases, a Multiple-In Multiple-Out (MIMO) module is employed for this purpose.

Table 10.2: Degree of replication of different NIPS-SPNs used in the experimental evaluation.

NIPS Variant	10	20	30	40	50	60	70	80
Number of Instances	5	3	2	2	1	1	1	1

To leverage the full bandwidth of the network connection while keeping the clock frequency of the SPN in an acceptable range, it may be necessary to replicate the SPN accelerator module several times in order to multiply the inference rate. The architecture of the TaPaSCo PE we used during the evaluation is equal to the one previously shown in Fig. 10.1.

10.6.4 *Experimental Evaluation*

In this section, we will discuss the results of the experimental evaluation using the SPN-accelerators presented in the prior sections. Each of the eight NIPS accelerator represents a different benchmark from the NeurIPS corpus [11], and the number indicates the number of inputs in the input-vector of the corresponding SPN. For example, NIPS10 will use 10 input-values, with each input-value being 8 bits wide. Thus the overall size of the input vector is 10 bytes or 80 bits.

Since all of the used NIPS-SPNs are able to run at 250 MHz, we have to use replication, to ensure that the full available network bandwidth can be exploited. At 250 MHz, we have to make sure that the SPNs can accept at least 50 bytes of data per cycle. Thus, for NIPS10, we need to replicate it five times to achieve this. Larger SPNs (like NIPS80) do not need to be replicated. The specific degree of replication that we used during evaluation is listed in Table 10.2.

The resulting throughputs are depicted in Fig. 10.9. It is important to note, that due to the point-to-point nature of the Aurora protocol, it did not make sense to include it in the evaluation. This is due to the fact that it would be incompatible with the concept of replicating accelerators to exploit the available bandwidth. Instead, the results are limited to the comparison of TCP, UDP and plain Ethernet.

Apart from that, Fig. 10.9 shows that the throughput is very close to the theoretical limit of 100 Gbps for all different protocols. This shows that the presented networking stack does not only work with synthetic benchmarks, but also with more real-world applications, like SPN inference. Additionally, the peak throughput was achieved for many different accelerators using different input-widths, which also highlights the flexibility of the stack.

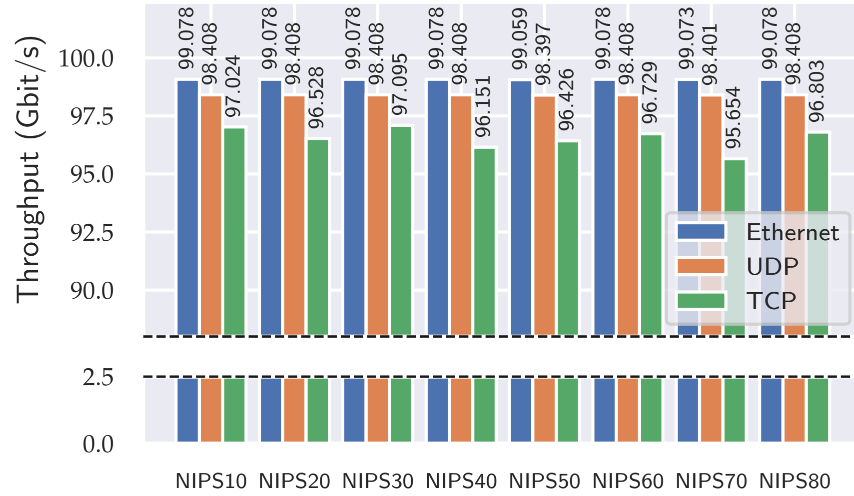


Figure 10.9: Throughput of different SPN variants using different network protocols for input and output data transfer.

10.7 CONCLUSION & OUTLOOK

In this work, we have integrated a high-throughput hardware network stack into the open-source TaPaSCo framework. During the integration process, several limitation of the existing stack have been removed and a combination with faster HBM memory was developed to allow for flexible configuration of the stack and better performance.

Next to that, we also developed a library of easy-to-use design primitives for network-attached accelerators in a modern HDL. The library is complemented by a simulation framework which leverages the Linux TCP stack to allow implementing testbenches for accelerators in high-level languages such as C/C++ or Python. The combination of the design primitives, the new simulation framework and TaPaSCo's automatic design-exploration framework make network-attached SoC-designs more accessible for researches and significantly facilitate the design process.

Our evaluation demonstrates that the integrated stack is able to achieve the maximum theoretical possible throughput. This finding has been confirmed in the case study, using a machine learning inference accelerator for Sum-Product Networks as an example, which also demonstrates how the design primitives can be used to attach existing accelerators to the network.

ACKNOWLEDGEMENTS

This work has been co-funded by the German Research Foundation (DFG) as part of project D2 within the Collaborative Research Center (CRC) 1053 MAKI and by the German Federal Ministry for Education

and Research (BMBF) with the funding ID ZN 01|S17050. The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software.

REFERENCES

[1] *25G Bit TCP Offload Engine (TOE)*. https://intilop.com/resources/product_briefs/25G_1K-Sess_TCP+UDP_Offload+MAC+Host_IFUltra-LowLatency(INT-25011).pdf. [Online, accessed June 2021].

[2] *Algo-Logic: Ultra-Low-Latency 10G TCP Endpoint*. https://www.algo-logic.com/10g-tcp-endpoint. [Online, accessed June 2021].

[3] *Aurora 64B/66B Protocol Specification (SP011 (v1.3) October 1, 2014)*. https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b_protocol_spec_sp011.pdf.

[4] *Aurora 64B/66B v12.0 (PG074 December 4, 2020)*. https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b/v12_0/pg074-aurora-64b66b.pdf.

[5] *AXI High BandwidthMemory Controller v1.0 (PG276 (v1.0) January 21, 2021)*. https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.

[6] *BlueNET*. https://git.esa.informatik.tu-darmstadt.de/net/bluenet.

[7] Andrew Boutros, Brett Grady, Mustafa Abbas, and Paul Chow. "Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis." In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2017, pp. 1–6.

[8] *BSV Documentation*. http://wiki.bluespec.com/Home/BSV-Documentation. [Online, accessed June 2021].

[9] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. "When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization." In: (2020). arXiv: 2010.06075 [cs.AR].

[10] Eric Chung et al. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave." In: *IEEE Micro* 38.2 (2018), pp. 8–20. DOI: 10.1109/MM.2018.022071131.

[11] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[12] *Enyx IP Cores*. https://www.enyx.com/ip-cores. [Online, accessed June 2021].

[13] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud." In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, not from Queries!" In: *CoRR* abs/1909.00607 (2019). arXiv: 1909.00607. URL: http://arxiv.org/abs/1909.00607.

[15] Mario Hock, Maxime Veit, Felix Neumeister, Roland Bless, and Martina Zitterbart. "Tcp at 100 gbit/s–tuning, limitations, congestion control." In: *2019 IEEE 44th Conference on Local Computer Networks (LCN)*. IEEE. 2019, pp. 1–9.

[16] Jaco Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. "High-Performance In-Network Data Processing." In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, United States*. 2019.

[17] Yong Ji and Qing-Sheng Hu. "40Gbps multi-connection TCP/IP offload engine." In: *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*. IEEE. 2011, pp. 1–5.

[18] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The tapasco open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems." In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2019, pp. 214–229.

[19] Sascha Mühlbach, M. Brunner, C. Roblee, and Andreas Koch. "MalCoBox: Designing a 10 Gb/s Malware Collection Honeypot Using Reconfigurable Technology." In: *IEEE Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE. 2010.

[20] *net-sim*. https://git.esa.informatik.tu-darmstadt.de/net/net-sim.

[21] *nstack*. https://github.com/jserv/nstack. [Online, accessed May 2021].

[22] Hoifung Poon and Pedro Domingos. "Sum-product networks: A new deep architecture." In: *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. 2011.

[23] Fabian Rathke, Mattia Desana, and Christoph Schnörr. "Locally adaptive probabilistic models for global segmentation of pathological OCT scans." In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2017, pp. 177–184.

[24] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. "Limago: an FPGA-based Open-source 100 GbE TCP/IP Stack." In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. Sept. 2019, pp. 286–292. DOI: 10.1109/FPL.2019.00053.

[25] Raquel Sánchez-Cauce, Iago París, and Francisco Javier Díez. "Sum-product networks: A survey." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[26] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. "In-Network Computation is a Dumb Idea Whose Time Has Come." In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. HotNets-XVI. Palo Alto, CA, USA: Association for Computing Machinery, 2017, pp. 150–156. ISBN: 9781450355698. DOI: 10.1145/3152434.3152461. URL: https://doi.org/10.1145/3152434.3152461.

[27] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware." In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 36–43. DOI: 10.1109/FCCM.2015.12.

[28] *smoltcp*. https://github.com/smoltcp-rs/smoltcp. [Online, accessed May 2021].

[29] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-based Accelerators." In: *IEEE International Conference on Computer Design (ICCD)*. IEEE. 2018.

[30] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. "Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs." In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020, pp. 1–10.

[31] Gustavo Sutter, Mario Ruiz, Sergio Lopez-Buedo, and Gustavo Alonso. "FPGA-based TCP/IP checksum offloading engine for 100 Gbps networks." In: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2018, pp. 1–6.

[32] *TaPaSCo*. https://github.com/esa-tu-darmstadt/tapasco.

[33] *tapip*. https://github.com/chobits/tapip. [Online, accessed May 2021].

[34] *TCP Offload Engine: LightSpeed TCP*. https://ldatech.com/Solutions/LightSpeedTCP. [Online, accessed June 2021].

[35] *TCP/IP & UDP Network Protocol Acceleration Platform (NPAP).* https://www.missinglinkelectronics.com/index.php/menu-products/menu-network-protocol-accelerator. [Online, accessed June 2021].

[36] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. "The Case For In-Network Computing On Demand." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303979. URL: https://doi.org/10.1145/3302424.3303979.

[37] *UltraScale+ Devices Integrated 100G Ethernet Subsystem v2.4 (PG203 April 4, 2018).* https://www.xilinx.com/support/documentation/ip_documentation/cmac_usplus/v2_4/pg203-cmac-usplus.pdf.

[38] *Vitis with 100 Gbps TCP/IP Network Stack.* https://github.com/fpgasystems/Vitis_with_100Gbps_TCP-IP. [Online, accessed June 2021].

[39] *Vivado Design Suite User Guide: High-Level Synthesis (UG902 (v2020.1) May 4, 2021).* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.

[40] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. "Network-attached FPGAs for data center applications." In: *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2016, pp. 36–43.

[41] Johannes Wirth, Jaco A. Hofmann, Lasse Thostrup, Andreas Koch, and Carsten Binnig. "Exploiting 3D Memory for Accelerated In-Network Processing of Hash Joins in Distributed Databases." In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Steven Derrien, Frank Hannig, Pedro C. Diniz, and Daniel Chillet. Cham: Springer International Publishing, 2021, pp. 18–32. ISBN: 978-3-030-79025-7.

[42] Zhong-Zhen Wu and Han-Chiang Chen. "Design and implementation of tcp/ip offload engine system over gigabit ethernet." In: *Proceedings of 15th International Conference on Computer Communications and Networks*. IEEE. 2006, pp. 245–250.

[43] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. "Achieving 100Gbps Intrusion Prevention on a Single Server." In: *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. USENIX Association, 2020, pp. 1083–1100. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.

[44] H. Zimmermann. "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection." In: *IEEE Transactions on Communications* 28.4 (1980), pp. 425–432. DOI: 10.1109/TCOM.1980.1094702.

Copyright Notice: © 2021 IEEE. Reprinted, with permission, from Marco Hartmann et al. "Optimizing a Hardware Network Stack to Realize an In-Network ML Inference Application." In: *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2021, pp. 21–32. DOI: 10.1109/H2RC54759.2021.00008.

11 EXPLOITING HIGH-BANDWIDTH MEMORY FOR FPGA-ACCELERATION OF INFERENCE ON SUM-PRODUCT NETWORKS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *Exploiting High-Bandwidth Memory for FPGA-Acceleration of Inference on Sum-Product Networks* by *Lukas Weber, Johannes Wirth, Lukas Sommer, Andreas Koch* in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. The contribution of the author of this thesis is summarized as follows.

» *As the corresponding and co-leading author, Lukas Max Weber contributed to the evaluated Sum-Product Network accelerators that were integrated with the memory infrastructure provided by Johannes Wirth. Lukas Sommer contributed the improved software interface. Johannes Wirth contributed the evaluation of the HBM performance and conducted corresponding performance measurements using the improved architecture. Resource utilization was measured by Lukas Max Weber, who also collected and analyzed the results. The manuscript was written by Lukas Max Weber and refined with feedback from the other co-authors.* «

ABSTRACT

Due to the memory wall becoming increasingly problematic in high-performance computing, there is a steady push to improve memory architectures, mainly focusing on better bandwidth as well as latency. One of the results of this push is the development of High-Bandwidth Memory (HBM) which is an alternative to the regular DRAM typically used by accelerator-cards.

This work adapts an existing accelerator architecture for inference on Sum-Product Networks (SPN) to exploit the HBM present on more recent high-performance FPGA-accelerator cards. The evaluation shows that the use of HBM enables almost linear scaling of the performance due to the embarrassingly parallel nature of batch-wise SPN inference. It is also shown that the only hindrance to this scaling is the limited bandwidth available for data-transfers between host and FPGA. Even with this bottleneck, the prior FPGA-based implementation is outperformed by up to 1.50x (geo.-mean 1.29x). Similarly, the CPU and GPU baselines are outperformed by up to 2.4x (geo.-mean 1.6x) and 8.4x (geo.-mean 6.9x) respectively.

Based on the evaluation, the scaling potential of HBM-based FPGA-accelerators is explored to give an outlook on what is to come with future generations of PCIe-based interfaces.

11.1 INTRODUCTION

Artificial intelligence and machine learning (ML) have become pervasive in our every-day life, being deployed in applications such as voice-based smart assistants or in medical applications. Most of the progress made in recent years has not only been enabled by improvements to the ML models themselves, but also by the constant improvement of the execution hardware, which needs to provide sufficient computational power to train models with multiple billions of parameters, and compute inference quickly enough for real-time applications.

Much work on the acceleration of machine learning models has focused on (deep) neural networks (NN). Next to GPUs and dedicated ASIC-accelerators built for the single purpose of accelerating machine learning training and inference, such as Google's TPU or Graphcore's IPU, FPGAs have proven to be a compelling platform for deep neural network (DNN) acceleration [2].

However, despite their broad adoption, deep neural networks can still suffer from serious limitations in real-world usage scenarios. This has sparked an increased interest in *probabilistic* models, which are much better able to cope with real-world uncertainties. While inference for many probabilistic models is intractable in the general case, so-called *Sum-Product Networks* (SPN) [13] combine the strengths of probabilistic models with tractable inference for real-world applications. These properties make SPNs not only an interesting candidate model for ML applications, but also an attractive application for acceleration on different target-platforms, including GPUs [14, 16, 17], custom ASIC processors [14] and also FPGAs [3, 15, 17].

In our prior work, we developed an architecture for high-throughput inference in Sum-Product Networks, based on FPGAs available in Amazon's AWS cloud [11]. As a single instance of the pipelined accelerator for SPN inference did not fully exploit the available FPGA resources, we developed a multi-core architecture, with multiple identical accelerators conducting inference in parallel. However, even though we employed up to four parallel memory banks, the memory accesses during the computations quickly became a bottleneck, in particular due to the relatively low arithmetic intensity of SPN inference.

A promising alternative to overcome this limitation is the use of *High-Bandwidth Memory* (HBM), which FPGA vendors are now increasingly integrating into their products. Being composed of dozens of small, independent blocks of memory, HBM allows multiple memory

accesses to be performed in parallel and thus increases the available memory bandwidth.

This work presents our contribution, which is an adapted accelerator architecture that exploits the highly parallel interface of HBM on FPGAs for high-throughput inference for Sum-Product Networks. The pipelined multi-core accelerator architecture is automatically generated from an SPN description, and is automatically integrated in a heterogeneous system. In addition, the hardware accelerator is combined with an efficient, multi-threaded software runtime interface on the host, to ensure a high-throughput supply of input data for the FPGA accelerator. Our contribution here includes the adapted overarching accelerator architecture, as well as the improved software runtime interface.

11.2 BACKGROUND

11.2.1 *Sum-Product Networks*

In recent years, research interest in machine learning and artificial intelligence has been very high. Especially DNNs have been researched and improved to a great extent. A different, less explored model are Sum-Product Networks (SPN) [13]. Stemming from the class of probabilistic graphical models, they are able to capture joint probability distributions over many different random variables. Their two major advantages are their *tractability* and their ability to handle *uncertainty*. With regard to tractability, inference on SPNs can be performed in linear time w.r.t. the size of an SPN. With regards to uncertainty, SPNs are able to handle uncertainties like missing features or unclear classifications, due to the fact that they compute actual probability values. A very interesting example for this is discussed in the work by Peharz et al. [12] which uses randomly generated SPNs for classification tasks. Confronting an SPN trained for the image classification benchmark MNIST with out-of-domain images yields lower probabilities and thereby indicates that the SPN is *uncertain* about the resulting classification.

In general, SPNs are directed acyclical graphs, comprising three distinct node types: 1) The leaf nodes represent univariate probability distributions over single random variables. 2) Product nodes represent factorizations of independent variables. 3) Sum nodes represent mixtures of distributions.

Using these three node types, SPNs are capable of capturing complex joint probability distributions. To achieve this, the general process is as follows: For each dataset, an independence check is performed to determine if there are any independent variables. If so, this is represented in the SPN using a product node. If this is not the case, the dataset is divided by clustering. The resulting sub-datasets are then

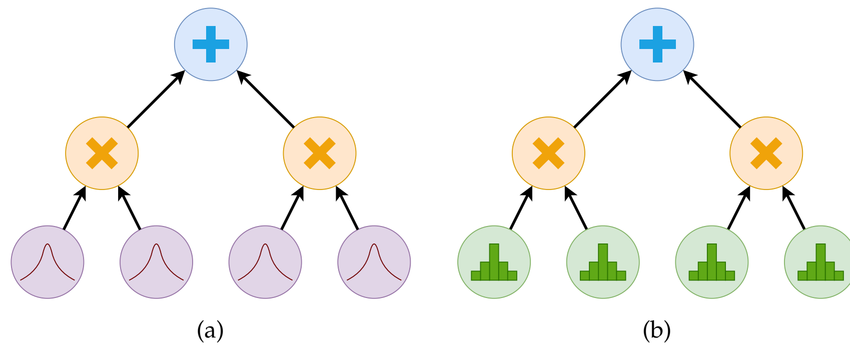


Figure 11.1: Different types of SPNs. (a) shows a typical SPN with Gaussian leaf nodes. (b) shows a Mixed SPN using histograms to approximate the gaussian distributions of (a).

recursively traversed, until the data can be represented using a single univariate distribution. Due to this very simple structure, SPNs are a very simple and concise way of representing complex joint probability distributions. To perform inference on an SPN, a simple bottom-up evaluation has to be performed.

In this work, we rely on a specific flavor of SPNs called Mixed SPNs [9]. The main difference between pure SPNs and mixed ones is the fact that the leaf nodes are approximated using simple histograms (c.f. Fig. 11.1). These histograms can easily be mapped to hardware as shown in [15, 17, 18]. Specifically, we will build upon our prior work [11], which explored the application of SPNs in a heterogeneous reconfigurable cloud (Amazon AWS F1 Instances with FPGA accelerators).

11.2.2 *High-Bandwidth Memory*

In their current UltraScale+ series, Xilinx offers some FPGAs which include High-Bandwidth-Memory (HBM) in addition to conventional off-chip SDRAM. As the name implies, this new type of memory provides significantly more memory bandwidth compared to off-chip SDRAM: According to Xilinx the HBM used on their FPGAs can achieve up to 460GB/s. However, this number can only be achieved when issuing multiple memory requests in parallel, making it necessary to adapt existing architectures in order to actually exploit the additional bandwidth.

The HBM on these Xilinx FPGAs has a capacity ranging from 4GB to 16GB and is split into two stacks. Each stack features 16 **memory channels** with a width of 256 bit, each connected to its own memory region. By default, each of these channels can only access its associated memory region. Each of these memory channels is exposed to the user logic via one AXI3 interface, resulting in a total of 32 AXI3 interfaces for the HBM.

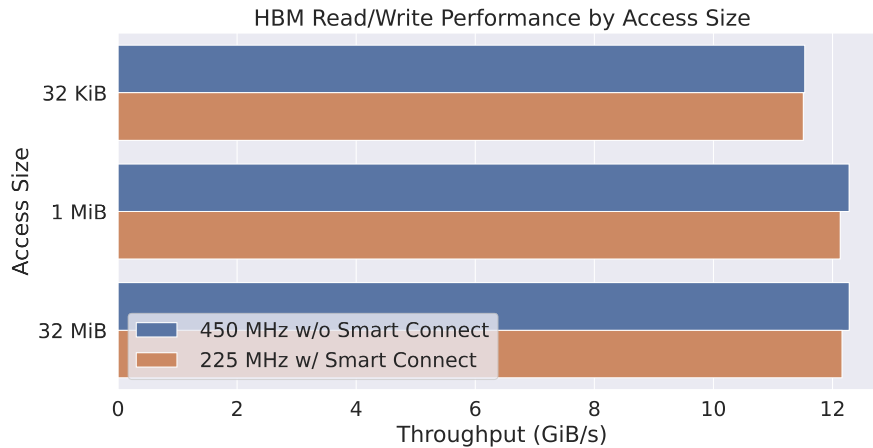


Figure 11.2: Maximum throughput when issuing linear read and write accesses in parallel to one HBM memory channel for two different configurations and different request sizes. The first configuration runs the block generating the accesses with the 450 MHz clock used by the HBM and natively connects both. The second configuration runs the PE at *half* the clock frequency but the interface width is *doubled*. An AXI Smart Connect is used to perform clock- and data-width-conversion.

Xilinx offers an optional crossbar which, when enabled, hides the partitioning from the user and allows to access the *entire* memory space from each AXI interface. However, this comes at the cost of additional latency and decreased performance, where the actual impact is highly dependent on the concrete access pattern. For the rest of this work, we will *not* use the crossbar, since we aim to explore the maximally achievable performance.

Figure 11.2 shows the performance for *one* HBM memory channel. The performance data is generated using a special benchmark hardware block which generates linear memory reads and writes in parallel, as this is the access pattern used by our SPN accelerators. There are two major insights: First, the throughput caps at a request size of 1 MiB, as no further performance improvements are observed beyond this. And second, there is no significant performance benefit when running the benchmark block at 450 MHz with a connection to the HBM at its native interface width versus running the block at *half* the clock frequency and in turn *doubling* the interface width. This is a valuable insight, as it is often not possible to run user logic at 450 MHz. Because we do not use the crossbar, the different HBM memory channels are completely independent and performance scales *linearly* w.r.t. to the number of channels/accelerators used.

11.3 APPROACH

In this section, we will introduce our approach for scaling up the number of SPN-accelerators using HBM. Additionally, we will discuss the motivation and reasoning for the upscaling.

11.3.1 *Motivation*

Considering the theoretical advantages of SPNs over other machine learning models, they have an obvious place in many real-world applications. The fast inference that can be achieved using FPGA-accelerated SPNs is an additional advantage. Since FPGAs are not as wide spread as GPUs, using the reconfigurable cloud is also a reasonable choice (as described in [11]). Unfortunately, looking at the architecture used by [11], there is an obvious problem: Due to the size of the SPN accelerators, as well as their memory-bandwidth requirements, it becomes increasingly hard to map them to Amazon AWS F1 instances. Looking at the NIPS80 benchmarks from [11], we can see that combining bigger accelerators with multiple memory controllers leads to a trade-off: Either we sacrifice memory controllers, which limits the overall throughput of the system by reducing the amount of data that can be accessed in parallel. Or, we reduce the number of accelerators, which means that fewer inferences can be handled concurrently. Specifically, the logic resources on the F1 are insufficient to hold the combination of four NIPS80 accelerators with four separate memory controllers. Thus, only two accelerators were used, which in turn slowed performance for that benchmark. Alternatively, it was possible to use a single memory controller in combination with three SPN accelerators, which also had a performance cost.

If we take into account the advantages of HBM memory (described in Section 11.2.2), it seems very reasonable to replace the use of on-board DRAM with use of on-chip HBM. The HBM controllers are implemented as hard IP and thus do not consume FPGA resources. This, in turn, should allow the use of more accelerator-cores. Since soft memory controllers are also sensitive to clocking constraints, their removal should also improve the problem of globally deteriorating clock frequencies encountered in [11]. Specifically, the use of additional soft memory controllers had a larger impact on the achievable clock frequency than the addition of extra SPN accelerators. Last but not least, the independent HBM blocks can be exclusively assigned to individual SPN accelerators, avoiding interference between them. This should also be another advantage over the shared use of on-board DRAM.

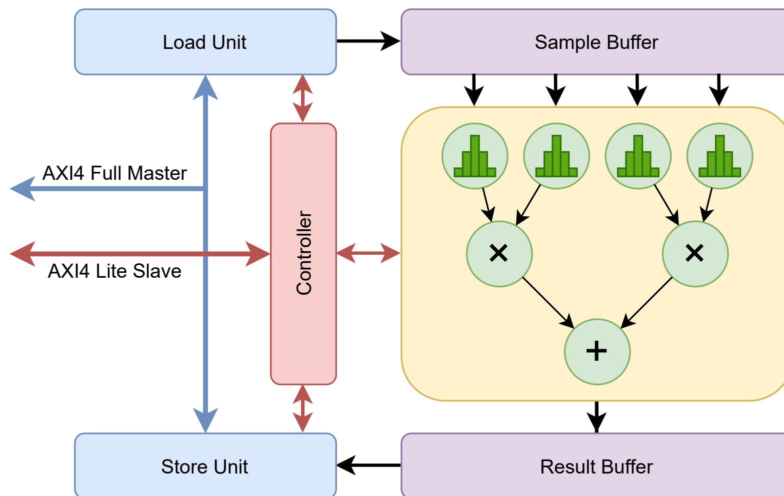


Figure 11.3: Architecture of the SPN-Accelerator

11.3.2 *SPN-Accelerator*

Fundamentally, the basic SPN accelerator-cores have a simple architecture. The accelerator is connected to a memory via a AXI4 Full interface, which typically enables access to the on-board DRAM. The same interface can be used to access on-chip BRAM or HBM without requiring big changes to the accelerator. To ensure compatibility to all kinds of AXI-based memories, we made our interface generation more generic to also cover HBM memories. Accelerators are controlled by an AXI4 Lite Interface, which exposes a simple register file to the user. Due to the increased address-width of the HBM-data-channel, we had to adapt the control registers to 64 bit. Within the accelerator (also depicted in Fig. 11.3), there are multiple submodules which orchestrate the batch-wise inference: First, the **Load Unit** loads the data from the memory and pushes it into the **Sample Buffer**. This buffer collects incoming data until a complete vector of input values has been built. Then, the vector is pushed into the **SPN Datapath**. The result values of the SPN Datapath are collected in a **Result Buffer**. The result buffer collects 64 bit result values, until a 512 bit word is complete. This word is then pushed into the **Store Unit**, which will handle the AXI4 Write to store the results back to memory.

The most important part of the accelerator is the **SPN Datapath**, which can be generated automatically from a textual description of the SPN. The textual description is compatible with the SPFlow library [10], which enables a very simple and streamlined development toolflow. SPNs can be easily trained and evaluated using SPFlow, afterwards exporting them to the textual description for hardware-generation. In addition to the SPFlow-compatibility, the generator offers great flexibility with regards to the used internal number format. In prior work, the different number formats were discussed in

more detail [17], but in general, the generator supports a Custom Floating Point (CFP) format as well as a Logarithmic Number System (LNS) format. Both formats can be configured at a very fine granularity. For CFP, the number of exponent- and mantissa-bits can be configured, as well as the used rounding scheme. For both formats, the generated digital arithmetic is optimized towards the use on FPGAs. The optimizations of LNS are further discussed in [18], while the CFP format is described in detail in [17]. For this work, we chose the suitable configurations determined in [17].

11.4 IMPLEMENTATION

To enable the use of HBM, we have made two distinct changes to our prior work: First, we adapted the on-chip architecture to use HBM in a manner that enables the use of many parallel SPN accelerators. In addition, we made some improvements to the software-interface to ensure that the parallelism provided by the many HBM channels is actually exploited.

11.4.1 *On-Device Architecture*

We use the open-source framework TaPaSCo [7] as a basis for our architecture. However, several modifications had to be made to accommodate the requirements for our use-case.

The biggest change is, of course, the use of HBM instead of off-chip DDR-SDRAM memory. We use a dedicated HBM block (and thus memory channel) per SPN accelerator. However, it is not possible to run the SPN accelerators at the 450 MHz of clock frequency used by the HBM. In order to achieve the same memory throughput, we run the accelerators at the more easily achievable half frequency (225 MHz), but double the interface width to 512 bit. As discussed in Section 11.2.2, this indeed does not affect memory performance. We use an AXI SmartConnect between the accelerator and the HBM, which is responsible for data-width- and clock-conversion. It also performs protocol conversion, as the accelerators use AXI4, while the HBM only supports AXI3. Additionally, we employ register slices on these AXI connections where necessary, to achieve routability. This setup ensures that there are no unnecessary dependencies between the accelerators which might impact performance.

11.4.2 *Parallel Runtime*

To allow users to easily interface with the SPN inference accelerators on the FPGA, we have developed a software runtime, based on the TaPaSCo API. In contrast to prior work, where important parameters and information had to be supplied manually by the user, the new

software runtime can now query the TaPaSCo system and the accelerator itself for these parameters, making it easier to interact with the accelerator. To this end, the accelerator was extended with a second execution mode to read out the configuration parameters specified at synthesis time.

In addition to providing an easy-to-use interface to the user, the second important task of the runtime is to orchestrate the execution of the accelerator instances on the FPGA.

As described in Section 11.4.1, accelerator instances on the FPGA are directly coupled to a dedicated HBM memory channel per instance, i.e., each accelerator instance only has access to a single HBM memory block. However, TaPaSCo currently does not support to split the device address space into distinct memory regions, so we cannot rely on TaPaSCo's memory management API to allocate and manage the HBM address space. Instead, our SPN runtime implements its own thread-safe device memory manager, which allows to manage the distinct HBM memory blocks separately. The device memory manager in our runtime supports allocation and freeing of memory blocks in a specific HBM block, making it possible to establish distinct address regions for each HBM block.

