Method for Adaptation of Algorithms to GPU Architecture

Vadim Bulavintsev[1], Dmitry Zhdanov[1]

[1]ITMO University, 49 Kronverksky Pr., St. Petersburg, 197101, Russia

Abstract

We propose a generalized method for adapting and optimizing algorithms for efficient execution on modern graphics processing units (GPU). The method consists of several steps. First, build a control flow graph (CFG) of the algorithm. Next, transform the CFG into a tree of loops and merge non-parallelizable loops into parallelizable ones. Finally, map the resulting loops tree to the tree of GPU computational units, unrolling the algorithm's loops as necessary for the match. The mapping should be performed bottom-up, from the lowest GPU architecture levels to the highest ones, to minimize off-chip memory access and maximize register file usage. The method provides programmer with a convenient and robust mental framework and strategy for GPU code optimization. We demonstrate the method by adapting to a GPU the DPLL backtracking search algorithm for solving the Boolean satisfiability problem (SAT). The resulting GPU version of DPLL outperforms the CPU version in raw tree search performance sixfold for regular Boolean satisfiability problems and twofold for irregular ones.

Keywords

GPU, SIMD, control flow graph, loop optimization, loop unrolling, DPLL,

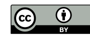
1. Introduction

Modern graphics processing units (GPU) execute computer vision and "big data" processing tasks efficiently. Those tasks belong to the class of "embarrassingly parallel" problems, which perfectly matches the "single instruction, multiple data" (SIMD) [1] hardware architecture of GPU. The ongoing boom in Machine Learning (ML) is fueled by the positive feedback loop of researchers - software industry interaction. Researches run ML models on GPUs, pointing industry engineers to implement the computational primitives the former can reuse. This cycle resulted in rapid development of sophisticated high-level ML libraries, such as TensorFlow [2] and others. However, there are many non-ML algorithms that could benefit from executing on a GPU. Unfortunately, adapting a non-ML algorithm to the GPU platform is generally hard since the platform is complex to program. Moreover, it can be hard to get good performance out of a GPU because of inefficient execution of branches by SIMD units, different types of device memory available, and many other GPU hardware quirks. These types of hardware peculiarities are typically abstracted in case of CPU programming, which makes it impossible to directly translate CPU code to GPU in most cases. GPU programming research typically

GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27–30, 2021, Nizhny Novgorod, Russia

v.g.bulavintsev@gmail.com (V. Bulavintsev); ddzhdanov@mail.ru (D. Zhdanov)

0000-0003-3099-1442 (V. Bulavintsev); 0000-0001-7346-8155 (D. Zhdanov)


© 2021 Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3027/paper100.pdf

focuses on optimizing a single narrow aspect of getting good performance from GPU, leaving the big picture out of scope. As a result, textbooks and guides on GPU programming drown the reader with highly detailed descriptions of architecture and programming techniques and tricks, forgetting to provide a generalized mental model of the device - software interaction and strategy for code optimization.The present work seeks to fill this gap by formalizing a method and strategy for adapting and optimizing arbitrary algorithms to the GPU platform.

The paper structure consists of the following sections:

Section 1 consists of a survey of prior research regarding GPU code optimization;

Section 2 explains the philosophy behind the method;

Section 3 describes the method as a sequence of steps;

Section 4 provides a detailed example of applying the method to adapt a complex algorithm to GPU platform, assessing the resulting performance;

Section 5 concludes the paper by briefly discussing the method's performance and prospects.

2. Prior works

Since early 2000s, the concept of general-purpose GPU programming made a leap from a curiosity to the "magic sauce" behind the ongoing industrial AI revolution. Many excellent GPU programming platforms were released in this period, ranging from vendor-specific (i.e. CUDA [3]) to hardware-independent, open-standard based OpenCL [4]. Later, domain-specific SDKs and platforms, such as TensorFlow [2] arrived.

While trying to abstract the hardware details, these platforms either force the programmer to use rigid library primitives (e.g. matrix multiplication) or add too much abstraction trying to encompass too many possible hardware architectures [5]. This lack of middle ground incentivizes the GPU research community towards solving the problem with one of the following strategies:

- design a perfect programming language for programming GPUs [6];
- make the compiler smart enough to optimize the GPU code without intervention from the programmer [7], [8], [9];
- strike the balance between the two strategies above by extending an existing language with hints that would make it play well with a given set of compiler optimizations [10].

