

Solving a wave equation with CUDA

LAURENT MICHEL

May 25, 2011

1 Introduction

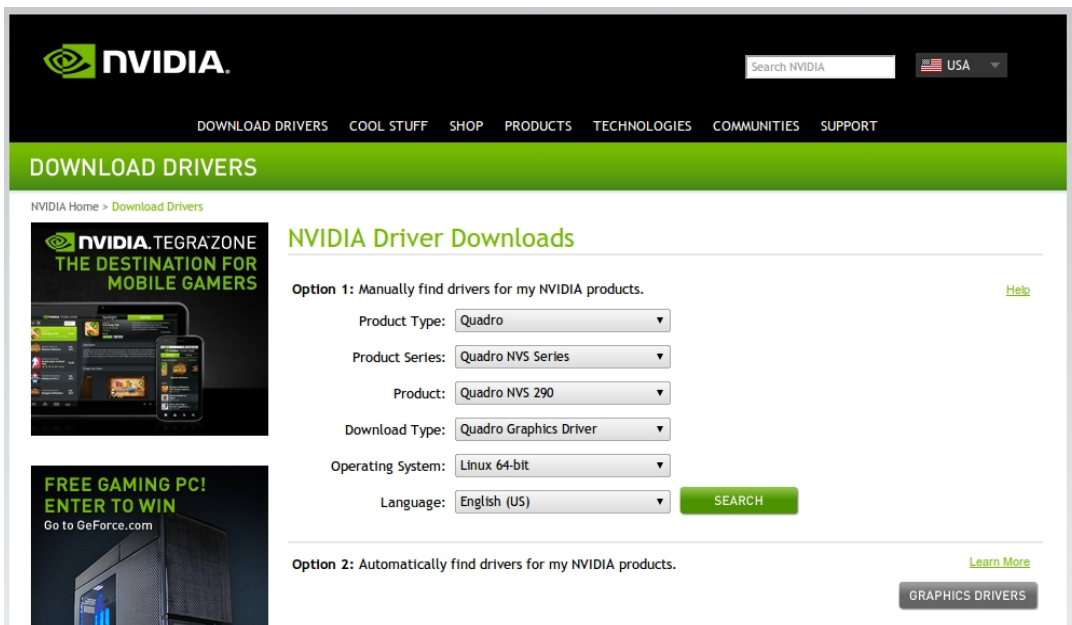
Parallel computing has been developed a lot during the last years. In particular, computer scientists have tried to first increase the CPU clock rate of our computers, and then to increase the amount of actual CPUs. With this latter increase, they are faced to the problem of making them smaller and smaller, until their size becomes such that quantum effects get involved in the process and make their production impossible. In the same time, people have begun to use GPUs for their calculations, which are known to contain thousands of processing units. For the scientific community, this way of proceeding was not optimal, because one had to translate the problems into graphical problems and then solve them by applying graphics functions (with OpenGL for example). As time went on, Nvidia has made available a new library allowing for an easier way to work with GPUs: CUDA. In this short report, I first explain how to get started with CUDA. Then, I explain very quickly what is different between CPUs and GPUs. Finally, I present an implementation of a numerical scheme to solve a 2D wave equation.

2 Configuration of CUDA

2.1 Installation of the nvidia drivers

Download the last version of the driver of your graphics card on

<http://www.nvidia.com/Download>



On Ubuntu 10.04, switch to runlevel 1

```
sudo init 1
```

to turn off the X server. Then, switch to runlevel 3

```
telinit 3
```

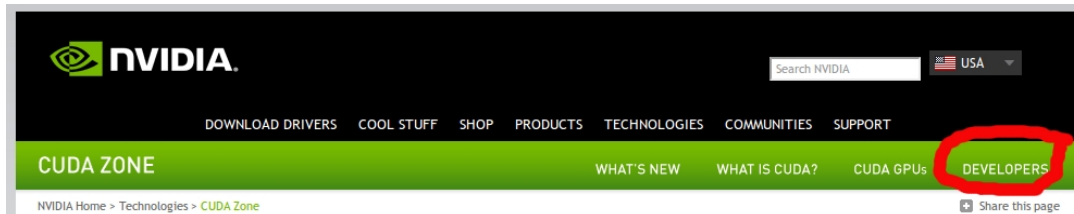
so that you can perform the installation via your personal account. Run the `nvidia` script

```
sudo ./NVIDIA-Linux-x86_64-270.41.06.run
```

and answer the questions you're asked.

