# Real-Time GPU Fluid Dynamics

Jesús Martín Berlanga

Scientific Computing
Computer Science Technical College (ETSIINF)
Technical University of Madrid (UPM)

January 19, 2017

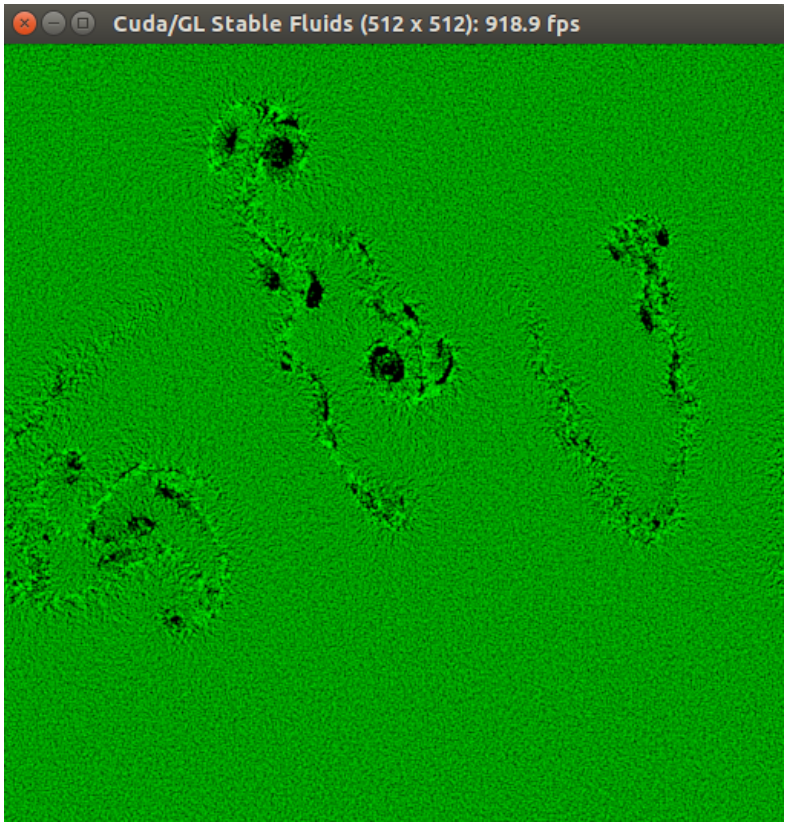


Figure 1: Interactive simulation visualization

# 1 Preface

Fluids are found in a multitude of different places: In the water of a river, in the cigar's smoke, in the water vapor that creates clouds... There is an increasing need for realistically representing these phenomena in all kinds of graphics applications (e.g. in a video game).

In the *Real-Time CPU Fluid Dynamics*[1] prequel, the reader was introduced to basic fluid concepts and formulae in order to code a *CPU* fluid implementation. This paper is a follow-up base for where it is further discussed how to parallelize the (already previously explained) fluid algorithm for a high performance. It is recommended to read *Real-Time CPU Fluid Dynamics*[1] before trying to understand this paper, as a common ground of concepts, is established in the CPU implementation document.

This CUDA GPU implementation is almost the same as the *CUDA/OpenGL Fluid Simulation* demo. Some changes have been made, for example: in order to measure the fps performance of the GPU implementation against the CPU implementation, an automatic test where the fps average is calculated has been added.

This document will prove particularly useful to whoever wants to extend his learning from the NVidia documentation on *GPU* fluids since, to date and as far as the author is concerned, *CUDA/OpenGL Fluid Simulation*[3] is not very specific explaining how the code works; and *GPU Gems: Chapter 39. Fast Fluid Dynamics Simulation on the GPU*[2] is outdated, not taking the advantages of *CUDA* over *Cg* (*C for Graphics* - shading language) for general purpose computing.

The source code along other resources can be found at `rtfluids.bdevel.org`.

# Contents

# 2 Introduction

Here is explained how a CPU fluid simulation can have its performance increased by using CUDA, a GPU solution. Before presenting any implementation details we will review some of the CUDA key-points. After that, the author explains how to deploy a thread layout for fluid parallel computing. Furthermore, the reader can find GPU optimized alternatives to CPU libraries. Finally, we will compare the CUDA implementation performance to the CPU implementation.

