Journal of Physics: Conference Series **219** (2010) 042001 doi:10.1088/1742-6596/219/4/042001

GPU's for event reconstruction in the FairRoot Framework

M. Al-Turany
Planckstr.1, 64291 Darmstadt, Germany
E-mail: m.al-turany@gsi.de

F. Uhlig
Planckstr.1, 64291 Darmstadt, Germany
E-mail: f.uhlig@gsi.de

R. Karabowicz
Planckstr.1, 64291 Darmstadt, Germany
E-mail: R. Karabowicz@gsi.de

Abstract. FairRoot is the simulation and analysis framework used by CBM and PANDA experiments at FAIR/GSI. The use of graphics processor units (GPUs) for event reconstruction in FairRoot will be presented. The fact that CUDA (Nvidia's Compute Unified Device Architecture) development tools work alongside the conventional C/C++ compiler, makes it possible to mix GPU code with general-purpose code for the host CPU, based on this some of the reconstruction tasks can be send to the graphic cards. Moreover, tasks that run on the GPU's can also run in emulation mode on the host CPU, which has the advantage that the same code is used on both CPU and GPU.

1. Introduction

The FairRoot framework [1, 2, 3], is an object-oriented simulation, reconstruction and data analysis framework based on ROOT [4], and the Virtual Monte-Carlo (VMC) interface [5]. It includes core services for detector simulations and offline analysis. The framework, is designed to optimize the accessibility for beginning users and developers, to be flexible (i.e. able to cope with future developments), and to enhance synergy between the different physics experiments at/or outside the FAIR project. FairRoot supports many systems and compilers. The nightly build system compiles and test the status of the project on many different platforms. The accumulated results from each of these platforms are sent to a web server. The results will be displayed on dashboards where all developers can immediately see if there is any problem on any of the different systems. These tasks are achieved by using the open source tools CMake, CTest and CDash [7] with a set of macros that define certain tests. In figure 1 the different implementations of FairRoot at different experiments are shown [1, 3, 6, 8].

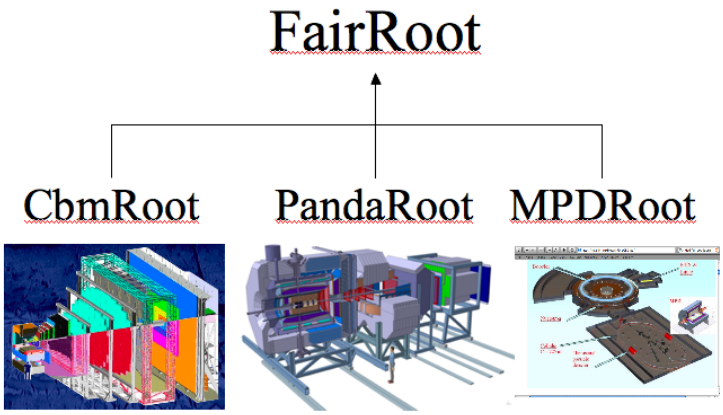

Figure 1. FairRoot at different experiments

2. GPU's and CUDA

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. Moreover they are getting cheaper and more powerful [9]. With the Nvidia Compute Unified Device Architecture (CUDA) [10], one can get orders-of-magnitude performance increases over standard multi-core processors, while programming with a high-level language such as C [9].

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 (figure 2). 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 data structure and determine how to divide the data into smaller 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 elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches

2.1. CUDA

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:

- nvcc C compiler
- CUDA FFT and BLAS libraries for the GPU
- Profiler
- gdb debugger for the GPU
- CUDA runtime driver (also available in the standard NVIDIA GPU driver)
- CUDA programming manual

