GPGPU opportunities for the LHCb trigger

Public Note

Issue: 1
Revision: 0

Reference: LHCb-PUB-2014-034
Created: April 29th, 2014
Last modified: May 22, 2014

Prepared by: Alexey Badalov[a], Daniel Cámpora[b], Gianmaria Collazuol[c], Marco Corvo[d], Stefano Gallorini[c], Alessio Gianelle[c], Elisabet Golobardes[a], Donatella Lucchesi[c], Anna Lupato[c], Niko Neufeld[b], Lorenzo Sestini[c], Rainer Schwemmer[b] Xavier Vilasís-Cardona[a]

[a]La Salle-URL, Spain
[b]CERN,Switzerland
[c]INFN-Padova,Italy
[d]Università di Ferrara, Italy

LHCb-PUB-2014-034
26/05/2014

Abstract

This note describes arguments to study the use general purpose graphic processing units to improve the performance of the LHCb trigger, presents the current developments in the integration into the Gaudi framework and the implementation of algorithms and points towards possible R& D directions.

Document Status Sheet

1. Document Title: GPGPU opportunities for the LHCb trigger			
2. Document Reference Number: LHCb-PUB-2014-034			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
Draft	1	2014-04-16	Outline
Draft	2	2014-05-04	First Compilation of Contributions
Draft	3	2014-05-06	First Full Version
V 1.0	4	2014-05-20	First version with corrections

Contents

1 Introduction 1

2 GPGPU overview 2

3 Key points for the integration 4
- 3.1 Hardware 4
- 3.2 Software 4
- 3.3 Algorithms 5

4 Offloading mechanism 5

5 VELO 6
- 5.1 VELO Pixel 6
- 5.2 Track model 7
- 5.3 Tracking performance indicators 8
  - 5.3.1 Reconstruction efficiency 8
  - 5.3.2 Purity 8
  - 5.3.3 Ghost fraction 8
  - 5.3.4 Clone fraction 9
- 5.4 FastVelo 9
- 5.5 FastVELO GPU implementation 9
  - 5.5.1 Implementation details 9
  - 5.5.2 Preliminary results 11
- 5.6 VELO Pixel implementation 15
  - 5.6.1 Sequential findByPairs 15

5.7 A GPU implementation for VELO Pixel . . . 16
5.8 Algorithm design . . . 16
5.8.1 Track seeding . . . 16
5.8.2 Track formation . . . 16
5.8.3 Track selection . . . 16
5.8.4 Performance results . . . 17

6 Research Topics . . . 19
6.1 FastVELO . . . 19
6.2 Hardware . . . 19
6.2.1 Monolithic servers . . . 20
6.2.2 External PCIe chassis . . . 20
6.3 Software framework . . . 20
6.4 Algorithms . . . 21

7 References . . . 21

List of Figures

1 A pixel plane captures the X-ray image of the sample, mounted on a rotated platform. 3
2 Execution of multiple Gaudi pipelines concurrently using a classical CPU algorithm vs. offloading to a GPU. . . . 6
3 A sketch of the current VELO detector. . . . 7
4 Schema of the 48 sensors of the pixel VELO subdetector, placed along the Z axis. The subdetector system is divided in two sides, placed in +X and -X, with 24 sensors each. . . . 7
5 Tracking performance comparisons between the sequential FastVelo and FastVelo on GPU. (Left) Tracking efficiency as a function of the true track momentum P_{true}. (Right) Impact parameter resolution as a function of $1/P_{T,true}$. . . . 12
6 Tracking execution time and speedup versus number of events using a 2012 MC sample of $B_s \to \phi\phi$ decays ($\nu = 2.5$). The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz). . . . 13
7 Tracking execution time and speedup versus number of events using a sample of No-Bias data collected during 2012 run ($\mu = 1.6$). The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz). . . . 13
8 Tracking execution time and speedup versus number of events using a 2015 MC sample of b-inclusive decays generated with $\nu = 4.8$. The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz). . . . 14
9 Number of events processed per seconds versus the number of instances of FastVelo tracking running on HLT1. Time measurements are taken with an Intel Xeon E5-2600, 12 cores (24 with hyper-threading). . . . 14
10 Track forwarding example. A track has been created from 17 hits, in the +Y side of the detector. . . . 17
11 Distribution of hits per event for considered datasets. . . . 18
12 GaudiMT design. . . . 21

List of Tables

1 Tracking efficiencies obtained with FastVelo on GPU, compared with the results obtained by original FastVelo code (only the VELO tracking running on HLT1). The efficiencies are computed using 1000 $B_s \to \phi\phi$ MC events, generated with 2012 conditions. 12
2 Stages of the VELO pixel subdetection algorithm. The percentages indicate the relative amount of time each stage takes. 15
3 Average multiplicity of hits in dataset. 15
4 Conditions of datasets used. 18
5 Results for the sequential implementation. A total of 2000 events have been processed, in groups of 500 across each of the above datasets. 18
6 Parallel approach Physics results for the *gpuKalman* stage. 18
7 Parallel approach Physics results for the *gpuKalman* and *gpuPostProcessing* run sequentially one after the other. 18
8 Test setup. 19
9 Algorithms and execution times. 19

1 Introduction

CP violation is one of the necessary conditions to generate a baryon asymmetry in the Universe. Understanding the origin of the CP violation mechanism is a key question in physics. In the SM, CP violation is fully described by the CKM mechanism [1]; this mechanism, while successful in explaining the current experimental data, it is known to generate insufficient CP violation to explain the observed baryon asymmetry in the Universe. Additional sources of CP violation, which are predicted by many extensions of the SM, are needed. New particles beyond the SM may enter loop-mediated processes such as $b \to q$ Flavor Changing Neutral Currents (FCNC) transitions with q=s or d, leading to discrepancies between measurements of CP asymmetries and their SM expectations. b hadron decays in completely hadronic final states provide a rich set of observables that are rather precisely known in the SM but could potentially receive sizable corrections from new heavy particles appearing in the loop.

The presence of physics beyond the SM can also be detected by looking for its contribution to charmless $b \to sqq$ (q = s; d) decays. Particularly interesting is the decay channel $B_s^0 \to \phi\phi$; forbidden at tree level it proceeds via the $b \to ss\bar{s}$ penguin transition where new particles can enter. In this decay channel it is possible to look for CP violation beyond the SM by studying the interference between the mixing and decay of B_s^0 mesons to CP-eigenstates, which is characterized by a CP-violating phase, ϕ_s. In the SM, the value of ϕ_s for this process is expected to be close to zero. In the decay of a pseudoscalar B_s^0 meson to a pair of vector mesons, three different amplitudes arise corresponding to each possible relative orbital angular momentum among the vector mesons. However, the same final state can be reached through decays involving scalar resonances. The total decay amplitude is then a coherent sum of these contributions. A time dependent angular analysis is then needed to disentangle the different amplitudes and measure the weak phase ϕ_s. Since the oscillation in the decay time distribution, which gives sensitivity to the measurement of ϕ_s is exactly opposite for B_s^0 and anti-B_s^0 decays, determination of the initial meson flavor is also required. Given the complexity and the requirements of this measurement, in order to achieve the precision necessary to be sensitive to NP, a large number of events is needed. Currently, the $B_s^0 \to \phi\phi$ decay is selected by the HLT2 either by using a topological trigger or by identifying events containing a K^+K^- pair with an invariant mass close to that of the phi meson. The latter selection has the important feature of avoiding cuts on the impact parameter of the tracks, which would bias the lifetime. In the first step, kinematic constraints such as the transverse momentum of the final state particles are used to reduce the rate by means of a neural network-based selection; this allows running the comparatively slow particle identification algorithm using the RICH sub-detector on the events selected by the first neural network. This information is then passed to the second neural network, which uses both kinematic and particle ID information to

make the final selection. The best possible efficiency is around 50% in the upgraded conditions; a new trigger approach is needed to improve it.

One of the most stringent restrictions upon reconstruction algorithms for the software trigger is their throughput. Data rates require fast execution times, which eventually limit the type of algorithms to be used. One may not count anymore on the fast development of processors to expect a given algorithm to become faster just because the CPU clock frequency increases: clock frequencies are frozen for more than ten years now. The trend has moved towards having several cores, ranging from two to, likely, manycores in the near future. For this reason, we might expect improvements in execution times coming from a clever use of the multicore structure and parallelisation. Therefore, sensibility advises to build up a program to study and exploit the possibilities of parallelisation of the algorithms involved in the reconstruction and also in the trigger. Among the candidate architectures to support these algorithms we find general purpose graphics processing units (GPGPUs). GPUs are specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control so they deliver better performances on algorithms that can be cast into a highly parallel form. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. Any of the reconstruction algorithms fitting this scheme is likely to have an improved performance when executed on a GPGPU. In the literature we find several applications of GPGPUs to high energy physics, but, because of the similarities with our goals, we may single out two : the RICH ring reconstruction for the NA62 experiment [2] and the track reconstruction for ALICE [11]. Upon this grounds, a study has started in LHCb to evaluate the possible role of GPGPUs in the new trigger approach required by the upgrade.