# 3 CUDA Basics

CUDA is a general purpose computing toolkit. Using the collection of functions at the CUDA API, a set of specifiers, and the CUDA compiler, the programmer can parallelize a C function (kernel) for N threads. Since the fluid simulation is based on a grid of vector velocities, we have to pay special attention to the optimization of two-dimensional arrays memory accesses.

## 3.1 Memory and Thread Hierarchy

Threads are grouped into blocks that share the same streaming multiprocessor (SM). Thus, they share the same shared memory and the same L1 cache (introduced in Fermi architectures) for that SM. For this reason, it's of extreme importance that different blocks are as independent from one another as possible, this reduces the penalty associated with synchronization to access the content stored in external shared memory as well as the penalty associated with accessing a higher level of cache.

For convenience, each kernel spawns N threads that can be indexed in one dimension (each thread is identified by an unique index, linear layout), two dimensions (each thread is identified by two indexes, grid layout), or even three dimensions. This is really valuable for the programmer.

Both memory and thread hierarchy explanations are well depicted in fig. 2 where the reader can see, for two kernels, their respective two grid memory layouts (grid 0 and grid 1). In addition, the grid-0 thread layout and indexing are also illustrated at the left.

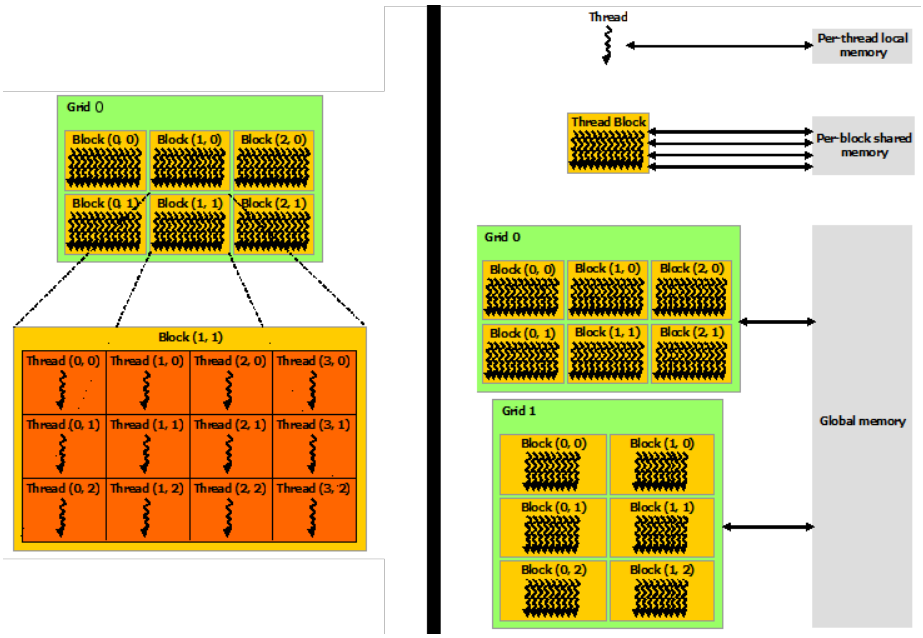


Figure 2: Grid of thread blocks at left, memory hierarchy at right (from the *CUDA Toolkit Documentation*)

When the blocks are totally independent from each other the block abstraction benefits are more clear: blocks can be distributed across several GPUs, at different machines.

One final note, because a block is restrained to a SM, the maximum number of threads per block is limited to the maximum number of threads the SM supports (Up to 1024 threads in modern GPUs).

The numbers of cores per SM is called the warp size, which is 32 in all (See Table 13. Technical Specifications per Compute Capability at CUDA Toolkit Documentation) NVidia multiprocessors.

### 3.1.1 Two-Dimensional Arrays Memory Access

We commonly find memory access patterns were we access a grid with two indexes, $x$ and $y$.

```
// host-equivalent to: memPtr = malloc(sizeof(cData)*width*height)
cudaMalloc(&memPtr, sizeof(cData)*width*height);
...
array_index = width*y + x
```

Because of inherent design characteristics of memory and cache lines, the width of the thread block, as well as the 2D array width, must be a multiple

of the warp size in order to ensure a maximum performance, as is described in the CUDA guide of best practices. This, for example, will ensure that all warps (SM) can access 128 bytes chunks of data in only one L1 cache request and do not need more requests as happens in fig. 3.