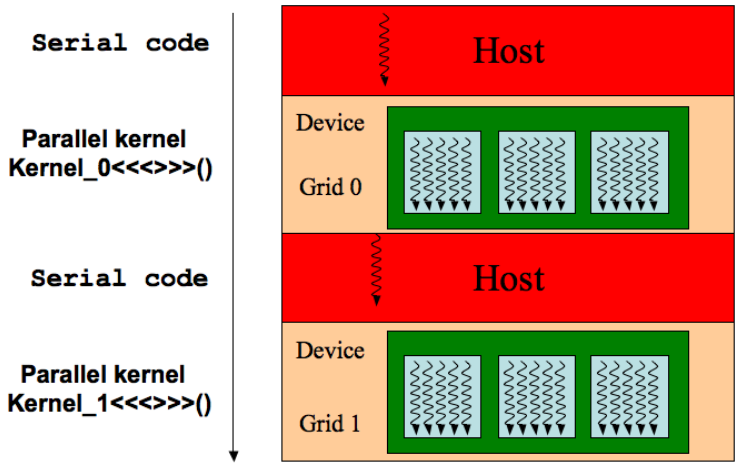

Figure 2. Mixing CPU and GPU code

2.2. GPU threads and CPU threads
The main differences between GPU and CPU threads can be sumarized as following:

- GPU threads are extremely lightweight
- CPUs can execute 1-2 threads per core, while GPUs can maintain up to 1024 threads per multiprocessor (8-core)
- 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.

2.3. CUDA integration into FairRoot
The integration of CUDA into FairRoot is done in two steps, first integrating the build process and second the library, this was achieved as following:

2.3.1. Building System Using FindCuda.cmake [11] CUDA is integrated into FairRoot building system very smoothly. The users do not have to take care of Makefiles or which compiler should be called (e.g. NVCC or GCC).

2.3.2. FairCuda: ROOT interface An interface is implemented which enables the use of GPU's implemented function from within a ROOT CINT session. The CUDA implemented kernels are wraped by a class (FairCuda) that is implemented in ROOT and has a dictionary. From a ROOT CINT session the user simply call the wraper functions which call the GPU functions (kernels).

3. Track fitting with CUDA
The design of FairRoot [2] allows exchanging algorithms (Tasks) easily (figure 3), the track fitting task in PandaRoot [6] can be replaced at runtime by a GPU Task, the results discussed below where obtained using these tasks.

3.1. Hardware
In this work a Tesla c1060 card is used [12]. The technical specification of this card are in table 3.1. This card was used inside a PC with 2.5 GHz Intel Xeon Quad-Core and 16 GB memory. The

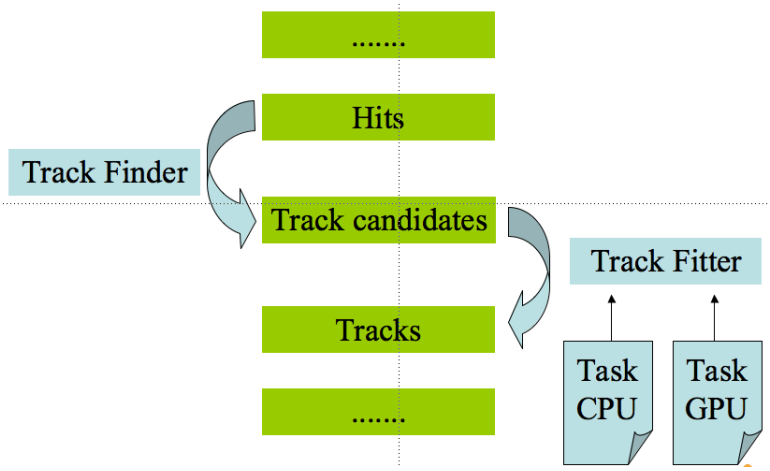

Figure 3. Reconstruction chain in PandaRoot

Tesla products are intended for the high performance computing, the main difference between Tesla products and ordinary video cards is the lack of ability to output images to a display. The primary function of Tesla products are to aid in large scale calculations for professional and scientific fields, with the use of CUDA.

Form Factor	10.5" x 4.376", Dual Slot
Number of Tesla GPUs	1
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
Auxiliary Power Connectors	6-pin & 8-pin
Thermal Solution	Active fan sink
Software Development Tools	C-based CUDA Toolkit

Table 1. Tesla C1060 specification

3.2. Test configuration

The Panda detector simulation was used for this test (figure 4). One GeV Muons were transported through the central tracker of the Panda detector. Events were created with different numbers of primary tracks (50, 100,1000 and 2000 tracks per event) and in each run 50 events where transported. Moreover, ideal track finding was used (simply copying the Geant tracks to the track candidates objects)

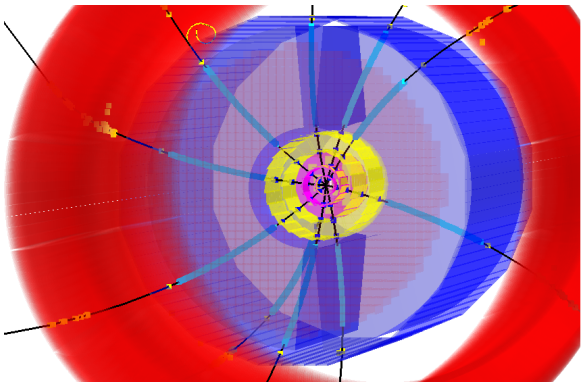

Figure 4. Parts of the Panda Detector used in the test

3.3. Track fitting implementation with CUDA

One of the track fitting algorithms used in Panda experiment is a helix fit based on the work of Chernov et. al. [13]. The algorithm was ported to CUDA and used from a task using the FairCuda interface described in section 2.3.2 .

