Multicanonical Simulations with OpenCL

Jörg Brünecke

May 2, 2018

Contents

List of Figures

Abstract

The Ising model is one of the most fundamental conceptual models when it comes to describing the connection of macroscopic properties caused by microscopic changes in lattices. A basic variation of the Ising model assumes only interaction between nearest neighbouring lattice sites and simple spin states (either up or down). Despite its simpleness the simulation of such a system gets out of hand quickly, already with small lattice sizes. While larger central processing unit (CPU) cluster computers have been used for some time to tackle these kinds of problems, the use of graphical processing units (GPUs) has provided a new tool to significantly reduce computation times in this field of science. In combination with smart concepts like multiplication sampling the analysis of critical behaviour of such kinds of systems has become feasible. One tool to conduct the mentioned simulations is cudamuca—an Ising lattice simulation software running on CUDA-ready GPUs (Gross et al. 2017). This report represents the documentation accompanying the efforts of creating a functional equivalent implementation of cudamuca in OpenCL.

0.1 Introduction

The context of this report's subject is the simulation of spin lattices, their behaviour at critical points and the computation of associated physical quantities. This first sections are intended to provide a quick introduction into the modelling of such systems. For the understanding of later sections it is particularly interesting to gain an understanding how the prototypical two-dimensional Ising model can be simulated. The middle and later parts are concerned with actual computation of 2d Ising lattices on GPUs. Based on the an implementation of multicanonical simulations in CUDA, a port was created in OpenCL and its capabilities compared to the original implementation.

TODO: ein bisschen ausführlicher, Redundanz mit Abstract vermeiden oder mit Abstract mergen und dann Introduction raus

0.2 The Ising Model

Assume a quadratic lattice with edge length $L \in \mathbb{N}$ and $V = L^2$ sites. We assign a value $\sigma_i \in \{-1, 1\}$ to each lattice point. The value of σ_i indicates the direction of a magnetic spin in a simplified model of a two-dimensional ferromagnetic material, corresponding to a parallel ($\sigma_i = 1$) or anti-parallel ($\sigma_i = -1$) alignment with the z-axis (perpendicular to the lattice plane). Summing over all lattice sites we find that the Hamiltonian in presence of an external magnetic field B parallel to the z-axis is given by

$$\mathcal{H} = -B \sum_{i=0}^{V} \sigma_i - \sum_{ij} J_{ij} \sigma_i \sigma_j \tag{1}$$

such that the energy is determined by the spin direction with respect to the magnetic field and by the coupling constant J_{ij} which describes the mutual influence of two spins at the sites i and j, respectively.

At a temperature T we find with $\beta = 1/k_B T$ the lattice magnetisation as

$$M = \sum_{i,j} \frac{\exp(-\beta \mathcal{H}(\sigma_{ij}))}{\sum_{i,j} \exp(-\beta \mathcal{H}(\sigma_{ij}))} \left(\sum_{i=0}^{V} \sigma_i \right). \tag{2}$$

With the partition function

$$\mathcal{Z} = \sum_{\sigma} \exp(-\beta \mathcal{H}(\sigma)) \tag{3}$$

we can provide an expression for the probability of a given spin lattice configuration σ:

$$w(\sigma) = \frac{\exp(-\beta \mathcal{H}(\sigma))}{\mathcal{Z}}, \tag{4}$$

which in turn leads to the average energy of the configuration:

$$E = \sum_{\sigma} w(\sigma)\mathcal{H}(\sigma). \tag{5}$$

Several basic quantities can be derived in a straightforward way. We find the internal energy per site as

$$u = \frac{U}{V}, \tag{6}$$

with

$$U = -\mathrm{d}\ln\mathcal{Z}/\mathrm{d}b = \langle\mathcal{H}\rangle \tag{7}$$

and the specific heat

$$C = \frac{\mathrm{d}u}{\mathrm{d}T} = \beta^2\frac{\langle E^2\rangle - \langle E\rangle^2}{V} = \beta^2 V\left(\langle e^2\rangle - \langle e\rangle^2\right), \tag{8}$$

where $\mathcal{H} \equiv E = eV$.

Two simplifications are often made when it comes to simulating Ising lattices: (i) $B = 0$ and (ii) the coupling constant J_{ij} is considered to be short range in the sense that only direct spin neighbours interact. Both assumptions then simplify the Hamiltonian to

$$\mathcal{H} = -J\sum_{\langle ij\rangle}\sigma_i\sigma_j, \tag{9}$$

where $\langle ij\rangle$ means the that the summation is restricted only to the nearest neighbour spin pairs.

Despite such simplifications the Ising model is an important utility to understand real-world physical phenomena, for instance phase transitions. As simple as the concept is, the state space of already small lattices is overwhelmingly large, making straightforward simulations a hard-to-solve problem.

0.3 Simulation of the Ising model

The straightforward approach to Ising model simulations is provided by a computer program which changes binary values representing spins in accordance to the previously described model. In practice, this requires a data structure for the storage of the spins. The elementary type of such structure has to represent at least two values—one for the up-spin and one for the down-spin. A boolean type would be sufficient for that basic model, however signed integers are more useful because with those translation between the integer representation of a boolean variable ($true \equiv 1$, $false \equiv 0$) and the spin representations ($up \equiv 1$, $down \equiv -1$) is not necessary.

Once the memory-related issues have been coped with, the dynamics of the spin interactions have to be modeled and implemented. In particular, a process for the transition between spin states, with which the system evolves has to be created. It is necessary to consider which intermediate states of the

simulated system resembles meaningful physical states in real-world systems. Trying to answer questions in that domain quickly leads to the insight that naive implementations are not feasible solutions because of the sheer size of possible configurations.

For instance, the state space of a two-dimensional Ising lattice of the size 20 by 20 has 2^{400} possible spin configurations. Solving the partition function via simple iteration over the possible states is virtually impossible —even for such comparably small lattices. A random sampling with data extrapolation is hard to accomplish either since the interesting and meaningful states occupy a narrow region in the vast state space. In order to still be able to practically work with the model a sampling bias in the shape of the Boltzmann weight can be introduced:

$$\mathcal{P}(\{\sigma_i\}) = \frac{e^{-\beta\mathcal{H}(\{\sigma_i\})}}{\mathcal{Z}}, \tag{10}$$

where $\{\sigma_i\}$ represents a sampled spin configuration. Sampling in this manner runs under the term *importance sampling*. Each sample $\{\sigma_i\}'$ depends only on its predecessor $\{\sigma_i\}$, however not directly on the development that led to this state (*Markov Chain*). The probability W for the transition from one state to the next has to satisfy the following conditions

$$w(\{\sigma_i\} \to \{\sigma_i\}') \geq 0, \forall\{\sigma_i\}, \{\sigma_i\}' \tag{11}$$

$$\sum\nolimits_{\{\sigma_i\}'} W(\{\sigma_i\} \to \{\sigma_i\}') = 1, \forall\{\sigma_i\}, \{\sigma_i\}' \tag{12}$$

$$\sum\nolimits_{\{\sigma_i\}'} W(\{\sigma_i\} \to \{\sigma_i\}')\mathcal{P}(\{\sigma_i\}) = \mathcal{P}(\{\sigma_i\}'), \forall\{\sigma_i\}' \tag{13}$$

