LHCb Kalman filter cross architecture studies

Daniel Hugo Cámpora Pérez

Universidad de Sevilla, CERN. Geneva, Switzerland.

E-mail: `dcampora@cern.ch`

Abstract. The 2020 upgrade of the LHCb detector will vastly increase the rate of collisions the Online system needs to process in software, in order to filter events in real time. 30 million collisions per second will pass through a selection chain, where each step is executed conditional to its prior acceptance.

The Kalman filter is a fit applied to all reconstructed tracks which, due to its time characteristics and early execution in the selection chain, consumes 40% of the whole reconstruction time in the current trigger software. This makes the Kalman filter a time-critical component as the LHCb trigger evolves into a full software trigger in the Upgrade.

I present a new Kalman filter algorithm for LHCb that can efficiently make use of any kind of SIMD processor, and its design is explained in depth. Performance benchmarks are compared between a variety of hardware architectures, including x86_64 and Power8, and the Intel Xeon Phi accelerator, and the suitability of said architectures to efficiently perform the LHCb Reconstruction process is determined.

1. Introduction

The LHCb detector will be upgraded in 2020 [1] to acquire data at an instantaneous luminosity of $2 \times 10^{33}\text{cm}^{-2}\text{s}^{-1}$ and to collect a dataset of at least 50 fb^{-1}. At the same time the first stage of filtering in the Data Acquisition process, currently known as hardware level trigger, will be discontinued in favor of a full software trigger. Consequently the throughput that the software level trigger will need to sustain in order to maintain a steady triggering rate will dramatically increase, due to both the increase in rate of events processed in software, and the influx of larger events.

To be able to cope with the increased data rate, several hardware architectures are currently being explored. While the current LHCb High Level Trigger farm is composed solely of Intel Xeon processors, in the last few years many High Performance Computing sites are adopting other alternative hardware architectures, such as IBM Power X, FPGAs, or manycore architectures like GPUs or Intel Xeon Phi. This has provoked an interest in the High Energy Physics community, and more concretely within the LHCb collaboration, and the question is whether these architectures are suitable for performing the High Level Trigger in a sustainable way, taking into account the economical, power consumption and software maintainability aspects.

The Kalman filter [2] is a linear quadratic estimator, which is applied to estimate the trajectory of the particles as they travel through the LHCb detector. Table 1 shows a time characterization of the first stage of the LHCb High Level Trigger (HLT1). The Kalman filter algorithm is the single largest time contributor in the LHCb software chain, taking about 60 % of the HLT1 reconstruction time.

Following Amdahl's law [3], the achievable performance gain of an algorithm is bounded by its parallelizable portion. Due to the nature of the LHCb experiment, many particles

Algorithm	Time contribution (%)
Kalman filter	59.80
Hlt1UpgradeForwardHPT	22.44
PrVeloHlt	6.38
PrVeloUTHlt	3.25
HltUpgradePV3D	2.96
Hlt1UpgradeTwoTrackMVAUnit	2.82
PrPixelStoreClusters	1.36

Table 1: Time characterization of most time consuming algorithms in the first stage of LHCb High Level Trigger (HLT1). Only algorithms with a time contribution of more than 1% are shown.

travel through the detector simultaneously and independently, which makes the Kalman filter a massively parallelizable problem in this context.

In this paper, I present a hardware architecture independent design of a Kalman filter algorithm, *Cross Kalman*[1]. A focus is given to the LHCb use case, taking advantage of the fact that many independent particles can be processed simultaneously. I explore possible performance gains over the current LHCb Reconstruction software, and compare the speedup obtained over a variety of architectures.

2. The LHCb Kalman filter

The Kalman filter algorithm is typically implemented following a two-stage design, consisting of a *Predict* and an *Update* stage, performed sequentially. The *Predict* stage uses the built trajectory to extrapolate where the particle is expected to be on the next timestamp. *Update* uses the detector information (a *hit* in the detector) to correct the forming trajectory.

In the LHCb use case, a Kalman filter is applied to a particle trajectory throughout the detector, correcting it along the way using the detected hits of the particle. In hindsight, reconstructing one particle trajectory consists in applying repeatedly *Predict* and *Update* sequentially, hit after hit. A schematic of this process is shown in Figure 1.