2.2 Installation of the CUDA library

Go to the developers' page



Follow the link to the CUDA downloads and select the toolkit you need. Run the `nvidia` script

```
./cudatoolkit_3.2.16_linux_64_ubuntu10.04.run
```

and answer the questions you're asked. When this is done, add the following lines to your `$HOME/.bashrc`

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${CUDA}/lib:${CUDA}/lib64
export PATH=$PATH:${CUDA}/bin
```

2.3 Installation of the CUDA sample codes

On the same webpage you got the CUDA Toolkit, you'll find GPU Computing SDK code samples you may want to try out. The installation is a bit trickier: first launch the downloaded script

```
./gpucomputingsdk_3.2.16_linux.run
```

answer the questions you're asked, and install the following additional packages (on Ubuntu 10.04):

```
libgl1-mesa-dev
libglu1-mesa-dev
libxmu-dev
freeglut3-dev
libxi-dev
```

As soon as this is ready, go to the directory where you've freshly installed the code samples, and

```
cd C
make
```

3 Structure of a C++ project with CUDA source files

When we develop some code to solve PDEs, we may end up with lots of source files. When no CUDA code is involved, handling such a project is pretty easy, since we compile every source file with the same compiler. Here we need a different compiler for CUDA source files, hence we need to be a bit careful when linking binaries together. In this section, I propose a very simple CUDA project, with very few files, and show how to proceed so that everything works out. First, we write a main program that calls a function `doStuff()` whose purpose here is to make use of CUDA. Since this piece of code meets the C++ standards, we can compile it with `g++` as usual.

```
#include "extern.h"
```

```
int main () {  
    doStuff();  
    return 0;  
}
```

The external header `extern.h` contains the declaration of function `doStuff()`:

```
#include <iostream>  
#include "cuda.h"  
using namespace std;  
void doStuff();
```

and the actual actions of this function are described in the CUDA file `kernel.cu`:

```
#include "extern.h"  
  
__global__ void kernel(void) { }  
  
void doStuff() {  
    kernel<<<1,1>>>();  
    cout << "Hello World !" << endl;  
}
```

To link all of this code together, we have to proceed in the following way:

```
g++ -I${CUDA}/include -I. -c main.cpp -o main.o  
nvcc -I${CUDA}/include -I. -c kernel.cu -o kernel.o  
g++ -Wl,-rpath,${CUDA}/lib64 -L${CUDA}/lib64  
    -lcuda -lcudart  
    main.o kernel.o  
    -o main
```

For large projects, a makefile is preferable:

```
INCLUDE    = -I${CUDA}/include -I.  
LIBS       = -Wl,-rpath,${CUDA}/lib64  
           -L${CUDA}/lib64  
           -lcuda  
           -lcudart  
  
CXX        = g++  
LD          = g++  
CXXFLAGS    = -O2 -Wall $(INCLUDE)  
LDFLAGS     = -O2
```

```

NVCC          = nvcc
NVCCFLAGS    = $(INCLUDE)

OBJ = main.o kernel.o
EXE = main

all: $(EXE)

.SUFFIXES: .cu
.cu.o:
    $(NVCC) $(NVCCFLAGS) -c $< -o $@
.cpp.o:
    $(CXX) $(CXXFLAGS) -c $< -o $@

