CUDA – A student's experience

Compute Unified Device Architecture

Stephen Neithardt

EPFL, June 24, 2010

After many months of work, I've come to the point where this CUDA project is stable, and has results close to the ones Bastien got last year. Performance isn't yet optimal, but already improvements can be seen. I hope to cover what I've come through.

1 Progress

December

This month was dedicated to learning CUDA. A small example program from the documentation was re-written, and many issues related to this kind of programming were discovered. Synchronization, memory read/write patterns, and control flow among others.

January

After a break for Christmas holidays, I took on the task of understanding the original code, and how things were implemented. Thinking started for the CUDA implementation, and rough drafts of pseudo-programing got issued.

February

Conversion of the original code, to change the way memory is stored. Now all data is stored in linear memory, enabling direct copies to the graphics card.

The first versions rolls, but nothing is stable. Performance is very fast, yet nothing has any sense. Much looking for errors must be done. The whole CUDA part is re-written once, to no avail.

March

Second re-write, and a little less problems arise. But still nothing stable. Third re-write. Tested a conversion from double to float types, to see the difference. Performance got faster (again), but no other noticable improvements. Still not stable. The problem lies elsewhere.

April

After some time messing around, Bastien points out something that looks fishy. I look into it, and the main error is erradicated. The code becomes better. A fourth re-write takes down most of the problems. Two little errors in other files are corrected during the last parts of the month. The last week is dedicated to tests, and comparisons with old data. Things are looking good. Time to proceed.

May

Some things needed to be tackled, and corrected, in the last parts of the program. Now, the simulation is correct, yet quite less speedy. The time is greater for small lattices. But for bigger ones, the time taken is almost the same, getting a huge improvement over the old code. Benchmarking is started by the end of the month.

June

Speed tests are finished, most test simulations are done. The writing of this document is started.

Future

What can be done to improve the actual code :

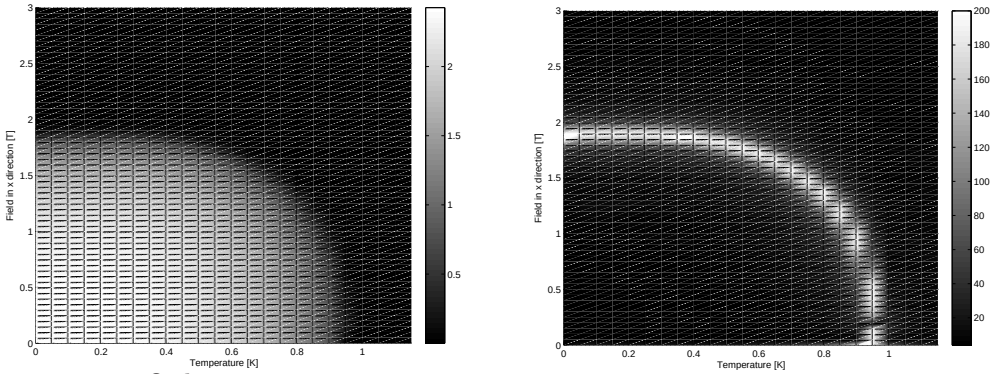
- Profiling of actual code, to see where improvements are necessary.
- Implement a different convergence check and twink the damping during execution.
- Tackle the memory issues, as in where the data resides, and how it is accessed. Optimize bandwidth use.
- Take the whole program to float precision. This could yield a 8 times speedup on the device part, as more units are available for calculating floats than for doubles.[1]

[1] See the SIMT architecture section 5.1

2 Results

This section covers some simulations results obtained during this CUDA project.

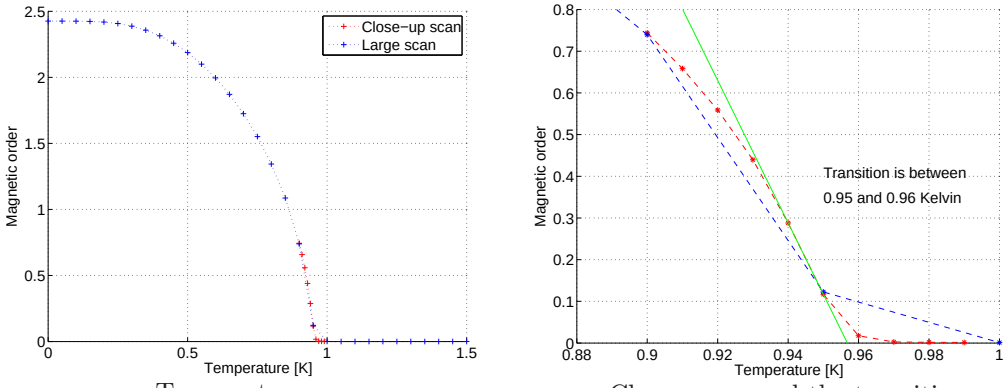
A 10x10x10 cells simulation of $Ho_{.44}Y_{.56}$ dopings took about 37 hours to complete, for a total of 1464 calculated points. That's an average of 91 seconds per point calculated. It's noteworthy that points far from transitions take far less time to accomplish than close to it. The total number of steps was 35416, making it 3.77 seconds of calculation per step. The phase diagram is shown in Fig. 1, where the magnetisation in the z axis is chosen as order parameter for the lattice, and is plotted on color scale. The result is consistent with previous calculations.



