Accelerating the RICH Particle Detector Algorithm on Intel Xeon Phi

Christina Quast and Rainer Schwemmer
CERN
Meyrin, Canton of Geneva, Switzerland
Email: cern@christina-quast.de,
rainer.schwemmer@cern.ch

Angela Pohl, Biagio Cosenza and Ben Juurlink
Technische Universität Berlin
Berlin, Germany
Email: {angela.pohl, cosenza,
b.juurlink}@tu-berlin.de

***Abstract*—At the LHC, particles are collided in order to understand how the universe was created. Those collisions are called events and generate large quantities of data, which have to be pre-filtered before they are stored to hard disks. This paper presents a parallel implementation of these algorithms that is specifically designed for the Intel Xeon Phi Knights Landing platform, exploiting its 64 cores and AVX-512 instruction set. It shows that a linear speedup up until approximately 64 threads is attainable when vectorization is used, data is aligned to cache line boundaries, program execution is pinned to MCDRAM, mathematical expressions are transformed to a more efficient equivalent formulation, and OpenMP is used for parallelization. The code was transformed from being compute bound to memory bound. Overall, a speedup of 36.47x was reached while obtaining an error which is smaller than the detector resolution.**

***Index Terms*—Intel Xeon Phi, Knights Landing, OpenMP, Vectorization, Parallel Programming**

I. INTRODUCTION

At the Large Hadron Collider (LHC) at CERN, particles are collided in order to understand how the universe evolved. In the LHCb experiment, proton-proton collisions are used to investigate the matter-antimatter asymmetry of the universe. An array of detectors collects data, which is preprocessed in a filter and forwarded to a hard drive based storage system. This filter performs a fast track reconstruction and comes to a preliminary decision which particle type might have caused the collision results. This paper focuses on the optimization of the filtering process.

When a particle travels through a medium with a speed faster than light, it emits photons, the so-called Cherenkov radiation [7]. These photons are emitted in a cone around the particle track and the cone's opening angle depends on the particle's speed. Using a Ring-Imaging Cherenkov (RICH) detector, a pixel matrix captures these photons as Cherenkov rings. Based on these rings, the particle's speed and consequently its type can be inferred. This is done with the RICH particle detector algorithm. Figure 1 presents an overview of the algorithm while a thorough description can be found in section III.

The process of inferring particles is computationally intensive, however. That is why the experiment is currently limited to rarely-occurring special cases. Nonetheless, the LHCb trigger, which decides which collision data to keep, and the readout system will be upgraded in 2018. After the upgrade, the LHCb experiment strives for a hardware-trigger-free readout using an event filter farm, which will process an event every 25 ns. With a collision rate of 40 MHz, the downlink from the detector to the event filter farm has to be increased from 500 Gbit/s to 40 Tbit/s [7]. It is therefore critical to speed up the RICH particle detector algorithm with a similar order of magnitude to keep up with the forthcoming upgrades.

As of today, the algorithm's production code, written in C++, is run on a x86-compatible CPU cluster. To keep up with the future detector hardware environment, we ported the particle identification algorithm to Intel's Xeon Phi Knights Landing (KNL) processor, a platform designed for HPC workloads [6]. In order to utilize all features this HPC platform has to offer, the following techniques were applied to the current production code:

- improving memory access patterns and data layouts in memory
- exploiting manual code vectorization using intrinsics
- exploiting multi-threading using OpenMP [5]
- replacing mathematical functions with their optimized counterparts
- using numerical approximations while obeying accuracy requirements

Using all of these techniques combined, we were able to achieve a 36.47x speedup over current production code running on the KNL with 256 OpenMP threads and 5263x over a single thread.

This paper is organized as follows: Section II discusses related work in the areas of algorithmic improvements on KNL, code vectorization, and mathematical optimizations. Section III details the RICH particle detector algorithm and presents the core mathematical functions that were optimized. Section IV briefly presents the platforms that were used in this work, while Section V thoroughly describes all optimization steps. Results are shown in Section VI and conclusions are provided in Section VII.

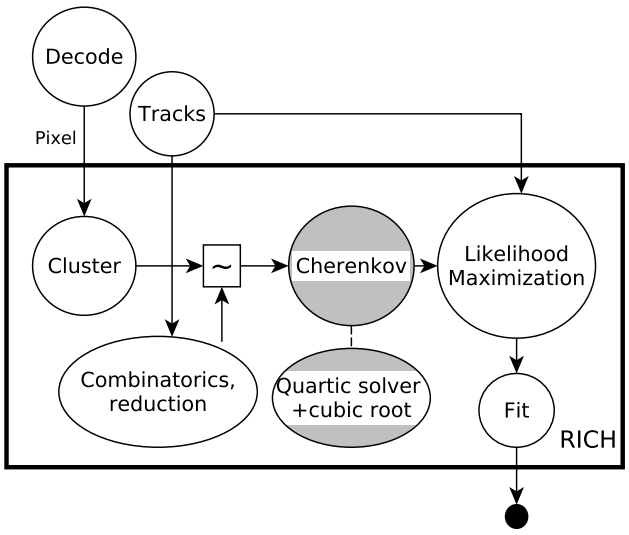

Fig. 1: Structure of the RICH algorithm. Grey circles show hotspots of algorithm which this paper focuses on.