These generic rules can be satisfied by a variety of update rules, e.g. the heat bath algorithm, the Glauber algorithm or the Metropolis algorithm. The simulations discussed in this report make use of the latter i.e. the update (flip) of a spin is dependent on the energies of the spin configuration before (E_{t-1}) and after (E_t) an update:

$$\omega(\{\sigma_i\}_{t-1} \to \{\sigma_i\}_t) = \begin{cases} 1 & \text{if } E_t < E_{t-1} \\ \exp\left(-\beta(E_t - E_{t-1})\right) & \text{if } E_t \geq E_{t-1} \end{cases}$$

i.e. for a proposed update of one randomly picked spin, the update is always accepted if the energy of the new state is lower than the energy before the update. Consequently, the proposed update is accepted with a certain probability dependent on the energy difference between the two states for updates possibly leading to a higher energy.

In actual implementations this decision is based on comparison between a state's normalised energy measure and a number $r \in [0, 1)$ generated by a uniform random number generator. If $W \leq r$ the update is performed leading to a new state. Otherwise the test is conducted with another randomly picked spin. The procedure has to be repeated necessarily often between measurements

of observables. Otherwise the short-interval measurements would ineffectively refer to almost identical states. A complete sequence of spin-flip-choices is called a sweep. Complete is meant in the sense that on average an update for all degrees of freedom (or spins) was proposed.

It is important to be aware of the fact that we cannot generally assume equivalence of physically meaningful states and snapshots of the simulated system. It is particularly crucial to let the systems settle during a *thermalisation* process. Such an initial phase after the start of the Markov chain with an arbitrary lattice configuration is supposed to result in an equilibrium in which we can estimate the expectation value $\langle \mathcal{O} \rangle$ of an observable $\mathcal{O}$ as the arithmetic mean

$$\langle \mathcal{O} \rangle = \sum\nolimits_{\{\sigma_i\}} \mathcal{O}(\{\sigma_i\}) \mathcal{P}^{\mathrm{eq}}(\{\sigma_i\}) \approx \overline{\mathcal{O}} = \frac{1}{N} \sum_{j=1}^{N} \mathcal{O}_j \tag{14}$$

where $\mathcal{O}_j = \mathcal{O}(\{\sigma_i\}_j)$ is the measurement value of the j^{th} configuration and N is the number of measurements.

Why is this necessary? Imagine an initial state with complete randomized spin orientations. Physically, this compares to a relatively high temperature, at least higher than the critical temperature T_c. Consequently a simulation at temperature $T > T_c$ will approach an equilibrium state in a quick way. Simulations conducted at $T \lesssim T_c$ however, can be considered significantly lower than the corresponding initial state. Virtually forcing the system into the lower temperature is considered a *quench*. Here a thermalisation phase of long enough duration is necessary to reach a states with the characteristic domains of equal spin direction. The typical relaxation time for equilibrium scales as

$$\tau_{relax} \sim L^z, \tag{15}$$

where L is the system length and $z \approx 2$ the critical exponent

0.3.1 From single to multi histogram techniques

Simulating canonical ensembles at a defined temperature T_0 generates physical relevant data for only this temperature. Usually however, the intent of such simulations is to make statements for a wider parameter range. This can be achieved by combining the data of several overlapping energy histograms at different $\beta_0 < \beta_i < \beta_N$ mit $i, N \in \mathbb{N}$ which opens up the possibility to reweight all histograms to a common reference β. With the surplus of generated data and under the condition that neighbouring histograms have a large enough overlap, the correlated data sets can be combined in a common histogram and reweighted to another $\beta \in [\beta_0, \beta_N]$ (Janke 2008).

Let $\Omega(E)$ the number of states at energy E. Then we can write the partition function for a defined $\beta_0 = 1/k_B T_0$ as:

$$Z(\beta_0) = \sum_{\{\sigma_i\}} e^{-\beta_0 H(\{\sigma_i\})} = \sum_{E} \Omega(E) e^{-\beta_0 E} \propto \sum_{E} P_{\beta_0}(E), \tag{16}$$

where $P_{\beta_0}(E)$ represents the unnormalized energy histogram. Consequently,

$$\Omega(E)e^{-\beta_0 E} \propto P_{\beta_0}(E) \tag{17}$$

With a histogram generated in the course of a Monte-Carlo simulation at an inverse temperature β_0, we can calculate $P_{\beta_0}(E)$. The histogram for an arbitrary β can be determined by weighting the histogram at β_0 with the factor $\exp\left[-(\beta-\beta_0)E\right]$, i.e.

$$P_\beta(E) \propto \Omega(E)e^{-\beta E} = \Omega(E)e^{-\beta_0 E}e^{-(\beta-\beta_0)E} \propto P_{\beta_0}(E)e^{-(\beta-\beta_0)E}. \tag{18}$$

The useful β-range is limited by the statistical errors in the numerical calculation of P_{β_0} in the Monte Carlo simulations. Particularly because the outer fringes of the histogram, where the statistical errors are most significant, are also the most important contributors for the calculation of P_β if β and β_0 are distant from one another. A rough criterion for the reweighability of a certain histogram is that the peak location of the resulting histogram does not exceed the point where the input histogram's energy has decreased to a value in the order of 30 to 50 percent of its peak.

In order to overcome this limiting factor it is advisable to combine the information from several, say m, Monte Carlo simulations at $\beta_i, i = 1 \ldots m$ with N_i measurements. Then, all resulting histograms are reweighted to a common reference β_0. Combine all such reweighted histograms into one by computing error weighted averages. In this way we get a superimposed version of all single simulation runs which can be reweighted to virtually any β (within the already described limits).

0.3.2 The multicanonical approach

Multicanonical sampling addresses the problems of classical multi-histogram sampling by the use of weight vectors which artificially equalize the sampling probability of virtually all possible transition states. This is equivalent to the procedure described in the previous sections. The new part is that the weight vectors are adjusted in an iterative manner. This process is shaped in such a way that the configuration of the next sampling iteration becomes more suitable to sample similar numbers of data points in each area of the histogram, instead of sampling according to the physical energy state distribution. To this end the canonical Boltzmann distribution

$$P_{\text{can}}(\{\sigma\}) \propto e^{-\beta\mathcal{H}(\{\sigma\})} \tag{19}$$

is replaced by the multicanonical distribution

$$P_{\text{muca}}(\{\sigma\}) \propto W(Q(\{\sigma\}))e^{-\beta\mathcal{H}(\{\sigma\})} \equiv e^{-\beta\mathcal{H}_{\text{muca}}(\{\sigma\})} \tag{20}$$

with the multicanonical weight factor $W(Q)$. Q is to be understood as a macroscopic observable, for instance the energy of magnetization of the observed system. From equation 20 we infer the multicanonical hamiltonian

$$\mathcal{H}_{\text{muca}} = \mathcal{H} - \frac{\ln W(Q)}{\beta} \tag{21}$$

While this seems counter intuitive at first glance, the advantages of this method become quickly clear —making statements about physical behaviour with small numbers of data points is always inconvenient. Creating larger amounts of data also for the rare states gives much better statistics for analysis. This is in particular useful as soon as a system is expected to change from one probable state to another via a lower-probability region in state space.

