A short guide to CUDA C

For physicists with multi-core graphics cards

This is a theoretical at the
Computer-Oriented Quantum Field Theory Research Group

Institute for Theoretical Physics

Faculty of Physics and Earth Sciences

University of Leipzig

David Plotzki

Leipzig, October 16, 2012

Contents

1. Introduction

1.1. About this document

The purpose of this guide is to give a quick introduction to *CUDA C*, NVIDIA's extension for the C programming language to allow running code parallely on graphics cards. Moreover, the technique is applied to problems in computational physics, namely the generation of random numbers and simulations of the q-state Potts model.

If you seek a complete documentation with more profound information, please refer to the *NVIDIA CUDA C Programming Guide* [1]. CUDA is also available for Fortran.

The following is based on version 4.2 of the CUDA toolkit. Newer versions will most probably be compatible to the methods and code shown here. It is assumed that you already have CUDA installed and running. For help, please refer to NVIDIA's *Getting Started Guide* [2] for your operating system.

All code can be found online: `http://www.physik.uni-leipzig.de/~plotzki/cuda/`

1.2. Modern graphics cards

Graphics rendering is a task which can be divided into several processes that run independently from each other. To carry out its instructions, each process only requires a little piece of the whole picture. Because of this nature, all these processes can run in arbitrary order and, more importantly, at the same time. That is why the processors on graphics cards are designed to run several of these processes, so-called *threads*, at once. To achieve this, they have far more processing cores than a regular CPU.

1.3. CUDA – the link between hardware and software

NVIDIA's CUDA (Compute Unified Device Architecture) provides an interface for developing parallel code and running it on compatible graphics devices. For this purpose, CUDA provides an abstraction layer between the programmer and the hardware. Using CUDA's extensions for languages such as C and Fortran, users are able to write code that runs simultaneously on a certain number of processors on the graphics card. There are tools to provide each thread with the data it needs, as well as functions to synchronize data between threads. Once the problem is formulated, CUDA will map it to the actual hardware and execute it there.

Even though this abstraction layer allows for running parallel code on unknown hardware, the programmer should nevertheless know the graphic card's specifications very well in order to optimize the program.

2. The hardware

2.1. Compute capability and GPU specifications

NVIDIA distinguishes CUDA compatible graphics cards by their *Compute Capability*. For example, a *Tesla C1060* device has Compute Capability 1.3. Devices of a certain Compute Capability have some important features and specifications in common that the programmer can always rely on. For example, devices with Compute Capability 1.3 and higher support double-precision floating point numbers.

For a detailed overview, see Appendix G in the *NVIDIA CUDA C Programming Guide* [1].

In the following, a CUDA compatible device will simply be called *CUDA device*.

To get the technical specifications of your CUDA device, you can run the `deviceQuery` program from the *NVIDIA GPU Computing SDK* [3]. For a *Tesla C1060*, the output looks like this:

```
Device 0: "Tesla C1060"
  CUDA Driver Version / Runtime Version          4.0 / 4.0
  CUDA Capability Major/Minor version number:    1.3
  Total amount of global memory:                 4096 MBytes (4294770688 bytes)
  (30) Multiprocessors x (  8) CUDA Cores/MP:    240 CUDA Cores
  GPU Clock rate:                                1296 MHz (1.30 GHz)
  Memory Clock rate:                             800 Mhz
  Memory Bus Width:                              512-bit
  Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(8192) x 512, 2D=(8192,8192) x 512
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   No
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      No
```

The following sections will explain most of these technical parameters.

2.2. GPU structure

2.2.1. Multiprocessors and warps

Structure of a multiprocessor (MP)

CUDA devices come with a certain number of *multiprocessors*. A multiprocessor is a SIMD unit (single instruction, multiple data) that houses a number of scalar processors (so-called *CUDA Cores*), special purpose processors and several kinds of memory.

For CUDA devices with Compute Capability 1.x, each multiprocessor (fig. 2.1) has

- **8 CUDA Cores:** scalar processors for **integer and single-precision** floating-point arithmetic,
- **1 double-precision** floating-point arithmetic unit, and
- **2 special units** for **transcendental functions** such as $\exp(x)$ or $\sin(x)$.

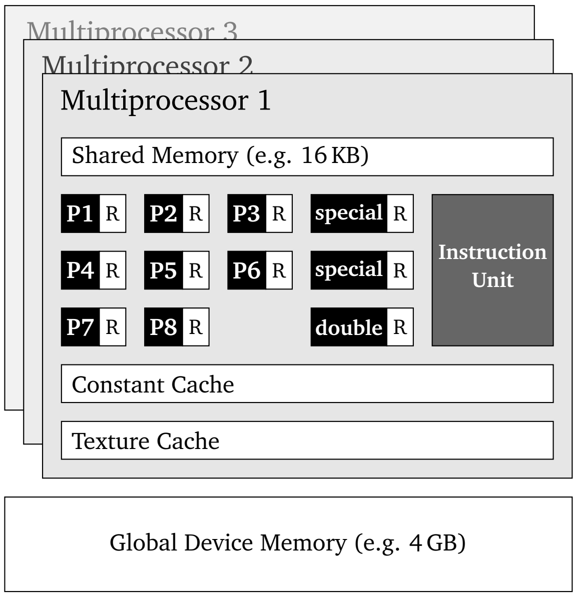

Figure 2.1.: General structure of a CUDA Device with 8 CUDA Cores per multiprocessor. The letter R denotes processor registers.

Warps

Multiprocessors usually execute a whole *warp* of threads at once. The *warp size* specifies the number of threads within a warp. Up to the date of this document, all CUDA devices operate with a warp size of 32, i.e. this number is fixed for devices with compute capability 1.0 through 3.0.

The time it takes to execute a warp depends on what kind of instruction the threads need to carry out. For example it takes

- **4 clock cycles** for an integer or single-precision floating-point instruction such as `rcp` (reciprocal) or `rsq` (reciprocal square root),

- **4 clock cycles** for 24-bit integer multiplication,
- **16 clock cycles** for 32-bit integer multiplication,
- **16 clock cycles** for the $\log(x)$ function on single-precision floating-point numbers,
- **32 clock cycles** for $\exp(x)$ or $\sin(x)$ on single-precision floating-point numbers, and
- **32 clock cycles** for a double-precision floating-point instruction.

2.2.2. Memory

CUDA Devices come with different kinds of memory. The main difference which matters most is how close they are to the actual processors since this determines how fast the data can be accessed by the threads. A rule of thumb is: the bigger the size of a certain kind of memory, the more time it takes to access it.

Global memory

This is the main memory of the graphics card. It can be accessed from the host program that runs on the CPU and all processors on the GPU. Read and write accesses from the CUDA Cores take about 200. . . 300 clock cycles each. It's a very slow memory, but it holds a lot of data.

Shared memory

The shared memory can only be accessed by the CUDA Cores, the special instruction units and the double-precision units that share a common **multiprocessor.** For our example, each multiprocessor has 16 KB of shared memory. It is designed for data exchange between all the threads that one multiprocessor executes.

Access times are much shorter than for the global memory.

Registers

Each CUDA core comes with its own registers to store variables locally. This is the fastest memory, but it is also most limited in size. Since a processor can only access its own registers, they can't be used for shared data.

3. CUDA C and its programming model

3.1. CUDA C files and compilation

The standard extension for files that contain CUDA C code is `.cu`. Usually, no special header files or libraries are required since `.cu` files are compiled with NVIDIA's proprietary compiler `nvcc`.

Code sections which will run on the CUDA device fully support C but only a limited number of C++ features. Namely, these are polymorphism, default parameters for functions, operator overloading, namespaces and function templates. Devices of Compute Capability 2.x and higher also support classes. For details see Appendix D of the *NVIDIA CUDA C Programming Guide* [1].

The CPU part can be written in complete C++. The following listing shows an example `makefile`. It compiles the source file `main.cu` and creates an executable binary named `square`. Calling `make clean` will remove the compiler's object files and the created binary, allowing for a full recompilation.

```
square:main.o
    nvcc -o square main.o

main.o:main.cu
    nvcc -c main.cu

clean:
    rm *.o *.gch square
```

3.2. Kernels and threads

CUDA introduces an abstract programming model for parallel code. The source code that will be run as one of many parallel threads is called *kernel*. This means that the instructions given in one kernel function will be carried out simultaneously by the CUDA Cores on the Graphics Card.

In CUDA C, kernel functions are preceded by the keyword `__global__`. Functions that will run on the CUDA device but will be called by a main kernel function are preceded by the keyword `__device__`.

3.3. Blocks and grids

Blocks

Kernels are executed as bunches of threads arranged in *blocks*. The size of a block is defined by the programmer and depends on what kind of data will be processed. Blocks of threads can be one-, two- or three-dimensional. For example, multiplying two 3×3 matrices would best be done with a two-dimensional 3×3 block, where each thread computes one element of the resulting matrix.

Each thread is identified by its coordinate within the block. A kernel function retrieves this coordinate from CUDA through provided variables and thus knows its position. Examples of block structures are shown in Fig. 3.1

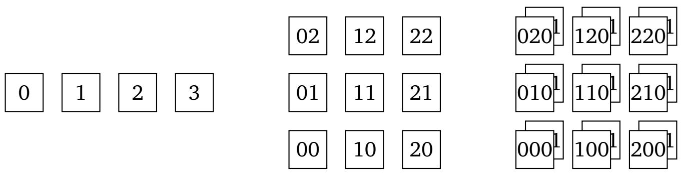

Figure 3.1.: Examples of one-, two- and three-dimensional blocks of threads. Each thread has a unique coordinate within the block.

Grids

Grids are one- or two-dimensional structures that hold blocks of threads. Usually a block is treated by one multiprocessor. Splitting a problem into several blocks and arranging them in a grid tells CUDA to share the workload between several multiprocessors.