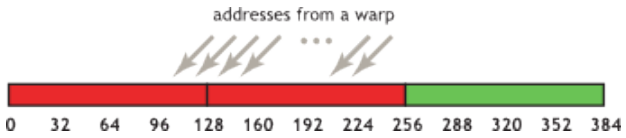


Figure 3: Unaligned sequential addresses that fit into two 128-byte L1-cache lines (from the *CUDA Toolkit Documentation*)

From the programmer point of view, all he has to do is to call *cudaMallocPitch()* when he needs to allocate a 2D array of memory: this function will make sure the memory is properly aligned for best performance.

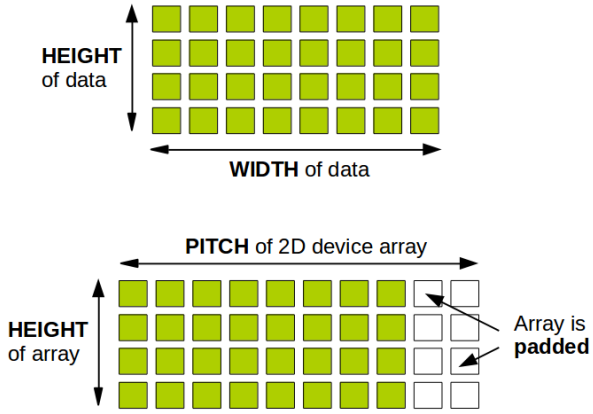


Figure 4: Padding is added, facilitating thread-blocks' memory accesses to be aligned to L1 cache when the block size is multiple of the warp size

Because of this memory alignment, a padding is introduced, and we no longer index memory positions with the data width: we have to use the pitch value (provided by *cudaMallocPitch*) to access memory. This is shown in fig. 4 (As the reader can notice, what we are calling pitch is what we were calling *pdx* when we had to iterate though an in-place fast Fourier transform array. Please, refer to the *Real-Time CPU Fluid Dynamics*[1] paper). The code we explained at the beginning of this section transforms to:

```
cudaMallocPitch(&memPtr, &pitch, sizeof(cData)*width, height);
...
array_index = pitch*y + x
```

Anyway, the programmer still have to make sure the block size is multiple of 32.

### 3.2 Kernels

A kernel is defined adding the *__global__* specifier before a function definition. When launching the kernel, the programmer can specify the number of blocks (NB) and threads per block (TPB) with $<<NB, TPB>>$ in the function call after the function name and before the parenthesis for arguments.

In order to use a thread (and block) grid indexing layout, *dim2* syntax (see code sample below) need to be used.

The function is executed $NB \cdot TPB$ times in parallel. *blockIdx.x*, *blockIdx.y*, *threadIdx.x*, *threadIdx.y* are the variables which individually identify each thread in the grid of blocks and threads. These identifiers will help us to access the matching array index position for each thread (in case there is 1 thread for each array position).

```
__global__ void MatAdd(float A[NN], float B[NN], float C[NN]) {
    // If there is only one block: blockIdx.x and blockIdx.y are always 1
    int array_index = N*threadIdx.y + threadIdx.x
    C[array_index] = A[array_index] + B[array_index];
}
int main() {
    ...
    dim2 numBlocks(1, 1); // in this case only 1 block: (1,1)
    dim2 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```

## 4 CUDA Fluids Implementation

The main changes needed to port a CPU implementation to a CUDA implementation are described in this section. First, we present alternatives to CPU interpolation and the CPU *fftw* library. Second, we explain how to access OpenGL vertex buffer from a kernel. After that, we need to allocate memory in the GPU device that the kernels will use for the simulation step computation. Finally, we have to decide what thread layout we will use.

### 4.1 Libraries Alternatives

#### 4.1.1 cuFFT

*cuFFT* is a CUDA library that provides the same functionality the *fftw* provides. The *cuFFT* was designed with the purpose of replicating the original *fftw* design. Even the memory layout can be set in a *FFTW-compatible* mode as the code used below shows.