This note depicts the programme being followed for this evaluation. We start by a brief sketch of other GPGPU applications related to high energy physics. Then we explain what are the key feature that need to be taken into account in the framework of the LHCb experiment. We continue by explaining our current solution to offload calculations into the GPU with the LHCb framework. Next we show the current efforts and results to implement tracking algortithms for the LHCb VErtex LOcator (VELO), in its current form and in the form to be used in the upgrade. Finally, we mention some possible future directions of development.

2 GPGPU overview

The architecture of GPGPUs is made of a large number of computing units with very little control structures and very little cache memory in contradistinction to ordinary CPUs which require large control structures and cache memory with fewer, yet larger, arithmetic units. GPUs are designed to fit the requirements of graphics processing to deliver high quality image rendering in real time. Any algorithm which may have similar requirements to image rendering in terms of need to process small units repeatedly is a candidate to run faster on a GPU than on a CPU. One may not forget, however, two key points in order to maximally benefit from the GPU architecture. The first one is the need to offload data to the GPU in an efficient manner: an optimisation of the data throughput is essential to preserve the gain in speed coming from the GPU architecture. The second is the programming language and the optimal algorithm implementation for which vendors offer specific solutions [4]. Regarding High Energy Physics applications, we find different approaches to face problems in a variety of environments.

In reference [5], for instance, we find a proposal for the reconstruction of tomographies in the European Synchrotron Radiation Facility. The synchrotron light, in the X-ray domain, illuminates a sample on a rotating platform. A pixel detector plane behind the sample and perpendicular to the incoming rays takes the image of back projection, as shown in figure 1. This system takes several thousands of images per second, allowing a 3D reconstruction of the inside of the sample.

The authors report an improvement in performance and achieve a near real-time reconstruction speed. They accelerate the original algorithm by shifting most of the computations to the GPU using the CUDA toolkit. The optimized version is able to reconstruct a typical 3D image (a 3 gigavoxel image from 2000 projections) in mere 40 seconds using a GPU server equipped with 4 graphic cards, I/O time excluded. This is approximately 30 times faster, compared to the previously used 8-core Xeon server.

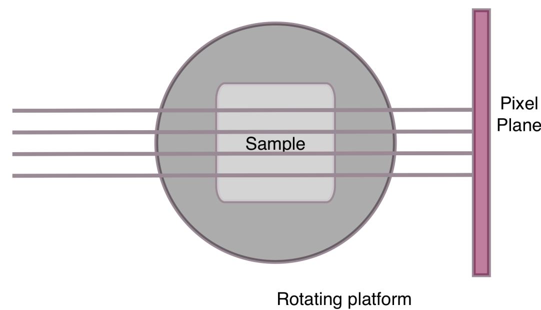

Figure 1 A pixel plane captures the X-ray image of the sample, mounted on a rotated platform.

Reference [6] tests the integration of GPUs into the ITMS data acquisition framework in fusion experiments. The system implements a self-adaptive sampling data acquisition mechanism, that is: acquiring data with a variable sampling rate that is continuously adapted to input signal bandwidth. In this way, the volume of generated data is reduced without losing any information. The GPU is kept in charge of adaptation of the algorithm computation, yielding high performance.

For the nuclear physics facility FAIR [7], reference [8] explains how they moved one particular algorithm in the PANDA detector simulation, the track reconstruction one, to a GPU obtaining faster results.

The applications mentioned above, are tests, off the production systems, showing the convenience of moving some algorithms into a GPU. To finish, we would like to mention two particularly interesting applications at the trigger level, which actually are applied to a real detector. The first one is in the NA62 [9] experiment where a GPU is used to perform signal processing on the data acquired from the RICH to reconstruct Cherenkov rings traces [2, 10] for its identification. The second is the implementation of a cellular automaton algorithm to reconstruct the charged particle traces at the ALICE tracker [11, 12].

However, there is currently only one real production system making use of GPUs, namely the ALICE High Level Trigger [11]. This is a bit special in the sense that contrary to the usual way a trigger works, reducing the amount of data by *selecting* particularly interesting events for permanent storage and discarding others, the ALICE HLT reduces the amount of data by replacing the enormous raw dataset from the main tracking detector (a Time Projection Chamber) by a higher-level reconstructed object, which contains all the essential information, but takes up much less space. It is a very promising application; a multi-threaded CPU-only version they can gain a factor of 5 to 6, and cleverly using cheap gaming graphics cards, rather than the expensive dedicated co-processor cards, we gain another factor of 3 or 4. Note that one of the most important differences between gaming cards and dedicated co-processor cards for high-performance computing (HPC) is hardware reliability, in particular for 24/7 operation. In HPC, where a single computational problem can have a very long running time, a processor crash can lead to significant delays, so frequent check-pointing is used to remedy this. In High-Energy Physics even for complex events the processing time is short and events are statistically completely independent from each other, so a processor crash will only reduce the processing power of the overall system but not impair the functioning of the experiment as a whole. This is why it is possible to use cheap gaming cards. Still, the management effort goes up when one has to cope with intrinsically less reliable hardware. Finally, let us mention that other planned heavy ion facilities expect to follow suit.

3 Key points for the integration

The efficient use of GPGPUs necessarily needs to take into account three essential points : how will it be integrated to the existing hardware, how will it fit into the current software and what are the candidate algorithms which may be offloaded to the GPU. In the case of LHCb, these points are developed in the following. Actually, these boundary conditions are relevant to establish the final performance of the system and may need or imply solutions which would look non-optimal from a purely GPGPU point of view.

3.1 Hardware

Currently, the hardware infrastructure of the LHCb software trigger farm is based on a network of CPU servers optimized for the current HLT workload. The servers are very compact and consist, essentially, of the most cost-efficient CPUs and the necessary amount of memory. Detector data I/O is done via integrated network interfaces. Since these servers are also optimized for high density datacenters, they currently have no physical space or power for installing an accelerator card.

For the test installation, a new set of machines has to be acquired; these will have sufficient space and power to host accelerator cards. Rack space is currently not an issue, so cheaper, low-density solutions can be used. These machines will need to have at least one slot for a double-height, full-length PCIe Gen3 card plus the required power supply. Servers like these are currently available, for example: Supermicro 7047GR-TRF or similar. The machines can then be connected to the current read-out network and receive data samples copied from the nominal data stream.

3.2 Software

The entire software infrastructure at LHCb is built around the Gaudi framework [13], which was devised under the assumption that a large number of powerful CPU cores handled a stream of data, with each core passing a piece of data through a sequence of algorithms. Each Gaudi algorithm takes data from the Transient Event Store (TES), processes it using libraries, services, and tools provided by the framework, and then places the output back into TES. This architecture presents an obstacle to the use of GPUs. The benefits of such hardware are best realized when processing events in large batches, not one by one in independent concurrent pipelines.

An algorithm running as a part of the Gaudi framework, can choose to take advantage of multiple processor cores independently of the framework, but it is limited to processing only a single piece of data at a time. It is possible to run several instances of the framework in parallel. In this case, each instance applies the same algorithms to different data.

Massively parallel hardware provides significant gains in efficiency by utilizing vectorization and single-instruction, multiple-data (SIMD) architectures in HEP experiments [12]. It performs best when applying the same computations to multiple pieces of data over large datasets. Given the small sizes of individual events at LHCb (about 60 KB raw event size), a Gaudi algorithm cannot properly take advantage of GPUs. This is why a mechanism for computing data in batches outside of the pipeline is needed.

We deal with Gaudi's limitations by using an offload mechanism. Those algorithms that run on the CPU remain within the Gaudi framework. However, those algorithms that require a GPU get their data from Gaudi, as they normally would, then send it to a special Gaudi service we call "GpuService", which instructs it which kernel to apply, and wait for a response.

The GpuService Gaudi service communicates with an external process called GpuServer. This process "owns" a parallel computation device and hosts the kernels that use it. The server process handles communication, scheduling, and a few auxiliary tasks for aiding kernel development. The details of this GpuService are explained in section 4.

Kernel development is done in C++ very much like it would be in a standalone program. We do not force any libraries of frameworks on the developer. The only requirement is that the kernel implements a function that takes an array of data elements and fills an array with corresponding results.

The server accumulates data elements as it receives them from several disctinct Gaudi processes and calls the developer's function once it has gathered enough instances; when the function is done, the server distributes the results to proper clients. If the function cannot process the incoming data, it can throw an exception; the server will pass the exception to clients.

3.3 Algorithms

The most important algorithms implemented in the LHCb software concern reconstruction of particle trajectories. The track reconstruction consists conceptually of three distinct stages:

- The *pattern recognition* finds the patterns of detector signals, typically produced by charged particles on their way through the detector and forms tracks from these hits. The main algorithms performing the current pattern recognition in LHCb are the "VELO tracking" [23], in which a standalone search is made for straight line segments in the in the vertex detector, which is also known as the VELO (VErtex LOcator), and the ' 'Forward tracking" [25], in which the VELO tracks are used as seeds to find so-called long tracks, which traverse the entire spectrometer, by searching for hits in the tracking stations ("T- stations") which are located behind the spectrometer magnet. Other algorithms are also used to find downstream and upstream tracks, which do not traverse the entire spectrometer, but only the detectors after and respectively before the magnet.
- The *track fit* uses a Kalman filter [14] to obtain the best estimate of the track parameters of the corresponding particle, including corrections due to energy loss and multiple scattering. Typically, the CPU time spent in fitting tracks is several times of what is needed for pattern recognition. The tracks found by the Forward tracking are fitted in this way.
- An additional stage is required to remove duplicate tracks and refine the track selection. In LHCb, this stage is implemented in the so-called "Clone Killer" algorithm [26], in which the best tracks from among those that share many hits are selected and stored as final output of the track reconstruction.