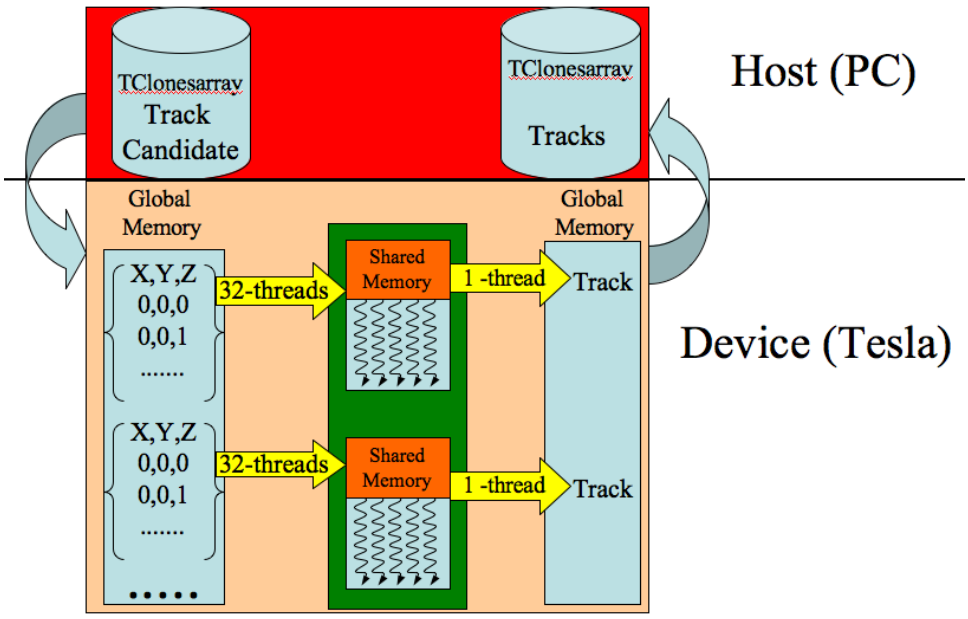

Figure 5. Track fitting schematic

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 100GB/s, this would translate to only 20GF/s for single-touch use of data, thus getting higher performance requires local data reuse. To realize this the list of track candidates where copied first to the global memory of the device then each thread block (32-threads in this example) copied the track candidates to its shared memory and worked on it, the results were then copied back to the global memory and finally back to the host.

4. Results

The track fitting was compared in four different modes:

(i) GPU Float: The algorithm is implemented for GPU in single precision accuracy
(ii) GPU Double: The algorithm is implemented for GPU in double precision accuracy
(iii) CPU : original algorithm
(iv) GPU Emulation: The GPU implementation runs on the CPU in GPU emulation mode.

The results obtained from the different runs are summarized in table.2 and figure 6

Mode/(Track/event)	50	100	1000	2000
GPU (Emu)	6.0	15.0	180	370
CPU	3.0	5.0	120	220
GPU (Double)	1.2	1.5	3.2	5.0
GPU (Float)	1.0	1.2	1.8	3.2

Table 2. Results: Time in ms needed to fit the corresponding number of tracks using the different algorithms and/or mode