```
// TODO: update kernels to use the new unpadded memory layout for perf
// rather than the old FFTW-compatible layout
cufftSetCompatibilityMode(planr2c, CUFFT_COMPATIBILITY_FFTW_PADDING);
cufftSetCompatibilityMode(planc2r, CUFFT_COMPATIBILITY_FFTW_PADDING);
```

This has one drawback, the layout will not follow the high-performance design we have previously thought it was possible to achieve with *cudaMallocPitch*.

We can easily transform between real and complex numbers with cuFFT:

```
// real to complex (fordward FFT)
cufftExecR2C(planr2c, (cufftReal *)vx, (cufftComplex *)vx);
cufftExecR2C(planr2c, (cufftReal *)vy, (cufftComplex *)vy);
// complex to real (inverse FFT)
cufftExecC2R(planc2r, (cufftComplex *)vx, (cufftReal *)vx));
cufftExecC2R(planc2r, (cufftComplex *)vy, (cufftReal *)vy));
```

### 4.1.2 Textures

When accessing data stored in a texture though floating-point indexes, the returned values are implicitly interpolated when using a linear filter. For this reason, we will bind a texture (with a linear filter) to *dvfield* and access *dvfield* values through a texture instead of manually interpolating them.

It is important to indicate that because how textures are designed to work with pixels (where each sample is positioned in the exact center of its corresponding pixel), when addressing for example through the index *(0.5, 0.5)* we are not interpolating the results from the vectors positioned at *(0,0)*, *(0,1)*, *(1,0)* and *(1,1)*, we are only accessing at the velocity stored at the sample positioned at *(0,0)*. In other words, we need to sum *.5* to both indexes to get the interpolated value we were looking for.

The original CPU manual interpolation, originally performed at the advection step, can be changed to:

```
// first of all, update texture with dvfield velocity values
updateTexture(...);
// note how 0.5 is added to each calculated array index (gtidx and fi)
ploc.x = (gtidx + 0.5f) - (dt * vterm.x * dx);
ploc.y = (fi + 0.5f) - (dt * vterm.y * dy);
// vterm will hold the interpolated result
vterm = tex2D(texref, ploc.x, ploc.y);
```

## 4.2 Memory Allocation

Kernels need access to the velocity field memory. Kernels executed in a GPU device only have access to the device memory. In the CPU version, we allocated the velocity field in the *hvfield* with *malloc*. In the GPU version, we

need to allocate the velocity field (*dvfield*, device velocity field) in the GPU device, with a CUDA function. For best performance, we allocate *dvfield* with *cudaMallocPitch*.

```
// Allocate and initialize device data
cudaMallocPitch((void **)&dvfield, &tPitch, sizeof(cData)*DIM, DIM);
```

The FFTW-compatible memory layout requires to ordinarily allocate memory with *cudaMalloc*. We will index this array the same way we did in the CPU version, using the *pdx* variable for the calculated padding.

```
// Temporary complex velocity field data
cudaMalloc((void **)&vxfield, sizeof(cData) * PDS);
cudaMalloc((void **)&vyfield, sizeof(cData) * PDS);
```

## 4.3 Vertex Buffer Mapping

CUDA provides handy functions to access the vertex buffer object (vbo), making possible to map it to a (device) pointer as we did in the CPU version.

```
struct cudaGraphicsResource *cuda_vbo_resource; // handler
cData *p; // mapped vbo pointer

