DE GRUYTER OPEN Acta Univ. Sapientiae, Informatica **8,** 2 (2016) 171–185

DOI: 10.1515/ausi-2016-0008

Jet browser model accelerated by GPUs

Richárd FORSTER
Eötvös Loránd University
Faculty of Informatics
email: forceuse@inf.elte.hu

Ágnes FÜLÖP
Eötvös Loránd University
Faculty of Informatics
email: fulop@caesar.elte.hu

Abstract

In the last centuries the experimental particle physics began to develop thank to growing capacity of computers among others. It is allowed to know the structure of the matter to level of quark gluon. Plasma in the strong interaction. Experimental evidences supported the theory to measure the predicted results. Since its inception the researchers are interested in the track reconstruction. We studied the jet browser model, which was developed for 4π calorimeter. This method works on the measurement data set, which contain the components of interaction points in the detector space and it allows to examine the trajectory reconstruction of the final state particles. We keep the total energy in constant values and it satisfies the Gauss law. Using GPUs the evaluation of the model can be drastically accelerated, as we were able to achieve up to 223 fold speedup compared to a CPU based parallel implementation.

1 Introduction

The huge measurement data is generated in the experiment of high energy particle physics e.g. in the ATLAS experiment (CERN) $\sim 40 \times 10^6$ events per second are detected which requires 64 TB/sec. Every year at about one

Computing Classification System 1998: I.1.4
Mathematics Subject Classification 2010: 58A20
Key words and phrases: jet, trajectory reconstruction algorithm, database of experimenthal particle physics parallel computing, GPU, CUDA

milliard events are measured, this represents three milliard simulation of the event each year and the number of detected particles is growing exponentially. The evaluation requires large computer capacity. Many fundamental questions raise in this research. One of these is the trajectory reconstruction. This process contains more levels. The first is a clearing the pure measurement data. Next process contains the path reconstruction. This method includes two types of elaboration. One of them is an online process to apply the level of assembly program. In the next step the valuable stored data is studied by high level computer program in batch mode.

In our article we work on the first level to use directly measured data set. A jet browser algorithm was published in [1] to study the particle orbit in 4π calorimeter.

The GPUs are providing an easy to program parallel model [10] that makes us capable to achieve higher precision in our computations, while also reaching out for bigger datasets [5]. The ever evolving and increasingly efficient architecture of the GPUs makes it also a viable option to run the applications even on a laptop these days without relying on supercomputers and very special not consumer level hardwares. Thus we implement a CUDA based parallel implementation of the Jet browser to broaden the limit of the original algorithm.

2 Jet physics

The jet physics [6] plays important role in the high energy physics. We present the process from the collision of protons to observable particles by detector (Figure 1).

It contains three different parts. The first range is the *parton* session, which contains the quark gluon plasma with strong interaction on the distance 10^{-15}m. In the theoretical model the jet are produced in hard scattering processes generating high transverse momentum quarks or gluon. The next part of the process is *particle*, where the constituents are formed from the quarks and gluon. This is called *hadronisation* procedure. In the last phase we detect the final state objects, which are the components of the electromagnetic and hadronic showers. The theoretical studying applies the Monte Carlo simulations. At the *parton* and *particle* process PYTHIA [8] software package is used to calculate and at *detector* phase the final state particle is simulated by GEANT [11].

We mention that, there are more kind definitions of the jet, because the laws of physics are different between short-range and the finale hadronisation.

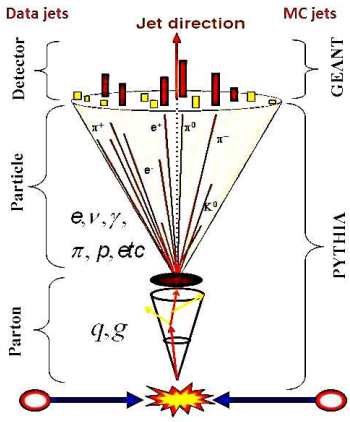

Figure 1: The structure of jet

It is important, that the definition is adequate with both theoretically and experimentally.

