Eur. Phys. J. Plus (2011) **126**: 1
DOI 10.1140/epjp/i2011-11001-5

THE EUROPEAN PHYSICAL JOURNAL PLUS

Regular Article

Optimization of HEP codes on GPUs

Applying NVIDIA's CUDA computing model to HEP software

M. Al-Turany[a]

GSI, Planckstrasse 1, 64291 Darmstadt, Germany

Received: 2 November 2010 / Revised: 19 November 2010
Published online: 7 January 2011 –

Abstract. The graphics processor units (GPUs) have evolved into high-performance co-processors that can be easily programmed with common high-level language such as C, Fortran and C++. Today's GPUs greatly outpace CPUs in arithmetic performance and memory bandwidth, making them the ideal co-processor to accelerate a variety of data parallel applications. Here, we shall describe the application of the data parallelism model supported in CUDA and GPUs with some example usage in high-energy physics software.

1 Introduction

In the last few years, the graphics processor units (GPUs) have moved away from the traditional fixed-function 3D graphics pipeline toward a flexible general-purpose computational engine. As demand arose for more flexibility, GPUs became ever more programmable, more powerful and cheaper. Efforts to exploit the GPU for non-graphical applications have been underway since 2003 [1]. The early efforts that used the graphics APIs for general-purpose computing were known as GPGPU (General-Purpose computation on GPUs). Early approaches to computing on GPUs cast computations into a graphics framework, allocating buffers (arrays) and writing shaders (kernel functions). In late 2006, NVIDIA introduced its "Compute Unified Device Architecture" (CUDA) to make data parallel computing on a GPU more straightforward [2]. Here, we shall describe the application of the data parallelism model supported in CUDA and the GPU with some example usage in high-energy physics software.

While the GPGPU model demonstrated great speedups, major drawbacks were still to be solved. First, it required the programmer to have a detailed knowledge of graphics APIs and GPU architecture. Second, non-graphical problems had to be expressed in terms of vertex coordinates, textures and shader programs (*e.g.* one has "to fool" the GPU), greatly increasing program complexity. Moreover, basic programming features such as random reads and writes to memory were not supported, greatly restricting the programming model. Lastly, the lack of support for double-precision-floating point arithmetic meant that some scientific applications could not be run on the GPU. With CUDA the picture changes dramatically, CUDA permits working with familiar programming concepts while developing softwares that can run on a GPU[3].

2 CUDA

CUDA (Compute Unified Device Architecture) is a GPGPU technology developed by NVIDIA that allows programmers to use the C/C++ programming language to code algorithms for execution on the GPU. It is a scalable parallel programming model and a software environment for parallel computing. CUDA, is freely available and the CUDA development tools work alongside the conventional C/C++ compiler, so one can mix GPU code with general-purpose code for the host CPU (fig. 1). CUDA automatically manages threads, *i.e.* does not require explicit management for threads in the conventional sense, which greatly simplifies the programming model. However, developers must analyze the data structure and determine how to divide the data into small chunks for distribution among the thread processors. The GPU is especially well suited to address problems that can be expressed as data-parallel computations, *i.e.* the same program is executed on many data elements in parallel. 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

[a] e-mail: m.al-turany@gsi.de

elements and has a high arithmetic content, the memory access latency can be hidden with calculations instead of big data caches.

