A High-Throughput Kalman Filter for Modern SIMD Architectures

Daniel Hugo Cámpora Pérez[1,2(✉)], Omar Awile[1], and Cédric Potterat[3]

[1] CERN, CH-1211 Geneva 23, Geneva, Switzerland
dcampora@cern.ch
[2] Universidad de Sevilla, C/San Fernando, 4, 41004 Sevilla, Spain
[3] Universidade Federal do Rio de Janeiro (UFRJ),
Caixa Postal 68528, Rio de Janeiro 21941-972, Brazil

Abstract. The Kalman filter is a critical component of the reconstruction process of subatomic particle collision in high-energy physics detectors. At the LHCb detector in the Large Hadron Collider this reconstruction must be performed at an average rate of 30 million times per second. As a consequence of the ever-increasing collision rate and upcoming detector upgrades, the data rate that needs to be processed in real time is expected to increase by a factor of 40 in the next five years. In order to keep pace, processing and filtering software must take advantage of latest developments in hardware technology.

In this paper we present a cross-architecture SIMD parallel algorithm and implementation of a low-rank Kalman filter. We integrate our implementation in production code and validate the numerical results in the context of physics reconstruction. We also compare its throughput across modern multi- and many-core architectures.

Using our Kalman filter implementation we are able to achieve a sustained throughput of 75 million particle hit reconstructions per second on an Intel Xeon Phi Knights Landing platform, a factor 6.81 over the current production implementation running on a two-socket Haswell system. Additionally we show that under the constraints of our Kalman filter formulation we efficiently use the available hardware resources.

Our implementation will allow us to better sustain the required throughput of the detector in the coming years and scale to future hardware architectures. Additionally our work enables the evaluation of other computing platforms for future hardware upgrades.

Keywords: Kalman filter · Data-intensive parallel algorithms
Numerical methods

1 Introduction

The LHCb detector at CERN will be upgraded in 2020 [1] to acquire data at an estimated rate of 30 MHz, requiring to process a data throughput of 40 Tbit/s. At the same time the first stage of filtering in the Data Acquisition process, also

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 378–389, 2018.
https://doi.org/10.1007/978-3-319-75178-8_31

known as hardware level trigger, will be discontinued in favor of a full software trigger [2]. 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 under consideration. While the current LHCb software 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 ARM 64, IBM Power X, FPGAs, or manycore architectures such as GPGPUs or Intel Xeon Phi. This has raised the question within the High Energy Physics community whether these architectures are also suitable for performing the software trigger in a sustainable way. To answer this question, performance, economical, power consumption and software maintainability aspects need to be taken into account.

In this work we will consider the Kalman filter component used in the LHCb software framework. The Kalman filter is a linear quadratic estimator, first introduced by Kálmán in 1960 [3], that has been extensively used to estimate trajectories in various systems [4,5]. In its discrete implementation [6], it consists in a *predict* stage where the state of the system is projected according to a given model, and an *update* stage where the state is adjusted taking into account a measurement. In particular we consider here a filter that is low-rank.

In LHCb the Kalman filter is applied to estimate particle trajectories (*tracks*) as they travel through the particle detector [7]. Tens of millions of collisions per second occur in the detector, each requiring tens of thousands of filter computations. The Kalman filter is therefore the single largest time contributor in the LHCb software chain, taking about 60% of the first stage software trigger reconstruction time.

According to Amdahl's law [8], the achievable performance gain of an algorithm is bounded by its parallelizable portion. Due to the nature of the LHCb experiment, many particles travel through the detector simultaneously and independently. Hence, the Kalman filter is considered a petascale embarrassingly parallel problem in this context. Here we present a hardware architecture independent Kalman filter algorithm and implementation, *Cross Kalman*[1] extending beyond previously presented results [9].

In contrast to the work by Cerati et al. [10], we do not use our Kalman filter for track finding, but instead, we filter fully built tracks. That allows us to take into account the number of tracks and nodes when envisioning a scheduling strategy. Resulting in an effective use of the SIMD capabilities of the processors under study.

We explore performance gains over the current LHCb particle reconstruction software [11], and compare the speedup obtained over a variety of architectures. Additionally, we validate our implementation and integrate it back in the LHCb reconstruction framework, observing a performance gain on existing hardware.