3 Jet algorithm

The experimental data which is contains of large number of particles to measure their four-momentum. The particle's energy, impulses p_T and position retrieve from data set. Two main jet definitions [2] are being used. One of them is Cone Jet and the other is the k_T Jet algorithm. In the cone jet algorithm we apply that the jet stays in circular range in the plain of the detector to describe by angles to search the stable energy state. The k_T algorithm can be used, where the values and direction of particle's momentum have the same order of magnitude, therefore the finale state showers will be collinear. These definitions fulfill both theoretical and experimental model also. Several advanced procedures [4, 7, 3] has been developed in this research field.

4 Parallel graph based trajectory reconstruction

In [1] a graph based trajectory reconstructing method was introduced. As all reconstruction algorithms, this is also very computation intensive, where even if the input dataset isn't very large, the combination of those inputs can generate a lot of work. To effectively speed up the process now we introduce a

CUDA based GPU implementation for the same problem, as the GPUs proved to be a valuable device in parallel computations [5].

4.1 Jet browser model

The jet analysis and the jet reconstruction method are applied to study the structure of jet in high energy physics. A shower is a narrow cone of hadrons and other particles which are produced in *detector* phase. These constituents are measured by detector to determine the trajectories and the type of constituents [1].

In 2009 year Gy. Vesztergombi has presented a new idea of a 4π detector [9]. A few particle trajectories (e^-, e^+, γ) can be reconstructed by a model, which was published [1]. The component of shower take part in the electromagnetic and strong interaction, decaying in two or more particles, but we neglect the strong interaction, because the cross section is very small. We can measure the point of the elements (e^-, e^+), which consists of components of the three dimensional Euclidean space. The jet reconstruction model involves the energy and charge conservation. It has been proven that the orbits correspond to weighted directed tree graph $G(\Psi, E, V, w)$.

This method consists of three steps: We find all of neighbor detected points and fit a straight line to them. Next we merge these small pieces for a long orbit. At the end we determine the common points of the trajectories.

Let us denote the set of measured points by V_p in the Euclidean space. The three-points-straight is accepted, if the fitting error is smaller then the value of ε_Y, ε_Z. These quantity depend on the experimental and theoretical considerations. The set S_{stp} contains the three-point-straight. This can be constructed by recursive using the set V_p with finite steps.

In the second part of the model we merge the short peaces for a long trajectories. In this case we have taking into account the curvature of the orbits and the distance between two different straights. The insertion was successfully, if the peaces were very close to each other. The time sequence of the measured points has strong consecutively, therefore the postfix and prefix map can be defined unique on this set.

During the hadronisation we needed to find the decay points to develop an algorithm, which can solve the original problem of this article.

Three types of decay point were introduced *Children case*, *Parent-child cases* and *Undetected parent*. We study, when two or more orbits create from the same points, it means the *Children case*. The *Parent-child case* continues the building *ChildStraights*, then we study the energy dominant particle case to

construct a more complicated tree graph. At the end we need take into account that situation, when a particle is not measured. In the experimental particle physics there are a few particle which we can not detected by this type of the calorimeter (i.e. γ). Then we apply that the energy is conservation during this process.

The experiment consists of a beam (direction of the shoot is parallel with Z axis). It interact with target due to result electromagnetic shower

In [1] a graph based trajectory reconstructing method was introduced. As all reconstruction algorithms, this is also very computation intensive, where even if the input dataset isn't very large, the combination of those inputs can generate a lot of work. As these ideas were proposed a couple of years ago, the originally used architecture for the computation may not be so efficient now. Back then a grid cluster was used to run the algorithm, but now the GPUs have outgrown the performance of smaller CPU clusters with their much more efficient design and seriously parallel architecture. Hence our current implementation involves CPUs, running the algorithm in parallel and GPUs, doing the computation in massively distributed manner, as they can effectively speed up the process as we already shown it in [5].

4.2 Implementation

The machine used for implementation has a GeForce GTX 980M with "computing capability" 5.2 [10] and an Intel Core i7-4710HQ CPU (Table 1).