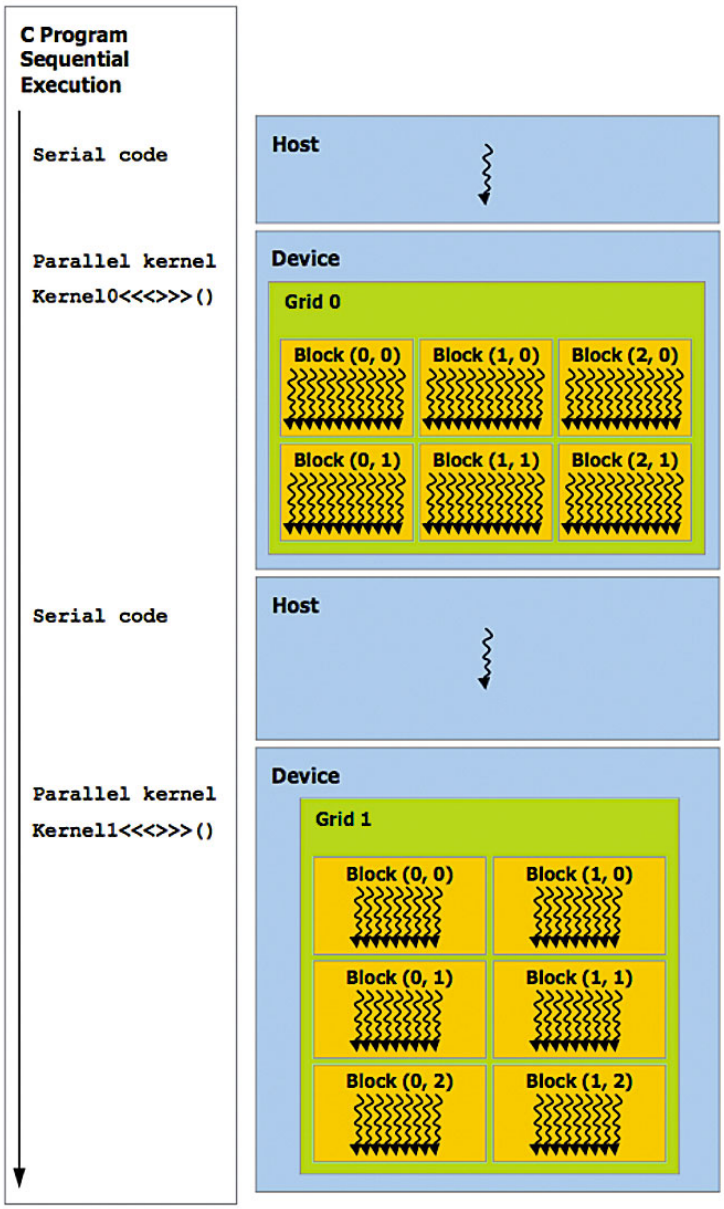

Fig. 1. Mixing CPU and GPU code.

CUDA extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. The CUDA toolkit provides a reasonable set of tools for C language application development. This includes:

- C/C++ compiler,
- Visual Profiler,
- CUDA-gdb debugger,
- GPU-accelerated BLAS (Basic Linear Algebra Subprograms)library,
- GPU-accelerated FFT (Fast Fourier Transform) library,
- GPU-accelerated Sparse Matrix library,
- GPU-accelerated RNG library (pseudorandom and quasirandom numbers generators),
- CUDA runtime driver (also available in the standard NVIDIA GPU driver),
- CUDA programming manual.

2.1 CUDA programming model

GPU is viewed as a compute device operating as a coprocessor to the main CPU (host). A CUDA program consists of several phases that are executed on either the host (CPU) or a device (GPU). The phases that exhibit little or no data parallelism are implemented in host code. Typically, a program supplies a single-source code encompassing both host and device code and the NVIDIA C Compiler (NVCC) separates the two. The host code is straight ANSI C code and is compiled with the host's standard C compilers and runs as an ordinary process, while the device code (kernel) is compiled by the NVCC and executed on a GPU device. Calling a kernel involves specifying the name of the kernel plus an execution configuration, *i.e.* defining the number of parallel threads in a group (thread block) and the number of groups to use when running the kernel for the CUDA device (grid) (fig. 2). Threads in a block can cooperate together, efficiently share data through the so-called shared memory. However, threads in different blocks in the same grid cannot directly communicate with each other.

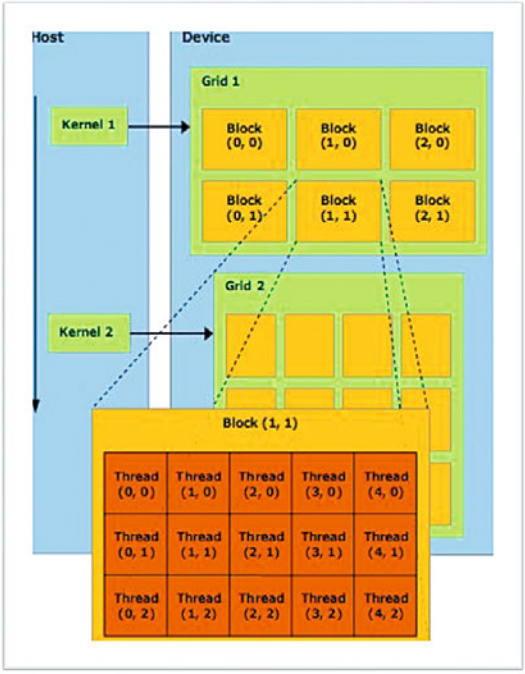

Fig. 2. Kernel execution on GPU.