main: $(OBJ)
    $(LD) $(LDFLAGS) $(OBJ) $(LIBS) -o $@

```

4 GPU properties

Traditionally, a CPU contains a few cores (arithmetic logical units ALU) that share their memory in a common cache and a common virtual memory (DRAM). On one single computer, it is possible to use shared or distributed memory models (with OpenMP or MPI). In the former case, each core of a same computer can communicate without any problems, while a latency occurs in the latter. Usually, we have a few cores per computer and, for large parallel applications, we link together a certain amount of computers to build a cluster, so that work load and memory are distributed over different computers.

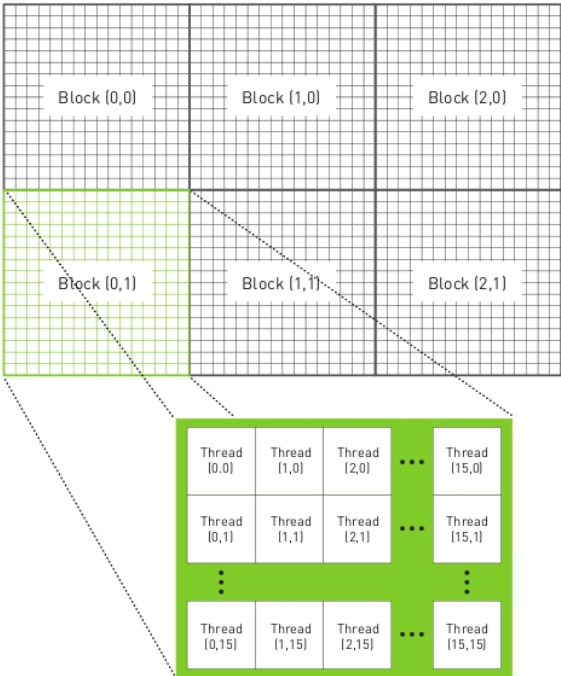
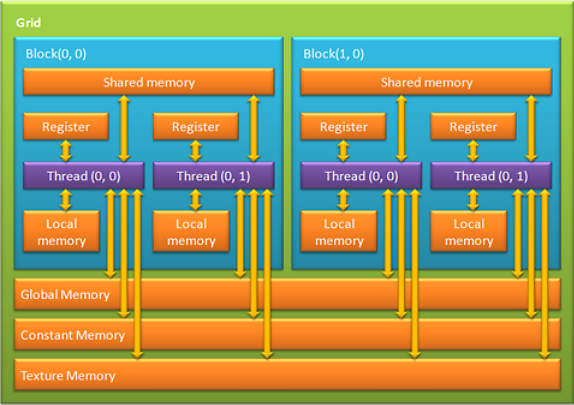
GPUs work a bit differently in the sense that a GPU consists of thousands of arithmetic logical units. These ALUs are grouped in blocks sharing the same memory. Memory cannot be accessed directly from one block to the other. Additionally, in the same way CPUs work with DRAM memory, GPUs work with a reasonably large amount of global memory. Depending on the user needs, the ALUs can be ordered 1, 2 or 3-dimensionally, as showed by the pictures below. Apart from global memory, a GPU has constant and texture memory that may affect performance if used appropriately.

One-dimensional block arrangement



Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Two-dimensional block arrangement



When coding on GPUs, we need to be careful with hardware limitations, which are basically given by the compute capability. The following figures give a brief summary of these limitations:

Feature Support (Unlisted features are supported for all compute capabilities)	Compute Capability				
	1.0	1.1	1.2	1.3	2.0
Integer atomic functions operating on 32-bit words in global memory (Section B.10)	No	yes			
Integer atomic functions operating on 64-bit words in global memory (Section B.10)	No	Yes			
Integer atomic functions operating on 32-bit words in shared memory (Section B.10)					
Warp vote functions (Section B.11)					
Double-precision floating-point numbers	No			Yes	
Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.10)	No	Yes			
__ballot__ (Section B.11)					
__threadfence_system__ (Section B.5)					
__syncthreads_count__(), __syncthreads_and__(), __syncthreads_or__ (Section B.6)					

Technical Specifications	Compute Capability				
	1.0	1.1	1.2	1.3	2.0