4.2.1 CUDA memory hierarchy

The threads running on a CUDA capable GPU can access data from multiple memory spaces (Figure 2) [10]. Each thread has its own private local memory. The blocks containing the threads has access to a shared memory that is visible from all the contained threads. During the execution the whole set of threads launched in the grid have access to the same global memory. Additionally there are two read-only memory spaces, that can also be accessed by all the threads. These are the *constant* and *texture* memories. The global, constant, and texture memories are optimized for different usage scenarios.

	GeForce GTX 980M
Technical Specifications	Compute Capabiliting 5.2
Transistors (Million)	5200
Memory (GB)	4
Memory Bandwidth (GB/s)	160
GFLOPs	3189
TDP (watts)	125
	i7-4710HQ
Transistors (Million)	1400
Connected memory (GB)	24
Memory Bandwidth (GB/s)	25.6
GFLOPs	422
TDP (watts)	47

Table 1: GeForce GTX 980M and Core i7-4170HQ technical specifications

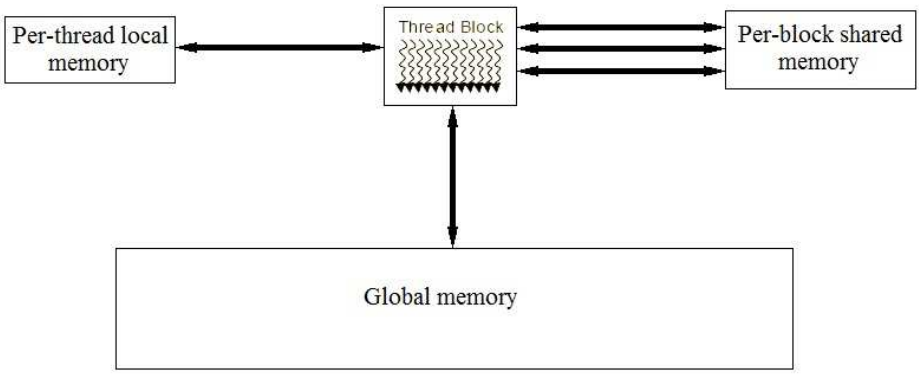

Figure 2: CUDA capable GPU's memory hierarchy

4.2.2 Design choices

From the memory hierarchy (Section 4.2.1) our solution uses the global, shared and local memory. For initialization we move the array storing the original detected points to the GPU's global memory. We process on this input to generate the required triplets, that will build up the trajectories. As for the triplets we have to match the points of three consecutive detector layers, we can map this to a 3 dimensional block to process on the GPU. As the maximum length in the Z coordinate can be only 64, we decided to set it to it's maximum value and align the rest accordingly. As one block can have 1024 threads maximum, we make our blocks to be $(x = 4, y = 4, z = 64)$ in size. Because we would like to check all points from one layer to all the others in the next and also on the third one, this process will generate a huge number of read operations (1024 comparisons by each block) on the device's global memory. While caching is available [10] on the GPU used for implementation, it is still time consuming to fetch all the data from the device memory. Hence at the beginning of the computation we further push the data from the global memory into shared memory, drastically decreasing the required time to proceed, considering it can be 100 times faster [10] compared to the global memory. This is because the global memory is on the card, while the shared memory and the registers are on the chip. As a result of this, the triplets will be the generated online and will be stored on the device. An example of block assignment is shown on Figure 3.

The triplets stored in device memory contains a pointer to the possible next part, the child element and there is also a pointer to the previous chunk, the parent element, basically making a linked list at the end. The process on the triplets is similar to the point matching done before in terms of block settings. In this case we only need a two dimensional block as we only need to check the triplets with each other, so we set the size of the block in the number threads to be $(x = 32, y = 32)$.

4.2.3 Algorithm

Following the stated principles in 4.2.1 and 4.2.2 the host side of the final algorithm for the triplet generation is in Figure 4.

