A fast local algorithm for track reconstruction on parallel architectures

Daniel Hugo Cámpora Pérez*†
* CERN
CH-1211 Geneva 23
Geneva, Switzerland
Email: dcampora@cern.ch

Niko Neufeld*

Agustín Riscos Núñez†
† Research Group on Natural Computing
Universidad de Sevilla
ETSI Informática, Av. Reina Mercedes, s/n,
41012, Sevilla, Spain

***Abstract*—The reconstruction of particle trajectories, tracking, is a central process in the reconstruction of particle collisions in High Energy Physics detectors. At the LHCb detector in the Large Hadron Collider, bunches of particles collide 30 million times per second. These collisions produce about 10^9 particle trajectories per second that need to be reconstructed in real time, in order to filter and store data. Upcoming improvements in the LHCb detector will deprecate the hardware filter in favour of a full software filter, posing a computing challenge that requires a renovation of current algorithms and the underlying hardware.**

We present Search by triplet, a local tracking algorithm optimized for parallel architectures. We design our algorithm reducing Read-After-Write dependencies as well as conditional branches, incrementing the potential for parallelization. We analyze the complexity of our algorithm and validate our results.

We show the scaling of our algorithm for an increasing number of collision events. We show sustained tests for our algorithm sequence given a simulated dataflow. We develop CPU and GPU implementations of our work, and hide the transmission times between device and host by executing a multi-stream pipeline.

Our results provide a reliable basis for an informed assessment on the feasibility of LHCb event reconstruction on parallel architectures, enabling us to develop cost models for upcoming technology upgrades. The created software infrastructure is extensible and permits the addition of subsequent reconstruction algorithms.

I. Introduction

LHCb is a large particle physics detector operating at the CERN Large Hadron Collider [1]. From 2020 on it will produce data at a rate of 40 Tbit/s [2]. A data selection will be performed in order to record interesting events [1] from a particle physics standpoint. The data acquisition system will be upgraded [3] to process all events in a commodity processor farm, deprecating the current hardware trigger. The increase in data rate and the removal of the hardware trigger pose a real-time computing challenge.

Different solutions are being studied to be able to process this enormous volume of data. The current LHCb trigger farm is composed solely of Intel Xeon-based servers, however the recent adoption of alternative architectures and accelerators in other detectors' data acquisition systems are an indication that other solutions may also be feasible [4] [5] [6]. Software demonstrators are fundamental towards implementing new architectures to the LHCb trigger farm, where price performance and software maintainability aspects should be taken into account.

Track reconstruction consists in determining the trajectories of particles from the signal pixel *hits* left on their path. The upgraded vertex locator (Velo) detector will span 52 consecutive silicon pixel modules, placed very closely to the interaction point [7]. The Velo reconstruction constitutes the first stage of tracking in LHCb. Tracks created at this stage are used for determining the locations of the collisions, and serve as a seed and are extended to subsequent LHCb tracking detectors. Hence, the Velo reconstruction is fundamental for the correct functioning of LHCb.

Various track reconstruction techniques have been explored in literature. Local methods find tracks iteratively. The baseline LHCb Velo reconstruction algorithm consists in a *track forwarding* technique, based on finding candidate pairs and extending them over iterative detector modules [8]. The need for flagging visited hits sequentially makes this technique unsuitable without modification to parallel architectures. Finding all compatible triplets can be parallelized dropping the flagging mechanism, like in the seeding phase of [9]. However, this is inefficient for densely populated detectors. Local methods are commonly used in conjunction with an estimator like the Kalman filter [10] to fit forming tracks and select hits [11]. Spatial reductions like KD-tree structures [12] or search windows help reduce the dimensionality of hits under consideration.

On the other hand, global methods adapt an equivalent formulation of the problem, where solutions map to tracks. The Hough transform [13] [14] converts all hit points into a histogram representation in polar coordinates, where peaks are equivalent to compatible hits. The Retina algorithm [15] builds a heatmap for each hit to determine compatible tracks. The *automata* technique [16] [4] consists in creating a weighted graph representing the connectivity of every hit, and traversing it to find the best tracks.

We present *Search by triplet*, a fast local method optimized for Velo track reconstruction on parallel architectures. We sort hits in all modules and define tight search windows. We adapt the track forwarding technique to expose parallelization

[1] An *event* corresponds to a single crossing of the Large Hadron Collider proton beams.

with an iterative two-step tracking. We iterate over each detector module only once, maximizing temporal and spatial locality. We flag hits while maintaining parallelizability of each individual step, avoiding Read-After-Write (RAW) data dependencies. We employ a least-squares fit for track fitting, given the expected tracks in the Velo region are straight lines due to the lack of magnetic field interaction. We use *Monte Carlo* simulation of LHCb particle collisions. This allows us to validate our algorithm by comparing trajectories generated by the simulation, also referred to as *true particle trajectories*, against the reconstructed tracks obtained as output of our algorithm.

We develop our algorithm using the SIMT programming model [17], targeting GPGPUs. In order to efficiently use the resources available on GPUs, we create a software framework for performing data parallel event reconstruction. We employ a dynamic GPU memory manager to handle algorithmic data requirements, which allocates and frees GPU memory segments based on a data dependency tree. Our framework can run several GPU *streams* in parallel. We hide the latency of data transmissions by employing a pipeline that reconstructs events while performing memory read and write operations.

We translate our algorithm to the SPMD programming model [18], producing a vectorized algorithm suitable for CPUs. We discuss the design of our algorithm and assess its performance and scalability on modern CPUs and GPUs. We run our software in several streams and study how many concurrent streams are required for saturating our GPUs.

Our work will directly impact the decision on what hardware to acquire for the upcoming upgrade of the processing farm of LHCb. The developed GPU framework is extensible and allows for other parts of the reconstruction to be implemented and evaluated on many-core architectures.

II. Velo reconstruction

The upgraded Velo detector will be a pixel-based particle detector [7], spanning a total of 52 detector modules. A schematic of the detector is shown in Figure 1. The detector modules are placed in two sides, with 26 modules on each side. The interaction region marks where the collisions are expected to occur. The nominal acceptance angle of the LHCb detector is $15-300\ mrad$ in the forward region. The Velo detector will detect by design all particles produced in *primary vertices*[2] in the LHCb coverage angle on at least 3 modules [19].

