Cross-architecture Kalman filter benchmarks on modern hardware platforms

Cámpora Pérez D H[1,2], Awile O[1], Bouizi O[3] and Neufeld N[1]

[1] CERN, CH-1211 Geneva 23, Geneva, Switzerland
[2] Universidad de Sevilla, C/San Fernando, 4, C.P. 41004, Sevilla, Spain
[3] Intel France, 2, rue de Paris, 92196 Meudon Cedex, France

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 process of the event reconstruction that, 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 it a time-critical component as the LHCb trigger evolves into a full software trigger in the upgrade.

The algorithm Cross Kalman allows performance tests across a variety of architectures, including multi and many-core platforms, and has been successfully integrated and validated in the LHCb codebase. Since its inception, new hardware architectures have become available exposing features that require fine-grained tuning in order to fully utilize their resources.

In this paper we present performance benchmarks and explore the Intel® Skylake and Intel® Knights Landing architectures in depth. We determine the performance gain over previous architectures and show that the efficiency of our implementation is close to the maximum attainable given the mathematical formulation of our problem.

1. Introduction

The projected upgrade of the LHCb detector at CERN in 2020 [1] will increase the estimated collision rate to 30 MHz, requiring data processing at 40 Tbit/s. As part of the upgrade, the first stage of filtering known as hardware level trigger will be discontinued in favor of a full software trigger [2]. Consequently, the processing rate required in the software trigger will increase by a factor 40, due to both the increase in rate of events processed in software, and the influx of larger events.

The Kalman filter is a linear quadratic estimator invented by Rudolf E. Kálmán in 1960 [3], that allows accurate estimation of the state of a system. In its discrete formulation it consists of a *predict* phase where the state is propagated according to its physical model, and an *update* phase where the measurements observed are integrated in the state prediction. Since its inception, it has successfully been applied to a range of applications including space tracking [4] and meteorology [5].

In high-energy physics experiments, the Kalman filter is a fundamental part of the reconstruction of particle trajectories coming from particle collisions. In LHCb, the Kalman filter consumes 40% of the reconstruction time, rendering it a critical component in the reconstruction. In order to cope with the increase in data rate, the software in the LHCb reconstruction is being updated to current hardware standards.

Cross Kalman is a cross-architecture Kalman filter optimized for a variety of SIMD processors [6] [7]. It is optimized for low-rank Kalman filters on multi and many-core processors, and we have successfully integrated it in the LHCb reconstruction software. It also serves as a self-contained package to evaluate different architectures in the context of the LHCb upgrade.

In this paper, we focus on the evaluation of modern Intel® platforms, including Xeon Phi™ and Xeon® Scalable Processors. Our software takes into account features like the support of the AVX-512F ISA extension and high bandwidth MCDRAM. We evaluate the scalability and price performance of Cross Kalman in all platforms. In addition, we evaluate our Kalman filter integrated in the multithreaded LHCb framework *Gaudi* [8]. We show the ability of Cross Kalman to scale to all considered configurations, and draw conclusions over the platforms under analysis.

2. A vectorized Kalman filter

In the LHCb software level trigger, particle trajectory reconstruction (*track reconstruction*) involves a complex selection chain where the trajectory of each particle is determined in several steps, due to the various subdetectors that compose the LHCb tracker [9]. Several of these stages involve simplified fits, in the form of square root fits or simplified Kalman filters. In contrast, Cross Kalman performs a fit in three steps: For each track, a fit is performed in the *forward* direction, followed by the *backward* direction, and finally a *smoother* is applied to average both states [10].

Additionally, a track is composed of two kind of nodes. *Signal* nodes are produced by detector signals, whereas virtual *reference* nodes are placed at certain positions in the track in order to determine track parameters up-front for later stages of the reconstruction. While signal nodes trigger a Kalman filter step composed of *predict* and *update* phases, reference nodes only trigger the *predict* phase in either direction. Figure 1 depicts two particles traversing the LHCb detector. The particle at the top would trigger three *predicts* before the first *update*, in the forward direction.