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24	32			48
Maximum number of resident threads per multiprocessor	768	1024			1536
Number of 32-bit registers per multiprocessor	8 K	16 K			32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2 ²⁷				
Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of instructions per kernel	2 million				

5 Computation of the solution of a wave equation

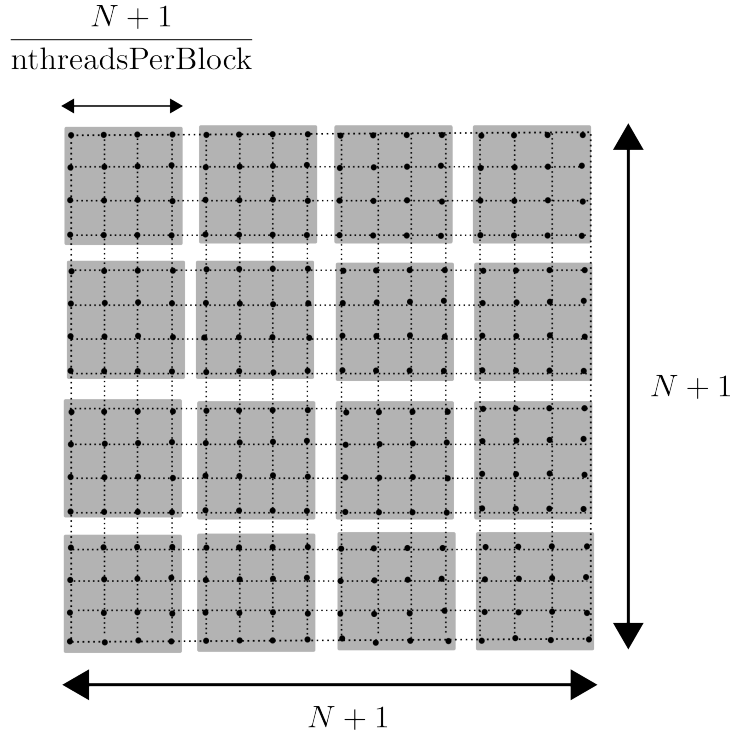
Aim is finally to solve the following problem: let $\Omega = [0, 1] \times [0, 1]$, $f, u_0 \in \mathcal{C}^0(\Omega)$; find a $\mathcal{C}^2(\Omega)$ -function $u(x, y, t)$ satisfying the wave equation

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2}(x, y, t) - \Delta u(x, y, t) &= f(x, y, t) && \text{in } \Omega, \text{ for } t > 0 \\ u &= 0 && \text{on } \partial\Omega, \text{ for } t > 0 \\ u(x, y, 0) &= u_0(x, y) && \text{in } \Omega \\ \frac{\partial u}{\partial t}(x, y, 0) &= 0 && \text{in } \Omega. \end{aligned}$$

To solve this problem, I am using a centered finite difference scheme for both the time and spatial derivatives. Let's split each direction of Ω into $N + 1$ points. This means that we have N intervals along each direction, and hence $h = \frac{1}{N}$ is the spatial step. Denote by τ the time step. Accordingly, let $x_i := ih$, $y_j := jh$, $0 \leq i, j \leq N$, and $t_n := n\tau$, $n \geq 0$. The numerical scheme reads therefore

$$\begin{aligned} \frac{u_{ij}^{n+1} - 2u_{ij}^n + u_{ij}^{n-1}}{\tau^2} - \frac{u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n - 4u_{ij}^n}{h^2} &= f(ih, jh, n\tau), \quad 1 \leq i, j \leq N-1 \\ u_{0,j}^n = u_{N,j}^n = u_{i,0}^n = u_{i,N}^n &= 0, \quad n \geq 0, 0 \leq i, j \leq N \\ u_{i,j}^0 &= u_0(ih, jh), \quad 1 \leq i, j \leq N-1 \\ u_{i,j}^1 &= u_{i,j}^0, \quad 1 \leq i, j \leq N-1. \end{aligned}$$

I arrange the computational domain in such a way that each thread is responsible for exactly one unknown (black dots on the following picture). For this to be possible, simulations need to be run with $\frac{N+1}{\text{nthreadsPerBlock}}$ blocks each consisting of nthreadsPerBlock threads. Due to hardware limitations, since I am using a two-dimensional grid of threads, I can't work with blocks of dimensions more than 22 threads by 22 threads (the total amount of threads per block must not be larger than 512).



The implementation of this scheme is detailed on the next pages.