0.4 Running Computations on GPUs

GPUs are the modern tool of choice for computation of highly parallelizable tasks. While these devices have been used for quiet some time as support for graphics intensive applications in 3d visualisation or computer gaming, they are heavily used for scientific computation nowadays. Already ten years ago Owens et al. (2008) named the *Foldig@home* project in biophysics as well as molecular dynamics or electrostatic field simulations as successful examples. The development of GPU hardware and its use in science has not declined since then. On the contrary, the use of GPUs can be observed in almost every domain of physics.[1]

Since at least the development and adoption of computational purpose programming languages for GPUs like OpenCL and CUDA and of abstraction layers in Mathematica, Matlab or Python are GPUs established tools in science. Meanwhile dedicated computation devices can be purchased which not even possess graphics output interfaces anymore.

GPUs were originally optimised to solve computer graphics problems. These kinds of problems did in many cases not require the high accuracy as the more general purpose CPUs. For this reason GPUs used to know about data types with rather unconventional bit widths as for instance 9 or 12 bits. Wider data types as 32 bit integers or floats are rather new innovations for such devices. 64 bit data types or implementations of mathematical functions for such data types are still missing in some hardware (for example the low end Intel GPUs). However, high end devices and devices developed for scientific use implement these functions in hardware and programming application programming interfaces (APIs) to make these necessary types easily accessible.

GPUs are usually connected to the host computer via standard interfaces as PCIe. The data transfer rates through these connectors are quiet small compared to the computational throughput of the GPUs. For that reason it is worth thinking about parallelizability and whether it makes sense and in which way data is transferred to and from the video memory on the graphics card.

[1] https://arxiv.org/find/all/1/ti:+GPU/0/1/0/all/0/1

0.4.1 GPU architecture and workflow

The big advantage of GPUs over CPUs is the abundance of specialised purpose processing units. In particular, GPUs feature hardware for the storage and manipulation of vectors and matrices. Many operations on such objects can be broken down into quick-to-solve tiny problems, the results of which can be combined to the result of a superior problem. Exactly this is the strength of GPUs.

Figure 1 illustrates the concept on a basic level. Input data is copied from the computer's main memory to the GPU's memory (1). The host computer instructs the GPU, how to conduct computations (2). The GPU uses the data in its memory to simultaneously compute many results (3). These are then copied back to the host memory for further examination (4).

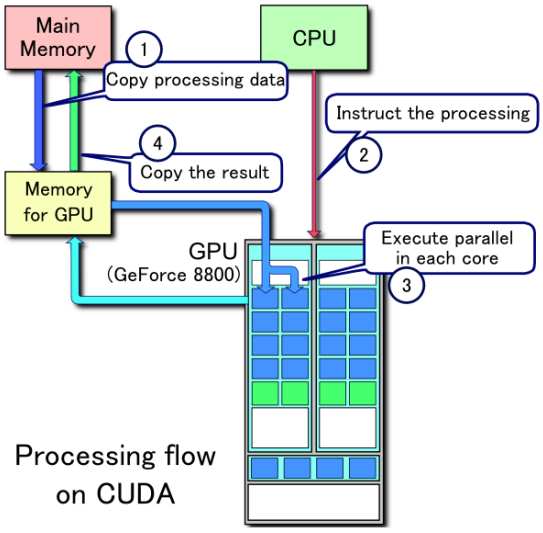

Figure 1: CUDA processing flow as an example for GPU processing, numbers indicating order of application flow, source: wikimedia commons / Tosaka

The computational problems are solved by several hundreds to thousands GPU cores (or compute units), which feature problem specific sub units or processing elements as arithmetic and logic units (ALUs), for instance. A problem that is supposed to be solved on a GPU is optimally divided into a number of independent sub problems and implemented in so called *kernels*—functions which are applied to a certain set of input data. Kernel instances optimally run in parallel on the compute units of a GPU. Their most important property is the independence of execution order—so-called *kernels* ingest input values and output a computation result which later might be integrated into a more general solution. Usually this is accomplished by not only modeling problem sub units but also providing sub-chunks of the overall data set in parallel which the kernels can be applied to. In order to provide the input data in an efficient

manner understanding of the data flow from and to and also within a GPU is crucial.

The size of random access memory (RAM) on upper end GPUs matches in orders of magnitude (i.e. several GB) with what is known from conventional workstation RAM only recently. The available memory is provided in several levels of hierarchy. There is a large area of shared memory, accessible by all compute units at any time. Compute units have their own local memory areas and so have single kernel instances, too. Access times to the global address space are generally the slowest, accesses to memory close to the computations are significantly faster. While sharing of memory is to be avoided if possible, it is sometimes necessary to exchange information between parallel processes at one time or another. Using globally accessible memory may be a way to accomplish this. However, memory being used by all kernels simultaneously raises the issue of race conditions when two or more compute units have potential write access to common memory areas. To prevent out-of-sync computations and race conditions it is necessary to write-access memory in an atomic manner. Atomicity in turn goes hand in hand with longer run times. One design pattern to work around such problems is to provide the compute units with dedicated caches with sizes of several tens of kilobytes on recent GPUs. Since access is constrained to one particular computational unit, this allows to operate on memory without regard of operations in other compute units. Although the size of the caches does not seem very large, the significantly faster access can lead to much faster program execution. For instance, repeated incrementation of an integer in the local cache of many compute units and adding up all integers later is much faster than atomic incrementation of a single integer by every compute unit in global memory. Finer grain can be accomplished by using the memory unique to single kernel instances which run parallel on a compute unit.

With the basic knowledge we can map a path to a program running on a GPU. As always it is advisable to formulate and analyse the nature of the question. This helps in breaking down the problem in independent computational tasks which can be run in parallel. The concept of parallel computing requires to think about the available and generated data and in which way data access can be partitioned, too. Some parts of the data can be local to kernels or compute units, some have to be globally accessible. If atomicity is kept in mind programming with most of the data globally accessible might be less complex in some cases. However, in the spirit of performance optimisation accesses to global memory should be minimized. Caching mechanisms internal to the hardware architecture might not be obvious, but performance critical. Thus, it is advisable to think about the data layout in memory to make use of caching and prevent cache misses.