Another relevant algorithm used in the LHCb reconstruction concerns Particle Identification (PID) using RICH detectors. The algorithm used for pattern recognition using Ring Imaging Cherenkov (RICH) system uses the knowledge of particle trajectories through the detector to predict where photons should be expected on the photodetector plane, for a given choice of particle mass-hypotheses. This prediction is compared to the observed distribution of detected photoelectrons, and a likelihood is calculated [15].

All these reconstruction algorithms are good candidates for parallelization in a many-core architecture such as GPGPUs.

4 Offloading mechanism

We facilitate data batching for GPUs with an extension to Gaudi. We run a computation server on each GPU-equipped machine. The computation server owns the hardware device. Multiple Gaudi processes send data to this computation server, asking it to perform specific algorithms. The process combines data into batches and runs the algorithms in such a way as to maximize throughput without creating too much latency. Figure 2 compares the data flow in the original Gaudi framework with our offload mechanism.

Communication between Gaudi processes and the computation server can be done on the same node or across a network. Same-node communication is done via Unix named sockets as an efficient synchronized means of inter-process communication. Other methods, such as network sockets or shared memory can easily be substituted.

Having a central computation server also offers other advantages[16]. The server keeps a log of each algorithm's performance. It can also be asked to record to disk the data coming in and out. Playing back recorded input and testing it against recorded output is a great way of testing algorithms for performance and correctness as they evolve.

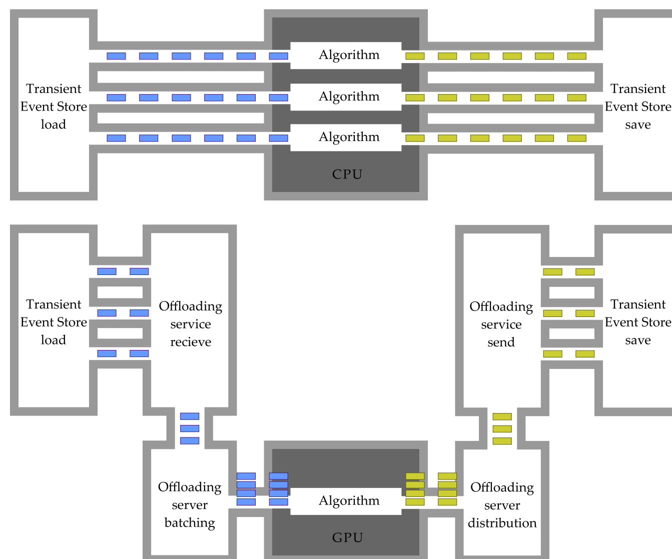

Figure 2 Execution of multiple Gaudi pipelines concurrently using a classical CPU algorithm vs. offloading to a GPU.

5 VELO

The Vertex Locator (VELO) is a silicon strip detector that provides precise tracking very close to the interaction point. It is used to locate the position of any primary vertex within LHCb, as well as secondary vertices due to decay of any long lived particles produced in the collisions. Efficient track reconstruction in the VELO with as few as possible wrong matches is a crucial item for the overall LHCb tracking performance. The current VELO detector [22] is formed by 21 stations, each consisting of two halves of silicon-strip sensors, which measure R and ϕ coordinates. Each half is made of two type of sensors: R sensors with strips at constant radius covering 45° in a so called sector or zone (four zones/sensor), and ϕ sensors with nealry radial strips. A sketch of the VELO detector is shown in Fig. 3. The path of a particle traversing the VELO is considered to be a straight line; this is motivated by the fact that the magnetic field integral in the VELO region is sufficiently small (≈ 0.01 Tm). In addition, each of the particles is assumed to originate from the interaction region, around the beam axis from which B mesons fly for an average distance of ≈ 1 cm, making this an acceptable assumption.

5.1 VELO Pixel

In the upcoming upgrade to happen in 2019, after Long Shutdown 2, the VELO subdetector will be upgraded to a pixel module-based Vertex Locator.

The Pixel VELO will consist of 48 sensors placed along the Z axis. Two rows of 24 sensors will be placed along each of the X axis sides in a similar way to the current installation. Strips will be replaced by a pixel array detection system, with 6 chips of 256×256 pixels each.

Figure 4 shows the sensors placed along the Z axis. The reader is referred to [19] for more details regarding the current upgrade plans.

5.2 Track model

The reconstruction process requires a track model to represent particle trajectories. These models are then used not only by the VELO tracking software, but also by other subdetectors' tracking, such as the one carried out in the TT and the tracking stations or even by the posterior physics analysis.

The tracks seen in the VELO are straight lines; a track in our representation is a five-dimensional real-valued vector called *track state*:

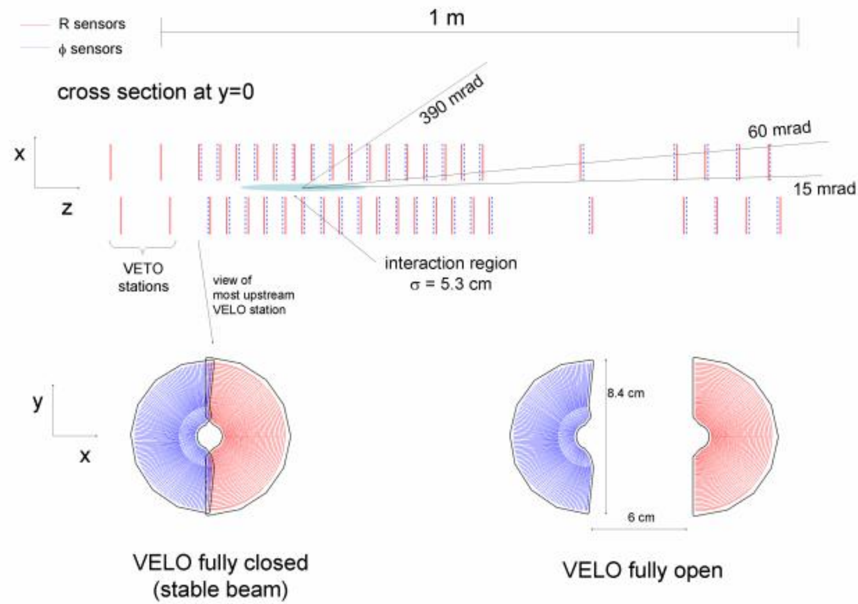

Figure 3 A sketch of the current VELO detector.

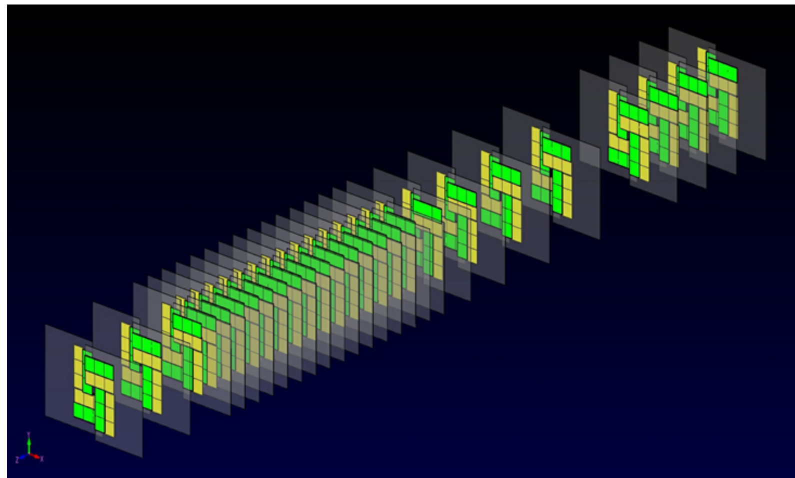

Figure 4 Schema of the 48 sensors of the pixel VELO subdetector, placed along the Z axis. The subdetector system is divided in two sides, placed in +X and -X, with 24 sensors each.

$$Track\ state(z) = \begin{pmatrix} x \\ y \\ t_x \\ t_y \\ q/p \end{pmatrix}$$

The track state can be interpreted as an equation of the line, with additional information about the charge and momentum of the particle producing the track. $tx = \partial x/\partial z$ and $ty = \partial y/\partial z$ are the track slopes in the xz and yz plane. The last parameter q/p is an estimate charge divided by momentum for the reconstructed particle, used to reconstruct trajectories in a magnetic field.

In order to obtain these parameters accurately for each reconstructed particle, an iterative process known as a Kalman Filter is applied. A Kalman Filter [14] is a recursive method that produces a statistically optimal estimate of a series of measurements. We use the Kalman Filter to minimise the χ^2 of each contribution cluster to our track model, equivalently to a least-squares fit. Using this fit, one can tell apart good clusters from the bad for a constructed track and account for noise-like effects, based on its contribution.