In the Velo region, the effect of the LHCb magnet is negligible. Particle tracks detected in the Velo detector are therefore expected to be straight lines. Reconstructed Velo tracks serve as seeds for reconstructing particle trajectories through the other LHCb tracking detectors, and allow the reconstruction of vertices where the collisions happened. Additionally, Velo reconstruction occurs early in the LHCb reconstruction process. Therefore, the Velo reconstruction is of paramount importance towards a successful trigger.

[2]A primary vertex is the reconstructed location of an individual particle collision.

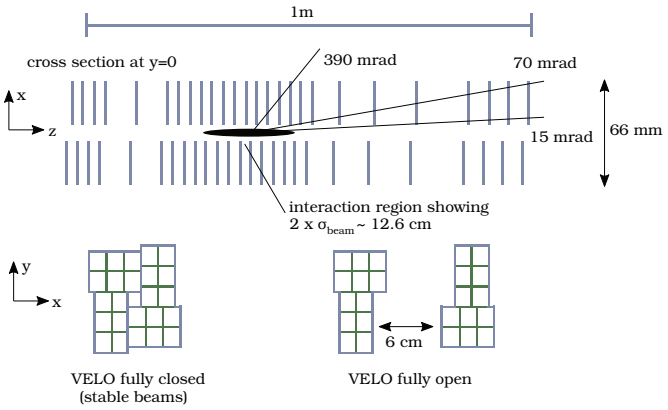

Figure 1: A schematic of the upgraded Velo detector. The top image shows a section in the XZ plane, with detector modules laying in two sides. The images at the bottom show a frontal view of each module in the XY plane, with subdivisions indicating detector chips. Each detector chip has a resolution of 256×256 pixels.

The physics quality of found tracks can be evaluated according to three indicators [20]. Particles are considered *reconstructible* in the Velo subdetector if at least three hits were left in different modules on its path.

- The *track reconstruction efficiency* is the probability to reconstruct a particle travelling through the detector, and can be determined by the ratio between the reconstructed tracks of reconstructible particles, over all the reconstructible particles:

$$\frac{N_{\text{reconstructed and reconstructible}}}{N_{\text{reconstructible}}} \tag{1}$$

- The *fake track fraction* is the ratio between the reconstructed tracks that are not associated to any Monte Carlo particle (*fake tracks*), and all the reconstructed tracks:

$$\frac{N_{\text{fake tracks}}}{N_{\text{reconstructed tracks}}} \tag{2}$$

- Finally, the *clone track fraction* refers to the fraction of tracks associated to the same Monte Carlo particle as another reconstructed track:

$$\frac{N_{\text{clone tracks}}}{N_{\text{reconstructed tracks}}} \tag{3}$$

In spite of the simplicity of Velo trajectories, Velo track reconstruction should maximize reconstruction efficiency, minimize fake fraction and clone fraction at a rate of up to 10^9 tracks per second. The Velo reconstruction algorithm is one of the main time contributors in the current first stage of software trigger [3], also referred to as *High Level Trigger 1*, and therefore it would have a high theoretical speedup if it were parallelized according to Amdahl's law [21].

A. Sequential algorithm

Track forwarding is a local method consisting in finding track candidates and forwarding them over the rest of detector modules. The nominal LHCb algorithm [8] finds a candidate pair of hits fulfilling a *compatibility condition* in neighbouring modules on the same side. Then, the forwarding phase consists in extrapolating the candidate's trajectory to subsequent modules, finding hits that fulfill an *extrapolation condition*. Tracks are forwarded until either no modules remain, or no hits fulfilling the extrapolation condition are found on two consecutive modules on the same side. Hits are flagged upon finding tracks of 4 or more hits, so they are not considered for other tracks. The process is repeated until no candidates remain.

Additional design decisions specific to the Velo detector have been taken in the sequential algorithm. Tracks are required to consist of at least three hits. Three-hit tracks are required to have no flagged hits and to pass a fit cut, since they could potentially be formed out of noise. This is less likely on tracks with more hits, as each additional track hit has to fulfill the extrapolation condition.

A number of modules can be missed in the forwarding phase. This stems from a physical condition: A particle may not leave a signal on a module in its path. The probability of a track missing a signal in a module while having left signals in the precedent and posterior modules is under 1%. However, the probability of a track missing two consecutive modules on the same side is under 0.01%. Therefore, the sequential algorithm allows for a missing module on the last signal side.

The sequential algorithm has been validated to deliver the required physics performance. However, in our opinion there are some fundamental design shortcomings. It should be noted that the solution found by the algorithm is deterministic, although it depends on the order in which hits are considered. Hits are sorted prior to the reconstruction taking place, and the order must be strictly followed for the results to be reproducible. Additionally, hits are required not to be flagged before checking the compatibility or extrapolation conditions. These two facts are implicit RAW dependencies, and make parallelization in the algorithm unfeasible without blocking conditions.

III. Search by triplet

We propose a data parallel approach to Velo reconstruction. Events are physically independent, and can be reconstructed in parallel. Within an event, several tracks can be reconstructed in parallel. Also, events are sufficiently small that they are amenable to be processed by relatively small kernels, avoiding register spilling.

The *Search by triplet* algorithm is composed of five sub-algorithms that are described independently. For all complexity considerations, we generalize our algorithm to m consecutive detector modules, and an average number of hits in each module n.

Sort by phi

Given a list of module hits as input, no assumption can be made as to the order of hits inside each module. This algorithm sorts each of the module hit sets increasingly according to φ, calculated as the *2-argument arctangent* for each hit with respect to the origin of coordinates. Given the expected number of hits is small, a method employing *shared memory*[3] is used for storing the newly calculated φ and finding the sort permutation. The permutation is then applied to hit coordinates, yielding sorted *Structure of Arrays* for each module. A parallel insertion sort method has been implemented for calculating the permutation. The complexity of this algorithm is $O(m \cdot n^2)$.