Order parameter Number of iterations

Figure 1: $Ho_{.44}Y_{.56}$ phase diagram, with a 10^3 lattice size.

To have a finer grained simulation, a single scan from $0K$ to $1.5K$ and no transverse field, at the same doping as before was performed. This gives a good curve to estimate the transition temperature at zero field. A second simulation with a smaller step was performed close to the transition, giving a better approximation of the temperature. The graph is shown in figure Fig. 2.



Temperature scan Close-up around the transition

Figure 2: Temperature scan for $Ho_{.44}Y_{.56}$, transition is 0.955 ± 0.005 K

The transition temperature calculated this way is 0.955 ± 0.005 K.

3 Benchmarking

To ascertain how well the CUDA code does, some benchmarking tests were done. First of all, a temperature scan, from $0K$ to $1.5K$ in steps of $0.1K$, with a transverse field of $0.5T$. The doping was 60% Ho, and the lattice size 10^3 cells. The old code does this scan in about 4 hours and 40 minutes. The CUDA code performs the same calculation in 56 minutes. A near five-fold improvement in time.

The second test involves a field scan, where the transverse field varies between $0T$ and $1T$ in steps of $0.1T$, at a temperature of $0.5K$. Again, the doping was 60% Ho, and the lattice 10^3 cells big. This time, the old code ran for a little less than 44 minutes. The CUDA version took just under 10 minutes to get the results. A bit less than the five-fold improvement, but improvement none the less.

A third test was performed with a very big lattice (50^3 cells) at a temperature of $0.4K$ and no transverse field. For this single point, with a doping of 30% Ho, it took over 15 hours[2] to complete for the old version. The new one ended after 1 hour and 42 minutes. This is nearly a nine-times faster simulation.

Finally, to see how things are dependent on lattice size, the same point was calculated repeatedly for different lattice sizes, with both codes. I chose $1K$, $0.5T$, and 100% Ho as fixed parameters. The lattice size varies from 1 to 10, then there is 15, 20 and 25. The number of cells is a cube function of lattice size, and thus the number of calculations is also in cube of the lattice size.

For very small to medium size lattices, the CUDA code is much less efficient. This comes from the much larger overhead for the CUDA code. Many things must be done before actual calculations can start. This includes memory transfers to and from the graphics card, synchronisation issues, and mapping the lattice on the grid. But then the parallelism of the algorithm kicks in, and the calculations get faster than the serial version as the size of the lattice grows. This is shown in the following table. Different lattice sizes were used to calculate the same point, at $1K$, no field, and no Yttrium doping.

N = ...	1	2	3	4	5	6	7	8	9	10	15	20	25
imf [s]	0.67	5.34	22.6	58.6	115	225	379	567	696	986	3900	8940	16850
cuda [s]	56.18	56.27	56.78	57.8	59.7	62.9	68.1	76.0	86.7	103	265	752	1243
ratio	0.021	0.095	0.40	1.01	1.93	3.6	5.6	7.5	8.0	9.6	14.7	11.9	13.6

This shows how the cuda code gets really better than the original one for big lattices. It must be noted that for high dopings, the original code gets better, whereas the CUDA code roughly keeps the same performance. Doping has nearly no effect on CUDA simulation time because non-interacting sites have to wait for the interacting ones to finish calculating before proceeding to diagonalisation.

4 Improving the code

One of the main places where improvements can be done, is in how the threads are mapped in the block, and how said blocks are mapped to the lattice. At the moment, the threads are arranged linearily in one direction of the lattice, and the blocks are arranged to cover the two other dimensions in a grid-like fashion. The gives odd numbered block sizes, which

[2] 15 hours, 17 minutes and 30 seconds

aren't optimal. Idealy, there should be 32 or 64[3] threads per block, to ensure occupancy, and take the highest advantage of the architecture of the graphics card.

A re-arranging of the blocks would then be mandatory. This should improve the speed of big to very big lattices, and ensure scalability of the algorithm.

Another aspect of optimisation that was totally overlooked in the actual working version, is memory access. Memory bandwidth is limited, and transfers from global to chip memory take time. This can be minimised by ensuring that most memory accesses are coalesced : the memory is read in a contiguous fashion, making it possible for the driver to take a whole chunk of linear memory in one go, instead of having to issue many different calls to different memory blocks. This requires planning ahead while allocating said memory, and how/when they are accessed from the threads during execution. There is no perfect cocktail for making the memory accesses optimal.