cudaGraphicsMapResources(1, &cuda_vbo_resource, 0);
cudaGraphicsResourceGetMappedPointer((void **)&p,
    &num_bytes,cuda_vbo_resource);
// advect particles using the already mapped pointer
advectParticles_k<<<grid, tids>>>(p,...);
// dont forget to unmap!
cudaGraphicsUnmapResources(1, &cuda_vbo_resource, 0);
```

## 4.4 Thread Configuration

First, we choose a warp-size multiple for the block horizontal size: $32 \cdot 2 = 64$. Then we have to choose a block vertical size. Because the typical maximum threads per block is 1024, we cannot choose 64 for the vertical dimensions, as we would have $64 \cdot 64 = 4096$ threads! For this reason we choose a 64x4 block size (256 threads per block).

We divide the domain size in 64x64 square tiles and assign each tile to a block. This means the four vertical threads will have to process 16 grid vectors each one in order to process the entire square: $64 \cdot 4 \cdot 16 = 64 \cdot 64$. This domain division is shown in fig 5.

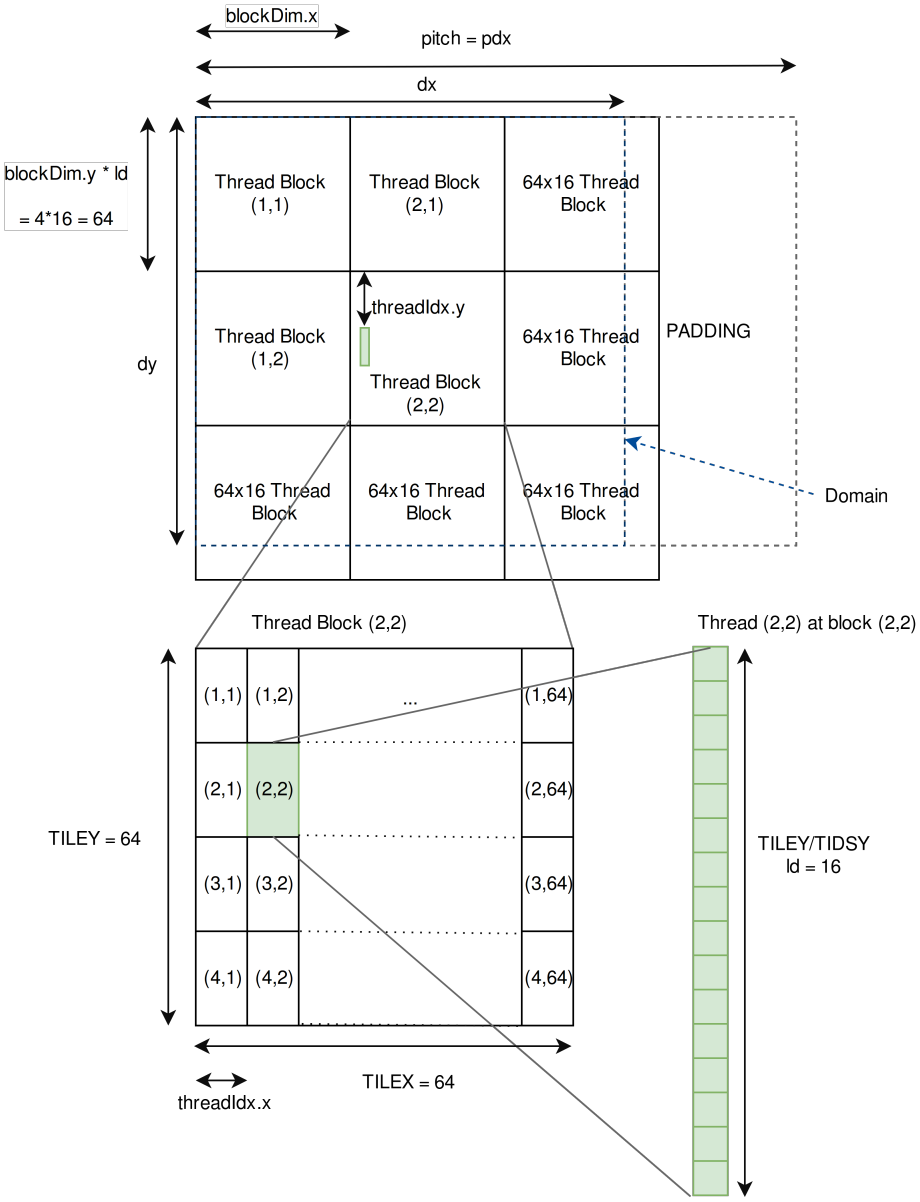


Figure 5: Domain size vs thread hierarchy

Once we know how the thread configuration works, we can create a template which will work for launching all simulation-step kernels. Pay close attention to the template commentary as it adds further explanations.

```
void simulationStepKernel_launcher() {
    // If the domain does not fit perfectly with the 64x64 tiles
    //  we still add the tiles where all the threads will not
    //  be taken advantage of.
    // We will have to be careful to not access an non-existent
    //  memory position for an non-existent domain coordinate.
    dim2 grid((dx/64)+(!(dx%64)?0:1), (dy/64)+(!(dy%64)?0:1));
    dim2 tids(64, 4);
    // We have to pass to the kernel the pitch value
    //    we obtained after using cudaMallocPitch
    simulationStepKernel_k<<<grid,
        tids>>>simulationStepKernel_k(tPitch,...);
}
__global__ void simulationStepKernel_k(size_t pitch, ...) {
    // We calculate the first grid coordinate position (gtidx,gtidy)
    //   were this thread need to operate. This is based on
    //   fig. 5.
    int gtidx = blockIdx.x * blockDim.x + threadIdx.x;
    int gtidy = blockIdx.y * (16 * blockDim.y) + threadIdx.y * 16;

      if (gtidx < dx) { // This check makes sure we don't try
      // to acess an non-existent memory position for an
      // non-existent domain coordinate.