Find candidate windows

In order to minimize the amount of candidates considered in subsequent steps, the first and last hits in the region of acceptance in the preceding and following modules are calculated for every hit. Figure 2 depicts this process. Hit c_0 would have one candidate on both the preceding and following modules, whereas c_1 would have one and two respectively. This process is repeated for every hit in every module that has a preceding and following module. All modules are processed in parallel. In order to find the first and last candidate, a binary search in φ is performed. The complexity of this algorithm is therefore $O(m \cdot n \cdot log(n))$.

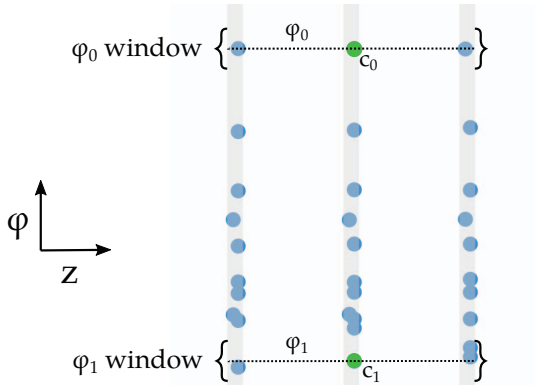

Figure 2: Three consecutive modules with hits are depicted. For hits c_0 and c_1, their respective φ angles and opening windows in the preceding and following modules are highlighted. c_0 has a compatible hit in the preceding module and another one in the following module, on the left and right respectively. c_1 by contrast has one hit in the preceding module and two in the following module.

Track seeding and track forwarding

The *track seeding* algorithm operates on three consecutive modules at a time. It assigns *threads*[4] to hits in the middle

[3]In our GPU implementation, the configurable L1-cache shared memory is employed, due to its low latency and high throughput. In our CPU implementation, main memory is employed.

[4]The CUDA terminology *thread* and *block* is employed here. Equivalently for the CPU implementation, *program instance* and *gang* [18].

module, and each of these threads checks the preceding and following modules for compatible hits. The previously calculated φ windows are employed to this end. For every hit in the middle module, all triplets in the search window are fitted and compared, and the best one is kept as a *track seed*. If there are no hits in either of the search windows, or the least-squares fit χ^2 is over a certain compatibility threshold, no track is formed for that hit.

The multiplicity of triplets to be analyzed varies from hit to hit. A variable workload has a negative impact on performance in parallel architectures, as threads in a block would become idle until all workloads are finished. For this reason, multiple threads can be assigned to process the same hit. In this fashion, if there is one hit with a very high workload, its performance impact is diminished as it will be processed in parallel. The amount of threads assigned to each hit is configurable in our algorithm. Additionally, in cases where the number of hits is under a certain threshold, threads are dynamically reassigned to process one of the hits left, minimizing idle threads.

Since several threads may process the same hit, a synchronization mechanism is required in order to guarantee that only the best triplet for every one middle hit is kept as a track seed. This synchronization mechanism utilizes shared memory, where every thread stores its best found triplet, alongside its fit χ^2. Once all threads have computed their assigned triplets, the χ^2 values assigned to the same middle hit are compared, and only the best fits for each middle hit are kept. After all found triplets have been checked, threads assign to the next hit.

This tiled processing mechanism for finding triplets is applied in first instance to the modules that are further apart from the collision point, as they present the lowest hit multiplicity. This algorithm yields a deterministic solution, that is, the obtained set of triplets is independent of the order in which hits are processed. Each triplet is the *seed* of a forming track, and in the forwarding phase we will try to extend them by looking for hits on the following modules.

Track forwarding operates on forming tracks and *forwards* them to a specified module. Threads are assigned to forming tracks. For every track, the segment defined by its last two hits is extrapolated to the working module. Then, a binary search is performed in φ in the module. The extrapolated segment is checked against the hits as a function of their distance in the module (dx, dy) and the distance along the beam axis from the last hit to the current one (dz): $\frac{dx^2 + dy^2}{dz^2}$. The hit that minimizes the extrapolation function and is under a certain threshold is then appended to the forming track, which is kept for a posterior track forwarding step. A configurable number of modules with no compatible hits are allowed when forming a track. If this number is exceeded, three-hit tracks are stored in a *weak tracks* container for posterior consideration, and tracks with four or more hits are stored in the final tracks container.

When a compatible hit is found, track forwarding *flags* all hits of the forming track. The flag can then be used in the track seeding algorithm, imposing the condition that all hits in a track seed be unflagged. Flags are populated in track forwarding, and are read in track seeding. Therefore, this imposes a Read-After-Write dependency between forwarding and seeding, and the requirement of inter-algorithm synchronization.

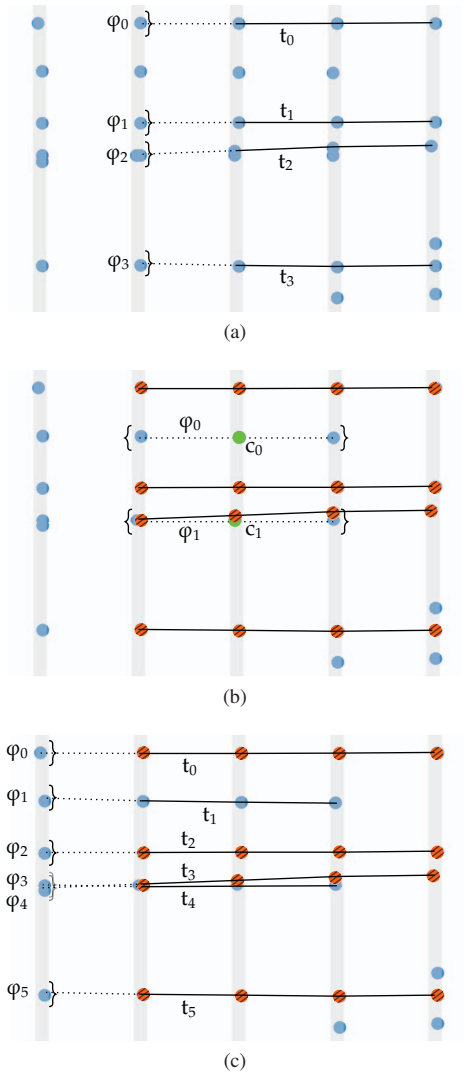

Figure 3: Three processing stages of *Search by triplet* are depicted. (a) *Track forwarding* operates on the second module from the left. For each track t_i, the segment given from its last two hits is extrapolated to the processing module, and φ_i is calculated. A search window is opened and the hit within this window that minimizes the extrapolation function is chosen. If a compatible hit is found, all hits in the track are flagged. (b) *Track seeding* operates in the middle module. The highlighted hits c_i are considered for creating new seeds. Flagged hits are ignored. (c) *Track forwarding* in the leftmost module. All forming tracks are considered for the search. Since tracks t_3 and t_4 present overlapping search windows, they may be extended with the same hit.

