

Bachelor project

GPU

Michiel de Reus

For my Bachelor project, I cooperated with VORtech BV. VORtech is founded in 1996 and situated in Delft. It is a combination of an engineering firm and a software house. One of the products they maintain is CONTACT, a software package used to calculate contact areas for three-dimensional frictional contact problems. The goal of my project was to investigate whether certain parts of the algorithms used could be parallelized and run on the Graphics Processing Unit. I used the CUDA architecture from NVIDIA to accomplish this. CUDA is an extension of the C programming language, and adds certain keywords and functions to it. This article will mainly focus on CUDA itself, and less on the mathematical background of the algorithms.

The last few years, Graphics Processing Units (GPU) are more and more used to perform general computations instead of just creating graphical scenes. This technic is called GPGPU, which is an acronym for 'General-Purpose computation on Graphics Processing Units'. There are different platforms to achieve this. The platform used in my project is CUDA, which is an acronym for 'Compute Unified Device Architecture'. This architecture can be used on all CUDA enabled devices from NVIDIA, which in practise are all recent devices. We will first look at the physical device layout. After this the programming model will be described. Finally we will consider a simple example of some matrix operation.

Device layout

Figure (1) shows the layout of a typical CUDA device. We see that a device consists of a number of (streaming) multiprocessors, and the global memory. Global memory is the bulk memory which stores all the big data structures. Its typical size is about 512 MiB to 1.5 GiB (this is actually the number you see when you buy a new graphics card). Since the global memory is not located on-chip, it has a relatively high latency.

The multiprocessors are independent processing units. They each consist of 8 (streaming) processors, 16 KiB of shared memory and 8192 or 16384 registers. Furthermore they have one Instruction Unit. One consequence of this is that each Streaming Processor on a particular Streaming Multiprocessor will execute the same instruction, but the data on which the instructions work can vary. This is the so called 'single thread multiple data' architecture. This is an important thing to realize when developing your parallel algorithm.

The shared memory is located on-chip and can be accessed much faster than the global memory, and so are the registers. The main difference is that registers are only accessible from one Streaming Processor, while the shared memory is accessible from all the processors of a multiprocessor.

The number of multiprocessors varies. For a simple graphics card in a notebook it can be 4, while high end devices can have up to 30 multiprocessors.

Programming model

When performing calculations on the GPU using CUDA, a so-called kernel is launched. This is a function which is executed on the GPU.

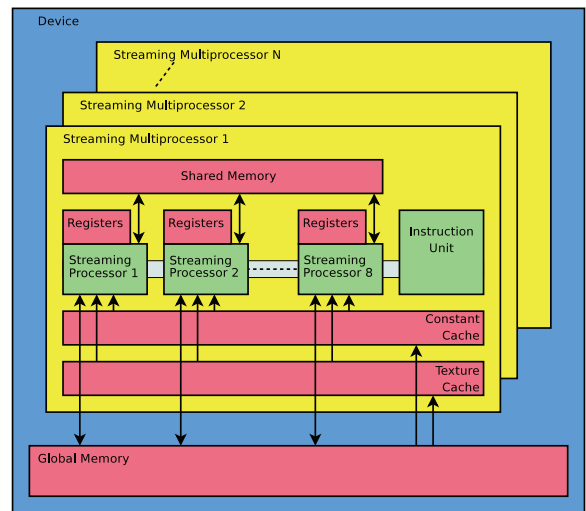


Figure 1: Device layout.

When programming in a parallel way, the total work has to be divided into chunks which can be processed independently from each other. In the CUDA programming model, the total work is divided into blocks. All blocks together constitute the grid. Different blocks are executed independently of each other, and it is not possible to communicate or synchronize between different blocks. Even the order in which different blocks are executed is not defined.

Each block is again divided into a number of threads. Synchronization between different threads of a block is possible, and usually necessary when shared memory is used. Figure (2) shows the division of the work load schematically. As mentioned before, the threads are executed using the 'single instruction, multiple thread' model, which is comparable to the 'single instruction, multiple data' model. Each thread executes the same instructions, using different operands.

Kernel Launch

As mentioned before, the kernel is the function that is executed on the GPU. On the kernel launch different parameters are set. In particular what the dimension of the grid are, and how much shared memory each block uses. The dimensions of the grid can be passed using special CUDA syntax:

```
Kernel<<<dimGrid, dimBlock>>>(...);
```

here `dimGrid` and `dimBlock` determine the structure of the grid and the blocks. It is possible to use a two or three dimensional structure. Apart from these special parameters, you can pass other parameters in the same way you are used to when invoking normal functions. For example a pointer to some allocated memory in the global memory, or some coefficient used in the calculations.