2.2 CUDA memory model

CUDA exposes all the different types of memory on the GPU(fig. 3). During their execution, threads may access data from different memory spaces on a GPU device. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. Moreover, threads can access two additional read-only memory spaces; the constant and texture memory spaces. Global, constant, and texture memory spaces lie in the same physical memory, however they are optimized for different memory usages. The global memory is not cached (except for the newest generation of NVIDIA cards, *i.e.* FERMI). However, constant and texture memories have caches. Constant cache is as fast as read from register when all threads read the same address and slowdown linearly with number of different addresses read by all threads. The texture cache is optimized for 2D spatial locality, so threads of the same warp (the GPU multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*) that read texture addresses that are close together in 2D will achieve best performance. Reading device memory through texture fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

- address calculations are performed outside the kernel by dedicated units,
- packed data may be broadcast to separate variables in a single operation.

An example of texture memory usage is described below (3.2.1).

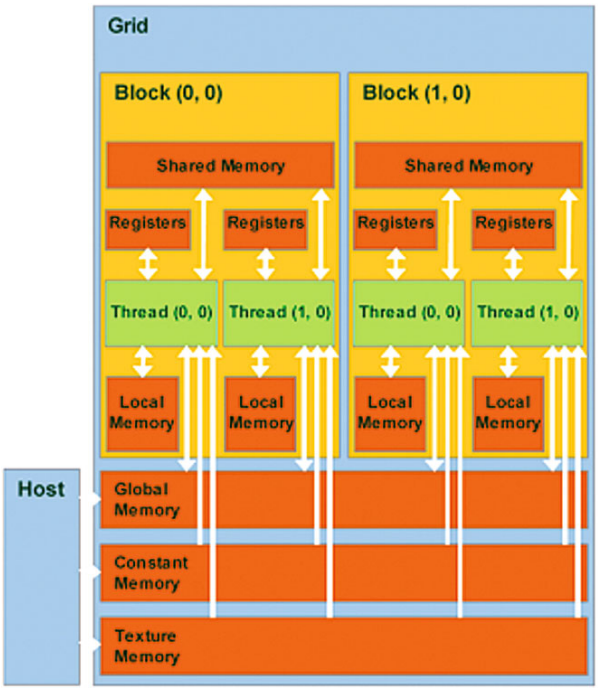

Fig. 3. Memory model of NVIDIAs GPU.

2.3 GPU threads and CPU threads

The main differences between GPU and CPU threads can be summarized as follows:

- GPU threads are extremely lightweight,
- CPUs can execute 1-2 threads per core, while GPUs can maintain more than 1000 threads per multiprocessor (1024 on the GTX 200 and 1536 on the GTX400 generation),
- CPUs use SIMD (single instruction is performed over multiple data) vector units, and GPUs use SIMT (single instruction, multiple threads) for scalar thread processing. SIMT does not require developers to convert data to vectors and allows arbitrary branching in threads.

3 Optimization of HEP codes on GPUs

One of the most important performance challenges facing CUDA developers is the best use of local multiprocessor memory resources such as shared memory, constant memory, and registers. The reason is that while global memory can deliver over 100 GB/s, this would translate to only 25 Giga Float/s (*e.g.*: a 32-bit floating-point value occupies 4 bytes) if the data are used only once, thus getting higher performance requires local data reuse. In the following subsections, two examples will be presented and discussed on porting some of particle physics experiment software to GPU.

3.1 Track and vertex fitting software

One of the track fitting algorithms used in the PANDA experiment [4] is a helix fit based on the work of Chernov *et al.*[5]. The algorithm was ported to CUDA [6]. This fitting was ported using different strategies, the main difference was the usage of the different memories of the GPU card. In this work a Tesla c1060 card is used [7]. The technical specifications of this card are in table 1. This card was used inside a PC with 2.5 GHz Intel Xeon Quad-Core and 16 GB memory.

3.1.1 Only global memory

The first implementation of the fitting code was written in a way that the track candidates (list of hits) where copied for all tracks in an event to the global memory of the GPU card. For each track a block of threads (32-threads)

Table 1. Tesla C1060 specification.