Track seeding and track forwarding are the building blocks of our tracking algorithm. Figure 3 depicts five consecutive modules being processed. A track seeding stage (b) is interleaved between track forwarding stages (a) and (c). This mechanism benefits from temporal and spatial locality, since the data-flow is such that module hits are revisited after every forwarding stage. The module processed in the track forwarding stage in Figure 3a is revisited in the subsequent track seeding stage in Figure 3b. This control-flow is compatible with our flagging mechanism, and guarantees flags be populated prior to seeding stages.

Both seeding and forwarding exploit intra-event parallelism, and several independent events are assigned to independent blocks, for inter-event parallelism. The worst-case complexity of track seeding is $O(m \cdot n^3)$. Track forwarding performs a binary search on every module, and the maximum number of tracks created is bound by $m \cdot n$. Therefore, its worst-case complexity is $O(m^2 \cdot n \cdot log(n))$.

Weak track filter

The *weak track filter* algorithm operates on three-hit tracks, and appends them to the final tracks container given that two conditions are met: (1) all three hits must not be flagged, and (2) the least-squares fit χ^2 of the track must be under a certain threshold.

Additionally, a least-squares fit is calculated for every accepted track, required for subsequent reconstruction algorithms, and stored in an SOA container. The complexity of the weak track filter is $O(m^2 \cdot n)$.

IV. GPU SEQUENCE FRAMEWORK

We have developed an extensible GPU sequence framework[5] in order to perform parallel event reconstruction on many-core architectures. Our framework utilizes CUDA to offload computation to a GPU accelerator. We present here the results of the Velo reconstruction, although an evolving codebase is under development in order to accommodate the entire first stage of the software trigger High Level Trigger 1.

Figure 4 depicts an architectural view of the framework. Our framework reads simulated Monte Carlo events from input binary files, which have been generated in the LHCb reconstruction framework. Geometry descriptions of the detector are also read in this fashion, and are kept constant throughout the execution of the reconstruction sequence.

Control flow

Our framework is multi-threaded. Each of the CPU threads employs one GPU stream to guarantee asynchronous execution of events. A configurable number of events is executed in parallel on every GPU stream. Since every event is physically independent, no communication is required between CPU threads or GPU streams.

The reconstruction of physics events is performed in a sequence of algorithms executed on one CPU thread - GPU stream pair. This sequence is configurable, and consists in device-to-host and host-to-device data transmissions, as well as data decoding and reconstruction algorithms. In order to prevent execution stalls, all data transmissions are invoked through their GPU stream. A pipeline is effectively created when three or more *thread-stream* pairs are created, allowing for concurrent two-way transmission and execution.

[5]The GPU sequence framework and Search by triplet are available under https://gitlab.cern.ch/dcampora/search_by_triplet, tag v1.0.

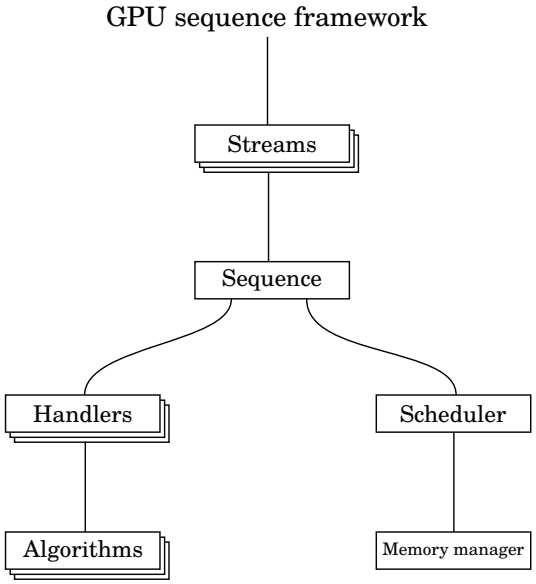

Figure 4: A schematic of the GPU sequence framework. Our software reads binary input files containing simulated Monte Carlo events. Several GPU *streams* can be executed in parallel, each of them with their own *sequence* of algorithms. The memory required by every algorithm in the sequence is managed by the *scheduler*, which employs a *memory manager* with a predetermined memory availability.

The sequence operates through *handlers* that encapsulate algorithms. A handler provides a common façade to an arbitrarily complex control-flow. Code repetition is avoided by encapsulating common tools behind handlers for algorithms such as *prefix sum* or *sorting*, that would otherwise require explicitly instantiating various algorithms.

Data flow

In CUDA, dynamic memory allocation operations such as *cudaMalloc* or *cudaFree* cannot be executed asynchronously, and require all streams to synchronize. The data flow has been developed to solve this central issue. We configure the amount of data to be reserved for every thread-stream pair and allocate it prior to launching the thread. An upper bound for the entire algorithm sequence is therefore necessary, and is currently obtained experimentally.

We have developed a *memory manager* that operates with the allocated memory of the thread-stream. It keeps a view of the memory in segments, and provides non-blocking *malloc* and *free* implementations.

Data dependencies are known a priori for each algorithm. Upon configuring the sequence, the *scheduler* iterates the dependencies and determines when arguments need to be allocated or freed. Prior to the execution of every algorithm in the sequence, the scheduler is invoked in order to prepare the required arguments. The scheduler employs an instance of the memory manager to achieve asynchronous memory management.

V. CPU IMPLEMENTATION

We have translated our code to the *Single Program Multiple Data* (SPMD) format employed by the *Intel SPMD Program Compiler* [18] [6]. This method allows our algorithm to be executed on any available ISPC target CPU [7], while preserving our algorithm design with minimal modifications, yielding the exact same result as the GPU counterpart.