```
#include <iostream>
#include "gpu_data.h"

using namespace std;

int main (int argc, char *argv[]) {
    if(argc != 4) {
        cerr << "Usage: " << argv[0] << " N tf nthreads" << endl;
        return 1;
    }

    int    N          = atoi(argv[1]);
    float  tf          = atof(argv[2]);
    int    nthreads    = atoi(argv[3]);

    float *u = new float [(N+1)*(N+1)];
    GPUData d(u, nthreads, N, tf);
    evolveState(d);
    delete [] u;

    return 0;
}
```

```

#include "gpu_data.h"
#include "math_functions.h"

__device__ float u0(float x, float y) { // initial state definition
    // This function is only called on the device, and never on the host, hence it has to carry the
    identifier __device__
    return x*(x-1)*y*(y-1);
}

__global__ void init(float *u, float *h, int *D) { // initial state filling
    int i = threadIdx.x + blockIdx.x*blockDim.x; // 2D-grid of pixels, each one being a problem unknown
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int idx = i + j*blockDim.x*gridDim.x;
    if(idx < (*D)*(*D))
        u[idx] = u0(i*(h), j*(h));
}

void GPUData::initState() {
    dim3 threads(_nthreads, _nthreads), blocks(_nblocks, _nblocks); // definition of grid of block and
    threads
    init<<<blocks, threads>>>(_uOld, _h, _D);
    init<<<blocks, threads>>>(_uCurr, _h, _D);
}

__device__ float f(float x, float y, float t) {
    return expf(-t)*(x*(x-1)*y*(y-1) - 2*(y*(y-1)+x*(x-1))); // definition of forcing term
}

__global__ void evolve(float *n, float *c, float *o, int *D, int *N, float *h, float *dt, float *t) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int idx = i + j*blockDim.x*gridDim.x;

    if(!i || i == *N || !j || j == *N) { // dealing with boundary conditions; this is not necessary, we
    could also
        n[idx] = u0(i*(h), j*(h)); // cut this part of the solution and add it when we want to
        display it
        return;
    }

    int left  = idx-1;
    int right = idx+1;
    int top   = idx + *D;
    int bottom = idx - *D;

    if(idx < (*D)*(*D)) // numerical scheme
        n[idx] = -o[idx] + 2*c[idx] + (*dt)*(*dt)*((*N)*(*N)*(c[left] + c[right] + c[bottom] + c[top]
        - 4 * c[idx]) + f(i*(h), j*(h), (*t)));
}

void evolveState(GPUData & d) {
    dim3 threads(d.nthreads(), d.nthreads()), blocks(d.nblocks(), d.nblocks()); // definition of grid
    of block and threads

    while(1) {
        evolve<<<blocks, threads>>>(d._uNext, d._uCurr, d._uOld, d._D, d._N, d._h, d._dt, d._t);
        if(d.incrementDT()) break;
        evolve<<<blocks, threads>>>(d._uOld, d._uNext, d._uCurr, d._D, d._N, d._h, d._dt, d._t);
        if(d.incrementDT()) break;
        evolve<<<blocks, threads>>>(d._uCurr, d._uOld, d._uNext, d._D, d._N, d._h, d._dt, d._t);
        if(d.incrementDT()) break;
    }
}

```

```

#ifndef _GPU_DATA_H
#define _GPU_DATA_H

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cassert>
#include <fstream>

#include "cuda.h"
#include "cuda_runtime_api.h"
#include "utils.h"

using namespace std;

// Wrappers for memory allocation and copy
template <class T> T *myCudaMalloc(int size) {
    T *tmp = NULL;
    cudaMalloc((void **)&tmp, size*sizeof(T));
    return tmp;
}

template <typename T> T *myCudaMallocMemcpy(T *p, int size) {
    T *tmp = myCudaMalloc<T>(size);
    cudaMemcpy(tmp, p, size*sizeof(T), cudaMemcpyHostToDevice);
    return tmp;
}