To our experience, the literature and didactic materials on the topic of GPU programming are lacking in the description of the big picture, focusing on technical details instead. In the present work we intend to bridge this gap by providing the programmer with a robust mental model of the optimization process.

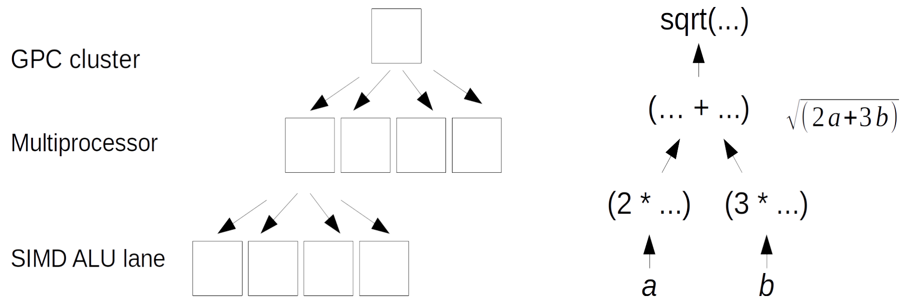


Figure 1: GPU as a tree of computational units vs. formula as a tree of computations

3. Method idea

3.1. Computation as a tree of operations

An algorithm, by definition, is a sequence of well-defined actions required to solve a particular problem. Some algorithms may involve a very high number of actions and thus must be executed by a computer. However, (almost) all algorithms are created by humans and expressed in human-readable forms, such as mathematical formulae or programming languages. To understand and manipulate complex algorithms, humans break those expressions down into smaller subroutines and compress repeating steps using loops [11]. Every algorithm written within the paradigm of structured programming [12], as well as any mathematical formula, can be expressed as a tree of subroutines or subformulae (Figure 1). Also, every loop in an algorithm can be unrolled into a fixed sequence[1] of repeating operations [14]. Thus, all finite algorithms or formulae can be unfolded into a human-comprehensible tree of operations.

3.2. GPU as a tree of computational units

Modern GPUs are designed to execute algorithms that primarily consist of a very high number of simple independent operations (e.g. floating-point multiplications and additions). GPU architecture can be split down into several organizational levels (OL), containing several computational units of the same type, such as ALUs or multiprocessors. Thus, every modern GPU can be represented as a tree of computational units 1.

In the GPU code, OLs existence is evident in the form of explicit and implicit synchronization primitives, warp shuffle instructions, atomic memory access instructions, thread identifiers and compiler intrinsics.

3.3. Adapting software to hardware

CPUs, GPUs and FPGAs represent different strategies of increasing performance of code execution:

[1] Of course, there exist algorithms with an unknown number of operations and programs that can never stop[13]. For simplicity, we only talk about algorithms consisting of a limited number of operations in this paper.

CPU tries to *be smart* about executing programs, with long instruction pipelines and branch predictors powering the superscalar paradigm. In a sense, CPUs try to do depth-first visiting of the computation tree;

GPU instead *relies on the programmer* or compiler exposing the parallelism of the executed algorithm. GPU's strategy can be loosely matched to breadth-first visiting of the computation tree;

FPGA *becomes the algorithm*, reconfiguring the hardware to match the computation tree.

Superficially, FPGA strategy of reconfiguration seems more promising in terms of performance. But there are reasons why GPUs are much more popular at the moment. One downside of FPGAs is those come at an increased price due to redundant interconnect fabric and physical limitations. Also, GPUs are mass-consumer devices, further pushing down their cost. Another point is that there are many more software engineers than there are hardware engineers. Overall, matching software to hardware makes for a better strategy than the other way around.

3.4. Limitations of GPU architecture

A GPU consists of several multiprocessors of SIMD architecture [1] accessing the same onboard memory, possibly through a small shared caching unit. To efficiently execute highly parallelizable tasks, GPU architecture sacrifices in flow control logic and memory access latency. Typical GPU multiprocessors consist of 16-32 wide SIMD ALUs, which cannot execute divergent branches of a program in parallel [3]. To hide memory access latency, requests to onboard memory are pipelined by running multiple thread batches on the same multiprocessor. While a single thread batch (named "warp" or "wavefront") runs, the other batches sleep. The registry file is shared by all the threads running on the same multiprocessor, resulting in a tradeoff of registry pressure vs the number of pipelined warps. Further complicating GPU programming, its memory controller is optimized to fetch data in continuous ranges (so-called *coalesced access*). Deviating from this pattern can slow down the program execution considerably [3].