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

2 Cross Kalman

In LHCb track reconstruction a particle trajectory consists of *signal nodes* originating from detector signals. Additionally, virtual *reference nodes* are placed in large trajectory sections that have no detector signals. As opposed to signal nodes, reference nodes trigger a prediction with no update in the Kalman filter. Reference nodes improve trajectory prediction, at the cost of introducing additional complexity in the algorithm.

For a given particle trajectory, the Kalman filter is applied twice: First, a fit in the forward direction, positive in the Z axis, is followed by a fit in the backward direction, processing the nodes in reverse order. Afterwards, a smoothed state is calculated averaging both states. This introduces a dependency between the stages with little room for parallelization. However, a particle collision generates many independent particles that can be reconstructed at the same time, allowing us to envision a horizontally parallel scheme.

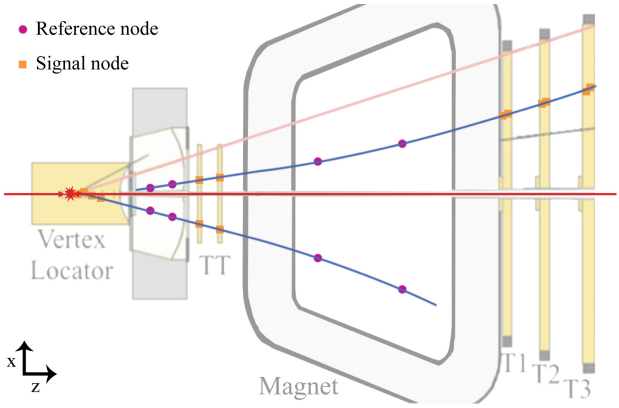

Fig. 1. Schematic of two particles (blue) traversing LHCb subdetectors. A particle collision is indicated by the two red arrows meeting in the center of the *Vertex Locator* subdetector. Particles produced from the collision traverse tracking subdetectors; here the Vertex Locator, TT and T1, T2 and T3 stations are depicted. A magnet bends the trajectory of produced particles according to their momentum and charge. (Color figure online)

For either direction, the first encountered signal node does not have any preceding signal data. *Reference parameters* according to their position are generated and fed onto those nodes, and the prediction is applied to these parameters. Figure 1 shows two particles traversing the LHCb detector with various nodes. When performing the forward fit, the top particle carries out three predictions from reference parameters before doing the first update. From that point on, all states are predicted from previous states, however only signal nodes trigger

an update. The particle at the bottom performs a single prediction from reference parameters, given the first node is a signal node. Finally, when doing the backward fit, a similar procedure follows: The bottom particle requires three predictions before the first update while the top particle requires one.

Furthermore, given a node, the resulting state is calculated as the average between its forward updated state and its backward predicted state. However, if the node has no preceding signal node in one of the directions, the smoother copies the updated state of the other direction.

Given this problem formulation, we describe the design of our algorithm in the following parts: the control flow, the data structures and an efficient implementation for performing the math computations.

2.1 Control Flow

Since the control path of processing a particle trajectory diverges depending on the nature of its nodes, we have divided each particle trajectory in three stages: `pre`, `main` and `post`. `pre` is the forward trajectory from the first node until a signal node is encountered, inclusive. Similarly, `post` is the backward trajectory from the last node until a signal node is encountered, inclusive. Finally, `main` includes the remaining nodes. The forward fit processing logic differs between `pre` and `main`, while for the backward fit processing logic differs between `post` and `main`.

In order to fully exploit the capabilities of SIMD architectures, we employ a static scheduler that assigns node calculations to SIMD lanes. Since the execution of nodes from different particles is independent, we execute them in a horizontally parallel scheme. In order to minimize branches and guarantee instruction locality, we generate three such schedulers, one for each stage.

The amount of nodes processable at a time depends directly on the SIMD width of the processor. Hence our scheduler accepts a configurable vector width. It is also able to detect at compile time the supported vector width of the platform. There are no restrictions on the width of the lane, allowing this design to also target manycore architectures, where wider vector units are available.