Prior work [11] also showed that overlapping the data-transfers between host and device with the execution on the accelerator can reduce overall execution time. To implement such a scheme, each compute job is broken down into multiple sub-jobs, according to a user-specified block-size. Each CPU thread then performs the same sequence of tasks: First, the data is transferred to the on-chip HBM. Then the SPN-accelerator is invoked and the CPU-thread waits for it to finish. As soon as the accelerator finishes the inference task, the CPU thread triggers the transfer of the results from HBM to the host.

By assigning multiple CPU threads to one accelerator instance on the FPGA, we can effectively overlap data transfers and computations, as one thread will be able to perform data transfers for block $n + 1$, while another thread is waiting for the FPGA accelerator instance to complete computation of block n.

In the prior work, up to four threads per SPN accelerator were used to achieve maximum throughput. In our current implementation, measurements have shown that the DMA over PCIe bandwidth is already fully utilized with just two threads per SPN accelerator.

11.5 EVALUATION

To evaluate how the use of HBM impacts SPN inference, we will first take a look at the hardware utilization and (potential) scaling capabilities of our approach. Afterwards we compare the results against our prior work focusing on AWS F1 [11], which mainly assesses the

performance of SPN inference in a cloud computing setting. For all benchmarks we rely on datasets from the well-known NIPS corpus[1].

11.5.1 *Resource Utilization*

Compared to our prior work, there are three significant changes which impact the resource utilization: 1) Due to the exclusive use of HBM, it is not necessary to include soft DRAM controllers in our design. The HBM controllers are hardened IP, which means they do not require logic resources. Conversely, using a soft DRAM controller requires a significant amount of FPGA-resources. 2) While we are using the same SPNs as [11], we exploit additional prior work, which made the internal arithmetic format more flexible w.r.t. to the bitwidth, and also optimized the arithmetic for the SPNs [17, 18]. 3) Our evaluation was performed using a Bittware XUP-VVH accelerator card, which features a Xilinx UltraScale+ VU37P FPGA. In comparison, Amazon AWS F1 instances feature a similar FPGA, which does not have HBM capabilities. Both FPGAs are from the UltraScale+ series, but the AWS FPGA has slightly fewer logic resources. Additionally, all designs targeting the F1 instances have to include a shell for the host interface, which also incurs a resource overhead.

In addition to these differences, the results provided by [11] use varying numbers of SPN accelerators and memory controllers. For a valid comparison, we initially limit the scope to benchmarks with four accelerator-instances with a corresponding memory channel each (i.e., up to and including NIPS40). To contrast these with our new HBM-capable architecture, we built corresponding designs that feature four accelerator-instances, each connected to a dedicated HBM channel. The results are shown in Table 11.1.

It is obvious that our new approach is more resource efficient in almost all resource types. Interestingly, the accelerators used in [11] generally require fewer LUTs used as Memory. The change to the custom floating point arithmetic here (originally developed in [17]) could explain this change. In terms of LUTs used as Logic, BRAM and DSPs, our work typically requires approx. 66% fewer resources. LUTs used as memory are slightly increased (except for NIPS40). The number of registers used here is roughly 50% less than in [11].

Overall, the resource requirements have vastly decreased in comparison to [11]. This opens up the potential to further replicate the accelerator to scale up. This allows us, e.g., to fit up to eight NIPS80 accelerators on the FPGA compared to only two in [11].

1 http://archive.ics.uci.edu/ml/datasets/bag+of+words

Table 11.1: Resource Utilization of the comparable NIPS-based benchmarks. "New" columns show the result of this work, while [11] refers to our prior work.

	kLUTs as Logic		kLUT as Mem		kRegs		BRAM		DSP	
Example	New	[11]	New	[11]	New	[11]	New	[11]	New	[11]
NIPS10	169.8	376.0	66.9	45.4	275.1	530.2	122	360	200	612
NIPS20	180.5	467.0	69.6	54.4	320.7	650.6	126	388	448	1356
NIPS30	230.9	577.3	70.4	62.6	354.4	765.4	122	364	696	2100
NIPS40	241.2	664.1	72.9	75.1	401.6	907.1	132	380	976	2940
Available	1304.0	1182.0	601.0	592.0	2607.0	2364.0	2016	2160	9024	6840

11.5.2 *Performance Scaling*

To describe the scaling of our architecture, we take a closer look at the very small NIPS10 benchmark. For each processed sample, the input consists of 10 single-byte values. The result is a single double-precision value. This means that each processed sample entails a total data transfer of 144 bits. Using a single SPN accelerator, the architecture is able to process 133,139,305 samples per second. Multiplying by the number of input and result bits per sample reveals that the accelerator requires 2.23 GiB/s of memory bandwidth. Given the HBM performance discussed earlier (c.f. Fig. 11.2), this shows that a single HBM channel should easily be able to provide the data required for a single accelerator. Hypothetically, linear scaling should be possible to at least 32 accelerators, due to the 32 HBM channels (and completely disregarding the limited logic resources).

To test this hypothesis, we ran multiple performance benchmarks for each of the benchmark SPNs and measured the end-to-end execution time required for computing inference over 100,000,000 samples using up to eight concurrent SPN cores, each controlled by up to two control-threads. From the results, we conclude that using more than one control-thread only improves performance for less than four accelerators. Thus, *all of the results* presented in this section are measured with only one dedicated control-thread per SPN accelerator. The corresponding benchmark results are visualized in Fig. 11.4.

If we look at the right side of Fig. 11.4, the scaling for NIPS10 is obviously slowing at five or more SPN accelerators. Adding additional accelerators after that point does not yield any significant performance improvements. Using five accelerators, we are able to process 614,654,595 samples per second, which in turn requires approx. 10.3 GiB/s of memory bandwidth. Due to the use of up to eight independent HBM channels with approx. 12 GiB/s each, the available memory bandwidth should not be an issue.

To further investigate this point we performed a separate set of benchmark runs. In the second run, we disregarded the host-to-device data-transfer times and only measured the on-device computation including the HBM accesses. The results are shown on the left in Fig. 11.4. It is clear that almost linear scaling is achieved for up to eight concurrent accelerators. This makes sense, since the batch-wise inference on SPNs is *embarrassingly parallel*. This trend is also likely to continue at least until all 32 HBM channels are used by at least one SPN accelerator. Unfortunately, scaling up that far is not possible due to the limited FPGA logic resources, as well as routing scarcity.

After examining these results, we conclude that the issue is caused by the host-to-device data-transfers, which are performed using DMA-transfers via the PCIe 3.0 x16 interface of the accelerator card.

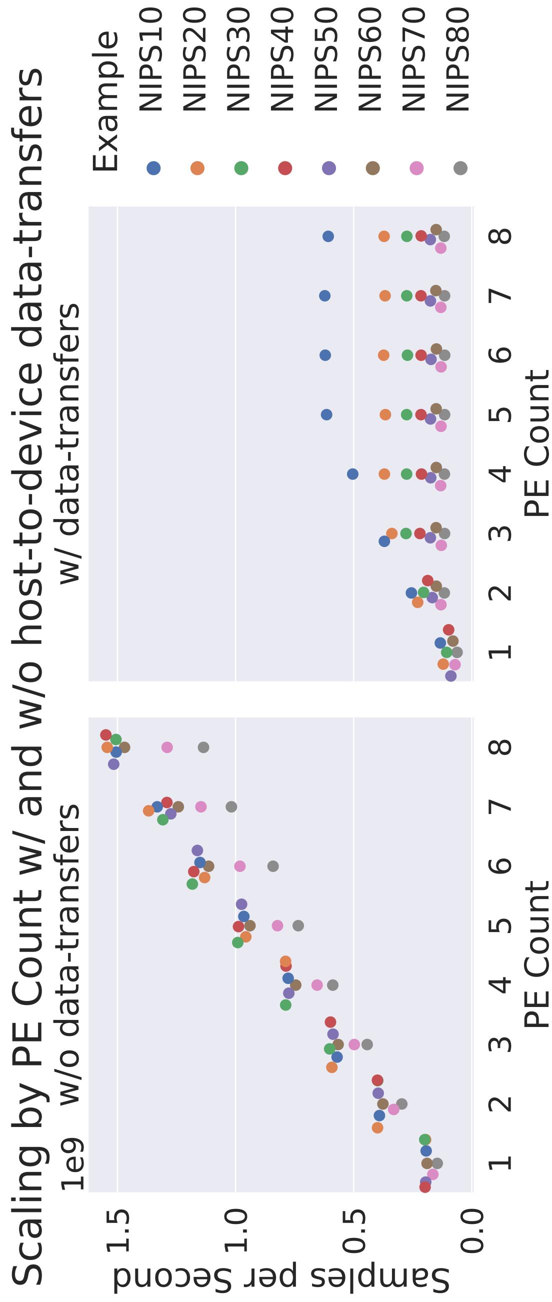


Figure 11.4: Comparison of peak performance in samples per second. On the left, host-to-device data-transfers are excluded to disregard a PCIe-based bottleneck. On the right, actual end-to-end performance is measured. Note that the inclusion of the data-transfer time leads to severely skewed scaling.

11.5.3 *Scaling Limitations*

The previous Section 11.5.2 shows that scaling is limited due to the host-to-device data-transfers as well as the available FPGA-resources. Disregarding these limitations, we want to give a perspective on the theoretical limitations of our approach focusing only on the HBM. According to the specification, the theoretical peak bandwidth of the HBM on the BittWare XUP-VVH platform is 460 GB/s (approx. 428 GiB/s). From the HBM benchmark shown in Fig. 11.2, we see that the practical performance is around 12 GiB/s per channel, assuming that that the blocks accessed are reasonably large.

Given the required data rates for the NIPS10 benchmark (144 bits per sample, 2.23 GiB/s), this means that a channel is easily able to accommodate at least four accelerators. Multiplying by the number of channels (32) would mean that overall, up to 128 NIPS10-accelerators could be used without any memory bandwidth limitations. The required memory bandwidth in that case would be $32 * 4 * 2.23$ GiB/s = 285 GiB/s, which is still well below the theoretical limit, as well as the practical limit ($32 * 12$ GiB/s = 384 GiB/s). Due to the independent nature of the HBM channels, it is relatively likely that this setup would actually allow linear scaling up to the practical limit of the HBM memory.

The HBM-scaling potential is further highlighted in Fig. 11.5. Using the data-sizes and samples per second for each benchmark, we calculated the required memory throughput of each of the benchmark SPNs. The resulting values are compared against the HBM throughput limitations. The comparison is drawn versus the single-channel result from our HBM benchmark shown in Fig. 11.2, the practical limitation imposed by 32 channels running at maximum channel throughput and the theoretical limit (as quoted by Xilinx). The vast memory bandwidth provided by HBM could theoretically allow the use of 64 accelerator-instances for all benchmarks, effectively boosting the current performance by up to 8x. For the smaller benchmarks NIPS10 and NIPS20, up to 128 instances could be served by the HBM, which in turn would double performance over 64 instances. While these values are currently out of reach, the improvements provided by the upcoming generations of PCIe will help improve the performance.

To put this into perspective, we look at NIPS80: Using 80 single-byte input values, we are able to process 116,565,604 samples per second. The input data alone requires a bandwidth of 8.7 GiB/s. When we consider the theoretical peak bandwidth of PCIe 3.0 x16, the one-directional theoretical limit is 15.754 GB/s (14.67 GiB/s), which is in practice never reached. For example, current PCIe-based DMA-engines like the Xilinx QDMA or Corundum [1] typically achieve speeds of 100 Gb/s which equates to 11.6415 GiB/s for single-direction transfers. The difference to our NIPS80 example can be explained by

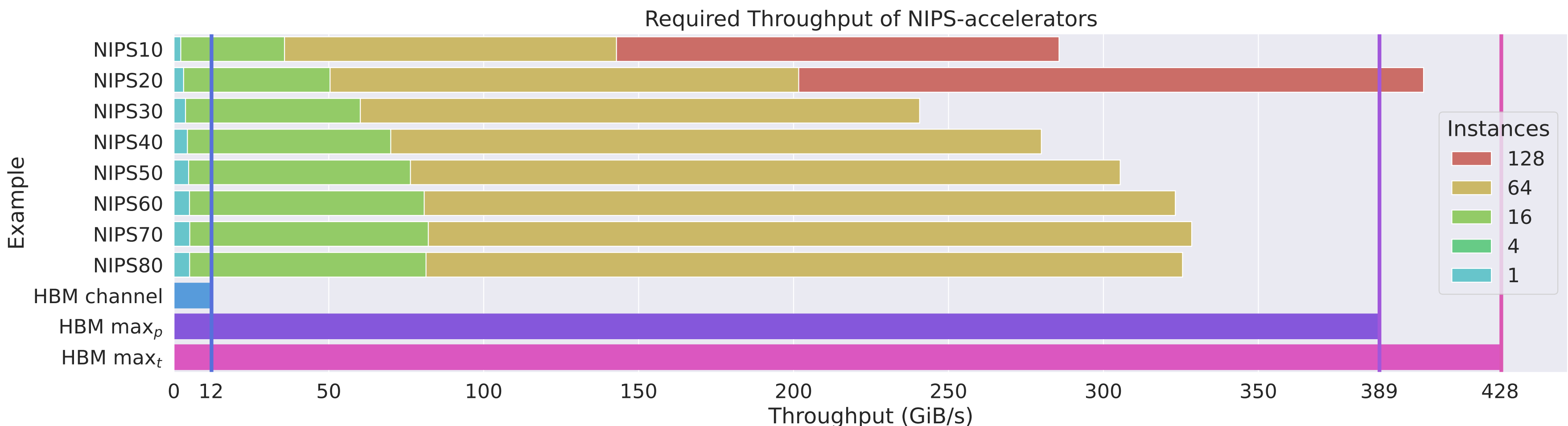


Figure 11.5: Scaling potential of the presented architecture under the assumption that logic resources are sufficient and host-to-device bandwidth is available. For each benchmark, we depict the required memory throughput depending on the number of instantiated SPN-cores. The throughput is compared against the maximum throughputs of a single HBM channel as measured in Fig. 11.2. Additionally, we compare against the practical maximum throughput scaled from our single-channel benchmarks (HBM $\max_p$), and the theoretical limit quoted by the vendor (HBM $\max_t$).

imperfect overlapping of the data transfers and the interference with the actual computation. Since the upcoming PCIe generations are specified to *double* the bandwidth with each generation, it is likely that corresponding DMA-engines will allow single-direction bandwidths of approx. 23 GiB/s, 46 GiB/s and 92 GiB/s for PCIe 4.0, 5.0 and 6.0 respectively. While this is still not comparable to the bandwidth of the on-chip HBM, it would definitively allow scaling much further.

In addition, this problem could also be circumvented by different approaches, where host-to-device data-transfers can be omitted due to shared memory, such as the Intel HARP prototype which unifies a high-performance FPGA with a server-grade CPU.

Last but not least, it is important to consider different approaches for delivering data to the SPN accelerators. In [3] for example, we used a streaming-based version of our accelerators to integrate them into a 100G network for in-network inference. The experimental results show that using a reasonable degree of replication, the SPN-accelerators are perfectly capable of performing inference at line rate. In light of the recent advancements in networking and shared memory systems, the potential of HBM becomes even more interesting as a reasonable option for buffering, especially when multiple 100G links are used to transport data in between multiple nodes.

11.5.4 *End-to-End Performance*

In the previous sections, we have discussed the performance results achieved in this work in the context of HBM performance and scaling properties. While the scaling potential is not fully exploited due to the bottleneck imposed by the host-to-device data-transfers, we still want to give a perspective on the overall end-to-end inference performance achieved with the HBM-based architecture.

To this end, we use the performance data **including** host-to-device data-transfers and compare it against the results reported by [11]. In this work, the AWS F1-based accelerator is evaluated against a Nvidia Tesla V100, as well as a 12-core Xeon E5-2680 v3 CPU. Both here and in [11], large inference runs are executed to measure the runtime. From the resulting total runtime and the known size of the datasets, the number of samples processed per second can be calculated. The resulting number of samples per second is shown in Fig. 11.6. The figure gives the best-case result for each target platform and each benchmark from this and the prior work [11].

From these results, it is clear that the Nvidia Tesla V100 is unsuitable for SPN inference due to its much lower overall performance. In contrast, the CPU baseline is able to outperform the AWS F1-based as well as the HBM-based implementation for the small NIPS10 benchmark. The reason for this is the lower compute intensity of smaller SPNs. Since NIPS10 just has a small number of nodes, the cost of the

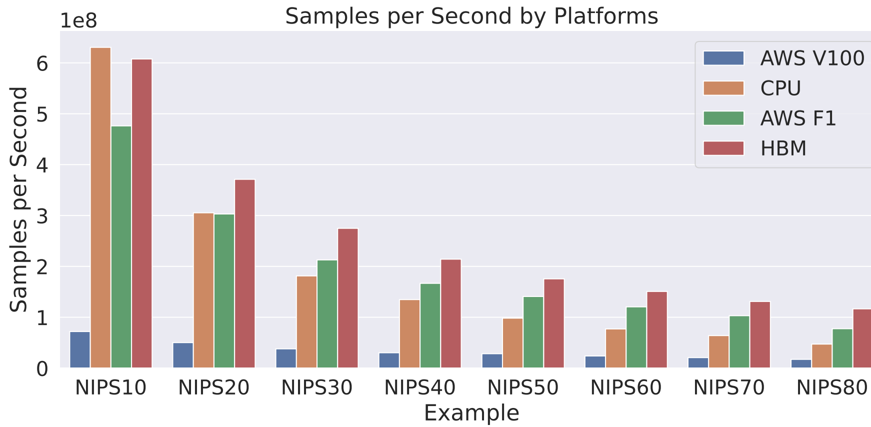


Figure 11.6: Peak performance measurements for the different benchmark SPNs on different target platforms. The number of samples per second is calculated from the end-to-end execution time. For AWS F1, V100 and HBM, host-to-device data transfers are *included* in the runtime.

data transfers outweighs the increased compute performance, which results in the CPU outperforming GPUs and FPGAs. However, the CPU's advantage vanishes for increasing SPN sizes: For NIPS20, our HBM-based implementation is already able to outperform the CPU by a speedup of **1.21x**. From NIPS20 to NIPS80 the advantage of the FPGA becomes even greater, yielding a maximum speedup of **2.46x** for NIPS80. The geometrical mean of the speedups is **1.6x** for the CPU. Regarding the V100, the maximum speedup of the HBM-FPGA is **8.4x**, and the geometrical mean of the speedups is **6.9x**.

In comparison to the previous FPGA-implementation, the speedup of using HBM is similar for almost all examples, and close to the geo.-mean speedup of **1.29x**. This is somewhat disappointing, given that our implementation here uses at least twice as many accelerator instances. This less-than-expected speedup can be explained by the scaling limitations due to the limited PCIe bandwidth, as discussed in Section 11.5.2. For the largest SPN (NIPS80), the prior work was not able to use more than two accelerator-cores (instead of four for the smaller benchmarks). In this case we achieve a speedup of **1.5x**.

In context of the adapted architecture from [3], we can get a perspective on the maximum performance of the NIPS80 accelerator: With the 99.078 Gbit/s peak throughput described there, and the 88 bytes of data per sample, we derive a theoretical peak performance of 140,748,580 samples per second. Comparing that to the measured peak performance of 116,565,604 samples per second we achieve in this paper, we see that the streaming-based architecture delivers about 17% increased performance. The reason for this is the much more streamlined architecture, which does not require any memory accesses. In addition, refresh cycles of the HBM also play a role at this level of

performance. Taking these factors into account, the HBM-based architecture is very close to its theoretical peak performance, which is capped by the maximum PCIe throughput. Lastly, it is important to realize that the HBM-based architecture targets a different use-case than the streaming-based one: While the in-network streaming implementation makes sense on a very large scale (i.e. data-centers), the HBM-based architecture could be used in smaller high-performance setups. This would also remove the necessity of costly 100G networking infrastructure associated with the streaming approach.

11.6 RELATED WORK

While SPNs are still a lesser known machine learning model, they are gaining traction in the field of machine learning. As discussed in Section 11.2.1, they have recently been used in the context of databases, specifically for cardinality estimation as well as approximate query processing [4]. To our knowledge, there is no prior work on accelerating SPN inference (independent of specific applications) apart from our own prior work. Our own prior work begins with [15], which introduced an automatic toolflow to map SPNs to the FPGA. In subsequent publications [17, 18], we looked into the impact of the arithmetic number formats. To this end, [18] introduced a custom logarithmic number system, which enables the computation of very small probability values, and decreases the number of hardware resources required. [17] introduces a customized floating point format, as well as posit number format based on PaCoGen [5]. In [17], the different number formats are optimized towards the SPN use-case and evaluated against each other. In [11], we adapted the original framework from [15] to the UltraScale+-based FPGAs in the Amazon AWS F1 instances. The evaluation showed that the reconfigurable cloud offers high inference performance without the need for expensive on-site FPGA-accelerator cards. Additionally, the F1 instances are able to outperform other state-of-the-art cloud-based hardware, such as a Xeon E5 CPU, and a Nvidia Tesla V100 GPU, for the more compute-intensive SPNs. In our most recent work [3], we adapted the SPN-accelerators to a streaming-based architecture, which in turn allowed us to integrate them into a 100G network for in-network processing of SPN inference. The corresponding work shows the potential of 100G networking for data-delivery to network-attached accelerators, as well as potential performance achievable using our SPN accelerators. It also shows inference throughput of 99.089 Gbit/s, coming very close to the practical limitations of 100G networking.

While there is (to our knowledge) no other work on hardware-acceleration of SPNs, there is similar work targeting Arithmetic Circuits (AC). Similar to SPNs, ACs are probabilistic graphical models. Both models share similarities and ACs can be transformed into SPNs

under certain conditions. Shah et al. [14] have presented a custom processor architecture based on ACs that is able to outperform the Nvidia Jetson TX2 embedded GPU by 12x.

Regarding the use of HBM, the available FPGA-specific research is still rather sparse. This is mainly due to the fact that HBM-enabled FPGA-accelerator cards are relatively new and typically come at a rather high cost. Despite this, there are two important works that explore the advantages and disadvantages of HBM. The work by Lu et al. [8] uses a number of micro-benchmarks to explore the different available memory technologies in recent FPGA-accelerator cards. Specifically, off-chip DRAM is compared against the on-chip HBM memory in a Xilinx Alveo U280 accelerator-card, which features a similar FPGA chip as the one used in our work. The paper goes into detail, how HBM and DRAM can best be used to achieve maximum bandwidth. The other work by Kara et al. [6] examines the use of HBM in the context of a columnar database. Using three database-specific workloads (selection, join, and stochastic gradient descent), the combination of FPGA and HBM is compared against a 14-core Xeon E5 and a dual-socket POWER9 system. In their work, they are able to outperform the server-grade CPU-based implementations by up to 12.9x for the join operation.

11.7 CONCLUSION

In this work we presented an improved accelerator architecture that exploits the parallelism of multiple HBM channels to speed-up inference on SPNs. While the overall performance of our adaption yields speedups of up to 1.5x over the prior work, as well as speedups of 2.46x and 8.4x over a data-center CPU and GPU respectively, the results still fell short of our expectations. We explored this issue and discovered host-to-device DMA transfers via PCIe to be the hard bottleneck.

From our experiments, we conclude that without the host-to-device data-transfers, and disregarding logic and routing resource limitations, almost linear scaling is possible for at least eight accelerators. This trend can likely be continued for up to 64 or even 128 accelerators. While our expectations for the use of HBM have not been met, the current implementation still outperforms prior implementations using the same approach, with CPU-inference of the small NIPS10 benchmark being the only exception. It is important to note that the accelerator card used, was attached using PCIe 3.0 x16. While this is the current de-facto standard, the first PCIe 4.0 devices are already available for end-users. Next-generation PCIe 5.0 and 6.0 devices are also planned to ship within the next two years. Given the ongoing effort in improving PCIe in the future, it is only a matter of time until the full potential of on-chip HBM can be fully exploited for even faster SPN inference. If we take other advancements like 100G networking into

account, it becomes clear that the data-delivery is a very important issue. Especially the combination of HBM and 100G networking could be very interesting for high-throughput data-processing in the context of machine learning and artificial intelligence.

REFERENCES

[1] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. "Corundum: An Open-Source 100-Gbps Nic." In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020.

[2] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. "A Survey of FPGA Based Neural Network Accelerator." In: *CoRR* (2017). arXiv: 1712.08934. URL: http://arxiv.org/abs/1712.08934.

[3] Marco Hartmann, Lukas Weber, Johannes Wirth, Lukas Sommer, and Andreas Koch. "Optimizing a Hardware Network Stack to Realize an In-Network ML Inference Application." In: *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2021.

[4] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, not from Queries!" In: (2019). arXiv: 1909.00607 [cs.DB].

[5] M. K. Jaiswal and H. K. H. So. "PACoGen: A Hardware Posit Arithmetic Core Generator." In: *IEEE Access* 7 (2019), pp. 74586–74601. ISSN: 2169-3536.

[6] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso. "High Bandwidth Memory on FPGAs: A Data Analytics Perspective." In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 2020.

[7] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *Applied Reconfig. Comp.* 2019.

[8] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. "Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking." In: *The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Association for Computing Machinery, 2021. ISBN: 9781450382182.

[9] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Floriana Esposito, Siraam Natarajan, and Kristian Kersting. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 2018.

[10] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. *SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks*. 2019. eprint: arXiv:1901.03704.

[11] Micha Ober, Jaco Hofmann, Lukas Sommer, Lukas Weber, and Andreas Koch. "High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud." In: *Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2019.

[12] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. *Probabilistic Deep Learning using Random Sum-Product Networks*. 2018. arXiv: 1806.01910 [cs.LG].

[13] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: a New Deep Architecture." In: *Proc. of UAI* (2011).

[14] N. Shah, L. I. Galindez Olascoaga, W. Meert, and M. Verhelst. "Acceleration of probabilistic reasoning through custom processor architecture." In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020.

[15] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *36th Intl. Conf. on Computer Design (ICCD)*. Oct. 2018.

[16] Lukas Sommer, Cristian Axenie, and Andreas Koch. "SPNC: An Open-Source MLIR-Based Compiler for Fast Sum-Product Network Inference on CPUs and GPUs." In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022.

[17] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. "Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs." In: *Intl. Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020.

[18] Lukas Weber, Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting, and Andreas Koch. "Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs." In: *Intl. Conference on Field-Programmable Technology (FPT)*. 2019.

Copyright Notice: © 2022 IEEE. Reprinted, with permission, from Lukas Weber et al. "Exploiting High-Bandwidth Memory for FPGA- Acceleration of Inference on Sum-Product Networks." In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 112–119. DOI: 10.1109/IPDPSW55747.2022.00028.

Part III

NEAR-DATA PROCESSING

12 NKV: NEAR-DATA PROCESSING WITH KV-STORES ON NATIVE COMPUTATIONAL STORAGE

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *nKV: Near-Data Processing with KV-Stores on Native Computational Storage* by *Tobias Vinçon, Lukas Weber, Arthur Bernhardt, Ilia Petrov, Andreas Koch* in *2020 16th International Workshop on Data Management on New Hardware (DaMoN)*. The contribution of the author of this thesis is summarized as follows.

» *As a co-leading author, Lukas Max Weber implemented FPGA-based hardware accelerators for the acceleration of GET, SCAN, and BC operations and integrated them into the COSMOS+ architecture. The nKV system, including NDP invocation and software-based parser and accessors, was designed and implemented by Tobias Vinçon. The experimental evaluation and the manuscript were joint work of Tobias Vinçon and Lukas Max Weber with support and feedback from Arthur Bernhardt, Ilia Petrov, and Andreas Koch.* «

ABSTRACT

Massive data transfers in modern key/value stores resulting from low data-locality and data-to-code system design hurt their performance and scalability. Near-data processing (NDP) designs represent a feasible solution, which although not new, have yet to see widespread use.

In this paper we introduce nKV, which is a key/value store utilizing *native computational storage* and *near-data processing*. On the one hand, nKV can directly control the data and computation placement on the underlying storage hardware. On the other hand, nKV propagates the data formats and layouts to the storage device where, software and hardware parsers and accessors are implemented. Both allow NDP operations to execute in host-intervention-free manner, directly on physical addresses and thus better utilize the underlying hardware. Our performance evaluation is based on executing traditional KV operations (*GET*, *SCAN*) and on complex graph-processing algorithms (*Betweenness Centrality*) in-situ, with 1.4×-2.7× better performance on real hardware – the *COSMOS+* platform [7].

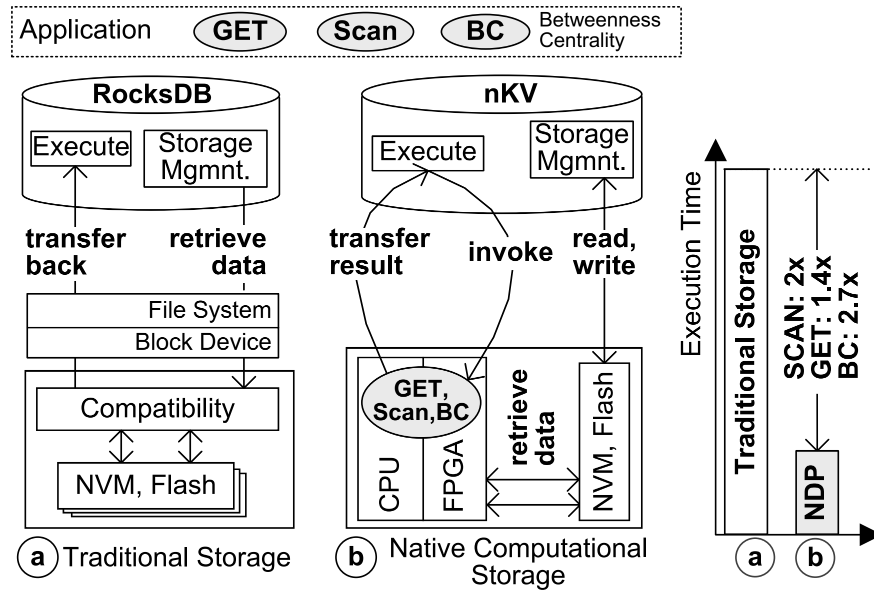


Figure 12.1: KV-Store transferring data along a traditional I/O stack (a); and (b) nKV executing operations in-situ on native computational storage.