4. Method description

Getting good performance from a GPU for a given algorithm is a matter of assigning the algorithm tree to the tree of the GPU's computational blocks in the most efficient way possible. The proposed method consists of the following steps (Figure 2):

Build the control flow graph (CFG) of the algorithm

Transform the CFG into a tree of parallelizable loops

Map the tree of loops to the tree of GPU computational units, according to the limitations of the GPU hardware.

The following subsections describe each step in detail.

4.1. Construct the control flow graph

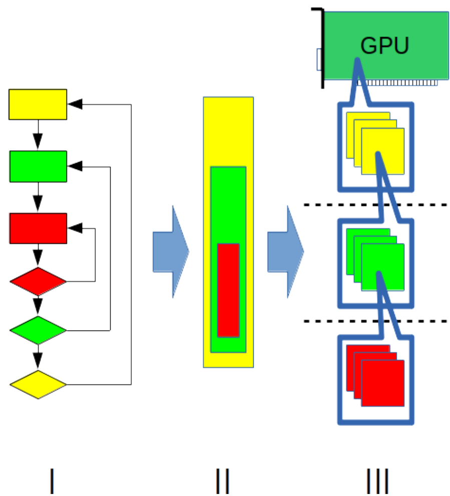


Figure 2: Algorithm adaptation steps

The Control Flow Graph (CFG) of a program consists of *basic blocks* (BB) connected by directed edges of control flow. Each BB represents a sequence of instructions with a single input and a single output point. CFG starts with the *entry block* and ends with the *exit block*. Thus, CFG represents all possible paths of execution of the associated program [15]. The CFG is a subtype of a flowchart and thus can be built manually. As an alternative the CFG can be built by an automated code analysis tool.

4.2. Build the tree of parallelizable loops

The step begins by representing the CFG as a hierarchy of natural loops. Informally, a natural loop is defined as a cycle formed by a single back edge, such that no edges from other parts of the CFG are leading into the loop body. A CFG that contains only natural loops is a *reducible* CFG. Any CFG can be transformed into a reducible one by node splitting. The splitting can be performed either by hand or automatically, e.g. by a compiler framework [16]. For the rest of the paper, we will discuss only reducible CFGs and natural loops.

After identifying natural loops, the CFG should be reduced to a tree of such loops. We are particularly interested in the loops without data dependencies between iterations, allowing for easy parallelization. Note that sometimes it is possible to transform a loop with data dependencies between iterations into a loop without. A formal way to do this is applying transformations of the loops in the polyhedral model [17]. To make the consequent matching step more straightforward, we join adjacent nodes representing loops with data dependencies. The rationale here is that if it is impossible to parallelize a pair of nested loops, we could as well represent those as a single loop or as a single, un-parallelizable BB. Next, for every parallelizable loop, the programmer should note its number of iterations. If the exact number of iterations for a loop is unknown, the programmer must estimate at least an approximate count of iterations relative to the upper-level loops. As the result, the programmer should get a tree of parallelizable loops.

4.3. Map the algorithm tree to the hardware tree

At this step, the programmer (or an automated tool) should be able to map the algorithm tree to the hardware tree. The loop unroll transformation [15] could be applied to split the nodes of the algorithm tree as necessary. After the initial mapping is done, the primary optimization strategy is to position the loop nodes and their variables as deep into the hardware tree as possible. The idea is that the lower the level of a computational unit, the closer it is to its associated memory store and the faster the computation. However, lower-level computational units typically have more limitations, such as the SIMD branching problem or gather-scatter limitations. Also, the

amount of storage associated with lower computational units become smaller.

Mapping a loop to a hardware level means assigning each iteration of the loop in a way that avoids waiting for results of computation performed by the other units of the same level. For example, mapping the loop of multiplication of v_i elements of the vector $\mathbf{V}$ to a scalar s to the *warp lanes level* means that lane k of a warp will execute $r_k = sv_k$, storing the result into the variable r_k local to the lane. Both v_k and r_k could be stored either in the registry file or in the onboard memory, but the operation stays local to the lane since no lane will have to *wait* for results from others.

One good example is mapping operations of a convolutional neural network (CNN) to a GPU. For a simple CNN, it is even possible to match every neuron to a SIMD lane one-to-one naively. That makes sense since CNNs' deal with reducing visual images into a small number of logical symbols, basically reversing the purpose that GPUs were initially designed for.