II. Related Work

Forty et al. [9] describes the RICH pattern recognition algorithm based on the knowledge about the physics of the detector. An estimation of the accuracy and physical limitations is given. The Cherenkov angle algorithm, which this paper focuses on, is also described in detail.

Färber et al. [7] researches the acceleration of the Cherenkov angle reconstruction for the LHCb experiment on an FPGA. They reach a calculation time of around $10\ ns$ per photon for $2 \cdot 10^6$ photons, which is around 50 times as long as our best time for per photon using all improvements and OpenMP threads.

Ramos and Hoefler [18] reported that it was difficult to achieve the maximum MCDRAM bandwidth with their memory benchmark in SNC4-flat mode tuned for the KNL architecture. Reaching this bandwidth was not possible with our code because it is compute bound. Instead of the nominal $340\ GB/s$, we reached up to 80% of this bandwidth.

Haidar et al. [10] researched the performance of the LU, QR and Cholesky factorization on the KNL in an attempt to design a programming model which would provide a portable and efficient matrix decompositions across hybrid environments. Using the MKL, they reached a performance of up to $1.4\ TFLOP/s$ for a Dynamic-MAGMA QR decomposition in double precision, which is about half the double precision FMA performance possible on this platform.

LaGrone et al. [13] performed benchmarks on a 16 core Nehalem to determine the overhead for different OpenMP constructs. They found out that the overhead increases faster than linear using the Intel C and GNU C compiler.

Wende et al. [20] explored the possibilities of a programmer to improve the performance of his code using SIMD. The authors presented examples where compiler vectorization failed, and how the programmer can adjust his code to help the compiler to vectorize. They enhanced code to enable compiler autovectorization, which led to a performance which was comparable to or most of the time even better than code using intrinsics. The authors suggest defining custom vector data types which are multiples of the hardware's native SIMD vector length or width, replacing scalar function arguments by the vector counterparts and extending function bodies with a SIMD loop (loop count equals SIMD vector length, so that the compiler can automatically generate vector instructions from that).

Pankiewicz et al. [15] describe an algorithm for efficiently calculating a polynomial and its derivatives. The algorithm is based on additions and multiplications, which can be calculated in one cycle on processors supporting Fused Multiply-Add (FMA) instructions.

III. Algorithm

A. Overview

Figure 1 depicts the full RICH particle detection algorithm. This code is one stage of a sequence of algorithms that aim to reconstruct the particle properties within the detector. Input to the algorithm are pixel images recorded by the RICH detector, whose sensor works similar to a digital camera. Photons could have spilled over to neighboring detector cells, and they need to be clustered before processing them further. As a second input, the algorithm receives the particle's track from a previous stage of the event reconstruction. Using this information, impossible combinations are eliminated early on. With this clustered, reduced photon set and the corresponding track information, the Cherenkov angle for every track and pixel cluster combination is calculated. Then, the probability of producing this set of Cherenkov angles is calculated for each of the detectable five particle types. A likelihood maximization stage finally selects the most probable combination of the five particles types (e, μ, π, K, p) which could have produced this exact combination of tracks and clusters.

B. Cherenkov Angle Calculation

After performing extensive profiling of the algorithm, we have found that the Cherenkov angle calculation was the computational bottleneck. It took up one third of the total execution time in the production code and therefore required a thorough analysis to identify possible improvements.

Figure 2 (b) depicts a particle track $\vec{t}$, as well as the track of its emitted photon $\vec{p}$. The angle between these two tracks θ_c is the opening angle of the cone that is captured as Cherenkov rings in the RICH detector. Knowing the Cherenkov emission angle θ_c enables particle type inference.

Figure 2 (a) presents a schematic of a particle passing through the RICH detector from the side view. The particle travels along the track t and emits a photon at point E. The photon is reflected at point M on the mirror segment, which has its center of curvature at point C, and is detected in point D. The Cherenkov angle θ_c is the angle between $E\vec{M}$ and the particle track t. The algorithm receives the radius R and

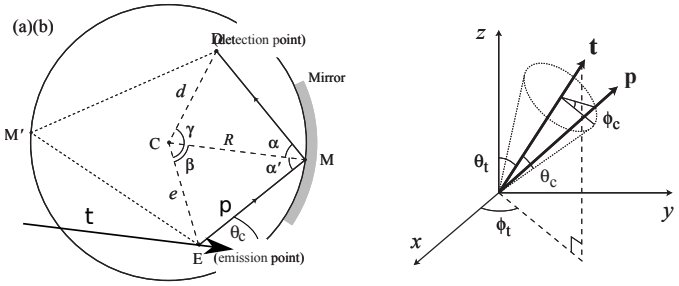

Fig. 2: Cherenkov angle reconstruction [9]

cartesian coordinates of the center of curvature of the mirror C as well as the coordinates of the emission point E and detection point D. The output of the algorithm is the spherical reflection point M.