More formally, given m particle trajectories with n_i nodes each and k processors, we want to assign nodes to processors minimizing the number of compute iterations. This problem is a variant of the number partitioning problem NPP [12], which is known to be NP-complete. Our scheduling algorithm orders the trajectories in descending order of nodes, and assigns nodes to processors following a Decreasing-Time Algorithm (DTA).

The same scheduler can be used for the forward fit, the backward fit, and the smoother. The forward and backward dependencies between node calculations are naturally resolved by traversing the scheduler in the respective direction. All tracks are processed on each stage prior to processing the next one. The smoother `pre` and `post` stages are processed after completion of the backward fit.

In our implementation we place particular emphasis on avoiding as much as possible memory copy operations and exploiting memory locality. We reuse data structures throughout the scheduler iterations replacing only necessary data portions when required to do so. Additionally, the data structures must be aligned and refer relatively to the same nodes in order for the smoother to be able to produce an average state from the previously calculated forward and backward states. Using our scheduler this requirement is trivially met.

2.2 Data Structures

The algorithm's main data structure is composed of three parts. A hardware-specific data backend stores data contiguously and aligned to the required SIMD width, and provides chunks of requested data agnostic to their contents. In order to avoid a performance impact of memory allocations of big chunks of contiguous space, data backends are created on demand and can store a configurable number of elements. Iterators point to the data backends and are configured with a structure size. We provide forward and reverse iterators in order to traverse the data as required.

We use Arrays of Structures of Arrays (AOSOA) as data views over the data backends. This kind of data structures benefit from locality when accessing any of their elements, and have been shown to work well with SIMD processors [13]. Further locality is preserved by storing these structures next to each other contiguously.

2.3 Efficient Vector Implementation

We have implemented the core routines of the fit and smoother algorithms using manual vectorization with the help of vector intrinsics libraries. An iterative fine-grained optimization has been carried out, testing several formulations, unrolling loops, inlining functions, changing compiler options and reordering code. Also, we have implemented the arithmetic backend with several libraries in our synthetic benchmark *Cross Kalman Mathtest*[2], namely the vectorization libraries VCL [14], UMESIMD [15], and the language extensions OpenCL and CUDA. Our implementations can efficiently target any sort of SIMD paradigm. Furthermore, a scalar implementation is provided as fall back. It allows to process single tracks, and it can run on architectures not supporting vectorization.

3 Results

We ran the experiments in this section under the conditions shown in Table 1.

Figure 2 shows the cross-architecture speedup. The leftmost bar shows the performance of the scalar implementation of the fit, obtained from the timings reported by the framework. Our Cross Kalman implementation outperforms the scalar implementation on the same hardware platform by a factor

[2] https://gitlab.cern.ch/dcampora/cross_kalman_mathtest.

Table 1. Run conditions.

The program was compiled with gcc 6.2.0, with options `-O2 -march=native`
Turbo Boost was on, where applicable
KNL was using quadrant and 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 logical 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
Results are validated against expected result from original algorithm
Results were obtained using double precision
The figure of merit is the average throughput #fits/time

of 3.03x. ThunderX shows the poorest performance of the architectures under study. Even though a speedup of 1.75x over the scalar implementation on E5-2630 v3 is observed, this is only due to optimizations in the software. When both architectures run Cross Kalman, the E5-2630 v3 outperforms ThunderX by 1.73x. This is likely due to a comparatively lower peak DRAM bandwidth and peak floating point performance on ThunderX. The peak value is observed on a quad-socket high-end Intel Haswell system. This is, however, also the most expensive of the tested systems. It is interesting to note that Intel Xeon Phi outperforms our dual-socket Broadwell system, rendering it the most competitive from a price/throughput standpoint.