12.1 INTRODUCTION

Besides substantial data ingestion, yielding an exponential increase in data volumes, modern data-intensive systems perform complex analytical tasks. To process them, systems trigger massive *data transfers* that impair performance and scalability, and hurt resource- and energy-efficiency. These are partly caused by the scarce bandwidth in combination with poor data locality, but also result from traditional (*data-to-code*) system architectures.

Near-Data Processing (NDP) is a *code-to-data* paradigm targeting in-situ operation execution. In other words, operations are executed as close as possible to the physical data location, utilizing the much better on-device I/O performance. NDP leverages several trends. Firstly, hardware manufacturers can fabricate *combinations of storage and compute* elements economically, and package them within the same device – so called NDP-capable *computational storage*. As a result, even commodity storage devices nowadays, have compute resources that can be effectively used for NDP, but are executing compatibility firmware (to traditional storage) instead. Secondly, with semiconductor storage technologies (NVM/Flash), the *device-internal* bandwidth, parallelism, and latencies are significantly better than the external ones (device-to-host). Both lift major limitations of prior approaches like ActiveDisks [1, 24] or Database Machines [5].

Wide-spread, high-performance persistent key-value stores like LevelDB or RocksDB [10] tend to rely on a traditional layered-storage stack (Fig. 12.1). It simplifies their architecture, allows for more flexi-

bility and eases data management and administration. However, layers within the DBMS (e.g. Storage Manager or access methods), but also underlying the file- and operating system encapsulate information and functionality necessary for the successful utilization of NDP techniques. *Firstly*, NDP operations executed on-device require the physical address ranges of the data to be processed. In traditional storage, address information is scattered along the layers of the storage stack (DBMS, File System, OS) and hidden behind layers of abstraction (Fig. 12.1). *Secondly*, NDP-operations need to navigate through and interpret the physical data on-device. To this end data formats and layout accessors are necessary on device. However, *data format definitions* are only available within the DBMS or sometimes within the application on top. Moreover, data layouts (page or record) and traversal methods for the data organization (files or LSM-trees) are typically hard coded in the DBMS and thus not available on device.

To address the above, in this paper, we present nKV, which is a key/value store utilizing *native computational storage* and *near-data processing* (Figure Fig. 12.1). nKV eliminates intermediary layers along the I/O stack (e.g. file system) and operates directly on NVM/Flash storage. nKV directly controls the physical data placement on chips and channels, which is critical for utilizing the on-device I/O properties and compute parallelism. Furthermore, nKV can execute various operations such as *GET* or *SCAN* or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM-cores and as hardware-software NDP (HW/SW-NDP) using corresponding FPGA-based accelerators. The necessary FPGA hardware is built in the form of simple processing elements that can be used to offload certain tasks from the ARM-cores. Under nKV we target *host-intervention-free* NDP-executions, i.e. the NDP-device has the complete address information, can interpret the *data format* and access the data in-situ without host interaction. To reduce data transfers nKV also employs novel *ResultSet*-transfer modes. nKV is resource efficient as it eliminates compatibility layers and utilizes freed compute resources for NDP. nKV performs 1.4×-2× better than RocksDB: *GET latency* – 1.4×; *SCAN* – 2×; *BC* execution time – 2.7×.

This paper is organized as follows. In the next section we describe the data organization of RocksDB and the challenges it poses to NDP. In Section 12.3 we describe the architecture of nKV and how those NDP-challenges are addressed in terms of interface extensions (Section 12.3.1), in-situ data processing (Section 12.3.2), as well as operations and algorithms (Section 12.3.3). The architecture of the underlying NDP hardware accelerators is described in Section 12.5. We discuss the experimental results in Section 12.6 and conclude in Section 12.8.

12.2 BACKGROUND

In contrast to traditional data organizations, where data is updated in-place, LSM-trees [22] have been proposed as an out-of-place update structure to tackle the sustained update and insertion rates of modern workloads and provide query capabilities at the same time. Classical LSM-trees [22] comprise multiple B-Tree-structured index components (C_0 to C_K, Fig. 12.2) that are stored in new locations and have constant size ratios $r = |C_{i+1}|/|C_i|, i \in [0, K)$. An insert or update operation hits the C_0 component that is located in memory. Once it reaches a size threshold, it is flushed to disk and is merged with the C_1 component. The merge processes gradually move data from C_0 to C_K, purge outdated KV-Pairs, reclaim space and indirectly ensure hot-cold data separation.

nKV builds on RocksDB [10], which introduces one independent LSM-Tree per column family to separate the access characteristics of different database objects. Modern LSM-Tree variants (surveyed in [20]) are multi-levelled. Modifications to an LSM-Tree are first placed in the main memory component C_0, which comprises a set of *MemTables* in RocksDB. These are realized as memory-efficient data structures such as SkipLists. Whenever a MemTable reaches a given size limit, it becomes *immutable* and a new one is created to accommodate further modifications. Later on, immutable MemTables are transformed into *Sorted String Tables (SST)* and flushed to the secondary storage (Fig. 12.2), whereas each LSM-tree component C_1..C_K comprises multiple SSTs. Thereby, the contained key-value pairs are placed into multiple *data blocks* in sort order of the key. Furthermore, an *index block* that comprises key-offset pairs pointing to each data block (a sparse index) is prepended. Index blocks reduce the access complexity to key-value pairs within the SST.

During the flush to C_1 no merge occurs for performance reasons. Consequently, overlapping key-value ranges of SSTs can occur (consider SST_{12}-SST_{1n}, C_1, Fig. 12.2). Merge steps to underlying layers $C_2 \ldots C_K$, called compactions, take either SSTs only on the level above or combine them with SSTs on the target layer, based on the given strategy (e.g. tiered or levelled). Either way, all KV pairs of the input SSTs are sorted, out-dated entries are pruned, and the results are stored in new SSTs on the target level (see dotted box, Fig. 12.2). Hence, key ranges in SSTs bellow C_1 do not overlap anymore. Yet, keys may appear on multiple levels with different values (consider $Key11$ or $Key70$), *to account for the temporal distribution of updates to a given key-value record*.

For instance $Key70$ has been updated multiple times: $Key70$ on C_1 is the most recent record and its existence invalidates $Key70$ on C_2 and C_3.

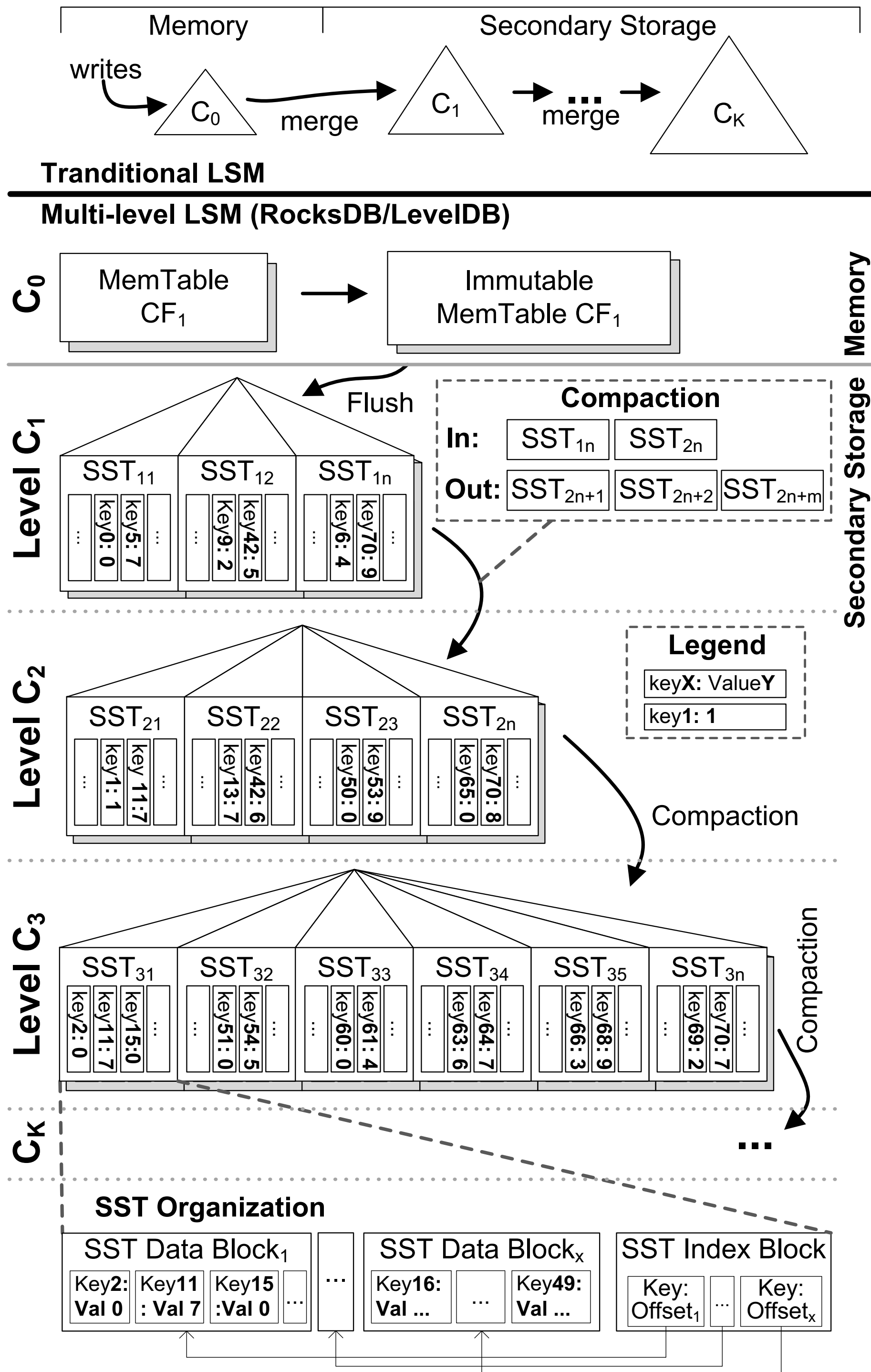


Figure 12.2: Conceptual organization of the multi-level LSM-Trees in RocksDB/LevelDB.

To *retrieve* a key-value record based on the key, the $GET(key)$ first traverses the MemTables and the immutable MemTables on C_0. If the respective key is not found, the index block of one or more SSTs in C_1 has to be read (as SSTs may overlap on C_1, but not on $C_2 ... C_K$). By parsing the key-offset pair, the data block, which might contain the key, can be identified and also has to be read from secondary storage. If the key is still not found, layers ($C_2 ... C_K$) have to be traversed similarly. Due to the data organization and the compaction process, a key can now reside only in a single SST per level. *Range scans* with or without key predicates behave similarly, but are more complex and are supported by other internal structures (like fence pointers). Consider $SCAN([Key68, Key70])$, which traverses all levels and retrieves $Key70$ from C_1, and $Key69$ and $Key68$ from C_3.

However, if a scan involves *value predicates*, e.g. $SCAN(0 \leq Val \leq 7)$, the only option is to iterate over the entire dataset, yielding a significant increase of I/O transfers, which in turn has enormous potential to be improved via NDP.

12.3 ARCHITECTURE OF NKV

Native computational storage. One of the underlying design principles behind nKV is that native storage enables efficient NDP (Fig. 12.1 and Fig. 12.3). In this sense nKV extends [28]. *Native storage* is storage that is operated without intermediary/compatibility layers of abstraction along the critical I/O path, and is directly controlled by the database. This means that nKV can directly operate on NVM/Flash storage using physical addresses and thus can precisely control physical placement of SST data. It is this physical placement that allows utilizing the on-device I/O bandwidth and the FPGA's compute parallelism.

nKV physically places *SST data blocks* and *SST index blocks* on different *LUNs* and *Channels* (see Fig. 12.2 and Fig. 12.4). This allows for reaching the internal bandwidth (Table 12.2) by requesting index and data blocks asynchronously and utilizing processing parallelism of FPGA-based processing elements (PEs). Besides, individual levels of

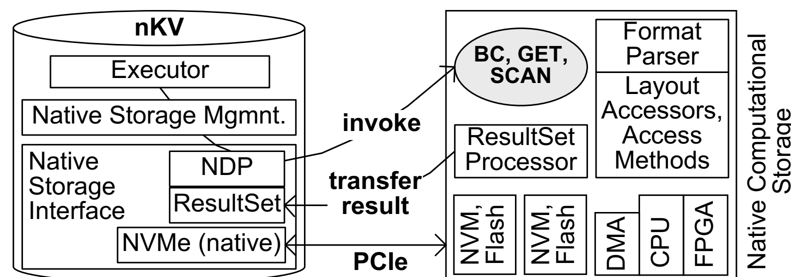


Figure 12.3: Architecture of nKV

the LSM-Tree are physically separated on different chips and LUNs to improve I/O throughput and parallelism since I/O-heavy compaction jobs do not block the entire device, reducing demand pressure on the bus.

Furthermore, nKV operates directly with physical addresses, to access (read or write) precisely the physical pages that are needed. This, in turn, is essential for reducing read- and write-amplification. Moreover, it inherently avoids costly host round-trips for logical-to-physical address translation. Native storage eliminates these *information hiding* effects incurred through layers of abstraction and thus simplifies the NDP-operations. Hence, native storage leads to leaner NDP-functionality.

Computation Placement. By using native computational storage, nKV can directly place computations on the heterogeneous on-device compute elements, such as ARM-cores or FPGA-based processing elements. nKV can execute various operations such *GET*, *SCAN* or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM-cores or with hardware support from the FPGA (cf. Section 12.5). The experimental evaluation indicates that some NDP operations such as *NDP_GET(key)* perform best on the ARM-cores, while other operations like *NDP_SCAN(value_condition)*, benefiting from parallelism, perform best on the FPGA. For its NDP-operations nKV utilizes *hardware/software co-design* to handle the proper separation of concerns and achieve best performance.

12.3.1 *NDP Interface Extensions*

NVMe support. nKV has a dedicated high-performance *user-space* and *in-DBMS* NVMe layer (Fig. 12.3). It is very lean and tightly integrated with the rest of nKV. The *native NVMe* integration can control multiple NVMe submission and completion queues either through dedicated threads or through the transactional context. Moreover, it reduces the I/O overhead as it allows the seamless creation of I/O and NDP tasks, the precise allocation of transfer buffers for the DMA engine, and prioritizable placement within the NVMe submission queues. The deep database integration additionally avoids expensive synchronization between user- and kernel-space, and shortens the I/O paths even further as no drivers are involved along the critical access paths. Internally, the native storage command set is translated to specific NVMe I/O and NDP tasks. Although these resemble the standardized NVMe commands, they define a new category - n over NVMe. In nKV, they can be scheduled either for *synchronous* or for *asynchronous* execution.

Command set. Besides the classical native storage interface, nKV introduces NDP-Extensions [28] in terms of a generic *NDP_EXEC()* command. It takes the following parameters, among others:

(i) *Command Identifier* – Unique identifier of the NDP function;

(ii) the *SearchKey* or *SearchKeyRange(s)* – for GET or SCAN;

(iii) the *ResultsSet Size*;

(iv) *AddressMappings* – these are ranges of physical addresses, where the physical data to be processed is located;

(v) *Min/Max Keys* – RocksDB supports a type of zone map range filter that can be used on device to skip processing some index/-data blocks;

(vi) *Miscellaneous* – command specific parameters such as data format definitions.

nKV composes the NVMe command based on the given parameters, current state and address information, and its transactional context. After placing it in the NVMe submission queue and DMA transferring the parameters to the device, the NDP command is then executed. The result set is handled by the ResultSet processor, which also observes the execution status. Finally the ResultSet is transferred to nKV by the DMA engine. An NDP-operation can invoke multiple low-level NDP commands synchronously or asynchronously.

12.3.2 *In-situ Data Access and Interpretation*

Under nKV, the NDP-device can interpret the *data format* and *access* the data without *host intervention* (synchronization with the host) [29]. To this end, nKV extracts definitions of the *Key- and Value-formats*, and passes them as input parameters to the NDP-commands. Moreover, the *data format* such as the *Key- and Value-formats* can be automatically extracted from the DB-catalogue (*system-defined*) or can be defined by the *application*.

nKV employs a thin on-device *NDP-infrastructure* layer that supports the execution and simplifies the development of NDP-operations (Fig. 12.3). It comprises *data format parsers* and *accessors* that are implemented in both *software* and *hardware* (Fig. 12.4). The in-situ *accessors* traverse and extract the contained sub-entities of the persistent data. Whereas, the in-situ *data format parsers* process the *layout* of each persistent entity, and extract the sub-entities by invoking further accessors (Fig. 12.4).

KV-Stores like LevelDB or RocksDB [10] organize the persistent LSM-Tree data into so called *String Sorted Tables (SST)* – see Section 12.2. To process a *GET(key)* request, for instance, nKV first identifies the respective *SST* and invokes an *NDP_GET* command with the corresponding physical address ranges, the respective *Key- and Value-formats* as well as further parameters. First, the *SST layout accessor* is invoked to access the data and the index blocks. Subsequently, the index block parser is activated to interpret the data and verify if the *key* is present and extract its

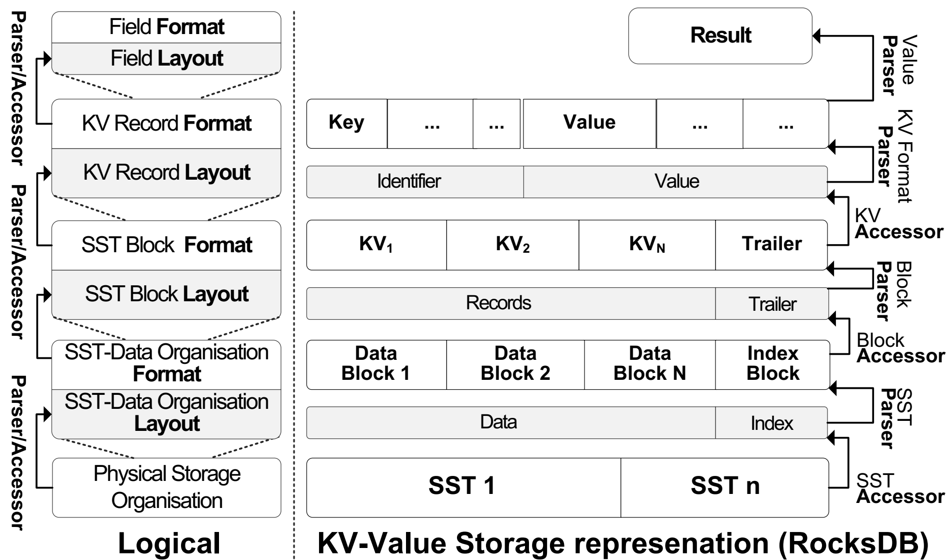


Figure 12.4: In-situ access and data interpretation in nKV, based on layout accessors and format parsers.

offset. If this is the case, data block accessor and parser are invoked to extract the *Key/Value entry*. In case of an *NDP_SCAN(key_val_condition)* operation, the KV accessor is subsequently invoked to extract it, followed by a *field* parser to extract its value and verify the *condition*. This way, nKV extends scans in classical KV-Stores. Typically, they are only able to process filter criteria on key-embedded attributes, but not filter predicates involving the value.

12.3.3 *Operations and Algorithms*

Lookup. KV-Stores offer fast (low-latency, high parallelism) retrieval of a value, based on its key, through the *GET(key)* operation. In nKV, this operation first performs a lookup within main-memory components (MemTables, Fig. 12.2) regardless of execution model.

If the search key is not found, the lookup will proceed scanning the deeper LSM-Tree levels by processing their indices first and eventually their associated data blocks. Both, index and data block scanning are I/Os intensive in the traditional stack (Fig. 12.1), while with NDP, these can be performed efficiently on-device.

Scan. As mentioned in Section 12.3.2, Scans can be performed either on Key- or Value-embedded attributes. The index blocks of the LSM-Tree might be leveraged to navigate to necessary data blocks for key-attribute scans, similar to the Lookup operation. However, there is no auxiliary structure to accelerate scans on value-embedded attributes. Either way, multiple data blocks have to be examined depending on the selectivity of the filter-predicate. Consequently, Scans usually

result in a high amount of data transfers, which NDP can significantly reduce.

User-defined Function: Betweenness Centrality. Many applications involve more complex algorithms. Such user-defined functions (i.e. Betweenness Centrality) can also be supported by nKV. The specific algorithm implemented within nKV relies on [6] and measures the degree, to which nodes stand among each other. The logic involves shortest-path searches over the given nodes and therefore results in a variety of lookups and scans involving random and sequential I/O.

12.3.4 *Data Consistency, Database Maintenance and NDP*

In parallel to the execution of NDP functionality, nKV allows the processing of database maintenance e.g. compactions. Such parallel operations might result in new data or even changes to the LSM-Tree. Yet, as nKV's NDP-operations are executed on a snapshot of the physical data, concurrent modifications do not effect its consistency. This can be ensured by firstly, the underlying mechanism of Copy-on-Write (CoW), secondly the precise placement through the native storage interface, and thirdly, the direct control of the physical GC by nKV. Moreover, nKV requires no on-device bad-block re-mapping like other native storage management solutions [4], since bad-block management and wear-leveling are handled directly within the database engine [23]. Thus native storage management [23] leverages the the above issues by DBMS managed physical-to-logical address mapping and data placement.

12.3.5 *Result Set Handling*

Unnecessary data transfers may occur depending on how the result of an NDP-operation is transferred back. Therefore, ResultSet management additionally helps to avoid unnecessary stalls of processing resources. nKV aims to reduce the data transfer overhead caused by a Volcano-style *record-at-a-time* model. Instead it aims to bulk-transfer the *ResultSet*. The former is simpler, but leads to more frequent shorter burst transfers causing bus overhead. The latter results in fewer, but longer bursts leveraging the throughput-optimized PCIe.

Each NDP call in nKV defines a maximum ResultSet size as a parameter. The NDP ResultSet-Processor (Fig. 12.3) allocates on-device resources for it: either in DRAM, or if the contention is high, it allocates a temporary region on Flash. As long as the actual result size does not exceed the predefined one, the ResultSet is sent back as bulk DMA-transfer, to leverage the full performance of the DMA engine. Alternatively, it may be pipelined to the next NDP-operation. Furthermore, nKV has a built-in extension mechanism that in the worst case

may preemptively request more physical space from the DBMS, as it manages the logical-to-physical address mapping [23].

12.4 HARDWARE-ARCHITECTURE

The *COSMOS+* platform [7] is a PCIe-based extension-card. It contains all the required hardware-modules to function as a regular NVMe-based SSD. It can be fitted with up to 2 DIMM-extensions containing Flash modules. The available Toggle-NAND Flash-extensions can be configured in SLC or MLC mode. In SLC mode, each cell stores a single bit, while in MLC mode two bits are stored in each cell. In comparison, data retrieval is faster and simpler in SLC mode, which in turn offers higher performance. Additionally, in SLC mode cells can be programmed and erased more often. In this work, they are configured as SLC with 16 dies organized in two channels.

The main computational engine of the *COSMOS+* platform is a Xilinx Zynq-7000 SoC (XC7Z045-3FFG900) that combines two 667 MHz ARM Cortex-A9 cores with an FPGA (Fig. 12.5). In the *COSMOS+* platform [7], the FPGA-portion is used to implement the required SSD-infrastructure. This infrastructure is made up of two separate domains: The first one is responsible for accessing the flash memory. It comprises one or many Tiger4-controllers with corresponding low-level flash interfaces. For each channel, a distinct Tiger4-controller, as well as a low-level interface, is required.

The second domain contains primarily an NVMe-Core, which allows access from the host to the device via the NVMe interface. The NVMe-Core also wraps the actual low-level PCIe-interface.

Both of these domains are running at different clock-frequencies. While the flash domain uses a 100 MHz clock, the NVMe-Core is running at 250 MHz. When planning to extend this platform, the following aspects are relevant:

1) The amount of resources on the FPGA-portion (PL) of the SoC is limited. While the platform can be fitted with more flash-DIMMs, this also requires more flash controllers. This in turn impacts the resource requirements. In this work, one flash-DIMM is used with 2 flash controllers. While this limits the available flash memory and the corresponding parallelism, it also frees up resources for different use (i.e. computational processing elements).

2) Since the different domains are running at different clock-frequencies, the extensions should be able to run at the same clock-frequencies. In the case of the *COSMOS+* platform, this is not a huge problem, since most hardware-accelerators can run at 100 MHz and can therefore reside in the flash-domain.

A simplified view of the architecture is depicted in Fig. 12.5. It also includes the nKV hardware *extensions* described in this work (Section 12.5).

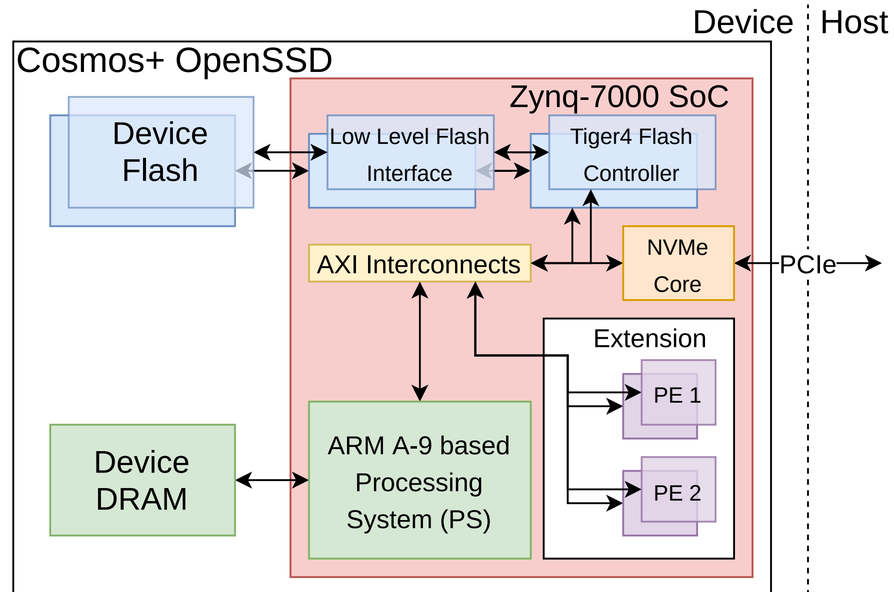


Figure 12.5: A simplified view of the architecture of the *COSMOS+* OpenSSD [7], including the proposed extension.

12.5 HARDWARE-ACCELERATION

Using the baseline architecture (Fig. 12.5), specific processing elements (PEs) are implemented, allowing to move computation from software running on the ARM-cores to the programmable logic of the SoC, potentially improving latency, throughput, and available parallelism. The PEs are written in Chisel3 [3] using a relatively simple architecture that can be subdivided into four distinct domains (cf. Fig. 12.6):

The control-domain consists of a register file, holding a number of control registers, which are accessible using an AXI4-Lite interface. The corresponding addresses are mapped into the address space of the processing system (PS), so that the ARM-cores can read and write these registers and thereby control and configure the PE. The control registers hold the required parameters for the functionality provided by the processing elements (e.g. the memory addresses of the input and output). In addition, the signaling to the ARM-cores is also done using these registers. One register indicates whether the PE is busy, while another can be used to trigger the execution.

The memory-domain contains a load and a store unit. These are connected to the PSs DRAM-interface, allowing the PE to access the device DRAM. Both the load and the store unit perform data transfers in chunks of 32 KB, which corresponds to the size of a single data-block in our RocksDB-configuration. The transfers are performed using AXI4 bursts to maximize throughput. The data-width of the AXI4-Bus is 64 bits and the AXI burst length is 16. Generally, longer bursts allow higher throughput, due to the sequential access pattern.

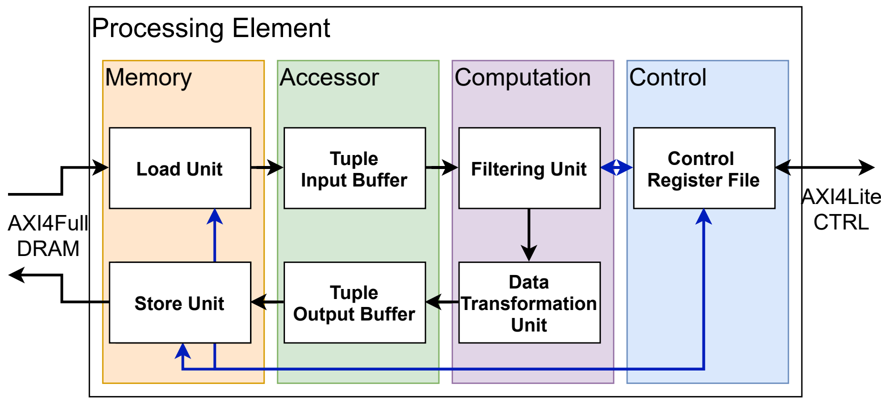


Figure 12.6: The overall Microarchitecture of the proposed Parser Processing Elements.

Unfortunately, the Zynq-7000 family only supports a maximum burst length of 16.

The accessor-domain is responsible for converting the different data granularities (64 bit words vs tuples) between the memory and the computational domain. The tuple input buffer will buffer incoming 64 bit data words from the load unit, until a complete tuple is available. This will then be passed to the filtering unit. Similarly, the tuple output buffer will receive a resulting tuple and split it into words of 64 bits to allow transfer to memory via the store unit. In this context, a tuple corresponds to a key-value pair (kv-pair).

The computational domain is comprised of two distinct modules. The first one is the filtering unit, which accepts single kv-pairs as input. Depending on the control registers, the filtering unit will then pass on matching kv-pairs to the data transformation, while non-matching ones are discarded. In the current implementation, the filtering unit can be configured to apply a single predicate on a kv-pair. This is done using three parameters: the column selector ($i \in [0, n-1]$, where n is the number of distinct data-fields), the compare operator ($op \in \{nop, =, \neq, >, \geq, <, \leq\}$) and a reference value (c) to compare against. Considering a kv-pair $t = (t_0, t_1, t_2, \ldots, t_{n-1})$, the following expression is evaluated: $r = t_i \; op \; c$. If r is true, the kv-pair will be passed on, else it will be discarded.

The last module transforms the data into the required output format. This corresponds to a projection of tuples and allows the automatic removal of RocksDB-metadata or unnecessary tuple elements. The transformed tuple is passed back to the accessor-domain, to be stored back to the device DRAM. The complete microarchitecture of the PEs is also depicted in Fig. 12.6.