As a first step, we calculate $\sin(\beta)$ using the fourth grade polynomial, which describes the trigonometric relation between the distance of the detection point d and emission point e from the center of the mirror curvature (Eq. 1). The coefficients of the polynomial have to be calculated first, using the known values R, d and e, where $d_y = d\sin\gamma$ and $d_x = d\cos\gamma$. Next, the reflection point M is obtained by rotating the vector $\vec{e}$ around C by an angle β and scaling it to length R.

$$\begin{aligned} &4e^2d^2\sin^4(\beta) - 4e^2d_yR\sin^3(\beta) \\ &+ \big(d_y^2R^2 + (e+d_x)^2R^2 - 4e^2d^2\big)sin^2(\beta) \\ &+ 2ed_y\big(e-d_x\big)R\sin(\beta) + \big(e^2-R^2\big)d_y^2 = 0 \end{aligned} \tag{1}$$

Finally, the Cherenkov emission angle θ_c can be calculated from the reflection point M, using the formula $\cos(\theta_c) = \vec{p}\cdot\vec{t}$ where $\vec{t}$ is the unit vector along the track direction and $\vec{p}$ the unit vector along the trajectory of the photon [9]. Since the calculation of the Cherenkov angle θ_c is independent for each photon, the code can be efficiently parallelized so that photon emission angles are calculated by different threads and vector units without the need of extensive synchronization.

IV. Platforms

A. Intel Xeon Phi (Knights Landing)

Intel's Xeon Phi 7210 platform is based on the Silvermont mircoarchitecture, with significant changes for HPC use. It is a manycore CPU with 64 cores and up to four hyperthreads per core [3]. Each core has a 32 KB L1 cache, and the 1 MB L2 cache is symmetrically shared between cores. Two cores are organized into one tile, and the L2 cache is coherent across all tiles. KNL utilizes a mesh infrastructure to connect them.

Every core is extended by two vector processing units, which support the AVX-512 Foundation (F) instruction set, and incorporate an Extended Math Unit (EMU) for scientific functions, such as exp, log, or sin.

Furthermore, KNL utilizes Multi-Channel DRAM (MCDRAM), a High Bandwidth Memory (HBM). The MCDRAM can be used as a transparent Last Level Cache (LLC), sitting between the L2 cache and main memory. Alternatively, it can be configured in flat mode to extend the DRAM, delivering up to $340\ GB/s$ while a DDR-RAM delivers up to $80GB/s$.

As opposed to its predecessor, codenamed Knights Corner, the KNL instruction set is based on standard x86 ISA and the compiled code is therefore binary compatible to all x86 architectures. The peak performance of the KNL is 6 TFLOPS for single precision and 3 TFLOPS for double precision floating point arithmetics, which can be achieved when using fused-multiply-add (FMA) instructions exclusively.

B. Intel Xeon Platinum 8170 (Skylake)

Since the latest generation of Xeon Phi platforms is binary compatible to x86, it is possible to compare performance of the same code base on different hardware platforms. We therefore chose a recent Intel Xeon Platinum 8170 processor, codenamed Skylake, as a reference. It is a 26 core CPU with up to two threads per core, running at $2.1\ GHz$. It also supports the AVX-512 instruction set. Skylake has an attached $35.75\ MB$ L3 cache and three Ultra Path Interconnect (UPI) links between the processors.

V. Optimization Techniques

A. KNL Hardware Configuration

KNL offers different clustering modes to tune memory organization to an application's needs. Cores and attached MDCRAM are grouped into logical Non-Uniform Memory Access (NUMA) units and the user can chose between all-to-all, quadrant, hemisphere and sub-NUMA cluster configurations. The all-to-all mode is recommended for debugging purposes only [19].

Since the RICH algorithm is a streaming problem, where photons can be buffered temporarily, high bandwidth is more important than low latency. It is therefore not necessary to use sub-NUMA clusters for manual latency tuning. This leaves the choice between hemisphere and quadrant mode, where sets of tiles are divided into clusters of two or four, respectively. To optimize for high bandwidth, photon data must be located in HBM instead of DRAM. In quadrant mode, for example, the latency of accessing the HBM is about $10\ \%$ higher than when accessing the DRAM, but the bandwidth of the HBM is about seven times higher. Also, data can be held spatially local in smaller groups in this setup. Hence the latency of L2 cache misses is shorter due to a shorter worst case path [1].

We therefore configured the KNL in quadrant mode with MCDRAM in flat mode, and pinned the execution of the program to MCDRAM, so that the photon data was located in this HBM instead of DRAM.

B. Memory Layout

For each photon, we store three times three floats, i.e. the x, y, and z values of the emission point E, the center of curvature C, and the detection point D, as well as the radius R. In addition to these 40 Bytes, we store the calculated output, i.e. the x, y, and z coordinates of the spherical reflection point M, which results in a total of 52 Bytes per photon.

The current production code utilizes Array of Structures (AoS) to store photon data. To enable contiguous memory access patterns, we therefore transformed the data layout to Structure of Arrays (SoA). It is now possible to access the same input data of several photons without gather/scatter operations. We did not tested more complex intermediate layouts such as tiled-AoS [11].