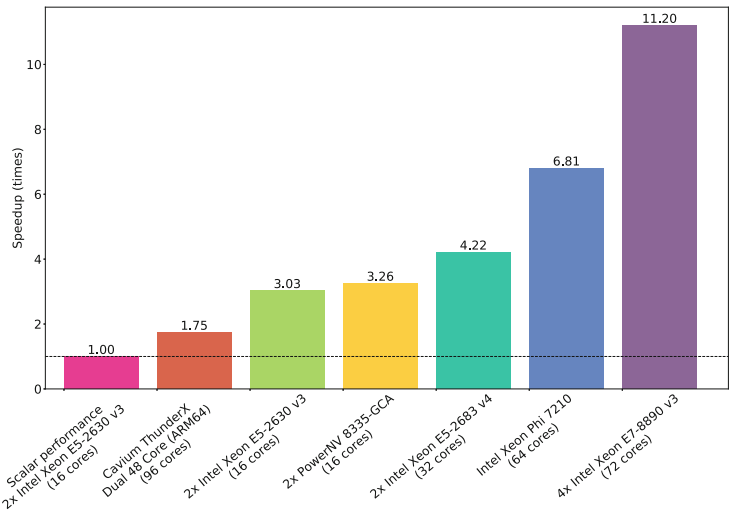

Fig. 2. Performance of Cross Kalman against the scalar implementation of the fit across several architectures.

A throughput scalability plot for all architectures is shown in Fig. 3a. The processor that shows less performance degradation up to using all of its cores is ThunderX. On the IBM Power8 architecture we are able to scale linearly while no Simultaneous MultiThreads (SMTs) are being used. Using 2 SMTs per processor, a performance improvement of 32% is observed. Moving from 2 to 4, a further 15% is gained, while moving from 4 to 8 no performance benefit is observed. On the Intel architectures we observe an almost linear scaling until we reach the limit of physical cores. The Intel Xeon Phi processor shows a 27% gain from using 2 HyperThreads, and a further 9% from using 4. We do not obtain any gain from HyperThreads on other Intel processors, which we attribute to the higher bandwidth of MCDRAM on Intel Xeon Phi.

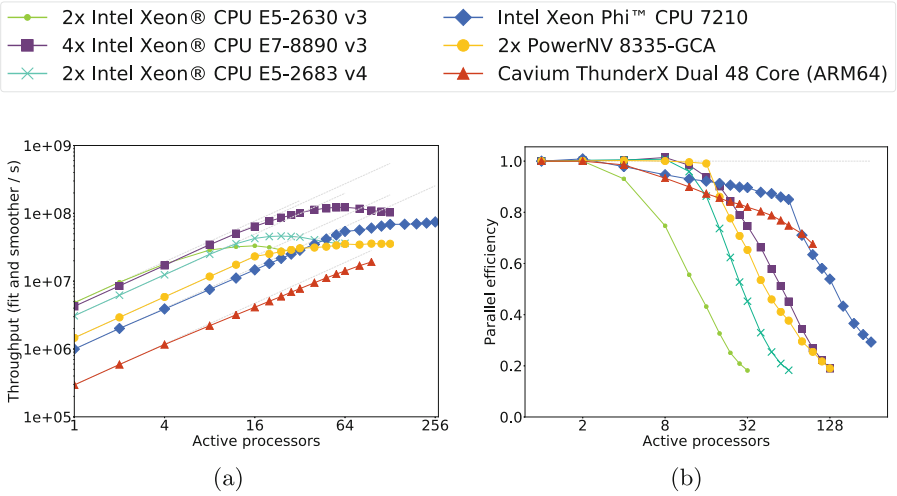

Fig. 3. (a) Throughput of Cross Kalman across various architectures. For each architecture, an increasing number of processors is enabled. Additional SMTs are only used on high core counts. (b) Parallel efficiency against active processors. The PowerNV processors shows no performance degradation using all its physical cores. In contrast, Xeon Phi shows a parallel efficiency of 85% (64 processors), ThunderX 68% (96 processors), E5-2630 v3 43% (16 processors), E7-8890 v3 40% (72 processors) and E5-2683 v4 45% (32 processors).

Figure 3b shows a parallel efficiency graph. All Xeon processors diverge from perfect scaling before the other processors under study. Xeon Phi and ThunderX show performance gains using all of their available processors, with a speedup of 74.98x and 64.88x respectively. For PowerNV, its optimal configuration is reached when configured with 96 processors (24.44x), where the performance flattens out. As expected on all tested hardware platforms, parallel efficiency is significantly degraded when using SMT. PowerNV shows a parallel efficiency of 1.0 until it starts using additional SMTs. We observe a similarly abrupt decrease in parallel efficiency in Xeon Phi when using additional HTs. The Xeon processors efficiency

drop even without HTs. With all their physical cores active, we see 40–45% efficiency, which could be due to the memory requirements of the application.