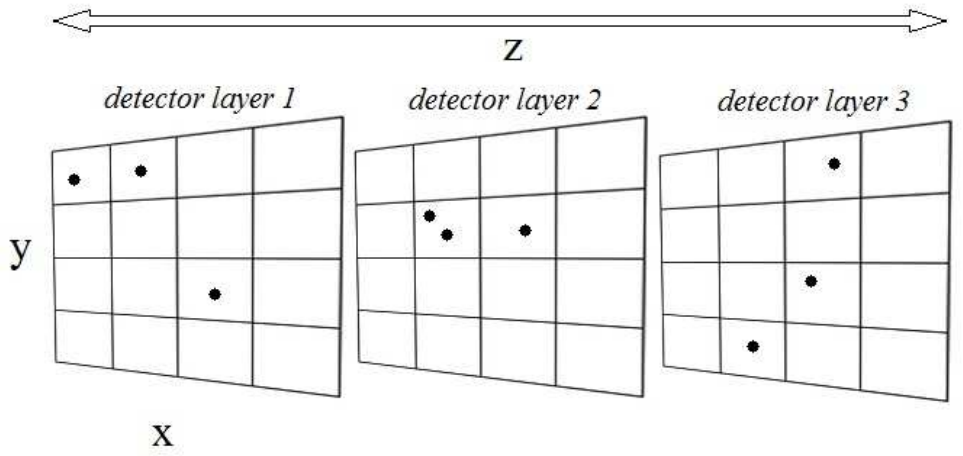

Figure 3: The detector layers as they are represented in a three dimensional CUDA block.

```
1:  procedure makeTriplets(P, L, NP)
2:      CudaMalloc(T)
3:      DT, DP, DNP ← CudaMemcpyToDevice(T, P, NP)
4:      start ← 0
5:      for each i ∈ 0..L − 2 do
6:          x, y, z ← NP[i]/4 + 1, NP[i + 1]/4 + 1, NP[i + 2]/64 + 1
7:          threads(4, 4, 64)
8:          blocks(x, y, z)
9:          makeTripletsKernel(DT, DP, DNP, i, start)
10:         start ← start + NP[i]
11:     end for
12: end procedure
```

Figure 4: The host part of the triplet generation, inputs are the points, the number of layers and the number of points per layers.

The implemented kernel function, which needs to be called from the host side to initiate the computations on the GPU is detailed in Figure 5. As this is the most time consuming kernel, it incorporates the shared memory to compute the triplets as fast as possible.

```
1: procedure makeTripletsKernel(DT, DP, DNP, i, start)
2:     j ← blockIdx.x ∗ blockDim.x + threadIdx.x
3:     k ← blockIdx.y ∗ blockDim.y + threadIdx.y
4:     l ← blockIdx.z ∗ blockDim.z + threadIdx.z
5:     SP ← Memcpy DP to shared memory
6:     if Line found on (SP[j], SP[k], SP[l]) then
7:         T[start + j] ← (SP[j], SP[k], SP[l])
8:     end if
9: end procedure
```

Figure 5: Device kernel to generate triplets, inputs are the array for the triplets, the detected points, the number of points per detector layer, the index of the first layer being tested and the index, where the triplets are starting in the array for the given layer.

The host side of the algorithm used to generated the lines in the trajectories can be found in Figure 6.

```
1: procedure makeTriplets(T, NT)
2:     x, y, z ← NT/32 + 1, x, 1
3:     threads(32, 32, 1)
4:     blocks(x, y, z)
5:     makeLinesKernel(T)
6: end procedure
```

Figure 6: The host side algorithm for the line generation, inputs are the triplets in the device memory and the number of them.

The device function required for the line generation on GPU is in Figure 7.

```
1: procedure makeLinesKernel(T)
2:     i ← blockIdx.x * blockDim.x + threadIdx.x
3:     j ← blockIdx.y * blockDim.y + threadIdx.y
4:     if T[j].next == NULL then
5:         if could fit line on (T[j], T[i]) then
6:             T[j].next ← T[i]
7:             return
8:         end if
9:     end if
10: end procedure
```

Figure 7: The device function to generate the lines, input is the array allocated in the device memory for the triplets.

