CUDA Tutorial 00

What is CUDA and how may it serve you?

Martin A. Kruse - July 2013

This document is part of a series of tutorials intended for the people at the extraterrestrial physics research group at the Christian Albrechts Universität zu Kiel. Everyone else is invited to use and distribute (as is) these documents, however the computers mentioned here are not available to the public.

The full series of tutorials can be found online at
http://www.ieap.uni-kiel.de/et/people/kruse/,
though access to this site later than 2026 might
be impossible due to thermonuclear war.

Suggestions and corrections are very welcome at kruse@physik.uni-kiel.de.

The short answer to the question above: CUDA is a computational framework which utilizes the processing power of NVIDIA's graphics cards and makes their infrastructure (partially) available to the programmer by very simple extensions of high-level programming languages (C/C++, Fortran, Python). If you are somewhat familiar with one or more of these languages, programming CUDA devices is a piece of cake.

Computations in computer graphics involve a great number of arithmetic calcutations with few to no need to execute divergent instructions (few if-statements). This fact resulted in graphics processing units with many (up to a few thousand) relatively slow lightweight arithmetical logical units compared to the heavy and very fast units found in standard CPUs. There are many applications outside of computer graphics which can utilize that architecture efficiently, and many physics applications can be sped up significantly with CUDA.

The rule of thumb: If you have a great deal of calculations that can be done mostly independently from each other, you might want to consider programming with CUDA. A very good example is the evaluation of a few equations on hundreds or thousands of grid points as in computational fluid dynamics. Another is the simulation of many particle systems.

In this tutorial you will find an introduction to CUDA and the underlying architecture of graphics cards. It will be relatively simple so that you have a good idea what CUDA is and how you might utilize it in your applications without going to much into detail. If you are interested in a more complete introduction, I recommend getting familiar with the "Get Started" ressources from NVIDIA (https://developer.nvidia.com/get-started-cuda-cc). The link directs you to the ressources concerning C/C++, which will also be used during these tutorials. Somewhere on that site you will find ressources for Fortran and Python as well.

1 The guts of your graphics card

Your graphics card is an extension of your PC which is responsible for creating the visual experience you encounter when using your computer. Everything you see on your monitor, apart from the boot process or your terminal-only session, is brought to you by your graphics card. The purpose of the *graphics processing unit* (GPU) is to relieve the *central processing unit* (CPU) by utilizing specialized hardware adapt to graphics processing. Consider your graphics card as a specialized computer insider your universal computer, the PC.

From now on, only NVIDIA GPUs are considered as CUDA is a NVIDIA-specific platform. The hardware in other vendors' graphics cards are very similar, though some parts have different names. The GPU is composed of a small number of so called *Streaming Multiprocessors* (SMs), whiche house the arithmetic units, the CUDA cores. Every graphics card has its own memory (RAM or VRAM), which is typically situated near the GPU itself. Communication is possible between the CPU and GPU as well as between the RAM of the PC (the *host*) and the graphics card (the *device*). Figure 1 depicts this paradigm.

1.1 Streaming Multiprocessor

For the programmer, the Streaming Multiprocessor is probably the most interesing hardware component of the device. It consists of a number of arithmetic (CUDA) cores, which do the actual computations, on-chip (shared) memory, memory controllers for access to off-chip (global) memory, local and very fast (register) memory, schedulers for the great number of threads to be managed and a bunch of other components which are of minor interest for this introduction. For all intents and purposes, you can consider a SM as a computer of its own. The exact number of components of one such SM is dependant on the compute capability of the device (see the following section), and the number of SMs on one device varies. For instance, on one of the GTX Titans which are resident in our CUDA computer Prometheus, there are 14 SMs (for some marketing reasons dubbed SMX by NVIDIA) as depicted in figure 2.

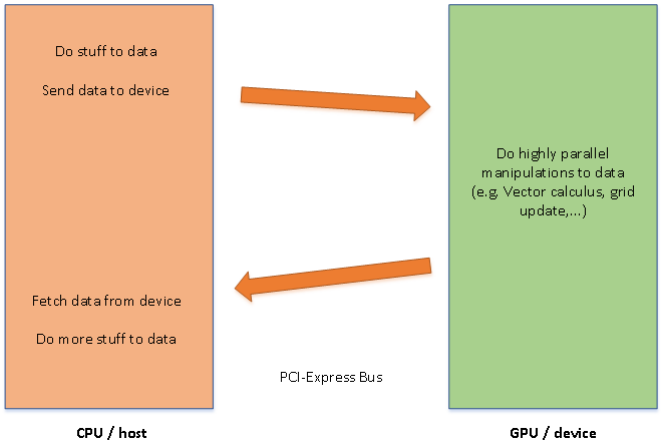

Figure 1. CUDA generic program flow

Figure 2. Kepler architecture (compute capability 3.x) Streaming Multiprocessor (from NVIDIA Kepler Whitepaper).