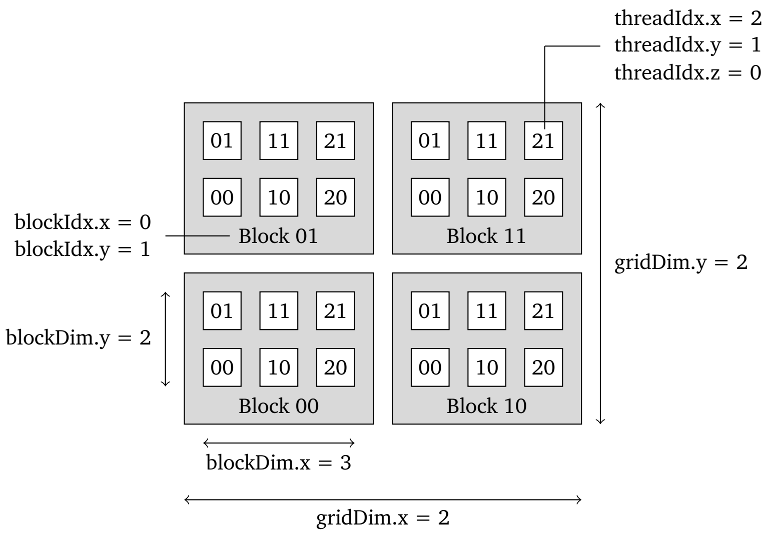

Figure 3.2.: Example of a two-dimensional grid of 2×2 blocks, each containing 3×2 threads. The blocks have a unique coordinate within the grid. The labels show the names by which the coordinates can be accessed in a CUDA C kernel function.

3.3.1. Example: thread and block coordinates

The following code listing shows an example of a kernel function that identifies its position within a two-dimensional grid of three-dimensional blocks and calculates its unique ID, i.e. the thread's sequential number id $\in [0, N-1]$ for a total of N threads.

```
 1  // Kernel:
 2  void __global__ getPosition()
 3  {
 4      int x  = blockIdx.x * blockDim.x + threadIdx.x;
 5      int y  = blockIdx.y * blockDim.y + threadIdx.y;
 6      int z  = threadIdx.z;   // Grids are only 1D or 2D structures, never 3D
 7
 8      int id =  z * (gridDim.x * blockDim.x) * (gridDim.y * blockDim.y)
 9              + y * (gridDim.x * blockDim.x)
10              + x;
11  }
12
13  // CPU Code:
14  int main (int argc, char const* argv[])
15  {
16      // grid and block sizes:
17      dim3 blockSize(128, 64, 8);
18      dim3 gridSize(5, 6);
19
20      // execute kernel function on GPU:
21      getPosition<<<gridSize, blockSize>>>();
22
23      return 0;
24  }
```

CUDA provides the thread's coordinates through the `threadIdx` and `blockIdx` objects which are available within a kernel function (see fig. 3.2). `threadIdx` contains the members `x`, `y` and `z`, specifying the thread's three coordinates within its block. `blockIdx` contains the members `x` and `y` which give the position of the thread's block within the grid.

`blockDim` provides the block's width, height and depth through its members `x`, `y` and `z` while `gridDim` gives the grid's size with its members `x` and `y`.

As can be seen in line 21, the syntax for calling a kernel function and executing it on the GPU is the kernel function's name, followed by two CUDA parameters in triple angle brackets <<<...>>>. These parameters are the grid's size and the size of a block. Both can be integers (for one-dimensional structures) or objects of type `dim3`. This example creates a 5×6 grid of blocks, and each of these blocks has a size of $128 \times 64 \times 8$ threads.

Here, the total number of blocks in the grid is 30. Since this is the number of multiprocessors of our example CUDA Device, each one can handle a whole block. The number of threads per block is a multiple of 32, the warp size. This means the task is nicely balanced.

3.4. Exchanging data between host and device

CUDA provides the following functions for data exchange between the host and the CUDA device.

- `cudaMalloc(void** pDevice, size_t size)`
  This function allocates `size` bytes in the global device memory (see fig. 2.1). The function will store the address of the allocated memory in the pointer `pDevice`, therefore a pointer to that pointer must be provided.

- `cudaMemcpy(void* pTo, const void* pFrom, size_t size, direction)`
  This function copies `size` bytes of data from `pFrom` to `pTo`. The first two parameters are pointers to the address spaces on device or host. The last parameter is a keyword which specifies the direction. It can be:

  – `cudaMemcpyHostToDevice` to copy data from the host system to the device,

  – `cudaMemcpyDeviceToHost` to copy data from the device to the host,

  – `cudaMemcpyDeviceToDevice` to copy data within the device, or

  – `cudaMemcpyHostToHost` to copy data within the host's memory.

- `cudaFree(void* pDevice)`
  When you don't need the memory at `pDevice` anymore, use this function to deallocate it. As the function name implies, this works only for memory on the CUDA device. To deallocate host memory, employ the standard C/C++ methods such as `free()` or `delete`.

3.4.1. Example: squaring an array of numbers

This is a simple demonstration of exchanging data between host and device. An array of integer numbers is copied to the CUDA Device, each number is squared by a thread, and then the squared numbers are copied back to the host's RAM.

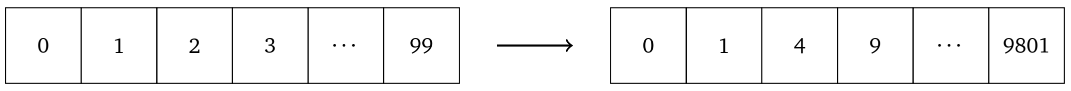

Figure 3.3.: Squaring an array of numbers.

```
#include <iostream>

// Kernel:
__global__ void square(float* numbers)
{
    // get the array coordinate:
    unsigned int x  = blockIdx.x * blockDim.x + threadIdx.x;

    // square the number:
    numbers[x] = numbers[x] * numbers[x];
}


```

```
14 // CPU Code:
15 int main (int argc, char const* argv[])
16 {
17     const unsigned int N = 100; // N numbers in array
18
19     float data[N];      // array that contains numbers to be squared
20     float squared[N];   // array to be filled with squared numbers
21
22     // number to be squared will be the index:
23     for(unsigned i=0; i<N; i++)
24         data[i] = static_cast<float>(i);
25
26     // allocate memory on CUDA device:
27     float* pDevData;        // pointer to the data on the CUDA Device
28     cudaMalloc((void**)&pDevData, sizeof(data));
29
30     // copy data to CUDA device:
31     cudaMemcpy(pDevData, &data, sizeof(data), cudaMemcpyHostToDevice);
32
33     // execute kernel function on GPU:
34     square<<<10, 10>>>(pDevData);
35
36     // copy data back from CUDA Device to 'squared' array:
37     cudaMemcpy(&squared, pDevData, sizeof(squared), cudaMemcpyDeviceToHost);
38
39     // free memory on the CUDA Device:
40     cudaFree(pDevData);
41
42     // output results:
43     for(unsigned i=0; i<N; i++)
44         std::cout<<data[i]<<"^2␣=␣"<<squared[i]<<"\n";
45
46     return 0;
47 }
```

After the `data` array is generated, `cudaMemcpy` copies the data into the global memory of the CUDA Device (see fig. 2.1). Each processor reads a number from there, squares it, and then copies the result back to it. Data exchange between the CUDA Cores and the global device memory is relatively slow and inefficient. Whenever a variable is used more often within the kernel function, it should be copied to the multiprocessor's shared memory (see the next section) or the processor registers (by declaring it locally, as done with the array coordinate `x` in line 7). Accessing and altering data there is much faster.

3.5. Using shared memory and synchronizing threads

To place a variable in a multiprocessor's shared memory, the keyword `__shared__` is used. It can then be accessed much faster than in global memory, but only by the threads that run on this exact multiprocessor.

Another important feature is thread synchronization. Since threads work independently from each other, they sometimes need to wait at a certain point until all threads have reached it. This is crucial if the calculations that follow depend on data generated by the other threads.

Such a synchronization point can easily be set in a kernel function by calling `__syncthreads()`. When a thread reaches this function, it will wait for the other threads **in the current block**. This is very important: the function will not synchronize threads throughout different blocks.

3.5.1. Example: Conway's Game of Life

This example illustrates the concepts of shared memory and thread synchronization. Conway's Game of Life [4] is a cellular automaton: it starts with a two-dimensional board of sites. Each site has one of two states: it is either *dead* or *alive*.

For every step in the evolution of this world, a set of four rules is applied simultaneously to each site. The new state of a site is determined by the number of neighbours that are alive (tab. 3.1). For periodic boundary conditions, each site has 8 neighbours: the sites which are adjacent in the horizontal, vertical or diagonal direction. In the language of a lattice model, these are nearest and next-nearest neighbours.

current state	neighbours	new state
alive	<2 alive	dead
alive	2 or 3 alive	alive
alive	>3 alive	dead
dead	3 alive	alive

Table 3.1.: The four rules in Conway's Game of Life.

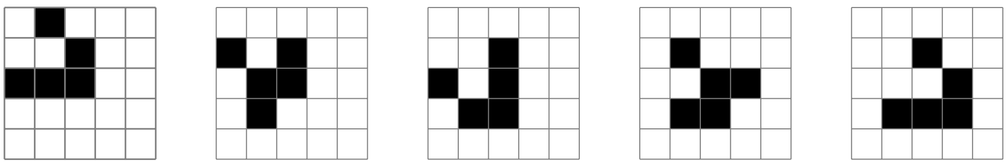

Figure 3.4.: The evolution of a *glider* in Conway's Game of Life. It takes four iterations for the glider to return to its original shape and accomplish one "step". Black sites are *alive*, white ones are *dead*.

To implement this algorithm, we will create a very simple world of 16×16 sites which will only run in one block (i.e. on one multiprocessor). The goal is to keep it simple: splitting the world to run in different blocks would mean that we had to synchronize the site states between the blocks after each iteration.

In the code sample (see section A.1 in the annex for the complete listing) we create a one-dimensional array (line 101) that saves the states of all sites in the two-dimensional world. In this world we set up a little glider (fig. 3.4, lines 111. . . 115).

The world array is copied to the GPU's global memory (line 122), then the kernel function is called on the GPU (line 129). It takes the pointer to the world and the number of iterations as parameters.