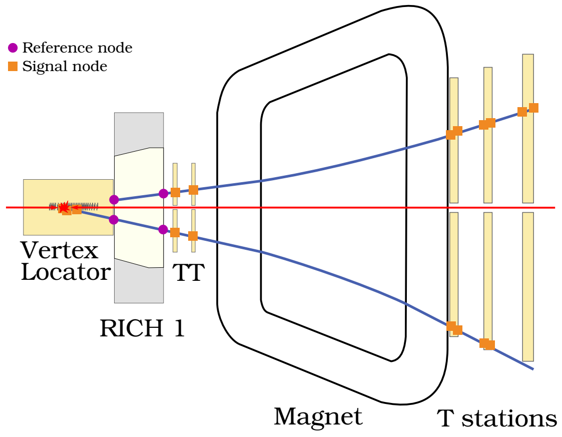

Figure 1: Schematic of two particle trajectories traversing LHCb detectors (in blue). The collision at their origin (in red) takes place in the Vertex Locator subdetector. The tracks are composed of signal nodes, produced in the Vertex Locator, TT and the three tracking stations T1, T2 and T3. A magnet bends the tracks between the TT and the tracking stations. Reference nodes are placed at predefined locations in order to determine their track parameters.

In order to adapt the inherent complexity of the problem to any SIMD architecture, we have developed Cross Kalman following three design aspects: *flow control*, *data structures* and *backend implementation*.

SIMD processors require non-divergent workflows to operate at peak performance. In Intel® processors, vector units perform the same operation over an array of data. This requirement has become more evident as the vector units have become wider. Therefore, we employ a static scheduler to assign computations to SIMD lanes. Since the execution of nodes in different tracks is independent, we execute them in a horizontally parallel scheme. We allow for a configurable lane width, which makes our scheduler scale to the considered SIMD platforms.

The LHCb Kalman filter needs to be processed in the backward direction as well. Our scheduler guarantees this requirement trivially by traversing the scheduled object in the reverse direction.

The *data structure* that we use in our application is Array of Structures of Array (AOSOA). This data structure has been shown to work well with SIMD architectures [11], as it maximizes locality. In Cross Kalman, the data is assumed to be already in this format in the order we require to process it. This assumption is safe, as the schedule can be generated prior to populating these data structures (cf. section 2.1).

The data is allocated in contiguous chunks of aligned memory. In order to avoid overheads from allocating big chunks of memory, *data stores* are created on demand and can store a configurable number of elements. In order to be able to traverse the data stores, iterators and reverse iterators that are aware of the store size are provided.

Finally, several *backend implementations* are provided to perform the fit computations. A scalar implementation, alongside Vectorclass [12] and UMESIMD [13] vectorized implementations are provided. We also provide OpenCL and CUDA implementations in our proto-application *Cross Kalman Mathtest*, showing that we can run on many types of SIMD architectures.

2.1. Integration

Our software has been integrated and validated in the Gaudi framework. Following the pre- and post-condition requirements of the current Kalman filter application *TrackMasterFitter* (TMF), we have developed *TrackVectorFitter* (TVF) as a SIMD aware drop-in replacement for TMF. TVF implements the design of Cross Kalman, with the necessary code modifications and data adaptations.

In order to minimize the impact of data transposition, we generate the static schedule prior to any data generation. This way, memory references are populated as the Cross Kalman filter would expect them. Then, the data is populated in-place. This method improves the data generation performance, and allows for future optimizations by vectorizing the methods for data generation. Additionally, the output data is required to adhere to Gaudi's data layout for the processes that follow in the selection chain after the Kalman filter. We transpose the data only once, at the end of TVF's execution. These optimizations reduce the overhead of changing the data structure from AOS to AOSOA. Nevertheless, we have observed a performance penalty which currently is 3.4% of the total execution time of TVF.

Finally, a version of Gaudi with multithreaded support is currently under development. We have made our application reentrant in order to support this version of the framework.

3. Results

The results in this section were obtained with Cross Kalman v1.7, compiled with `gcc 6.2.0` and flags `-O2 -march=native`. The events used, were generated with the official LHCb Monte Carlo event generator, and the results were validated against the existing Kalman filter implementation. NUMA pinning and TBB tasks were used to efficiently load the nodes. Table 1 describes the nodes used for these tests. Xeon® E5-2630 v3 (Haswell) is the current production platform at LHCb. Xeon® E5-2683 v4 and Xeon® Platinum 8170 represent medium-range and high-range processors from the Broadwell and Skylake family respectively. Xeon Phi™ 7210 (Knights Landing) is a many-core alternative to the more traditional Xeon® based solutions.