Building atop of the baseline architecture of COSMOS+ (Fig. 12.5), we developed an extended architecture, which contains additional processing elements. In particular, we built two different processing elements for the specfic evaluation dataset (the database-of-research-

Table 12.1: FPGA-Resource Utilization of the Baseline and extended Architectures, including hierarchical utilizations of relevant sub-modules.

	Slices		BRAM		DSPs	
	abs.	%	abs.	%	abs	%
Baseline	14544	26.61	78	14.31	0	0
Tiger4	8174	5.81	15	2.75	0	0
NVMe-Core	4312	7.89	29	5.32	0	0
LL Flash	475	0.87	5	0.92	0	0
Extended	35667	65.26	101	18.53	0	0
paper-PE	33103	15.14	0	0.0	0	0
ref-PE	4012	1.84	0	0.0	0	0
Available	54650	100.00	545	100.00	900	100

papers): One for the data of the paper themselves (paper-PE), and another one for the data of the references (ref-PE). Initial experiments showed, that the paper-PE can process a 32 KB block of data faster than the two Tiger4s controllers are able to provide it (due to the flash latency). Thus, we employ a single paper-PE in the final architecture. For the handling of the paper references in the database, much more data has to be processed multiple times. In this case, the flash latency becomes less relevant, since the reference data is cached in the on-device DRAM and does not have to be fetched from flash memory each time. Thus, the architecture can keep multiple ref-PEs busy. To keep the interconnects balanced, we opted for seven ref-PEs, yielding a total of eight PEs (including the single paper-PE). Generally, it would be possible to increase the number of PEs, but all active PEs compete for access to the on-device DRAM. Thus, there will be a point of diminishing returns considering overall throughput as soon as the full memory bandwidth is saturated. Instead of replicating PEs for more throughput, it might be more reasonable to use multiple different PEs to increase flexibility of the hardware acceleration.

The baseline and extended hardware designs were synthesized and implemented using Xilinx Vivado v2019.1. The resulting resource utilizations are reported in Table 12.1, both in terms of absolute numbers, as well as relative to the resources available on the Zynq 7045 chip. The baseline results indicate that the Zynq has many spare resources. Even though small, a large fraction of its hardware resources are unused. The main reason for this lies in the flash-configuration. For a platform with two flash-DIMMs and the full parallelism, eight flash controllers and flash interfaces are needed. In our design, we only use one flash-DIMM with two controllers/interfaces, which vastly reduces the resource-requirements.

These free resources are then leveraged by our extended architecture to offload computations from the ARM-core to the FPGA. In doing

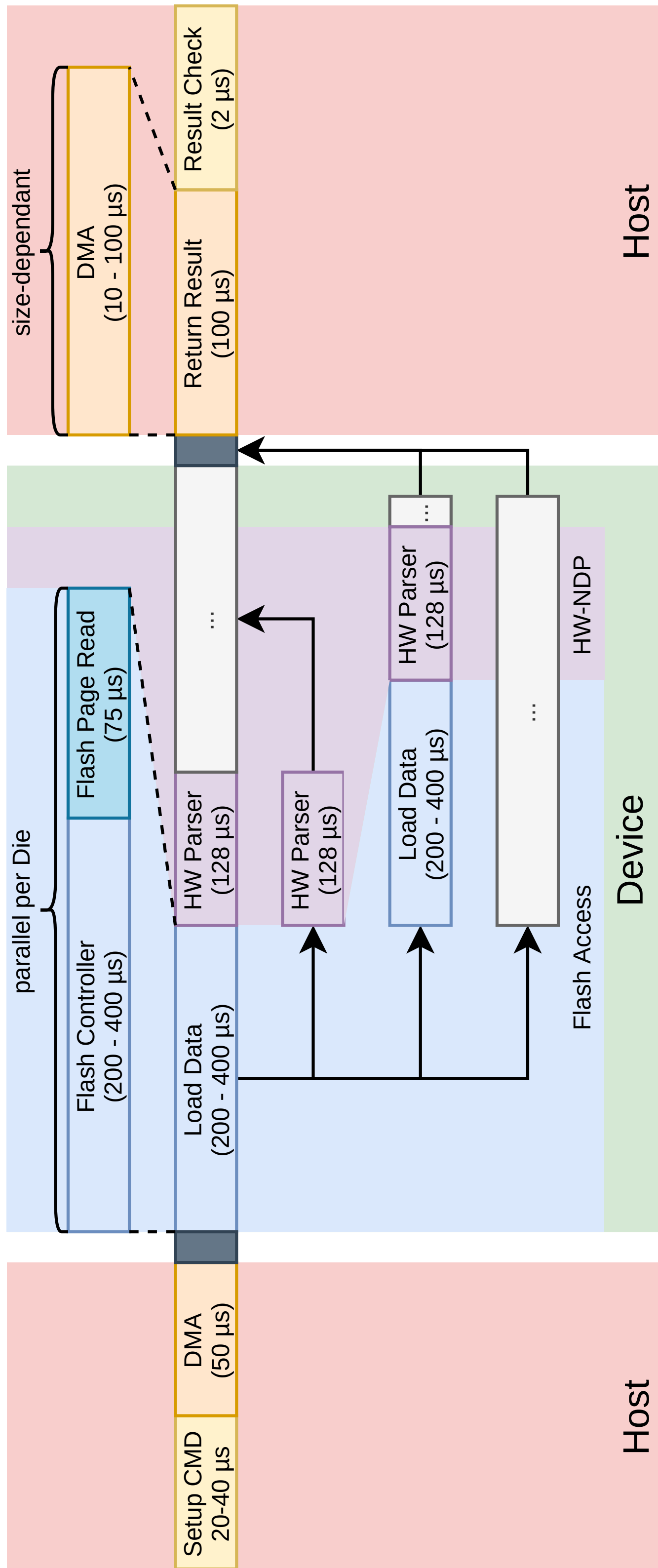


Figure 12.7: Break-Down of Execution Times within the NDP Stack with HW support.

so the hardware accessors and format parsers can be instantiated multiple times. In fact, nKV uses two different kinds of parsers with up to seven instances.

Another interesting point is the vast difference in resource requirements between the paper-PE and the ref-PE. The reason for this lies in the different sizes of the parsed kv-pairs. The kv-pairs processed by the paper-PE are 136 bytes each, while the kv-pairs processed by the ref-PE are merely 36 bytes each. Apart from the data-size, the number of distinct data fields also plays an important role, due to the number of required comparators.

Finally, there is a complete lack of DSP utilization. DSPs are hard-wired special-function slices which provide fast arithmetic and logical operations, that are typically relevant in the context of digital signal processing. For our work, DSPs could be interesting for arithmetic comparisons as well as pattern recognition.

For future extensions of this work, the above could be exploited to reduce Slice-utilization, or to implement more complex functionality within the PEs.

12.6 EVALUATION

For the evaluation, the *COSMOS+* board [7] (see Fig. 12.5) is attached over PCIe 2.0 x8 as an NVMe block device supported by Greedy FTL to realize traditional storage. The host server is equipped with a 3.4 GHz Intel i5, 4GB RAM and runs Debian 4.9 with ext3. We configure both RocksDB [10] and COSMOS+ [7] with a 5MB cache. COSMOS+ is directly mapped into the userspace and controlled by *native NVMe*.

We evaluate nKV on a *2.4GB* research paper graph dataset from Microsoft Academic Graph [27]. It comprises approx. *48 million* Key/Value-pairs: *3.8M* papers, *40M* references, *18K* venues, and *4.2M* authors. For each experiment, we report the average execution times of three cold test runs. The baseline for our experiments is RocksDB using block-device storage (*Blk*) on top of *GreedyFTL* and *ext3*. Performance results of GET(key), SCAN(value_predicate) and BC are reported for three different stacks: *Blk* as baseline, software NDP (*NDP:SW*) on the ARM and FPGA-assisted NDP (*NDP:SW+HW*).

12.6.1 *Low-level Flash Properties*

Physical data placement and the on-device Flash characteristics play an essential role in nKV. The following Table 12.2 shows the on-device latency and bandwidth, measured by directly issuing page reads to the Flash Management Unit. The level of parallelism is controlled by data placement on either different Channels, LUNs or both. While a single page-read takes approx. 300 µs, careful data placement on Channels and LUNs reduces latency down to 94 µs with full paral-

Table 12.2: Flash Latencies and Bandwidth (BW) of the *COSMOS+ OpenSSD* for different levels of parallelism.

	Pages	**Parallelism**	**Duration per Page [¯s]**	
	1	1 Ch. 1 LUN	300.00	
	4	2 Ch. 2 LUN	113.50	
	8	2 Ch. 4 LUN	*94.12*	
Access	**Pages**	**Parallelism**	**BW [MB/s]**	**IOPS**
Random	1500	1 Ch. 1 LUN	52	3000
	1500	2 Ch. 1 LUN	102	6000
	1500	1 Ch. 8 LUN	108	6000
	1500	*2 Ch. 8 LUN*	*213*	*13000*
Seq.	*640*	*2 Ch. 8 LUN*	*217*	*13000*

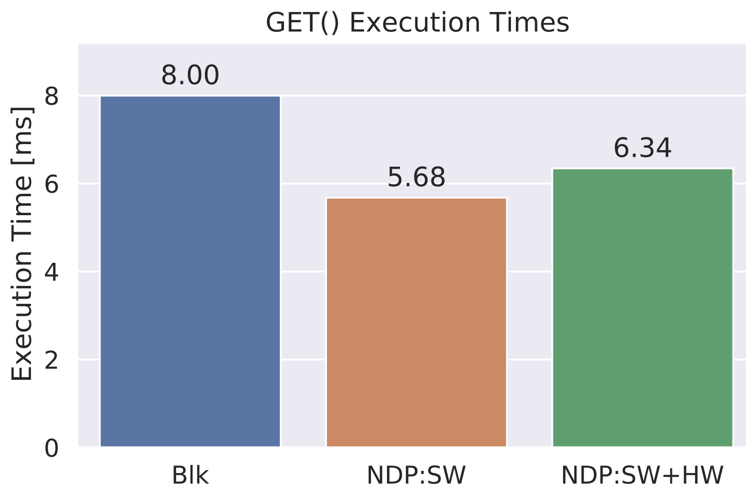


Figure 12.8: GET execution times for Blk, NDP:SW and NDP:SW+HW.

lelism (Table 12.2). However, an upper limit of around 217 MB/s can be observed for sequential and random workloads, which is due to the bus limitations of COSMOS+.

12.6.2 *Experiment 1: Lean Native Stack*

One important conceptional characteristic of NDP with nKV is the removal of traditional compatibility layers to simplify the access stack. To verify this property, we execute a *GET(key)* command. We compare the results of our baseline (Blk) against nKV with software NDP (NDP:SW), and nKV with Parsers-PE support (NDP:SW+HW) – Fig. 12.8.

nKV utilizes the native data placement and **improves the round-trip time by 1.4**× due to *native NVMe* and mapping the device into its userspace. This reduces the execution time from 7.9 ms to 5.7 ms, as

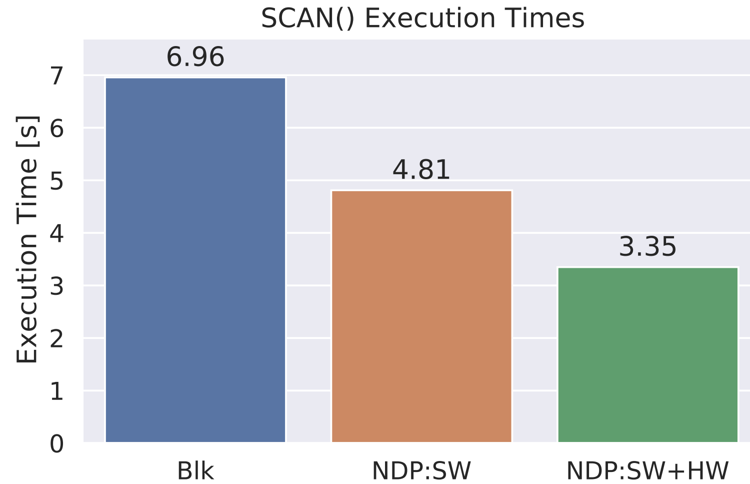


Figure 12.9: SCAN execution times for Blk, NDP:SW and NDP:SW+HW.

shown in Fig. 12.8. Interestingly, there is no benefit from hardware PEs since the gains from concurrent executions are limited due to the sequential nature of first reading and then processing Flash data.

12.6.3 *Experiment 2: Data Transfer Reduction*

While the relatively simple GET-operation does not benefit from the hardware PEs, this changes for bigger and more complex operations like the SCAN. In addition to the vastly reduced latency, the use of NDP has additional effect on the overall system. While all three implementations read similar amounts of data from flash (492.3 MB $\pm$ 0.2 MB due to caching), the required DMA data transfers vary. For NDP-operations, a single DMA transfer is required to push down the additional parameters. We draw the following conclusions. Firstly, efficient ResultSet handling reduces the transfer overhead by employing large bulk DMA transfers. Secondly, due to on-device filtering the amount of data to be transferred also decreases depending on the predicate selectivity.

The execution time is reduced by **more than 2x** (from 6.96 s to 3.35 s) as shown in Fig. 12.9.

12.6.4 *Experiment 3: Native Computational Storage*

Native Computational Storage plays an essential role for nKV. Especially with complex graph analysis operations like Betweenness Centrality (BC) the concepts of native data placement, flash parallelism and computation placement can be leveraged to improve the performance. An execution on a smaller graph, benefits the software implementation. For a large number of edges, the complexity is high and multiple HW

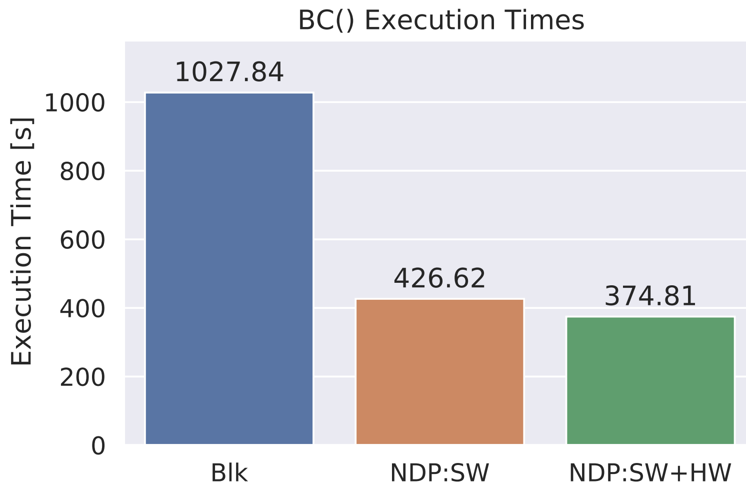


Figure 12.10: Betweenness centrality (BC) execution times for Blk, NDP:SW and NDP:SW+HW.

Parsers can be utilized to improve performance. With a total of 7 HW Parsers nKV achieves **2.7x speed-up** for 2.037.755 edges (Fig. 12.10).

12.6.5 *Experiment 4: Execution Parallelism*

In large graph processing the number of applied HW Parsers is important to balance between FPGA utilization, memory bandwidth limitations and performance. nKV allows to configure the number of HW Parsers individually for each NDP operation. In Fig. 12.11 BC is executed with 2.037.755 edges using a different number of ref-PEs. Clearly, increasing the number of PEs per operation, yields better speed-ups. Using seven PEs instead of three PEs results in a **speed-up of 1.25x**. While the data suggests that more parsers are better, it is important to note that all instances compete for DRAM accesses. Thus, adding more parsers will yield diminishing returns due to memory contention and increased randomness in the memory access patterns.

12.7 RELATED WORK

The Near-Data Processing is deeply rooted in *database machines* [5] developed in the 1970s-80s or Active Disk/IDISK [1, 17, 25] from the late 1990s. Besides dependence on proprietary and costly hardware, the I/O bandwidth and parallelism are claimed to be the limiting factors. While not surprising, given the characteristics of magnetic/mechanical storage combined with Amdahl's balanced systems law [11], this conclusion needs to be revised. Storage devices built with modern semi-conductor storage technologies (NVM, Flash) are offering high raw bandwidths with high levels of parallelism *on-device*.

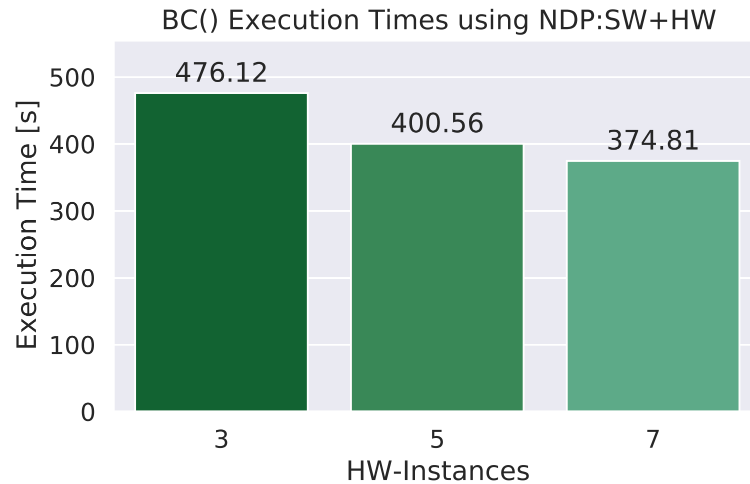


Figure 12.11: Betweenness centrality (BC) execution times for for NDP:SW+HW using 3, 5 and 7 instances of the ref-PE hardware parser.

With the advent of Flash technologies and reconfigurable processing elements, Smart SSDs [9, 16, 26] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [30]. Biscuit [12] is a timely proposal for a general NDP framework. JAFAR [2, 31] is one of the first systems to target NDP for DBMS (column-store) use, whereas [15, 19] target joins besides scans. The use of NDP in the realm of KV-Stores has been investigated in [8, 18]. Kanzi [13], Caribou [14] and BlueDBM [21] are RDMA-based distributed KV-Stores investigating node-local operations.

Much of the prior work on persistent KV-Stores and NDP focusses on *bandwidth* optimizations. NoFTL-KV [28] addresses the problem of *write-amplification*. The NDP extensions demonstrated by nKV target the *read-amplification*, *latency improvements* and *computational storage*.

12.8 CONCLUSION

In this paper we introduced nKV – a key-value store designed for native computational storage and near-data processing. nKV controls physical data placement directly and hence the on-device I/O parallelism. Along the same lines, nKV can place NDP operations on different compute elements on device (ARM or novel FPGA PEs) and also configure the hardware per operation accordingly, e.g. the number of hardware parsers used. Both placement methods impact the performance of NDP operations, GET is faster on the ARM, while SCANs are faster with hardware support. nKV is based on the principle of explicit cross-layer data formats, hence hardware or software layout

accessors and format parsers are deployed an can be used for different operations.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for the valuable comments, which significantly improved the quality of the paper. This work has been partially supported by *BMBF PANDAS – 01IS18081C/D*; *DFG Grant neoDBMS – 419942270*; *HAW Promotion, MWK, Baden-Würrtemberg, Germany*.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS 1998*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. "Chisel: Constructing hardware in a Scala embedded language." In: *Proc. DAC 2012*. 2012.

[4] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. "LightNVM: The Linux Open-Channel SSD Subsystem." In: *Proc. FAST 2017*. 2017.

[5] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: ed. by A. R. Hurson, L. L. Miller, and S. H. Pakzad. 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

[6] Ulrik Brandes. "A Faster Algorithm for Betweenness Centrality." In: *Journal of Mathematical Sociology* (2001).

[7] *COSMOS Project Documentation*. http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Technical_Resources. OpenSSD Project. Jan. 2019.

[8] Arup De, Maya Gokhale, and et. al et. "Minerva: Accelerating Data Analysis in Next-Generation SSDs." In: *Proc. FCCM 2013*. 2013.

[9] Jaeyoung Do, J. Patel, D. DeWitt, and et. al et. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD 2013*. 2013.

[10] Facebook. *RocksDB*. https://github.com/facebook/rocksdb. 2020.

[11] Jim Gray and Prashant J. Shenoy. "Rules of Thumb in Data Engineering." In: *Proc. ICDE 2000*. 2000.

[12] Boncheol Gu, Andre S. Yoon, and et al. et. "Biscuit: A Framework for Near-Data Processing of Big Data Workloads." In: *Proc. ISCA 2016*. June 2016.

[13] Masoud Hemmatpour, Mohammad Sadoghi, and et al. "Kanzi: A Distributed, In-memory Key-Value Store." In: *Proc. Middleware 2016*. 2016.

[14] Zsolt István, David Sidler, and Gustavo Alonso. "Caribou: Intelligent Distributed Storage." In: *Proc. VLDB 2017*. 2017.

[15] Insoon Jo and et al. et. "YourSQL : A High-Performance Database System Leveraging In-Storage Computing." In: *Proc. VLDB 2016*. 2016.

[16] Yangwook Kang, Yang-suk Kee, and et al. "Enabling cost-effective data processing with smart SSD." In: *Proc MSST 2013*. May 2013.

[17] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. "A Case for Intelligent Disks (IDISKs)." In: *SIGMOD Rec.* (1998).

[18] Jungwon Kim and et al. "PapyrusKV: A High-performance Parallel Key-value Store for Distributed NVM Architectures." In: *Proc. SC 2017*. 2017.

[19] Sungchan Kim, Sang-Won Lee, Bongki Moon, and et al. "In-storage Processing of Database Scans and Joins." In: *Inf. Sci.* (2016).

[20] Chen Luo and Michael J. Carey. "LSM-based storage techniques: a survey." In: *The VLDB Journal* 29.1 (2020), pp. 393–418.

[21] Sang-woo Jun Ming, Arvind, and et al. "BlueDBM: An Appliance for Big Data Analytics." In: *Proc. ISCA* (2015).

[22] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)." In: *Acta Inform.* (1996).

[23] Ilia Petrov, Andreas Koch, Sergey Hardock, Tobias Vincon, and Christian Riegger. "Native Storage Techniques for Data Management." In: *Proc. ICDE* (2019).

[24] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. "Active disks for large-scale data processing." In: *Computer (Long. Beach. Calif).* 34.6 (2001), pp. 68–74.

[25] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. "Active Storage for Large-Scale Data Mining and Multimedia." In: *Proc. VLDB 1998*. 1998.

[26] Sudharsan Seshadri, Steven Swanson, and et al. "Willow: A User-Programmable SSD." In: *USENIX, OSDI* (2014).

[27] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. "An Overview of Microsoft Academic Service (MAS) and Applications." In: *Proc. WWW 2015*. 2015.

[28] T. Vincon, S. Hardock, C Riegger, J. Oppermann, A. Koch, and I. Petrov. "NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management." In: *Proc. EDBT 2018*. 2018.

[29] Tobias Vincon, Arthur Bernhardt, Lukas Weber, Andreas Koch, and Ilia Petrov. "On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems." In: *Proc. HardBD @ ICDE 2020*. 2020.

[30] Louis Woods, J. Teubner, and G. Alonso. "Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances." In: *Proc. SIGMOD 2013*. 2013.

[31] Sam Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. "Beyond the Wall: Near-Data Processing for Databases." In: *Proc. DAMON* (2015).

Copyright Notice: Material from: Tobias Vinçon et al. "NKV: Near-Data Processing with KV-Stores on Native Computational Storage." In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 2020. DOI: 10.1145/3399666.3399934.

13 NKV IN ACTION: ACCELERATING KV-STORES ON NATIVE COMPUTATIONAL STORAGE WITH NEAR-DATA PROCESSING

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *nKV: Near-Data Processing with KV-Stores on Native Computational Storage* by *Tobias Vinçon, Lukas Weber, Arthur Bernhardt, Ilia Petrov, Andreas Koch* in *2020 16th International Workshop on Data Management on New Hardware (DaMoN)*. The contribution of the author of this thesis is summarized as follows.

» *As a co-leading author, Lukas Max Weber contributed the NDP functionality using FPGA-based hardware accelerators and integrated them into the architecture of the COSMOS+. The design and implementation of the nKV system, including the NDP invocations of software-based accessors for the GET, SCAN, and BC operations, was contributed by Tobias Vinçon, who also implemented the demonstration walk-through and user interface. The manuscript was primarily written by Tobias Vinçon with the support and feedback of the other co-authors, including Lukas Max Weber.* «

ABSTRACT

Massive data transfers in modern data-intensive systems resulting from low data-locality and data-to-code system design hurt their performance and scalability. Near-data processing (NDP) designs represent a feasible solution, which although not new, has yet to see widespread use.

In this paper we demonstrate various NDP alternatives in nKV, which is a key/value store utilizing *native computational storage* and *near-data processing*. We showcase the execution of classical operations (*GET*, *SCAN*) and complex graph-processing algorithms (*Betweenness Centrality*) in-situ, with 1.4×-2.7× better performance due to NDP. nKV runs on real hardware - the *COSMOS+* platform.

13.1 INTRODUCTION

Near-Data Processing (NDP) is a *code-to-data* paradigm targeting in-situ operation execution, i.e. as close as possible to the physical data location, utilizing the much better on-device I/O performance. NDP leverages several trends. Firstly, hardware manufacturers can fabricate

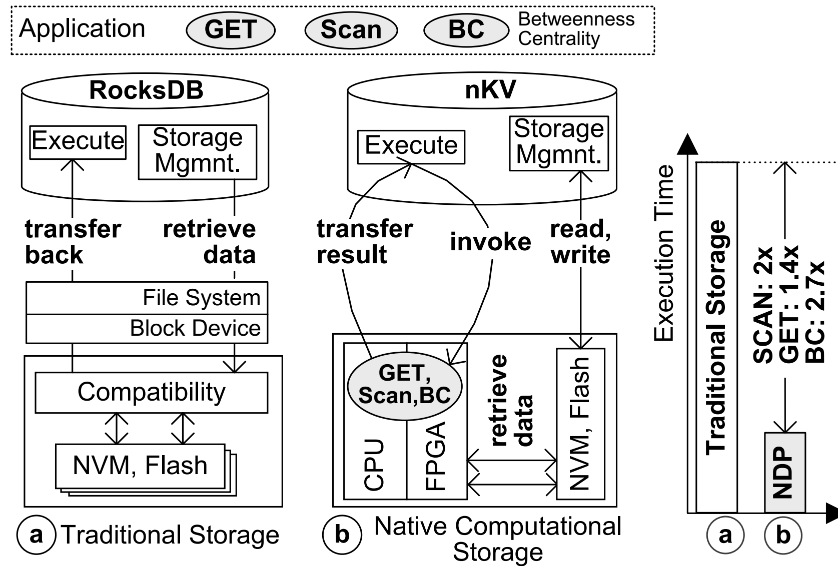


Figure 13.1: KV-Store transferring data along a traditional I/O stack (a); and (b) nKV executing operations in-situ on native computational storage.

combinations of storage and compute elements economically, and package them within the same device – so called NDP-capable *computational storage*. As a result, even commodity storage devices nowadays have compute resources that can be effectively used for NDP, but are executing compatibility firmware (to traditional storage) instead. Secondly, with semiconductor storage technologies (NVM/Flash) the *device-internal* bandwidth, parallelism, and latencies are significantly better than the external ones (device-to-host). Both lift major limitations of prior approaches like ActiveDisks or Database Machines.

In this paper, we demonstrate nKV, which is a RocksDB-based key/-value store utilizing *native computational storage* and *near-data processing* (Fig. 13.1). nKV eliminates intermediary layers along the I/O stack (e.g. file system) and operates directly on NVM/Flash storage. nKV directly controls the physical data placement on chips and channels, which is critical for utilizing the on-device I/O properties and compute parallelism. Furthermore, nKV can execute access operations like *GET* or *SCAN*, or more complex graph processing algorithms such as *Betweenness Centrality* as *software NDP* on the ARM cores or with FPGA hardware support (NDP:HW+SW). Under nKV we target *intervention-free* NDP-execution, i.e. the NDP-device has the complete address information, can interpret the *data format*, and access the data in-situ (without any *host interaction*). To reduce data transfers nKV also employs novel *ResultSet*-transfer modes. nKV is resource efficient as it eliminates compatibility layers and utilizes freed compute resources for NDP.

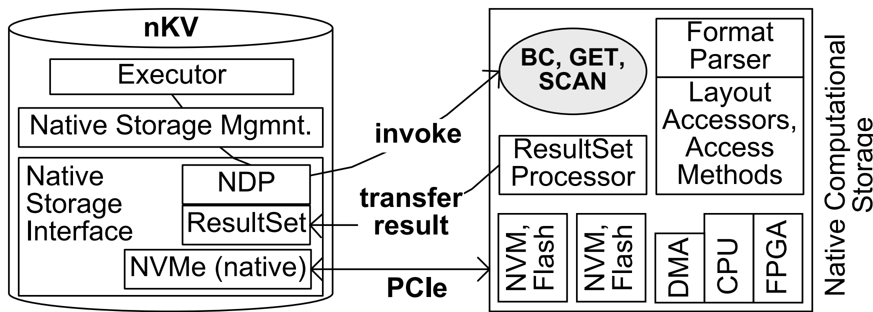


Figure 13.2: Architecture of nKV

We demonstrate nKV for the use-case of a database of research papers, and on a *2.4GB* graph dataset with *48 million* KV-pairs. Our demonstration scenarios involve interacting with the paper DB, browsing and analyzing it: (a) *Analysis scenario (BC)*: verifies if the 10-year best paper award was awarded the most prominent paper from 10 years ago and offers some unexpected insights; (b) *Latency-based (GET)*: we let the audience pick a paper from the BC ResultSet and display its details; (c) *Bandwidth-based (SCAN)*: we retrieve other papers from same Venue/Author/Year. nKV performs 1.4×-2× better than RocksDB: *GET latency* – 1.4×; *SCAN bandwidth* – 2×; *Betweenness Centrality* – 2.7×.

13.2 ARCHITECTURE OF NKV

This section offers a brief overview of the key architectural modules of nKV. More details are provided in [16].