The kernel function (line 31) – which is executed by each thread – gets its coordinates from the predefined CUDA variables (lines 34, 35). It calls the function `getId` (line 38) to convert the coordinates to the index in the one-dimensional world array (id). Note that this function also runs on the CUDA device and is therefore preceded by the keyword **`__device__`** (line 12).

Now the world array is copied from the GPU's global memory to the multiprocessor's shared memory (see fig. 2.1). To do this, an array of equal size is created in the memory which is shared by all threads. This is done with the keyword **`__shared__`** (line 41) and implies that it's valid for all threads of the current block. Each thread then copies the state of the site for which it is responsible from the global memory into the shared memory (line 42). Before the actual iterations of the Game of Life are started, all threads must have copied their state. To ensure this, we force each thread to wait until all threads in the block have reached this point by calling the function **`__syncthreads()`** (line 45).

The iterations follow. After each iteration the threads need to wait for each other (line 84) until the new state is determined. The state is then copied back to the shared memory (line 87) and before a new iteration starts, the threads are synchronized again (line 90).

4. Applications in computational physics

4.1. Generating random numbers

4.1.1. Linear congruential random number generators

The requirements on a good random number generator are high: it needs to have a long periodic length and virtually no correlations. Some of the widely used pseudo-random number generators such as the Mersenne-Twister [5] use very large vectors and matrices and therefore consume too much memory to run them efficiently on the graphics card.

CUDA pioneers like Preis *et al.* [6] and Weigel [7] used an array of linear congruential random number generators (LCRNGs) with a period of 2^{32}. Since each LCRNG is initialized with a seed taken from another LCRNG, overlaps between the series of the individual generators cannot be confidently avoided.

4.1.2. Combined Tausworthe generator

Howes and Thomas [8] implement a combination of a Tausworthe generator [9] with a period of 2^{88} and an LCRNG (period 2^{32}), resulting in a generator with a period of about 2^{120}. Even here, overlaps cannot be confidently avoided when using random seeds, but the probability for such an event is much smaller than for a pure LCRNG due to the bigger periodic length.

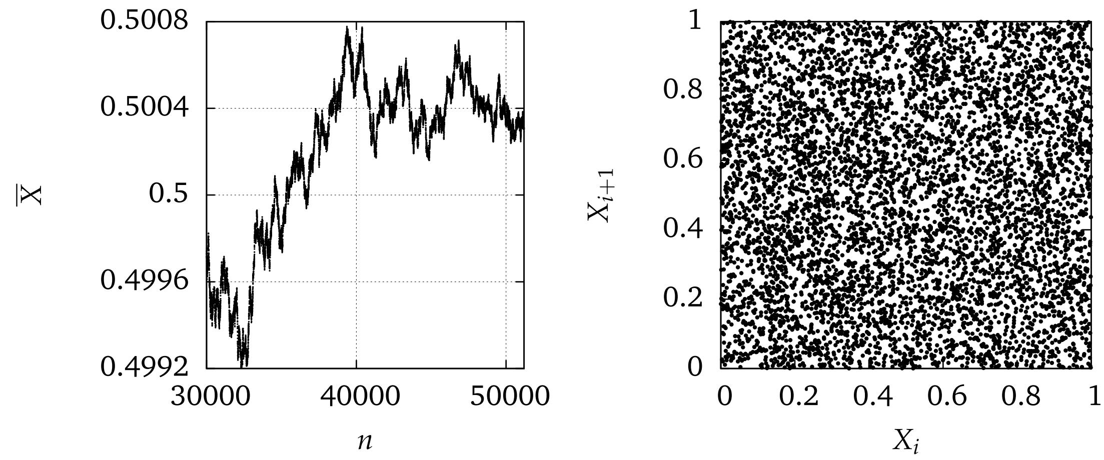

Figure 4.1.: Left: Running average $\overline{X}(n) = \sum_{i=1}^{n} X_i/n$ for $n = 30\,000 \ldots 51\,200$ of the used Tausworthe generator. Right: plot of $x = X_i$, $y = X_{i+1}$, $i = 1, 3, 5 \ldots$ for 6400 number pairs from the used Tausworthe generator.

```
#include <iostream>
#include <cstdlib>

#define N_BLOCKS            32  // number of blocks
#define N_THREADS_PER_BLOCK 32  // number of threads per block
#define N_RN                (50 * N_BLOCKS * N_THREADS_PER_BLOCK)

#define ITERATIONS          10000 // number of runs

__device__ unsigned Tausworthe88(unsigned &z1, unsigned &z2, unsigned &z3)
{
    // Three-step generator with period 2^88
    unsigned b = (((z1 << 13) ^ z1) >> 19);
    z1 = (((z1 & 4294967294) << 12) ^ b);

    b = (((z2 << 2) ^ z2) >> 25);
    z2 = (((z2 & 4294967288) << 4)  ^ b);

    b = (((z3 << 3) ^ z3) >> 11);
    z3 = (((z3 & 4294967280) << 17) ^ b);

    return z1 ^ z2 ^ z3;
}

__device__ unsigned LCRNG(unsigned &z)
{
    const unsigned a = 1664525, c = 1013904223;
    return z = a * z + c;
}

__device__ float TauswortheLCRNG(unsigned &z1, unsigned &z2, unsigned &z3, unsigned &z)
{
    // combine both generators and normalize 0...2^32 to 0...1
    return (Tausworthe88(z1, z2, z3) ^ LCRNG(z)) * 2.3283064365e-10;
}

// Kernel:
__global__ void createRandomNumbers(float* seeds, float* randoms)
{
    int x  = blockIdx.x * blockDim.x + threadIdx.x;

    // get seed values; 2^32-1 = 4294967295; using CUDA type casting to unsigned int
    unsigned z1 = __float2uint_rn(4294967295 * seeds[4*x    ]); // Tausworthe seeds
    unsigned z2 = __float2uint_rn(4294967295 * seeds[4*x + 1]);
    unsigned z3 = __float2uint_rn(4294967295 * seeds[4*x + 2]);
    unsigned z  = __float2uint_rn(4294967295 * seeds[4*x + 3]); // LCRNG seed

    // number of RNs to produce in this thread:
    const unsigned n  = N_RN / N_BLOCKS / N_THREADS_PER_BLOCK;

    // create RNs and immediately copy them to global memory:
    for(unsigned i = 0; i<n; i++)
        randoms[n*x + i] = TauswortheLCRNG(z1, z2, z3, z);
}


```

```
// Host function (CPU Code)
int main (int argc, char const* argv[])
{
    // create 4 seed values for each thread:
    float seeds[4 * N_BLOCKS * N_THREADS_PER_BLOCK];
    for(unsigned i=0; i<(4 * N_BLOCKS * N_THREADS_PER_BLOCK); i++)
         seeds[i] = drand48();

    // create array that will finally get all random numbers from the CUDA device:
    float randoms[N_RN];

    // allocate memory on CUDA device:
    float* pDevSeeds;
    cudaMalloc((void**)&pDevSeeds, sizeof(seeds));

    float* pDevRandoms;
    cudaMalloc((void**)&pDevRandoms, sizeof(randoms));

    // copy seeds to CUDA device:
    cudaMemcpy(pDevSeeds, &seeds, sizeof(seeds), cudaMemcpyHostToDevice);

    for(unsigned i=0; i<ITERATIONS; i++)
    {
        // execute kernel function on GPU:
        createRandomNumbers<<<N_BLOCKS, N_THREADS_PER_BLOCK>>>(pDevSeeds, pDevRandoms);

        // copy data back from CUDA Device to 'data' array:
        cudaMemcpy(&randoms, pDevRandoms, sizeof(randoms), cudaMemcpyDeviceToHost);

        // ... do stuff with generated random numbers before next iteration ...
    }

    // free memory on the CUDA Device:
    cudaFree(pDevSeeds);
    cudaFree(pDevRandoms);

    return 0;
}
```

This code sample uses the CUDA cores to generate 100 random numbers in each thread (line 3). For 16 blocks with 32 threads each, this gives 51 200 random numbers per kernel call. Once the seeds have been copied to the memory of the graphics card, the kernel function can be called as often as needed. The generated numbers have to be copied back to the host system after each kernel call.

For 10 000 kernel calls (generation of 512 million random numbers) and subsequent memory transfer operations, the CUDA code ran approximately four times faster on the GPU than on a single CPU core on the test system (tab. 4.1).

	Cores	Execution time
NVIDIA NVS3100M, 1.47 GHz	16 CUDA Cores on 2 MPs	(3.4 ± 0.1) s
Intel Core i7 M620, 3.0 GHz	1 CPU Core	(14.0 ± 0.1) s

Table 4.1.: Timing results for generating 512 million random numbers.

4.2. Lattice models and Markov chain Monte Carlo techniques

4.2.1. The Potts model as an example of a lattice model

The Potts model [10, 11] is a lattice model where each site of the lattice is occupied by a spin whose state is represented by an integer in the range $[0, q-1]$ (see fig. 4.2). Therefore, it is more specifically called *q-state Potts model*, since each spin is in one of q possible states. The spins interact with each other; their coupling energy is denoted by J. The system's Hamiltonian is

$$H = -J \sum_{\langle i,j \rangle} \delta_{s_i s_j} \; . \tag{4.1}$$

In this notation, $\langle i, j \rangle$ stands for nearest-neighbour interaction (for each site i in the sum, only the spin's interactions with its nearest neighbours j are added up), δ means the Kronecker delta and s_i is the value of spin i. This means that only such neighbouring spins contribute to the total energy which are in the same state.