5. Algorithm adaptation example

To demonstrate our method in action, we adapt the DPLL algorithm to the GPU platform. DPLL is a widely known backtracking search algorithm for solving the Boolean formula satisfiability problem [18]. The algorithm features multiple nested loops, complex control flow and intensive memory access. To this moment, authors are not aware of any implementation of DPLL that runs entirely on a GPU.

5.1. DPLL algorithm description

The DPLL algorithm is the most popular algorithm for deciding the satisfiability of a Boolean problem expressed in the conjunctive normal form (CNF). Since the time of its discovery in 1961 [18], DPLL was enhanced in every aspect, yet the core idea remained the same:

- guess a variable and assign a value to it;
- simplify the problem according to the guess
- repeat the above steps until either the solution is found or a contradiction is encountered, in which case
- backtrack and try a different value for the guessed variable

Effectively, DPLL is a tree walk algorithm (Figure 3). To avoid complicating the example we only discuss the basic DPLL here.

5.2. Step 1, building a CFG

A simplified CFG for DPLL algorithm is shown at Figure 4. 99% of computation happens inside the Boolean Constraint Propagation (BCP) procedure, which is thoroughly optimized in modern SAT solvers [19]. The idea of BCP is to infer as much as possible from a single guessed variable. For every variable guessed, BCP looks into each clause that contains the variable's literals that could render the formula unsatisfiable. If the clause contains only a single free literal after the assignment, that literal's variable is added to the propagation queue. If BCP results in an

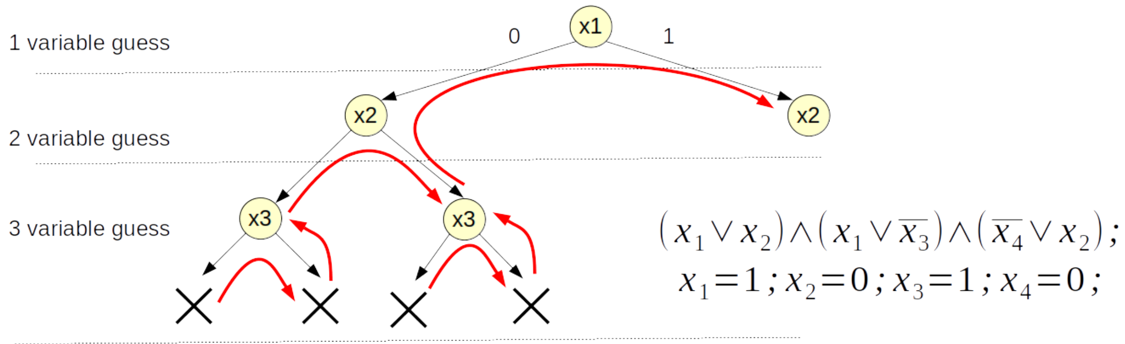


Figure 3: Tree walk in DPLL

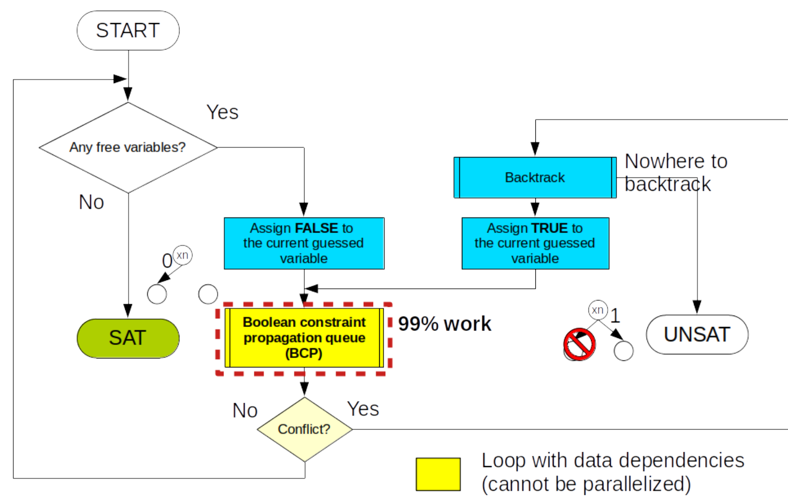


Figure 4: DPLL algorithm flowchart

assignment conflict, the solver invokes the backtracking procedure. Otherwise, DPLL proceeds to guess variables until every clause in the formula is satisfied.