5.3 Tracking performance indicators

In order to obtain a measure of the correctness of a reconstruction, several metrics exist. These performance indicators are used when the reconstructed particles are known, like under Monte Carlo event generation.

5.3.1 Reconstruction efficiency

Let F be the set of particles reconstructed, and R the reconstructible set of particles, that is, the set of particles expected to be reconstructed. The reconstruction efficiency is then defined as

$$\epsilon = \frac{|F \cap R|}{|R|}$$

5.3.2 Purity

The purity of a track is defined as

$$purity = \frac{N_{correct}}{N_{total}}$$

where $N_{correct}$ is the number of measurements that originated from the particle that produced the track and N_{total} is the total number of measurements of that specific track.

5.3.3 Ghost fraction

Out of all reconstructed particles, some may come from measurements which are produced by noise effects, like multiple scattering on silicon detectors. Others may simply come from a bad selection of measurements. Particles reconstructed in this way are referred to as *ghosts*, since they do not represent real particles.

The ghost fraction of a reconstruction is defined as

$$ghost\ fraction = \frac{N_{ghost}}{N_{total}}$$

Here, N_{ghost} refers to the number of ghost tracks, which are tracks with a purity of less than 70%.

5.3.4 Clone fraction

A reconstructed particle is made up by several measurements. These measurements may be picked up separately to reconstruct several particles, instead of a single entity. Particles reconstructed in this way are called *clones*.

The clone fraction is then

$$clone\ fraction = \frac{N_{clones}}{N_{tracks}}$$

Reconstruction efficiency and purity are parameters that must be maximized for a good reconstruction. Production-ready good values for the VELO should typically be 99% of the forward reconstructible particles. The number of ghosts and clones must be minimized.

5.4 FastVelo

"FastVelo" [23] is the algorithm developed for tracking of the current VELO and was written to run online in the HLT tracking sequence. For this reason, the code was optimized to be extremely fast and efficient in order to cope with the high rate and hit occupancy present during the 2011-2012 data collection. FastVelo is highly sequential, with several conditions and checks introduced throughout the code to speed up execution and reduce clone and ghost rates.

The algorithm can be divided into two well-defined parts. In the first part (RZ tracking), all tracks in the RZ plane are found by looking at four neighbouring R-hits along the Z axis ("quadruplet"). The quadruplets are searched starting from the last four sensors, where tracks are most separated. Then the quadruplets are extended towards the lower Z region as much as possible, allowing for some inefficiency. The resulting RZ track offers a first estimate of the azimuth, based on the R-sensor zone geometry. In the second part of the algorithm (space tracking), 3D tracks are built by adding the information of the ϕ hits to the RZ track. A first processing step is to define the first and last ϕ sensor to use, then the algorithm starts from the first station with hits searching for a triplet of nearby ϕ hits (in the same sector of the RZ track). The triplet is then added to the RZ track to form a 3D tracklet, so that the track parameters can be estimated. More than one 3D candidate may be constructed, as all the hits within the same 45° sector of all encountered ϕ sensors are compatible. These 3D segments are then extrapolated towards the interaction region by adding hits in the next stations compatible with the tracklet, until two consecutive stations are missed. At least three R and three ϕ hits are required to be assigned to the straight line for it to be considered a valid track. The final 3D track is re-fitted using the information of R and ϕ hits, while hits with the worst χ^2 are removed from the track.

Hits already used in a track are marked as used and not further considered for following iterations ("hit tagging"); this is done to reduce the number of clones produced by the algorithm, avoiding encountering the same track several times. The full FastVelo tracking includes additional algorithms for searching R-hit triplets and unused ϕ hits; these algorithms ran only at HLT2 during 2012. However, the GPU implementation of FastVelo reported in this note refers only to the VELO tracking running on HLT1 during the RUN1

5.5 FastVELO GPU implementation

5.5.1 Implementation details

In this section the attempt to parallelize the sequential FastVelo algorithm for GPU architecture will be described. The GPU implementation is restricted only to the part of the sequential code running on the HLT1 tracking sequence.

The strategy used for developing a parallel version of FastVelo takes advantage of the small size of the LHCb events ($\approx$ 60kB per event, $\approx$ 100 kB after the upgrade) implementing two level of parallelization: "of the algorithm" and "on the events". With many events running concurrently, it can be possible, in principle, to gain more in terms of time performances with respect to the only parallelization of the algorithm.

One of the main problems encountered in the parallelization of FastVelo concerns hit tagging, which explicitly spoils data independence between different concurrent tasks (or "threads" in CUDA language). In this respect, any implementation of a parallel version of a tracking algorithm relying on hit tagging implies a departure from the sequential code, so that the removal of tagging on used hits is almost unavoidable. The main drawback of this choice is that the number of combinations of hits to be processed diverges and additional "clone killing" algorithms (intrinsically sequential and not easy to parallelize) have to be introduced to mitigate the increase of ghost and clone rates. Another issue encountered in the development of the parallel version of FastVelo is due to the R-ϕ geometry of the current VELO that impose a splitting of the tracking algorithm in two sequential steps (RZ tracking plus 3D tracking).

The approach described in this note follows closely the sequential algorithm; therefore, also the tracking algorithm implemented on GPU is based on a local search ("local" method): first seeds are formed by looking only to a restricted set of sensors (quadruplets), then the remaining hits on the other sensors are added to build the full tracks.
The outline of the implementation chosen to parallelize FastVelo can be summarized as follows:

- The algorithm searches for long tracks first, using only the last five sensors downstream the VELO (upstream for backward tracks). Four threads (one for each sensor zone) find all possible quadruplets in these sensors. Then, each quadruplet is extended independently as much as possible by adding R-hits of other sensors. The R-hits of each RZ track are marked as used; potential race-conditions are not an issue in this case, because the aim is to flag an hit as used for the next algorithms.
- Next, the remaining sensors are processed: each thread works on a set of five contiguous R-sensors and find all quadruplets on a zone of these sensors. A check is done on the hits in order to avoid hits already used for the long tracks. In a sense, the algorithm gives more priority to the long tracks with respect to the short ones.

  At this stage the number of quadruplets belonging to the same tracks is huge and a first "clone killer" algorithm is needed to protect against finding the same track several times. All pairs of quadruplets are checked in parallel: each thread of the clone killer algorithm takes a quadruplet and computes the number of hits in common with the others; if two quadruplets have more than two hits in common, the one with worst χ^2 is discarded (here, the χ^2 is defined as the sum of residual of the position of the R-hits of the RZ track with respect to the predicted position given by fitted track).
- Next, each quadruplet is extended independently as much as possible by adding R-hits of other sensors on both directions. After this step, all possible RZ tracks are built. The number of clones generated by the algorithm is still huge, and another clone killer algorithm similar to the one implemented in the previous step is used to reduce the fraction of clone tracks to a tolerable value. In order to detect clones, a check is made for all possible track pairs: if two tracks shares more than 70% of their R-hits, the shortest track, or the one with worst χ^2, is discarded.

  It should be noted that this procedure of cleaning clones follows the same lines of the one implemented in the original FastVelo algorithm ("mergeClones"), the only difference being that in FastVelo the clone killer algorithm is applied only to the full 3D tracks (almost at the end of the tracking), while in the parallel implementation, without hit tagging, we are forced to introduce it well before in the tracking sequence in order to reduce the number of tracks in input to the next steps.
- Next step is to perform full 3D tracking by adding ϕ hits information. Each RZ track is processed concurrently by assigning a space-tracking algorithm to each thread.
  This part is almost a re-writing in CUDA language of the original space-tracking algorithm, with the notable exception of the removal of tag on the used ϕ hits. A minor modification with respect the original code is that in the parallel version the handling of RZ tracks in the sensors overlap regions was simplified to avoid recursive calls present in the sequential algorithm.
  When all 3D tracks have been found, a final cleanup is done on the tracks to kill the remaining clones and ghosts; the clone killer algorithm is the same of the one used in the previous steps, with the exception that now the χ^2 is based on the information of both R and ϕ hits of the track.

Another approach without tagging on the long tracks has been implemented: tracking efficiencies have been found very close to the present algorithm with slightly worst timing performances ($\approx$ 10%). The results described in this note are then referred to the implementation with tagging of the long tracks.

5.5.2 *Preliminary results*

In this section the timing performances and tracking efficiencies obtained with FastVelo on GPU will be compared to the sequential algorithm running on 1 CPU core. Other studies will be presented using a multi-core CPU. These preliminary results refer only to the tracking time without including data transfer time from CPU to GPU, and vice-versa. The reason for this choice was dictated in part by the approach of exploiting the parallelization over the events, where each thread is assigned to an event. This strategy cannot be easily implemented using the standard software framework, originally developed to process sequentially one event at time. A simple offloading mechanism has been developed which is able to load data on GPU memory (in RAW or MDF format) decoding the information of

the VELO hits from raw banks. After the tracking on GPU, the tracks are sent back to the original sequential framework.