	Xeon® E5-2630 v3	Xeon® E5-2683 v4	Xeon Phi™ 7210	Xeon® Platinum 8170
# cores	8	16	64	26
base frequency	2.40 GHz	2.10 GHz	1.30 GHz	2.10 GHz
Cache	20 MB L3	40 MB L3	32 MB L2	35.75 MB L3
TDP	85 W	120 W	215 W	165 W
DRAM configuration	64 GB	64 GB	96 GB 16 GB MCDRAM	192 GB
STREAM bandwidth	41 GB/s	76 GB/s	77 GB/s (DRAM)	101 GB/s
Recommended price (ark.intel.com, August 2017)	\$667.00 - \$671.00	\$1846.00	\$1881.00	\$7405.00 - \$7411.00

Table 1: Machine configurations under study.

In Figure 2a we show a performance comparison across the architectures under consideration. The leftmost bar represents the scalar performance of Cross Kalman on the production platform. When running the vectorized implementation on the same processor, we observe a 1.36× speedup. We note also a 1.27× speedup of the Skylake processor over Knights Landing. A price performance comparison is shown in Figure 2b. The Xeon Phi™ Knights Landing processor is the most cost-effective solution for Cross Kalman, being 40% cheaper than the production platform. Medium and high-range CPUs are not cost-effective for our workload.

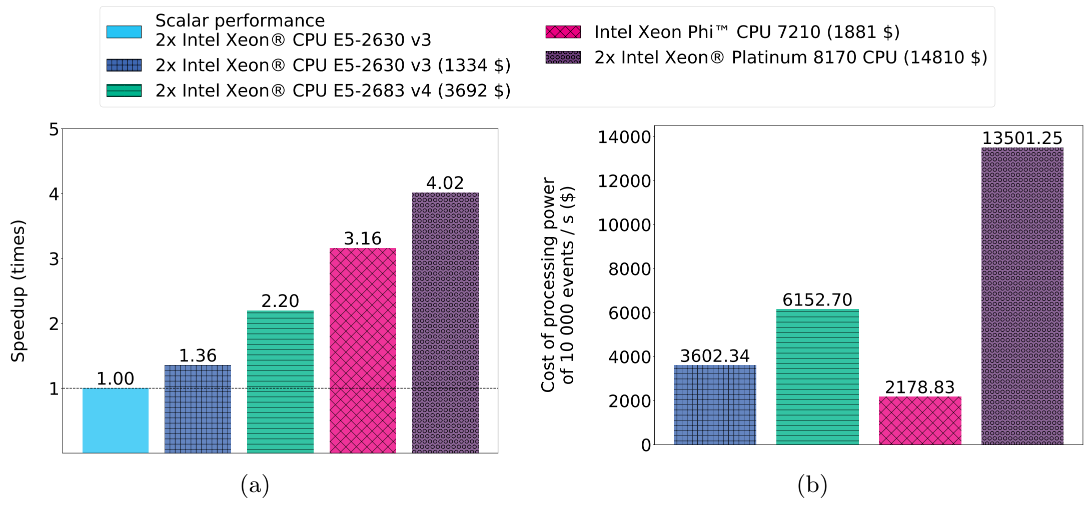

Figure 2: (a) Performance of Cross Kalman relative to scalar implementation across several architectures (higher is better). The best configuration for each architecture in terms of active threads is used, shown in Figure 3a. The leftmost bar represents Cross Kalman's scalar implementation, running on a dual-socket Xeon® E5-2630 v3 processor. (b) Price required to obtain a throughput of 10 000 events per second (lower is better). The Xeon Phi™ processor shows a better *price for throughput* ratio than its competitors.

Figure 3a shows the scalability of Cross Kalman in the architectures under study. Our workflow saturates Xeon® processors when configured with all physical processors, and we do not gain by adding extra Hyperthreads (HTs). The notable exception is the Knights Landing platform, where we gain an additional 31% with 2 HTs and 13% with 4 HTs. We attribute this behavior on Knights Landing to its higher bandwidth MCDRAM, which we believe is in the case of our application only saturated when using all available Hyperthreads.