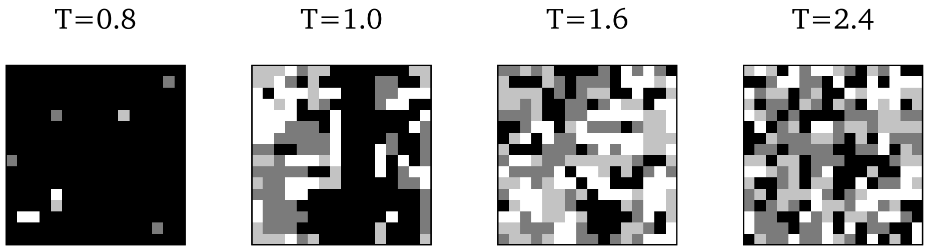

Figure 4.2.: Examples for spin configurations of the Potts model for $q = 4$ states on a square lattice of side length $L = 16$ at different simulation temperatures T. The Boltzmann constant is assumed to be $\mathrm{k_B} = 1$, resulting in dimensionless energies and temperatures $T = \mathrm{k_B} T_{\text{abs}}/J$. The coupling constant is $J = 1$. The Metropolis algorithm has been used to create the samples. Starting with a random configuration, the samples were taken after 10 000 sweeps. The spin values are colour-coded from 0 (black) to 3 (white).

4.2.2. Metropolis algorithm

The Metropolis algorithm [12] can be used to approximate the equilibrium state of a system and also to gradually sample its phase space to get densities of states for observables such as the energy or an approximation for the partition function.

The probability to find a spin configuration $\{s\}$ at a given temperature $\mathrm{k_B} T = 1/\beta$ obeys the Boltzmann distribution (we assume $\mathrm{k_B} = 1$)

$$P^{\text{eq}}\left[\{s\}\right] = \frac{\mathrm{e}^{-\beta H[\{s\}]}}{Z(T)} = \frac{\mathrm{e}^{-\beta H[\{s\}]}}{\sum\limits_{E} \Omega(E) \mathrm{e}^{-\beta E}} \; , \tag{4.2}$$

where $Z(T)$ denotes the partition function and $\Omega(E)$ the energy density of states.

The Metropolis algorithm now performs a random walk through the system's phase space along a reversible Markov chain. This means that starting from a given spin configuration $\{s\}$, a new configuration $\{s'\}$ is suggested and the algorithm accepts this new configuration with a certain transition probability W:

$$\cdots \xrightarrow{W} \{s\} \xrightarrow{W} \{s'\} \xrightarrow{W} \{s''\} \xrightarrow{W} \{s'''\} \xrightarrow{W} \cdots \ . \tag{4.3}$$

The algorithm has to make sure that it "visits" each spin configuration with a probability that complies with the Boltzmann distribution (4.2). Furthermore, it must obey detailed balance, which means microscopic reversibility [13] must be given:

$$W\left(\{s\} \to \{s'\}\right) \cdot P^{\mathrm{eq}}\left[\{s\}\right] = W\left(\{s'\} \to \{s\}\right) \cdot P^{\mathrm{eq}}\left[\{s'\}\right] \ . \tag{4.4}$$

The Metropolis algorithm accomplishes this behaviour by randomly choosing a single lattice site and suggesting a random new value for its spin. The spin is flipped with the probability

$$W = \min\left[\mathrm{e}^{-\beta(H'-H)}, 1\right] \tag{4.5}$$

which depends on the energy $H = H\left[\{s\}\right]$ of the current configuration and the energy $H' = H\left[\{s'\}\right]$ of the suggested new configuration.

4.2.3. Implementation in CUDA C

While a common implementation on a CPU would consider only one spin at a time, the advantage of a multi-core architecture is its ability to propose many spin flips simultaneously. However, since the energy in the Potts model (4.1) depends on the state of adjacent spins, we cannot simultaneously propose flips of any neighbouring, interacting spins. The sampled configurations would not obey the Boltzmann distribution (4.2) anymore and detailed balance (4.4) would be broken.

A solution proposed by Preis *et al.* [6] decomposes the lattice into two sets of non-interacting spins. For nearest-neighbour interactions they are arranged in a checkerboard configuration (fig. 4.3) and in the following will be called *black* and *white* spins. Note that their "colour" says nothing about the spin's state. It is just another property to distinguish the spins that do not interact.

For the complete code listing please see the file *potts.cu* or section A.2 in the annex. A simplified flowchart of the program is shown in figure 4.8. This implementation keeps track of the spins by maintaining two lists of spin IDs: one array for the *black* spins, the other one for the *white* spins. The spin states themselves are stored in a third array, with the array index corresponding to the spin ID. The spins are initialized on the CPU with a random state (lines 149…172).

For the simulation, the CUDA kernel function that carries out the Metropolis algorithm is called twice during each sweep (lines 270…274). Once for all the *black* sites and once for all the *white* sites (fig. 4.4).

This ensures that only non-interacting spins are flipped simultaneously. Since the spin values are stored in a one-dimensional spin list and their neighbours are managed with a

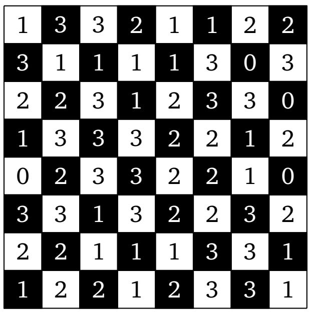

Figure 4.3.: Checkerboard decomposition of an 8×8 spin lattice into two sets of non-interacting spins (*black* and *white*). The numbers denote the example spin states from the lower left section of the 4-state Potts model lattice at $T = 1.6$ in figure 4.2.

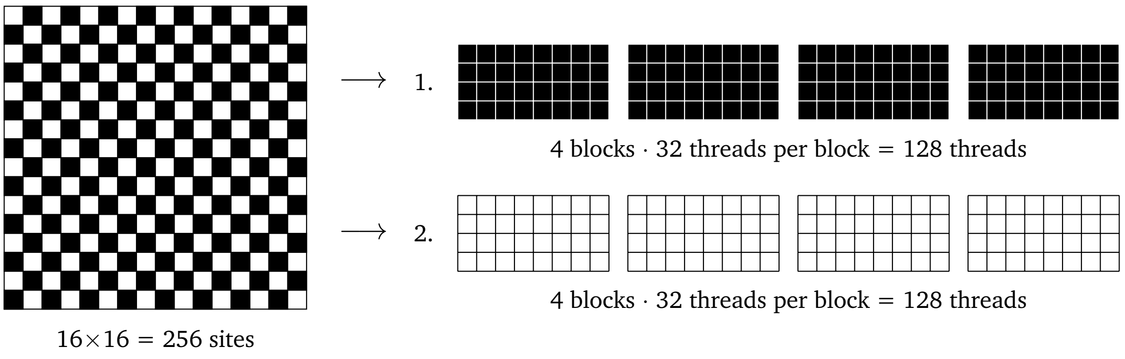
1.
4 blocks · 32 threads per block = 128 threads
2.
4 blocks · 32 threads per block = 128 threads
16×16 = 256 sites

Figure 4.4.: Checkerboard decomposition and assignment of the sites to blocks and threads. In this example, the Metropolis algorithm for a lattice of 16×16 sites is carried out in two parts. At first, a kernel function is called on the CUDA device to handle all the *black* sites. After the call is finished, another kernel carries out the Metropolis algorithm for all the *white* sites. Each thread is responsible for one (potential) spin flip.

simple neighbourhood table, there is no need to represent the actual lattice geometry on the GPU. Each spin is assigned to a thread; they are distributed in such a way that each block takes care of a number of spins that is a multiple of 32 (the warp size).

Before the actual Metropolis algorithm starts, the energy of the sample lattice is calculated once on the CPU (lines 213. . . 223). To calculate the energy after spin flips have been made throughout the Metropolis algorithm, a kind of bookkeeping is introduced which considers differences of the energy to the initial configuration. Whenever a spin is flipped, the resulting change in energy is recorded on a list. This minimizes the data transfer between the host CPU and the CUDA device, since after each sweep only one energy difference per block needs to be transferred (lines 279. . . 286).

To verify the results from the CUDA simulation, the values for the mean energy $\langle E\rangle(T)$ and the specific heat capacity $C(T)$ are compared to the exact results derived from Beale's exact solution [14] for the density of states $\Omega(E)$ of the two-dimensional Ising model (fig. 4.6).

The Ising model [15] is a spin model similar to the Potts model. Each spin on the lattice is in one of two possible states: $s_i = \pm 1$. For nearest-neighbour interaction $\langle ij \rangle$ the system's Hamiltonian is

$$H_{\text{Ising}} = -J \sum_{\langle ij \rangle} s_i s_j + h \sum_i s_i \; . \tag{4.6}$$

Again, J denotes the coupling energy and s_i the value of spin i. The system's symmetry can be broken with an external magnetic field h, but in this example we consider it to be zero: $h = 0$.

Due to this analogous nature, observables of the Potts model for two spin states ($q = 2 \rightarrow s_i \in \{0, 1\}$) can be linked to the Ising model's observables. We consider the temperature $T = 1/\beta$, the mean energy $\langle E \rangle(T)$ and the specific heat capacity $C(T)$:

$$T_{\text{Ising}} = 2T_{\text{Potts}} \; , \tag{4.7}$$

$$E_{\text{Ising}} = 2E_{\text{Potts}} + 2JN \; , \tag{4.8}$$

$$C_{\text{Ising}} = \beta^2_{\text{Ising}} \left[\langle E^2_{\text{Ising}} \rangle - \langle E_{\text{Ising}} \rangle^2 \right] = \beta^2_{\text{Potts}} \left[\langle E^2_{\text{Potts}} \rangle - \langle E_{\text{Potts}} \rangle^2 \right] = C_{\text{Potts}} \; . \tag{4.9}$$

The energy conversion depends on the number of spins N. The constant $2JN$ does not play a role for the specific heat capacity C which is conceptually the energy's variance.

Additionally, the Binder parameter U_4 [16] has been measured for two-dimensional (fig. 4.6) and three-dimensional (fig 4.7) lattices. It is defined as

$$U_4(T) = 1 - \frac{\langle m(T)^4 \rangle}{3 \langle m(T)^2 \rangle^2} \tag{4.10}$$

and takes higher moments of the magnetization m into account. For the Ising model, the magnetization is the sum of all spin values:

$$m = \frac{M}{N} = \frac{1}{N} \sum_{i=1}^{N} s_i \; . \tag{4.11}$$

To calculate the magnetization, the spin values from the 2-state Potts model were remapped to the values of the Ising model:

$$s_{\text{Potts}} = 0 \longleftrightarrow s_{\text{Ising}} = -1 \; , \tag{4.12}$$

$$s_{\text{Potts}} = 1 \longleftrightarrow s_{\text{Ising}} = +1 \; . \tag{4.13}$$

4.2.4. Execution times

The most important aim of parallel processing is to save time. Therefore, it is useful to compare the execution times of a program running on the CPU directly to the one running on the CUDA device.