Furthermore, the compiler does not align data to cache line boundaries per default. Therefore, unaligned load and store instructions are used. To avoid these unaligned memory accesses, we directed the compiler to align data structures on cache line boundaries via `__attribute__((__aligned__(64)))` and used aligned allocators only.

C. Algorithm Adjustments

1) Replacing the quaternion with a rotation matrix: In the production RICH code, quaternions are used for vector rotations. The implementation of one quaternion includes several layers of C++ abstractions and is deeply entangled with the CERN HEP data processing framework Gaudi [4].

$$\begin{aligned} \mathbf{L} &= \begin{pmatrix} 0 & n_z & -n_y \\ -n_z & 0 & n_x \\ n_y & -n_x & 0 \end{pmatrix} \\ \mathbf{M} &= \vec{e} \cdot \left(I + \frac{\sin(\beta)}{|\vec{n}|} + \frac{1-\cos(\beta)}{|\vec{n}|^2} L^2 \right) \\ where\ \vec{n} &= \frac{\vec{e} \cdot \vec{d}}{|\vec{e} \cdot \vec{d}|} \end{aligned} \tag{2}$$

The quaternion function expects an angle β and a vector as parameters. Internally the function then calculates $\sin\beta$ and $\cos\beta$ to perform the actual rotation. Since we already have $\sin\beta$ from the polynomial solving stage, we decided to replace this library function with our own rotation matrix, where instead of the implicit $\cos(\arcsin(\sin\beta))$ we can use the previous result directly and replace $\cos\beta$ with its trigonometric equivalent: $\sqrt{1-sin^2\beta}$. In this manner, three trigonometric functions were replaces by a square root, a subtraction and a multiplication, which reduces the number of executed micro-operations to three (see [8]).

To reduce the number of divisions performed to normalize the vector, the reciprocal of the norm was calculated in advance, so that afterwards each vector could be normalized with one multiplication, which takes only one micro-operation to perform instead of the 18 ops a division would take [8].

2) Replacing the quartic solver with Newton-Raphson: If the general form of a function is known, instead of solving a quartic equation, it is also possible to find the function root in a restricted range of function arguments by using iterative approximation. The computationally intensive quartic solver including a cubic root could be replaced with an iterative method like Newton-Raphson. The geometry of the problem puts certain constraints on the quartic equation, which allows us to always end up with the correct of the four possible roots and ensures that this solution is not complex. The possible emission angles between 0 and 60 constrain the value of $\sin(\beta)$ to between 0 and 0.5. Furthermore, the function is monotonically increasing within this interval, which ensures the stability of Newton-Rhapson.

Starting in the middle of the possible interval, single float precision can be reached already after four iterations.

We chose to implement Newton as specific case of Householder using Horner's method. The algorithm is described in [15]. There, the values of r_0, r_1 are calculated as follows:

$$r_i = d^i \left(\sum_{j=0}^{n} (a_j \cdot x^j) \right) / dx^i \tag{3}$$

Here, n is the degree and a_j are the coefficients of the polynomial. Newton-Raphson requires an order of one, therefore $i \in 0, 1$. The next iteration of x is calculated as follows:

$$x = x - g \cdot \frac{r_0}{r_1} \tag{4}$$

Since this routine should also work with vectorization, it is important to keep the number of iterations constant to not introduce unnecessary branching. The factor g was introduced to increase convergence speed for the cases where the root is far away from the initial value. This value was determined experimentally to be 1.04 and approaches the needed resolution with one iteration less for far off starting values. At the same time it slightly decreases convergence speed for closer starting values, but not more than bad starting points.

While in the monomial form of a polynomial of degree n, we need at most n additions and $2n-1$ multiplications, using Horner's method we can reduce it to n additions and n multiplications [14]. The compiler can unroll the constant size for loops, which results in a branch-free code that is optimized for the grade of the polynomial. Furthermore, using Horner in conjunction with Householder makes the code FMA-friendly and improves processing speed for the root approximation.

While Householder of a higher order in conjunction with Horner would have been a possible candidate for the implementation with the better convergence rate [15], this method resulted in more calculations and floating point overflows, so that we favored Newton-Raphson.

D. SIMD implementation

To exploit the application's data level parallelism, the code was transformed to efficiently utilize the Single Instruction Multiple Data (SIMD) hardware extensions in the VPUs. The Xeon Phi supports the AVX-512 instruction set, and by using 512 bit wide registers, it is thus possible to process 16 single precision floating point numbers in parallel.

Furthermore, hardware scatter/gather operations were added in this latest SIMD ISA, as well as approximate exponential and reciprocal functions. These are of special interest for physics applications, where instructions such as `sqrt` or `div` are used often. If they can tolerate a certain approximation

Instruction	μops	Reciprocal Throughput
VSQRTPS	18	16
VRSQRT14PS	1	3
VDIVPS	18	32
VRCP28PS	1	3

TABLE I: Micro-operations and throughput for reciprocal `sqrt` and `div` functions on KNL [8]

error, using these instructions results in significantly less cycles and hence a substantial speedup can be achieved.