The resulting SPMD code is vectorized by the ISPC compiler. The execution model of ISPC executes a *gang of program instances* in parallel, using the vector units available in a processor. The execution of every instruction is masked, similarly to how a *warp* executes threads on a GPU. ISPC allows compilation with a configurable execution mask size and gang size. Additionally, the desired set of *vector extensions*[8] can be configured.

Our CPU implementation is compatible with the Monte Carlo events and geometry descriptions of the GPU sequence framework. We have predefined the Velo sequence, with the same set of algorithms as the many-core model. Events are executed in parallel across different CPU threads via a minimal multi-threading wrapper, while intra-event parallelism is handled by ISPC assigning work to vector units. In order to be able to rigorously compare both implementations, we have avoided any usage of the C++ standard library for common algorithms.

VI. PERFORMANCE ANALYSIS

We have carried out a performance analysis over a variety of hardware, described in tables I and II. The CPUs under analysis are from two different vendors, Intel and AMD. The Skylake processor *Silver 4114* supports the AVX512 instruction set, whereas the Broadwell and EPYC processor only support AVX2. A dual-socket configuration for each server has been tested, with two identical processors of each kind.

The GPUs have different memory types, gaming cards have GDDR5 whereas the scientific card Tesla V100 is equipped with High Bandwidth Memory (HBM2). The *10-series* gaming cards implement the NVIDIA Pascal architecture, the scientific card implements the Volta architecture, and the *RTX 2080 Ti* implements the more recent Turing architecture. The CUDA compute capability of either of the cards is enough to support our implementation of Search by triplet. The memory of the cards impacts the amount of streams and events that can be executed concurrently.

Feature	Intel Xeon Broadwell E5-2630	Intel Xeon Silver 4114	AMD EPYC 7301
# cores	20	20	16
Max freq.	3.1 GHz	3.0 GHz	2.7 GHz
Cache (L3)	25 MB	13.75 MiB	64 MiB
DRAM	64 GiB	64 GiB	64 GiB
SIMD capability	AVX2	AVX512	AVX2
MSRP	667 $	694 $	948 $

Table I: CPU hardware used for our tests. We compare a Broadwell processor (Intel Xeon E5-2630), a Skylake processor (Intel Xeon Silver 4114) and an AMD processor.

Feature	Geforce GTX 1060	Geforce GTX 1080 Ti	Geforce RTX 2080 Ti	Tesla V100
# cores	1280 (CUDA)	3584 (CUDA)	4352 (CUDA)	5120 (CUDA)
Max freq.	1.81 GHz	1.67 GHz	1.545 GHz	1.37 GHz
Cache (L2)	1.5 MiB	2.75 MiB	6 MiB	6 MiB
DRAM	5.94 GiB GDDR5	10.92 GiB GDDR5	10.92 GiB GDDR5	32 GiB HBM2
CUDA capability	6.1	6.1	7.5	7.0
MSRP	249 $	699 $	1199 $	8899 $

Table II: GPU hardware used for our tests. We compare a mid-class gaming graphics card (Geforce GTX 1060), two high-end gaming graphics card (Geforce GTX 1080 Ti and Geforce RTX 2080 Ti) and a scientific card (Tesla V100).

We employ a validation method based on well-established metrics for our algorithm (cref. section II). We obtain a deterministic result across all devices. The use of Monte Carlo data for validation is the standard for validating reconstruction algorithms. The presented results have been validated to produce acceptable physics performance.

We run a configurable number of events s for a number of repetitions r. In each repetition, event data submission and retrieval are performed. The amount of streams t is also configurable. We measure the performance of our sequence by using external counters. We obtain the *wall clock* execution time, and factor in the number of events that have been processed. Our framework presents the performance of a run as the number of events executed per unit of time, measured as frequency (Hz).

Search by triplet presents several free parameters that alter the computing performance. Each of the discussed algorithms are encapsulated in one CUDA kernel, and can be tweaked with respect to the number of blocks and number of threads on each invocation. We have identified, by using local search, the parameter values that provide best performance for the entire sequence, and the resulting configuration is shown in table III. Even though individual kernels may be faster under other configurations, these values empirically showed

[6]Search by triplet SPMD is available under https://gitlab.cern.ch/dcampora/search_by_triplet_spmd, tag v1.0.

[7]At the time of writing, ISPC supports as targets: x86 with SSE2, x86-64, ARM and NVIDIA PTX.

[8]The following vector extensions were tested: SSE2, SSE4, AVX, AVX2, AVX512 (Skylake).

the best performance-to-resource-usage ratio, resulting in a more efficient CUDA scheduler resource assignment. We have found this configuration to provide best performance across all tested devices.

Kernel	# blocks	# threads
sort by phi	# events in execution	64
find candidate windows	{# events in execution, # Velo middle modules}	128
track seeding and track forwarding	# events in execution	32
weak track filter	# events in execution	256

Table III: Best configuration found in local search for each CUDA kernel. We have optimized our configurations minimizing the overall wall clock execution time. Individual algorithms may get faster with different configurations, but the effect on the overall performance is also impacted by resource usage, since other concurrent streams may be blocked.

The configuration of *Search by triplet SPMD* has also been tweaked for each of the CPUs under consideration. A mask of 32 bits was found to yield the best performance for all processors. This is to be expected, as the ISPC guidelines state the mask should have a length of the most used datatypes, which are 32-bit types in our algorithm. The gang size and vector extension has also been tested, and table IV depicts the optimal configurations found for each processor. In the AMD processor, the preferred vector extension and gang size were AVX1 and four, in contrast with the Intel Broadwell processor, which could be due to the differing number of ports and functional units available on both processors.

Processor	Vector extension	Mask size	Gang size
Intel Xeon Broadwell E5-2630	AVX2	32	8
Intel Xeon Silver 4114	AVX512	32	16
AMD EPYC 7301	AVX1	32	4

Table IV: Best configuration for Search by triplet SPMD for each processor. Both Intel processors benefit from their latest available instruction set. The AMD processor benefits from an AVX1 configuration with a gang size of 4, despite supporting AVX2. This could be due to particularities involving the number of ports and functional units of the processor.