The measurements of the tracking time for GPU have been taken using the standard CUDA timer ("CUDA event"), while the timing for CPU has been taken from the detailed FastVelo profile given by the LHCb reconstruction program Brunel [24]. Tracking efficiencies for both GPU and CPU have been obtained from the standard tools provided by Brunel.

The GPU model used for these tests is an NVidia GTX Titan (14 Streaming multiprocessors, each equipped with 192 single-precision CUDA cores), while the CPU is an Intel(R) Core(TM) i7-3770 3.40 GHz.

Two MonteCarlo (MC) samples have been used to evaluate the tracking and timing performances: $B_s \to \phi\phi$ events generated with 2012 conditions (with pile-up of $\nu = 2.5$[a]) and b-inclusive decays produced at with an upgraded scenario for 2015 ($\nu = 4.8$). Timing performances have been compared also with real data using a NoBias sample collected during 2012 ($\mu = 1.6$). In the $B_s \to \phi\phi$ MC sample, the average number of hits per sensor is ≈ 17, while the average number of reconstructed VELO tracks per event is ≈ 80.

The comparison of tracking efficiencies between the GPU implementation and the original FastVelo algorithm for different categories of tracks is shown in Tab. 1 [b]. The efficiencies obtained by FastVelo on GPU are quite in agreement with the sequential FastVelo; in particular, clones and ghosts are at the same level of the original code. Fig. 5 shows the tracking efficiency as a function of the true track momentum P_{true} and the resolution of the impact parameter as a function of $1/P_{T,true}$ obtained by the two algorithms; the overall agreement is good, showing that the GPU implementation does not introduce any distortion on the resolution of the track parameters.
The speed-up obtained by the GPU algorithm with respect to FastVelo running on a single CPU core as a function of the number of processed events is shown in Figs. 6- 8 for the three datasets. The GPU algorithm behaves differently according to the occupancy of hits in the VELO (the average occupancy in the 2015 MC sample is a factor of two higher than 2012 samples); the maximum speed-up obtained by the GPU algorithm with respect to the sequential FastVelo is $\approx$ 3x for the 2012 datasets, while it decrease to $\approx$ 2x for the 2015 sample. The speedup as a function of the number of events can be explained by the fact that the GPU computing resources are more efficiently used as the number of events increases (there are more threads running at the same time).

The comparison of the timing performance has been done using also a multi-core CPU (Intel Xeon E5-2600, 12 cores with hyper-threading and 32 GB of memory). A instance (job) of FastVelo was sent to each core at the same time, with each job processing the same number of events (for this study the number of events/job was set to 1000). The total number of events processed per second as a function of the number of instances is plotted in Fig. 9: the throughput of a single core goes down the more instances are running in parallel (this is due to memory IO pressure, the CPU scaling down its frequency when a lot of cores are running to stay within its power budget).
In the case of $B_s \to \phi\phi$ MC events, the rate of processed events on the multi-core CPU, using all the 24 logical cores, is $\approx$ 5000 events/sec, while on GPU the rate decrease down to $\approx$ 2600 events/sec. However, the number of processed events per second is not a real measure for performances, because it has no meaning when comparing different computing platforms or even computing architectures. A better estimator for these performance studies is the rate of events normalized to the cost of the hardware (events/sec/cost): the GPU gaming-card cost a small fraction of the server used in the HLT farm, so also a moderate speed-up (e.g. 2x) compared to a Xeon CPU can bring a real saving to the experiment (provided the GPU is reasonably well used).

Next steps of these studies will include a development of the full FastVelo tracking on GPU (the part running on HLT2) and the remaining tracking algorithms, such as the Forward tracking [25].

[a] ν is the number of total elastic and inelastic proton-proton interactions per bunch crossing, while μ represents the number of visible interactions per bunch-crossing. LHCb labels simulated event samples according to ν.

[b] Only the VELO tracking running on HLT1 has been implemented on GPU, so that the quoted efficiencies and timings refer to FastVelo in the HLT1 configuration.

Track category	FastVelo on GPU		FastVelo	
	Efficiency	Clones	Efficiency	Clones
VELO, all long	86.6%	0.2%	88.8%	0.5%
VELO, long, p > 5 GeV	89.5%	0.1%	91.5%	0.4%
VELO, all long B daughters	87.2%	0.1%	89.4%	0.7%
VELO, long B daughters, p > 5 GeV	89.3%	0.1%	91.8%	0.6%
VELO, ghosts	7.8%		7.3%	

Table 1 Tracking efficiencies obtained with FastVelo on GPU, compared with the results obtained by original FastVelo code (only the VELO tracking running on HLT1). The efficiencies are computed using 1000 $B_s \to \phi\phi$ MC events, generated with 2012 conditions.

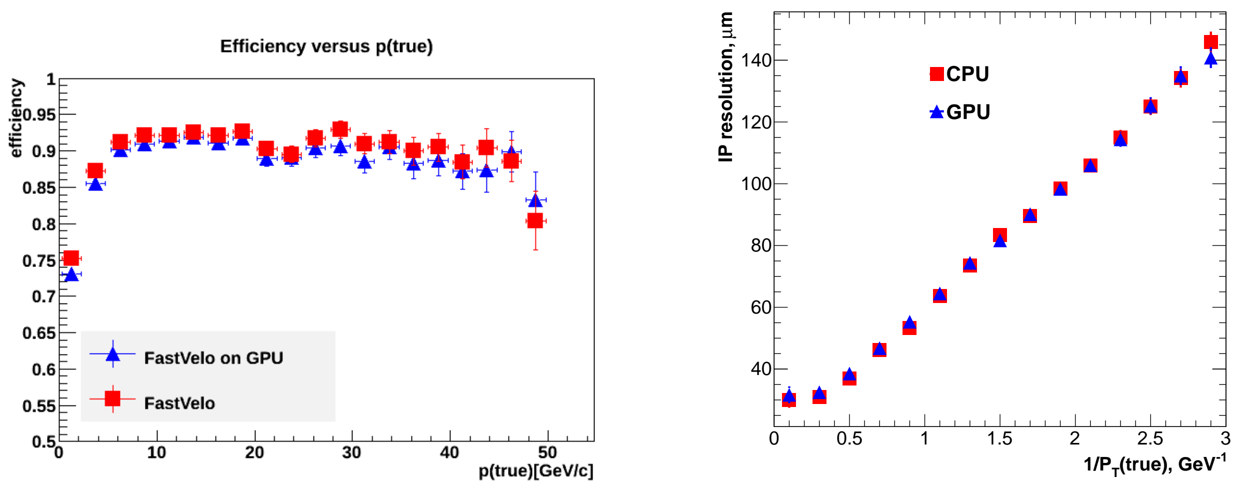

Figure 5 Tracking performance comparisons between the sequential FastVelo and FastVelo on GPU. (Left) Tracking efficiency as a function of the true track momentum P_{true}. (Right) Impact parameter resolution as a function of 1/$P_{T,true}$.

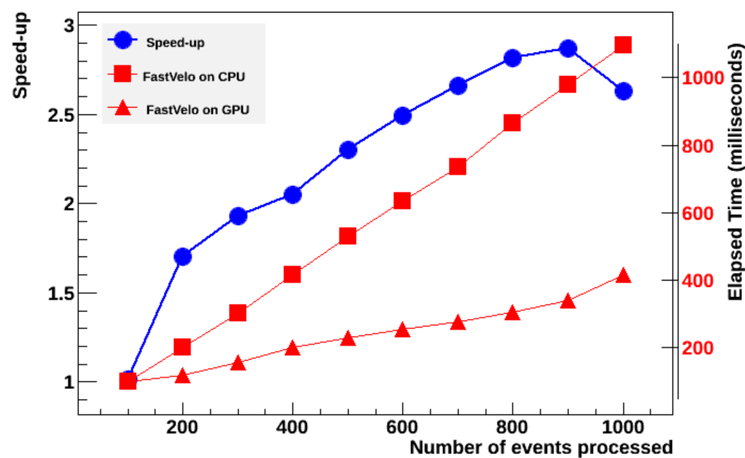

Figure 6 Tracking execution time and speedup versus number of events using a 2012 MC sample of $B_s \to \phi\phi$ decays ($\nu = 2.5$). The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz).

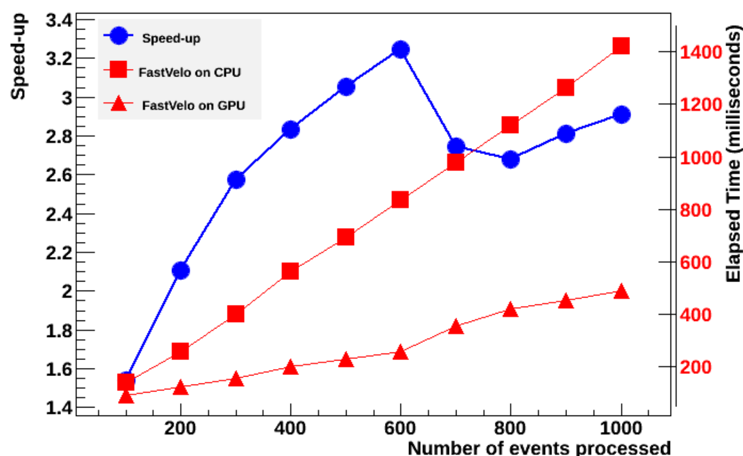

Figure 7 Tracking execution time and speedup versus number of events using a sample of No-Bias data collected during 2012 run ($\mu = 1.6$). The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz).