Efficiently utilizing SIMD hardware is still challenging, however, and the right programming model has to be chosen carefully based on the individual application [17]. Campora et al. therefore evaluated the C vector libraries Vc [12], Vector Class Library (VCL) [2], as well as gcc and Intel intrinsics. Based on perfomance, scalability, readability and maintainability, they rated each programming model for the LHCb reconstruction software use case [16]. Since both intrinsic options are compiler and architecture dependent, the choice was narrowed down to the Vc and VCL language libraries. While both exhibit similar performance, Vc is not geared towards vertical vectorization, which makes it less readable. Hence, Agner Fog's VCL is the library of choice. It provides an abstraction layer to the underlying hardware intrinsics, supports AVX-512, and when a function is not directly supported in hardware, it implements its own optimized version with available intrinsics.

With VCL, utilizing approximate functions in the Cherenkov angle calculation is done by replacing `div()` with `approx_recipr()` and `sqrt()` with `approx_recipr(approx_rsqrt())`, respectively. The micro-operations and throughput for the regular and reciprocal versions of `sqrt` and `div` are listed in Table I.

E. OpenMP

Besides data level parallelism, we need to exploit thread level parallelism to take advantage of KNL's 64 cores. For this purpose, we used the OpenMP framework for straight-forward multi-threaded execution, where the number of OpenMP threads and work group size could be set via command line. Due to the embarrassingly parallel nature of the problem, a parallel for loop over the input stream of photons was sufficient to parallelize the program.

One tunable parameter is the work group size and scheduling strategy. If dynamic scheduling is chosen, the OpenMP runtime determines the work group size automatically using the number of loop iterations and threads to be used. If static scheduling is selected, the scheduling order is determined at compile time and the programmer can chose the work group size manually. If the work group size is too small, it results in more OpenMP overhead from spawning and syncing threads. If the work group size is too big, the overall process can be stalled waiting for one slow thread to finish. To keep OpenMP overhead to a minimum, we subdivided the input stream in appropriately-sized chunks and set a static scheduler for the parallel for loop.

Code Version	t_{photon}	**Speedup**
Baseline	1000.26 ns	-
Baseline + OpenMP	7.13 ns	1.00
Baseline + OpenMP + ...		
Pinned to MCDRAM	6.63 ns	1.07x
Algorithmic adjustments	4.67 ns	1.53x
Vectorization + Mem. alignment	0.93 ns	7.64x
All of the above	0.196 ns	36.47x
Algorithm + Vectorization	0.83 ns	8.55x
Algorithm + MCDRAM	4.30 ns	1.66x
Vectorization + MCDRAM	2.04 ns	3.49x

TABLE II: Photon processing times and resulting speedups after code optimization

We chose static scheduling, because the number of photons is known in the beginning and the work does not contain branches, so the work to be done by each thread is known at compile time and all work packets need the same time to finish.

VI. EXPERIMENTAL RESULTS

A. Measurement Setup

To measure the impact of our different optimization techniques, it was crucial to define a workload large enough that no threads will be idle during execution. As an estimate for the number of photons processed in parallel, we used the formula

$$\begin{aligned} photons_{pex} &= vectorsize \cdot workgroupsize \cdot threads \\ &= 16 \cdot 1024 \cdot (64 \cdot 4)\ photons \\ &= 2^{22}\ photons \end{aligned}$$

To process a small stream of data, this number was multiplied with a factor of eight, resulting in an input workload size of

$$\begin{aligned} n_{photons} &= 2^{22} \cdot 2^3\ photons \\ &= 2^{25}\ photons = 33554432\ photons \end{aligned}$$

This data size fits well into an integer sized multiple of cache lines and is sufficient to ensure that all threads have enough data to process.

The processing time per photon was measured by taking the high resolution clock provided by `chrono`. A time stamp was taken before processing the OpenMP `parallel_for`-loops, i.e. before OpenMP's thread scheduling. The measured time therefore includes the associated overhead. To obtain the processing time per photon, the measured time was divided by the number of performed iterations and calculated photons:

$$t_{photon} = \frac{t_{all}}{n_{iter} \cdot n_{photons}}$$

B. Overview of Results

A summary of the photon processing times and speedups after applying aforementioned code optimizations is presented in Table II.

Compared to the scalar code version running on a single core, using all available hyperthreads with OpenMP yields a speedup of ≈ 143. The resulting processing time $t_{photon} =$