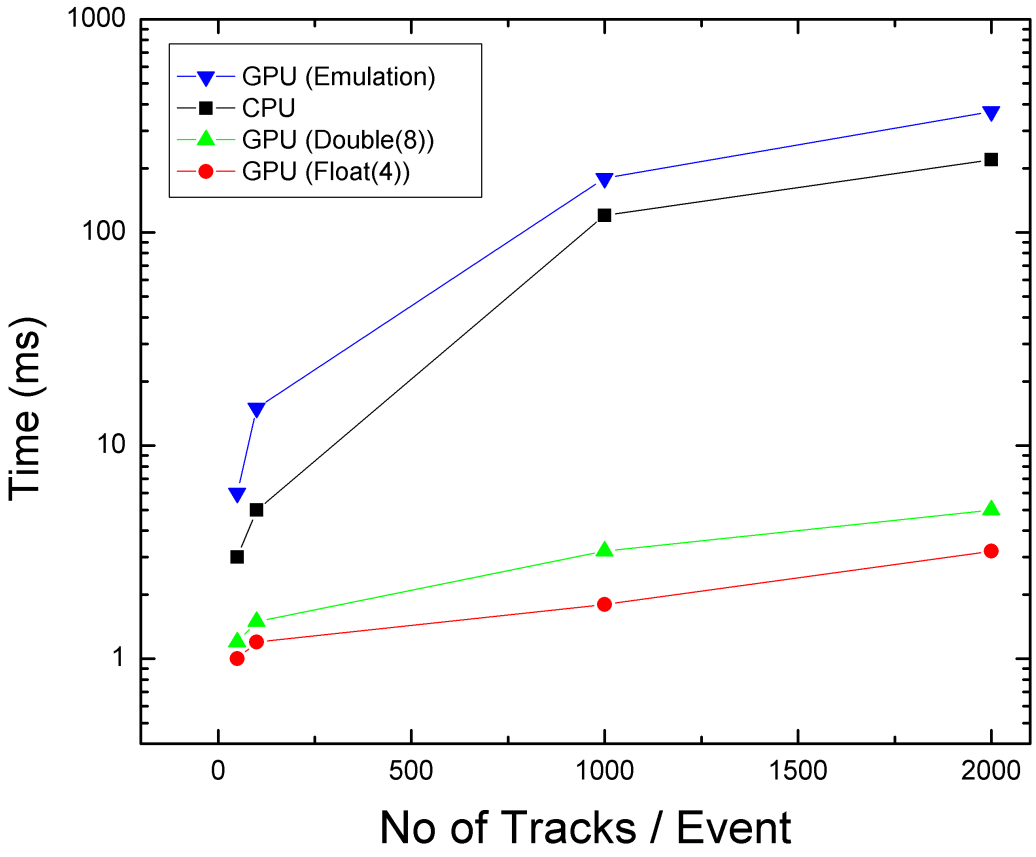

Figure 6. Time in ms needed to fit the corresponding number of tracks using the different algorithms and/or mode

From table 2 one can see that the emulation mode is almost twice slower than the native CPU code. In fact when running in device emulation mode, the programming model is emulated by the runtime. For each thread in a thread block, the runtime creates a thread on the host. The overhead of creating these threads is the reason for the performance lost. However one should keep in mind that this mode is meant for debugging.

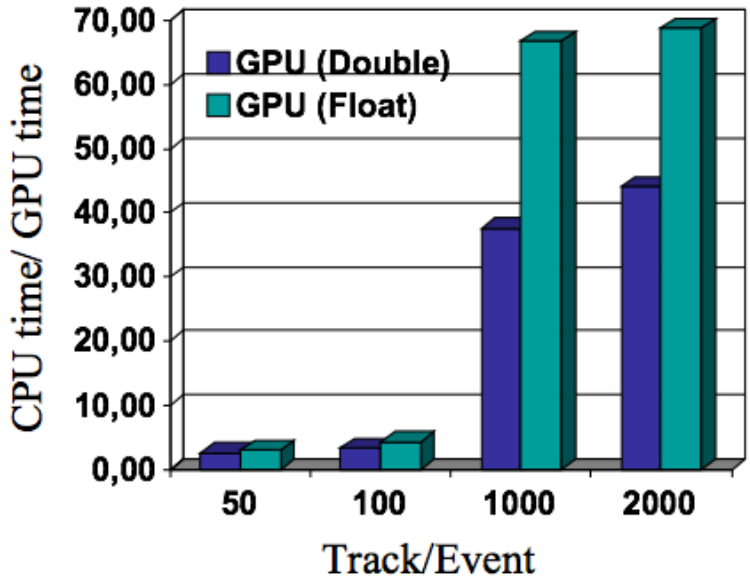

Figure 7. Ratio between time needed on CPU to the time needed by GPU for track fitting

The comparison between GPU and CPU time is summarized in figure 7. from this figure one can see that one can gain almost a factor 70 in this example. However using the "CUDA Occupancy Calculator" (i.e. a programmer tool that allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel [10]) we found that in this example we are using only 25 % of the hardware, which explain the difference between double and float calculations. i.e. using the full capacity of the Tesla card one expect to have about a factor ten in performance between float and double (see Table 1).