The peak performance configuration achieved with every processor is compared in Figure 5. The AMD EPYC processor underperforms when compared to its other CPU competitors. The AVX512 vector extensions in the Skylake processor show a discrete 6% performance speedup over the AVX2 Broadwell processor. Even though our CPU solution is vectorized and utilizes all available threads, all of the tested high-end and scientific GPUs outperform the CPUs in consideration.

The mid-class Geforce GTX 1060 yields a similar performance to the Intel processors under analysis. The projected speedup between the Geforce GTX 1060 and the

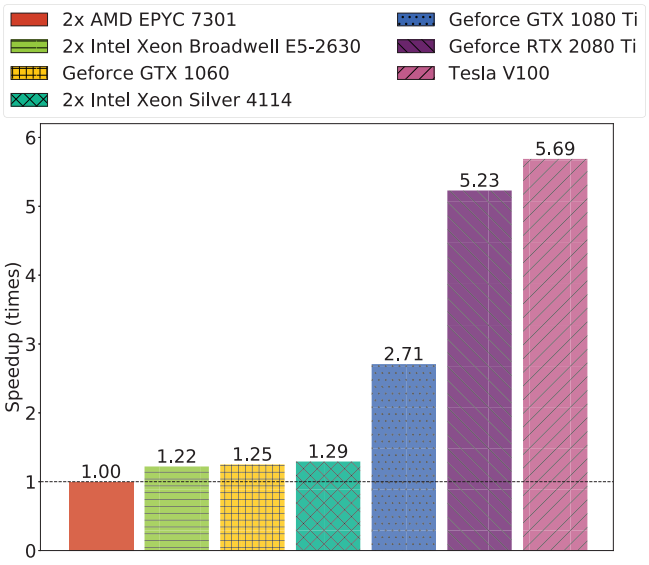

Figure 5: Speedup between the three CPU and the four GPUs under consideration. For each CPU, the performance of a dual-socket server, with the *Search by triplet SPMD* algorithm with their best ISPC configuration is shown. For each GPU, the performance of *Search by triplet* within the GPU sequence framework with their best parameter configuration is shown. CPUs underperform compared to GPUs. The performance scales to higher-end GPU devices.

Geforce GTX 1080 Ti according to their number of cores and maximum frequency is $2.58\times$, even though this does not take into consideration cache size or base frequency. We observe a speedup of $2.41\times$, showing our algorithm scales to higher-end architectures. We attribute the difference in performance across the two high-end gaming cards Geforce GTX 1080 Ti and Geforce RTX 2080 Ti to be a combined effect of both the increase in CUDA cores and in L2 cache, since we observe a $1.93\times$ speedup between them. The scientific card tops our speedup chart showing only a 9% speedup over the Geforce RTX 2080 Ti, despite being an older architecture. When factoring in the MSRP of the devices under consideration, the mid-class Geforce GTX 1060 becomes the graphics card that delivers the best price-performance ratio. The scientific card Tesla V100 delivers a worse price-performance than the gaming cards, due to its high MSRP.

In order to understand the impact of our work in the field, we can compare the performance obtained with the current *LHCb baseline* implementation [22]. Our SPMD implementation presents a speedup of $1.46\times$ over the LHCb baseline, under the same hardware configuration of a dual-socket Intel Xeon Broadwell E5-2630. The Geforce RTX 2080 Ti presents a speedup of $6.23\times$, and the Tesla V100 a speedup of $6.77\times$ when compared to the baseline results. We acknowledge the physics quality of the results are not identical between the baseline and our implementation, and that the

LHCb codebase is in active development and its performance has improved since. Nevertheless, we attribute the presented speedup to the combined impact of data structures, locality and vectorization of our algorithm design.

Figure 6 shows a breakdown of the contribution of each algorithm to the overall timing of the Velo track reconstruction. We observe our sequence is dominated by track seeding and track forwarding, as was to be expected from the computational complexity analysis. The weak track filter time fraction is negligible, since it operates in a small subset of leftover 3-hit tracks.

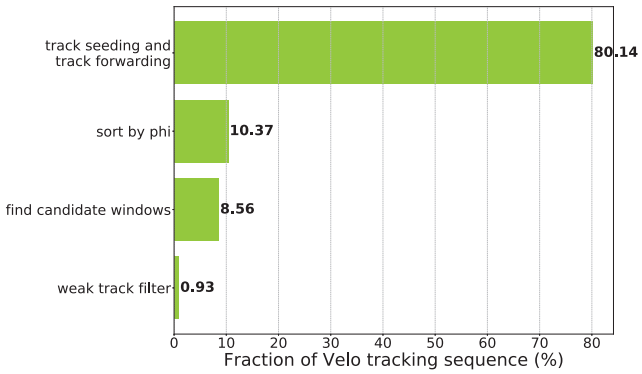

Figure 6: The contribution of each algorithm to the timing of the sequence is shown. Track seeding and track forwarding compose a single *kernel* and are therefore shown together. The track reconstruction sequence is dominated by track seeding and track forwarding, as would be expected from the complexity analysis.

A. GPU sequence results

A percentual comparison of profiled sequence execution and memory transmission data is shown in Figure 7. The sequence execution dominates the time distribution of the GPU. Given that we have created an effective asynchronous pipeline, memory submissions and memory retrievals are hidden behind the execution time of our sequence.

Figure 8 depicts two parameter scans for number of events s and number of streams t, respectively. A configuration of $s = 4096$, $t = 3$ turns out to be effective on all tested hardware. The Geforce GTX 1060 only requires two streams to achieve an effective pipeline. We attribute this to the lower amount of *streaming multiprocessors* on that device, which permits achieving a high occupancy with one stream, hiding the transmissions on the other concurrent stream. A higher number of streams does not increase the throughput. This fact, together with the scaled performance to high-end devices, indicate our software is *compute bound*.