NDP Interface Extensions. nKV defines NDP-Extensions besides the native storage interface. Furthermore, nKV has a dedicated high-performance *in-DBMS* NVMe layer (Fig. 13.2). It does not rely on an NVMe kernel driver, but is deeply integrated in the DBMS and hence runs in *user-space*. The *native NVMe* integration reduces the I/O overhead, as it avoids expensive switches between user and kernel space (drivers), and shortens the I/O even further, as no drivers are needed. *This lean stack improves execution times for I/O and NDP, especially for short-running calls e.g. GET.*

Computation Placement. By using native computational storage, nKV can place computations directly on the heterogeneous on-device compute elements, such as ARM CPUs or the FPGA. nKV can execute various operations such as *GET* or *SCAN*, or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM cores, or with *hardware* support from the FPGA. nKV demonstrates that hardware implementations alone cannot reach the best performance as pure software implementations do not. For its NDP-operations nKV utilizes *hardware/software co-design* to handle the proper separation of concerns and achieve best performance.

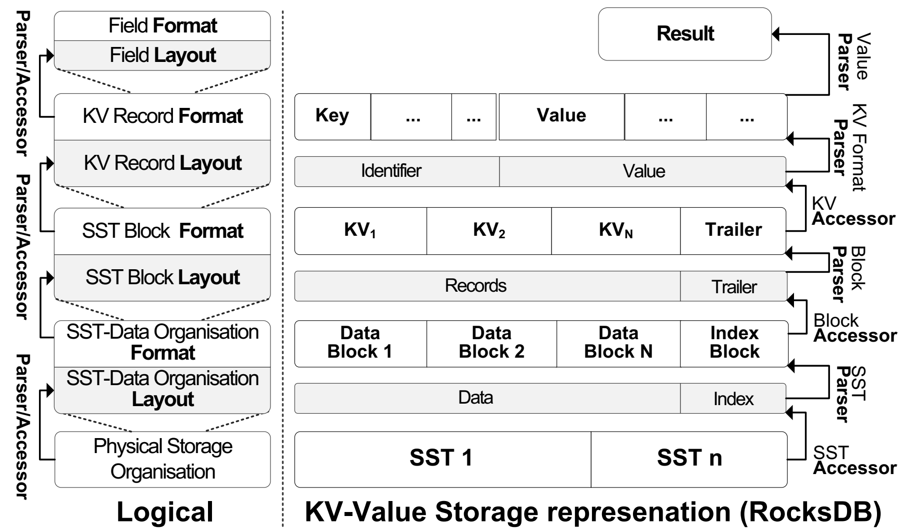


Figure 13.3: In-situ access and data interpretation in nKV, based on layout accessors and format parsers.

In-situ data access and interpretation. Under nKV the NDP-device can interpret the *data format* and *access* the data without *host intervention*. To this end, nKV extracts definitions of the *Key- and Value-formats* [15]. These are then passed as input parameters to NDP-commands. Moreover, the *data format* such as the *Key- and Value-formats* can be automatically extracted from the DB catalogue (*system-defined*), or can be defined by the *application*.

nKV employs a thin on-device *NDP-infrastructure* layer that supports the execution and simplifies the development of NDP-operations (Fig. 13.2). It comprises *data format parsers* and *accessors* that are implemented in both *software* and *hardware* (Fig. 13.3). The in-situ *accessors* are used used to traverse and extract the contained sub-entities of the persistent data. Whereas, the in-situ *data format parsers* process the *layout* of each persistent entity, and extract the sub-entities by invoking further accessors (Figure Fig. 13.3).

KV-Stores like LevelDB or RocksDB organize the persistent LSM-Tree data in to so called *Sorted String Tables (SST)*. To process a *GET(key)* request, for instance, nKV first identifies the respective *SST* and invokes an *NDP_GET()* command with the physical address ranges (of these SSTs), the respective *Key- and Value-formats* as well as further parameters. First, the *SST layout accessor* is invoked to access data and index blocks. Subsequently, the index block parser is invoked to interpret the data, verify if the *key* is present, and extract its location. If this is the case, the data block accessor and parser are invoked to extract the *Key/Value entry*. In case of an *NDP_SCAN(key_val_condition)* operation, the KV accessor is subsequently invoked to extract it, followed by a *field* parser to extract its value and verify the *condition*. The result are massive I/Os since especially SCANs must retrieve a huge number of data blocks.

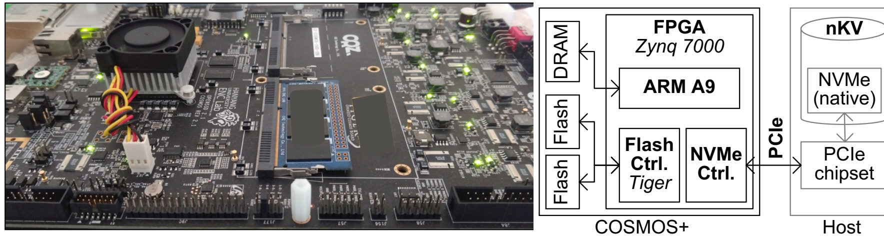


Figure 13.4: COSMOS+ and the Demonstration Setup

Native computational storage. To make efficient use of the on-device I/O nKV extends [14] and employs *native storage* (Fig. 13.1 and Fig. 13.2). This way it eliminates intermediary layers along the critical I/O path like the file system, and can operate directly on NVM/Flash storage using physical addresses. nKV can therefore precisely control physical placement of SST data, which is critical for utilizing the on-device I/O properties and compute parallelism. I.e. nKV physically places *SST data blocks* and *SST index blocks* on different *LUNs* and *Channels* to utilize the on-device parallelism and lower the processing latency. This accelerates especially the demonstrated I/O-intensive operations SCAN and BC significantly. *Native storage* is essential for reducing read- and write-amplification, and also for executing NDP-operations avoiding information hiding through these layers of abstraction.
ResultSet Handling. nKV aims to bulk-transfer the *ResultSet* of an NDP-Operation to avoid the data transfer overhead caused by a *record-at-a-time* model. Thus nKV materializes the ResultSet, partially or fully, depending on the NDP operation. It is then DMA-transferred with multiples of the COSMOS+'s DMA-engine transfer unit (4KB).

13.3 DEMONSTRATION WALK-THROUGH

Demo Setup.

The demonstration setup comprises a desktop PC as host equipped 3.4 GHz Intel I5 CPU, 4 GB RAM, connected to *COSMOS+* via NVMe over PCIe (Fig. 13.4). The COSMOS+ [2] has a Zynq 7045 SoC with an FPGA, two 667 MHz ARM A9 CPU Cores and an MLC Flash module configured as SLC. We configure both RocksDB and COSMOS+ with 5 MB cache.

We demonstrate nKV on the use-case of a database of research papers, and on a rather smaller *2.4GB* dataset due to practical run-time constraints of the demo. This graph dataset includes *48 million* Key/Value-pairs, comprising approx. *3.8M* papers, *40M* references, *18K* venues, and *4.2M* authors. *BC* operates on a graph with varying number of relevant edges: from *2.5K* to *2 million*. The audience will browse and analyze the paper set using a GUI (Fig. 13.5), triggering different operations on the paper graph in different scenarios.

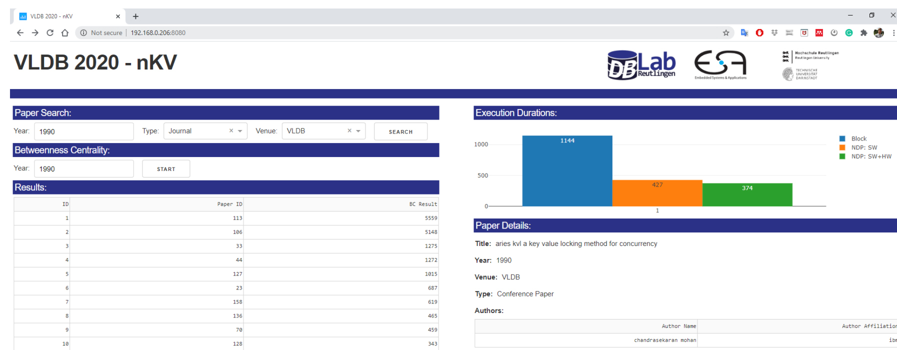


Figure 13.5: Interactive GUI.

13.3.1 *Demonstration Walk-Through*

1. Complex Graph Analysis – *BC*. The demo starts by letting the audience pick a *DB conference venue* and an *year* (e.g., *VLDB, 2000*). Subsequently, nKV executes *Betweenness Centrality* to determine the most prominent paper from that year. The audience can then verify if that paper had indeed been awarded the *10-year best paper award* ten years later. Expect some unexpected(!) insights.

Under the hood, nKV executes a complex NDP operation pipeline, comprising a SCAN followed by a BC. Based on the audience selection, nKV first filters out the relevant papers and references by running a *SCAN* and applying *val_condition* on the *values* of all paper KV-pairs. This is only possible since the data formats are available in-situ, and the *format parses* and *layout accessors* execute on-device. The intermediary result is materialized on-device, which is essential for such NDP-pipelines. Subsequently, BC is executed on the intermediary results. nKV switch between software NDP or software/hardware NDP. We demonstrate how the hardware accessors and parsers can be instantiated multiple times, and run in parallel on the FPGA yielding best results.

Observation: nKV executes NDP-pipelines and complex operations in-situ. Given the high parallelism and significant compute intensity, NDP:SW+HW yields best results.

2. Latency – *GET*. After the *BC* analysis, the audience can interactively pick a paper from the BC ResultSet and have its details displayed.

Under the hood, the NDP execution of GET is performed in SW and in NDP:SW+HW. Since only a single NDP_GET() is executed at a time, nKV utilizes native data placement, but not the on-device parallelism.

Observation: Latency-critical operations are 1.4× faster and best results are achieved with NDP:SW, closely followed by NDP:SW+HW (Fig. 13.7).

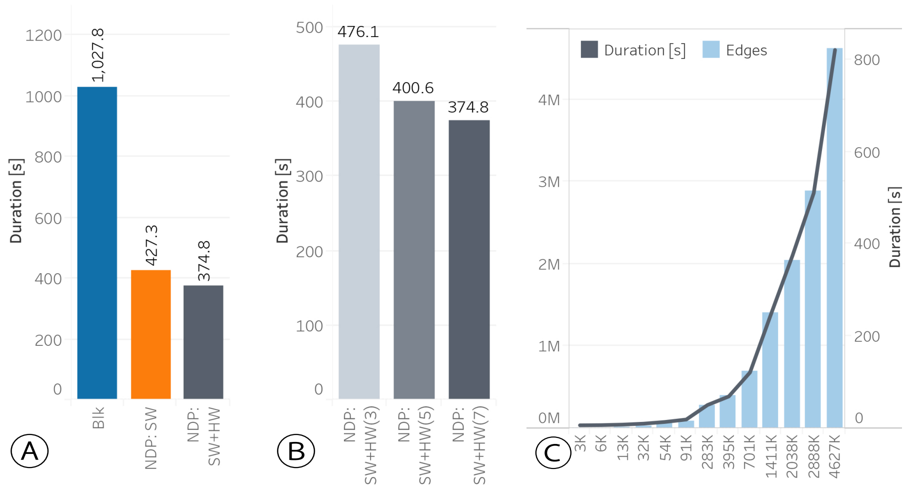


Figure 13.6: Betweenness Centrality: (A) BC on different stacks; (B) BC with different levels of parallelism; (C) BC execution time vs number of relevant edges (complexity).

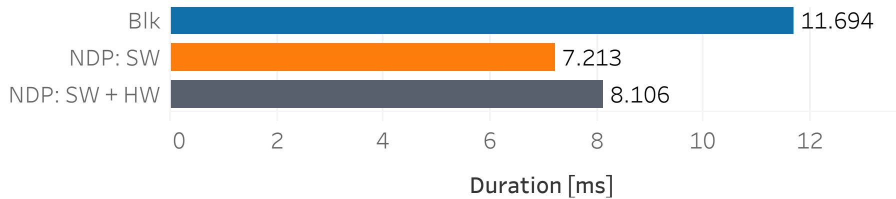


Figure 13.7: GET Latencies on different stacks.

3. Bandwidth – *SCAN*. After the audience has been presented the details of a paper (previous scenario), they can opt for retrieving other papers from the same Venue/Author/Year.

Under the hood, this results in an NDP *SCAN(value_condi- tion)*. The operation is performed with different selectivities and different result set sizes, based on the audience selection (Fig. 13.8). Importantly, the selection condition is on the value, which requires NDP format parsers and layout accessors to be evaluated in-situ. Conversely, the *Blk* RocksDB stack transfers the entire data to the host, to interpret the values there, apply the *val_condition*, and eventually discard most of the data. Fig. 13.9 shows the extra read volume transferred by the *Blk* to perform the same SCAN.

Observation: Bandwidth-critical scan and selection operations require I/O bandwidth and high hardware parallelism. Hence, using NDP:SW+HW yields the best performance and outperforms the traditional stack by 2×.

4. Parallelism and Native Computational Storage Last but not least we execute BC again, however this time we demonstrate the effect of *configurable parallelism* in native computational storage, whenever nKV executes a complex operation (Section 13.3.1).

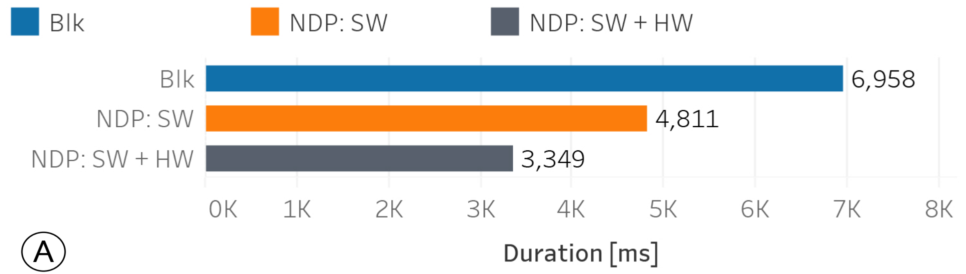


Figure 13.8: SCAN performance: (A) SCAN on different stacks

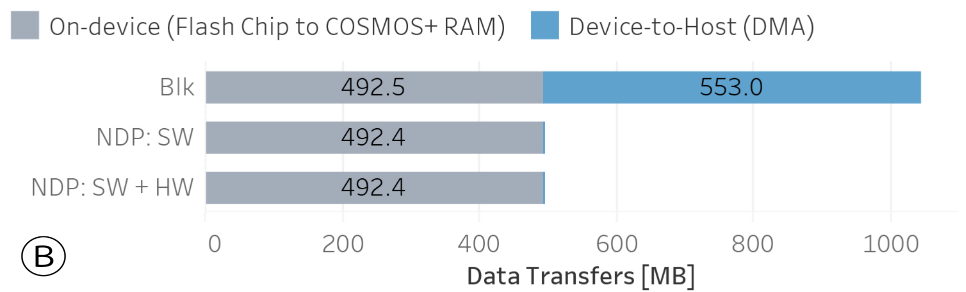


Figure 13.9: SCAN performance: (B) Data Transfer Volume

nKV can configure the degree of parallelism required by each NDP-operation. While the amount of compute parallelism is limited for NDP:SW, as there are few ARM cores, the same does not apply to the FPGA. As described in Section 13.2, there can be *multiple parallel instances* of the hardware accessors and parsers on the FPGA. These are relatively space-efficient, as 16 instances fit even into the small Zynq 7045 FPGA. Interestingly, operating with the maximum available parallelism does not always yield the the best results (Section 13.3.1).

Observation: nKV can employ the FPGA for NDP:SW+ HW, increasing the level of computational storage parallelism. However, this capability only translates into performance benefits for *complex* operations.

13.4 RELATED WORK

The Near-Data Processing approach is deeply rooted in well-known techniques such as *database machines* or Active Disk/IDISK. With the advent of Flash technologies and reconfigurable processing elements Smart SSDs [4, 8, 13] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [17]. JAFAR [1, 18] is one of the first systems to target NDP for Column-stores use, whereas [7, 10] target joins besides scans. Recently, Samsung announced its KV-SSD [11]. The use of NDP in the realm of KV-Stores has been investigated in [3, 9]. Kanzi [5], Caribou [6] and BlueDBM [12] are RDMA-based distributed KV-Stores investigating node-local operations.

Much of the prior work on persistent KV-Stores and NDP focuses on *bandwidth* optimizations. NoFTL-KV [14] addresses the problem of *write-amplification*. The NDP extensions demonstrated by nKV target the *read-amplification*, *latency improvements* and *computational storage*.

13.5 CONCLUSION

We demonstrate nKV, which is a key/value store utilizing *native computational storage* and *near-data processing*. We showcase the execution of classical operations (*GET*, *SCAN*) and complex graph-processing algorithms (*Betweenness Centrality*) in-situ, with 1.4×-2.7× better performance due to NDP. nKV runs on real hardware - the *COSMOS+* platform. nKV utilizes the the available I/O and compute parallelism on the native computational storage through direct data and operation placement. Complex operations (BC, SCAN) benefit from it, whereas others (GET) benefit from software NDP.

ACKNOWLEDGMENTS.

This work has been partially supported by *BMBF PANDAS – 01IS18081 C/D*; *DFG Grant neoDBMS – 419942270*; *HAW Promotion* and *KPK Service Computing MWK, Baden-Würrtemberg, Germany*.

REFERENCES

[1] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[2] *COSMOS Project Documentation*. http://www.openssd-project.org. OpenSSD Project. Jan. 2019.

[3] Arup De, Maya Gokhale, Steven Swanson, and et. al et. "Minerva: Accelerating Data Analysis in Next-Generation SSDs." In: *Proc. FCCM*. 2013.

[4] Jaeyoung Do, J. Patel, D. DeWitt, and et al. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD*. 2013.

[5] Masoud Hemmatpour, Mohammad Sadoghi, and et al. "Kanzi: A Distributed, In-memory Key-Value Store." In: *Proc. Middleware*. 2016.

[6] Zsolt István, David Sidler, and Gustavo Alonso. "Caribou: Intelligent Distributed Storage." In: *PVLDB* 10.11 (2017), pp. 1202–1213. ISSN: 2150-8097.

[7] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, S. Cho, Daniel D. G. Lee, and Jaeheon Jeong. "YourSQL: A High-Performance Database System Leveraging in-Storage Computing." In: *PVLDB* 9.12 (2016), pp. 924–935.

[8] Yangwook Kang, Yang-suk Kee, and et al. "Enabling cost-effective data processing with smart SSD." In: *Proc MSST*. 2013.

[9] Jungwon Kim and et al. "PapyrusKV: A High-performance Parallel Key-value Store for Distributed NVM Architectures." In: *Proc. SC*. 2017.

[10] Sungchan Kim, Sang-Won Lee, Bongki Moon, and et al. "In-storage Processing of Database Scans and Joins." In: *Inf. Sci.* (2016).

[11] *KV-SSD*. https://github.com/OpenMPDK/KVSSD. Samsung.

[12] Sang-woo Jun Ming, Arvind, and et al. "BlueDBM: An Appliance for Big Data Analytics." In: *Proc. ISCA* (2015).

[13] Sudharsan Seshadri, Steven Swanson, and et al. "Willow: A User-Programmable SSD." In: *USENIX, OSDI* (2014).

[14] T. Vincon, S. Hardock, C Riegger, J. Oppermann, A. Koch, and I. Petrov. "NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management." In: *Proc. EDBT*. 2018.

[15] Tobias Vincon, Arthur Bernhardt, Lukas Weber, Andreas Koch, and Ilia Petrov. "On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems." In: *Proc. HardBD at ICDE*. 2020.

[16] Tobias Vincon, Lukas Weber, Arthur Bernhardt, Andreas Koch, and Ilia Petrov. "nKV: Near-Data Processing with KV-Stores on Native Computational Storage." In: *Proc. DaMoN*. 2020.

[17] Louis Woods, J. Teubner, and G. Alonso. "Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances." In: *Proc. SIGMOD*. 2013.

[18] Sam Xi, O. a, M. Athanassoulis, and S. Idreos. "Beyond the Wall: Near-Data Processing for Databases." In: *Proc. DAMON* (2015).

Copyright Notice: Material from: Tobias Vinçon et al. "nKV in Action: Accelerating KV-Stores on Native Computation Storage with Near-Data Processing." In: *Proceedings of the VLDB Endowment, Volume 13*. 2020. DOI: 10.14778/3415478.3415524.

14

A FRAMEWORK FOR THE AUTOMATIC GENERATION OF FPGA-BASED NEAR-DATA PROCESSING ACCELERATORS IN SMART STORAGE SYSTEMS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems* by *Lukas Weber, Lukas Sommer, Leonardo Solis-Vasquez, Tobias Vinçon, Christian Knödler, Arthur Bernhardt, Ilia Petrov, Andreas Koch* in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. The contribution of the author of this thesis is summarized as follows.

> » *As the corresponding and leading author, Lukas Max Weber implemented the framework for automatically generating FPGA-based processing elements that he and Lukas Sommer originally conceived. Integration into the nKV system and software-based orchestration of processing elements were contributed by Tobias Vinçon. The evaluation was conducted by Lukas Max Weber. The manuscript was joint work by Lukas Max Weber, Lukas Sommer, and Tobias Vinçon, with feedback from the other co-authors.* «

ABSTRACT

Near-Data Processing is a promising approach to overcome the limitations of slow I/O interfaces in the quest to analyze the ever-growing amount of data stored in database systems. Next to CPUs, FPGAs will play an important role for the realization of functional units operating close to data stored in non-volatile memories such as Flash.

It is essential that the NDP-device understands formats and layouts of the persistent data, to perform operations in-situ. To this end, carefully optimized format parsers and layout accessors are needed. However, designing such FPGA-based Near-Data Processing accelerators requires significant effort and expertise. To make FPGA-based Near-Data Processing accessible to non-FPGA experts, we will present a framework for the automatic generation of FPGA-based accelerators capable of data filtering and transformation for key-value stores based on simple data-format specifications.

The evaluation shows that our framework is able to generate accelerators that are almost identical in performance compared to the manually optimized designs of prior work, while requiring little to

no FPGA-specific knowledge and additionally providing improved flexibility and more powerful functionality.

14.1 INTRODUCTION

The rate at which new data is produced and stored every day has constantly been increasing in recent years, and with the advent of the internet-of-things (IoT), this trend will continue in the foreseeable future. A substantial amount of the data produced every day is stored in database systems, such as key-value stores (KV-store). Of course, this data is not write-only: To make sense (and gain value) out of the stored data, it needs to be analyzed, ever more so now in the golden age of Big Data and Machine Learning.

Data analytics has been limited by slow I/O interfaces to the attached storage devices such as non-volatile memory (NVM). This severely hampered the processing of stored data. An interesting approach to overcome this limitation is *Near-Data Processing* (NDP): Instead of moving huge amounts of data from storage via the I/O-bottleneck to the CPU for analysis, which will eventually yield a result typically much smaller in size than the input data, Near-Data Processing places the computation much closer to the data. With hardware vendors being able to economically *integrate* processing units with non-volatile memories on a single chip or board, Near-Data Processing can help to overcome the limitations on data analytics imposed by slow I/O interfaces.

One example for a Near-Data Processing system for key-value stores was presented by Vinçon et al. in [18, 19]: By combining what they refer to as *Native Computational Storage*, which removes unnecessary abstraction layers and unifies information about data format and layout in a single layer with NDP capabilities, they were able to demonstrate speedups of up-to factor 2.7x for real-world data analysis. For their approach, they did not only use standard CPUs, but also leveraged the computational power and parallelism of FPGAs. However, the FPGA-based NDP processing elements (PEs) in their work were hand-crafted, requiring significant development effort and expertise.

In addition, not only do data storage formats *evolve* over time, but the specific data representation requirements in the actual NDP-operations also tend to change over time. Hand-crafting highly optimized NDP-accelerators becomes impractical in such scenarios, which may include data analytics on big data sets, or evolving feature vectors in machine learning.

In this work, based on the nKV architecture [18], we present a framework to *automatically* generate FPGA-based NDP accelerators from data format specifications. The generated PEs are able to filter and transform data from key-value stores, based on user-specified filter predicates and transformation rules. The PEs are integrated in a

system-on-chip (SoC) architecture for the Cosmos+ OpenSSD platform [17].

In the evaluation, we compare the performance of the automatically generated accelerators with hand-crafted designs and assess the impact of the data format on the hardware footprint of the generated accelerators.

14.2 MOTIVATION

In general, the development of hardware accelerators for specific applications is a tedious task that requires knowledge of the application domain, as well as expertise in accelerator development and device specifics. Typically, using the database specification, a corresponding hardware accelerator will be implemented using some form of Hardware Description Language (HDL) such as Verilog or VHDL. As soon as the accelerator design is finished, a suitable software interface has to be implemented. Depending on the target platform, this interface may vary. For the OpenSSD Cosmos+, the HW/SW interface has to be developed as device firmware, which is executed on the ARM-cores of the device. Since the architecture and the accelerator design impact how the accelerator is controlled, it is necessary to consider both when developing the software interface. As soon as the software interface is implemented, all of the components can be integrated. In this stage, the firmware is adapted to use the software interface to access the accelerator. Lastly, the hardware design (including the accelerator) has to be synthesized into a bitstream, which is used to program the FPGA-portion of the Zynq-7000 SoC on the Cosmos+. After compilation and synthesis has finished, the accelerated system can be deployed and used.

A major problem of this toolflow is the required cross-domain knowledge. Especially the PE development requires experience with hardware development, as well as a good knowledge of the target platform. Additionally, HW-SW dependencies exist, which makes it impractical to develop the software interface without a finalized accelerator design.

In this work, we aim to implement a framework which allows the *automatic generation* of the accelerator design by composing fixed architecture templates. These templates allow for the concurrent generation of the software interface. The merit of this approach is twofold: First, hardware development expertise is no longer required. The proposed framework is usable without any knowledge about hardware development or HDLs. Instead, the required information is provided to the tools in a simple C-style syntax. Additionally, the dependency between the accelerator design and the interface development is removed, allowing an overall faster development cycle.

14.3 NEAR-DATA PROCESSING BACKGROUND

14.3.1 *Background: Key-Value Stores*

In this work, we focus on Near-Data Processing for wide-spread, high-performance Key-Value (KV) Stores, in particular RocksDB [8]. In order to provide querying capabilities in combination with high sustained insert and update rates, modern KV-Stores often use out-of-place update approaches such as Log-Structured Merge-Trees (LSM-Tree) [14].

An LSM-tree employs multiple components $C_0...C_k$. All insertions and updates are performed on the first component C_0, typically located in memory. After C_0 reaches a defined size threshold, its content, i.e., the insertions and updates, is flushed to persistent memory and merged with component C_1. Over time, the merges will gradually move data from C_0 to C_k to ensure a separation of hot and cold data. During each merge process, outdated key-value pairs are purged and their space is reclaimed.

RocksDB uses LSM trees in a *multi-leveled* variant [13]. The component C_0 comprises multiple MemTables and is located in *volatile* memory, while the remaining components $C_1...C_k$ reside in *persistent* memory (e.g., Flash). Whereas the MemTables in C_0 are typically implemented using a memory-efficient structure such as skip-lists, the data is transformed into the so-called *Sorted String Tables* (SST) format during the flush from C_0 to the persistent component C_1. Each component $C_1...C_k$ in persistent memory comprises multiple SSTs. Each SST in turn is composed by an *index block* and a number of *data blocks*. The key-value pairs are stored in the data-blocks in key-sorted order.

During the merge process, as part of the LSM tree algorithm, the SSTs are *compacted*, i.e., outdated entries are pruned. Nevertheless, as the compaction process only happens as part of the merge process, multiple key-value pairs for the same key can be present on different levels of the LSM tree hierarchy. For example, a more recent key-value pair k, v' in component C_2 supersedes a pair k, v in component C_5. For performance, no compaction takes place during the flush from component C_0 to component C_1.

Access operations to the key-value store, such as GET or SCAN require to traverse multiple index blocks, starting at the MemTables in component C_0. Assuming that the key is not present there, *all* index blocks of every SST from C_1 need to be traversed (remember that no compaction is performed during the flush, thus multiple pairs for a key can be present in C_1), followed by traversing a single index block in the remaining index components $C_2...C_k$. $SCAN$ operations with a value predicate (e.g., $SCAN(0 < Value < 42)$) even require traversal of the *entire* data-set.

The NDP PEs generated by our tool-flow operate on SST files using parallelized NDP operations for faster access.

14.3.2 *nKV: Near-Data Processing Architecture*

The NDP PEs developed in this work is based on the nKV Near-Data Processing architecture developed by Vinçon et al. [18]. Their architecture and custom key-value store exploits two key insights: First, while intermediate layers and abstraction such as block devices and file systems simplify the architecture and implementation of key-value stores such as RocksDB, they also introduce inefficiencies and complicate the implementation of true near-data processing. For NDP to be effective, it needs to operate directly on the physical addresses of the data in the key-value store. Therefore, nKV uses native computational storage, i.e., the intermediate abstraction layers along the critical I/O path have been removed and nKV directly operates on Flash storage, using physical addresses.

Having control over the physical placement of data also allows nKV to optimize the *placement* of data. By distributing data on independent Flash channels and LUNs, nKV facilitates parallel access and processing of data [18]. Moreover, keeping the data of different LSM-tree index components separated on different Flash chips, avoids blocking of the entire bus by compaction jobs taking place as part of the LSM-tree merge.

The second important insight that underlies nKV is the fact that placing the computation closer to the data can significantly reduce the amount of data transferred, and consequently speed-up access. Many KV-store operations, such as the SCAN-operation on value predicates explained in the previous section, are very I/O-intensive, requiring much more data to be moved from storage to the processor than what is required for the final result of the operation. Using Near-Data Processing, i.e., placing the computation much closer to the data, does not only reduce the I/O complexity of the operations, but also allows for higher degrees of parallelism, as the device-internal bandwidth of storage devices (e.g., parallel access to different Flash channels) is typically much higher than the bandwidth of the I/O interface to the processor. In a similar fashion, NDP also achieves much shorter latencies.