4.3 Results

Because the problem at hand requires the comparison of all the possible combinations of the points in three consecutive layers and later the triplets are checked in a similar fashion, there is plenty of room for parallelization. Also the memory bound properties of the algorithm makes good use of the shared memory on the GPU. As we will see, the GPUs memory hierarchy (Section 4.2.1) makes them more suitable for these kind of applications. First, we take a look at the triplet generation using a CPU based parallel implementation and the GPU specific solution, comparing how they fair to each other. Then we do the same for the line generation.

The system used for development and testing is described in Table 2.

CPU	GPU	OS	Compiler	CUDA version
Intel Core i7 4710HQ	GeForce GTX 980M	Windows 10 Pro	Visual C++ 2013	7.0

Table 2: The test system

To evaluate the performance of our algorithms running on both CPU and GPU, we generated a dataset by simulating 2000, 4000, 8000, 16000 events using Geant4, running it with 100 MeV as the energy. The number of detected points were 19500, 38855, 77474, 154905 respectively (Figure 8). The simulated detector had 9 layers, giving us 3 batch of layers containing points to check for triplets.

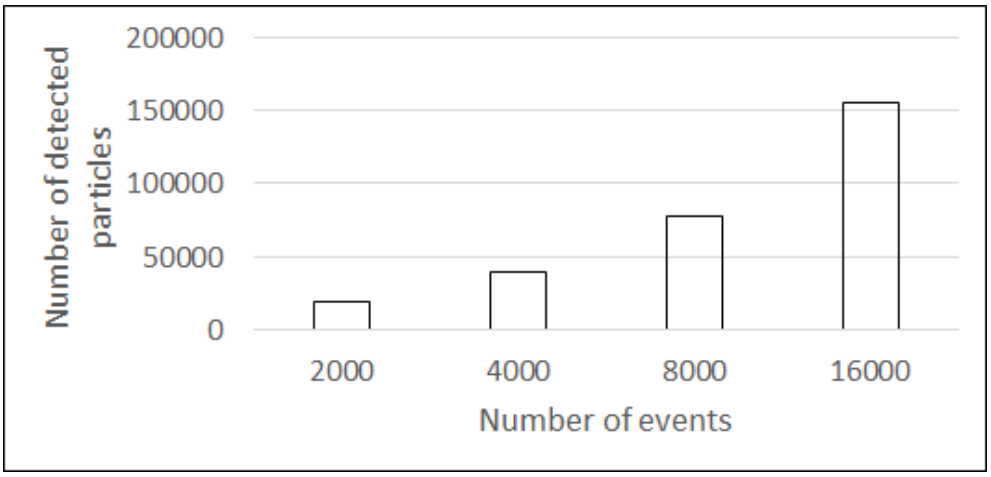

Figure 8: Number of detected points on the different number of events

Evaluating the same dataset on a version of the implementation, that does not use any of the shared memory on the GPU, the runtime is 5 times faster compared to the parallel CPU implementation. By changing the algorithm just slightly with moving the data to shared memory, we gain 27 times faster performance compared to the previous GPU results, which means compared to the CPU implementation the computation is up to 168 times faster in triplet generation and 223 times faster in line generation.

The runtime (Figure 9) on CPU was $\mathtt{timeT}_{\mathtt{cpu}} = 226.627\mathtt{s}$, the same computation took $\mathtt{timeT}_{\mathtt{gpu}} = 44.083\mathtt{s}$ on the GPU, while using the shared memory it was just $\mathtt{timeT}_{\mathtt{gpu}_{\mathtt{sh}}} = 1.607\mathtt{s}$.

In the following we will keep using the GPU implementation using the shared memory. In Figure 10 we can see as we increase the number of iterations the difference between the CPU's and GPU's runtime is increasing, while the CPU can take hours in some cases the GPU runs only for minutes.