// Data structure
class GPUData {
public:
    GPUData(float *u, int nthr, int npoints, float endTime) : fileName("output-"),
        _nthreads(nthr) {
        int D = npoints+1;
        float t = 0.f, tf = endTime, h = 1./npoints, dt = h/2.;

        if((float) ((float) D/_nthreads - (int) D/_nthreads) != 0.f) {
            cerr << "Incompatible number of threads" << endl;
            exit(1);
        }
        _nblocks = D/_nthreads;
        cout << "Defining " << _nblocks << " blocks and " << _nthreads << " threads" << endl;

        size = D*D;
        cudaDeviceProp prop; cudaGetDeviceProperties(&prop, 0);
        if(2*size*sizeof(float) > prop.totalGlobalMem) {
            cerr << "Size of solution too big to be handled on your GPU: " <<
                2*size*sizeof(float)/1024/1024 << " MB." << endl;
            cerr << "Available memory: " << prop.totalGlobalMem/1024/1024 << " MB." << endl;
            exit(1);
        }
        if(_nthreads*_nthreads > prop.maxThreadsPerBlock) {
            cerr << "Too many threads per block: " << _nthreads * _nthreads << endl;
            cerr << "Maximal allowed number of threads per block is " << prop.maxThreadsPerBlock
                << endl;
            exit(1);
        }

        _N = myCudaMallocMemcpy<int>(&npoints, 1);
        _h = myCudaMallocMemcpy<float>(&h, 1);
        _D = myCudaMallocMemcpy<int>(&D, 1);
        _size = myCudaMallocMemcpy<int>(&size, 1);
        _dt = myCudaMallocMemcpy<float>(&dt, 1);
    }
};

```

```

    _t = myCudaMallocMemcpy<float>(&t, 1);
    _tf = myCudaMallocMemcpy<float>(&tf, 1);

    _uOld = myCudaMallocMemcpy<float>(u, size);
    _uCurr = myCudaMallocMemcpy<float>(u, size);
    _uNext = myCudaMallocMemcpy<float>(u, size);

    initState();
}

~GPUData() {
    cudaFree(_N); cudaFree(_h); cudaFree(_D); cudaFree(_dt);
    cudaFree(_t); cudaFree(_tf); cudaFree(_uNext); cudaFree(_uCurr);
    cudaFree(_uOld);
}

void initState();
bool incrementDT() {
    float tmp_t = t()+dt();
    if(tmp_t < tf()) {
        cudaFree(_t); _t = myCudaMallocMemcpy<float>(&tmp_t, 1);
        return false;
    }
    return true;
}

float dt() { float tmp; cudaMemcpy(&tmp, _dt, sizeof(float), cudaMemcpyDeviceToHost); return
tmp; };
float t() { float tmp; cudaMemcpy(&tmp, _t, sizeof(float), cudaMemcpyDeviceToHost); return
tmp; };
float tf() { float tmp; cudaMemcpy(&tmp, _tf, sizeof(float), cudaMemcpyDeviceToHost); return
tmp; };
int nthreads() { return _nthreads; }
int nblocks() {return _nblocks; }

int    size;
int    *_size;
int    *_N;
float  *_h;
int    *_D;
float  *_dt;
float  *_t;
float  *_tf;
float  *_uOld;
float  *_uCurr;
float  *_uNext;
private:
    string fileName;
    int    _nthreads;
    int    _nblocks;
};

void evolveState(GPUData &);

#endif

```

6 Conclusion

It's been an interesting experience to try to code with GPUs. I have seen that there still are a few issues with CUDA. Apparently, the library is not really C++ friendly. Memory allocation is very "C-style", with `malloc` functions. Moreover, arguments to device functions must be pointers, what may make the code a bit harder to read. Finally, debugging is kind of tricky. Code can compile and run, while doing absolutely not what you want. This can be due to bad memory allocation or hardware limitations. For example, if you happen to use more than 512 threads per block on a graphics card with compute capability less than 2.0, then everything will be fine until you have a look at the actual output of the program. There are however ways to deal with this. There seems to be a few (graphical or console) tools developed by nvidia to debug CUDA code.