VII. Conclusion

We have presented *Search by triplet*, a new algorithm to efficiently perform track reconstruction on parallel architectures. Our algorithm takes inspiration from track forwarding

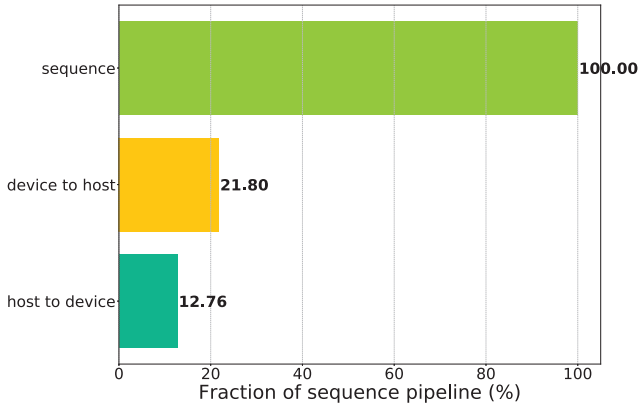

Figure 7: Pipeline of Velo tracking sequence in the GPU sequence framework. The timings of the pipeline were obtained by the *nvprof* command in a full sequence execution. The pipeline is dominated by code execution by a 78.20% margin. The transmissions will be hidden if enough streams are running asynchronously.

techniques. We have designed our algorithm removing RAW dependencies and revisiting detector modules in subsequent steps to maximize temporal and spatial locality. We have discussed worst-case complexity for each of its constituent parts. We have developed our algorithm in CUDA and we have optimized the launch parameter configurations. The algorithm has been validated against Monte Carlo simulated data.

We have presented *Search by triplet SPMD*, an SPMD realization of our algorithm geared towards parallel SIMD processors. We have carried over the design of our algorithm to CPUs, and we have optimized our compilation options for each of the processors under consideration.

We have compared the performance of our algorithm across a variety of parallel architectures. Our algorithm benefits from larger vector widths on Intel processors, and scales to high-end GPU architectures. The algorithm performs the Velo track reconstruction with a throughput of 57.36 kHz (AMD EPYC 7301) through 74.17 kHz (Intel Xeon Silver 4114) on CPUs, and 71.75 kHz (Geforce GTX 1060) through 326 kHz (Tesla V100) on the GPUs under consideration.

We have assessed the impact of our algorithm design decisions by comparing the performance of our SPMD implementation with the LHCb baseline implementation. We obtain a $1.46\times$ speedup with respect to the baseline implementation running on the same hardware. We acknowledge this codebase is in active development, and a dedicated study comparing track reconstruction approaches should be pursued.

We have also presented a new framework to perform physics reconstruction on many-core architectures *GPU sequence framework*. We have encapsulated our software into this framework. We have performed a parameter scan over the configurable number of events and streams of our application. An effective pipeline has been created under all studied devices

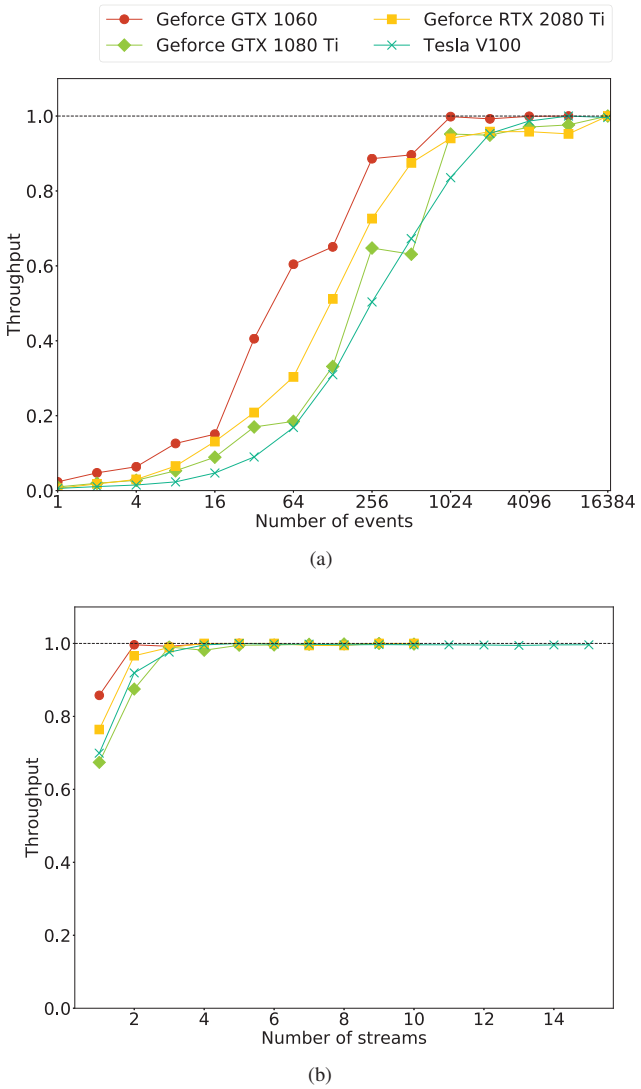

Figure 8: Two parameter scans are shown for the GPU sequence framework application. (a) The number of events parameter is scanned for all devices under consideration. The configuration used throughout all measurements is $t = 3$ and $r = 200$. Performance caps with 1024 events for Geforce cards and 2048 events for the Tesla card. (b) A scan of the number of streams is depicted, with configuration $s = 4096$ and $r = 200$. The Geforce GTX 1060 requires only 2 streams to achieve an effective pipeline, in contrast with the 3 streams required by the other cards. The memory capacity of each device limits the maximum number of concurrent streams under the tested configuration. The peak performances for the Geforce GTX 1060, Geforce GTX 1080 Ti, Geforce RTX 2080 Ti and Tesla V100 are 71.75, 155.33, 299.94 and 326.26 kHz respectively.

that hides transmission times. We have profiled the algorithms that conform our Velo reconstruction implementation, and we have identified the main time consumers. Our framework employs a custom memory manager to allocate and free memory segments as required in an asynchronous manner.

Our track reconstruction algorithm is an indication that other LHCb subdetectors may be amenable to be reconstructed efficiently on many-core architectures. We have shown a translation of our GPU algorithm performs adequately on CPUs, while maintaining the same SIMD-oriented design choices. We will study the applicability of our design to other subdetector-specific geometries and conditions.

Our framework can be extended with additional reconstruction algorithms. We intend to do a detailed cost-analysis of our application for the upcoming LHCb upgrade. The performance of our application will be a determining factor to adopt GPUs in LHCb's trigger server farm.