0.4.2 CUDA

CUDA is the proprietary programming interface for NVIDIA GPUs. It allows to interact with the hardware by allocation of video memory, the transfer of data from and to the GPU and its processing. As a disadvantage of CUDA can be

seen that it can be run only on hardware with NVIDIA chips. The upside of the same fact might be that CUDA and GPU drivers can be optimized to best performance, which might not be the case for more generic approaches. There are CUDA bindings for a variety of programming languages with different distance to the hardware level, for instance C/C++, Fortran or Python. In addition to solid-programming-language bindings, there are wrappers for certain scripting languages and computer algebra systems (CAS's) which provide an easy access to parallel computations. However, with regard to the topic described in this report I will keep to describing the CUDA in the context of the *native* C/C++ interface.

As described earlier it is characteristic for GPU programming to break down problems in many sub units which can be approached independently. The smallest such unit in CUDA terminology is the *thread*, which computes a result from some input data. The actual algorithm that is executed in a thread has to be implemented in a *kernel*. A thread, together with 31 other threads comprises a *warp*—a unit sparsely used in literature. More common is the subsummation of threads in *blocks*, the number of which is problem dependent and to be determined by the developer within certain limits. Several blocks are formally collected in *grids*. The CUDA runtime's task is to distribute the threads over the available hardware resources.

The basic concept of the GPU memory hierarchy as described in section 0.4.1 is of course existent in CUDA. Each of the threads can make use of private local memory. Threads comprising a thread block share memory during the lifetime of the block. Global memory can be accessed simultaneously by all threads. Two more types of memory are the *constant memory* and the *texture memory*. While the former increases performance by splitting serial read-only memory accesses into parallell requests, the latter caches data for reads caused by several (many at best) thread requests. Memory operations generally are faster with lower hierarchy level and in read-only mode.

Let us have a look at simple CUDA program for illustration. I took an introductory example out of the NVIDIA developer resources[2] for the purpose of a short primer. Let us first start with classical inclusion of libraries:

```
#include <iostream>
#include <math.h>
```

We then define the part that is supposed to do computations on the GPU hardware. The keyword __global__ indicates that the defined function is a CUDA kernel:

```
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
  y[i] = x[i] + y[i];
```

[2] https://devblogs.nvidia.com/even-easier-introduction-cuda/

```
}
```

Next, we define the classical C++ main() function—the programs actual entry point. In this function GPU memory is allocated with cudaMallocManaged () which gets an address and the amount of bytes to be stored as parameters. In the following codefor loop the memory is initialized with ones and twos as floating point numbers.

```
int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory - accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
```

After the preparation is finished the previously defined kernel function add() is run on the NVIDIA device. Afterwards we wait for the GPU to finish computations with cudaDeviceSynchronize(), then evaluate the results of the computations and do memory janitoring before gracefully exiting the program.

```
  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
  maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

A similar structure can be expected from all moderately simple CUDA programs. There is a definition of one or several kernels, i.e. functions supposed to run on graphics hardware, there is memory allocation on the GPU and the host and there is copying of memory from and to the GPU.

0.4.3 OpenCL

OpenCL offers an interface for distributed computing on different kinds of hardware, among which are GPUs. In this way OpenCL combines the abilities of computation frameworks as MPI[3], which allows to distribute computational tasks over several dedicated devices, with the GPU programming abstraction. In that sense, OpenCL can be seen as a more generic approach than CUDA, which however will not be of further interest in this context. That OpenCL is still similar to CUDA in regard to GPU programming becomes clear when we try to relate their terminologies. Technically the better part of the CUDA concepts can be found in OpenCL, too. CUDA's *thread* is OpenCL's *work item*, a *thread block* is equivalent to a *work group*, which consist of several *work items*. A *grid* is an *NDRange*. The same mirroring can be found with memory types: CUDA's *shared memory* corresponds to OpenCL's *local memory*, the *constant memory* has the same name in both cases and *texture memory* is equivalent to the *image*. (Du et al. 2012)

An OpenCL program usually consist of two parts. One part runs on the host and is responsible for the organising tasks, as resource dispatching, GPU memory allocation and data transfer between different types of memory. The other part is implemented in the so-called OpenCL *kernel*, which in the best case contains the complete heavy-load-logic and is executed by the computation device (the GPU in our case).

The existence of OpenCL bindings and wrappers for a number of programming languages (e.g. C++, Fortran, .NET, Erlang oder Python) makes it possible to write the glue code in a language of choice. The performance critical part however (*kernel*) conventionally had to be written in a C flavour (only recently a kind of C++ has become usable, too). These C/C++ subsets feature some problem specific or hardware specific extensions, but have significant constraints when compared with classical C/C++ (for instance no recursion, no function pointers, no dynamic arrays).

Let us examine the simple adding program from the previous section in OpenCL syntax.