Two test systems have been used. The first was the compute cluster *Grawp* at the Institute for Theoretical Physics (ITP) at the University of Leipzig, specifically one Intel Xeon E5520 processor[1] to measure the CPU computation times and an NVIDIA Tesla C1060[2] as a CUDA device.

The second test system was a ThinkPad T410 with an Intel Core i7 M620 CPU[3] and an NVIDIA NVS3100M[4] graphics card.

The parallel code has been rewritten to run within a single thread on a CPU. The CPU program was built using the C++ compiler from *gcc 4.4.4* with *O1* optimization for a 64 bit x86 Linux architecture. The same parameters applied to the compilation of the CUDA version: NVIDIA's *nvcc 4.2* compiler uses *gcc 4.4.4* for the C code.

The execution times were measured with the Linux `time` command, adding the times the programs spent in user and kernel (sys) mode.

Each program was initialized with a random lattice and then ran a simulation at 45 different temperatures ($T_{\text{Ising, 2D}} = 0.8 \ldots 5.2$ and $T_{\text{Ising, 3D}} = 2.3 \ldots 6.7$, both in steps of $\Delta T = 0.1$), carrying out 50 000 sweeps for each temperature.

The measured execution times are shown in table 4.2 along with the thread and block arrangements. Figure 4.5 shows the same values on a logarithmic scale. The times are given in execution time per spin, in the following defined as

$$\tau = \frac{\text{total program execution time}}{N_{\text{spins}} \cdot N_{\text{sweeps}} \cdot N_{\text{temperatures}}} \, . \tag{4.14}$$

Whenever possible, the number of threads per block was set to be a multiple of 32, the warp size, to take advantage of block cycles. The optimal distribution of blocks and threads per block depends on factors such as the amount of shared memory required per block and the number of registers per thread. NVIDIA offers a multiprocessor occupancy calculator [17] to help with the fine-tuning.

[1]Intel Xeon E5520, 2.27 GHz, 8 MB L3 cache, 4×256 KB L2 cache
[2]NVIDIA Tesla C1060, 1.30 GHz clock rate, 240 CUDA Cores on 30 multiprocessors
[3]Intel Core i7 M620, 3.0 GHz, 4 MB L3 cache, 2×256 KB L2 cache
[4]NVIDIA NVS3100M, 1.47 GHz clock rate, 16 CUDA Cores on 2 multiprocessors

				Grawp		ThinkPad	
	L	blocks	threads/block	τ_{CPU}[1]	τ_{CUDA}[2]	τ_{CPU}[3]	τ_{CUDA}[4]
2D square	2	1	2	100.0 ns	7911.1 ns	57.8 ns	9133.3 ns
	4	1	8	94.4 ns	1922.2 ns	56.9 ns	2330.6 ns
	8	1	32	95.1 ns	527.1 ns	56.5 ns	672.9 ns
	16	4	32	93.9 ns	133.2 ns	55.5 ns	178.0 ns
	32	16	32	94.7 ns	33.9 ns	55.4 ns	63.0 ns
	64	64	32	94.7 ns	10.7 ns	57.5 ns	35.9 ns
	128	128	64	94.5 ns	5.3 ns	57.6 ns	28.1 ns
	256	256	128	94.5 ns	4.4 ns	64.0 ns	25.7 ns
3D cubic	2	1	4	105.6 ns	4011.1 ns	63.9 ns	4827.8 ns
	4	1	32	100.7 ns	559.7 ns	61.8 ns	706.3 ns
	8	8	32	101.6 ns	71.6 ns	60.4 ns	115.1 ns
	16	64	32	101.4 ns	12.5 ns	60.4 ns	50.1 ns
	32	128	128	102.4 ns	6.7 ns	60.5 ns	40.4 ns

Table 4.2.: Execution time τ per spin of the single CPU and the CUDA implementations for the simulation of the Metropolis algorithm on the 2-state Potts model, as measured by the Linux `time` command. The measurements were taken on two different test systems (see text for details).

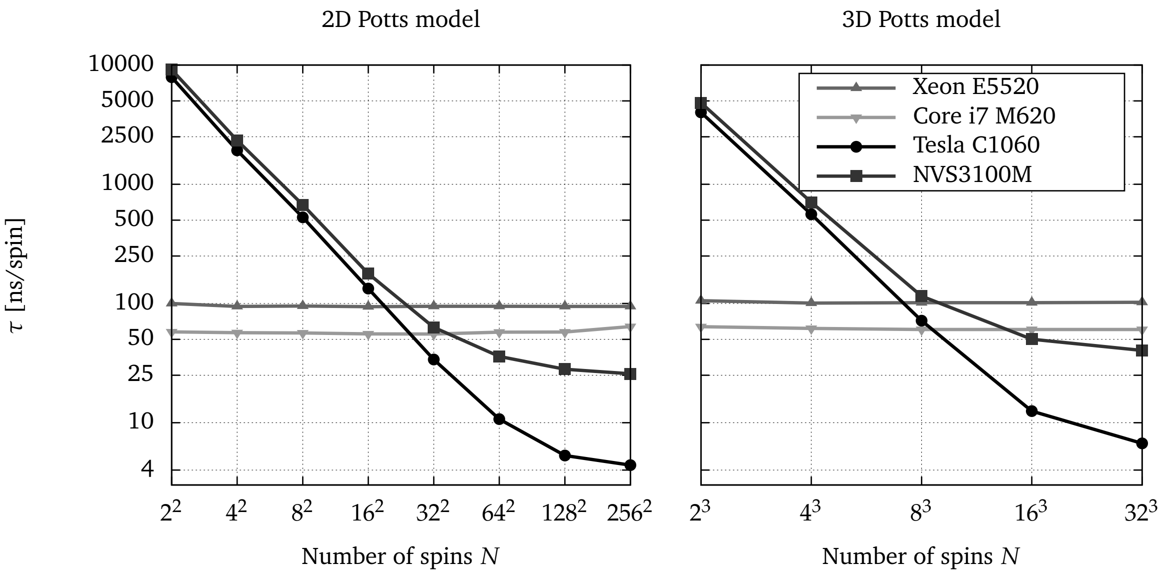

Figure 4.5.: Execution time τ per spin of the single CPU and the CUDA implementations for the simulation of the Metropolis algorithm on the 2-state Potts model, as measured by the `time` command. The values are listed in table 4.2. The time scales are the same for both graphs.

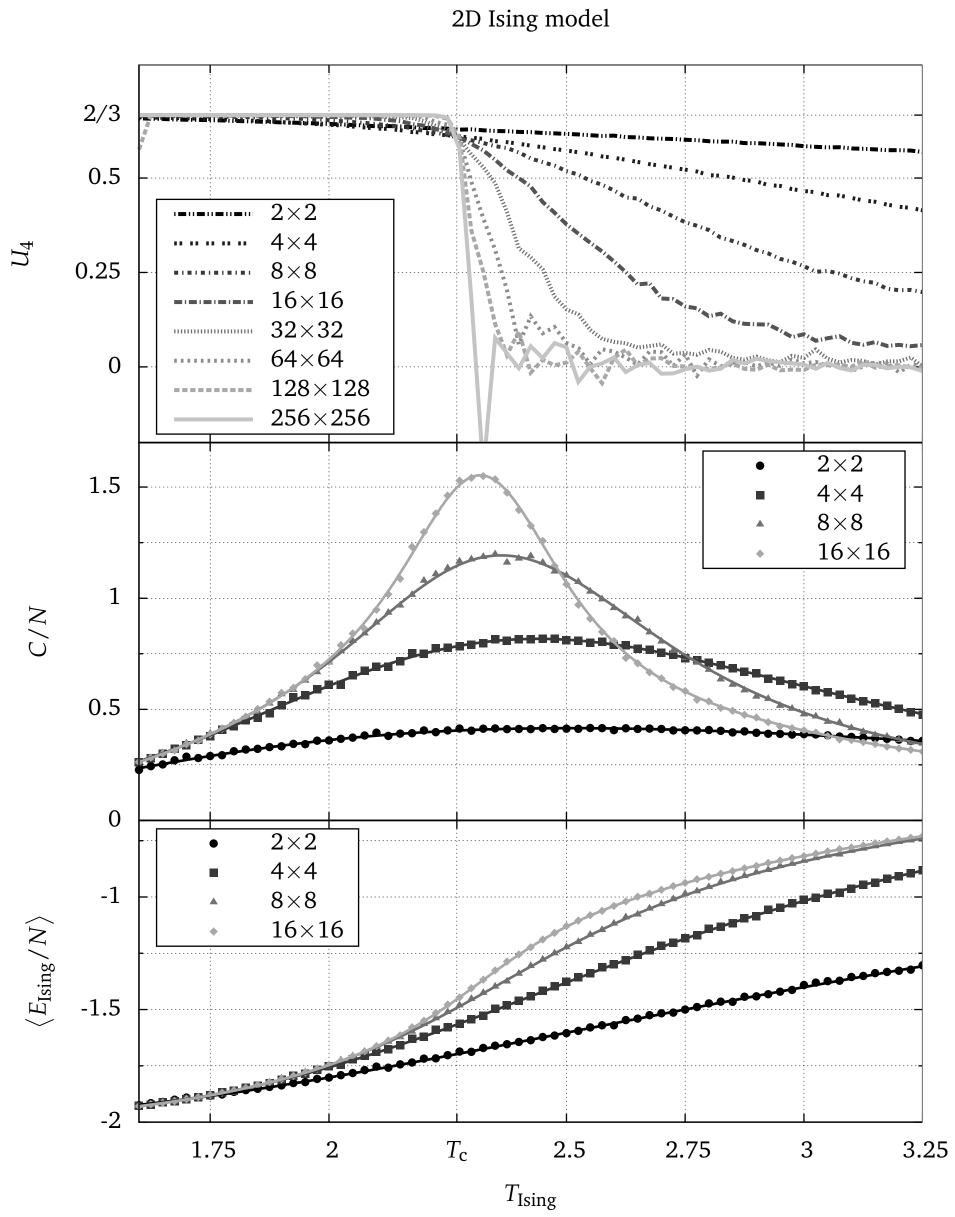

Figure 4.6.: Binder parameter U_4, specific heat capacity C and mean energy $\langle E \rangle$ per spin on different lattice sizes, as measured during the Metropolis simulation of the 2D Potts model ($q = 2$) on the CUDA device. In the diagrams for C and $\langle E \rangle$, the points denote the measurements whereas the solid lines represent the values derived from Beale's exact solution [14] for the density of states. The simulation performed 30 000 sweeps per temperature; the observables were measured after each sweep, starting at the 6 000th sweep. For the 2D Ising model the critical temperature is found at $k_B T_c / J \approx 2.269$ [18].