Number of Streaming Processor Cores	240
Frequency of processor cores	1.3GHz
Single Precision floating point performance (peak)	933 GFLOPS
Double Precision floating point performance (peak)	78 GFLOPS
Floating Point Precision	IEEE 754 single & double
Total Dedicated Memory	4GB GDDR3
Memory Speed	800MHz
Memory Interface	512-bit
Memory Bandwidth	102GB/sec
Max Power Consumption	187.8 W
System Interface	PCIe x16

calculated the fit parameter of the track. The results were copied back to the host memory (PC memory). We consider this first implementation as the *naive fitting* (see fig. 4). The data (track candidates, *e.g.*: all hits in one event) where simply copied to the global memory and the fitting loop where simply unrolled. The performance of the fitting was almost twice as slow as on single CPU. The fact that in this implementation all threads have to work on data in the global memory explains the slow-down compared to CPU implementation.

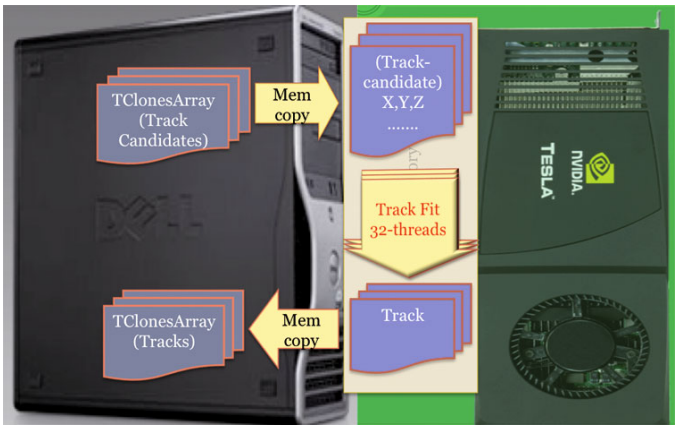

Fig. 4. Track fitting using only the global memory (Naive Fitting).

3.1.2 Implementation using shared memory

To use the shared memory, the list of track candidates (one complete event) is copied first to the global memory of the device then each thread block (32-threads in this example) copies one track candidate to its shared memory and works on it; after all tracks are fitted the results are copied back to the host (see fig. 5).

The major difference to the naive fitting (sect. 3.1.1) is that the data is now in the shared memory (very fast access) and also the copy of the data from global to shared memory is done on demand by all threads in a block at once. The GPU starts a certain number of blocks simultaneously: this number depends on the block size, number of multiprocessors, etc [8]. Different blocks (in this example each block fits one track) need different time, and whenever a block is finished the GPU starts the next one in case the number of blocks to run is more than what the GPU can run simultaneously. Thus, one could face the situation that at the time where one block is copying data, another one is fitting another track and a third one is already writing the results back to the global memory.

3.1.3 Implementation using page-locked memory

CUDA runtime provides functions to allocate and free page-locked (also known as pinned) host memory as opposed to regular pageable host memory allocated by *malloc*. Using page-locked host memory has several benefits: the bandwidth between host memory and device memory is higher if the host memory is allocated as page-locked and even higher if,

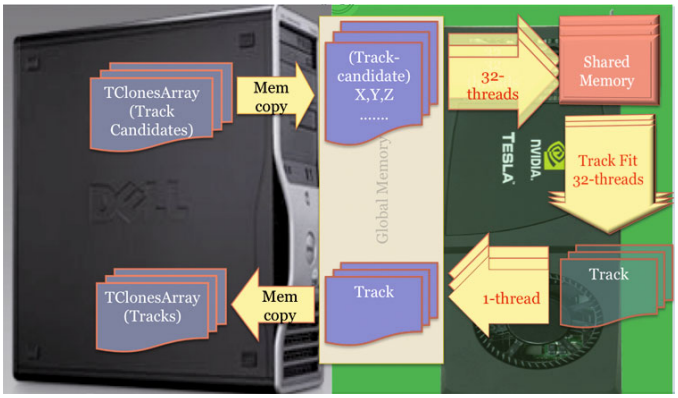

Fig. 5. Track fitting using shared memory.

in addition, it is allocated as write-combining [8]. Copies between page-locked host memory and device memory can be performed concurrently with kernel execution. Moreover, page-locked host memory can be mapped into the devices address space, eliminating the need to copy it to or from the device memory.