5.3. Step 2, building a tree of loops

DPLL CFG can be viewed as a hierarchy of loops with a (generally) unpredictable number of iterations:

1. the top-level loop L_1 of guessing a variable value, i.e. tree walk;
2. the BCP loop L_2 for simplifying the formula;
3. checking the clauses of a variable - loop L_3;
4. checking the literals of a clause for conflicts or logic inference opportunities - loop L_4.

Only L_1 iterations can be performed in parallel since it is trivial to break the search tree into any desired number of sub-trees[20]. L_2 iterations are interdependent because of every iteration

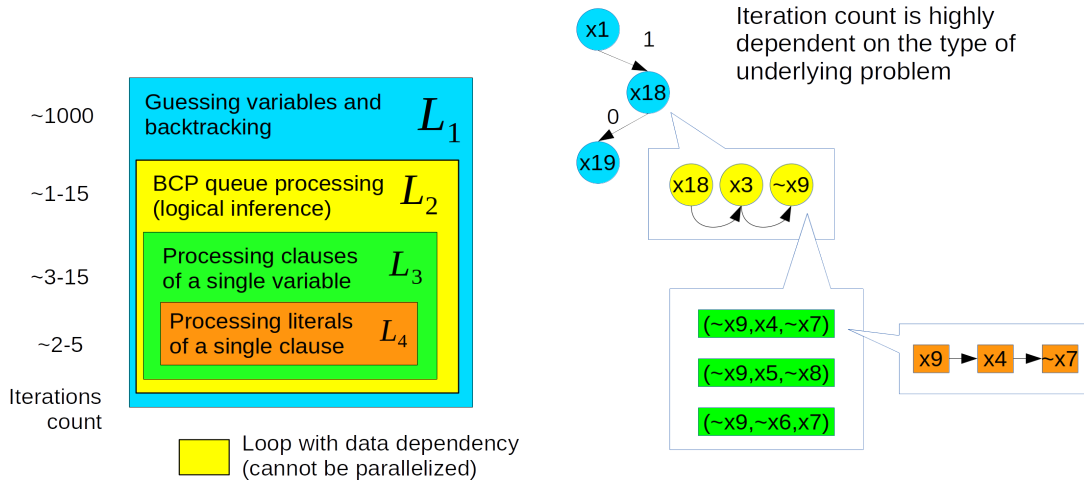


Figure 5: Loops hierarchy in DPLL

changing the state of the formula's variables. Finally, L_3 and L_4 iterations can be parallelized. The resulting tree contains just a single branch in each level, making it look like a hierarchy (Figure 5).

Loops' iteration counts are highly dependent on the nature of the underlying problem expressed by the CNF. Our estimation of the average iteration count is based on typical parameters of problems used in the yearly SAT race competition [21].

5.4. Step 3, mapping algorithm tree to GPU tree

For our analysis, the most important aspects of the architecture are the warp size and the maximum number of in-flight warps. Let us consider different variants of assigning the algorithm loops to a GPU.

Take 1 We start by trying to fit as many loop levels to as low a level of the GPU as possible to utilize the fast on-chip memory. Here, we are naively putting all the loops $L_1..L_4$ at the warp lane level. The result is too much state data kept by each thread, which does not fit into the on-chip memory and registry file.

Take 2 To decrease the memory usage, we try to move $L_1..L_3$ a step up from the lane level to the warp level. The move helps with the memory problem because all the common parts (i.e. CNF clauses, Boolean values states) are now shared by a single warp, leaving only literals-check specific data to the lane level. However, there are often not enough literals to fill a single warp, which results in many lanes idling.

Take 3 We can do better by *combining* L_3 with L_4 and assigning the result to the lane level, while keeping L_1 and L_2 at the warp level. Still, the number of L_4 iterations multiplied