As we already saw the results for 2000 events, here we describe the numbers for the other ones. As such on 4000 events the runtime on CPU was $\mathtt{timeT}_{\mathtt{cpu}_{4000}} = 1817.95\mathtt{s}$, the same computation took $\mathtt{timeT}_{\mathtt{gpu}_{\mathtt{sh}_{4000}}} = 10.387\mathtt{s}$ on the GPU. On 8000 events the times were the following: on CPU we finished in $\mathtt{timeT}_{\mathtt{cpu}_{8000}} = 13647.3\mathtt{s}$, while on the GPU in $\mathtt{timeT}_{\mathtt{gpu}_{\mathtt{sh}_{8000}}} = 80.998\mathtt{s}$. For 16000 events on the CPU we could not finish in a reasonable time. The computation was running for more than 15 hours and it still couldn't finish. On the other hand the GPU could give back results in $\mathtt{timeT}_{\mathtt{gpu}_{\mathtt{sh}_{16000}}} = 644.446\mathtt{s}$.

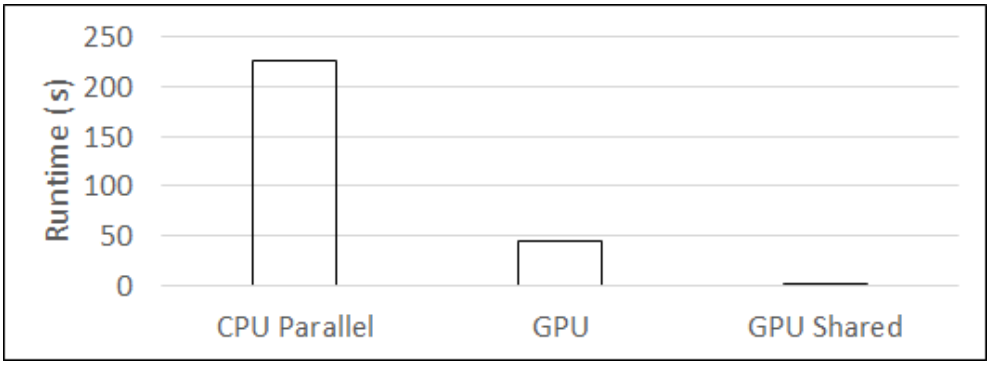

Figure 9: Triplet generation time on 2000 events

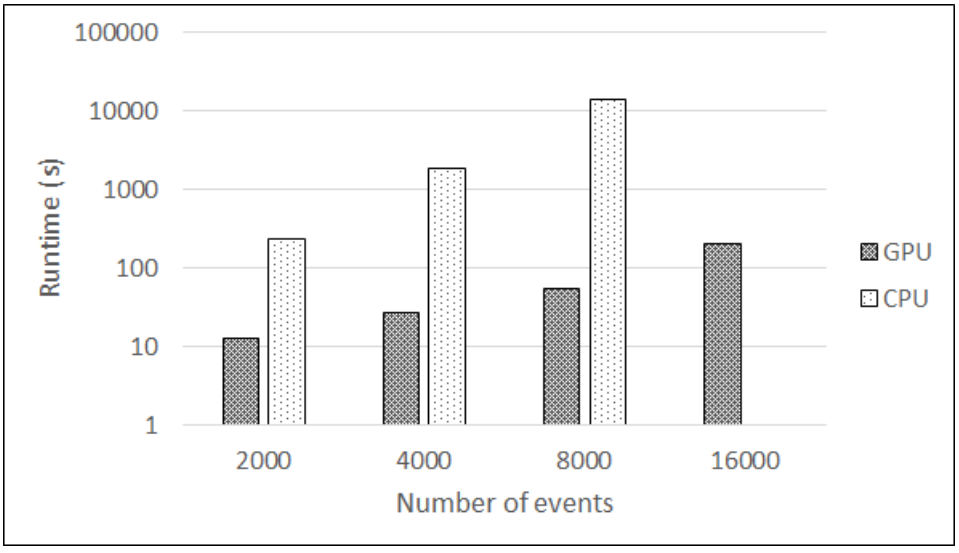

Figure 10: Triplet generation time on CPU and GPU

In Figure 11 we can see that the performance difference is also very clear in the generation of the lines. In this case while the CPU takes minutes to finish, the GPU can be done in seconds. Also here we have a runtime value for CPU under 16000 events, thanks to the lower number of combinations, that needs to be computed.