Furthermore, in LHCb the trajectory reconstruction for a particle is performed in three steps: First, a *Forward* Kalman filter is performed, in the upstream direction of the detector (*right* in figure 1). Then, a *Backward* Kalman filter is performed, in the downstream direction of the detector. Finally, a smoothed trajectory, consisting of the average of the Forward and Backward trajectory, is calculated by the *Smoother* component.

Due to the sequential nature of the Kalman filter process, there is no room for parallelization of the reconstruction of a single particle trajectory. However, $\mathcal{O}(100)$ particles are reconstructed within each collision independently. This fact poses the motivation for the algorithm design.

3. Cross Kalman algorithm design

The design of the Cross Kalman algorithm has been carried out with a Single Instruction Multiple Data (SIMD) architecture in mind, focusing on three design aspects: the control flow, the underlying data structures and a fine-grained optimization of the most time-consuming parts.

3.1. Control flow

As mentioned above, particles are processed hit by hit in the Forward direction, followed by the Backward direction and the Smoother. Conceptually, the processing of hits of different

[1] https://gitlab.cern.ch/dcampora/cross_kalman.

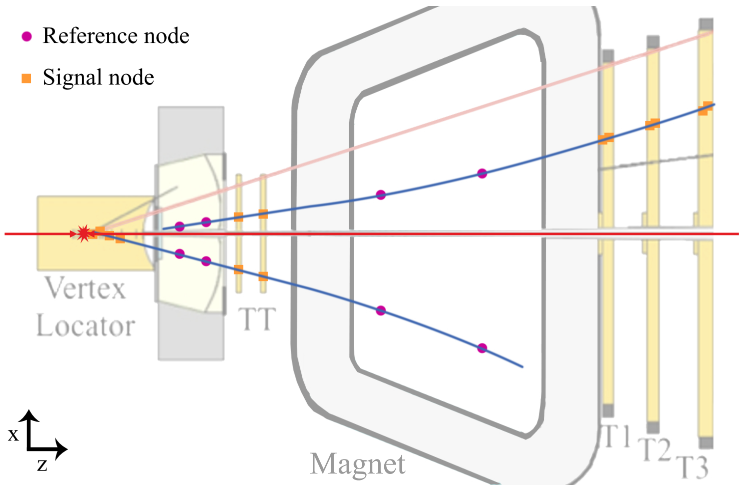

Figure 1: Schematic of LHCb Kalman filter process. Two particles collide inside the VELO subdetector (in red). Two particles are reconstructed (in blue), leaving *signal nodes* in the VELO, TT and tracking stations. *Reference nodes* are placed at certain points in the trajectory to trigger state calculations for posterior usage in the reconstruction.

particles is independent, and so the Kalman filter can be parallelized horizontally, which is envisioned in three stages as well: Forward, Backward and Smoother. In each of the stages, the hits of different particles can be processed in parallel. Given that different particles may have a different number of hits, a *static scheduler* is needed to guarantee a maximum resource utilization in each computation step.

The scheduler accepts a *width* parameter to decide how many hits are to be processed in one step of the computation. The width of the scheduler should fit with the SIMD width, and the choice of the most performant width depends on the architecture. Furthermore, the design of the algorithm is not tied to a particular precision, but rather it accepts a configurable *precision length* to choose the according floating point standard at compilation time. These two factors will determine how many elements are executed in parallel at each computation step: For a CPU supporting Advanced Vector Extensions (AVX) and choosing double floating point precision, up to four elements will be processable at a time, whereas for a CPU supporting AVX-512 and single precision, up to sixteen elements would be processable at a time.

Since particle trajectories contain varying number of hits, some steps will not fully fill the requested scheduler width. In order to minimize the number of steps, the particles are preordered by decreasing number of hits.

When considering the type of scheduler to implement, static and dynamic schedulers were considered. A static scheduler determines its order of execution prior to start executing iterations. On the other hand, a dynamic scheduler chooses the elements to execute on-the-fly. The results of the Forward and the Backward stages will be used by the Smoother. If one were to use a dynamic scheduler, the output of the first two stages would not be guaranteed to be aligned, and in order for the Smoother to produce the correct output, it would have to construct aligned data structures, requiring additional memory copy operations. The static scheduler circumvents this by design: the scheduler is calculated once, iterated to process the Forward, and iterated in reverse for the Backward. That way, the Smoother will not require any additional memory operations.