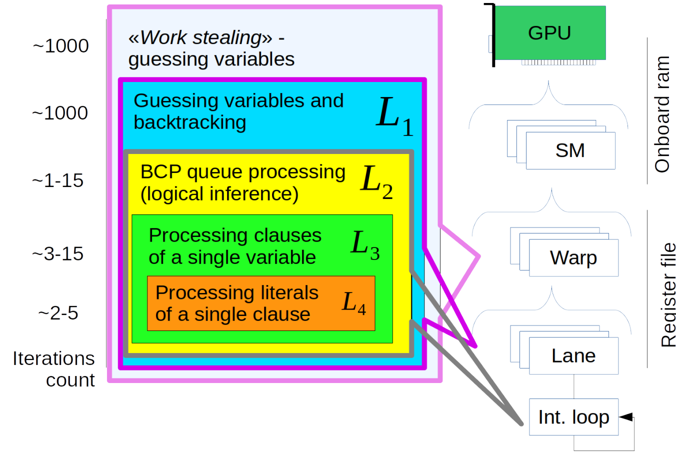


Figure 6: Matching DPLL loops hierarchy to GPU computational layers

by L_3 iterations is not always enough to fill more than a single warp. Unfortunately, our model shows that L_2 cannot be parallelized, so we cannot merge it with $L_3..L_4$. Also, the L_1 can exhaust its assigned iterations pretty fast, leaving the whole multiprocessor idle.

Take 4 To solve the problem of idle multiprocessors, we add a specialized L_0 loop of work-stealing [22], so threads of a warp can now dynamically ask each other for search tree branches that still must be checked. The same kind of exchange happens between warps. Also, to avoid lanes idling because SIMD ALUs are out of work, we apply the nested loops merge transform [23] to merge L_2 and L_3. However, the merge requires us to parallelize L_1, getting us back to the "*Take 1*" variant with added work stealing (Figure 6). To solve the registry pressure problem, we store common CNF data at the GPU level.

5.5. Performance of the GPU-adapted DPLL

We implemented[2] the "*Take 4*" version of DPLL described above for NVIDIA GPUs in CUDA C language. To estimate the efficiency of the adaptation method, we also built a CPU version of the same algorithm. Table 1 contrasts the Boolean constraints propagation speed of the GPU-adapted DPLL variant running on a GPU to the performance of the same code running on a CPU. The performance is measured in millions of literal checks per second.

- the classic Pigeonhole problem [24],
- a synthetic benchmark with a regular structure [24].

As Table 1 shows, the adaptation procedure enabled efficient execution of the DPLL algorithm on a GPU. However, GPU performance is very dependent on the structure of the underlying problem. The CPU implementation of DPLL is much more robust to changes in the problem structure. Our earlier observation explains this effect with the fact that CPUs are designed to be much more adaptable than GPUs (see Section 2).

[2]https://github.com/ichorid/ringsat

Table 1
GPU vs CPU performance of the adapted DPLL algorithm

Benchmark problem	Constraints propagation speed (mln literals / second)	
	CPU (Core i7-8700k, 1 thread)	GPU (RTX2060)
"hole8.cnf" (pigeonhole problem)	46	**166 (+260%)**
"dubois25.cnf" (synthetic problem)	55	**436 (+690%)**

6. Conclusion

We proposed a method for adapting the algorithm to GPU architecture and demonstrated it by adapting a complex search algorithm (DPLL) to GPU. The method is based on the mental model of matching the computation tree to the hardware tree. The model allows the programmer to convert an algorithm to GPU hardware while avoiding iterations of trial and error and guiding architectural choices towards optimal performance. The method is not bound to any particular programming language, but instead based on common notions of loops and structured programming, providing a convenient mental framework for GPU code optimization. One possible direction for future research is creating extensions for existing profilers and IDEs to provide the programmer with hints about the estimated performance of the GPU code.

References

[1] M. J. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers C-21 (1972) 948–960. URL: http://ieeexplore.ieee.org/document/5009071/. doi:10.1109/TC.1972.5009071.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: A System for Large-Scale Machine Learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA, 2016, pp. 265–283. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[3] NVIDIA, P. Vingelmann, F. H. Fitzek, CUDA, release: 10.2.89, 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[4] J. E. Stone, D. Gohara, G. Shi, OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems, Computing in Science Engineering 12 (2010) 66–73. doi:10.1109/MCSE.2010.69.

[5] R. Dolbeau, F. Bodin, G. C. de Verdiere, One OpenCL to rule them all?, in: 2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS), IEEE, 2013, pp. 1–6.

[6] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, R. David, E. Hajiyev, PENCIL: A Platform-Neutral Compute Intermediate Language for

Accelerator Programming, in: 2015 International Conference on Parallel Architecture and Compilation (PACT), IEEE, San Francisco, CA, 2015, pp. 138–149. URL: https://ieeexplore.ieee.org/document/7429301/. doi:10.1109/PACT.2015.17.