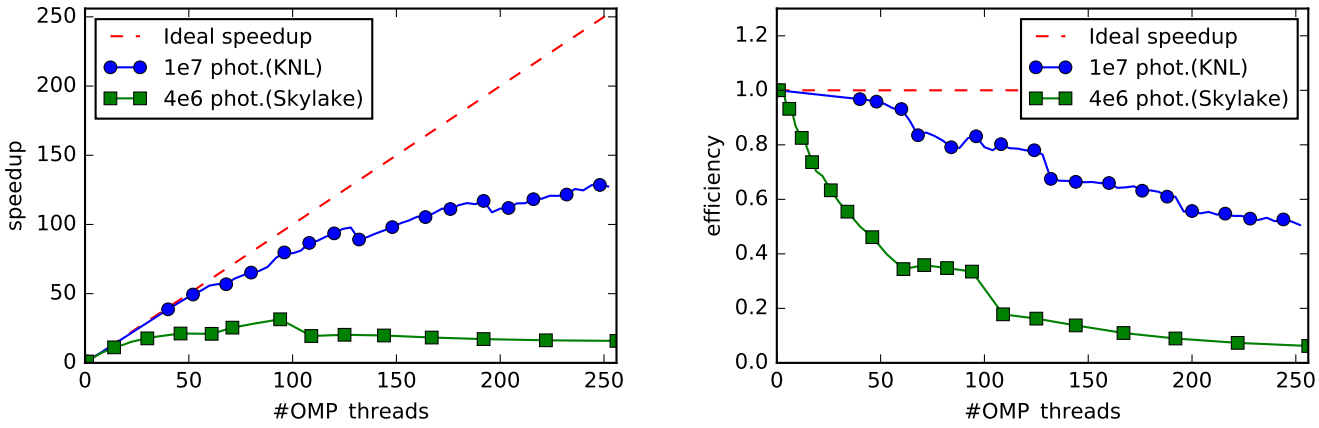

Fig. 3: Strong scaling speedup (left) and efficiency (right) for $1 \cdot 10^7$ (KNL, OMP work group size 128) and $4 \cdot 10^3$ photons (Skylake, OMP work group size 1024)

7.13 ns is taken as a baseline to compare all further optimization techniques, since they were applied to this OpenMP-enhanced code base.

Pinning the algorithm's processing data to MCDRAM improves the processing time by a mere 7%. Applying this technique only, memory accesses are sped up, but the algorithm is still compute bound and cannot exploit the higher bandwidth.

Changing the math of the algorithm results in less cycles to perform the same calculations, and this alone yields a speedup of 1.53x. The biggest improvement is achieved after vectorizing the code and applying memory alignment restrictions. The processing time is reduced by a factor of 7.64x, which is comparable to the results achieved on our Xeon reference machine (7.7x); in conjunction with the algorithmic adjustments, the code runs 8.55x faster.

With these two modifications, the code is no longer compute bound, but memory bound. If we now pin the program data to the faster MCDRAM instead of using DRAM, a significant overall speedup of 36.47x is achieved. The processing time per photon has been reduced to $t_{photon} = 0.196\ ns$

Based on the MCDRAM specifications, we can calculate the theoretical minimal processing time per photon to assess the quality of our code optimization. According to the hardware specification, the MCDRAM's bandwidth is limited at 340 GB/s. Taking the photon's data structure size of 52 B into account, the minimal photon processing time equates to

$$t_{min} = \frac{data\_size}{BW_{MCDRAM}} = \frac{52B}{340\ GB/s}$$

$$= 0.153\ ns$$

Hence, using OpenMP for thread scheduling, MCDRAM, algorithmic and mathematical improvements, and vectorization, we are able to process one photon in $0.195ns$, which is 28% below the theoretical limit for memory bound problems.

C. Multithreading Efficiency

Fig. 3 shows speedup and efficiency. The speedup is calculated by dividing the time the calculation takes using one OpenMP thread by the time it takes using n threads (see Eq. 5). The efficiency is the speedup divided by the number of threads n.

$$speedup = \frac{t_{1thread}}{t_{nthreads}} \tag{5}$$

In the efficiency plot in Figure 3 it can be seen that the efficiency is a plateau with an increasing number of threads unless the thread number is in vicinity of multiples of the 64 physical cores for the KNL and multiples of the 26 cores for the Skylake CPU, where we see a sudden drop in efficiency. The efficiency is worse for the Skylake than for the KNL, where the efficiency drops already by 0.4 between using one and 25 threads.

An ideal speedup based on Amdahl's law is shown as a red dashed line in Figure 3. When using multiple threads, thread scheduling and management introduces an overhead which diminishes the speedup.

At 52 threads, the Skylake has an efficiency of about 35%, while the KNL reaches an efficiency of 80% for the maximum number of hardware threads it supports. With every additional hyperthread used, the speedup graph drops by around 10. The inclination is 1 for the KNL up until around 50 threads, and about 0.6 starting from 100 threads for the KNL. The degradation can be justified with the limited hardware resources of a core, which have to be used by all hyperthreads simultaneously.

D. OpenMP

In Figure 4 we can see the OpenMP overhead distributed over the number of OpenMP threads used for calculation. The overhead in nanoseconds is calculated as follows, where t_1 is the runtime per photon using one thread, and t_n is the time

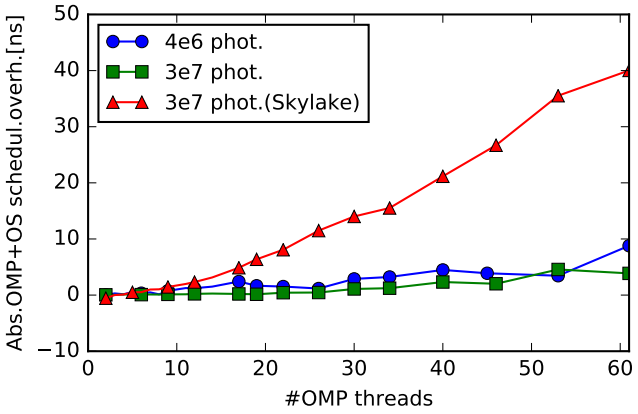

Fig. 4: Absolute OpenMP overhead.

