PROCEEDINGS OF SCIENCE

Applying CUDA Computing Model To Event Reconstruction Software

Mohammad Al-Turany*
GSI DARMSTADT
E-mail: m.al-turany@gsi.de

Florian Uhlig
GSI DARMSTADT
E-mail: f.uhlig@gsi.de

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. With the Nvidia Compute Unified Device Architecture (CUDA), one can get orders-of-magnitude performance increases over standard multi-core processors, while programming with a high-level language such as C. In this work we implement some parts of the the event reconstruction algorithms in CUDA and compared the performance and scalability on different NVIDIA cards.

13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research
February 22-27, 2010
Jaipur, India

*Speaker.

PoS(ACAT2010)014

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. Moreover they are getting cheaper and more powerful [1]. With the Nvidia Compute Unified Device Architecture (CUDA) [2], one can get orders-of-magnitude performance increases over standard multi-core processors, while programming with a high-level language such as C [1].

2. CUDA

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 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 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

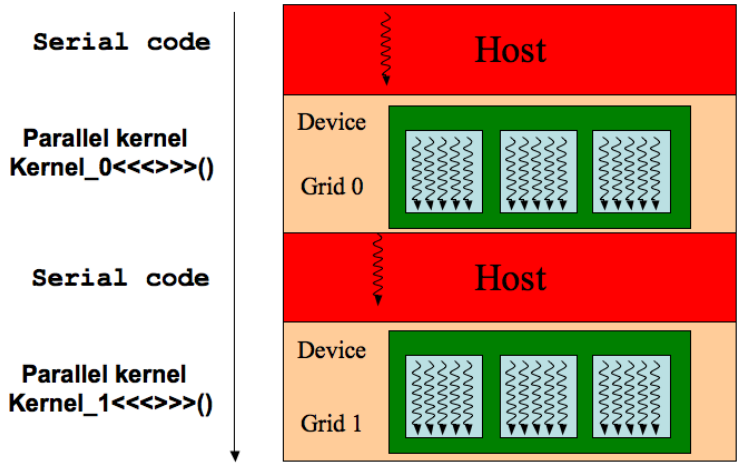

Figure 1: Mixing CPU and GPU code

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:

- 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

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) (figure 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.

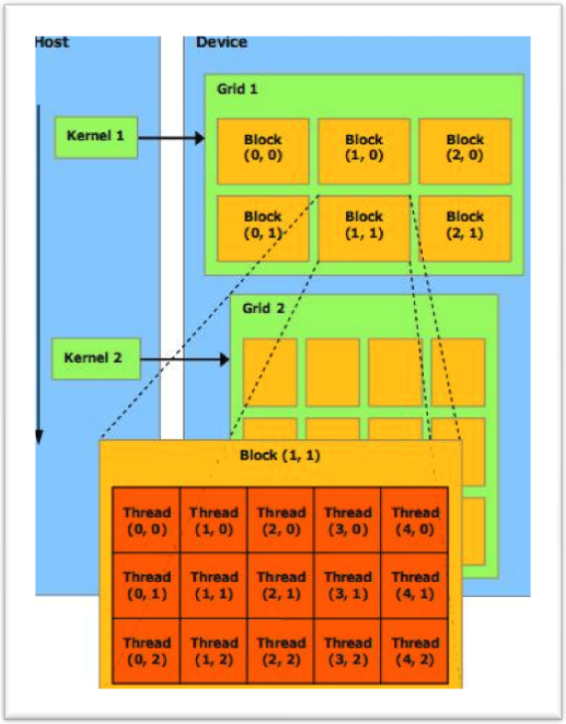

Figure 2: Kernel execution on GPU

2.2 CUDA memory model

CUDA exposes all the different types of memory on the GPU(figure 3):