```
TODO: equivalent opencl code
```

Upon closer examination of the two implementations of the example add program it becomes obvious that CUDA and OpenCL do not only have differences in terms of style. The most important probably being the selection of a platform and devices within the platform. This reflects the more generic concept of OpenCL which could employ other computing devices than GPUs. Another one is that OpenCL requires the setup of an environment on the host computer in preparation for the kernel execution on the GPU. In particular the dedicated compilation of the kernel is eye-catching. This is different in CUDA which allows a seamless integration of the GPU related programming with the rest of the workflow. Something similar is true for memory management. While

[3] http://mpi-forum.org/

we can handle video memory in CUDA almost just like host RAM, this is different in OpenCL. There, we have to create buffers and much more explicitly copy data back and forth between the video and the host RAM in order to start computations and evaluate results on the CPU.

0.5 Cudamuca

Cudamuca is available in two flavours[4]. There is a reference implementation in C++ running on multi-core CPU machines. And there is an implementation in CUDA intended to be run on GPUs. Both implementations are supposed to generate identical results if the random number generators (RNGs) and algorithms are fed with identical seeds and parameters. Since the main focus of this report is GPU related I will not give attention to the conservative CPU approach. Instead, I will describe the CUDA implementation first in a moderately abstracted level in order to show the basic ideas behind the implementation. I will later cover the differences of the OpenCL implementation.

0.5.1 The original (CUDA-) implementation

The conceptual ideas cudamuca is based on are provided by Gross et al. (2017). Their article combined with the corresponding source code provides the basis of this section. In the following I will not provide source code listings. However it might make sense to have the sources at hand because this report refers to certain functions or parts of the code. I will mainly follow the process of `main()` and cover the tasks accomplished by subroutines where it seems appropriate.

Includes

Cudamuca includes and makes use of the 3rd-party library random123[5] for generating pseudo random numbers. The employed RNG algorithm is stateless and counter based and in this way avoids the necessity of data transfer between the involved compute units (Salmon et al. 2011). The two includes of `muca.hpp` and `ising2d_io.hpp` make helper functions for data analysis, benchmarking and command line parsing available. Apart from that the included libraries are basic system libraries for data I/O, mathematical functions and time stamping.

Memory

A distinction between two fundamentally different types of memory has to be made, which is reflected by a certain variable naming scheme: memory in the host computer's RAM is referenced by variables with name prefixes `h_`, while memory on the GPU is referenced by variables with name prefixes `d_`. Apart from the variables storing trivial values as lattice size, loop counters or computation

[4] https://github.com/CQT-Leipzig/cudamuca
[5] https://www.thesalmons.org/john/random123/releases/1.09/docs/

constants, there are several crucial data structures representing the spin lattice, the energy histograms and the reweighing vectors computed by the program. For these structures memory is reserved in the host computers RAM, which might be initialized and afterwards copied to video RAM represented. After computations on the GPU have manipulated the data, memory might be copied back to the host's RAM for further analysis or manipulation.

The spin lattice is mapped to a continuous memory area of 8 bit wide signed integers. The number of elements this memory holds is calculated as the product of the WORKER (CUDA *threads*) count and the number of lattice sites. As a result the allocated memory represents as many lattices as there are WORKERs. All lattice sites are randomly initialized with values out of {-1, 1} using random numbers generated by the RNG library. All of the crucial operations on the lattice are conducted in the graphics device's memory. The lattice is only copied to the GPU and never back to the host.

The overall energies of individual lattices are computed for each worker. Memory globally accessible by all threads of the size WORKERS $*$ **sizeof**(**int**) is allocated on the device and initialized by code running on the GPU for that purpose.

For the iterative approach of multicanonical simulations, weights are necessary for rescaling the probability of parts of the energy landscape. To this end, cudamuca sets up a global weight array as well as a histogram array of length $N = L^2$, both of which are initialised with zero float and integer values, respectively.

Based on benchmarking Gross et al. (2017) decided to store the histogram not in local or private memory of the block or work items, respectively. These memory types seem to be predestined to be chosen for their generally higher speed, compared to global memory. After experimenting with different variations of memory use the decision was made in favour of the global histograms and weights for performance reasons. A disadvantage accompanying this choice is that mutating parallel operations on this global memory have to be atomic to prevent faulty results. Otherwise flawed computations must be expected, for instance when several threads increase the value of the energy bins simultaneously. Parallel writes which are to be considered by design may then lead to the situation where one write destroys the effects of another write. Take for instance, the incrementation of the energy bins in the energy histograms. If more than one thread increases the value of a bin at the same time only one of the incrementations survives and manifests in the histogram.

The naive approach to storing data in global memory needed by all threads would be to put the spin configurations in linear order next to each other. As Gross et al. (2017) point out, however, it is much more efficient to put "all the first spins next to each other, then all the second spins" and so on. As a result, access of parallel running threads to the same lattice sites require only one physical access to global memory. Successive accesses are accelerated by caching mechanisms. Since the selection of the spin flip order is (pseudo-) random, it is necessary that all threads use the same order of spins, in order for the described coalescing memory access to work. Operations on global memory

always cause a transfer of at least one cache line (128 bytes).[6]

Another memory related optimization was achieved by using texture memory [7] for the weights stored in GPU memory. Texture memory is classically used in actual graphics software for rendering surfaces. It includes caching methods which, in some cases, can generate significant performance benefits. CUDA contains an API to make use of texture memory in general purpose computations. In cudamuca this is accomplished by creating a texture memory object (instead of just using global memory) for the weights vector and using the corresponding texture access functions. Because of the caching, threads reading the weights in parallel get the requested data faster than they would with conservative reads from global memory.