In this implementation only the memory allocation is different from sect. 3.1.2. The track candidates are copied to the page-locked memory and then each thread block reads its data directly to the shared memory. In this concept the data is not moved first to the global memory as before. The results from this implementation compared to that in sect. 3.1.2 are summarized in table 2.

Table 2. Time in ms needed to fit all tracks in one event.

Track/event	50	100	1000	2000
CPU	3.0	5.0	120	220
GPU (Shared Memory)	1.0	1.2	6.5	12.5
GPU (Pinned Memory)	0.2	0.4	5.4	10.5

3.1.4 Track and vertex fitting: conclusion

The speed-up for different event sizes and configurations is summarized in table 3.

Table 3. CPU/GPU time.

Track/event	50	100	1000	2000
GPU (Shared Memory)	3.0	4.2	18	18
GPU (Pinned Memory)	15	13	22	21

Using GPUs for track/vertex fitting we were able to get a factor 20 in performance compared to the CPUs (track fitting alone even a factor 60 [6]), however, one has to determine how to divide the data into small chunks for distribution among the thread processors. Depending on the data sizes and the nature of the problem the use of pinned memory can improve the performance by factors. *I.e.*, this comes from the fact that the data is now copied asynchronously to the working-threads shared memory; for small event size this will hide the copying latency more efficiently than the normal procedure (sect. 3.1.2).

3.2 Track parameter propagation

Track parameter propagation has often been handled by the Runge-Kutta-Nystroem method [9], which is designed to solve second-order differential equations, such as the equation of motion of a particle in a magnetic field. The basic idea of this method is to divide the integration interval into steps and solve these independently. The Geant3 [10]

algorithm based on Runge-Kutta method for solving the kinematic equations of charged particles in a magnetic field was ported to CUDA. The algorithm itself is hardly parallelizable, however, one can propagate all tracks in an event in parallel. For each track, a block of 8 threads is created, the particle data is copied by all these 8 threads at once, then one thread does the propagation. The use of 8 threads is only meant to accelerate the copy process between the global and shared memory. However, the major problem for parallelization of track propagation is the usage of the field map. Field maps are typically used as lockup tables with some interpolation algorithms for values between the measured (or calculated) points of the table. For performance and multi-access issues, one can parameterize (fit) the field map [11], however, this has some drawbacks: fits are specific for certain maps and must be refitted whenever the map changes (something which can happen in a running experiment or always happens during the design phase of an experiment); it is also hard to do with good accuracy. Finally fitting is not always possible for all maps.

3.2.1 Field map in texture memory

Field maps are usually used as three-dimensional look-up tables with some interpolation algorithm to give the field value between the points. The distance between the points is usually chosen so that a linear interpolation is accurate enough to give reasonable values for the field. In this work the dipole field from the PANDA [12] experiment is used. The field map (three-dimensional array) is bound to the texture memory of the device, where the field values are accessible from all threads in the grid. Moreover, the linear interpolation of the field is done by a dedicated hardware [8]. The out-of-range texture coordinates are set to CLAMP mode [8]: *i.e.* out-of-range texture coordinates are clamped to the valid range. (Values below 0 are set to 0 and values greater or equal to N are set to $N - 1$.)

3.2.2 Hardware

In this work we tried to test different hardware with different numbers of cores in order to investigate also the scalability of the code in this situation. The specification of the hardware (GPU devices) used in this work are summarized in table 4.

Table 4. Hardware used.

Card	Quadro NVS 290	GeForce 8400 GT	GeForce 8800 GT	Tesla C1060	Fermi GTX 480
CUDA Cores	16 (2×8)	32 (4×8)	112 (14×8)	240 (30×8)	480 (15×32)
Memory (MB)	256	128	512	4096	1536
Frequency (GHz)	0.92	0.94	1.5	1.3	1.4
Compute capability	1.1	1.1	1.1	1.3	2.0
Warps/Multiprocessor	24	24	24	32	48
Max. No. of threads	1536	3072	10752	30720	23040
Max Power Consumption (W)	21	71	105	200	250

3.2.3 Results

To make the test, 1 GeV protons were generated and sent with different starting angles through the field. Different events were generated by changing the number of protons per event. The tracks (protons) in each event were propagated at once through the dipole field (1.5 meters distance between starting and final plane). The results obtained for different cards and events are summarized in table 5.