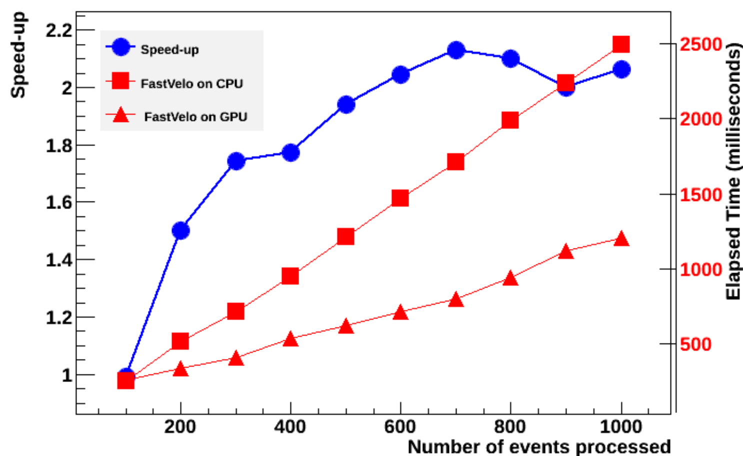

Figure 8 Tracking execution time and speedup versus number of events using a 2015 MC sample of b-inclusive decays generated with $\nu = 4.8$. The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz).

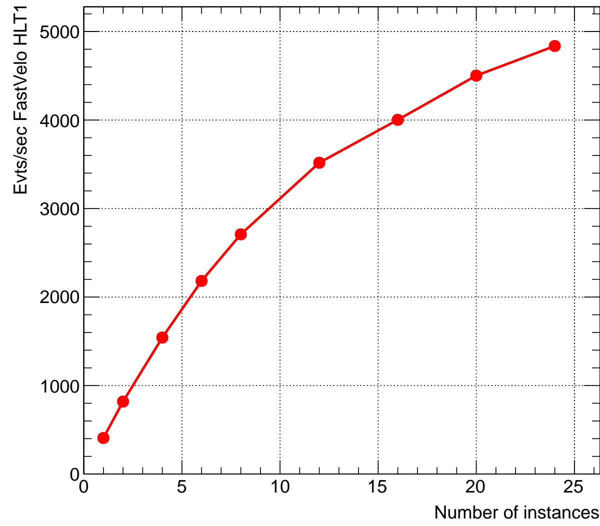

Figure 9 Number of events processed per seconds versus the number of instances of FastVelo tracking running on HLT1. Time measurements are taken with an Intel Xeon E5-2600, 12 cores (24 with hyper-threading).

5.6 VELO Pixel implementation

The VELO Pixel sequential algorithm is composed of three separate stages, as depicted in table 2. The constructs and data types which will be later utilized in the subsequent sections of the algorithm execution are initialized in the *prepare* stage. The generation of tracks, based on the selection of hits in each pixel array, is performed in the *findByPairs* algorithm. Finally, these results are then converted to a format compatible by the Brunel framework in the *storeTracks* stage.

prepare	10%
findByPairs	78%
storeTracks	12%

Table 2 Stages of the VELO pixel subdetection algorithm. The percentages indicate the relative amount of time each stage takes.

Out of these three sections, we will narrow down our interest to the **findByPairs** stage. The motivation behind this decision is easy to grasp by looking at what the purpose of the other stages are. *prepare* and *storeTracks* deal with the generation and conversion of the input and output of findByPairs, into a format understandable by the framework. For our parallelization analysis, we focus on the execution of the main track search algorithm. For brevity, we mention the decisive design points in this analysis. A full description can be found at [20].

A further consideration is worth mentioning. Amdahl's Law, formulated in equation 1, predicts a theoretical maximum limit to the speedup obtainable by an algorithm running on N cores compared to running on a single core, bound to the percentage of code which is parallelizable P in the studied algorithm. In our case study, findByPairs takes 78% of the execution time, therefore this must be taken into account towards a meaningful study of parallelization.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

5.6.1 *Sequential findByPairs*

The VELO clusters average multiplicity is depicted in table 3. If we were to analyze all the possible combinations between hits in different sensors, the combinatorics would scale up, and the execution

Average number of hits per sensor	22.6
Average multiplicity ($hit \times hit$)	771.15
Average multiplicity ($hit \times hit \times hit$)	1544.7

Table 3 Average multiplicity of hits in dataset.

of such an algorithm would be extremely inefficient. The current findByPairs implementation relies on different cuts to break and continue the execution of the loops the track search requires.

A *Local method* is implemented for this search. The *track seeding* is done by selecting pairs of clusters between two neighbouring sensors on each side of the subdetector. The compatibility of clusters is determined by a *tolerance factor* in the angle the hits form with the Z axis. The search starts on the further end from the collision point, where the multiplicity of clusters per sensor is less. The *track formation* is subsequently done by adding compatible clusters with respect to their square root fit with the forming track. If more than three compatible clusters are found within the track formation, its forming clusters are marked as used, effectively decreasing the search window for future tracks.

Finally, a *track selection* stage requires clusters in tracks to present certain properties between each other, in order to minimize *clone* and *ghost* fractions. Although the details of such conditions are subject to change, it is worth noting these incur into *Read-After-Write* (RAW) dependencies among forming tracks, resulting in an arbitrary creation of tracks depending on the order in which the clusters are computed.

5.7 A GPU implementation for VELO Pixel

The current VELO Pixel sequential implementation presents difficulties in order to be ported *as is* to a typical SIMD design. If several events were to be processed simultaneously on attached execution units, *control branches* would cause divergences in the execution flow, thus not taking advantage of vector units in state-of-the-art manycore architectures[c]. A similar issue arises upon parallelizing single event executions. Additionally, RAW dependencies impose a further constraint upon parallelizing single event executions, due to the need of sequentially process clusters, adding a virtual dependency between generated tracks.

Instead, we have redesigned the algorithm by analyzing the feasability of a GPU algorithm for the VELO tracking, studying the requisites and mapping the parallel features of our subdetector to an implementation. Our motivation for such a study springs from similar tracking approaches, like the tracking algorithm in production in the ALICE detector [21].

5.8 Algorithm design

5.8.1 Track seeding

We base our seeding to finding *hit triples*, which as its name indicates, is a tracking seeding algorithm based on the search for triplets of clusters. Three neighbouring clusters are required to form a seed. For each cluster in sensor $n+4$, all clusters in sensor $n+2$ and sensor n are considered. Let's consider a cluster h_1 in sensor $n+4$, a cluster h_2 in sensor $n+2$ and a cluster h_3 in sensor n. A tracklet is created if h_1 and h_2 are under the *maximum slope* threshold, and if h_3 is under the root fit *tolerance* level, in a similar way to the sequential implementation previously described.

Let m be the number of sensors, and n the average of clusters in each sensor. The computational complexity of such a seeding is $O(n^2)$ per hit, and $O(n^2 + m \cdot n)$ for the entire subdetector. The average case $\Theta(m, n)$ depends on the probability of the seed forming a tracklet. This probability is lower than in the case of the two hits-seeding, and it will have an impact in the time results we will analyze later.

[c] This phaenomenon is known as *warp divergence* for NVIDIA GPUs.

5.8.2 Track formation

Taking generated seeds, the resting sensors are probed for compatible tracks, following the fit tolerance principle. Figure 10 depicts this process. A constant error is taken into consideration for the least squares method.

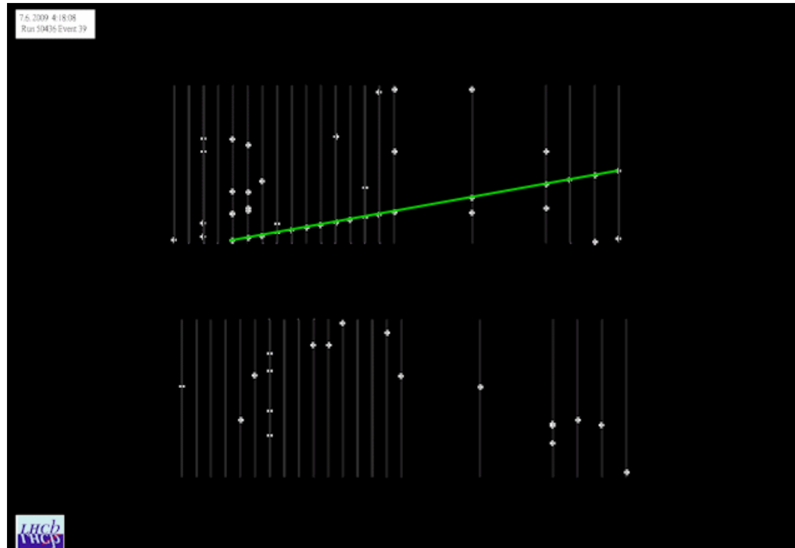

Figure 10 Track forwarding example. A track has been created from 17 hits, in the +Y side of the detector.