Figure 4 shows the throughput of the fit and smoother as the vector width is increased. In order to obtain the results of these figures, we used our synthetic benchmark, that allows us to execute the bulk of the computation of the application in a portable and generic way. The tests were compiled against the UMESIMD library. The scalar performance of the application is very poor in this setting, because scalar data is emulated in the UMESIMD library by a vector of width one. The smoother application scales slightly better than the fit, which we believe is due to its higher arithmetic intensity. We observe the same scaling for single and double precision, as is depicted by the two gray scaling lines being very close to each other. Single precision produces a deviation from the expected results in 1% of the experiments.

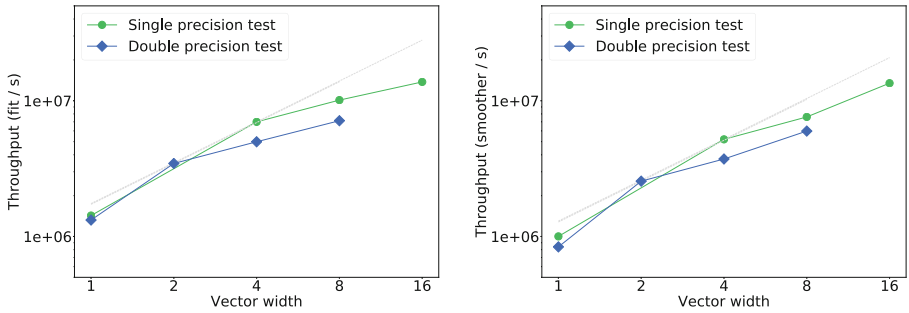

Fig. 4. Throughput of program as vector width increases, for single and double precision, under Intel Xeon Phi 7210. Left: fit throughput. Right: smoother throughput. We observe a scaled throughput for 128-bit vectors between single precision (width 4) and double precision (width 2). The smoother scales better than the fit for wider vector units, due to its higher arithmetic intensity.

Figure 5 shows a Roofline plot [16] for the fit and smoother processes. We ran for the Roofline benchmarks both the fit and smoother with 10 000 000 experiments. A high number of experiments is required in order to avoid data being cached from its generation to its execution, which would affect the arithmetic intensity of the application. This effect does not carry over to the full Cross Kalman code. The arithmetic intensity of the fit process is at about 0.5 FLOP/Byte, while the smoother is arithmetically more intensive at around 0.8 FLOP/Byte. Both fit and smoother performances are in the arithmetic-intensity regime limited by memory bandwidth and not peak floating point performance. However, our measurements show that we currently do not attain peak performance.

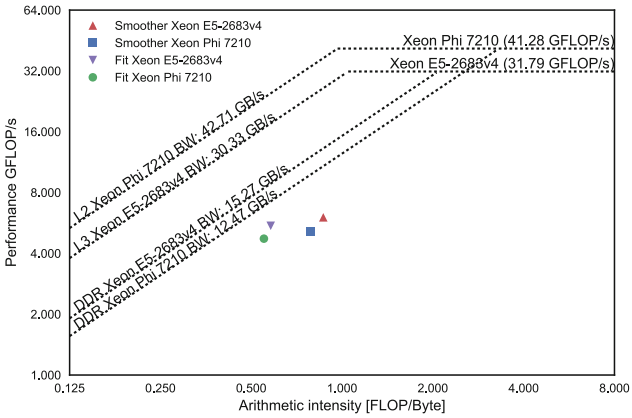

Fig. 5. Roofline model of Broadwell E5-2683v4 and Xeon Phi 7210 platforms. The performance of Cross Kalman Mathtest for the fit and smoother is shown for both platforms.

4 Validation

We have developed a module that implements the Cross Kalman filter inside the LHCb execution chain, named *TrackVectorFitter* (TVF). This module is already available to LHCb users and serves as the foundation for the numerical results described in this section. We have validated the physics performance of TVF against the original implementation under the current LHCb run conditions, and also under the foreseen conditions of the upgrade.