Inside the kernel it is important that each thread can identify itself, so it can determine which calculations it should perform. There are special, built-in variables available for this purpose. A thread can determine in which block it

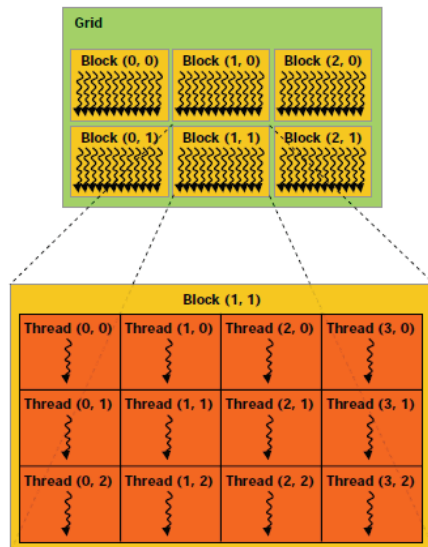


Figure 2: Grid layout (from CUDA Programming Guide).

resides, by using the `blockIdx` variable, which has up to 3 components. Inside the block the thread can determine its unique id by using the `threadIdx` variable, which has up to 2 components. How these variables are used can be seen in the example later on.

Memory management

When programming in C it is important to make sure that memory is allocated before it is used, and the same holds for the use of the global memory on the GPU. Special `malloc`- and `free`-like function are provided by CUDA. Other routines can be used to copy data from the regular internal memory to the global memory on the GPU and the other way around.

Because different types of memory are available, it is important to think about what kind of memory you want to use for what purpose. Data that is used more often should be kept near the processors, that is on-chip, while data that is only used once can reside in the global memory. Of course you should also keep an eye on synchronization issues. When reading from the shared memory, it is important to be sure that another thread is finished with the calculations on that particular piece of data.

Matrix example

As an example we will now consider a simple, component wise, matrix multiplication (Equivalent to the Matlab statement $C = A \cdot B$). Such a calculation is particularly suitable to parallelize, because a lot of the calculations can be performed independently from each other. Assume we have two square matrices A and B with dimensions 32×32 . A particular entry of C is given by

$$C_{ij} = A_{ij} \cdot B_{ij}, \text{ for } 1 \leq i, j \leq 32.$$

For simplicity we will only use four blocks, each consisting of 16 threads. We will also just use the global memory. Each block calculates a quarter of the entries of C . A simple kernel to achieve could look like this:

```
__global__ void Prod(float *dA, float *dB, float *dC) {
    int i_0 = blockIdx.y * 16;
    int j = blockIdx.x * 16 + threadIdx.x;
```

```
    for(int i = i_0; i < i_0 + 16; i++) {
        dC[i * 32 + j] = dA[i * 32 + j] * dB[i * 32 + j];
    }
}
```

Here we see that each thread executes 16 multiplications, in the for loop. This means that each blocks performs $16 \cdot 16 = 256$ multiplications, so the total number of multiplications will be $4 \cdot 256 = 1024$, exactly the number of components of C . Furthermore it is important to realize that the pointers point to the global memory on the GPU, not to the regular internal memory of the system.

This kernel can be invoked by:


```
Prod<<<(2,2), 16>>>(dA, dB, dC);
```

Here we see that the grid is divided in two by two blocks. Before invoking the kernel, the matrices A and B need to be allocated and transferred to the GPU. This can be done by using functions like:

```
cudaMalloc((void**)&dA, sizeof(float) * 32 * 32);
```

```
cudaMemcpy(dA, hA, sizeof(float) * 32 * 32, cudaMemcpyHostToDevice);
```

Getting started

As you can see using CUDA for parallel computing is not very hard, specially when you have a C background. The CUDA Programming Guide gives detailed information on all the routines available, and the most recent CUDA Toolkits come with a lot of easy-to-use tools to analyze your programs and try to find out how your code can be optimized. There is also a debugging tool `cuda-gdb` which is comparable to the standard `gdb` debugger, but of course, has special capabilities to analyze the behavior of different threads. There is also a big online community of developers, where a lot of information can be found. Of course here I just mentioned a few of the possibilities, and all I can say is, try it for yourself! 

References

- [1] NVIDIA, *CUDA Programming Guide*.
- [2] NVIDIA GPU Computing Developer Home Page, <http://developer.nvidia.com/object/gpucomputing.h>

