PROCEEDINGS OF SCIENCE

Many-core studies on pattern-recognition in the LHCb experiment

Stefano Gallorini[*†]
Author University and INFN Padova
Address: Via Marzolo 8, 35131, Padova, Italy
E-mail: stefano.gallorini@cern.ch

The LHCb experiment is entering in its upgrading phase, with its detector and read-out system re-designed to cope with the increased LHC energy after the long shutdown of 2018. In this upgrade, a trigger-less data acquisition is being developed to read-out the full detector at the bunch-crossing rate of 40 MHz. In particular, the High Level Trigger (HLT) system, where the bulk of the trigger decision is implemented via software on a CPU farm, has to be heavily revised. Since the small LHCb event size, many-core architectures such as General Purpose GPU (GPGPU) and multi-core CPUs can be used to process many events in parallel for real-time selection, and may offer a solution for reducing the cost of the HLT farm. Track reconstruction and vertexing are the more time-consuming applications running in HLT and therefore are the first to be ported on many-core. In this proceeding, we present our solution for porting the existing tracking algorithm of the VErtex LOcator (VELO) detector to GPGPU, and we show the achieved performance.

Technology and Instrumentation in Particle Physics 2014,
2-6 June, 2014
Amsterdam, the Netherlands

[*]Speaker.
[†]On behalf of the GPU@LHCbTrigger working group

PoS(TIPP2014)192

1. Introduction

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, many-cores in the near future. For this reason, we might expect improvements in execution times coming from a clever use of the multicore structure and parallelization. Therefore, sensibility advises to build up a program to study and exploit the possibilities of parallelization 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), specialized for compute-intensive, highly parallel computation. GPGPUs may offer a solution for reducing the cost of the HLT farm for the LHCb upgrade and R&D studies have started to evaluate the possible role of this architecture in the new trigger system.

In the following section we discuss our preliminary attempt to port the existing tracking algorithm of the VErtex LOcator (VELO) detector on GPGPU, and we show the achieved performances.

1.1 FastVelo

The VELO [1] 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. The VELO detector 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 nearly radial strips. A sketch of the VELO detector is shown in Fig. 1.

"FastVelo" [2] is the algorithm developed for tracking of the current VELO and was written to run online in the High Level Trigger (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 for 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. 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. The triplet is then added to the RZ track to form a 3D tracklet, so that the track parameters can be estimated. These 3D segments are then extrapolated towards the interaction region by adding hits in the next stations compatible with the tracklet. The final 3D track is re-fitted using the information of R and ϕ hits, while hits with the worst χ^2 are removed

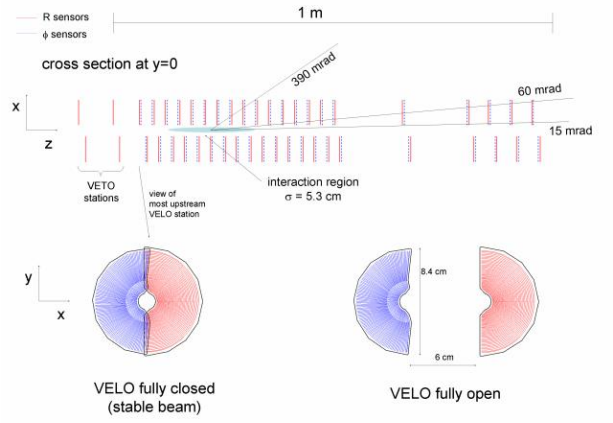

Figure 1: A sketch of the current VELO detector.

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 work refers only to the VELO tracking running on HLT1 during the RUN1.

2. GPU implementation

The strategy used for porting FastVelo to GPU architectures 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. The GPU algorithm was adapted to run on GPU using the NVIDIA Compute Unified Device Architecture (CUDA) framework [3].

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.

- Then 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, 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.

3. 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 decoding the information of the VELO hits from raw data. 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 [4]. 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. A MonteCarlo (MC) sample of $B_s \to \phi\phi$ events generated with 2012 conditions (with pile-up of $\nu = 2.5$[1]) has been used to evaluate the tracking and timing performances. 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 [2]. 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. 2 shows the tracking efficiency as a function of the true track momentum P_{true} as 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. 3- 4. The maximum speed-up obtained by the GPU algorithm with respect to the sequential FastVelo is $\approx 3\times$ for the 2012 datasets. 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).

[1] 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.

[2] 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.

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. 5: 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. 2$\times$) compared to a Xeon CPU can bring a real saving to the experiment (provided the GPU is reasonably well used).

Next steps of this work 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 [5]. In addition, we plan to test FastVelo on GPU in parasitic mode during the RUNII in 2015 in order to assess the impact and the feasibility of the many-core solution on the HLT infrastructure.