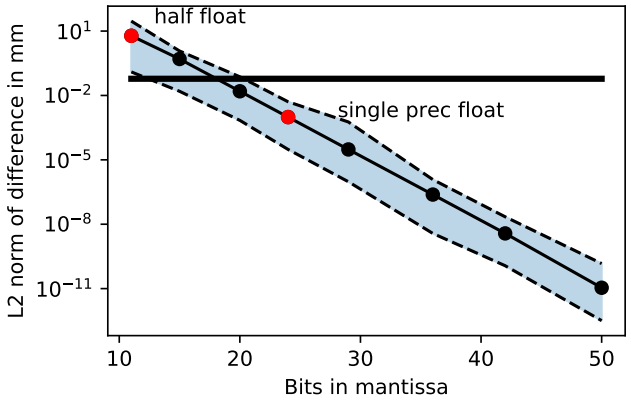

Fig. 5: Precision error based on mantissa size; shaded area shows error range, black line represents minimum required precision)

needed to calculate the reflection point for one photon using n OpenMP threads [13]:

$$T_{overhead} = n \cdot T_{nthreads} - T_1 \tag{6}$$

The overhead was measured until 64 cores, because after that hyperthreads are used, which leads to additional hardware scheduling overhead and resource scheduling allocation which OpenMP does not account for. The absolute Skylake OpenMP overhead per photon increases, while for the KNL the overhead stays approximately constant.

With perfect scaling, the overhead would be negligible. The overall overhead for the Skylake increases linear by approximately $0.7\ ns$ per thread added. The increase of overall overhead for the KNL remains at $0.13\ ns$ per thread added.

E. Accuracy Requirements

The detector has a length of approximately $1\ m$ and a resolution of $0.58 \cdot 10^{-3}\ rad$. A photon emitted at the furthest distance from the mirror will have an uncertainty of $1\ m \cdot 0.58 \cdot 10^{-3}\ rad \approx 0.06\ mm$ in its reflection point on the mirror. Ensuring a numeric precision that is higher than $0.06\ mm$ means that the detector resolution remains the dominant source of error. Consequently, the approximated functions introduced during code optimization do not impact overall results.

To verify the precision of the new implementation of the quartic solver, a Python version of the code was created based on the C++ code version. Using the Python module `mpmath`, the mantissa bits for the floating point numbers can be specified. The Python version was run with varying mantissa bits and performed with 10000 photons per mantissa size as input to the algorithm. The numeric resolution over this range of mantissa bits is plotted in Fig. 5. The L2-distance between the values is calculated using double precision (53 mantissa bits) and compared to the value calculated with less mantissa bits, where single precision (24 mantissa bits) and half floats (11 bits in mantissa) are marked explicitly. The distance between two points was calculated using the L2 norm of the distance:

$$d(\vec{a}, \vec{b}) = \sqrt{(|x_a - x_b|^2 + |y_a - y_b|^2 + |z_a - z_b|^2)} \tag{7}$$

As can be seen in the figure, a mantissa size of 21 is sufficient to meet the precision requirements. In other words, the error inferred into the lower bits of the mantissa by using approximate functions does not impact the overall quality of the results, whose error can be significantly larger due to measurement inaccuracies. With the given RICH detector precision, it is thus neither necessary to use double precision values, nor does the use of approximated functions introduce critical error.

F. Performance Outlook after System Upgrade

In 2018, the LHCb hardware will be upgraded. After the upgrade, data must be processed after each collision. Each collision produces $130\ kB$ of data at a frequency of $40 MHz$, which results in an overall bandwidth of $130\ kB \cdot 40\ MHz = 5.2\ TB/s = 40\ Tbit/s$. Using the software trigger, which the Cherenkov pattern recognition algorithm is part of, it is scaled down to $2 - 5\ GB/s$, which will be stored to hard drives.

In one collision event, around 50 tracks with 200 photons are registered, summing up to 10^4 photons per event. Assuming $52\ B$ per photon, one KNL needs to process photons with a minimum bandwidth of $10^4 \cdot 40\ MHz \cdot 52\ B = 20\ TB/s$. Using the MCDRAM with a bandwidth of $340\ GB/s$, we would need $\frac{20\ TB/s}{340\ GB/s} \approx 59$ KNLs to supply the required bandwidth. Using data pools, each photon can be described in $32\ B$, so that we would need $\frac{10^4 \cdot 40\ MHz \cdot 32\ B}{340\ GB/s} \approx 38$ KNLs to handle the minimum bandwidth.