Figure 3b shows the scalability of the Gaudi algorithm *TrackBestTrackCreator*, which contains TVF. The Gaudi framework is run in multithreaded mode with as many active processors as indicated in the X axis. The behavior of this workload is very different than Cross Kalman. The application benefits from the HTs in both Xeon® architectures, however not in Xeon Phi™. Both in Xeon Phi™ and Skylake, the peak performance is observed when configured with 80 cores. We conclude Gaudi is not yet optimized for many-core execution.

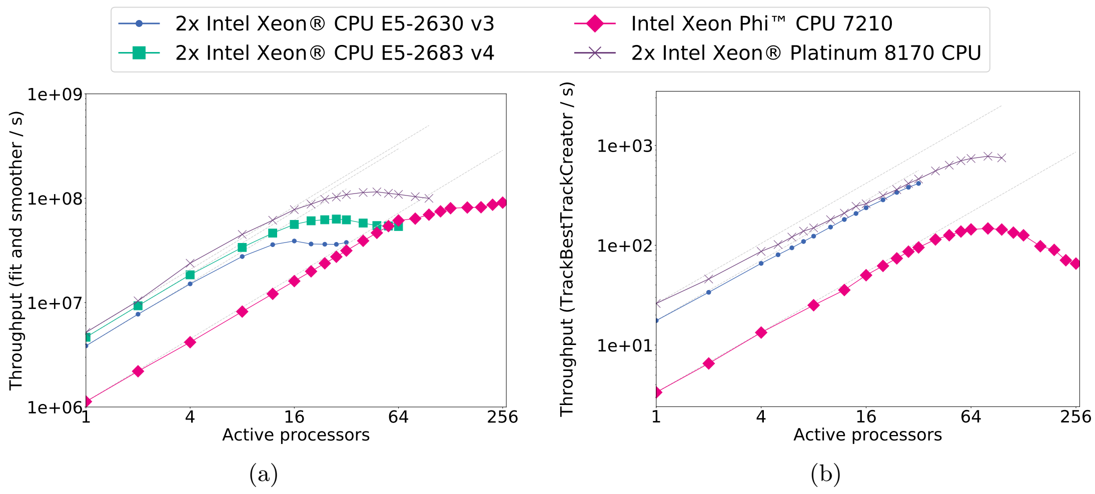

Figure 3: (a) Throughput of Cross Kalman across architectures. All processors show a degradation of performance when using HTs with the exception of Xeon Phi™. (b) Throughput of algorithm executing TVF in the Gaudi framework. Xeon® processors scale better than in Cross Kalman, in contrast to Xeon Phi™.

Finally we present a Roofline model [14] of our fit and smoother programs in Figure 4. The scalar implementation is provided by an emulated vector mode in UMESIMD, and we observe the overhead introduced by this emulation causes the scalar fit and smoother to underperform. We saturate the Skylake processor with our fit implementation when configured with SIMD lane width 8. Even though the smoother formulation shows a higher arithmetic intensity, we observe there may still be room for optimization.

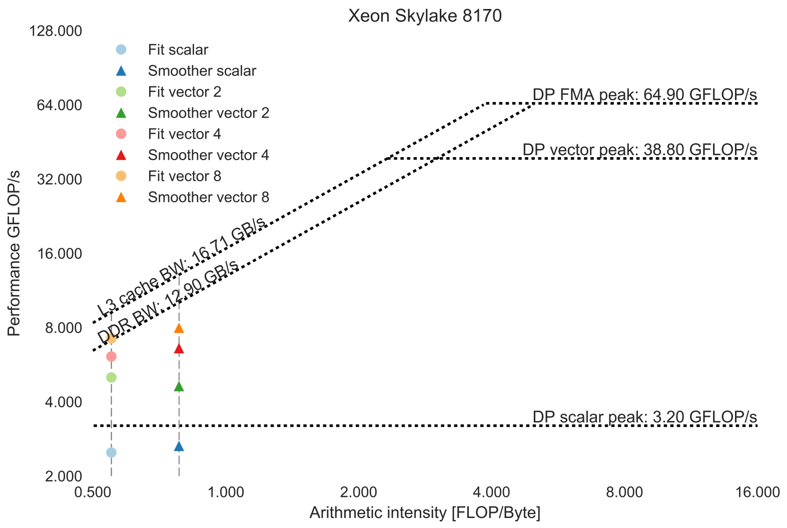

Figure 4: Roofline model of fit and smoother processes under Xeon® Platinum 8170 (Skylake). Different double precision vector width configurations are shown.