In general, KV-Stores employ concrete data formats defined by either the application or by the database object itself (e.g. table), when applied as a DBMS storage engine, the data catalog. The nKV architecture exploits on-device data access and allows for data format interpretation in-situ. While information about the layout and format of data is scattered and encapsulated across multiple abstraction layers in classical KV-stores, nKV removes these layers and introduce on-device infrastructure for data format parsers and accessors in both

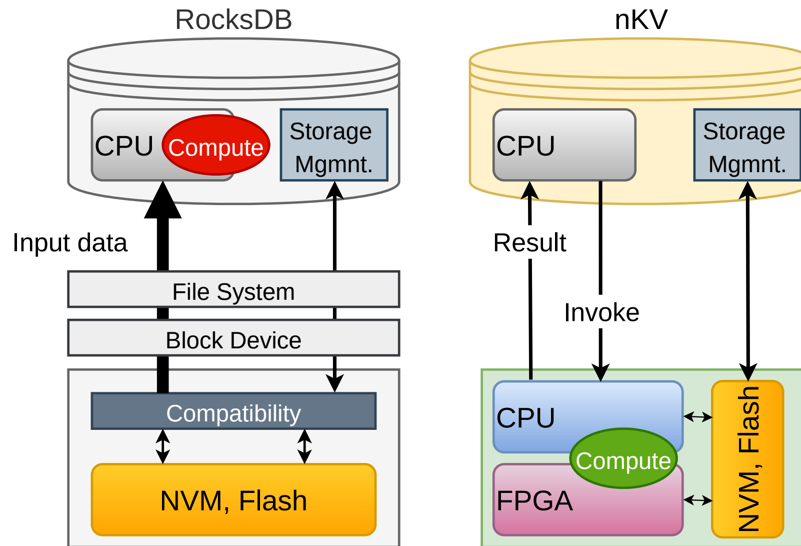


Figure 14.1: Comparison of traditional KV-store and the nKV-architecture with native computational storage and Near-Data Processing.

soft- and hardware. The infrastructure operates on the SST format and allows interpretation of the data format and data access *without host intervention*.

The difference between the nKV architecture, with its native computational storage and use of NDP, and traditional KV-store setups, such as RocksDB, can be seen in Fig. 14.1: While RocksDB has to retrieve large amounts of data from the storage through intermediate layers to perform the requested operation on the host CPU, the nKV architecture can leverage the full device-internal bandwidth of the Flash and perform the requested operation on-device, eventually transferring only the much smaller result set back to the host.

While the prior FPGA-based NDP PEs for nKV were designed manually, this work will target the existing nKV architecture, and provide an *automated* tool-flow for generating FPGA-based hardware accelerators for NDP operations.

14.4 NEAR-DATA PROCESSING ACCELERATOR GENERATION

Our implementation targets the Cosmos+ OpenSSD platform [17], which features a Xilinx Zynq-7000 SoC (XC7Z045). Additionally, the Cosmos+ offers two kinds of memory: Fast but volatile DRAM, and slow but persistent Flash memory.

The Cosmos+ baseline architecture enables it to be used as a "plain" NVMe SSD. To this end, the programmable logic (PL) of the Zynq SoC is used to implement an NVMe interface as well as controllers for the the attached Flash memory. Specifically, the Tiger4 Flash memory controller is used [17]. This baseline architecture is extended with FPGA-based NDP processing elements (PEs) in [18], which supports

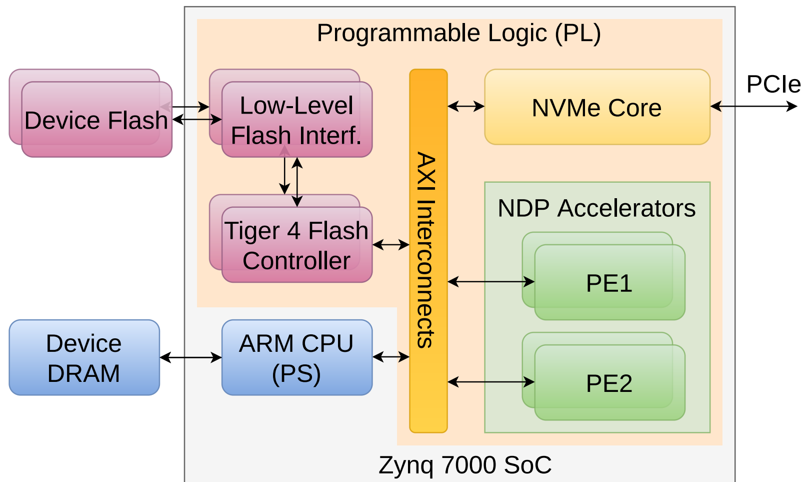


Figure 14.2: Overall system architecture based on the Cosmos+ OpenSSD platform, extend with FPGA-based NDP accelerators.

hardware/software co-execution for NDP in conjunction with the Zynq ARM cores. While [18] uses manually developed PEs, in this work we will introduce a way to automatically generate them from abstract specifications .

When adding FPGA-based PEs, a balance between Flash parallelism and compute parallelism has to be struck, since both the Flash memory controllers and the PEs compete for FPGA resources on the reconfigurable portion of the Zynq-7000. In this work, we use a single Flash DIMM and two separate Flash controllers for the Flash memory. The resulting system architecture is shown in Fig. 14.2.

To reduce the implementation complexity, the PEs are not directly coupled to the Flash memory. Instead, the data is first buffered in DRAM, and the results are also initially collected in DRAM. While this might seem counter-intuitive, this detour does not have significant negative performance impact due to two issues: First, the overall Flash bandwidth achievable using two Tiger4 controllers is only about 200 MB/s. Second, most of the data will be accessed multiple times, and thus profits from being stored in faster DRAM (compared to the relatively slow Flash memory).

14.4.1 *NDP Accelerator Architecture Template*

While the concrete functionality of the accelerators is automatically generated to match the specified filtering and data transformations, all accelerators use the same *architectural template* as a basis. This template, which is also depicted in Fig. 14.3, comprises four main components. The first component, the control component (Fig. 14.3.a) is simply a register file, which is mapped into the memory space of the on-chip ARM core. The registers can then be used for communication between CPU and PE.

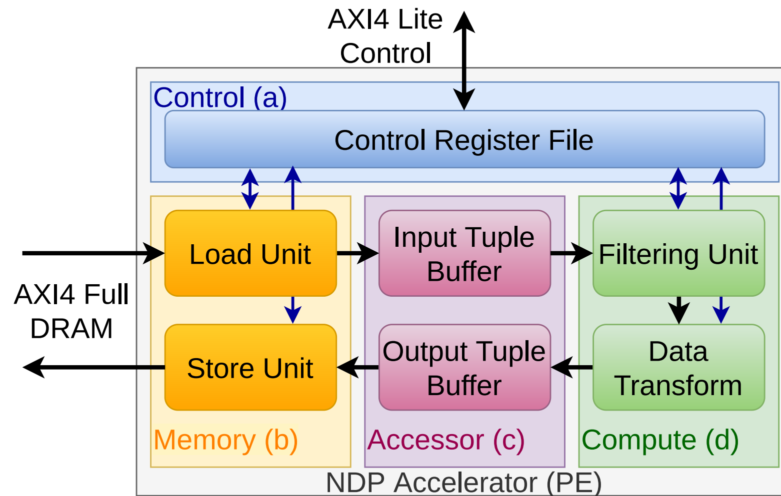


Figure 14.3: Architectural template used by the generated NDP accelerators.

The second component, marked (b), of the template is concerned with loading and storing data from/to memory. As described in the previous section, the PEs do not have direct access to the Flash memory. Instead, the input data is loaded from the DRAM via the corresponding AXI4 interface provided by the Zynq PS. The loading and processing of data takes place at a granularity of 32KB blocks.

The two tuple buffers in the accessor component, marked (c), are responsible for converting between the native bit-size of the memory interface (64 bit on Zynq-7000), and the actual size of a tuple in the KV-store (i.e., a key-value pair).

The computation component, marked (d) in Fig. 14.3, consists of two main functional units: The filtering unit will discard any tuple that does not match a user-specified predicate. Predicates can evaluate elements of the key, as well as the value and, in contrast to prior work [18], can also be defined across *multiple* columns. This is achieved by the option of chaining multiple filtering units, each evaluating a single predicate. The number of filtering stages is configurable, and the framework will automatically generate the required logic.

The second functional unit is the data transform unit, which transforms the tuples that passed the filter, as defined by the user. Example for transforms include discarding RocksDB meta-data, or unnecessary columns. Both units, the filtering unit as well as the data transformation unit, are generated automatically, as described in the next section.

14.4.2 *Automatic Generation of NDP Accelerators*

In general, the underlying abstraction of most contemporary databases is *structured* application data. An example for this structuring are relational databases, that impose a *database scheme* on all of the stored

```
/* @autogen define parser Point3DTo2D with
chunksize = 32, input = Point3D, output = Point2D,
mapping = {output.x = input.y, output.y = input.z }
*/
typedef struct { uint32_t x, y, z; } Point3D;
typedef struct { uint32_t x, y;    } Point2D;
```

Figure 14.4: Example Code showing how a PE is defined for automatic generation. The generated PE will automatically transform data from the `Point3D`-type to `Point2D`-type, discarding the field `x`. Additionally, the `Point3D`-structs can be filtered using predicates on all of the present fields (`x`, `y` and `z`).

data. As an alternative to relational databases, key-value stores employ a less structured way of storing data. While key-value stores typically do not enforce a structure, most applications still use structured data. Thus, the application might use string-based key-value stores to store the binary data, maintaining the application-level structuring of the data outside the KV-store. The application would then use an internal record-based datatype (e.g. `structs`), and transform this data into a corresponding key-value pair. The resulting key-value pair obviously has the same structure as the underlying `struct`.

For our automatic generation, we have to assume that the data is structured, as we would not be able to interpret the value data for filtering or other processing otherwise. Typically, an application will use data-classes or structs to represent this structure. By interpreting these type-definitions, our tools can generate the matching hardware NDP units for the specified data structures. In our framework, we rely on C-inspired type-definitions, as well as annotations for the specification of the PEs. This allows the database engineer to reuse his application code for the generation of PEs. An example for the specification of a PE is given in Fig. 14.4.

From the parsed type-definitions and annotations, an internal representation of these types is built. This internal representation is limited to data-types that are suitable for hardware-processing. Specifically, integers and single/double precision floating point types are supported. In addition to these primitive types, it is also possible to work with (nested) arrays and (nested) structs. For byte-arrays, it is also possible to flag them as string-data using a `prefix` annotation. If the annotation is given, the corresponding byte-array will be split into a `prefix` that is handled as a regular field, while the rest of the byte-array is not used for predicate-evaluation. The reason for this lies in the potential sizes of strings, which makes them very hard to process in hardware.

For example, the output of the Tuple Input Buffer is just a sequence of bits containing the complete data of the corresponding struct. With the information gathered by the contextual analysis, these bits can

be interpreted. For example, consider a `struct Point` which encodes the coordinates x, y and z (all 32 bit integers) of a point in three-dimensional space. The hardware now knows, that the first 32 bits encode x, while the second 32 bits encode y, etc. Using this information, it is now possible to filter points that lie behind a certain threshold (filtering), or project the 3D-data into a two-dimensional space (data transformation).

Contextual Analysis As described previously, the contextual analysis phase of our tools is responsible for computing the data-layouts from the parsed representations of the type-definitions. To simplify this process, the contextual analysis performs multiple transformations on the struct data-type. The input to the contextual analysis are trees representing the struct-types. Each node describes a different part of the overall structs, with leaf nodes representing actual primitive types (e.g., integers), while regular nodes can be nested structs or arrays. In the first step, arrays that are annotated to represent strings are transformed into structs, which contain a `prefix`-field followed by an array, which contains the rest of the string (`postfix`). After strings are resolved in this manner, the next step removes arrays completely from the tree, by flattening them into structs with a corresponding sequence of scalar element fields. In essence, an array `uint_32t [2]` becomes the struct `struct {uint_32t elem_0, elem_1;}`. Since the data layout is identical for both, this scalarization simplifies the following steps. In a final step, the contextual analysis determines the largest *relevant* field. Relevant fields are those that can be used for filtering predicates. In our case, this includes all primitive fields *except* string-postfixes. Using the size of the largest field, the contextual analysis then determines, whether other fields have to be padded. The padding ensures that all relevant fields can be processed in a single comparator unit.

Memory Interface The memory interface contains a Load- and a Store-Unit, both having access to the PS-DRAM via a shared AXI4 Full interface. In contrast to [18], we opted for more flexible units. Vinçon et al. rely on fully static units that always load and store *complete* data blocks (32 KByte). While this keeps the hardware footprint minimal, it is not very efficient with regard to the use of memory bandwidth. Due to the Data Transformation step, which often removes elements such as metadata from the tuples, the output is almost always smaller than 32 KByte. As memory contention is a major bottleneck, reducing the number of memory accesses will improve the performance. In our work, the Load- and Store-Unit can be configured (using the Control Register File) to store variable amounts of data, thereby reducing unnecessary memory accesses and memory contention.

Tuple Buffers The Tuple Buffers transform the unstructured data retrieved from memory into processable structured data, and back again for storage. To do this, a buffer is used to group the incoming stream of 64 bit words, until one or more complete tuples are

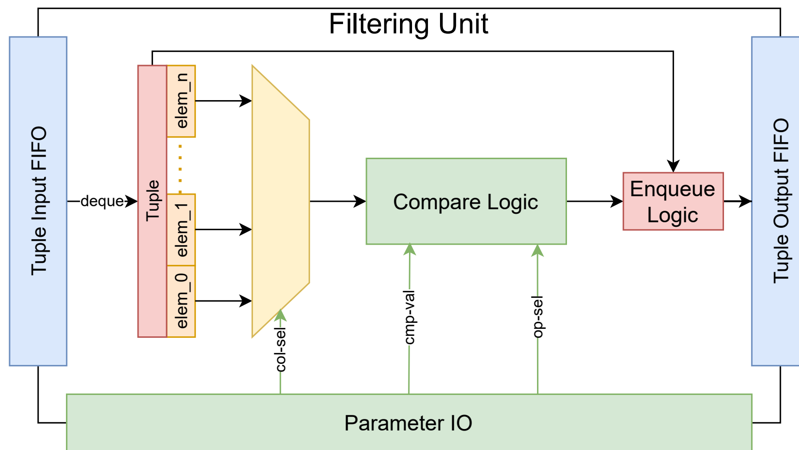


Figure 14.5: Internal structure of the Filtering Unit.

available. According to the padding and type information gathered by the contextual analysis phase, this word is split into a vector of correspondingly padded words. A second vector contains all of the disregarded string-postfixes. The string-postfixes are carried along the computations, but cannot be accessed. The Output Buffer reverses the transformation of the Input Buffer, so that the result can be stored back by the Store Unit.

Filtering Unit This module provides the selection-functionality on the incoming stream of tuples. To do this, hardware is generated that allows the comparison of tuple-members against a given reference-value using a set of compare-operations. An important extension over the work presented in [18] is the fact that the set of operators can be easily extended in our toolflow. Each operation is represented using a function mapping two data-words to a boolean value, which in turn is used to determine, whether a tuple is filtered out. Using a user-defined set of operations or the pre-defined standard set of operations ($\neq$, $==$, $>$, $>=$, $<$, $<=$, nop), the Compare Unit is generated. Since our toolflow relies on the Chisel3-framework [3] for the implementation of the actual hardware, this also enables flexibility. For example, the framework supports interfacing to Verilog and VHDL, which in turn allows addition of custom compare-operations. A schematic view of the filtering unit is shown in Fig. 14.5.

The input and output are FIFOs. In each cycle, a present tuple is dequeued from the input FIFO and one of its elements is selected using a multiplexer. This element is used as input to the Compare Unit which also uses the *compare_value* and *operator_select* to determine the exact operation to perform. The resulting signal is used to determine, whether the current tuple is to be enqueued into the output queue. A very important advantage of this architecture is the chainability. Due to the clear interface, this unit can be chained multiple times to

allow the evaluation of multiple predicates in a pipeline, which was not possible with the architecture in [18].

Data Transformation Unit The Data Transformation Unit is automatically generated from the given struct-types. Both input and output are tuple-FIFOs. During the generation of the Data Transformation Unit, the framework will automatically match each (nested) field of the output-struct to the appropriate (if any) field of the input-struct. Using this mapping of input- to output-fields, hardware will be generated that implements this transformation. In general, there are three cases: 1) When the input and output are of the same struct-type, tuples are simply passed through. 2) If the output-struct contains *only* (nested) fields that are also present in the input-struct, the mapping is automatically derived. 3) If the output struct-type contains (nested) fields that are not present on the input, the user has to specify which (nested) input-field is to be used. While this is very flexible, it also requires user interaction in the form of corresponding annotations. An example for this is shown in Fig. 14.4 with the `mapping`-key. Using this key, it is defined that y and z are used for the projection into 2-d space. Without a mapping, the toolflow would default to the second case and use x and y for the projection.

Composition All of the described modules are then composed into a PE. Due to their latency-insensitive design, the corresponding interfaces can be directly wired-up. Additionally, all modules are automatically connected to their respective control registers. The control register file is automatically configured to provide the required number of registers.

14.4.3 *Automatic Generation of the Software Interface*

In addition to automatically generating the PEs for performing the NDP operations, we also added a tool pass, which automatically generates a *software-interface* for controlling the PEs. The reasoning behind this is to allow a database-engineer to use the PEs without any additional knowledge about how they work and how they are controlled.

Using the information about the Control Register File and the behavior of the PEs, we generate the software-interface bottom-up: First, we generate compiler-macros for encoding the different addresses. From these macros, we built simple software-functions for accessing the different control registers. In a final step, we use these access-functions to built more complex functionality, such as synchronous and asynchronous filtering functions using one or multiple of the filtering stages. For debugging-purposes, functions are generated for printing the state of the PE and for outputting the corresponding data-types. All generated functions are collected in a single header-only library

```
/** Control Register Addresses. */
#define START 0
#define BUSY 4
[...]
#define FILTER_OP_0 60
#define CYCLE_COUNTER 64
/** Generated Functions */
uint32_t filter_sync(...) {...}
uint32_t filter_async(...) {...}
void wait_until_done(...) {...}
```

Figure 14.6: Snippet from the generated software-interface that can be used to interact with the PEs.

file, which can then be added to the project by the database-engineer in order to exploit the PEs.

An example-snippet of the generated header-only library file is given in Fig. 14.6.

14.5 EVALUATION

We will first compare our automatically generated PEs against the hand-crafted units used in [18]. Since [18] has already shown that the NDP approach outperforms the typical non-NDP approach, we will omit this discussion. Then, we will examine the hardware utilizations of the generated PEs and determine their usability on the OpenSSD Cosmos+ SSD platform. All hardware-syntheses are run targeting the Xilinx Zynq-7000 SoC (XC7Z045). In all designs, the Flash controllers and processing elements are clocked at a frequency of 100 MHz, while the NVMe-Core is clocked at 250 MHz, which is in line with the original baseline. While a higher frequency could improve the performance of the PEs, the main bottleneck in this architecture is the available Flash bandwidth.

Performance For the performance evaluation, we use the same benchmarks as in [18]. They work on a sample dataset for a publication reference graph. The nodes of the graph are papers published in journals and conferences. The edges of the graph are references between those papers. Overall, the dataset is comprised of 3,775,161 Paper-Entries and 40,128,663 references between them. For the evaluation, we run GET- and SCAN-operations using the same software-NDP baseline as well as the adapted algorithm, which uses the corresponding PEs. Note that for both operations the execution is implemented in a hybrid way, where the software executes a very general algorithm and exploits the hardware whenever datablocks have to be filtered or transformed.

The resulting NDP-runtimes for GET are shown in Fig. 14.7 (a). Note that both the NDP hardware and software runtimes we report

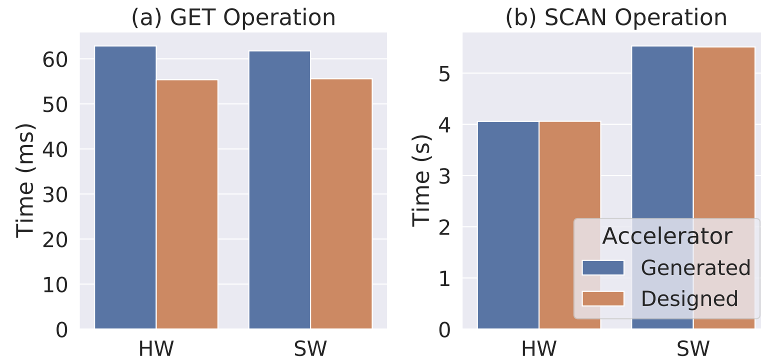


Figure 14.7: Execution times of the GET and SCAN operations, comparing our work to the work provided in [18]. For both Operations execution is executed with HW-acceleration (HW) and without (SW).

for GET are slightly slower (ca. 10%) than those given in [18]. This is due to updated firmware for the COSMOS+ board, which traded some performance for higher reliability. As described in [18], it also makes sense that the GET-operation does not profit greatly from hardware support, since it is sequential and the configuration-overhead (i.e., writing control registers) of accelerators is too high to make an overall difference. Even though, the GET-operation does not improve, the automatically generated PEs are similar in performance in comparison to the ones used by [18].

The SCAN operation has much longer runtimes, making the minor firmware-induced timing variations between [18] and our measurements negligible. As in [18], the hardware-accelerated NDP SCAN is faster than the software version. The performance of our generated accelerator is on par with the manually optimized one as shown in Fig. 14.7 (b). Using the generated PEs slightly increases the runtime by 0.018 seconds from 5.512 seconds to 5.530 seconds.

An additional extension of our work is the possibility to generate PEs featuring multiple filtering stages. Using multiple pipelined filtering stages allows the implementation of more complex NDP-functionality. Moreover, due to the use of elastic pipelines, additional filtering stages will only add very small increases to the overall execution times. Since the filtering stages are able to process a tuple per cycle, the increase in latency of additional filtering stages will be marginal. Especially for compute-bound tasks, this would give the hardware accelerators an edge over the use of the on-device ARM-cores.

Hardware Utilization We generated accelerators that provide the same filtering and transformation functionality as [18] and compare our hardware utilization against theirs. Specifically, we use 1 paper-PE to process the nodes in the graph and 7 ref-PEs to process the edges. Since [18] only reports slices for the PEs, we limit our comparison to

Table 14.1: FPGA Resource Utilization of the PEs used in [18] and our work. The design contains the complete COSMOS+ OpenSSD platform as well as 1 paper-PE and 7 ref-PEs.

	Slice Util. (abs.)		Slice Util.(%)	
	[18]	Our Work	[18]	Our Work
Overall	40821	41934	74.70	76.73
paper-PE	9480	14348	17.35	26.25
ref-PE	1277	1446	1.41	2.65
Available	54650	54650	100.00	100.00

slices as well. Please note that each of our generated accelerators also uses a single BRAM slice, which was not the case for the custom built processing elements of [18].

Table 14.1 shows the corresponding utilization results. It is noteworthy that for both of the PE-types, the resource utilization has grown. Some of this can be attributed to the improved Load- & Store units, which have become more flexible. Specifically, instead of always processing blocks of a certain size, our infrastructure can be configured to load only partial data blocks. Analogously, the Store-Unit can be configured to write back partial blocks. Since the Data Transformation will typically strip data away, this reduces the overall amount of data read and written, which in turn reduces memory contention. Also, note that the overall increase is *less* than expected, considering the size increases of the individual PEs. This is due to a more efficient use of interconnects in our refined architecture template.

We also evaluated the amount of hardware required for multi-staged filtering, as well as for different tuple sizes. For the first part, we take a closer look at the correlation between tuple-sizes and required hardware. For this part of the evaluation, we rely on out-of-context synthesis. In out-of-context syntheses, only a selected part (in our case the PE) is synthesized without the rest of the surrounding architecture. The resulting utilizations represent the amount of logic resources required *without* very dense packing. For the generation of the PEs, we used a number of different input formats that feature tuple sizes ranging from 64 bits up to 1024 bits. For of these sizes, we specified a `struct` with the corresponding number of `uint32_t` and `uint8_t` values. Input and output types are identical and mapped automatically. For each size, we generate a PE that is able to compute on the complete tuple (at the granularity of 32-bit fields) and another PE, where half of the data is discarded using string-prefixes.

The results are shown in Fig. 14.8. An interesting observation is the fact that for smaller PEs, the use of string-prefixing yields a higher slice-requirement. At a first glance, this would make the prefixing

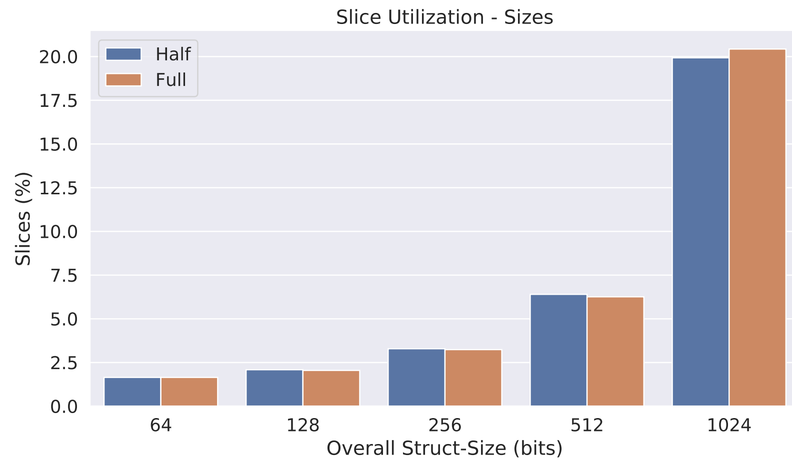


Figure 14.8: Out-of-Context Slice Utilization of generated PEs in correlation to the size of the processed tuples. Half refers to accelerators using the prefixing, whereas Full refers to the ones using all data.

irrelevant. To understand why prefixing is still necessary in some cases, we have to consider that the critical part of our hardware is the Filtering Unit with the compare operations at its core. In Fig. 14.9, all fields have a width of 32 bit, which means that the corresponding compare-operators are also 32 bit operators. For the 1024 bit struct, the corresponding string-data would have an overall size of 512 bits. A full-width compare unit would vastly increase the amount of required hardware. Thus it is still reasonable to use the prefixing.

Lastly, we take a closer look at the multi-stage feature and the resulting hardware-requirements. For this part of the evaluation, we reuse the same data-formats as in the previous step, but focusing on 256 bit structs only. For both (with and without string-prefixes), we built accelerators with up to 5 filtering stages for more complex predicates. Of these, especially the 2-staged ones are interesting, since they could be used to implement RANGE_SCANs. Again, the utilization results were obtained using out-of-context synthesis.

Looking at the results shown in Fig. 14.9, we can see an almost linear correlation between the number of stages and the slice requirement. Additionally, we observe that the increase per additional stage is small compared to the overhead incurred by the fixed part of the template (Load/Store Unit, Tuple Buffers). This implies that multi-stage filtering incurs only minor additional cost, while offering a lot more flexibility.

14.6 RELATED WORK

The first approaches for Near-Data Processing, moving computation closer to the data date back to as early as the 1970s. However, approaches such as database machines [5] or ActiveDisk [1, 12, 15] were

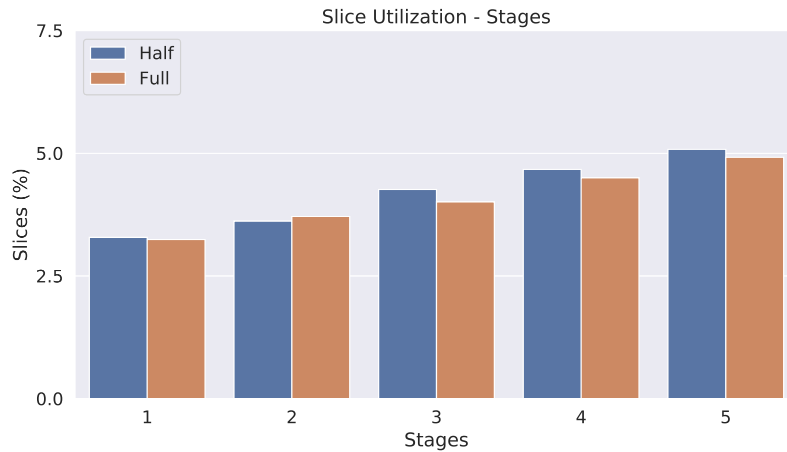


Figure 14.9: Out-of-Context Slice Utilization (in percent) of generated PEs in correlation to the number of filtering stages. Additional stages increase resource requirement in a linear fashion, but provide more flexibility. The use of string-prefixing (Half) has only minor impact.

severely limited by the I/O-limitations and memory bandwidth of mechanical hard-drives.

Only after the wide-spread availability of modern non-volatile storage solutions, e.g., Flash-based SSDs, significant advances in the performance of Near-Data Processing systems became possible. Approaches such as SmartSSD [7, 11, 16] exploit the much higher I/O-bandwidth of modern storage devices as, for example, provided by parallel, independent Flash-channels. JAFAR [2, 20] was one of the first systems focusing on Near-Data Processing for DBMS. Biscuit [10] was another approach targeting NDP for DBMS, namely MySQL. In contrast to our work, they only employed the ARM-based CPUs found in commodity SSD hardware for software-based Near-Data Processing, but also identified the lack of a usable framework for programming NDP PEs as an important issue. Our framework allows to automatically generate FPGA-based Filtering and Data Transformation units from simple user-input. It thus offers a solution to make FPGA-based NDP acceleration accessible to non-FPGA experts.

With their HRL architecture [9], Gao et al. present a new hardware architecture targeting NDP that combines fine-grained reconfigurable regions, as found on FPGAs, with coarse-grained regions as common in Coarse-Grained Reconfigurable Arrays (CGRA). Their overall system architecture combines this accelerator with DRAM in an Hybrid Memory Cube (HMC), but does not include non-volatile memories.

Architectural challenges and other considerations on how to integrate FPGAs into Near-Data Processing architectures were discussed by Dhar et al. [6] and Becher et al. [4]. While Dhar et al. envisioned an architecture featuring Flash storage and a combination of FPGA and High-Bandwidth Memory (HBM), with the FPGA processing

data cached in HBM, the ReProVide architecture proposed by Becher et al. uses a combination of an ARM CPU and an FPGA, similar to our approach. In the multiple dynamically reconfigurable regions of the FPGA, different pre-synthesized NDP PEs can be used. However, these accelerators must be hand-crafted and cannot be generated automatically.

14.7 CONCLUSION & OUTLOOK

In this work we have developed a framework for the automatic generation of FPGA-based accelerators for the use with Near-Data Processing applications. Our evaluation shows that our automatically generated accelerators provide almost identical performance compared to a setup with hand-crafted hardware accelerators. This is worthwhile, since our approach effectively removes the need for custom hardware development and lowers the entry barrier for hardware-accelerated databases. Moreover, our multi-staged filtering approach enables more powerful computations with minimal overhead.