Acknowledgment

The authors would like to thank D. vom Bruch for fruitful discussions and code reviews. Thanks to V. Gligorov for his guidance and support, and to P. Fernández Declara for framework discussions. Thanks to C. Potterat and M. Rangel for early discussions on the development of the first prototype of Search by triplet. Thanks to D. Rohr for ideas for obtaining better performance, and to R. Quagliani for continuous discussions to improve the physics performance. We would also like to thank the LHCb computing and simulation teams for their support and for producing the simulated LHCb samples used to develop and benchmark our algorithm.

REFERENCES

[1] The LHCb Collaboration, "LHCb detector performance," *International Journal of Modern Physics A*, vol. 30, no. 07, p. 1530022, mar 2015. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0217751X15300227

[2] ——, "Framework TDR for the LHCb Upgrade: Technical Design Report," Tech. Rep. CERN-LHCC-2012-007. LHCb-TDR-12, Apr. 2012. [Online]. Available: https://cds.cern.ch/record/1443882

[3] ——, "LHCb Trigger and Online Upgrade Technical Design Report," Tech. Rep. CERN-LHCC-2014-016. LHCB-TDR-016, May 2014. [Online]. Available: https://cds.cern.ch/record/1701361

[4] D. Rohr, S. Gorbunov, and V. Lindenstruth, "GPU-accelerated track reconstruction in the ALICE High Level Trigger," *J. Phys. Conf. Ser.*, vol. 898, no. 3, p. 032030, 2017.

[5] P. Sen and V. Singhal, "Event selection for much of cbm experiment using gpu computing," in *2015 Annual IEEE India Conference (INDICON)*, Dec 2015, pp. 1–5.

[6] D. vom Bruch, "Online Data Reduction using Track and Vertex Reconstruction on GPUs for the Mu3e Experiment," *EPJ Web of Confernces*, vol. 150, no. 00013, 2017. [Online]. Available: https://www.epj-conferences.org/articles/epjconf/pdf/2017/19/epjconf_ctdw2017_00013.pdf

[7] The LHCb Collaboration, "LHCb VELO Upgrade Technical Design Report," Tech. Rep. CERN-LHCC-2013-021. LHCB-TDR-013, Nov 2013. [Online]. Available: http://cds.cern.ch/record/1624070

[8] O. Callot, "FastVelo, a fast and efficient pattern recognition package for the Velo," CERN, Geneva, Tech. Rep. LHCb-PUB-2011-001. CERN-LHCb-PUB-2011-001, Jan 2011, lHCb. [Online]. Available: http://cds.cern.ch/record/1322644

[9] D. Funke, T. Hauth, V. Innocente, G. Quast, P. Sanders, and D. Schieferdecker, "Parallel track reconstruction in CMS using the cellular automaton approach," *Journal of Physics: Conference Series*, vol. 513, no. 5, p. 052010, jun 2014. [Online]. Available: http://stacks.iop.org/1742-6596/513/i=5/a=052010?key=crossref.85cff4ebb76ffe912b706a3d23b5f608

[10] R. E. Kálmán, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, Mar. 1960. [Online]. Available: http://dx.doi.org/10.1115/1.3662552

[11] D. H. Cámpora Pérez and O. Awile, "An efficient low rank kalman filter for modern simd architectures," *Concurrency and Computation: Practice and Experience*, vol. e4483, 2018.

[12] R. H. C. Lopes, I. D. Reid, and P. R. Hobson, "A well-separated pairs decomposition algorithm for k-d trees implemented on multi-core architectures," *Journal of Physics: Conference Series*, vol. 513, no. 5, p. 052011, jun 2014. [Online]. Available: https://doi.org/10.1088%2F1742-6596%2F513%2F5%2F052011

[13] C. Cheshkov, "Fast hough-transform track reconstruction for the alice tpc," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 566, no. 1, pp. 35 – 39, 2006, tIME 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900206008059

[14] L.-B. Niu, Y.-L. Li, M. Huang, B. He, and Y.-J. Li, "Track reconstruction based on Hough-transform for nTPC," *Chinese Physics C*, vol. 38, no. 12, p. 126201, dec 2014. [Online]. Available: http://stacks.iop.org/1674-1137/38/i=12/a=126201?key=crossref.04ab3117f629b4f1118b49f222f1d94c

[15] A. Abba, F. Bedeschi, F. Caponio, R. Cenci, M. Citterio, A. Cusimano, J. Fu, A. Geraci, M. Grizzuti, N. Lusardi, P. Marino, M. Morello, N. Neri, D. Ninci, M. Petruzzo, A. Piucci, G. Punzi, L. Ristori, F. Spinella, S. Stracka, D. Tonelli, and J. Walsh, "An "artificial retina" processor for track reconstruction at the full lhc crossing rate," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 824, pp. 260 – 262, 2016, frontier Detectors for Frontier Physics: Proceedings of the 13th Pisa Meeting on Advanced Detectors. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900215012607

[16] A. Glazov, I. Kisel, E. Konotopskaya, and G. Ososkov, "Filtering tracks in discrete detectors using a cellular automaton," *Nucl. Instr. and Meth.*, vol. A329, pp. 262–268, 1993.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[18] Intel, "Intel spmd program compiler." [Online]. Available: https://ispc.github.io/

[19] R. Aaij *et al.*, "Performance of the LHCb Vertex Locator. Performance of the LHCb Vertex Locator," *JINST*, vol. 9, no. CERN-LHCB-DP-2014-001. CERN-LHCB-DP-2014-001. LHCB-DP-2014-001, p. P09007. 61 p, May 2014, comments: 61 pages, 33 figures. [Online]. Available: http://cds.cern.ch/record/1707015

[20] M. Schiller, "Track reconstruction and prompt k^0_S production at the LHCb experiment," Dissertation,, University of Heidelberg,, 2011.

[21] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[22] M. De Cian, A. Dziurda, V. Gligorov, C. Hasse, W. Hulsbergen, T. E. Latham, S. Ponce, R. Quagliani, H. F. Schreiner, S. B. Stemmle, J. Van Tilburg, M. J. Zdybal, and J. M. Williams, "Status of HLT1 sequence and path towards 30 MHz," CERN, Geneva, Tech. Rep. LHCb-PUB-2018-003. CERN-LHCb-PUB-2018-003, Mar 2018. [Online]. Available: http://cds.cern.ch/record/2309972