The gain in performance for different cards and events is summarized in table 6. The gain was calculated by dividing the CPU time over the GPU time. From table 6, one can see that the performance scales as expected for all numbers of cores and sizes of events except for the Fermi card which even though has twice as much cores as the Tesla, performs much better than the Tesla for small events, but almost as the Tesla for large events. To understand this discrepancy, one has to consider table 7 which presents the resource usages of the different cards for this example. The last row in table 7 shows the maximum numbers of tracks that can be propagated in parallel. For Tesla and Fermi this number is the same, but for small events the limiting factor is the memory which is much faster in the case of Fermi. For large events the memory operations are hidden by the calculation time which is almost the same for both cards (row two in table 7).

Table 5. Time in ms needed to propagate all tracks in one event.

Track/Event	CPU	Quadro NVS 290	GeForce 8400 GT	GeForce 8800 GT	Tesla C1060	Fermi GTX 480
10	2.4	0.9	0.8	0.7	0.4	0.18
50	11	2.5	1.8	1.0	0.4	0.21
100	21	4.4	2.9	1.7	0.5	0.25
200	42	8.9	5.6	2.9	0.9	0.41
500	104	23	13.2	5.6	1.3	0.75
1000	210	42	25.7	10.1	1.9	1.30
2000	412	82	52.2	19.5	3.0	2.40
5000	1054	200	125	50.0	6.0	5.70

Table 6. Gain in performance for different cards.

Track/Event (CUDA Cores)	Quadro NVS 290 16	GeForce 8400 GT 32	GeForce 8800 GT 112	Tesla C1060 240	Fermi GTX 480 480
10	3	3	3.5	6	13
50	4.4	6	11	28	52
100	4.8	7.3	12.3	47	84
200	4.8	7.5	14.5	49	102
500	4.5	7.9	18.5	80	138
1000	5	8.1	21	111	161
2000	5	8	21	137	171
5000	5	8.4	21	175	184

Table 7. Resource usage in different cards.

	Quadro NVS 290	GeForce 8400 GT	GeForce 8800 GT	Tesla C1060	Fermi GTX 480
MultiProcessor	2	4	14	30	15
Frequency (GHz)	0.92	0.93	1.5	1.3	1.4
Compute capability	1.1	1.1	1.1	1.3	2.0
Active Blocks/Multiprocessor	2	2	2	4	8
Tracks propagated in parallel	4	8	28	120	120

4 Conclusion

CUDA permits working with familiar programming concepts while developing softwares that can run on a GPU. By using GPUs for certain problems in physics experiments, one can win orders of magnitudes in performance compared to the CPUs, however one has to choose carefully on which level the parallelization should take place and how to divide the data into small chunks for distribution among the thread processors (GPUs). For track propagation (sect. 3.2) we choose to parallelize on track level, by track/vertex fitting (sect. 3.1) we choose to parallelize on hit level. Finally, understanding the different memory regions of the GPU is also crucial for getting better performance and help simplifying some problems.

References

1. M. Pharr, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (Addison-Wesley, 2005).
2. NVIDIA, `http://developer.nvidia.com/object/cuda.html`.
3. CUDA, Supercomputing for the Masses, `http://www.ddj.com/hpc-high-performance-computing/`.
4. `http://www-panda.gsi.de/`.
5. N. Chernov, G. Ososkov, Comput. Phys. Commun. **33**, 329 (1984).
6. M. Al-Turany, F. Uhlig, R. Karabowicz, J. Phys.: Conf. Ser. **219**, 042001 (2010).
7. Tesla C1060 Computing Processor, `http://www.nvidia.com/object/product_tesla_c1060_us.html`.
8. Nvidia, *NVIDIA CUDA Programming Guide*, 2.3.1 edition, 8 (2009).

9. Handbook of National Bureau Of Standards, procedure 25.5.20.
10. *GEANT - Detector Description and Simulation Tool*, March 1995 edition (CERN, Geneva, 1993).
11. S. Gorbunov, I. Kisel *An analytic formula for track extrapolation in an inhomogeneous magnetic field*, CBM-SOFT-NOTE 001, GSI, March 2005.
12. J.G. Messchendorp, Strong interaction studies with PANDA, arXiv:1001.0272v1.