Architectural specifications (per SM)	**Compute capability**			
	1.0 G8x/G9x	**2.1** Fermi	**3.0** Kepler	**3.5** Kepler
Number of integer and single-precision floating-point cores	8	48	192	
Number of single-precision special function units (Sine, Cosine, etc.)	2	8	32	
Number of double-precision floating-point cores	0		64	
Number of warp schedulers	1	1	4	
Number of instructions issued at once by scheduler	1	2		
Technical specifications				
Maximum number of threads per block	512	1024		
Number of blocks per SM	8		16	
Number of warps per SM	24	48	64	
Number of threads per SM	768	1536	2048	
Number of 32-bit registers per SM	8K	32K	64K	
Maximum number of 32-bit registers per thread	128	63		255
Maximum amount of shared memory per SM	16KB	48KB		

Table 1. Major differences between select compute capabilities. The names under the compute capabilities give examples of chip codenames with that capability.

1.2 Compute capability

The first CUDA software development kit (SDK) was released in 2007. As you can imagine, the hardware has changed significantly since then. To accomodate such changes, NVIDIA has introduced the paradigm of compute capabilities, which classify graphics cards according to their hardware setup. The fact that each graphics card has one of just a few (mostly backwards compatible) compute capabilities makes it easy for the programmer to develop software for a wide range of devices. For most CUDA programmers, this means that only two main factors have to be considered when writing software: The compute capability of the device and the number of SMs on it. Wikipedia has a very good summary of what distinguishes the different compute capabilities (http://en.wikipedia.org/wiki/CUDA), the most important ones for us are repeated in table 1. You do not have to understand the ramifications of this table yet. Just return here (or better: to Wikipedia) when questions regarding the compute capability arise.

If not stated otherwise, all values given during these turorials refer to compute capability 3.5 as this is what the GTX Titan supports and what is relevant for developing our software.

2 When you should utilize the power of your graphics card... and when better not to

The worlds second fastest supercomputer (as of June 2013) at the Oak Ridge National Laboratoy named Titan utilizes a great number of NVIDIA Tesla cards, which are almost identical to the

consumer graphics card GTX Titan (hence the name) built in our CUDA computer Prometheus. It seems that graphics cards are a good choice for coping with *high performance computing* (HPC) tasks. As you will see, this statement is somewhat but not unconditionally true. Some examples where CUDA might do some good have been named in the introduction. Let us take a look at an example, where a CUDA implementation will be much slower than a standard CPU implementation: The numerical integration along a one-dimensional grid.

Let r_i denote the position of the i-th grid point starting at $i = 0$ (going up to, say, $i = 1,000$), x and y two arbitrary quantities to be evaluated at the grid points (where they will be referred to as x_i and y_i) and an analytical expression linking these two quantities:

$$y = \int x dr$$

With a boundary condition at $r = r_0$ and known values for x at all grid points, this equation can be solved numerically (introducing some error, of course) by moving along the grid from left to right and computing

$$y_{i+1} = y_i + x_i(r_{i+1} - r_i)$$

After $1,000$ of these steps, the values y_i are computed at every single grid point. However simple, this calculation can not be sped up with CUDA enabled graphics cards. The reason is very easy to understand: There is no way to compute y_{i+2} without computing y_{i+1} first. You might have thousands of compution cores at your disposal, this task can utilize only one core efficiently.

However, if you had many of these grids, say, $n = 10,000$, then you could work upon the grids in parallel, while each grid for itself would be processed sequentially. In this case, CUDA will speed up things significantly.

Fortunately, most of our computing tasks here at the institute feature independent calculation steps, hence CUDA might help you here and there. If you have read this far and still ask yourself if CUDA programming can help you speed up things, I am inclined to answer you with a definite "YES!".

3 Where to next?

The next few tutorials (with the number 01) will be concerned with organizational stuff like programming languages (Python, C/C++), operating systems (Windows, Linux), a GPU emulator called "Ocelot" and our CUDA computer Prometheus on which you can test and run your programs. None of these tutorials will go deeper into CUDA programming itself.

If you are interested in more about what can be done with CUDA without programming along, just skip the following sessions and keep on reading the tutorial with the number 02.

If you wish to lern CUDA programming, I recommend getting familiar with the 01-tutorials concerned with your developing enviroment. If you want to develop complex programs utilizing CUDA, I strongly recommend programming in C/C++ because of better performance of the resulting programs and because I have some experience I can share. On the other hand, fast prototyping is more easily done utilizing Python, so if you just want to speed up specific parts of your program (e.g. multiplication of large matrices), that can be implemented very fast using the Pyhton packages avialable for CUDA. See tutorial 01p for more on this topic.

Prometheus is set up with all necessary Python and C/C++ packages, so go ahead with whatever suits your needs best. The CUDA paradigm is independent of the specific programming languages.