While filtering and transformation of data are wide-spread use-cases that can easily be realized using our framework, more computational and analytical tasks could also be performed using this architecture. In future work, we will investigate, how we can leverage the data-parallelism of the architecture to perform more compute-intensive tasks. Using our architecture, it is possible to access and process all tuple-elements in parallel, which could offer great potential for faster analysis of the processed data.

ACKNOWLEDGEMENTS

This work has been supported by BMBF PANDAS - 01IS18081C/D, DFG Grant neoDBMS - 419942270 and HAW Promotion, MWK, Baden-Württemberg, Germany.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS 1998*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[3] J. Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language." In: *Proc. DAC 2012*. 2012.

[4] Andreas Becher et al. “Integration of FPGAs in Database Management Systems: Challenges and Opportunities.” en. In: *Datenbank Spektrum* 18.3 (Nov. 2018). ISSN: 1610-1995. (Visited on 08/22/2020).

[5] Haran Boral and David J. DeWitt. “Parallel Architectures for Database Systems.” In: 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

[6] Ashutosh Dhar et al. “Near-Memory and In-Storage FPGA Acceleration for Emerging Cognitive Computing Workloads.” In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2019.

[7] Jaeyoung Do, J. Patel, D. DeWitt, and et. al et. “Query Processing on Smart SSDs: Opportunities and Challenges.” In: *Proc. SIGMOD 2013*. 2013.

[8] Facebook. *RocksDB*. https://github.com/facebook/rocksdb. 2020.

[9] Mingyu Gao and Christos Kozyrakis. “HRL: Efficient and flexible reconfigurable logic for near-data processing.” In: *2016 IEEE Intl. Symp. on High Performance Computer Architecture (HPCA)*. 2016.

[10] Boncheol Gu et al. “Biscuit: a framework for near-data processing of big data workloads.” In: *ACM SIGARCH Computer Architecture News* (June 2016). ISSN: 0163-5964. (Visited on 08/21/2020).

[11] Yangwook Kang, Yang-suk Kee, and et al. “Enabling cost-effective data processing with smart SSD.” In: *Proc MSST 2013*. May 2013.

[12] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. “A Case for Intelligent Disks (IDISKs).” In: *SIGMOD Rec.* (1998).

[13] Chen Luo and Michael J. Carey. “LSM-based storage techniques: a survey.” In: *The VLDB Journal* 29.1 (2020), pp. 393–418.

[14] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. “The log-structured merge-tree (LSM-tree).” In: *Acta Inform.* (1996).

[15] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. “Active Storage for Large-Scale Data Mining and Multimedia.” In: *Proc. VLDB 1998*. 1998.

[16] Sudharsan Seshadri, Steven Swanson, and et al. “Willow: A User-Programmable SSD.” In: *USENIX, OSDI* (2014).

[17] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. “Cosmos+ OpenSSD: A NVMe-based Open Source SSD Platform.” In: *Flash Memory Summit* (2016).

[18] Tobias Vinçon et al. "NKV: Near-Data Processing with KV-Stores on Native Computational Storage." In: *Proc. 16th International Workshop on Data Management on New Hardware*. Portland, Oregon: ACM, 2020. ISBN: 9781450380249.

[19] Lukas Weber et al. "On the necessity of explicit cross-layer data formats in near-data processing systems." In: *Distributed and Parallel Databases* (2021). DOI: https://doi.org/10.1007/s10619-021-07328-z.

[20] Sam Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. "Beyond the Wall: Near-Data Processing for Databases." In: *Proc. DAMON* (2015).

Copyright Notice: © 2021 IEEE. Reprinted, with permission, from Lukas Weber et al. "A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems." In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 136–143. DOI: 10.1109/IPDPSW52791.2021.00028.

Part IV

SUM-PRODUCT NETWORKS IN NEAR-DATA PROCESSING

15

EXPLOITING SUM-PRODUCT NETWORK INFERENCE TO OFFLOAD DATABASE QUERY CARDINALITY ESTIMATION TO FPGA-BASED COMPUTATIONAL STORAGE DEVICES

BIBLIOGRAPHIC INFORMATION

The content of this chapter has been submitted for publication as the work *Exploiting Sum-Product Network Inference to Offload Database Query Cardinality Estimation to FPGA-based Computational Storage Devices* by *Lukas Weber, Yannick Lavan, Torben Kalkhof, Carsten Heinz, Andreas Koch* in *ASPLOS 2024: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding and leading author, Lukas Max Weber investigated the feasibility of regular Sum-Product Networks for the Cardinality Estimation use case and performed the corresponding empirical evaluation. Yannick Lavan provided an in-depth mathematical perspective on the approach's limitations in more complex queries. Based on the findings, Lukas Max Weber designed and implemented a framework to generate hardware accelerators capable of performing Cardinality Estimation. He also conducted performance and resource utilization experiments. The resulting manuscript was written by Lukas Max Weber, with a corresponding section by Yannick Lavan. The manuscript was refined with feedback from the other co-authors. «*

ABSTRACT

Cardinality Estimation is a key operation in database systems. Based on prior work, we evaluate the use of Sum-Product Networks (SPNs) as a means to estimate cardinalities of database queries. We show their applicability to different classes of queries analytically and empirically, discussing advantages and disadvantages.

Based on prior work and our additional analysis, we then build a framework to generate hardware accelerators for cardinality estimation. Our framework is able to generate latency- and throughput-optimized accelerator variants using a flexible fixed point number format for the actual inference. We extend the functionality of prior work to enable the marginal and range-based queries that are relevant for cardinality estimation.

With our framework, we implement different architectures for different use-cases, including Near-Data Processing on smart storage

devices, as well as general purpose server-side cardinality estimation. Using a PCIe-based accelerator card, we achieve an estimation latency of 6.62 µs. This corresponds to a speed-up of almost 40x over CPU-based prior work. Additionally, we show an estimation latency of less than 2 µs in a Near-Data Processing setup on smart storage. Furthermore, we investigate the error bounds of the flexible fixed point number format used and the corresponding resource utilization.

15.1 INTRODUCTION

In recent years, machine learning and big data have become active fields of research within computer science. With the development of models such as ChatGPT by OpenAI, interest in AI has increased further. These models deal with an ever-increasing amount of data that is produced and stored every day, making data storage and management an equally important field. However, while there have been efforts to improve storage systems, there is still much more potential for optimization. New GPUs and CPUs often come with specific optimizations for machine learning. For example, AMD's recently announced Ryzen 7000 chips will feature XDNA AI Engines specifically targeted at neural networks. Progress in the storage domain has been far slower, though.

While consumer storage devices have advanced in capacity, latency, and throughput, most of these advances are not specific to machine learning. One potential improvement to storage devices for machine learning applications is the use of Near-Data Processing (NDP), which offloads computational load from the CPU to the storage device. NDP is especially interesting in applications where the stored data can be pre-processed *on* the storage device itself using data-reductive operations such as selections. By performing these operations on the storage device, bandwidth on the PCIe-bus can be freed up, and the transported data is of greater relevance to the application, as it contains less data that will be discarded after the transport. For example, consider training a model on a huge dataset containing functionally dependant data (e.g. age and birth date). Removing such redundancies using NDP projection avoids inefficient data transfers and will also increase performance, if data movement is a bottleneck.

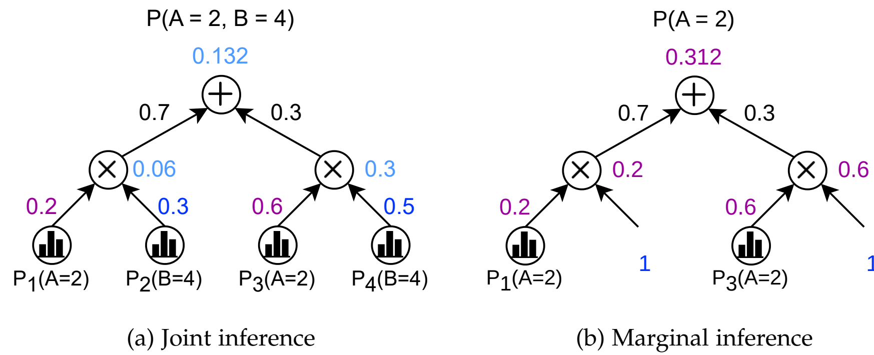
To implement NDP, smart (also called *computational*) storage devices such as the Samsung SmartSSD [3] or the Zynq-7000-based COSMOS+ OpenSSD [15] can be employed. The COSMOS+ OpenSSD is a regular NVMe-based SSD, but its flash controllers are implemented in the programmable logic (PL) of the Zynq-7000 SoC, and the Cortex A9 cores are used to run firmware. To enable NDP, the hardware on the PL and firmware of the COSMOS+ can be extended with user-defined functionality. Thanks to the FPGA-based SoC, simple NDP operations can be realized in hardware or software, as shown in

[19]. For more complex NDP operations, result handling becomes an important problem [16], as intermediary results must be materialized. Efficient materialization is only possible for results that fit into on-chip block RAM (BRAM) or on-board dynamic RAM (DRAM), making the prediction of result sizes of a given query highly relevant to determine the best result-handling strategy. This process is called *Cardinality Estimation* (CE).

While it is often beneficial to perform CE in an NDP manner, CE is also a key operation in non-NDP-database systems. Specifically, CE is generally used in query optimization, which translates complex database queries into a sequence of subqueries, called an execution plan. Optimally, data-reductive selections and projections are performed *early* in an execution plan, since this will reduce the runtime of later, more complex operations like joins.

A relatively novel approach to CE is the use of Sum-Product Networks (SPNs), as demonstrated in DeepDB [7]. While that work provides an interesting proof of concept for using SPNs in CE, it lacks detail on how queries are actually estimated using SPNs. Instead, the paper focuses on additional applications of SPNs in database and storage systems. SPNs are a type of probabilistic graphical model that can be used for a wide range of tasks, including classification, regression, and density estimation. In the context of CE, SPNs can be trained to estimate the cardinality of data sets, which has the potential to improve performance and accuracy over traditional methods. However, further research is needed beyond DeepDB to explore the feasibility and effectiveness of this approach in practical applications.

This paper has three main contributions. First, we evaluate the advantages and drawbacks of using SPNs for CE. Second, we evaluate the accuracy of SPNs for estimating cardinality using general-purpose datasets, and identify cases where the usage of SPNs requires special care. We also evaluate the impact of the corresponding cases empirically. Third, based on prior work, we build a framework for the automatic generation of FPGA-based accelerators for SPNs that can be used to perform cardinality estimation in both NDP- and non-NDP settings, and which can generate latency- and throughput-optimized variants for different use-cases. Finally, we discuss different applications for the generated accelerators and evaluate corresponding system designs that target different application scenarios. Overall, our work provides new insights into the feasibility and effectiveness of using SPNs for CE, and presents a framework for generating FPGA-based accelerators that can also be used in practical NDP applications.



(a) Joint inference

(b) Marginal inference

Figure 15.1: Inference example in an SPN, representing the joint probability distribution $P(A, B)$. In *joint* inference, all histograms output a corresponding value (a), while in *marginal* inference some histograms are marginalized and always output the value 1.0 (b).

15.2 BACKGROUND

15.2.1 *Sum-Product Networks*

Sum-Product Networks (SPNs) [11] are probabilistic circuits represented as directed acyclic Graphs (DAGs) that encode joint distributions over random variables. SPNs consist of three types of nodes: weighted sums, products, and leaves, which encode univariate or multivariate distributions over random variables. By performing a bottom-up pass through the DAG, joint and marginal inference can be efficiently performed on complete or partial evidence. Figure 15.1 provides an example of joint and marginal inference in an SPN.

Sum-Product Networks (SPNs) can be constructed by hand for a specific purpose, or automatically trained on a given dataset. Different training approaches can be classified as structure learning, weight learning, or a combination of both. Random generation of SPNs with subsequent weight learning has also been shown to be a practical approach [10]. For example, an SPN can be learned from scratch by performing structure learning to identify the relevant independent variables and their relationships, or refined by adjusting the weights of an existing SPN to improve accuracy. Relevant for this work is research on learning SPNs from relational databases [7], with the goals of performing CE and Approximate Query Processing (AQP) in database and storage systems. This involves learning the underlying structure of the database schema and using that information to construct an SPN that accurately estimates the number of distinct values in a given query. The ability to automatically learn SPNs from data makes them a powerful tool for a wide range of applications in machine learning and artificial intelligence.

15.2.2 *Cardinality Estimation*

CE is a key operation in database and storage systems and used to predict the size (cardinality) of query results. In relational databases, CE specifically estimates the number of rows in a table or in an intermediary result. CE is often used in query optimization to rearrange subqueries for improved performance. Most databases use simple approaches for CE based on approximating data distributions using histograms and/or cost models. However, recent work suggests that machine learning and learned models are also a valid approach for CE [7]. SPNs are a promising approach for CE because they can perform very precise estimations on actual *probabilities*, no *softmax* operator etc. is used. When executing on FPGAs, custom numerical types can be exploited to achieve more efficient implementations.

15.3 RELATED WORK

Cardinality Estimation The core idea of this work is derived from [7], which uses SPNs to perform CE, as well as AQP. In their work, Hilprecht et al. define Relational Sum-Product Networks (RSPNs), which are an extension of regular SPNs aimed towards relational databases. Specifically, RSPNs typically come in sets or ensembles which represent multiple datasets or database tables. Additionally, RSPNs support typical database-specifics like NULL values, handling of functional dependencies, and incremental training to keep the RSPN in sync with updates of the dataset. The authors show that CE and AQP using RSPNs is feasible. While there is prior work on CE for relational databases, most of those works do not use SPNs as model. The relatively extensive survey by Harmouch et al. discusses typical approaches for CE [5].

Near-Data Processing While first experiments towards NDP took place as early as the 1970s, most of these early approaches (like database machines [2] and ActiveDisk [1]) were rather unsuccessful, due to I/O and general bandwidth limitations. More recently, there has been work with Smart SSDs that exploit the higher bandwidth of typical flash memories [3].

Also, *bump-in-the-wire* processing has become more relevant, with a number of publications by Vincon et al. [19], which introduced the concept of cross-layer data formats in NDP-based key-value stores. By giving the storage device knowledge about the structure of the stored data, typical key-value store operations like GET and SCAN could be performed on a COSMOS+ OpenSSD. They describe corresponding implementations in [19], proving that more complex operations like SCAN can profit from software- and FPGA-based NDP. Furthermore, the work was later extended by an approach to generate the FPGA-based NDP operators *automatically* from annotated code [18]. Lastly,

the authors have shown the importance of result-set handling and suggest possible solutions in [16].

Sum-Product Networks The first relevant paper considering the generation of hardware accelerators for SPN inference was published in 2018 by Sommer et al. [13]. It originally employed a compiler-like approach to read SPNs from a textual representation and generate corresponding accelerators using FloPoCo 64-bit floating point operators. In later works, this approach was extended to different custom data types like a logarithmic number system [17], as well as posit numbers and a custom floating point number system [14]. In more recent works, the SPN accelerators were integrated in different architectures enabling inference in 100G networks [6] and using fast on-chip HBM [20].

In parallel, Shah et al. developed a custom architecture for executing inference on probabilistic circuits [4, 12]. Their approach also features improvements to the learning process that ensure that intermediary results are as precise as possible given different configuration parameters of the overall architecture.

15.4 SPNS AND CARDINALITY ESTIMATION

Before discussing the hardware implementation of probabilistic cardinality estimation as FPGA accelerators, we present a brief overview of key concepts driving the implementation approach. For a comprehensive discussion on incorporating SPNs into database systems, we refer the reader to Hilprecht et al. [7]. In this study, we focus on applying FPGA-based cardinality estimation for data distributions represented as SPNs over histogram leaf nodes. We assume *normalized* histograms, where each bucket signifies a *probability* rather than a density.

15.4.1 *Estimating Query Cardinalities*

In this section, we describe the probabilistic operations executed for corresponding database queries. Throughout this section, we represent the learned probability distribution for a database table T by $P(\mathbf{X})$, where $\mathbf{X}$ is the vector of random variables corresponding to each column in T. Generally, the estimated cardinality c of a query result is calculated as an expectation over the learned data distribution by determining the probability of the provided evidence vector $\mathbf{E}$ and multiplying it by the number of rows n in the table, as follows:

$$c = \mathbb{E}_{x \sim P(\mathbf{X}|\mathbf{E})}(\mathbf{X}) \approx n \cdot P(\mathbf{E}). \tag{15.1}$$

Next, we explain how to obtain $P(\mathbf{E})$ for various types of database queries.

Single-Column Equality Queries In the most trivial filtering case, we aim to estimate the outcome of filtering a table for a provided value in a specific column. The evidence provided to the SPN consists of a single random variable X_i corresponding to the column i of interest, resulting in the computation

$$c \approx P(X_i = x) \cdot n, \tag{15.2}$$

effectively marginalizing every other column. In SPNs, the marginalization is achieved by outputting the value 1.0 for leaf nodes of marginalized variables.

Range Queries In many applications, we are interested not only in single values for columns but also in ranges of data, such as "determine the number of scientists who have published *fewer* than five papers". Formally, we aim to evaluate the probability that the value of a random variable X_i falls within the range $[x_l, x_m]$ with $l < m$. Calculating the probability involves determining the probability of the or-event of several mutually exclusive events:

$$P(x_l \leq X_i \leq x_m) = \sum_{j=l}^{m} P(X_i = x_j). \tag{15.3}$$

Since non-leaf operations in SPNs remain constant w.r.t. j in Eq. 15.3, the sum propagates into the histogram leaves corresponding to X_i. This property enables us to compute the probability of single-column range queries without evaluating the entire SPN multiple times. Modifying the comparison operators within the query only changes the start and end indices for the sum operation.

AND Queries While the previous paragraphs focused on single-column queries, we can also combine queries. In this work, we restrict our scope to queries of the form `A <= 42 AND B = 123`, excluding queries like `A <= 42 OR B >= 10`, as the latter would require multiple SPN passes. Computing such a query is possible, but would require at least *three* passes over the SPN: One to determine `A <= 42`, and a second one for `B >= 10`. Since both subqueries might overlap on the underlying dataset, the overlap has to be determined as well (`A <= 42 AND B >= 10`), to ensure that the overlap is not included twice. Furthermore, we limit the range queries to one predicate over a *range* and any number of *equality* predicates, as we cannot propagate the sum over all outcomes as before. Formally, we assess the probability $P(x_l \leq X_i \leq x_m, \mathbf{E} = \mathbf{e})$, with X_i corresponding to the column for the range computation and $\mathbf{e}$ representing evidence for other columns of interest $\mathbf{E}$. This probability is obtained by computing the sum inside X_i's histogram nodes, setting all other histograms for relevant columns to the corresponding evidence, and outputting *1.0* for all histogram nodes of marginalized columns. For practical purposes, we later empirically evaluate the effect of approximating cardinalities by applying the histogram summing of single range queries to multiple columns.

15.4.2 *Hardware-Specific Model Optimization*

While histograms with only a few buckets can be realized easily on FPGA using on-chip BRAM, histograms with a larger number of buckets, e.g. for 32-bit integers would either require LUTs with an infeasible number of entries, or a merging of histogram buckets, which may result in loss of representation accuracy. While this can be a valid approach if many adjacent buckets are similar, it is not generally applicable. Instead, we propose a different solution: Due to the nature of SPNs to learn distributions over arbitrary data, we can make use of the underlying probabilistic semantics and simplify the resulting hardware.

Let X^n denote some n bit wide random variable. Then $P_L(X^n = x)$ denotes the probability output by a histogram leaf L for X^n taking the value x. Let us now assume w.l.o.g. that we slice X^n evenly into four bit chunks. Then we can view $P_L(X^n = x)$ as the joint probability $\hat{P}_L(X^4_{\lfloor n/4 \rfloor -1} = x_{n-1:4\cdot(\lfloor n/4 \rfloor -1)}, ..., X^4_1 = x_{7:4}, X^4_0 = x_{3:0})$, with X^4_i denoting the random variable corresponding to the i-th nibble of X^n and $x_{m:n}$ corresponding to the concrete assignments of bits m through n of x.

Leveraging this insight, we can either retrain the entire SPN by pre-processing the training data and bit-slicing each data point to the desired BRAM size or by learning compact SPNs representing each histogram node and replacing the histogram nodes with the corresponding SPN.

15.4.3 *Empirical Analysis*

To determine the feasibility of SPNs for CE, we initially built a software simulation, capable of executing queries. It supports actual execution by applying filters to the dataset, as well as performing the inference of a corresponding trained SPN. For this preliminary evaluation, we rely on the NIPS dataset, which encodes the frequencies of specific words in a corpus of ML publications as a big table. The advantage of the NIPS dataset is the use of many columns with small value ranges, which enables the enumeration of queries. While other datasets could also be targeted using the approach described in Section 15.4.2, the NIPS dataset with its increasing number of columns is well suited for scaling experiments. This is especially interesting later on, when evaluating performance and hardware utilization in Section 15.6.

Since we want to focus on the feasibility of employing SPNs for CE, we use the selectivity of queries for the analysis, as this is the actual output of the SPN inference. The cardinality is then obtained by multiplying with the number of entries in the dataset. We obtain the queries by performing a range analysis on each column. Using these ranges, we then enumerate all single-column (SC) queries and later

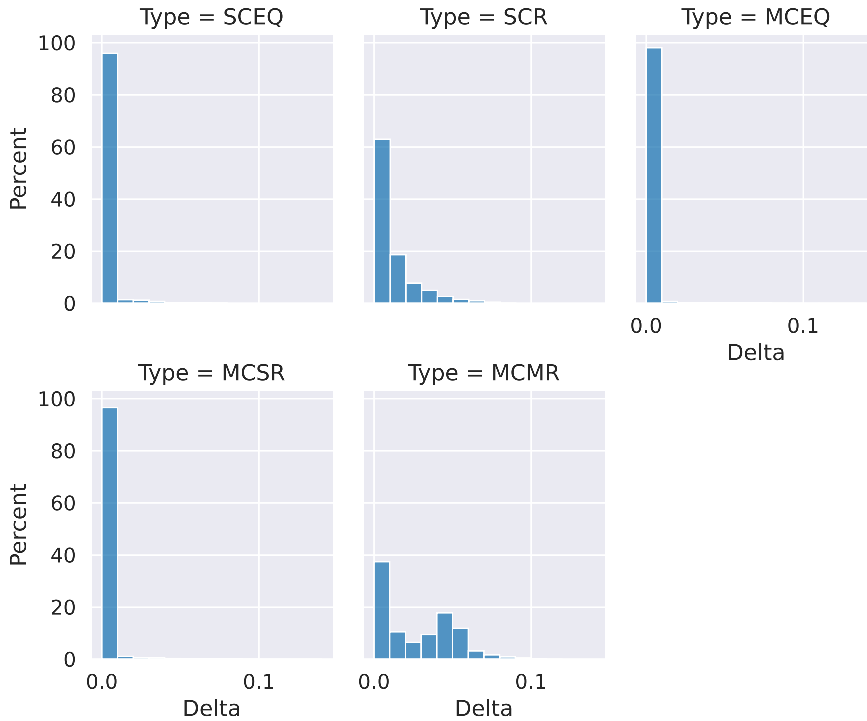


Figure 15.2: Observed estimation error for range queries on all NIPS datasets.

on combine multiple single-column queries to build multi-column (MC) queries. For SC queries, we further differentiate between equality (SCEQ) and range-based (SCR) queries. For MC queries, we differentiate between queries using only the equality operation (MCEQ), queries using a single range-based operation (MCSR), and queries using multiple range-based operations (MCMR).

We limit the analysis to MC queries targeting at most three columns, as bigger queries would most likely be subjected to query optimization/planning, which would divide the query into smaller or even atomic queries. Lastly, we also limit the analysis to queries yielding at least one result. The reason behind this constraint is simple: SPN inference returns very small values in those cases where the true selectivity is zero, yielding very small absolute errors. We exclude those cases from the examined queries, as they do not matter in typical use cases of CE, such as buffer allocation in heterogeneous memory hierarchies. Table 15.1 lists the datasets and the corresponding number of queries used in the analysis.

Single-Column Equality (SCEQ) First, we enumerate all SCEQ queries yielding at least one result. For each query, we then execute the query and the corresponding SPN inference to compute the actual and estimated selectivity. The leftmost subplot of Fig. 15.2 shows the distribution of the estimation error. It shows that the estimation error is below 0.05 for almost all queries over all datasets. The maximum estimation error is 0.075.

Table 15.1: Datasets and the corresponding number of examined queries

	SC		MC			
Dataset	EQ	R	EQ	SR	MR	Overall
NIPS5	111	710	9,782	134,682	200,000	345,285
NIPS10	227	1,426	92,548	200,000	200,000	494,201
NIPS20	480	2,882	129,092	200,000	200,000	532,454
NIPS30	740	4,400	165,960	200,000	200,000	571,100
NIPS40	1,027	5,980	200,000	200,000	200,000	607,007
NIPS50	1,255	7,626	200,000	200,000	200,000	608,881
NIPS60	1,573	9,308	200,000	200,000	200,000	610,880

Single-Column Range (SCR) For single-column range queries, we take a similar approach. As we can see in Fig. 15.2, the distribution of estimation errors changes. While most estimation errors are still relatively small, there are fewer cases with an estimation error of less than 0.01. While the histograms are checked for correct normalization, they typically cannot add-up to exactly 1.0 due to the use of double-precision floating point numbers. With the conversion to a fixed-point number format, additional conversion errors are introduced, which can add-up when querying a bigger number of buckets.

Multi-Column Equality (MCEQ) As discussed in Section 15.4, the error of this class of queries should not significantly increase over SCEQ, as the inference does not introduce any mathematical issues apart from potential precision errors due to the digital arithmetic. Accordingly, the estimation error behaves similarly to SCEQ. Additionally, the errors are also typically less than for the SCR queries, since they are computed over more histogram buckets.

Multi-Column Single Range (MCSR) For these queries, the estimation error also behaves as expected. Interestingly, the impact of the range-based subquery is not as prevalent as in SCR. While the range-based subquery will introduce an estimation error, its impact is not as high, as the rest of the computation is still relatively precise.

Multi-Column Multiple Ranges (MCMR) This class is certainly the most interesting. As discussed in Section 15.4, the estimation error in this class should be relatively high, since summing up multiple histograms actually violates the arithmetic precedence. To achieve an arithmetically correct result, we would have to transform all range-based queries into the corresponding set of equality-based queries, execute each combination and sum up the results. By summing up the buckets directly, we can perform the estimation in a single pass. As expected, the violation of precedence leads to more queries yielding higher estimation errors. But while more queries have higher estimation errors, the worst-case error still remains below 0.1, and is thus still in the same magnitude as for the other classes. So empirically, it

is practical to perform cardinality estimation for the given queries on the given dataset.

15.5 SPN ACCELERATORS

While SPNs are still a relatively novel ML model, there is already an extensive body of prior art on using FPGAs to accelerate SPN inference. In this work, we implement a separate framework that uses a similar approach to [13]. Instead of using the more complex encodings used in that prior work, we instead opt for a much less complex fixed-point based approach. Since SPNs rely on probabilities internally, all encoded numbers are limited to the range of $[0, 1]$. Thus, we can use a single bit to encode the integer portion of the corresponding numbers. Using the capabilities of Chisel3 for hardware construction, we made the number of bits for the fractional portion parametrized, which can then be used to trade-off hardware resources with precision. The key justification for the simpler fixed-point encoding compared to that of [13] lies in the different application-specific error requirements. In the more general prior work [14] the goal was a maximum relative error of 10^{-6} for all results to ensure that all results could be compared to other results. While there are certainly applications that require such tight error margins, CE does *not*, as we typically only want to distinguish cases with small or large result sets. The comparison of those cases is not as relevant, since multiple queries are chained and identifying a *single* data-reductive partial query suffices to reduce the result size of *all* subsequent partial queries. [7] shows that the estimation errors using SPNs are *much* smaller in almost all cases. Thus, using more complex number encodings for higher accuracy is not applicable for CE.

For this work, we implement three configurable fixed-point operators, starting with `FxAdder`, which adds two fixed-point numbers. If numerical issues lead to results > 1, the result is saturated to 1. Second, we implement `FxMultiplier` that multiplies two fixed-point numbers using tiled integer multiplication. The tiled multiplication is built from 24×17 bit multiplications to ensure that FPGA DSP slices are used for fast and area-efficient integer multiplication. The tile-size is configurable to enable easy porting to other architectures. Lastly, we implement `FxHistogram`, which is further described in the following paragraph. Using these operators, we can easily generate a fully spatial, fully pipelined datapath by traversing the SPN, generating corresponding operators and interconnecting them. To make the accelerators as reusable as possible, the fully pipelined datapath is bundled with some control logic and a buffer FIFO. The resulting core acts as a free-running kernel: Data is delivered and retrieved via corresponding AXI-Stream interfaces. Depending on the use-case, different wrappers can also be generated around the free-running kernel. We provide

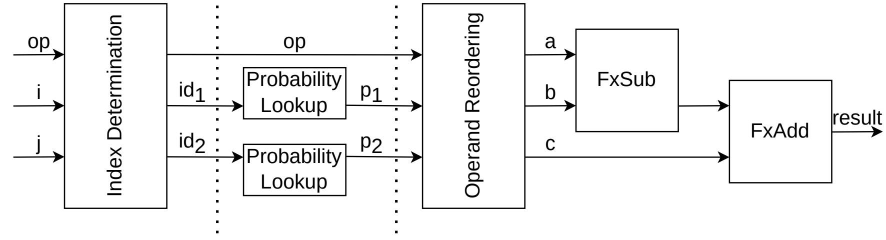


Figure 15.3: `FxHistogram` Module. Dotted Lines indicate pipeline stages. The number of pipeline stages for `FxSub` and `FxAdder` depends on the used encoding.

Table 15.2: Functionality of `FxHistogram` and its submodules. b denotes the memory storing the accumulated histogram buckets.

		Probabilities		Operands		
Op	Calculation	p_1	p_2	a	$-b$	$+c$
$=$	$(b[i] - b[i-1])$	$b[id_1]$	$b[id_2]$	p_1	p_2	0.0
$\neq$	$1 - (b[i] - b[i-1])$	$b[id_1]$	$b[id_2]$	1.0	p_1	p_2
$<$	$(b[i-1] - 0)$	$b[id_1]$		p_1	0.0	0.0
$\leq$	$(b[i] - 0)$	$b[id_1]$		p_1	0.0	0.0
$>$	$1 - (b[i] - 0)$	$b[id_1]$		1.0	p_1	0.0
$\geq$	$1 - (b[i-1] - 0)$	$b[id_1]$		1.0	p_1	0.0
R	$(b[j] - b[i])$	$b[id_1]$	$b[id_2]$	p_2	p_1	0.0
M				1.0	0.0	0.0

two wrappers: One for latency- and one for throughput-optimized inference (cf. Section 15.5.1).