The seeding and track formation are implemented in one algorithm, written specifically for the GPU. As we saw in table 3, the average multiplicity in our Montecarlo samples is of $22,7$ clusters per sensor. Following NVIDIA terms, we assign *blocks of threads* to process sensors, and *threads* to process individual clusters. Each block performs the seeding and track formation, considering the clusters in their assigned sensor as the first sensor, and performing a search until the lowest-Z sensor is reached. In this fashion, we can exclude the furthest sensors from blocks processing them as seeds, since no tracks would be found as there are not enough sensors to produce them. Hence, our computation is started with 44 blocks. We have decided to start each block with 32 threads, to match the WARP size and to account for sensors with more clusters, as the ones close to the collision point.

Let m be the number of sensors, and n the average number of clusters in a sensor. For the pair of hits seeding and track forwarding, we can see that in the worst case, each cluster requires the processing of all clusters in previous sensors. The computational complexity for the processing of one cluster is therefore of $O(m \cdot n)$. For all clusters, it is of $O(m \cdot n^2)$. The *seeding* and *formation* is implemented in one single algorithm we will refer to as *gpuKalman*.

5.8.3 Track selection

Up until now, the detected tracks didn't account for overlapping effects. Since the data is being processed in parallel by many threads, seeds may overlap with existing tracks in other threads, and a blocking mechanism preventing this would not be efficient on a parallel architecture. Let l be the length of a track after the Track Forwarding, then there may be up to $l - 2$ overlapping tracks with track length $>= 3$, therefore in the detected tracks bag. We consider a track to be *overlapped* if more than 60% of its clusters exist in other tracks.

Hence, for every detected track we must explore all other detected tracks in search for overlaps. Moreover, we must find a mechanism to define which tracks should remain, and which should not be included in the solution. In other words, in this stage the focus is in keeping the reconstruction efficiency high, while decreasing the *clone* and *ghost rate*.

In order to select which overlapped tracks should stay, we use a *precedence* value based on the tracks length and χ^2. We define an overlapped track t_1 to be *precedent* over an overlapping track t_2 if and

only if the number of clusters of t_1 is greater than those of t_2, or if it has a lower χ^2 if their lengths are equal.

For performing these checks in a more efficient way using GPU Computing, we use a technique known in many-core architectures as *tiling* [17] [18]. Given an amount of elements in main memory on a coprocessor device which require performing an operation between all of them, tiling consists in dividing the data load in *tiles* or chunks of a determined size, and performing a copy to a cache-like memory, followed by a *barrier* operation, so that all threads involved in the operation benefit upon reading / writing the data.

The computational complexity of this processing, given n detected tracks, is of $O(n^2)$. The settings we are using for our many-core algorithm is only one block, with 32 threads. Although this is low, we consider the performance should increase upon executing the algorithm with many events at the same time. The *selection* is implemented in a GPU algorithm we will refer to as *gpuPostProcess*.

5.8.4 Performance results

The datasets under use are described in table 4. Several kinds of production states should give different signatures in the VELO Pixels. Around 500 events are taken for each set of conditions. The average clusters per event is 2365, and a distribution of the clusters versus events is shown in figure 11.

Configuration	Simulation conditions	Production	ν
Upgrade MC	7TeV, VP2.UT.FT-MagDown	Bd_Kstmumu=DecProdCut	7.6
Upgrade MC	7TeV, VP2.UT.FT-MagDown	Bs_Phiphi=CDFAmp,DecProdCut	7.6
Upgrade MC	7TeV, VP2.UT.FT-MagDown	Bs_Phigamma=DecProdCut	7.6
Upgrade MC	7TeV, VP2.UT.FT-MagDown	Bs_Kstmumu=DecProdCut	3.4

Table 4 Conditions of datasets used.

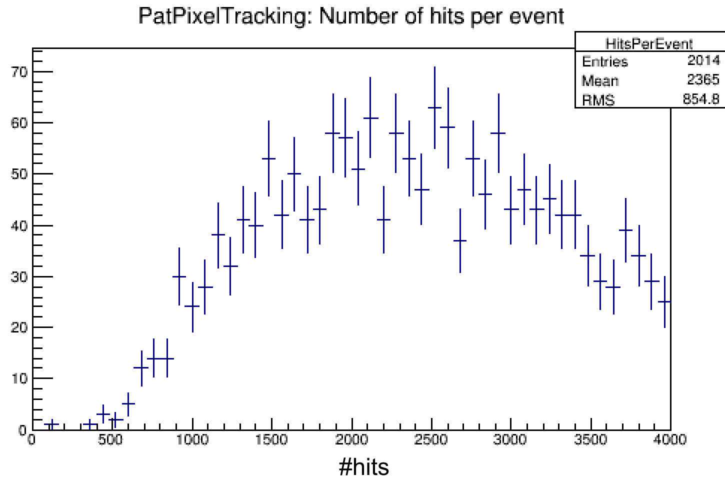

Figure 11 Distribution of hits per event for considered datasets.

Physics results

Table 5 contains the results of the sequential algorithm.

We present the figures for the Physics results in tables 6 and 7. As we can see, the reconstruction efficiency obtained in the bare gpuKalman algorithm is higher than performing both the processing and the post processing. In contrast with the sequential implementation results, our early implementation shows more work is needed in defining better fitting parameters.

The ghost fraction is severely cut down by applying the post-processing. Conversely, the reconstruction efficiency is reduced by a 17%, and the clone fraction goes up by 4% in the latter scenario.

Reconstruction efficiency	75.1%
Forward reconstruction efficiency	96.2%
Ghost Fraction	5.5%
Clone Fraction	23.1%
Purity	99.7%

Table 5 Results for the sequential implementation. A total of 2000 events have been processed, in groups of 500 across each of the above datasets.

Reconstruction efficiency	54.9%
Forward reconstruction efficiency	60.0%
Ghost Fraction	63.4%
Clone Fraction	32.4%
Purity	99.9%

Table 6 Parallel approach Physics results for the *gpuKalman* stage.

Reconstruction efficiency	37.7%
Ghost Fraction	26.3%
Clone Fraction	36.4%
Purity	99.9%

Table 7 Parallel approach Physics results for the *gpuKalman* and *gpuPostProcessing* run sequentially one after the other.

Performance results

Our main motivation in testing such a paradigm was focused on using an intrinsically different architecture, and testing the feasability and the gain in time, if any. Here we focus on the hit-triple timing results against the sequential ones.

	Test setup 1	Test setup 2
CPU	Intel(R) Xeon(R) CPU E5-2650	NVIDIA GTX 690
Frequency per core	2.00Ghz (2.80Ghz with TurboBoost)	1.00Ghz
Memory	32GB DDR3	2GB GDDR5
Cache	20MB (L3)	48KB (L1-like)
Number of cores	8 (16 with HyperThreading)	3072 CUDA cores (2x1536)

Table 8 Test setup.

Table 9 shows the times of our algorithm, both in the sequential and parallel setup. The sequential run executes the searchByPair algorithm, and the parallel the gpuKalman parallel algorithm on a GPU. The Physics results show more work is needed for the gpuPostProcessing stage, and its performance results are left out. The test setup is described in table 8.

Algorithm	Execution time
Sequential, findByPairs	$111.27ms$ per event.
Sequential, findByPairs (extrapolation to number of cores)	$13.91ms$ per event.
Parallel, gpuKalman	$10ms$ per event.

Table 9 Algorithms and execution times.

The parallel algorithm outperforms a single CPU core, as we would have expected. The gpuKalman stage, which is inherently parallelizable, shows an execution time of 10 ms per event, which shows in other words a $11\times$ speedup over the sequential algorithm. Doing a simple extrapolation, and considering a parallelization at a level of events from the CPU side, the GPU presents a $1.39\times$ speedup over the full CPU.

These results show the potential of a manycore architecture over a multi-core counterpart for an intrinsically parallelizable problem such as the VELO Pixel. Future work is needed to polish the Physics results our manycore approach delivers.

6 Research Topics

Possible future directions to continue the present work are detailed in the following subsections.

6.1 FastVELO

A unique opportunity to test the feasibility of the use of GPGPUs for the HLT integrated in the Online farm will be provided by the next runs in 2015, with GPU tracking algorithms running in parasitic mode during the data-taking. For the tests in 2015, we first port the algorithm currently used to reconstruct tracks in the vertex detector to the GPU architecture, avoiding as much as possible departures from the original algorithm. We know this strategy would not yield the greatest speedup, as this algorithm has been optimized to run on CPU; nevertheless, we choose this approach because our first goal is to understand how LHCb events and silicon data should be organized to be processed by parallel architectures and to allow an easier comparison with the current tracking algorithms in terms of physics performance.

6.2 Hardware

The current baseline for the new data center for the upgraded software trigger farm is a containerized solution. Contrary to the current data center, rack space will have to be taken into consideration to keep the size and the cost of the containers low. There are two possible solutions for this scenario right now:

6.2.1 Monolithic servers

There are several high density server solutions currently on the market for housing accelerator cards. Examples are the DELL 720d or SuperMicro 2027GR-TRFT. They can house between two and four GPU class cards in 2 Us of rack space. HPC qualified accelerator cards should work out of the box with these chassis, but for the use of gaming cards some research will have to be done to ensure that the cooling is compatible with the servers.