Figure 2 depicts some iterations computed with the described scheduler, with a width set to four. The last column depicts the particle-hit being processed at the moment. This scheduler can be iterated forwards or backwards, maintaining the sequentiality enforced by the Kalman filter process. Due to its flexible design, execution can be optimized on multi and manycore SIMD architectures, with varying vector widths.

```
it    in   out  act  vector (#particle-#hit)
#540: 0000 0001 1111 { 112-9 80-11 81-11 113-10 }
#541: 0001 1110 1111 { 112-10 80-12 81-12 79-3 }
#542: 1110 0000 1111 { 107-2 109-1 108-2 79-4 }
#543: 0000 0000 1111 { 107-3 109-2 108-3 79-5 }
#544: 0000 0000 1111 { 107-4 109-3 108-4 79-6 }
#545: 0000 0000 1111 { 107-5 109-4 108-5 79-7 }
#546: 0000 0000 1111 { 107-6 109-5 108-6 79-8 }
#547: 0000 0000 1111 { 107-7 109-6 108-7 79-9 }
#548: 0000 0000 1111 { 107-8 109-7 108-8 79-10 }
#549: 0000 0000 1111 { 107-9 109-8 108-9 79-11 }
```

Figure 2: Static scheduler iterations. The first column shows the iteration number. The second and third show the input and output mask, used to notify a change of particle. The fourth column is the action mask. The last column shows the hits being processed in parallel. This scheduler was run with a width of four.

3.2. Data structures

In order to improve the arithmetic intensity[2] of the algorithm, the data is structured following an AOSOA (Array of Structures of Arrays) design. For instance, for each step of the Forward and Backward Kalman filter, a state, covariance and χ^2 fit are calculated. For a single particle position, a state is a five-element vector $\begin{pmatrix} x & y & tx & ty & \frac{q}{p} \end{pmatrix}$, a covariance σ is a 15-element matrix (5×5 symmetric matrix), and the χ^2 is a single element [4]. Figure 3 shows the AOSOA data structure for a scheduler with width four.

$$\begin{vmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ tx_0 & tx_1 & tx_2 & tx_3 \\ ty_0 & ty_1 & ty_2 & ty_3 \\ \frac{q}{p}_0 & \frac{q}{p}_1 & \frac{q}{p}_2 & \frac{q}{p}_3 \\ \sigma_{0,0} & \sigma_{1,0} & \sigma_{2,0} & \sigma_{3,0} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma_{0,14} & \sigma_{1,14} & \sigma_{2,14} & \sigma_{3,14} \\ {\chi^2}_0 & {\chi^2}_1 & {\chi^2}_2 & {\chi^2}_3 \end{vmatrix}$$

Figure 3: AOSOA data structure for scheduler with width four.

These kind of data structures are highly efficient for SIMD processors, as they exploit locality, minimizing cache misses, and allow for horizontal parallelization in a natural way. In the presented algorithm, these data structures are generated with the execution of the scheduler, and the data are directly populated in them, avoiding extra memory copies. In addition, the data structure is allocated aligned to the required memory alignment by the specific architecture under execution.

3.3. Fine-grained optimization

In order to target specific Instruction Set Architectures (ISAs) of various architectures, the arithmetic bulk of the Kalman filter is written using a layer of abstraction, provided by a

[2] The arithmetic intensity of an application is a metric of how many arithmetic operations are performed per required byte transfer. We are interested in the FLOP arithmetic intensity, measured in $\frac{FLOP}{Byte}$.

vectorization library. Bindings to VCL [5] and UMESIMD [6] are provided, leaving it to the library provider to add compatibility for new architectures.

Furthermore, there is no conceptual imposition in the SIMD abstraction used. For instance, it would be possible to develop an OpenCL implementation taking advantage of the aforementioned control flow and data abstractions, targetting an arbitrary width and a manycore architecture or an FPGA.

Finally, a scalar implementation is also provided, in case the architecture under study does not support any SIMD extensions.

4. Results

Figure 4 shows the performance gain across architectures, using the here presented Kalman filter implementation. The figure of merit to compare the results is the combined throughput of the Forward and Backward Kalman filter and the Smoother:

$$\frac{\#\text{Forward} + \#\text{Backward} + \#\text{Smoother}}{\text{time}} \tag{1}$$

All the tests were run under the following settings:

The program was compiled with gcc 6.2.0, with options `-O2 -march=native`.
Turbo Boost was on, where applicable.
KNL was using flat memory mode, and pinned against the MCDRAM.
One process was spawned per Non-Uniform Memory Access (NUMA) domain, with as many TBB threads as cores in domain and pinned to its memory.
Ran 500 000 events, each event is a Threading Building Blocks (TBB) task.
Used Monte Carlo events from the LHCb Upgrade.
Double precision results are validated. Single precision results show a deviation in 1% of the results.