Logic

After the data structures are set up on the host and copied to the CUDA device, the simulation is started (function call `mucaIteration(...)`). In the spirit of multicanonical simulations this happens in several iterations, each of which begins with a thermalisation phase of the lattices in order to start measurements with a *realistic* energy distribution. Once the thermalization is finnished the Markow-chain-driven spin flipping is conducted for a predefined number of updates. Each of such updates results in an overall lattice energy which in turn corresponds to an energy bin in the histogram. Accordingly, the appropriate bins in each worker's individual histogram are incremented. After all updates were conducted, the resulting energy of the overall histogram is stored for each worker.

The computation of the energy of an individual lattice is done by iterating over the lattice sites and computing

-Value at i * (Value at RIGHT + Value at LEFT + Value at ABOVE + Value at BELOW),

where RIGHT, LEFT, ABOVE and BELOW represent the spins at the respective sites relative to the spin at i. These results are summed up to the overall energy. Where necessary, the lattice is extended by wrapping around the edges in order to fulfill continuous boundary conditions.

After one such iteration the histograms are copied back to the host RAM and undergo analysis which reveals a certain probability distribution of energy states used to compute the weight vectors for the next iteration steps. Then the process starts again with a new set of weight vectors unless the histogram has approached a flatness determined by the Kullback–Leibler –divergence[8], in which case the program terminates.

In summary, first many lattices are initialized and thermalized, then the simulation is run. Each simulation step is conducted by every worker. The

[6] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0

[7] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-memory

[8] https://en.wikipedia.org/wiki/Kullback-Leibler_divergence

number of particular occurring energy states of the lattices determines the energy probability distribution used for recomputing the weight vectors for the next simulation step.

The simulation produces statistics about the computation performance and more importantly the data describing the energy state development of the Ising lattice. Using this data the reweighing for example to the Boltzman energy distribution can be done as described in section **??**.

Production run

After weight vectors are computed in the described fashion a *production* run can be started, where the system undergoes thermalisation as described earlier. In this mode iterations are not used to reconfigure the weight vectors, but the weight vectors of the initial training phase are assumed to correspond to the intended sampling bias. Then cudamuca employs jack knifing, dividing the number of spin updates in 100 sub sets. From the resulting 100 histograms state density can be deduced and error analysis conducted.

Hardware parameters

It is performance critical to correctly determine the number of GPU workers. In particular thread block size (variable WORKERS_PER_BLOCK) is an important choice as it heavily impacts the number of concurrent computations (*latency hiding*). Additionally, the hardware can transparently put idling threads to work while others, waiting for I/O, would decrease computational efficiency. Picking the right number being crucial, Gross et al. (2017) chose to follow recommendations from the CUDA developer resources.

Based on the architecture communicated via the macro __CUDA_ARCH__[9] cudamuca determines the minimum number of blocks resident on the GPU (variable MY_KERNEL_MIN_BLOCKS). MY_KERNEL_MIN_BLOCKS and WORKERS_PER_BLOCK are provided for performance optimisation via the __launch_bounds__ qualifier.[10] Both kernel functions in cudamuca, computeEnergies(...) and mucaIteration (...) are defined in this way.

Random number generation

The required random numbers are generated with the framework Random 123, an implementation of Salmon et al. (2011). Random numbers are need at two code parts in the algorithm: (i) for finding the next spin to flip and (ii) for the decision whether to flip a spin or not, i.e. the importance sampling. For both purposes the underlying C function u01fixedpt_open_open_32_24() is used, which returns uniformly distributed numbers with the type float with 24 bit mantissa, in the open interval $(0, 1)$. Such a number is well suited to be compared to a probability distribution, i.e. for the importance sampling. The use of a floating

[9] https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#cuda-arch
[10] https://docs.nvidia.com/cuda/cuda-c-programming-guide/#launch-bounds

point number for picking the integer index of the spin to flip is less obvious. To that account the floating point number is multiplied by the overall spin count and the result converted to an integer. In this way, spins are picked uniformly from the whole lattice.

0.5.2 OpenCL implementation

The main task underlying this report is the implementation of cudamuca to OpenCL. As described earlier in section 0.4.3 most CUDA concepts can be translated one-to-one on OpenCL. The OpenCL implementation of the original CUDA code had to be written in OpenCL C—a language much more similar to C than to C++. As a result the crucial simulation code is written in C, rather than C++. For instance, the C API functions of the RNG were used in those parts of the code, instead of the C++ classes. Also, the use of standard C++ classes were not possible in the OpenCL GPU kernel part. Nonetheless, the ported code is intended to match the original implementation in so many aspects as possible. This can be assumed to be achieved since basically all CUDA functions could be replaced with OpenCL equivalents. The following sections describe how the port was approached and at which points difficulties were faced and which parts have a very idiosynchratic OpenCL character.

Includes

The included libraries for the host part of the port are basically identical to the CUDA implementation. Some of the includes are removed because of redundant inclusion in header files. The only real difference lies in the inclusion of the OpenCL headers via **#include** <CL/cl.hpp>.

OpenCL provides several extensions for the support of data types and operations on these types. The OpenCL kernel code requires the inclusion of two such extension. First, there is the requirement of double precision floating point numbers. Therefore the extension cl_khr_fp64[11] is enabled. Since the energy bin counts in cudamuca easily exceed unsigned 32 bit integer limits, operations on 64 bit integers are inevitable. These have to be atomic where several threads have write access to the same counters. Atomic operations on large integers is provided by the extension cl_khr_int64_base_atomics.[12]

Janitoring

A main difference between CUDA and OpenCL is the handling of device specific code. While in CUDA this is done in an almost subtle way where device functions are just defined with certain keywords and are then available to the program, the intertwining of C/C++ and OpenCL code is less close. In OpenCL a *kernel* is written, compiled during runtime and loaded to the device