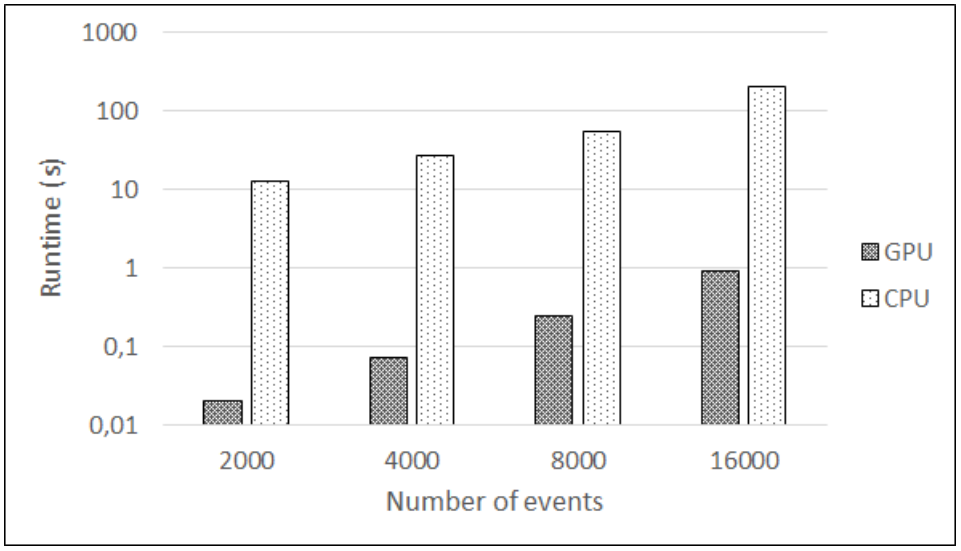

Figure 11: Line generation time on CPU and GPU

Taking a closer look on the Figure: on 2000 events, the runtime on the CPU is $\mathtt{timeL}_{\mathtt{cpu}_{2000}} = 12.685\mathtt{s}$ and on the GPU $\mathtt{timeL}_{\mathtt{gpu}_{2000}} = 0.02\mathtt{s}$. While on 4000 events the runtime on CPU was $\mathtt{timeL}_{\mathtt{cpu}_{4000}} = 26.027\mathtt{s}$, the same computation took $\mathtt{timeL}_{\mathtt{gpu}_{4000}} = 0.071\mathtt{s}$ on the GPU. On 8000 events the times were the following: on CPU we finished in $\mathtt{timeL}_{\mathtt{cpu}_{8000}} = 53.566\mathtt{s}$, while on the GPU in $\mathtt{timeL}_{\mathtt{gpu}_{8000}} = 0.247\mathtt{s}$. For 16000 events on the CPU we got $\mathtt{timeL}_{\mathtt{cpu}_{4000}} = 204.694\mathtt{s}$, while on the other hand the GPU could give back results in $\mathtt{timeL}_{\mathtt{gpu}_{16000}} = 0.916\mathtt{s}$.

One reconstructed event can be seen on Figure 12.