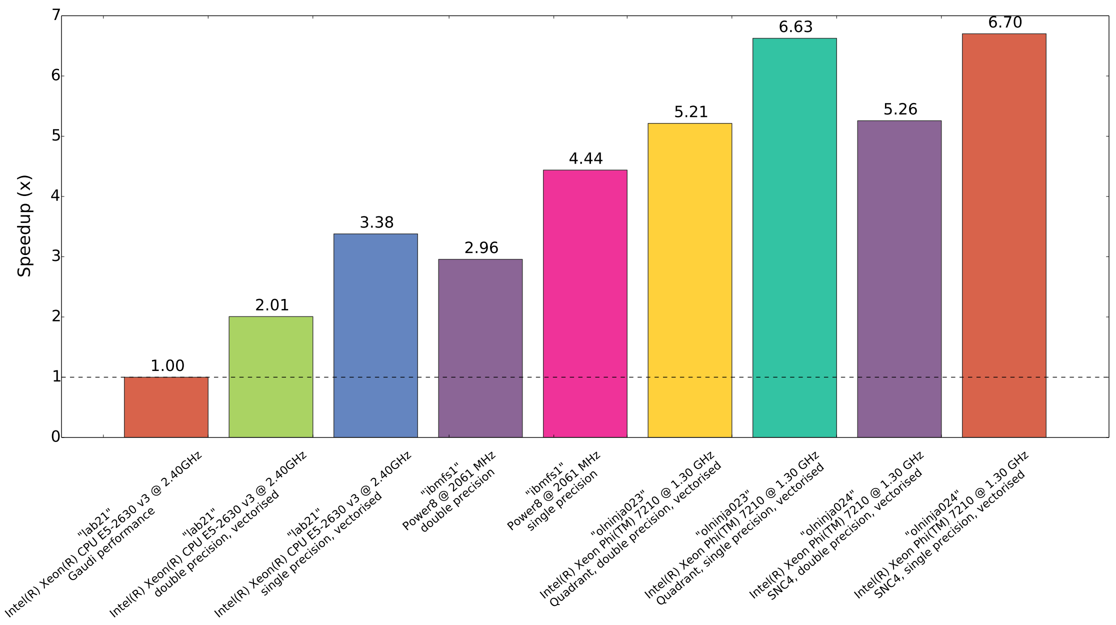

Figure 4: Speedup across several architectures and precisions.

The original implementation is shown as the leftmost bar. It is used as a baseline against which to compare performance. Using the same precision, the Xeon machine shows a 2.01×

speedup. It is worth noting that moving to the Intel Knights Landing (Xeon Phi) is cost-effective, when taking into consideration the difference in cost and performance versus the baseline under study.

Transitioning to single precision has a variable impact on performance, depending on the architecture under analysis, as shown in Table 2. It is particularly effective on current HLT hardware, where a performance boost of 1.68× is observed.

Hardware architecture	Effect of moving to single precision
Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz	1.68×
PowerNV 8335-GCA (Power8)	1.5×
Intel(R) Xeon Phi(TM) 7210 @ 1.30 GHz	1.27×

Table 2: Effect of moving to single precision across architectures.

The scalability of the application under various architectures is shown in the following. The Xeon architecture processor under analysis does not scale with the number of tasks in flight, as illustrated in Figure 5. This is partially due to Turbo Boost increasing the frequency of the processor for fewer core counts, which the vendor announces as 2.40 GHz base frequency versus 3.20 GHz max turbo frequency. Additionally, the memory footprint of Cross Kalman with higher core counts is quite large. There is no gain in using additional Hyper Threads per core on this processor.