[7] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, F. Catthoor, Polyhedral parallel code generation for CUDA, ACM Transactions on Architecture and Code Optimization 9 (2013) 1–23. URL: https://dl.acm.org/doi/10.1145/2400682.2400713. doi:10.1145/2400682.2400713.

[8] J.-Y. Liou, X. Wang, S. Forrest, C.-J. Wu, GEVO: GPU Code Optimization Using Evolutionary Computation, ACM Transactions on Architecture and Code Optimization 17 (2020) 1–28. URL: https://dl.acm.org/doi/10.1145/3418055. doi:10.1145/3418055.

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, A. Krishnamurthy, TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA, 2018, pp. 578–594. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[10] N. A. Kataev, S. A. Chernykh, Automation of program parallelization in SAPFOR, in: Scientific Service and Internet: Proceedings of the 22nd All-Russian Scientific Conference (September 21-25, 2020, online), 2020, pp. 362–376. URL: https://keldysh.ru/abrau/2020/theses/24.pdf. doi:10.20948/abrau-2020-24.

[11] H. Abelson, G. J. Sussman, Structure and interpretation of computer programs, The MIT Press, 1996.

[12] C. Böhm, G. Jacopini, Flow diagrams, turing machines and languages with only two formation rules, Communications of the ACM 9 (1966) 366–371. URL: https://dl.acm.org/doi/10.1145/355592.365646. doi:10.1145/355592.365646.

[13] P. Bernays, Alonzo Church. An unsolvable problem of elementary number theory. American journal of mathematics, vol. 58 (1936), pp. 345–363., Journal of Symbolic Logic 1 (1936) 73–74. URL: https://www.cambridge.org/core/product/identifier/S0022481200038998/type/journal_article. doi:10.2307/2268571.

[14] A. V. Aho, J. D. Ullman, Principles of compiler design, Addison-Wesley series in computer science and information processing, world student series ed ed., Addison-Wesley, Reading, Mass., 1977.

[15] F. E. Allen, Control flow analysis, ACM SIGPLAN Notices 5 (1970) 1–19. URL: https://dl.acm.org/doi/10.1145/390013.808479. doi:10.1145/390013.808479.

[16] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, San Jose, CA, USA, 2004, pp. 75–86. URL: http://ieeexplore.ieee.org/document/1281665/. doi:10.1109/CGO.2004.1281665.

[17] C. Lengauer, Loop parallelization in the polytope model, in: G. Goos, J. Hartmanis, E. Best (Eds.), CONCUR'93, volume 715, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 398–416. URL: http://link.springer.com/10.1007/3-540-57208-2_28. doi:10.1007/3-540-57208-2_28, series Title: Lecture Notes in Computer Science.

[18] M. Davis, H. Putnam, A Computing Procedure for Quantification Theory, Journal of the ACM 7 (1960) 201–215. URL: https://dl.acm.org/doi/10.1145/321033.321034. doi:10.1145/321033.321034.

[19] N. Eén, N. Sörensson, An Extensible SAT-solver, in: G. Goos, J. Hartmanis, J. van Leeuwen, E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, volume 2919, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 502–518. URL: http://link.springer.com/10.1007/978-3-540-24605-3_37. doi:10.1007/978-3-540-24605-3_37, series Title: Lecture Notes in Computer Science.

[20] O. S. Zaikin, S. E. Kochemazov, On black-box optimization in divide-and-conquer SAT solving, Optimization Methods and Software (2019) 1–25. URL: https://www.tandfonline.com/doi/full/10.1080/10556788.2019.1685993. doi:10.1080/10556788.2019.1685993.

[21] T. Balyo, M. Heule, M. Jarvisalo, SAT Competition 2016: Recent Developments, Proceedings of the AAAI Conference on Artificial Intelligence 31 (2017). URL: https://ojs.aaai.org/index.php/AAAI/article/view/10641.

[22] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, Journal of the ACM (JACM) 46 (1999) 720–748. Publisher: ACM New York, NY, USA.

[23] V. Bulavintsev, Flattening of data-dependent nested loops for compile-time optimization of gpu programs, International Journal of Open Information Technologies 7 (2019) 7–13.

[24] H. H. Hoos, T. Stützle, SATLIB: An online resource for research on SAT, Sat 2000 (2000) 283–292.