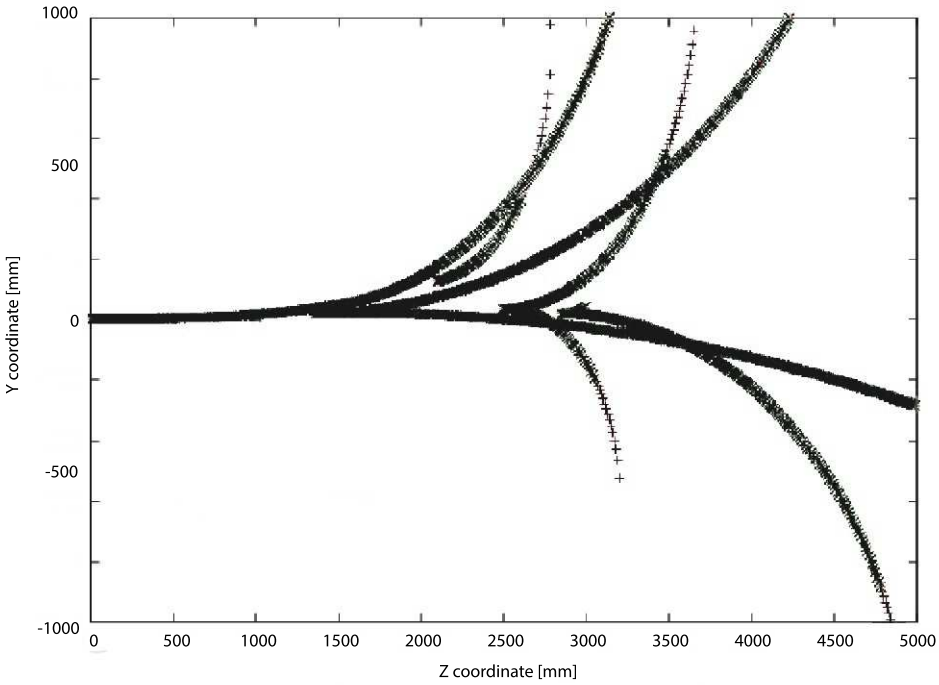

Figure 12: Reconstructed event on the Geant event. Black crosses with the noisy tail are the calculated points, while the full black parts are the Geant generated points.

5 Summary

As the GPUs are evolving, introducing new, more efficient architectures, it becomes easier to modify the existing applications and algorithms to be parallel. In this paper we were able to achive a 168 fold speed up compared to the CPU version, while computing the triplets of the trajectories. When calculating the full lines of the trajectories the system shows a 223 fold speed up in favor of the GPU.

In all cases the performance was definitely better on the GPU. On triplets $\mathtt{timeT}_{\mathtt{gpu}_{\mathtt{sh}_i}} << \mathtt{timeT}_{\mathtt{cpu}_i}$ and also for the lines $\mathtt{timeL}_{\mathtt{gpu}_i} << \mathtt{timeT}_{\mathtt{cpu}_i}, i \in [2000, 4000, 8000, 16000]$.

The performance of the GPUs make it possible to reconstruct a high volume of trajectories in parallel, finishing it in just a fraction on the runtime of the CPU.

References

[1] A. Agocs, Á. Fülöp, Jet reconstruction of individual orbits at many particles problems, *The 8th Joint Conference on Mathematics and Computer Science:MACS 2010*, Novadat Company, Komárno, Szlovákia 2011, pp. 123-138. ⇒172, 173, 174, 175

[2] R. Atkin, Review of jet reconstruction algorithms, *Journ. of Physics. Conf. Ser.* **645** (2015) 012008. ⇒173

[3] S. Catani, Yu.L. Dokshitzer, M. Olsson, G. Turnock and B.R. Webber, New clustering algorithm for multijet cross sections in e^+e^- annihilation, *Phys. Lett.*, **B269**, 3-4 (1991) 432-438. ⇒173

[4] S. D. Ellis, D. E. Soper, Successive combination jet algorithm for hadron collisions, *Phys. Rev. D* **48**, 7 (1993) 3160. ⇒173

[5] R. Forster, Á. Fülöp, Yang-Mills lattice on CUDA, *Acta Univ. Sapientiae, Inf.*, **5**, 2 (2013) 184–211. ⇒172, 174, 175

[6] M. E. Peskin, D. V. Schroeder, *Quantum Field Theory*, Westview Press, 1995. ⇒172

[7] S. Salur, Full Jet Reconstruction in Heavy Ion Collisions, *Nuclear Physics A* **830**, 1-4 (2009) 139c-146c. ⇒173

[8] T. Sjöstrand, S. Mrenna, P. Skands, A brief introduction to PYTHIA 8.1, *Computer Physics Communications* **178**, 11 (2008) 852-867. ⇒172

[9] Gy. Vesztergombi, Reflections about EXChALIBUR, the Exclusive 4π Detector, *Conf. "New Opportunities in the Physics Landscape at CERN"*, 2009. ⇒174

[10] CUDA C Programming Guide, NVIDIA Corp., 2016. ⇒172, 175, 177

[11] GEANT Detector Description and Simulation Tool, CERN Program Library Long Writeup, Geneva, 1993. ⇒172

Received: October 7, 2016 • Revised: November 9, 2016