[11] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/cl_khr_fp64.html
[12] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/cl_khr_int64_base_atomics.html

for execution. Let me be more specific and pick the crucial parts from the cudamuca OpenCL port. First, a choice has to be made in regard to the used hardware. This is accomplished by finding the available computing platforms with cl::Platform::get (...) [13] and then picking one among the found (for instance NVIDIA, AMD, Intel, Mesa). The platform corresponds to the vendor-specific OpenCL implementation. Within a platform exist valid choices for devices, which can be made by a call to cl::Platform::getDevices (...) ;. Examples for devices would be GPUs in the case of the NVIDIA platform and GPUs or CPUs in case of Intel or AMD. Software for listing available platform and devices are abundant on Github.[14] Once, a platform has been chosen the actual computing devices are framed within in a cl::Context[15] object. The use of this is implementation specific and not important in this case since we only use the GPU for computing, anyways. The context is used as a parameter to a cl::CommandQueue[16] object, which handles the dispatching of function calls to the compute kernel. For the existence of a valid kernel, an instance of the class cl::Kernel[17] has to be created and provided with the source code of the program supposed to run on the GPU. Next, the parameters that will be expected by the kernel code are set via cl::Kernel.setarg(). Only then can the GPU-code be run via cl::enqueueNDRangeKernel()[18] which puts a task into the command queue of the context.

We saw that the execution limits in CUDA were set in a quiet straightforward way (section 0.5.1). This also is accomplished in a more explicit way in OpenCL by calling cl::CommandQueue::enqueueNDRangeKernel(...) with the overall number of work items and the number of work items in the work group, i. e. we tell the device how much work there is to do and ho many tasks we expect to be dealt with in parallel. OpenCL deduces hardware parameters from the workload. This differs from CUDA where we set the hardware parameters and the tasks are distributed accordingly.

It is hard to argue against the statement that this whole procedure seems quiet cumbersome. However the complexity might be justified by the much more generic OpenCL approach in terms of compute device variability and vendor agnostics.

Memory

In CUDA, memory on the device is allocated using the function cudaMalloc() which returns a pointer to memory on the GPU. There is a very similar function in OpenCL made available through cl::Buffer(), an object which upon construction allocates and then represents memory on the GPU of the given size and read/write access privileges. In the OpenCL port of cudamuca the memory

[13] https://github.khronos.org/OpenCL-CLHPP/classcl_1_1_platform.html
[14] e.g. https://gist.github.com/courtneyfaulkner/7919509
[15] https://github.khronos.org/OpenCL-CLHPP/classcl_1_1_context.html
[16] https://github.khronos.org/OpenCL-CLHPP/classcl_1_1_command_queue.html
[17] https://github.khronos.org/OpenCL-CLHPP/classcl_1_1_kernel.html
[18] https://github.khronos.org/OpenCL-CLHPP/classcl_1_1_command_queue.html

for the lattice, the lattice energies and the histograms is allocated using this interface. The code structure is—apart from the different syntax identical to the original implementation. As in the CUDA case, memory representing the lattice is initialized in host RAM and then copied to the GPU using the function cl::enqueueWriteBuffer(). Also, energies and histograms are generated and copied to the GPU in the very same way as in the CUDA implementation, just with OpenCL functions. Unfortunately, there is no way to profit from the texture memory caching as is provided in CUDA. There is the supposedly correspondig OpenCl image2d, which I did not use for the implementation. (TODO: das kann ich doch noch machen. Siehe hier und hier)

Several C++ style type casts can be found in the original cudamuca implementation. Unfortunately, in OpenCL C we cannot cast types in the C++ way. There is, however, a set of functions convert_destType(sourceVar), where destType and sourceVar had to be plugged in.[19]

It is necessary to take care of the right choice of variable types. OpenCL provides a list of type specifiers for the appropriate type, signedness and width[20]. The recommended way to work with these is to use the OpenCL types like char, ulong, code and so on in kernel code. If a kernel variable is referred to in the application code the type name is to be prefixed with cl_ in order to prevent confusion programmer wise as well as compiler wise. A good example for the usage of this would be the allocation of memory on the GPU from the host. The amount of required memory could then be determined by a call to **sizeof**(cl_ulong). Consequently most data types were renamed in the port according to the list in table 1.

cudamuca original type	Opencl type in application	OpenCL type in kernel
unsigned	cl_ulong	ulong
float	cl_float	**float**
int8_t	cl_char	**char**
int	cl_int	**int**

Table 1: Equivalent choices of types in CUDA (original implementation), OpenCL host application and OpenCL kernel of the port; first, second, third column, respectively

Random123 in OpenCL

While CUDA allows to make use of C++ syntax this is not the case for OpenCL. As a consequence, the C++ wrappers of the Random123 library could not be used. However, there are C equivalents in Random123 for all underlying functions used in cudamuca which can be expected to generate the same results in the OpenCL kernel. In order to make sure C++ and C functions would

[19] https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/convert_T.html
[20] https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/scalarDataTypes.html

create the same lists of random numbers two simple programs were written for testing purposes.[21] In a later step, the functions were also included in CUDA and OpenCL programmes and were run on the GPU with the same seeds to generate lists of several 10 thousand random numbers. Afterwards the lists were compared and found identical.

Data processing

The most important part of the port is the one contained in the kernel, i.e. functions prefixed with `__global__` or `__device__` in the original CUDA implementation. The former are called from the host and can be seen as entry points to GPU operations while the latter are functions which are called only from code running on the GPU. An equivalent syntax is provided by OpenCL via the differentiation between function definitions prefixed by `__kernel` (called from the host when starting a kernel) and plain C function definitions which can be called only from within a kernel. There are two kernel functions provided by the declarations `__kernel` **void** computeEnergies(...) and `__kernel` **void** mucaIteration(...), where ... represents function parameters. Functions which can be called from within kernels only are EBIN(...), localE (...) , calculateEnergy (...) and mucaUpdate(...).