4. Conclusions

We have presented results of our evaluation of modern hardware architectures using Cross Kalman. We compared three dual-socket Xeon® architectures including a production Haswell LHCb server, and a many-core Xeon Phi^{TM} architecture. The Skylake processor outperforms the other considered alternatives in raw computational performance. At the same time, the Xeon Phi^{TM} processor under study is the most cost-effective solution for the Cross Kalman workload, offering the same performance as our production dual-socket Haswell system for 60% of its cost.

We have successfully ported our algorithm to the Gaudi multithreaded framework. We observe a different behavior in the scalability of Cross Kalman and Gaudi versions. We believe this is due to the inner workings of the framework, currently in development in its multithreaded form. In particular, many-core architectures do not seem to scale. We will further investigate the causes in the framework and try to mitigate this effect.

Cross Kalman saturates the Skylake platform for the fit process, while the arithmetic intensity of the smoother process seems to leave room for improvement, which we plan to study further. We will continue to track the performance of modern hardware architectures and adapt our software to it, and observe the evolution of the different platforms.

Acknowledgements

The authors would like to thank the High-Throughput Computing Collaboration at CERN openlab for fruitful discussions through the process of designing and writing the presented software. Thanks to C. Potterat for his contributions on the validation of the software. Thanks to F. Lemaitre for his contribution of the vectorized transposition code. In addition, thanks to W. Hulsbergen and R. Aaij for the mathematical discussions and data structure design. Finally, thanks to A. Riscos Núñez for his guidance and support.

References

[1] The LHCb Collaboration 2012 Framework TDR for the LHCb Upgrade: Technical Design Report Tech. Rep. CERN-LHCC-2012-007. LHCb-TDR-12 URL https://cds.cern.ch/record/1443882

[2] The LHCb Collaboration 2014 LHCb Trigger and Online Upgrade Technical Design Report Tech. Rep. CERN-LHCC-2014-016. LHCB-TDR-016 URL https://cds.cern.ch/record/1701361

[3] Kalman R E 1960 *Journal of Basic Engineering* **82** 35–45 ISSN 0098-2202 URL http://dx.doi.org/10.1115/1.3662552

[4] Mcgee L A and Schmidt S F 1985 Discovery of the Kalman filter as a practical tool for aerospace and industry Tech. rep. URL https://ntrs.nasa.gov/search.jsp?R=19860003843

[5] Houtekamer P L and Mitchell H L 1998 *Monthly Weather Review* **126** 796–811 ISSN 0027-0644 URL http://journals.ametsoc.org/doi/abs/10.1175/1520-0493%281998%29126%3C0796%3ADAUAEK%3E2.0.CO%3B2

[6] Cámpora Pérez D H *Journal of Physics: Conference Series* to appear

[7] Cámpora Pérez D H, Awile O and Potterat C *Euro-Par 2017: Parallel Processing Workshops* to appear

[8] Barrand G *et al.* 2001 *Comput. Phys. Commun.* **140** 45–55

[9] Schiller M 2011 *Track reconstruction and prompt K^0_S production at the LHCb experiment* Dissertation University of Heidelberg

[10] Hulsbergen W 2009 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **600** 471 – 477 ISSN 0168-9002 URL http://www.sciencedirect.com/science/article/pii/S0168900208017567

[11] Gou C, Kuzmanov G and Gaydadjiev G N 2010 *Proceedings of the 24th ACM International Conference on Supercomputing* ICS '10 (New York, NY, USA: ACM) pp 179–188 ISBN 978-1-4503-0018-6 URL http://doi.acm.org/10.1145/1810085.1810111

[12] Fog A 2012 VCL C++ vector class library URL http://www.agner.org/optimize

[13] Karpiński P and McDonald J 2017 *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores* PMAM'17 (New York, NY, USA: ACM) pp 21–28 ISBN 978-1-4503-4883-6 URL http://doi.acm.org/10.1145/3026937.3026939

[14] Williams S, Waterman A and Patterson D 2009 *Communications of the ACM* **52** 65 ISSN 00010782 URL http://portal.acm.org/citation.cfm?doid=1498765.1498785