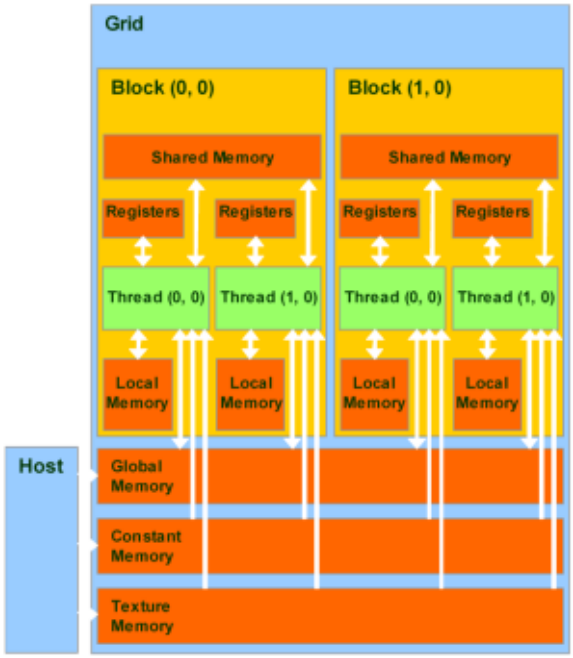

Figure 3: Memory model of NVIDIAs GPU

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. Constant and texture memories are read-only and accessible by all threads in a grid.

2.3 GPU threads and CPU threads

The main differences between GPU and CPU threads can be summarized 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.

3. FairRoot

The FairRoot framework [3, 4, 5], is an object-oriented simulation, reconstruction and data analysis framework based on ROOT [6] and the Virtual Monte-Carlo (VMC) interface [7]. 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.

3.1 CUDA integration into FairRoot

The integration of CUDA into FairRoot is done in two steps, which will be describe in the following sections.

3.1.1 Building System

Using FindCuda.cmake [8] 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). The script will search for CUDA toolkit and SDK installation using several standard paths and the environment variables which are created by the CUDA installer. Depending on the results of the search the building system will include the CUDA files in the build process or not. Optionally one can also include the CUDA files without even having the hardware (emulation mode [9]).

3.1.2 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).

4. Track propagation

4.1 Runge-Kutta propagator

The Geant3 algorithm based on Runge-Kutta method for solving the kinematic equations (Nystroem algorithm [10]) was ported to CUDA. The algorithm it self 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 do the propagation. The use of 8 Threads is only meant to accelerate the copy process between the global and shared memory.

4.2 Field map in Texture memory

Field maps are typically 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 so chosen that a linear interpolation is accurate enough to give reasonable values for the field. In this work the dipole field in the PANDA [11] experiment was used. The field map (three dimensional

array) is bind to the texture memory of the device, in this memory the field is accessible from all threads in the grid. Moreover, the linear interpolation of the field is done by a dedicated hardware [9]. The out of range texture coordinates is set to CLAMP mode [9], 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).

5. Hardware

Cards with different number of cores (from 16 up to 240 cores) were used to investigate the scalability of the code. The specification of the hardware (GPU devices) used in this work are summarized in table 1.

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

Table 1: Hardware used

6. Results

To make the test, protons with 1 GeV where generated and send with different starting angles through the field. Different events where generated by changing the number of protons per event. The tracks (protons) in each event where propagated at once through the dipole field (1.5 meter distance between starting and final plane). The results obtained for different cards and events are summarized in table 2.

The Gain in performance for different cards and events is summarized in table 3. The gain was calculated by dividing the CPU time over the GPU time.

7. Conclusion

CUDA permits working with familiar programming concepts while developing software that can run on a GPU. Using GPUs for track propagation 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 smaller chunks for distribution among the thread processors (GPUs). In this work we choose to parallelize on track level, by our previous

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

Table 2: Time in ms needed to propagate all tracks in one event

Track/Event	Qaudro NVS 290	GeForce 8400 GT	GeForce 8800 GT	Tesla C1060
10	3	3	3.5	6
50	4.4	6	11	28
100	4.8	7.3	12.3	47
200	4.8	7.5	14.5	49
500	4.5	7.9	18.5	80
1000	5	8.1	21	111
2000	5	8	21	137
5000	5	8.4	21	175

Table 3: Gain in performance for different cards

work [12] we choose to parallelize on hit level for track fitting. Finally understanding the different memory regions of the GPU is also crucial for getting better performance and help in simplifying some problems.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] NVIDIA CUDA Programming Guide, Version 2.3

[10] Handbook Nat. Bur. Of Standards, procedure 25.5.20

[11] J.G. Messchendorp, http://arxiv.org/abs/1001.0272v1

[12] M. Al-Turany, F. Uhlig and R. Karabowicz GPUs for event reconstruction in the FairRoot Framework *Journal of Physics: Conference Series 219 (2010) 042001*