However to get a 100% occupancy one should start 4 blocks/multiproccessor each with 256 thread/block, unfortunately such a configuration is not possible for this example (i.e. track candidates has a max. of 32 points, so starting more threads will not help, a single point cannot be handled by more than one thread). The fact that the Tesla card supports concurrent access means that different CPU threads can start different kernels on the device. One could use PROOF [14]) and CUDA together for problems which cannot use the full capacity of the cards.

5. Future and ongoing work

The concurrent access to the Tesla card from different CPU processes is under test and the first results will be published soon. However the integration of PROOF into FairRoot still have to be finished. Moreover some of the recent new features of CUDA could force a complete redesign of the interface to ROOT. i.e in CUDA 2.3 [10] the runtime provides functions to allocate and free page-locked (also known as pinned) host memory. Using page-locked host memory has several benefits (see CUDA 2.3 Programming Guide [10] for more details):

(i) Bandwidth between host memory and device memory is higher if host memory is allocated as page-locked and even higher if in addition it is allocated as write- combining.

(ii) Copies between page-locked host memory and device memory can be performed concurrently with kernel execution.

(iii) Page-locked host memory can be mapped into the devices address space, eliminating the need to copy it to or from device memory.

NVIDIA's new Fermi graphics architecture (next generation CUDA architecture), announced at the NVIDIA's GPU Technology Conference [15], will run real C++ applications, the company has confirmed. Template support, and pointers are promised. However, it is also stated that the generic C++ code will not magically benefit from being compiled under CUDA. And some extensions (as in the case of C) are there and has to be used. Such features will also have a great impact to the whole design of the software, and we expect that it will make it even easier to use GPUs in more and different regions as it is already now.

6. Conclusion

In contrary to GPGPU, which usually means, writing software for a GPU in the language of the GPU CUDA permits working with familiar programming concepts while developing software that can run on a GPU, Moreover CUDA compile the code directly to the hardware (GPU assembly language, for instance), thereby providing great performance. Using GPUs for track fitting one can win orders of magnitudes in performance compared to the CPUs, however one has to determine how to divide the data into smaller chunks for distribution among the thread processors (GPUs)

7. References

[1] M. Al-Turany, D. Bertini, and I. Koenig. CbmRoot: Simulation and analysis framework for CBM experiment. In S. Banerjee, editor, *Computing in High Energy and Nuclear Physics (CHEP-2006)*, volume 1 of *MACMILLAN Advanced Research Series*, pages 170–171. MACMILLAN India, 2006.

[2] D. Bertini, M. A-Turany, I. Koenig, and F. Uhlig. The fair simulation and analysis framework. In *International Conference on Computing in High Energy and Nuclear Physics (CHEP'07)*, volume 119 of *Conference Series*. IOP Publishing, 2008.

[3] M. Al.-Turany FairRoot: http://fairroot.gsi.de.

[4] R. Brun and F. Rademakers. Root - an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research A*, 389:81–86, Sep. 1997.

[5] R. Brun, F. Carminati, I. Hrivnacova, and A. Morsch. Virtual Monte-Carlo. In *Computing in High Energy and Nuclear Physics*, pages 24–28, La Jolla, California, 2003.

[6] PANDA Computing Group in GSI Sci. Rep. 2006, FAIR-EXPERIMENTS-02

[7] I. Kitware. Cmake: www.cmake.org.

[8] http://nica.jinr.ru/

[9] CUDA, Supercomputing for the Masses, http://www.ddj.com/hpc-high-performance-computing/

[10] NVIDIA http://developer.nvidia.com/object/cuda.html

[11] Abe Stephens http://www.sci.utah.edu/ abe/FindCuda.html

[12] Tesla C1060 Computing Processor, http://www.nvidia.com/object/product_tesla_c1060_us.html

[13] N. Chernov and G. Ososkov, Comp. Phys. Commun. 33 (1984) 329.

[14] The Parallel ROOT Facility, PROOF. G. Ganis, F. Rademakers and Jan Iwaszkiewicz http://root.cern.ch/drupal/content/proof

[15] Nvidia's GPU Technology Conference http://www.nvidia.com/object/gpu_technology_conference.html