References

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

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

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

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

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

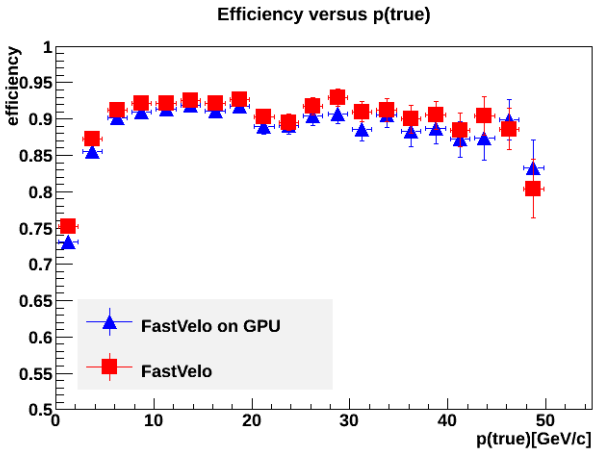

Figure 2: Tracking performance comparisons between the sequential FastVelo and FastVelo on GPU. Tracking efficiency as a function of the true track momentum P_{true}.

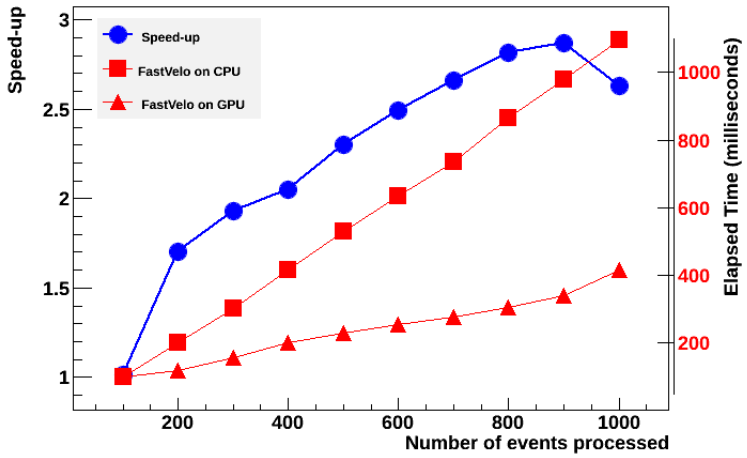

Figure 3: Tracking execution time and speedup versus number of events using a 2012 MC sample of $B_s \to \phi\phi$ decays ($\nu = 2.5$). The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz).

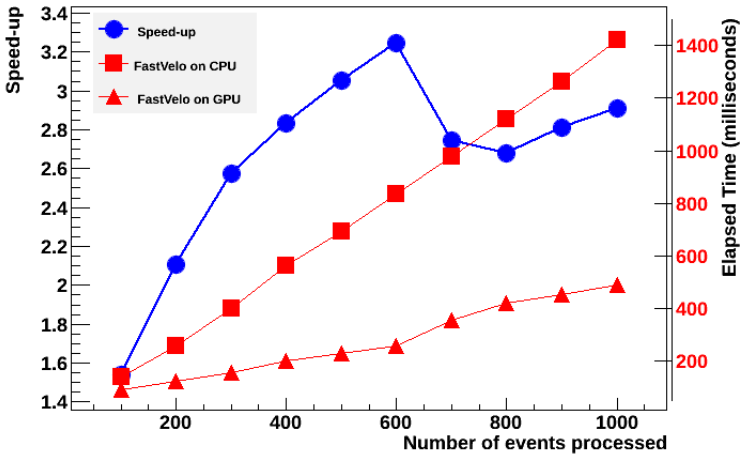

Figure 4: Tracking execution time and speedup versus number of events using a sample of NoBias data collected during 2012 run ($\mu = 1.6$). The GPU is compared to a single CPU core (Intel(R) Core(TM) i7-3770 3.40 GHz).

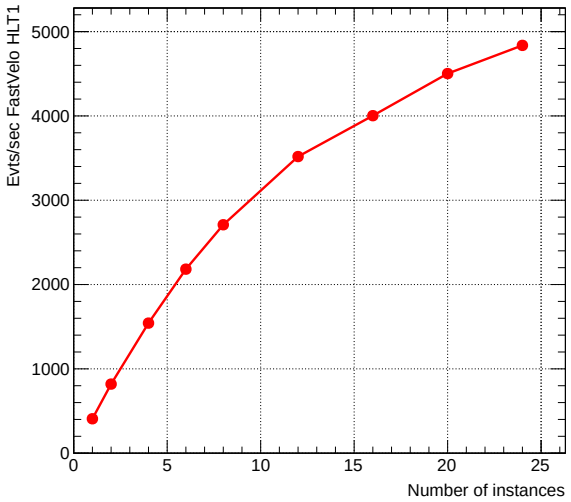

Figure 5: Number of events processed per seconds versus the number of instances of FastVelo tracking running on HLT1. Time measurements are taken with an Intel Xeon E5-2600, 12 cores (24 with hyper-threading).