An efficient way to minimise global memory access is to copy heavily used parts to chip memory, on a block by block basis. Each block loads the data it will need in further calculations, and does so only once in a coalesced manner, making the whole much faster, at the cost of many more lines of code.

Something that must be done in the close future, is put the intensively used memory bits, like the interaction matrix and other fixed parameters, in constant memory space. This special memory is cached on the chip. Therefore, only the first thread of the first block executing on the processor needs to make an access to global memory. All subsequent accesses to the same location will be done on the cached part of the memory, totally avoiding memory latency.

One could think that putting all data in these more efficient memories would be the solution. Sadly, these memory spaces are quite small. On the Tesla card, only 16kB of data can be stored on the chip of each processor. When different blocks are executing on the same processor[4] this memory must be shared between them, and is thus quite smaller. Because of this, it is necessary to make correct memory instructions, and beware of how the data is stored.

Here is a list summarizing what can be done to further improve the CUDA code.

- Re-arrange threads inside the blocks, and how the blocks map the lattice.
- Tackle memory issues, be it global, constant, or shared chip memory.
- See how *float* precision affects results and speed, over the current *double* precision.

4.1 Another way to do things ?

Instead of calculating the dipolar interaction in a parallel manner, another way can be devised. This is an idea that occured to me, and I'd like to put it down, in case somebody takes interest in it enough to impement it.

Rougly, it comes down to : each block totally calculates one single site, with a given number of threads. This would permit more flexibility in the lattice size, as there would be two levels of parallellism. The external, as in how many blocks – how many sites, are used to calculate the system. And the internal, as in how threads inside one block work together to calculate the interactions very fast.

[3] In fact, any multiple of 32, provided there is enough memory on the processor chip.
[4] which happens quite often, to reduce latency by calculating other things during idle time.

The downside to this idea, lies in the massive amount of micro-managing necessary inside the kernel to make efficient use of the internal parallellism. Coordinating threads inside a block is very difficult (if not impossible), once they have different instruction flows. Indeed, the architecture[5] doesn't like diverging instructions. This tends to really hinder the speed at which the program runs. So loads of optimisation on a kernel basis would be mandatory. And not necessarly successful.

This could be another work for a future programmer in the group.

5 CUDA programming

Here I'll cover some aspects of CUDA programming that are very different from our usual way of seeing code.

The surest way to learn CUDA, is to read the programming guide[6] available on NVIDIA's web site. It contains all the information about the language. Google search can also yield good results, by giving links to forums where people asked questions and such. This here is just an overview.

5.1 Architecture of an NVIDIA graphics card

Graphics cards are composed of numbers of Single Instruction Multiple Thread (SIMT) processors. The calculation units on these processors issue operations to batches of threads that all perform the same operation at the same time, in parallel. For a single tic of the processor, we get 32 operations done, on 32 different sets of data. This is what makes parallel processing so fast on graphics cards : the inherent parallel structre on a hardware level.

The downside of this type of architecture, is that all threads must have the same set of instructions to be able to perform them very fast, all at the same time. If grouped together, two sets of instruction are on the same SIMT processor, half the threads will be idle while the others are calculating, and conversly. This is one of the main issues while writing code.

The 32 threads active per processor is dependent on the graphics card. This value holds for most of NVIDIA's current cards, but isn't true for other companies ones. It must be noted that CUDA only works on NVIDIA cards, as there isn't any standard parallel language yet. These 32 threads are organised in **blocks**. Any size can be chosen for the block, but maxing out the usage of the processor requires the number of threads per block to be a multiple of 32. There is a maximum, also dependent on the card type, that is 512 for the Tesla we have down in the lab.

To further the parallel aproach, thread blocks are organised in a **grid**. Thus, each block is assigned to an operating SIMT processor, and they are executed in parallel, each processor working with its own threads. On the Tesla card, there are 30 multiprocessors, each containing one double precision unit, and 8 float precision units. This gives us a total of 240 processors available for float precision operations, and 30 for double.

[5] see section 5.1 for more information on this

[6] http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

5.2 Code structure

To execute code on the device[7], one must enclose the code in a **kernel** function. It's a function that is called by the host[8] yet executed on the device. But prior to that, it is necessary to copy all necessary data on the card, via a wrapper function. It's a wrapper because it contains all the control structures and memory flow needed by the kernel. This "double function call" makes for quite heavy code. Everything must be specified by hand, be it memory locations, register space usage[9] on the processor chip, or how many blocks will execute simultanuously.