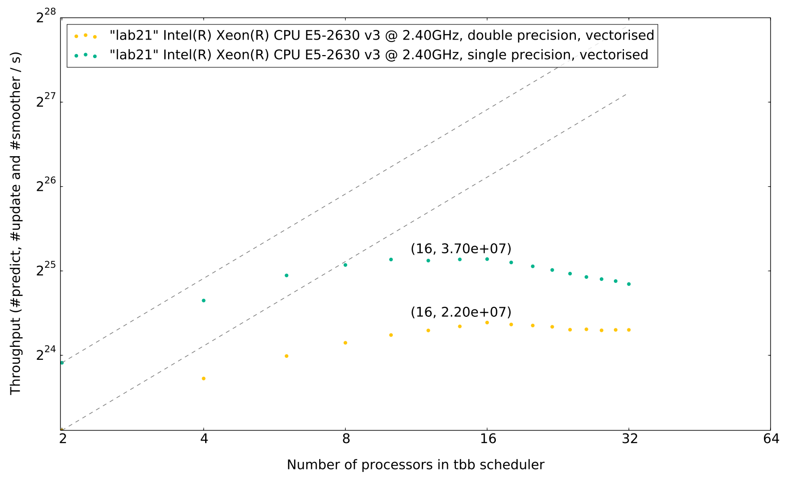

Figure 5: Scalability of Kalman filter for Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz.

The Knights Landing architecture scales better, as shown on the left in Figure 6. This machine has 64 physical cores, and the scalability until that point is almost linear. Using two Hyper Threads per core a marginal gain is observed, but the performance flattens out when using three or four Hyper Threads per core. It is interesting to note the throughput of the MCDRAM is not a bottleneck, regardless of the number of tasks in flight. The Power8 architecture under analysis also shows a good scaling, as shown on the right in Figure 6. Enabling Simultaneous Multi Threading (SMT) did not improve performance.

Figure 7 depicts a Roofline model [7] for an Intel Haswell processor. A Roofline model relates the arithmetic intensity of an application and its performance, with the capability of the specific hardware platform under consideration. The figure shows the arithmetic intensity of two particular processes of the Kalman filter. In order to increase the arithmetic intensity of the application, the *Predict* and *Update* steps (cf. Section 2) were combined into a single step called *fit*, in green in the plot. The *smooth* corresponds with the *Smoother*

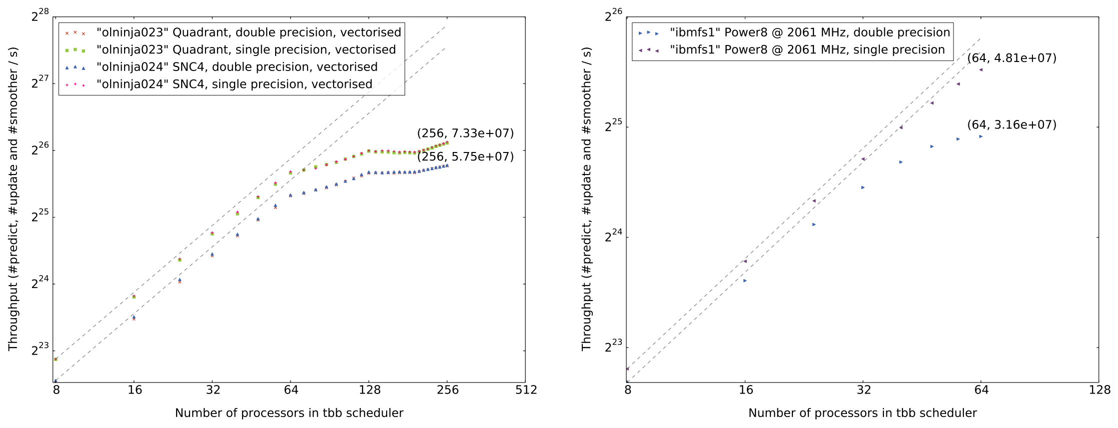

Figure 6: Left: Scalability of Kalman filter for Intel(R) Xeon Phi(TM) 7210 @ 1.30 GHz. Right: Scalability of Kalman filter for PowerNV 8335-GCA (Power8).

component, in blue in the plot. As the figure shows, the program is close to saturating the DDR RAM Bandwidth the CPU can support. Given the arithmetic intensity of this implementation of the Kalman filter, there is not much more performance one can extract out of this processor.