        // We access the 16 vectors each thread need to operate
        //  with. This coordinates are found moving in the +y
        //  axis direction from the first coordinate.
        for (int p = 0; p < 16; p++) {
            int gtdiyy = gtidy + p;

            // Again, we make sure we dont access a non-existent
            // coordinate
            if (gtdiyy < dy) {
                // Index taking the pitch into account
                // Since the pitch value is in bytes, we
                //  have to do the calculations in bytes.
                cData* f = (cData *)((char *)v + gtdiyy * pitch) + gtidx;
                // Directly use the f pointer to
                //  write-to or read-from

                // Finally, we calculate the memory index
                //  for vxfield and vyfield
                // This index works for both vxfield and vyfield
                int fj = gtdiyy*pitch + gtidx;
                // vxfield[fj]; vyfield[fj];
```

```
                // INSERT simulation-step specific code here
            }
        }
}
```

With the template already made, we only need to insert each simulation-step specific code after the $f$ index is calculated. This is straight-forward knowing the CPU implementation code, few minor changes are required.

### 4.4.1 External Forces

For the *addForces* method, as we only have to compute a small tile of 9x9 (when radius is 4), we only need to launch the kernel with a single 9x9 block thread.

```
#define FR    4        // Force update radius
dim3 tids(2*FR+1, 2*FR+1);
addForces_k<<<1, tids>>>(...);
```

# 5 Performance

After compiling the GPU version, a benchmark test is executed to obtain the fps performance results:

```
./fluidsGL stress_test > fps_results.txt
```

The CUDA version is able to maintain an average of 1449.922729 frames per second. An impressive 43.63 speed-up (1449.922729/33.206783) with respect to the best of the CPU builds.

This results shows how important GPUs have become in the field of general purpose computation. The GPU can be used to exploit highly parallel algorithms for the best possible throughput or speed performance.

The best visual way to compare the CPU version against the GPU version is to run both benchmarks at the same time and perceive how much faster the GPU version outruns the CPU version. This is what is done at the *Real-Time Fluid Dynamics: CPU vs GPU* demonstrative video, which can be watched at `rtfluids.bdevel.org`. Take into consideration that, due to the screen recorder impact on performance, the shown fps are slightly decreased.

The machine specifications where the builds have been tested are in the appendix A.

# References


[1] Jesús Martín Berlanga, *Real-Time CPU Fluid Dynamics*, Technical University of Madrid. Available online: `http:rtfluids.bdevel.org`.

[2] Mark J. Harris, *GPU Gems: Chapter 39. Fast Fluid Dynamics Simulation on the GPU*, NVIDIA, University of North Carolina at Chapel Hill. Available online: `http://http.developer.nvidia.com/GPUGems/gpugems_ch38.html`.

[3] Nolan Goodnight, *CUDA/OpenGL Fluid Simulation*, NVIDIA Corporation.

[4] Jos Stam, *Stable Fluids*, In Proceedings of SIGGRAPH 1999. Available online: `http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf`.

[5] *CUDA C Programming Guide*, NVIDIA. Available online: `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[6] *CUDA C Best Practices Guide*, NVIDIA. Available online: `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide`.

[7] *cuFFT Documentation*, NVIDIA. Available online: `http://docs.nvidia.com/cuda/cufft`.

# A Target machine specifications

1. OS: Ubuntu 14.04.4 LTS 64 bits
   kernel: Linux 3.13.0-57-generic (x86_64)
2. CPU: 8x Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz
3. RAM: 18488 MB
4. GPU: GeForce GTX 780
   CUDA Capability 3.5
   NVIDIA Driver Version: 361.93.02
5. Compilers:
   g++ version 4.8.4 (Ubuntu 4.8.4-2ubuntu1 14.04.3)
   icpc 17.0.1