Here is an example code showing this structure :

```
1  #include<various_stuff>
2
3  __global__ void my_kernel( arg3, arg4, ... );
4  __host__ type overhead( arg1, arg2, ... );
5
6  int main( int argc, char** argv ) {
7      (...)
8      type var = overhead( arg1, arg2, ... );
9      (...)
10     return 0;
11 }
12
13 __global__ void my_kernel( arg3, arg4, ... ) {
14     /* various calculations on data */
15 }
16
17 __host__ type overhead( arg1, arg2, ... ) {
18     (...)
19     my_kernel<<< BLOCKS, THREADS >>>( arg3, arg4, ... );
20     (...)
21 }
```

CUDA lacks a proper linker. Therefore, all CUDA related functions must reside in the same file. This is less of an issue for the overhead, which can call and use objects from other files in a C++ fashion. But once we get to the kernel, all the information must be in the file where the kernel is defined. This seems quite restrictive at first, but becomes inevitable when one realises that all calculations must be first transfered to the GPU. Thus, there is nothing else on the card than what we put there. This is why the compiler optimizes all code upon compilation, and linking from objects (even hypothetical CUDA objects) doesn't leave place for compiling effective code.

Before calling *my_kernel* one must copy the relevant data to the device. This is done by a special memory copy function. It serves both ways, and can even copy device to device operations. It only takes linear memory pointers as arguments, that need to be previously allocated.

[7] device is the term used in the documentation to refer to the graphics card

[8] the CPU that executes the main program.

[9] All arguments passed to the function are copied to registers, as are all variables declared in a kernel. If too many registers are used, there occurs dumping to global memory, drastically reducing efficiency.

```
__host__ type overhead( arg1, arg2, ... ) {
    type* host_pointer;
    host_pointer = (type*) malloc( byte_size );
    type* device_pointer;
    cudaMalloc( (void**)&device_pointer, byte_size );
    cudaMemcpy( device_pointer, host_pointer, byte_size,
        cudaMemcpyHostToDevice )
    // cudaMemcpyHostToDevice specifies the direction of the memory copy.
    // it can also be cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice.
    (...)
    my_kernel<<< BLOCKS, THREADS >>>( device_pointer, args, ... );
    (...)
    cudaThreadSynchronize();//wait for end of all CUDA functions.
    cudaMemcpy( host_pointer, device_pointer, byte_size,
        cudaMemcpyDeviceToHost );
    free( host_pointer );
    cudaFree( device_pointer );
}
```

It is important to see that the pointer to device memory must be passed to the kernel. The *cudaMalloc()*, and *cudaFree()* calls work the same as the standard *malloc()* and *free()* functions, yet work on the device memory space. There's a physical difference between the standard memory and the CUDA specific memory, both in hardware, and in software.

Inside a kernel, some specific variables give access to wide range control over data selection and calculations. Some tools augment this control, giving access to very find grained parallisation.

```
__device__ type device_function( input, output ) { (...) }

__global__ void my_kernel( device_pointer, args, ... ) {
    int tidx = blockIdx.x * blockDim.x + threadIdx.x;
    int tidy = blockIdx.y * blockDim.y + threadIdx.y;
    (...)
    __shared__ type my_data[];
    (...)// <- Load data from device_pointer (global) to my_data (local).
    __syncthreads();//wait for all threads in block to finish
    (...)
    device_function( my_data[tidx*size+tidy], results[tidx*size+tidy] );
    (...)
}
```

The variables *threadIdx* and *blockIdx* are coordinates locating the thread inside a block, and the block inside the grid. Each have their own unique numbers, and this enables complex user-defined patterns to be established, for the benefit of the algorythm. They have three arguments, accessed as a regular C-structure with fields *.x .y .z* containing unsigned integers. *blockIdx.z* is always equal to zero, and is never used in the current CUDA implementation.

Noteworthy is the *__syncthreads()* call, that stops all threads in block, and works as a wall. All must be there to proceed. It is usually done when threads mix data from other threads as input, and they must wait for all preceeding calculations to be done, to avoid errors. It also happens during memory loads, to ensure all memory has been loaded before calculations start from it.

A good way to get familiar with how the programming is done, is to read the examples given in the SDK from NVIDIA. There are many different types of applications, all with

their source code. This gives a good understanding of the basics. Some specific programs uncover the shadows from specific areas of the CUDA language.

6 Conclusions

The program is finally working on a CUDA basis. This brings nice improvements to speed for big simulations. I haven't been able to thoroughly compare with all preceeding versions of the $LiHo_{1-x}Y_xF_4$ simulation. The results are the same as those obtained from the version I started out of, even though some key parts of the calculations were changed while going to parallel. There is still some future for this code, as it can be improved, to further reduce the time it takes to make a large-scale simulation.

This work has strongly affected my programming skills. I now spend much less time on wrong paths, and can spot regular errors quite efficiently. On the other hand, intricated errors spanning multiple parameters and locations remain a substantial problem, as they shall remain for all human minds.