With the presented code improvements, photons can be processed on the KNL at a frequency of $\frac{1}{0.195\ ns} = 5.13\ GHz$. Each photon can be processed in $0.195 ns$, so that we would need $10^4 \cdot 40\ MHz \cdot 0.195\ ns = 78$ KNLs to compute all the photon emission angles, although the memory could deliver data even faster.

VII. CONCLUSIONS

In this paper, the hotspot of the RICH particle detection algorithm was analyzed and ported to an Intel Xeon Phi Knights Landing platform. For the Cherenkov angle reconstructions, photon data processing exhibits parallelism on multiple layers that can be exploited well with a manycore architecture. Multithreading was applied using the OpenMP framework, which spawns threads and distributes the work equally over all threads.

Due to the compute intense nature of the algorithm, the baseline production code was compute bound. Applying vectorization, data structure transformations and memory alignment restrictions increased the performance by a factor of 7.65x. The algorithm itself was improved mathematically, as slow trigonometric functions were replaced and efficient approximate instructions were introduced. Also, the Newton-Raphson method was used to determine the root of the polynomial instead of solving a quartic equation. Our analysis shows that the precision after the changes is within the acceptable error range. These additional changes increased the speedup to 8.55x and the problem became memory bound.

As a consequence, the data structure layout was changed from AoS to SoA, alignment restrictions were enforced and the processed data was pinned to KNL's MCDRAM during exection.

Combining all of these techniques, a speedup of 36.47x compared to production code was achieved for the Cherenkov angle calculation. Assessing the maximum available memory bandwidth, the improved code is merely 28% below the theoretical limit. We thus showed that a high speedup is feasible for HEP code using current vectorization and parallelization techniques.

ACKNOWLEDGMENT

The work is part of the HTCC collaboration between Intel and CERN. The authors would like to thank LHCb and especially Omar Awile for their technical and professional support.

REFERENCES

[1] https://colfaxresearch.com/knl-numa/. Accessed: 2017-05-21.

[2] http://www.agner.org/optimize/#vectorclass. Accessed: 2017-09-13.

[3] Intel xeon phi prozessor 7210. https://ark.intel.com/de/products/94033/Intel-Xeon-Phi-Processor-7210-16GB-1_30-GHz-64-core. Accessed: 2017-08-30.

[4] G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank, G. Gracia, J. Harvey, E.van Herwijnen, P. Maley, P. Mato, S. Probst, and F. Ranjard. Gaudi a software architecture and framework for building hep data processing applications. *Computer Physics Communications*, 140(1):45 – 55, 2001. CHEP2000.

[5] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[6] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 2015:9, 2015.

[7] C. Faerber, R. Schwemmer, J. Machen, and N. Neufeld. Particle identification on an fpga accelerated compute platform for the lhcb upgrade. *Real Time Conference (RT), 2016 IEEE-NPSS*, 2016.

[8] Agner Fog. 4. instruction tables. http://www.agner.org/optimize/instruction_tables.pdf. Accessed: 2017-07-30.

[9] R. Forty and O. Schneider. RICH pattern recognition. *LHCB/98-040*, 30 April 1998.

[10] Azzam Haidar, Stanimire Tomov, Konstantin Arturov, Murat Guney, Shane Story, and Jack Dongarra. Lu, qr, and cholesky factorizations: Programming model, performance analysis and optimization techniques for the intel knights landing xeon phi. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–7. IEEE, 2016.

[11] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic data layout optimizations for gpus. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 263–274, 2015.

[12] M Kretz. *Efficient use of multi-and many-core systems with vectorization and multithreading*. PhD thesis, Diplomarbeit, University of Heidelberg 2009. URL http://compeng. uni-frankfurt. de/index. php, 2009.

[13] James LaGrone, Ayodunni Aribuki, and Barbara Chapman. A set of microbenchmarks for measuring OpenMP task overheads. 2:594–600, 2011.

[14] Alexander M. Ostrowski. On two problems in abstract algebra connected with horner's rule. *Studies in Mathematics and Mechanics presented to Richard von Mises*, pages 40–48, 1954.

[15] W Pankiewicz. Algorithms: Algorithm 337: calculation of a polynomial and its derivative values by horner scheme. *Communications of the ACM*, 11:633, 1968.

[16] Daniel Hugo Cámpora Pérez and Ben Couturier. Simd studies in the lhcb reconstruction software. In *Journal of Physics: Conference Series*, volume 664, page 092004. IOP Publishing, 2015.

[17] Angela Pohl, Biagio Cosenza, Mauricio Alvarez Mesa, Chi Ching Chi, and Ben Juurlink. An evaluation of current simd programming models for c++. *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, page 3, 2016.

[18] Sabela Ramos and Torsten Hoefler. Capability models for manycore memory systems: A case-study with xeon phi knl. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 297–306. IEEE, 2017.

[19] A. Vladimirov and R. Asai. Clustering modes in Knights Landing processors: Developer's guide. *Colfax International*, May 11, 2016.

[20] Florian Wende, Matthias Noack, Thomas Steinke, Michael Klemm, Chris J Newburn, and Georg Zitzlsberger. Portable simd performance with openmp* 4. x compiler directives. pages 264–277, 2016.