The LHCb experiment uses Monte Carlo simulation to generate validation data sets. Particle collisions and their interaction with the detector are simulated. This simulation generates a data set that can be processed by the LHCb reconstruction software. Finally the reconstructed particles are compared to the Monte Carlo generated ground truth.

Track reconstruction validation is done using three metrics [17]. The *reconstruction efficiency* compares the reconstructed tracks to the expected tracks reported by the Monte Carlo truth. The *clone rate* reports how many track equivalent track pairs were found. The *ghost rate* reports how many tracks were reconstructed with nodes belonging to different particles or noise. Finally, tracks are categorized by their physical properties and category statistics are compared to statistics from the ground truth.

Comparing the Cross Kalman implementation TVF to the original track filter TMF we observe an identical reconstruction efficiency, clone rate and ghost rate under all tested scenarios. While the reconstruction of the track itself does not depend on the fit, the final track χ^2 is used in the different categories as a track quality cutoff. Hence, the identical reconstruction efficiency between the two algorithms validates TVF for its physical properties.

We have checked the performance of TVF against TMF under various scenarios. Table 2 shows comparative execution times for LHCb nightly tests. These tests are representative of the conditions under which the LHCb reconstruction runs in the production environment.

Table 2. LHCb test times in seconds, run in various conditions. All tests are run on a single core of an Intel Xeon E5-2650 v3. All timings refer to the algorithm *TrackBestTrackCreator*, configured with different filter settings. TMF is the original filter implementation. Internally, it executes a vertically vectorized code optimized for AVX on this setup. TVF refers to our implementation, compiled with either the SSE2 extension (default setting for x86_64) or AVX2+FMA. The *overall reconstruction speedup* refers to the entire reconstruction time of the test, compared between TMF and TVF AVX2+FMA.

Test name	TMF (AVX)	TVF SSE2	TVF AVX2+FMA	Overall reconstruction speedup
Magup2016	13.518	12.817	11.504	1.09x
Baseline-upgrade	93.713	93.839	91.014	1.03x
Sim08	8.307	8.134	7.986	1.02x

We observe a varying performance depending on the test under execution. Magup2016 shows gains of up to 9% in the overall reconstruction time, whereas baseline-upgrade and sim08 gains in TVF do not seem to impact much the overall performance. In the case of baseline-upgrade, we believe this is due to the configuration of such test. It uses a *full geometry* setting in its current form, which dominates the time distribution of the fit. We expect its performance to improve in the future.

5 Conclusions and Outlook

In this work we have presented Cross Kalman, an algorithm that is able to efficiently perform low-rank Kalman filters. Cross Kalman is particularly optimized for the LHCb particle tracking use case, but the presented algorithms and data structures can be applied to other situations where a large number of low-rank Kalman filters are used. Using this algorithm we were able to obtain up to 3x speedup over the previous scalar solution on the same hardware platform. Our implementation is flexible enough to accommodate for any kind of SIMD architecture and we have tested it a wide array of architectures. The choice of the Decreasing-Time Algorithm as a scheduling algorithm should be revisited, and we intend to explore other heuristics in the future. Our data structures allow us to efficiently perform the Kalman filter and smoother of many independent particles in parallel. Given the specific nature of our problem instances, it may be possible to reuse data structures across different particle trajectories, and further decrease the memory footprint of our application. In addition, we have showed that single precision performance scales similarly to its double precision counterpart.

An in-depth analysis of the precision requirements and numerical stability of the algorithm, taking into account also the possibility of alternative mathematical formulations, should be carried out. We expect that moving to single-precision and thus doubling the arithmetic intensity of our algorithms will significantly improve performance. Our software is validated and has been integrated in the LHCb codebase under the name TrackVectorFitter, making the overall reconstruction up to 9% faster for certain datasets.

We have verified that our implementation is able to scale to full hardware nodes and is able to adapt to the architectures under study. As expected enabling SMT does not yield further performance improvements with the notable exception of Intel Xeon Phi, which could be due to its higher memory throughput. However, other algorithms used in the LHCb software framework need to be adapted to make the most out of manycore architecture before a more definite answer can be given to the suitability of manycore hardware platforms such as Intel Xeon Phi for LHCb's software framework.