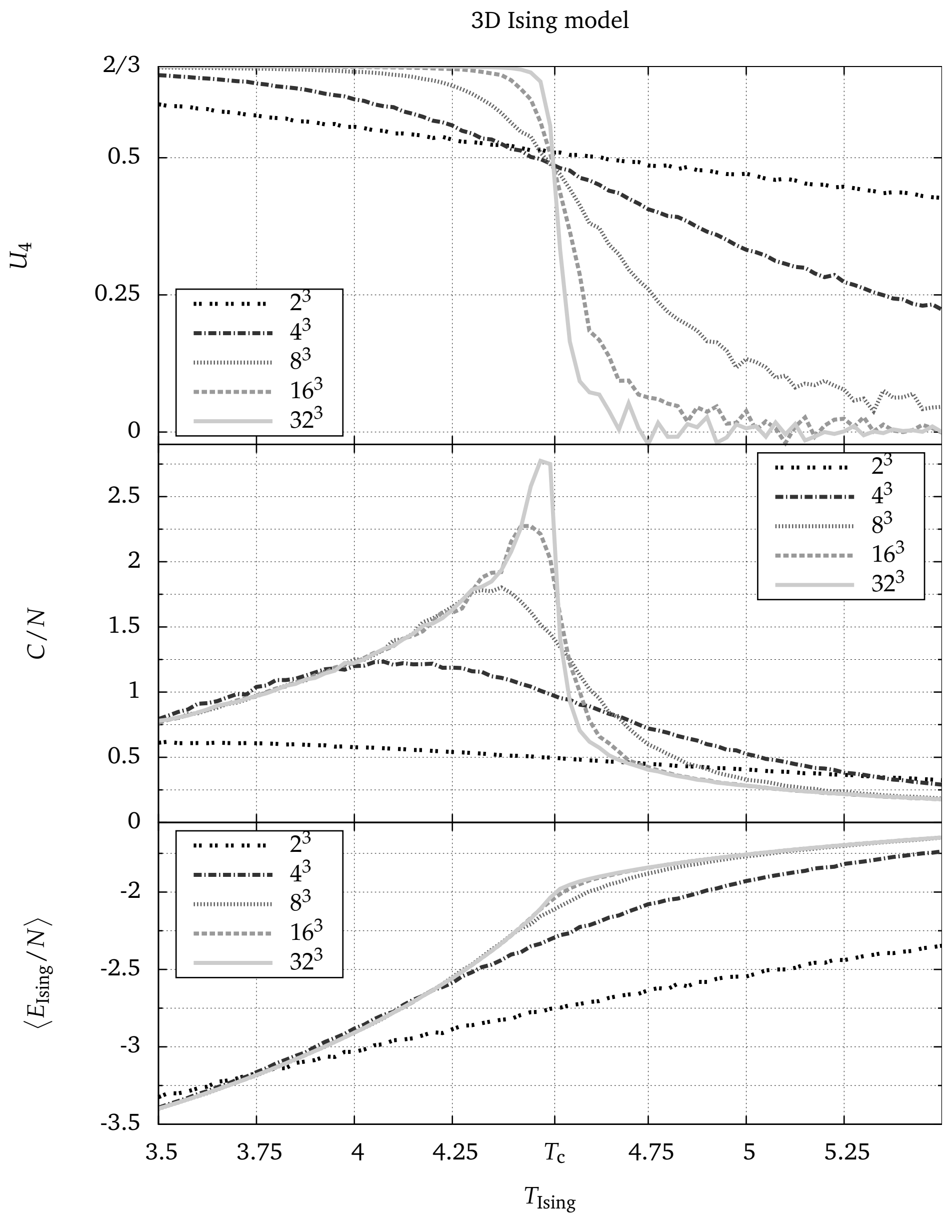

Figure 4.7.: Binder parameter U_4, specific heat capacity C and mean energy $\langle E \rangle$ per spin on different lattice sizes, as measured during the Metropolis simulation of the 3D Potts model ($q = 2$) on the CUDA device. The simulation performed 200 000 sweeps per temperature; the observables were measured after each sweep, starting at the 40 000th sweep. For the 3D Ising model the critical temperature is found at $k_B T_c / J \approx 4.512$ [19].

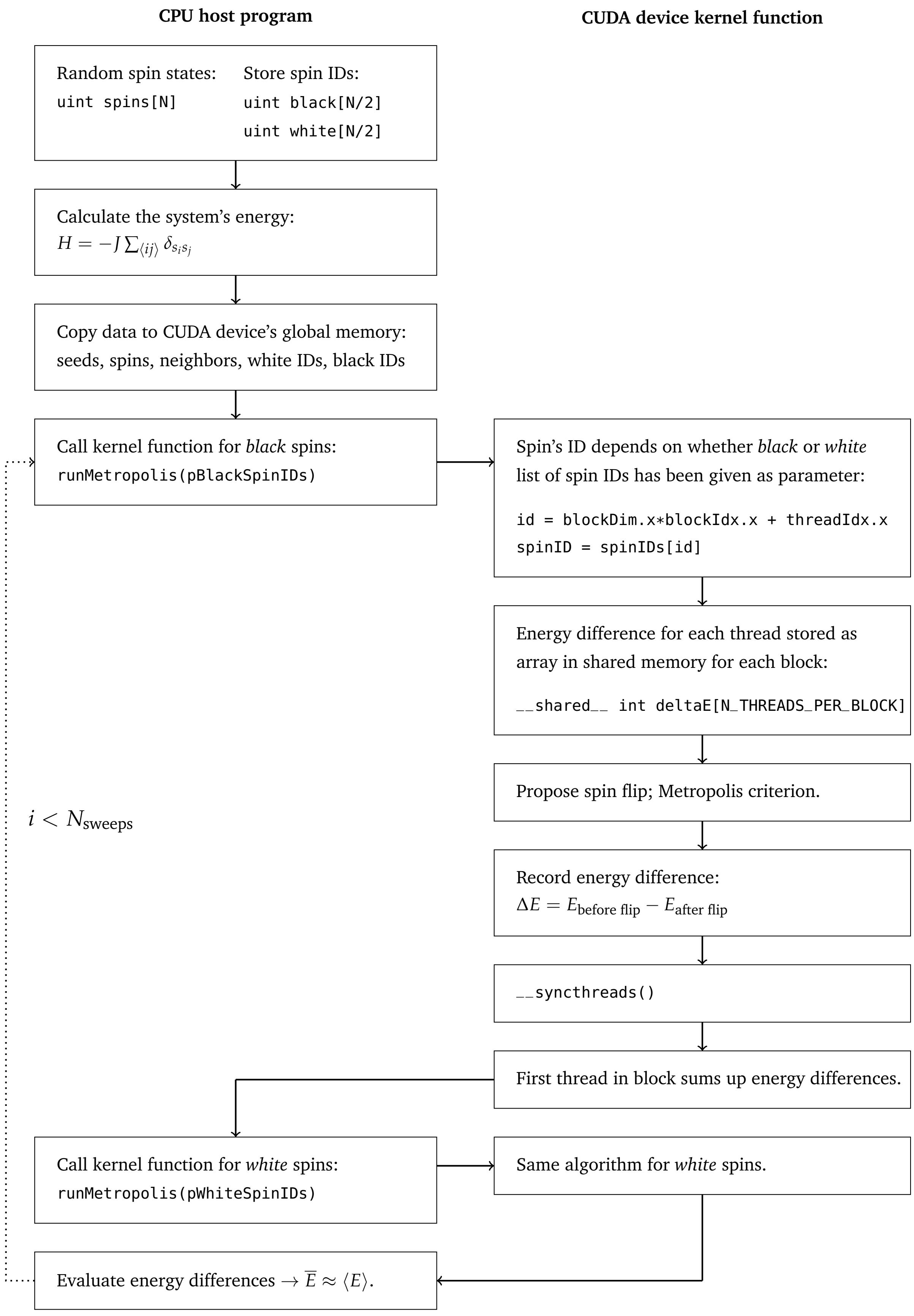
CPU host program
CUDA device kernel function
Random spin states:
uint spins[N]
Store spin IDs:
uint black[N/2]
uint white[N/2]
Calculate the system's energy:
$H = -J \sum_{\langle ij \rangle} \delta_{s_i s_j}$
Copy data to CUDA device's global memory:
seeds, spins, neighbors, white IDs, black IDs
Call kernel function for *black* spins:
runMetropolis(pBlackSpinIDs)
Spin's ID depends on whether *black* or *white* list of spin IDs has been given as parameter:
id = blockDim.x*blockIdx.x + threadIdx.x
spinID = spinIDs[id]
Energy difference for each thread stored as array in shared memory for each block:
__shared__ int deltaE[N_THREADS_PER_BLOCK]
Propose spin flip; Metropolis criterion.
$i < N_{\text{sweeps}}$
Record energy difference:
$\Delta E = E_{\text{before flip}} - E_{\text{after flip}}$
__syncthreads()
First thread in block sums up energy differences.
Call kernel function for *white* spins:
runMetropolis(pWhiteSpinIDs)
Same algorithm for *white* spins.
Evaluate energy differences $\rightarrow \overline{E} \approx \langle E \rangle$.