FxHistogram Module The new `FxHistogram` module is an improvement over prior work, which exclusively focused on *equality*-based bottom-up inference. It also provides other compare operations ($\neq, <, \leq, >, \geq$), as well as range and a marginalize operations.

The new module (shown in Fig. 15.3) enables the use of new operators, while the implementations from prior work could only compute equality-based inference. The first step in enabling the additional operations was the introduction of accumulated probabilities. In prior work, the probability lookup was limited to equality, so the value of each bucket could simply be stored in read-only memory using the *index* of the bucket as address. To enable the computation of *ranges* of buckets, we instead compute the accumulated probability. For each bucket, all probabilities up to and including the current one are summed and stored. By using this approach, the calculation for each of the corresponding operations can be achieved using two probability lookups. The corresponding addresses (idx_1 and idx_2) are determined in the Index Determination stage. Depending on the operation, up to two lookups happen concurrently, yielding the probabilities p_1 and p_2. The required calculations always employ three operands (a, b and c), of which up to two are fixed to 0.0 or 1.0, depending on the op-

eration performed. The Operand Reordering stage will reorder the incoming probabilities and add zeroes or ones accordingly, so that the correct final result can be computed using the normalized computation $a - b + c$. The exact mathematical calculations, the required lookups, and the final operands are shown in Table 15.2.

15.5.1 *System Integration*

Different approaches can be used to integrate the free-running kernel into a system. For streaming-based accelerators, integration is straightforward, but for memory-mapped approaches, two wrapper modules were implemented: a latency-optimized and a throughput-optimized variant. Both use an AXI4Lite interface to expose control registers, as well as an interrupt signal to enable asynchronous execution. The throughput-optimized variant also provides an AXI4 interface for batch processing.

Using the latency-optimized variant, we tested the accelerator on five setups for the NIPS40 dataset, achieving end-to-end execution times shown in Table 15.3. The software interface runs on the ARM cores in SoC-based platforms and on the x86 host CPU in PCIe-based platforms.

Setup (a) integrates the accelerator for a Near-Data Processing scenario on the COSMOS+ smart SSD. Execution of SPN inference takes 6.85 µs, when controlling the accelerator from a baremetal firmware running on the Cortex A9 of the Zynq 7000 SoC. Polling is used to determine whether the hardware execution has finished. While (a) employs actual smart SSD hardware, that COSMOS+ platform with its Zynq 7000 series device is showing its age. Thus, setup (b) tests the same accelerator architecture, but on a more recent SoC hardware (Ultra96, using an Zynq UltraScale+ MPSoC device), where it achieves a latency of 1.6 µs.

In contrast to the baremetal operation in (a) and (b), the setups (c), (d), and (e) use the *TaPaSCo* FPGA framework [8] for hardware system integration and as a runtime API. Setup (c) uses TaPaSCo on an Ultra96 running an embedded Linux. Setups (d) and (e) target a non-NDP CE scenario and use PCIe-based accelerator cards (VC709 and Alveo U280). Both are connected to their respective host via PCIe Gen3 x8 and x16 respectively. While inference is still relatively fast, the added overheads of the Operating System and interrupts, instead of the polling used for the smart SSD scenario, increase latency significantly. This is especially visible for the Ultra96, which now has a slower of latency of 26.7 µs. The reason lies in the now full-scale OS running on a small ARM Cortex A53 core, compared to the lightweight firmware employed when the same platform is used as a more modern smart SSD stand-in for (b).

Table 15.3: Hardware Platforms used in Evaluation. Latency measured for NIPS40.

Test Setup	**(a)**	**(b)**	**(c)**	**(d)**	**(e)**
Platform	COSMOS+	Ultra96	Ultra96	VC709	U280
Arch	Zynq 7000	Zynq US+	Zynq US+	PCIe	PCIe
Fabric	Kintex 7	UltraScale+	UltraScale+	Virtex 7	UltraScale+
Driver	none	none	TaPaSCo	TaPaSCo	TaPaSCo
Control	Polling	Polling	Interrupt	Interrupt	Interrupt
Freq.	200 MHz	440 MHz	440 MHz	200 MHz	420 MHz
Latency	6.85 µs	1.6 µs	26.7 µs	16.1 µs	21.3 µs
CPU	ARM	ARM	ARM	Ryzen	Epyc
	A9	A53	A53	1600X	7443P

For PCIe-based platforms, the impact of OSs and interrupts is less significant, due to the far higher performance of the x86 host CPUs. The resulting latencies are also better than for setup (c), even though the latency for (d) and (e) now includes PCIe transfers. While the U280 is the more recent device compared to the VC709, the VC709 still reaches a lower latency. This is due to the different host machines: The VC709 is connected to a workstation, while the U280 resides in a server with a more complex PCIe subsystem, which again increases latency.

15.6 EVALUATION

To evaluate our framework, we use the different sizes of the NIPS dataset as scalable benchmarks. Each dataset comes with a corresponding trained SPN, which was created using the SPFlow library [9]. As queries, we use the large library enumerated and combined queries we generated for the empirical analysis of SPN-based CE (Table 15.1). In addition to the different SPNs, we also evaluate the two accelerator objectives (latency- vs. throughput-optimized). Finally, we also evaluate the error margins of the fixed-point encoding, depending on the number of bits used. Thus, we generate the multiple versions of the accelerators using different encodings, resulting in 56 different accelerators (7 SPNs $\times$ 2 variations $\times$ 4 encodings).

15.6.1 *Benchmarks*

While the NIPS dataset we use here is not a traditional database or cardinality estimation benchmark, it offers a unique advantage in its *scalability*. The number of each benchmark indicates the size of the underlying entries in bytes. For example, a NIPS60 entry is comprised of 60 separate values (columns) that are encoded with a single byte

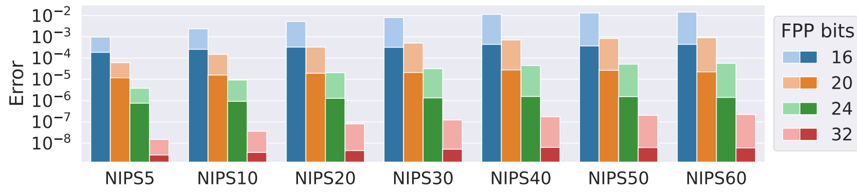


Figure 15.4: Arithmetic errors of the encoding depending on the combination of SPN and encoding. The darker bar in front is the maximum *measured* error, while the lighter bar behind is the maximum *theoretical* error.

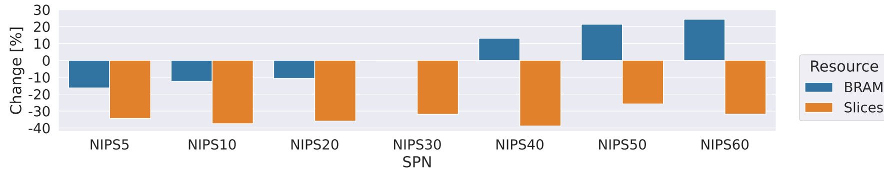


Figure 15.5: Changes in resource utilization of using a 24 bit fixed-point encoding compared against the closest matching custom floating-point (CFP) encoding from [14]. DSP utilization is identical for both encodings.

Table 15.4: SPN and Data Characteristics of the used benchmarks

	Characteristics			Data	
	$+$	$\times$	Hist.	Entry-Size	Max. Query Size
NIPS5	1	10	10	5 Bytes	20 Bytes
NIPS10	3	25	24	10 Bytes	40 Bytes
NIPS20	7	56	52	20 Bytes	80 Bytes
NIPS30	10	87	80	30 Bytes	120 Bytes
NIPS40	16	122	112	40 Bytes	160 Bytes
NIPS50	16	143	132	50 Bytes	200 Bytes
NIPS60	13	156	148	60 Bytes	240 Bytes

each. NIPS thus allows measuring error margins and throughput and latency depending when scaling the record sizes from 5...60.

The exact characteristics of the SPNs used in the evaluation are shown in Table 15.4. The Characteristics columns indicate the number of operations within the trained SPN. The Data columns describe how the structure of the training and query data. For example, NIPS50 uses a dataset comprised of 50 single-byte values per entry. The corresponding queries can thus be much bigger, since we need to pass up to two values and a corresponding query operator. For the throughput-optimized versions, we always need to pass the complete query, whereas the latency-optimized version defaults to marginalization. Thus, the latency-optimized version can be queried with as little as four bytes of data. The four bytes contain an additional byte for padding, because that simplifies the communication with the accelerator due to the memory alignment.

For the performance measurements, we use the full set of queries generated for the experimental evaluation from Table 15.1.

15.6.2 *Fixed-Point Encoding*

In a first step, we evaluate the arithmetic error introduced by the value encoding. Since it is based on fixed-point numbers, the maximum error can be derived depending on the encoding and the SPN. Additionally, we use the enumerated and generated queries to gain an empirical perspective. The results are shown in Fig. 15.4. The plot shows the *theoretical* maximum error in light color and the *empirical* maximum error in darker color for all datasets and four configurations of the fixed-point encoding. We observe that the theoretical and empirical maximum error always differ by *at least* a factor of 5. The worst case occurs if each column is queried using a range query. While this can occur in theory, it does not really make sense in the CE use-case, as more complex queries would be simplified by the DBMS query optimizer by decomposing the query into smaller ones. Additionally, it is clear that increasing the number of bits in the encoding will reduce the maximum error, since numbers can be represented more accurately. For a single dataset, adding four bits will reduce the error by an order of magnitude. Lastly, we observe an increased error towards the more complex versions of the NIPS dataset like NIPS50 and NIPS60. This is the case because the corresponding SPNs are more complex with more histograms and more operations. While more histograms also add more potential for conversion errors during accelerator generation, additional operations introduce more potential for accumulating or multiplying conversion errors. Most importantly for our application, though, the empirical error of the *arithmetic* is relatively small compared to the error introduced by the *CE* algorithm itself (cf. Section 15.4.3). Thus, narrow fixed-point encodings with 16

or 20 bits will most likely be sufficient for most practical CE use-cases. However, for the following evaluation of resource utilization, we will conservatively use a more accurate 24 bit representation.

Resource Utilization To gauge the resource efficiency of our design, we used the 24 bit fixed-point accelerators and compared them against the Custom Floating Point (CFP) variants from [14]. We synthesize for the VC709 used in that work to achieve comparable results. The changes in resource utilization from [14] to our work are shown in Fig. 15.5. The diagram shows that there are no changes in DSP utilization. This is the case since the multiplications in both variants are similar in size and can both be performed using two DSP slices per multiplication. The BRAM utilization for *smaller* SPNs is slightly reduced, but will increase for *larger* SPNs. This change is the result of three factors: 1) In our work, we need *two* probability lookups to enable the more complex operations, which increases the required memory. While this increase in BRAM utilization could potentially be alleviated by using dual-ported BRAM, this was not necessary in this work due to the overall low utilization of BRAM, as shown in Fig. 15.6. 2) The encoding used in our work is more compact, as no exponent needs to be stored. This should reduce the required BRAM by about 30%. 3) The use of accumulated histograms and the Index Determination stage allows us to store the buckets without replication. Since 1) and 2) are more or less constant, the changes are mostly due to 3), which is SPN-dependant. Lastly, we see a relatively constant reduction in Slice utilization. Overall, our accelerators are much more resource efficient than those in [14], even with the conservative 24 bit number format we used here. While BRAM utilization is increased, this is not as problematic, as overall device BRAM utilization is *below 5%* for all SPNs.

In addition to these relative numbers in comparison to prior work, Fig. 15.6 also shows the percentage utilizations of the accelerators using the 24 bit low-latency configuration on all of the different platforms. Note that the first two columns are architectures based on the Virtex-7 and Zynq-7000 fabric respectively, while the last two columns are based on the more modern UltraScale+ fabric. The available logic resources are listed in Table 15.5. As the ZC706 board uses the same FPGA device as the COSMOS+ smart SSD, we just report the ZC706 numbers here.

Note that the Ultra96v2 and ZC706 devices are SoCs, which also contain software-programmable ARM cores and other hardened IP, e.g., memory controllers, which reduces the number of configurable FPGA resources required for integrating the accelerators into a bigger system. Thus, utilizations on those devices is dominated by the accelerators themselves, while the designs on the VC709 and Alveo U280 also implement memory controllers and PCIe cores in soft logic, which increases configurable resource utilization.

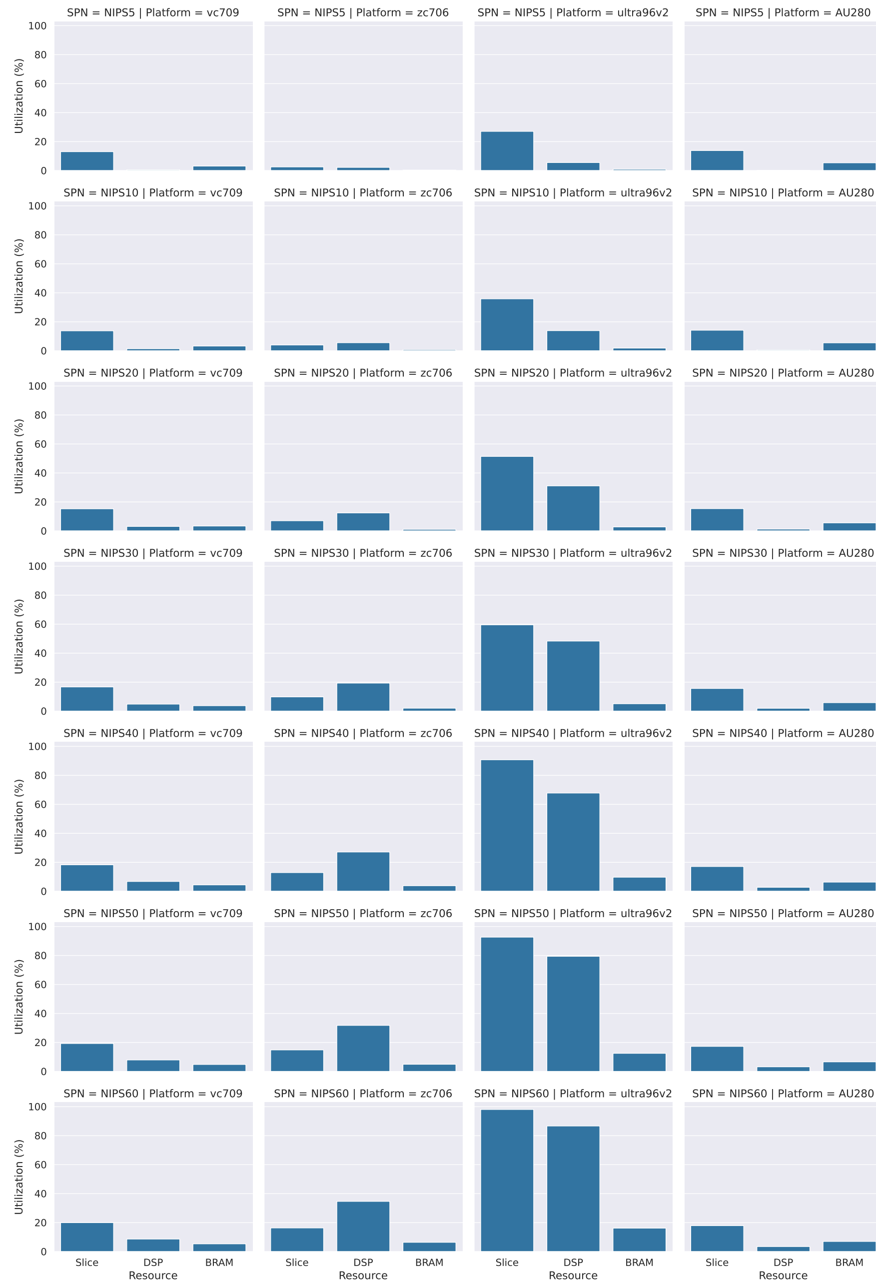


Figure 15.6: FPGA Resource Utilizations of the different accelerators in a 24 bit low-latency configuration on the different devices.

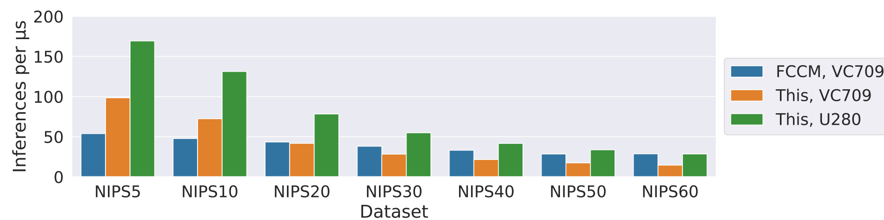


Figure 15.7: Throughput of the throughput-optimized accelerator variants on VC709 and Alveo U280 in comparison to single-core prior work on VC709 published at FCCM [14].

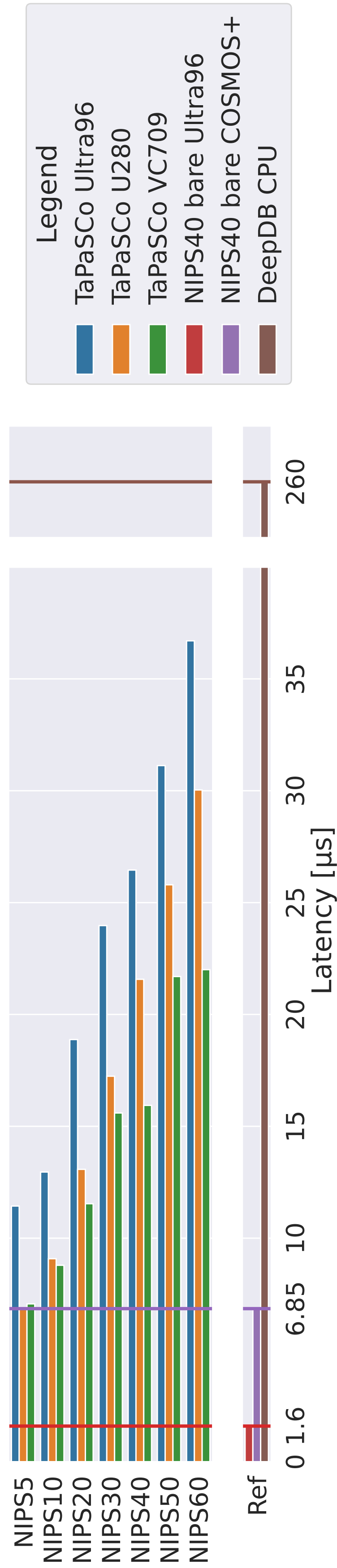


Figure 15.8: End-to-End inference latencies (less is better) of TaPaSCo-based architectures in comparison against baremetal NIPS40-accelerators on Ultra96 and COSMOS+ and the best reference CPU implementation of an RSPN from prior work [7].

Table 15.5: FPGA Resources of the different platforms used in Evaluation

	Slices	DSPs	BRAMs
Alveo U280	162960	9024	2016
Ultra96v2	8820	360	216
VC709	108300	3600	1470
ZC706 / COSMOS+	54650	900	545

Apart from this, the diagram shows the increasing size of the corresponding accelerators, going from the relatively compact NIPS5 to the much bigger NIPS60. Especially on the smaller Ultra96v2 (third column from the left), we can clearly see the cost of the bigger SPNs. In contrast, the other devices (VC709 and Alveo U280) would easily be able to hold even bigger SPN accelerators, or even multiple replicas of the same accelerator.

The latter could be used to maintain throughput even when performing CE for more complex queries requiring *multiple passes* over the SPN, as discussed in Section 15.4.1). In this manner, these, then longer running, CE operations could be *distributed* across the available SPN replicas.

Lastly, Table 15.6 reports the absolute utilizations on all four platforms for both the latency- and the throughput-optimized versions of all accelerators. These numbers especially highlight the difference between SoC- and PCIe-based architectures: For Soc-based architectures the additional overhead for DRAM access incurs little to no resource overhead, while PCIe-based devices require a lot more Slices to implement corresponding memory infrastructure.

15.6.3 *Performance*

For brevity, we focus our performance evaluation on accelerator variants just using the conservative 24 bit fixed-point encoding, as the encoding has no relevant impact on performance.

Latency First, we want to look at end-to-end latency. The measurements for the baremetal implementations of NIPS40 from Table 15.3 are used as reference. Additionally, we rely on latency measurements that reflect real-world implementations, including all necessary data movements. We achieve this by using the TaPaSCo-based system integrations (cf. test setups (c)-(e)), which also implies additional overhead for the operating system. The corresponding measurements are shown in Fig. 15.8. The figure shows that bigger SPNs have higher latencies, which can be attributed to larger data-transfers, as the actual computation takes less than 1 µs. Interestingly, this behavior is similar for PCIe-based and SoC-based platforms. While the SoC-based with their shared memory do not require the data transfers necessary for the PCIe-attached FPGAs, the slower ARM cores on the SoCs lead to

a loss in overall performance compared to the fast x86 cores for the PCIe hosts. While the latencies are high compared to the baremetal implementations from Table 15.3, we still outperform prior work [7] by *up to 40x*. Even for the biggest SPN (NIPS60), the speed-up is still more than an order of magnitude. As the prior work did not give detailed statistics, we compare against their *overall best* reported result of 260 µs for a fair comparison.

Throughput The CPU-based prior work [7] does not report throughput of their implementation, but we are still able to compare our throughput-optimized accelerators against prior work on regular SPN inference. While the original work [14] has been refined for maximum throughput using HBM [20] and 100G networking [6], the corresponding speedups are achieved by replicating accelerators and by overlapping data-transfers using multi-threading. While corresponding techniques could be applied to our accelerators as well, it makes the comparison between accelerators difficult, as it is hard to isolate the impact of multi-threading and overlapped data-transfers. We instead opt for a single-accelerator comparison against [14]. While the prior work is focused exclusively on inference, the underlying calculations are identical, as a singular inference pass over the SPN is performed. Additionally, we focus on PCIe-based accelerators, since the memory on the Ultra96v2 is much slower and hinders performance.

The throughputs of the different SPNs are shown in Fig. 15.7. Please note that the size of data-transfers is different for our approach: Since the histograms are not limited to equality anymore, we need to transport 4x more input-data per inference. The number of the NIPS benchmark indicates the number of inputs per inference run. In prior work, each input was just a *single* byte. Here, it is *four* bytes to enable the additional operations. For the output-data it is the other way around: In prior work each output was 8 bytes, in our case it is just 4 bytes. Due to the larger size of the input-data, this should still have a significant performance impact. Our expectation was an achievable throughput of roughly 0.25x of the prior work, but this is not the case. The reasons for this are improvements in the TaPaSCo API and architecture. Instead of using general-purpose AXI-Stream interconnects, we now rely on more optimized vendor IP. This allows us to outperform prior work for smaller SPNs. Using the PCIe Gen3 x16 based Alveo U280 with its newer UltraScale+ device increases this advantage even further, achieving a 3x increased throughput for NIPS5. Even for the biggest benchmark, we are still able to keep up with the much less data-intensive prior work.

15.7 CONCLUSION

In this work, we built on the idea of using SPNs for CE that was originally proposed in [7], and introduced a framework that exploits FPGAs

Table 15.6: Absolute FPGA Resource Utilizations on different platforms

	VC709			ZC706			Ultra96v2			Alveo U280		
	Slices	DSPs	BRAMs	Slices	DSPs	BRAMs	Slices	DSPs	BRAMs	Slices	DSPs	BRAMs
Latency-Opt., 24 bits												
NIPS5	56572.0	46.0	20.0	5577.0	2.0	20.0	2295.0	2.0	20.0	22552.0	108.5	23.0
NIPS10	59634.0	48.0	50.0	8758.0	4.0	50.0	3279.0	4.0	50.0	23106.0	110.5	53.0
NIPS20	66191.0	50.0	112.0	15362.0	6.0	112.0	4584.0	6.0	112.0	25084.0	112.5	115.0
NIPS30	72261.0	55.0	174.0	21449.0	11.0	174.0	5833.0	11.0	174.0	25448.0	117.5	177.0
NIPS40	79081.0	65.0	244.0	28181.0	21.0	244.0	7919.0	21.0	244.0	27852.0	127.5	247.0
NIPS50	83488.0	71.0	286.0	32494.0	27.0	286.0	8316.0	27.0	286.0	28192.0	133.5	289.0
NIPS60	86590.0	79.0	312.0	35727.0	35.0	312.0	8705.0	35.0	312.0	29216.0	141.5	315.0
Throughput-Opt., 24 bits												
NIPS5	71328.0	46.0	20.0	16023.0	2.0	20.0	4339.0	2.0	20.0	24939.0	108.5	23.0
NIPS10	74322.0	48.0	50.0	19341.0	4.0	50.0	5044.0	4.0	50.0	25867.0	110.5	53.0
NIPS20	80332.0	50.0	112.0	25387.0	6.0	112.0	6349.0	6.0	112.0	27686.0	112.5	115.0
NIPS30	96914.0	55.0	174.0	41552.0	11.0	174.0	8800.0	11.0	174.0	32902.0	117.5	177.0
NIPS40	91402.0	65.0	244.0	36375.0	21.0	244.0	8752.0	21.0	244.0	31068.0	127.5	247.0
NIPS50	118506.0	71.0	286.0	63177.0	27.0	286.0	8734.0	27.0	286.0	36823.0	133.5	289.0
NIPS60	112269.0	79.0	312.0	57150.0	35.0	312.0	8756.0	35.0	312.0	35525.0	141.5	315.0

to offload and accelerate CE. We have shown that independently of the required architecture (traditional DBMS vs. NDP), FPGA-based CE can easily outperform CPU-based prior work.

With regard to latency, using a PCIe-based accelerator for a traditional DBMS can make CE up to 40x faster. In embedded NDP setups, such as a smart computational storage platform, inference latencies can be reduced to the microsecond range on recent SoCs. While this shows that FPGA-based CE offers lower latencies than CPUs, the main advantage of FPGAs lies in throughput-oriented scenarios, such as the batch-processing of many CEs that can be exploited for traditional DBMS.

Since the CPU-based prior work does not report throughput, we instead compare against prior work on regular SPN inference. For smaller SPNs, our approach is able to outperform prior work even though CE requires the transfer of approximately 4x more data. For bigger SPNs, the throughput of prior work is only 2x of ours, despite the bigger data-transfers. On more recent hardware with PCIe Gen3 x16, performance is similar for the biggest SPN.

Overall, the accelerators are able to perform tens of inferences per µs (or tens of millions of inferences per s). We also show the advantages of using a simpler number encoding in use-cases were relative errors are not as important.

ACKNOWLEDGEMENTS

This manuscript was partially revised for grammar and clarity using the OpenAI Large Language Model GPT-4 from a human-written draft. None of the contents were auto-generated by the LLM. This work was partially funded by the Hessian Center for Artificial Intelligence, Germany.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS 1998*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: *Database Machines*. 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

[3] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD 2013*. 2013.

[4] Laura I Galindez Olascoaga, Wannes Meert, Nimish Shah, Marian Verhelst, and Guy Van den Broeck. "Towards Hardware-Aware Tractable Learning of Probabilistic Models." In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[5] Hazar Harmouch and Felix Naumann. "Cardinality Estimation: An Experimental Survey." In: *Proc. VLDB Endow.* 11.4 (Dec. 2017), pp. 499–512. ISSN: 2150-8097. DOI: 10.1145/3186728.3164145. URL: https://doi.org/10.1145/3186728.3164145.

[6] Marco Hartmann, Lukas Weber, Johannes Wirth, Lukas Sommer, and Andreas Koch. "Optimizing a Hardware Network Stack to Realize an In-Network ML Inference Application." In: *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2021.

[7] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, Not from Queries!" In: 13.7 (Mar. 2020), pp. 992–1005. ISSN: 2150-8097. DOI: 10.14778/3384345.3384349.

[8] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *Applied Reconfig. Comp.* 2019.

[9] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. *SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks*. 2019.

[10] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Xiaoting Shao, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. "Random Sum-Product Networks: A Simple but Effective Approach to Probabilistic Deep Learning." In: *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI)*. 2019.

[11] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: a New Deep Architecture." In: *Proc. of UAI* (2011).

[12] Nimish Shah, Laura I. Galindez Olascoaga, Wannes Meert, and Marian Verhelst. "Acceleration of probabilistic reasoning through custom processor architecture." In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020, pp. 322–325. DOI: 10.23919/DATE48585.2020.9116326.

[13] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *36th Intl. Conf. on Computer Design (ICCD)*. Oct. 2018.

[14] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. "Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs." In: *Intl. Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020.

[15] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. "Cosmos+ OpenSSD: A NVMe-based Open Source SSD Platform." In: *Flash Memory Summit* (2016).

[16] Tobias Vinçon, Christian Knödler, Arthur Bernhardt, Leonardo Solis-Vasquez, Lukas Weber, Andreas Koch, and Ilia Petrov. "Result-Set Management for NDP Operations on Smart Storage." In: *18th International Workshop on Data Management on New Hardware (DaMoN)*. 2022. DOI: 10.1145/3533737.3535097.

[17] Lukas Weber, Lukas Sommer, Julian Oppermann, Alejandro Molina, Kristian Kersting, and Andreas Koch. "Resource-Efficient Logarithmic Number Scale Arithmetic for SPN Inference on FPGAs." In: *Intl. Conference on Field-Programmable Technology (FPT)*. 2019.

[18] Lukas Weber, Lukas Sommer, Leonardo Solis-Vasquez, Tobias Vinçon, Christian Knödler, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. "A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems." In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 136–143. DOI: 10.1109/IPDPSW52791.2021.00028.

[19] Lukas Weber, Tobias Vincon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. "On the necessity of explicit cross-layer data formats in near-data processing systems." In: *Distributed and Parallel Databases* (2021). DOI: 10.1007/s10619-021-07328-z.

[20] Lukas Weber, Johannes Wirth, Lukas Sommer, and Andreas Koch. "Exploiting High-Bandwidth Memory for FPGA- Acceleration of Inference on Sum-Product Networks." In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022.