A basic difference between CUDA and OpenCL must be made when it comes to handling function parameters that are submitted from the host but used in function calls within a kernel. The CUDA implementation of cudamuca accomplishes this in two ways, (i) via parameters in the kernel function calls and (ii) via globally accessible memory on the GPU. The former is the normal way in OpenCL to provide the functions running on the GPU with parameters, however the latter is not within OpenCL specs. This leads to the necessity to pass every constant down through all functions. For example, the function EBIN (...) needs the parameter of d_N, which has to be passed down the call tree in the OpenCL implementation while it is a device wide global constant in CUDA. This is why most functions in the OpenCL port expect more parameters than in the CUDA implementation. Fortunately, because the parameters are passed and named accordingly, the function internals could be left as in the CUDA implementation almost entirely.

The original cudamuca implementation makes use of the CUDA math function expf().[22] The corresponding function in OpenCL is exp().[23] During initial tests it became clear, that both functions do not generate identical output from all input. While the random number generator creates identical lists of numbers the value of the expf() in CUDA and exp() in OpenCL often differs in the last digit. This leads to slightly diverging simulations, i.e. the qualitative behaviour of the OpenCL implementation is equivalent to the CUDA implementation. The differences become visible however, when energy histograms are plotted a while into the simulation (see section 0.6.2).

[21] https://git.bloerg.de/studium/test-rng
[22] https://docs.nvidia.com/cuda/cuda-c-programming-guide/#intrinsic-functions
[23] https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/exp.html

0.6 Results

0.6.1 Port

The original implementation of cudamuca could be ported from CUDA to OpenCL. The sources can be found at https://git.bloerg.de/studium/clmuca.

0.6.2 exp() ≠ expf()

Initial simulation runs with the ported cudamuca implementation revealed slightly different numeric results despite the qualitative behaviour of the simulation seemed appropriate. An error due to differing generation of random numbers could be ruled out. After some research the cause was found in the differing accuracy of the single precision expf() function used in the CUDA implementation and the single precision exp() function used in OpenCL. While the former is the cmath implementation[24], the latter is a dedicated implementation for OpenCL[25]. Table 2 contains records from a list of 100 000 random numbers and the corresponding results of the cmath expf() and the OpenCL exp(), respectively. The upper half of the table lists rows with differing results for the function calls, whereas the lower half lists lines where the results are identical. Where the numbers differ, a clear tendency towards the smaller number can be observed in OpenCL. This allows to suspect that the cause lies within different ways of rounding. A thorough analysis of the CUDA and the OpenCL implementation of the math functions should reveal whether this is true, was however not conducted in this context.

Code for generating exemplary lists can be found in the exp-issue repo.[26]

0.6.3 Impact of the exp-issue

The issue raised in the previous section does not seem to influence the simulations in a qualitative way. Though, effects are clearly visible in the energy histograms as can be expected because the comparison of the exp()/expf() functions to a random number controls whether an energy histogram update is conducted or not. Of course, the outcome of this conditional can be one way or the other, depending on a single digit. In figure 2 can be seen, that in the first few training iterations the blue (OpenCL) and green (CUDA) graphs have exactly the same coordinates. With increasing count of generated random numbers the probability to stumble over a random number the exponential function results of which differ, increases. In the illustrating example, starting with iteration four, the graphs diverge slightly but remain very similar for the whole simulation run.

[24] http://en.cppreference.com/w/c/numeric/math/exp
[25] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/exp.html
[26] https://git.bloerg.de/studium/exp-issue

random number r	expf(r) (CUDA)	exp(r) (OpenCL)
0.462963	1.58878	1.58877
0.839878	2.31609	2.31608
0.645550	1.90704	1.90703
0.318770	1.37544	1.37543
0.518338	1.67924	1.67923
0.842220	2.32152	2.32151
0.853306	2.34740	2.34739
0.790821	2.20521	2.20520
0.635833	1.88860	1.88859
0.358126	1.43065	1.43064
0.487836	1.62879	1.62879
0.243214	1.27534	1.27534
0.267595	1.30682	1.30682
0.773415	2.16715	2.16715
0.72402	2.06271	2.06271
0.859754	2.36258	2.36258
0.408421	1.50444	1.50444
0.411015	1.50835	1.50835
0.174236	1.19034	1.19034
0.576942	1.78059	1.78059

Table 2: Intentionally picked lines from longer list of unsorted random floats r and exponential function results for cmath's expf(r) and OpenCL exp(r). Lines in the upper half show minor difference in the last digit of the function result. Lower half lines from the same list do not.

0.6.4 Qualitative equivalence of the port

For the purpose of basic functionality testing, an $L = 8$ 2d Ising lattice was simulated with both the original cudamuca implementation and with the OpenCl implementation. The final histogram of the production run was compared to the exact solution for an $L = 8$ lattice provided by Beale.[27] Figure 3 illustrates the very close similarity of both implementations with the exact solution. This becomes more clear, when the deviations of the simulation results from the exact solution are plotted next to each other. Figure 4 shows the difference of the logarithm of the Beale solution state density and the logarithm of the simulation's state densities. It is obvious that the OpenCL simulation is at least as close as the CUDA simulation.

0.7 Benchmark

TODO: run-time-Vergleich OpenCL - CUDA: 8, 16, 32?

[27] http://spot.colorado.edu/ beale/, http://www.physik.uni-leipzig.de/ janke/teaching/beale_8x8.html

0.8 Discussion

Ausarbeiten:

- The issue with exp/expf was found. It can be expected that its impact is of minor importance. Already the use of higher accuracy data types could cause similar behaviour.
- The OpenCL implementation is slower than the CUDA implementation. This could be caused by not having textures, insufficient exploiting of caching.
- Performance gains maybe by using OpenCL image (not tried), use of local memory for histograms, weights.

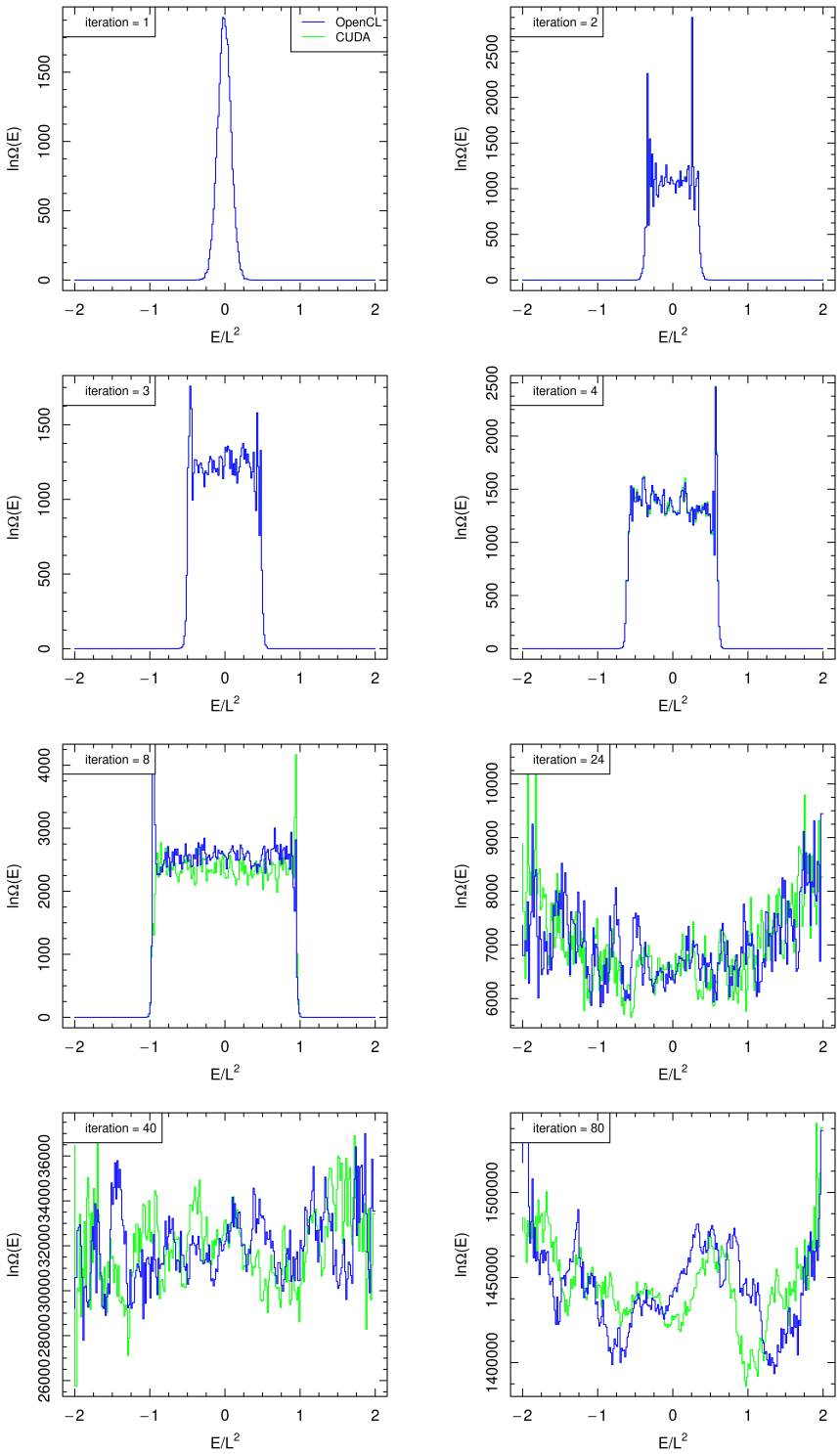

Figure 2: $ln\Omega(E)$ of an Ising lattice with $L = 16$ during training iteration step indicated in upper left of the diagrams. At iterations with lower numbers, histograms of CUDA implementation (green graph) and OpenCL implementation (blue graph) are identical, then start to diverge slightly. TODO: Achsen schön machen.

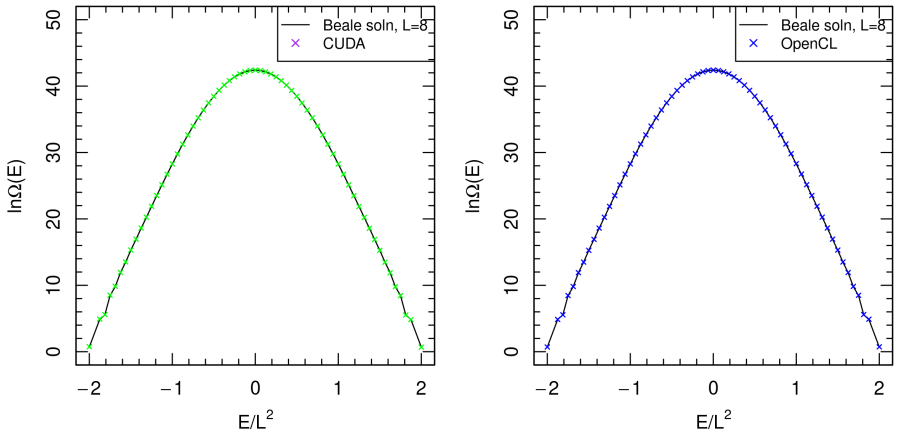

Figure 3: Logarithm of the density of states of the simulated 2d Ising model in comparison to exact Beale solution ($L = 8$). Data points mark simulation results for original CUDA implementation (left hand side) and OpenCL port (right hand side). Simulated values are normalized with respect to maximum in Beale's solution.

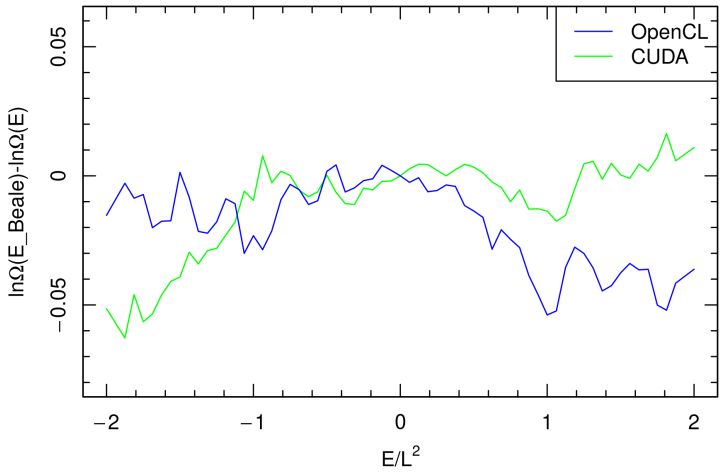

Figure 4: Difference of the Logarithms of the density of states of the exact Beale solution and the CUDA/OpenCL simulation for an $L = 8$ lattice.