Given the arithmetical intensity of our formulation, our application utilizes efficiently the processors under study. We intend to port our software to GPU accelerators and further analyze our software scalability. 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, and early access to Intel hardware. 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 Xeon Phi architecture. In addition, thanks to W. Hulsbergen and R. Aaij for the mathematical discussions and data structure design. Finally, thanks to N. Neufeld and A. Riscos Núñez for their guidance and support.

References

1. The LHCb Collaboration: framework TDR for the LHCb upgrade: technical design report. Technical report CERN-LHCC-2012-007. LHCb-TDR-12, April 2012. https://cds.cern.ch/record/1443882
2. The LHCb Collaboration: LHCb trigger and online upgrade technical design report. Technical report CERN-LHCC-2014-016. LHCB-TDR-016, May 2014. https://cds.cern.ch/record/1701361
3. Kalman, R.E.: A new approach to linear filtering and prediction problems. J. Basic Eng. **82**(1), 35–45 (1960). https://doi.org/10.1115/1.3662552
4. Mcgee, L.A., Schmidt, S.F.: Discovery of the Kalman filter as a practical tool for aerospace and industry. Technical report, November 1985. https://ntrs.nasa.gov/search.jsp?R=19860003843
5. Houtekamer, P.L., Mitchell, H.L.: Data assimilation using an ensemble Kalman filter technique. Mon. Weather Rev. **126**(3), 796–811 (1998). http://journals.ametsoc.org/doi/abs/10.1175/1520-0493%281998%29126%3C0796%3ADAUAEK%3E2.0.CO%3B2

6. Welch, G., Bishop, G.: An introduction to the Kalman filter. Technical report, Chapel Hill, NC, USA (1995)
7. Hulsbergen, W.: The global covariance matrix of tracks fitted with a Kalman filter and an application in detector alignment. Nucl. Instrum. Methods Phys. Res. Sec. A: Accel. Spectrom. Detect. Assoc. Equip. **600**(2), 471–477 (2009). http://www.sciencedirect.com/science/article/pii/S0168900208017567
8. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the 18–20 April 1967, Spring Joint Computer Conference, pp. 483–485. AFIPS 1967 (Spring). ACM, New York (1967). http://doi.acm.org/10.1145/1465482.1465560
9. Cámpora Pérez, D.H.: LHCb Kalman filter cross-architecture studies (2016)
10. Cerati, G., Elmer, P., Lantz, S., McDermott, K., Riley, D., Tadel, M., Wittich, P., Würthwein, F., Yagil, A.: Kalman filter tracking on parallel architectures. J. Phy. Conf. Series **664**(7), 072008 (2015). http://stacks.iop.org/1742-6596/664/i=7/a=072008
11. Aaij, R., Fontana, M., Le Gac, R., Zacharjasz, E.A., Schwemmer, R., Fitzpatrick, C., Albrecht, J., Grillo, L., Szumlak, T., Yin, H., Couturier, B., Stahl, S., Williams, M.R.J., Vries, D., Andreas, J., Seyfert, P., Wanczyk, J., Esen, S., Neufeld, N., Hasse, C., Vesterinen, M.A., Nikodem, T., Quagliani, R., Polci, F., Dziurda, A., Jones, C.R., Matev, R., De Cian, M., Del Buono, L.: Upgrade trigger: biannual performance update. Technical report, February 2017. https://cds.cern.ch/record/2244312
12. Mertens, S.: The easiest hard problem: number partitioning, October 2003. arXiv:cond-mat/0310317
13. Gou, C., Kuzmanov, G., Gaydadjiev, G.N.: SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In: Proceedings of the 24th ACM International Conference on Supercomputing, pp. 179–188. ICS 2010. ACM, New York (2010). http://doi.acm.org/10.1145/1810085.1810111
14. Fog, A.: VCL C++ vector class library (2012). http://www.agner.org/optimize
15. Karpiński, P., McDonald, J.: A high-performance portable abstract interface for explicit SIMD vectorization. In: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2017, pp. 21–28. ACM, New York (2017). http://doi.acm.org/10.1145/3026937.3026939
16. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65 (2009)
17. Schiller, M.: Track reconstruction and prompt K_S^0 production at the LHCb experiment. Dissertation, University of Heidelberg (2011)