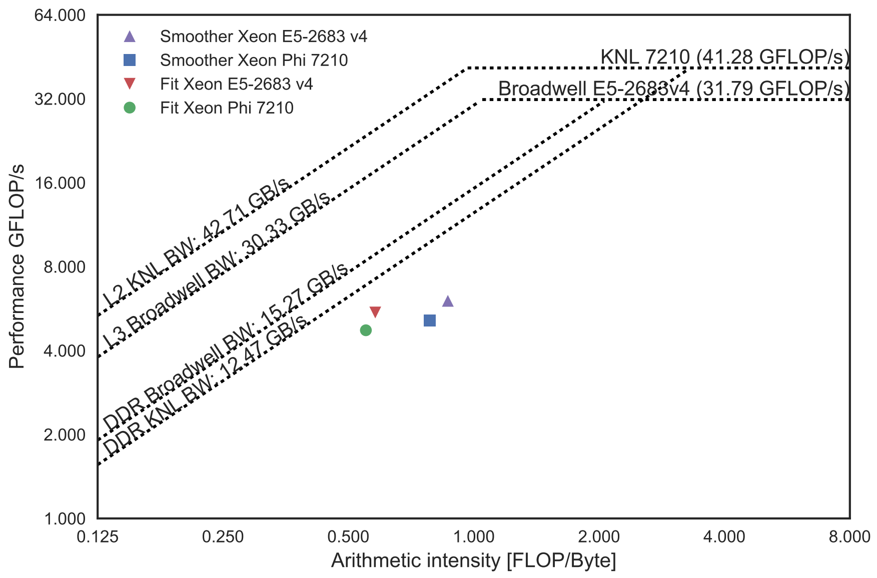

Figure 7: Roofline model for Intel(R) Xeon(R) E5-2683 v3 @ 2.00 GHz.

5. Conclusions and outlook

I have briefly discussed the requirements of software throughput for the upcoming upgrade of the LHCb trigger. In order to cope with its stringent real-time requirements, a new Kalman filter algorithm has been designed, and several hardware architectures and their amenability to processing a substantial portion of the LHCb reconstruction have been explored.

These studies suggest the Intel Knights Landing architecture will be a strong contender in the choice of a hardware architecture for the Upgrade of LHCb, due to its familiar programmability and the strong scaling it shows for a higher number of cores. When porting reconstruction software to these architectures however, one should take into consideration that there will be a software framework behind the scenes that will impact the performance and possibly change memory requirements. To this end, the Cross Kalman algorithm is being integrated with the framework, and a follow up study should relate the results obtained here with what the framework delivers in the final product.

It would be interesting to explore other architectures, such as ARM64, GPUs or FPGAs. As the literature in the field suggests, single precision may have even a deeper impact on these architectures. Even though precision can be configured at compile time in Cross Kalman, integration with the framework will be costly in terms of development time, and validation of these results will require a dedicated study. Nevertheless, the results suggest that there would be a high impact in terms of performance if a transition to single precision is successfully made.

Acknowledgements

The author would like to thank the HTCC at CERN for fruitful discussions through the process of designing and writing the presented software, and early access to Intel hardware. Special thanks go to O Awile for the guidance and the creation of the Roofline model plot. Furthermore, thanks to F Lemaitre for his contribution of the vectorized transposition code, and to O Bouizi and S Harald for the low-level code discussions and for providing early results and insight on the KNL architecture. In addition, thanks to C Potterat for the Physics validation, and to W Hulsbergen and R Aaij for the mathematical discussions and data structure design. Finally, thanks to my supervisors N Neufeld and A Riscos Núñez for their guidance and support.

References

[1] The LHCb Collaboration (2014). LHCb Trigger and Online Upgrade Technical Design Report. CERN-LHCC-2014-016 ; LHCB-TDR-016.

[2] Kalman R E (1960). A New Approach to Linear Filtering and Prediction Problems. Transactions of the ASME–Journal of Basic Engineering, 82 (Series D): 35-45.

[3] Amdahl G M (1967). Validity of the single processor approach to achieving large scale computing capabilities. AFIPS spring joint computer conference.

[4] Hulsbergen W (2008). The Global covariance matrix of tracks fitted with a Kalman filter and an application in detector alignment. DOI: 10.1016/j.nima.2008.11.094.

[5] Fog A (2012). VCL C++ vector class library. Gnu public license. http://www.agner.org/optimize.

[6] Karpinski P (2015). UME::SIMD A library for explicit simd vectorization. https://github.com/edanor/umesimd.

[7] Williams S W, Waterman A and Patterson D A (2008). Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. Technical Report No. UCB/EECS-2008-134.