In case of a scenario where the accelerators are supporting the planned event building system, these would be the preferred solution. A two slot server can service the PCIe40 read-out card and an accelerator card. This assumes that the necessary network ports are integrated into the mainboard of the server, which is usually the case. At the same time these servers allow easy access to the inserted cards for cables, which would be necessary for the read-out board.

6.2.2 External PCIe chassis

Another solution for housing PCIe cards are dedicated boxes that can contain up to 16 accelerator cards. These chassis have no CPU of their own and offer only power, cooling and several external PCIe interfaces to connect to servers. They can be hooked up to conventional servers via special cables, given that they have an external PCIe interface too. These cables have a typical range of 7 m which allows cabling within a single and maybe neighbouring rack. The downside of these chassis are the additional rack space they take up, which can be mitigated by having more compact compute servers though. The big advantage they offer is a certain independence from the CPU server. These chassis would be the preferred solution if most of the computing work of the trigger could be offloaded to accelerators.

As in the monolithic server case, proper cooling has to be researched, should gaming cards be used. Another point to be researched is the I/O capability of these boxes. Usually they are used for their high density of computing power. Since typical HPC applications are working on a given data sample much longer than in HEP, data has to be moved in and out of the box less frequently. We have to ensure that the I/O capability of the boxes are sufficient to feed any accelerator cards with enough data to be worthwhile. In a monolithic server solution this is typically an out of the box feature.

6.3 Software framework

The Gaudi framework was conceived for an underlying sequential processing paradigm where individual events follow an execution chain with control and data dependencies along the computation stages. However, as multi and manycore architectures are becoming commonplace nowadays, there is an ongoing effort towards parallelizing the Gaudi framework in a project known as Gaudi Multi-Threaded (Gaudi-MT).

Gaudi-MT has been crafted as a component addition to Gaudi, to support concurrency. Figure 12 shows a preview of its future architecture. In sequential Brunel, there is only an Event Loop Manager (*EventLoopMgr*), which performs all the computation required for each event. In Gaudi-MT however, three components are added to this picture, a Scheduler (*AlgScheduler*), a Whiteboard and an Algorithm Pool (*AlgPool*).

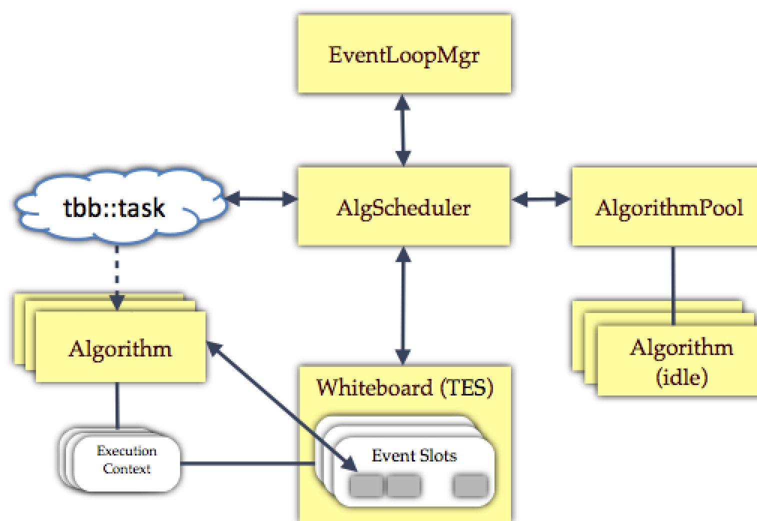

Figure 12 GaudiMT design.

The AlgScheduler gathers events from the EventLoopMgr, and upon verifying the algorithms can be executed, it issues *tasks* which are executed asynchronously. An algorithm can be executed if it is available in the AlgorithmPool, and if its data provided by the Transient Store is ready. Equivalently, an algorithm can be executed when its control and data dependencies are met.

The GPU Manager described in this document would benefit from such a design. With a multi-event capable framework, batches of events can be sent off to a coprocessor-equipped server, drastically decreasing the communication overheads.

6.4 Algorithms

The study carried out in this document refers only to a section of the LHCb detector. As an extension to the VELO tracking, the UT and Forward Tracking are natural candidates to be studied in detail with manycore architectures. The data locality could be exploited without need of additional flow control. As has been shown in other experiments, other subdetector algorithms like the RICH present a good case for analysis in a manycore architecture.

There is a tendency towards manycore architectures, as most HPC centers move into hybrid server-based farms and the core count keeps increasing. Using processors such as the Intel Xeon/Phi will require a shift in the computing paradigms. Our current solutions for solving the LHCb triggering may have to be revisited. Developing triggering software to use coprocessors is not only interesting from a performance point of view, but it also shows how well our software maps into the upcoming exascale processors.

Acknowledgments

We acknowledge support from INFN, Mineco and GenCat.

We also acknowledge fruitful and interesting discussions from our colleagues participating at the GPU@LHCb Trigger meetings.

7 References

[1] Phys. Rev. Lett. **10** (1963) 531; Prog. Theor. Phys. **49** (1973) 652

[2] G.Collazuol, G.Lamanna, J.Pinzino,M.S.Sozzi, *Fast online triggering in high-energy physics experiments using GPUs* Nuclear Instruments and Methods in Physics Research **A 662**, p 49 (2012)

[3] S.Gorbunov et al. *ALICE HLT High Speed Tracking on GPU* IEEE Trans.Nucl.Sci. Volume 58 (4) p 1845 (2011).

[4] NVIDIA Corp. *CUDA C programming guide* PG-02829-001 v6.0, February 2014

[5] S.Chilingaryan et al. *A GPU-Based Architecture for Real-Time Data Assessment at Synchrotron Experiments* IEEE Trans.Nucl.Sci. Volume 58 (4) p 1447 (2011).

[6] J.Nieto et al. *Exploiting Graphic Processing Units Parallelism to Improve Intelligent Data Acquisition System Performance in JETs Correlation Reflectometer*, IEEE Trans.Nucl.Sci. Volume 58 (4) p 1714 (2011).

[7] H.H.Gutbrod (Editor in Chief) *FAIR Baseline Technical Report Executive Summary* ISBN 3-9811298-0-6 (2006)

[8] M.Al-Turany et al. *GPUs for event reconstruction in the FairRoot Framework* 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09), Journal of Physics: Conference Series 219 (2010) 042001.

[9] The NA62 Collaboration, *NA62 Technical Design*, NA62-10-07, CERN, Geneva (2010).

[10] G.Pantaleo et al. *Real-Time Use of GPUs in NA62 Experiment*, Proceedings of CNNA2012.

[11] S.Gorbunov et al. *ALICE HLT High Speed Tracking on GPU* IEEE Trans.Nucl.Sci. Volume 58 (4) p 1845 (2011).

[12] T.Kollegger, D.Rohr *Alice High Level Trigger Tracking*, Proceedings of CNNA2012.

[13] Corti, G.; Cattaneo, M.; Charpentier, P.; Frank, M.; Koppenburg, P.; Mato, P.; Ranjard, F.; Roiser, S.; Belyaev, I.; Barrand, G., *Software for the LHCb experiment*, Nuclear Science, IEEE Transactions on , vol.53, no.3, pp.1323,1328, June 2006

[14] R. Kalman. *A new approach to linear filtering and prediction problems*.
Transactions of the ASME, Journal of Basic Engineering, D82:35-45, 1960.

[15] R.W. Forty, O. Schneider, *RICH pattern recognition*, LHCb-98-040 (1998)

[16] `https://lbonupgrade.cern.ch/wiki/index.php/Gaudi_GPU_Manager`

[17] Wen-mei Hwu, David Kirk
Lectures of ECE 498, University of Illinois

[18] Wen-mei Hwu, David Kirk
Programming Massively Parallel Processors: A Hands-on Approach.

[19] Rodriguez Perez P., LHCb VELO Group. *The LHCb VELO Upgrade*.
http://arxiv.org/abs/1302.6035

[20] D. H. Cámpora Pérez, F. Sancho Caparrini, M. A. Martínez del Amor, N. Neufeld. A Study of a Parallel Implementation for the Pixel VELO Subdetector.

[21] David Rohr. Diploma Thesis, ALICE TPC Online Tracking on GPU based on Kalman Filter.

[22] P R Barbosa-Marihno et al. *LHCb VELO (VErtex LOcator): Technical Design Report*, CERN-LHCC-2001-011 (2001).

[23] O. Callot, *FastVelo, a fast and efficient pattern recognition package for the Velo*, LHCb-PUB-2011-001 (2011)

[24] *The Brunel project*, http://lhcb-release-area.web. cern.ch/LHCb-release-area/DOC/brunel

[25] O. Callot, S. Hansmann-Menzemer, *The Forward Tracking: Algorithm and Performance Studies* LHCb-015-2007 (2007)

[26] A Perieanu, *A Fast Algorithm to Identify and Remove Clone Tracks*, LHCb-2008-020 (2008)