Figure 4.8.: Simplified flowchart of the CUDA implementation of the Metropolis algorithm for the Potts model. The spin values are stored in a one-dimensional array. A neighbourhood table is used.

5. Conclusions

For small lattices ($N < 256$ spins) the parallel architecture of the CUDA devices cannot be used efficiently. The multiprocessors carry out the Metropolis algorithm with too few active threads and their memory not optimally used. For bigger lattices the simulations ran up to a factor of 20 faster than on the system's CPU.

Further optimizations usually include techniques that take advantage of the underlying hardware. Momchil Ivanov [20] reports Metropolis update times that are up to a factor of 8 faster than in this work. This result is reached not only by the slightly better hardware, but among other techniques by an intelligent reordering of the spin arrays, taking advantage of the GPU's memory structure and read/write mechanisms.

The main focus of this work stayed on code simplicity and easy code portability. The speed comparison between CPU and GPU included the measurement of the energy after each sweep. For real-world scenarios even more observables will need to be evaluated during the simulation, increasing the amount of data that needs to be transferred from the GPU to the host system.

In the future, the speed advantage of GPUs over CPUs will probably increase further with respect to higher clock rates and more processing cores. However, even if CPUs might not be able to gain proportionally higher clock rates, their number of cores also tends to increase and their memory and thread management will yield further speed improvements.

The decision on whether to implement a simulation on a CPU or a GPU will depend on the one hand on technical factors such as how much memory is needed per parallel entity, how much data needs to be transferred between the GPU and the host system and to what degree a model can be parallelized. On the other hand human factors also play a role, such as how much time it will take to test the new implementations, to fine-tune the parameters for optimal speeds, and even how long it will take to learn sophisticated programming techniques like CUDA C, the manufacturer-independent OpenCL, or the Message Passing Interface (MPI) intended for usage with many CPUs.

With deep knowledge of the hardware, simulations can usually be adapted to reach high efficiencies. Often, the drawback is a loss of universality and simplicity of the code. Graphics cards will play an important role when it comes to time-intensive computing tasks, especially in the scientific community where more and more people adopt parallel mechanism for new projects and gain practical experience with this young technology.

A. Code listings

A.1. Conway's Game of Life

```
#include <iostream>

#define ALIVE   1
#define DEAD    0

// world size in x and y direction:
#define WX 16
#define WY 16       // 16*16 = 256 sites
#define N  WX*WY    // number of sites

// Device function: get world array index from world coordinates
__device__ unsigned getId(int x, int y)
{
    // periodic boundary conditions:
    while(x >= WX)
        x -= WX;

    while(x < 0)
        x += WX;

    while(y >= WY)
        y -= WY;

    while(y < 0)
        y += WY;

    return x + y * WX;
}

// Kernel:
__global__ void runConway(unsigned short* world, unsigned iterations)
{
    // get the world coordinate:
    int x  = blockIdx.x * blockDim.x + threadIdx.x;
    int y  = blockIdx.y * blockDim.y + threadIdx.y;

    // id in 1D array which represents the world:
    unsigned id = getId(x, y);

    // create a copy of the world in local shared memory:
    __shared__ unsigned short sites[N];
    sites[id] = world[id];

    // wait for all threads to finish copying:
```

```
    __syncthreads();

    // run the defined number of steps (iterations):
    for(unsigned i=0; i<iterations; i++)
    {
        // determine new state by rules of Conway's Game of Life:
        unsigned short state    = sites[id];
        unsigned short newstate = state;

        // calculate number of alive neihbors:
        unsigned short aliveNeighbors = 0;

        for(short x_offset=-1; x_offset<=1; x_offset++)
        {
            for(short y_offset=-1; y_offset<=1; y_offset++)
            {
                if(x_offset != 0 || y_offset != 0)  // don't count itself
                {
                    unsigned neighborId = getId(x + x_offset, y + y_offset);
                    aliveNeighbors += sites[neighborId];
                }
            }
        }

        // decide about new state:
        if(state == ALIVE)
        {
            if(aliveNeighbors < 2 || aliveNeighbors > 3)
                newstate = DEAD;
            else
                newstate = ALIVE;
        }
        else // if DEAD
        {
            if(aliveNeighbors == 3)
                newstate = ALIVE;
        }

        // wait for all threads to determine new state:
        __syncthreads();

        // save spins in shared memory:
        sites[id] = newstate;

        // wait for all threads to copy new state to shared memory:
        __syncthreads();
    }

    // copy spins back to global memory:
    world[id] = sites[id];
}

// Host function (CPU Code)
int main (int argc, char const* argv[])
{
    // matrix values:
```

```
    unsigned short world[N];

    // number of steps in Conway's Game of Life:
    unsigned iterations = 3;

    // create dead world:
    for(unsigned i=0; i<N; i++)
        world[i] = 0;

    // set up a glider:
    world[1 + 0 * WX] = ALIVE;      // 010000...
    world[2 + 1 * WX] = ALIVE;      // 001000...
    world[0 + 2 * WX] = ALIVE;      // 111000...
    world[1 + 2 * WX] = ALIVE;      // 000000...
    world[2 + 2 * WX] = ALIVE;      // ......

    // allocate memory on CUDA device:
    unsigned short* pDevWorld;      // pointer to the data on the CUDA Device
    cudaMalloc((void**)&pDevWorld, sizeof(world));

    // copy data to CUDA device:
    cudaMemcpy(pDevWorld, &world, sizeof(world), cudaMemcpyHostToDevice);

    // set block and grid dimensions:
    dim3 blockSize(WX, WY);
    dim3 gridSize(1, 1, 1);     // just 1 block for this easy example

    // execute kernel function on GPU:
    runConway<<<gridSize, blockSize>>>(pDevWorld, iterations);

    // copy data back from CUDA Device to 'data' array:
    cudaMemcpy(&world, pDevWorld, sizeof(world), cudaMemcpyDeviceToHost);

    // free memory on the CUDA Device:
    cudaFree(pDevWorld);

    // output results:
    for(unsigned y=0; y<WY; y++)
    {
        for(unsigned x=0; x<WX; x++)
        {
            std::cout<<world[x + WX*y];
        }
        std::cout<<"\n";
    }

    return 0;
}
```

A.2. Metropolis algorithm for the Potts model

```
#include <iostream>

#define L 16    // lattice length; must be even number!
#define D 2     // dimensions
#define N 256   // number of spins on square lattice (N=L^D)
#define Q 2     // number of Potts states

#define N_SWEEPS    100000

/* Distribute N/2 spins over threads and blocks:
   since we do "black" and "white" spins seperately, we only need half the threads:
   N_BLOCKS * N_THREADS_PER_BLOCK = N/2 */
#define N_BLOCKS            4   // number of blocks
#define N_THREADS_PER_BLOCK 32  // number of threads per block


/**********************************
 *  GPU RANDOM NUMBER GENERATION  *
 **********************************/

__device__ unsigned Tausworthe88(unsigned &z1, unsigned &z2, unsigned &z3)
{
    // Three-step generator with period 2^88
    unsigned b = (((z1 << 13) ^ z1) >> 19);
    z1 = (((z1 & 4294967294) << 12) ^ b);

    b = (((z2 << 2) ^ z2) >> 25);
    z2 = (((z2 & 4294967288) << 4)  ^ b);

    b = (((z3 << 3) ^ z3) >> 11);
    z3 = (((z3 & 4294967280) << 17) ^ b);

    return z1 ^ z2 ^ z3;
}

__device__ unsigned LCRNG(unsigned &z)
{
    const unsigned a = 1664525, c = 1013904223;
    return z = a * z + c;
}

__device__ float TausworthеLCRNG(unsigned &z1, unsigned &z2, unsigned &z3, unsigned
     &z)
{
    // combine both generators and normalize 0...2^32 to 0...1
    return (Tausworthe88(z1, z2, z3) ^ LCRNG(z)) * 2.3283064365e-10;
}


/******************************
 *  GPU METROPOLIS ALGORITHM  *
 ******************************/

```

```
__global__ void runMetropolis(int *seeds, unsigned short* spins, unsigned*
    neighbors, unsigned* spinIdList, int* energyDifferences, float beta)
{
    const unsigned id = blockDim.x*blockIdx.x + threadIdx.x;

    // spin id from list of black or white spins:
    const unsigned spinId = spinIdList[id];

    // get seed values:
    unsigned z1 = seeds[4*id    ];  // Tausworthe seeds
    unsigned z2 = seeds[4*id + 1];
    unsigned z3 = seeds[4*id + 2];
    unsigned z  = seeds[4*id + 3];  // LCRNG seed

    // energy differences for this block:
    __shared__ int deltaE[N_THREADS_PER_BLOCK];
    deltaE[threadIdx.x] = 0;

    unsigned short spinstate = spins[spinId];   // get spin state from DRAM
    unsigned short nb[2*D];                     // neighbor states

    // get neighbor states:
    for(unsigned n=0; n<2*D; n++)
        nb[n] = spins[neighbors[2*D*spinId + n]];

    // propose random new spin state:
    unsigned short newstate = floor(TauswortheLCRNG(z1, z2, z3, z) * Q);

    // energy difference: E'-E
    int E_before = 0;
    int E_after  = 0;

    for(unsigned short n=0; n<2*D; n++)
    {
        if(spinstate == nb[n])
            E_before++;

        if(newstate == nb[n])
            E_after++;
    }

    // acceptance probability:
    float dE = __int2float_rn(E_before - E_after);
    float pAccept = __expf(-beta*dE);

    if(TauswortheLCRNG(z1, z2, z3, z) <= pAccept)
    {
        spins[spinId] = newstate;   // flip spin
        deltaE[threadIdx.x] = E_before - E_after;   // note energy difference
    }

    // store new seed values in DRAM:
    seeds[4*id    ] = z1;   // Tausworthe seeds
    seeds[4*id + 1] = z2;
    seeds[4*id + 2] = z3;
    seeds[4*id + 3] = z;    // LCRNG seed

    __syncthreads();

    // sum up this block’s energy delta:
    if(threadIdx.x == 0)
    {
        int blockEnergyDiff = 0;
        for(unsigned i=0; i<blockDim.x; i++)
            blockEnergyDiff += deltaE[i];

        energyDifferences[blockIdx.x] += blockEnergyDiff;
    }
}


/******************************
 *  HOST FUNCTION (CPU PART)  *
 ******************************/

int main(int argc, char const *argv[])
{
    // each spin has value 0,..,Q-1
    unsigned short spins[N];

    // calculate lattice volume elements:
    unsigned volume[D];
    for(unsigned i=0; i<=D; i++)
    {
        if(i == 0)
            volume[i] = 1;
        else
            volume[i] = volume[i-1] * L;
    }


    /* Determine the "checkerboard color" (black or white) for each site and
       initialise lattice with random spin states: */
    unsigned w=0, b=0;
    unsigned white[N/2], black[N/2];    // store ids of white/black sites

    for(unsigned i=0; i<N; i++)
    {
        // Sum of all coordinates even or odd? -> gives checkerboard color
        int csum = 0;
        for(int k=D-1; k>=0; k--)
            csum += ceil((i+1.0)/volume[k]) - 1;

        if((csum%2) == 0)   // white
        {
            white[w] = i;
            w++;
        }
        else                // black
        {
            black[b] = i;
            b++;
        }

        // random spin state:
        spins[i] = floor(drand48() * Q);
    }


    // neighborhood table:
    unsigned neighbors[2*D*N];

    // calculate neighborhood table:
    for(unsigned i=0; i<N; i++)
    {
        unsigned short c=0;

        for(unsigned short dim=0; dim<D; dim++) // dimension loop
        {
            for(short dir=-1; dir<=1; dir+=2)   // two directions in each dimension
            {
                // neighbor's id in spin list:
                int npos = i + dir * volume[dim];

                // periodic boundary conditions:
                int test = (i % volume[dim+1]) + dir*volume[dim];

                if(test < 0)
                    npos += volume[dim+1];
                else if(test >= volume[dim+1])
                    npos -= volume[dim+1];

                neighbors[2*D*i + c] = npos;
                c++;
            }
        }
    }


    // create 4 seed values for each thread:
    unsigned seeds[4*N/2];
    for(unsigned i=0; i<4*N/2; i++)
    {
         seeds[i] = static_cast<unsigned>(4294967295 * drand48());
    }


    // calculate energy (Potts model)
    int E = 0;
    for(unsigned i=0; i<N; i++)
    {
        for(unsigned j=0; j<2*D; j++)
        {
            if(spins[i] == spins[neighbors[2*D*i + j]])
                E--;
        }
    }
    E /= 2; // count each interaction only once


    // copy seeds to GPU:
    int *devPtrSeeds;
    cudaMalloc((void**)&devPtrSeeds, sizeof(seeds));
    cudaMemcpy(devPtrSeeds, &seeds, sizeof(seeds), cudaMemcpyHostToDevice);

    // copy spins to GPU:
    unsigned short *devPtrSpins;
    cudaMalloc((void**)&devPtrSpins, sizeof(spins));
    cudaMemcpy(devPtrSpins, &spins, sizeof(spins), cudaMemcpyHostToDevice);

    // copy neighborhood table to GPU:
    unsigned *devPtrNeighbors;
    cudaMalloc((void**)&devPtrNeighbors, sizeof(neighbors));
    cudaMemcpy(devPtrNeighbors, &neighbors, sizeof(neighbors),
        cudaMemcpyHostToDevice);

    // copy white ids to GPU:
    unsigned *devPtrWhite;
    cudaMalloc((void**)&devPtrWhite, sizeof(white));
    cudaMemcpy(devPtrWhite, &white, sizeof(white), cudaMemcpyHostToDevice);

    // copy black ids to GPU:
    unsigned *devPtrBlack;
    cudaMalloc((void**)&devPtrBlack, sizeof(black));
    cudaMemcpy(devPtrBlack, &black, sizeof(black), cudaMemcpyHostToDevice);

    // each block calculates energy difference to initial state:
    int energyDifferences[N_BLOCKS];
    for(unsigned i=0; i<N_BLOCKS; i++)
        energyDifferences[i] = 0;

    int *devPtrEnergyDifferences;
    cudaMalloc((void**)&devPtrEnergyDifferences, sizeof(energyDifferences));
    cudaMemcpy(devPtrEnergyDifferences, &energyDifferences, sizeof(
        energyDifferences), cudaMemcpyHostToDevice);


    int E_before_simulation = E;
    long long sum_E = 0;
    const float T = 1.25;   // simulation temperature

    for(unsigned i=0; i<N_SWEEPS; i++)
    {
        // White spins:
        runMetropolis<<<N_BLOCKS, N_THREADS_PER_BLOCK>>>(devPtrSeeds, devPtrSpins,
            devPtrNeighbors, devPtrWhite, devPtrEnergyDifferences, 1.0f/T);

        // Black spins:
        runMetropolis<<<N_BLOCKS, N_THREADS_PER_BLOCK>>>(devPtrSeeds, devPtrSpins,
            devPtrNeighbors, devPtrBlack, devPtrEnergyDifferences, 1.0f/T);

        // Sum up energy after a thermalization time for mean energy value:
        if(i > 0.2*N_SWEEPS)
        {
            // get energy changes from the GPU:
            cudaMemcpy(&energyDifferences, devPtrEnergyDifferences, sizeof(
                energyDifferences), cudaMemcpyDeviceToHost);

            E = E_before_simulation;
            for(unsigned t=0; t<N_BLOCKS; t++)  // take energy changes into account
                E += energyDifferences[t];

            sum_E += E;
        }
    }

    cudaFree(devPtrSeeds);
    cudaFree(devPtrSpins);
    cudaFree(devPtrNeighbors);
    cudaFree(devPtrWhite);
    cudaFree(devPtrBlack);
    cudaFree(devPtrEnergyDifferences);

    return 0;
}
```

B. Glossary

Block An array of threads that will run on the CUDA device. The threads are identified by one-, two- or three-dimensional coordinates within the block.

CPU Central Processing Unit, the main processor on a computer's mainboard.

Compute Capability A revision number that guarantees a certain set of features, specifications and instructions for all CUDA devices that carry this number. For details, see Appendix G in the *NVIDIA CUDA C Programming Guide* [1].

CUDA Core scalar processor that forms part of a multiprocessor. Each multiprocessor comes with several CUDA cores for parallel execution.

CUDA Device In this document, a device (usually a graphics card) that supports CUDA instructions.

Device In CUDA terms, the device is always the CUDA Device. It executes the device code.

Global memory The main memory on the graphics card which each multiprocessor and the CPU host have access to.

Grid An array of blocks of threads. The blocks are identified by one- or two-dimensional coordinates within the grid.

GPU Graphics Processing Unit, hardware designed for highly parallel graphics calculations.

Host In CUDA terms, the host is the system that hosts the GPU. Host code runs on the CPU and host memory is the host system's RAM.

Kernel A function that will be executed in many threads on the CUDA device.

Multiprocessor (MP) Processing unit on the graphics card that houses a certain number of scalar CUDA cores, memory, and special purpose processors. Each CUDA device comes with a certain number of MPs.

Shared memory Local memory on each multiprocessor. Access to this memory is restricted to the components of a multiprocessor.

Thread Smallest unit of a parallel program. Usually many threads execute the same instructions simultaneously, i.e. work on the same problem with different data.

Warp size The number of threads a multiprocessor treats at once. Up to the date of this document, the warp size is 32 for all CUDA devices (compute capability 1.0 through 3.0)

Bibliography

[1] *NVIDIA CUDA C Programming Guide, 2011.* `http://developer.nvidia.com/cuda-downloads`.

[2] *NVIDIA CUDA C Getting Started Guide, 2011.* `http://developer.nvidia.com/cuda-downloads`.

[3] *NVIDIA GPU Computing SDK.* `http://developer.nvidia.com/gpu-computing-sdk`.

[4] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, pages 120–123, October 1970.

[5] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[6] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J. Schneider. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics*, 228(12):4468–4477, July 2009.

[7] Martin Weigel. Simulating spin models on GPU. *Computer Physics Communications*, 182(9):1833–1836, September 2011.

[8] Lee Howes and David Thomas. GPU Gems 3, Chapter 37: Efficient Random Number Generation and Application Using CUDA. Technical report, `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch37.html`, 2011.

[9] Pierre L'Ecuyer. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–214, January 1996.

[10] J. Ashkin and E. Teller. Statistics of Two-Dimensional Lattices with Four Components. *Physical Review*, 64(5-6):178–184, September 1943.

[11] R. B. Potts and C. Domb. Some generalized order-disorder transformations. *Mathematical Proceedings of the Cambridge Philosophical Society*, 48(01):106, October 1952.

[12] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087, 1953.

[13] Lars Onsager. Reciprocal Relations in Irreversible Processes. I. *Physical Review*, 37(4):405–426, February 1931.

[14] Paul Beale. Exact Distribution of Energies in the Two-Dimensional Ising Model. *Physical Review Letters*, 76(1):78–81, January 1996.

[15] Ernst Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, February 1925.

[16] Kurt Binder. Finite size scaling analysis of Ising model block distribution functions. *Zeitschrift für Physik B Condensed Matter*, 43(2):119–140, June 1981.

[17] *CUDA Occupancy Calculator, NVIDIA Corporation*. `http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation`.

[18] Lars Onsager. Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition. *Physical Review*, 65(3-4):117–149, February 1944.

[19] A L Talapov and H W J Blöte. The magnetization of the 3D Ising model. *Journal of Physics A: Mathematical and General*, 29(17):5727–5733, September 1996.

[20] Momchil Ivanov. How fast are local Metropolis updates for the Ising model on a graphics card. In Mathias Winkel, editor, *Guest Student Programme 2011*, pages 35–45. Forschungszentrum Jülich, 2011.