Project Ising Model

Programming of Parallel Computers

Roel Deckers
roel@codingcat.nl
930830-T150

Abstract

The Ising model is a statisticall model for ferromagnetism. A popular algorithm for simulating this model is the so called Metropolis algorithm. Useful results require many iterations however, and therefore performance should be considered when implementing the Metropolis algorithm. In this report a GPGPU implementation using OpenCL is presented and discussed.

1. *Introduction*

1a) *The Ising Model*

The Ising model is a statistical model for ferromagnetism. It concerns the magnetic dipole moments of atomic spins, in the model these spins can be in one of two states, spin-up (+1) or spin-down (-1). Usually these spin elements are laid on an equidistant grid with a periodic boundary condition. This model can be used to determine several interesting properties of ferromagnets, such as phase-transitions, total-energy, total magnetism and heat capacity in vacuo or the presence of an external magnetic field.

In this report we focus on equidistant grids in two dimensions of varying sizes. As our focus is on optimizing the performance of the algorithm and not the underlying physics we will only present the gradient of the energy with respect to the binding strength in between spin-sites in vacuo as a proof-of-correctness.

1b) *Metropolis Algorithm*

A step of the Metropolis algorithm works as described in algorithm 1, this step is repeated several times in order to reach an equilibrium state. It should be noted that there are a variety of ways to pick which spin-site to check. This can be done by selecting a random site, or by walking over each point sequentially as well as using red-black coloring. We will employ the latter technique and defer any discussion on the statisticall properties of these various sampling methods as these would go beyond the scope of this report.

Algorithm 1 Metropolis Algorithm step in Two Dimensions

1: $Coords \leftarrow$ An appropriately chosen spin-site. ▷ Spin sites can be selected in various ways.
2: $C \leftarrow$ Spin value at $Coords$
3: $N \leftarrow$ Spin value at $Coords + (0, +1)$
4: $E \leftarrow$ Spin value at $Coords + (+1, 0)$
5: $S \leftarrow$ Spin value at $Coords + (0, -1)$
6: $W \leftarrow$ Spin value at $Coords + (-1, 0)$
7: $f \leftarrow N + E + S + W$
8: $r \leftarrow$ A random number from the uniform distribution $[0, 1]$
9: **if** $r \leq \exp(-2C(Jf + B))$ **then** ▷ J is the binding constant, B the external magnetic field.
10: Flip the value at $Coords$.

2. *Implementation*

When updating the Ising Model using the Metropolis algorithm and a red-black updating scheme one can update half of the points in parallel at each step. This highly parallel scheme lends itself well to be adopted to massively parallel accelerators such as GPU (For this report we have targeted a AMD r280x). Although the algorithm has a relatively high memory-access to computation ratio steps can be taken to reduce this ratio. A difficulty when implementing the Metropolis algorithm in OpenCL is the fact that OpenCL does not have a build in random number generation function and thus one has to implement their own. And because many random number generators require the user to keep track of a state, and we need a large amount of independent generators for all the sites this adds a great deal of overhead. Luckilly a simple linear congruential generator of the form $x \leftarrow x * \texttt{0xDEECE66D} + \texttt{0xB} \mod 2^{32}$ seems to be adequate and using some tricks we will discuss later, we have managed to reduce the memory requirement for the states to half a bit per spin-site.

We will now quickly describe the various implementations we will show in this report.

2a) *Naive-transfer*

This is a straigthforward implementation of the Metropolis Algorithm using red-black coloring and for each iteration (updating all red or black points) random numbers are generated on the host and transferred asynchronouslly to the device. This implementation serves as a baseline worst-case performance.

2b) *Naive-rng*

This implementation is non-optimized but generates the random numbers on the device using the previously discussed LCG.

2c) *Optimized*

This is the fully optimized implementation. The applies optimizations are

1. *Cached flip chances:* Looking at algorithm 1 we note that f can only ever take 10 distinc values at most, and it is therefore possible to cache these values in a lookup table instead of evaluating the expensive exponential function. We do so by generating the table on the host and then storing it in the fast (and cached) constant memory.

2. *Double red-black coloring:* use a double red-black coloring scheme. First the total grid is dividided into blocks of 128×128 sites and colored red-black. Each block is then updated using another red-black scheme. This speeds up the simulation by allowing us to reduce the loads from global memory by first caching a block into the much faster local memory. It has been suggested by others that it is possible to perform several updates on the cached block before moving on to the next block iteration, we have implemented this and the amount of "inner loops" can be controlled by the parameter k. It seems however that doing so leads to incorrect results.[1]

3. *Reduced data-load by storing spins in a single bit:* Because spins can only be up or down, it is theoretically (and practically) possible to store them in a single bit each. We have done so by organizing the data in a layout where the first 32 bits correspond to the first 32 red spin-sites, followed by the first 32 black spin-sites and then it repeats. Proccessing this data requires a bit

[1] This might be a misreading of the paper I used as a reference, as it was a single line without any other context or code, or due to a different random number generator, as the quality of the results is somewhat dependent upon the quality of the random numbers.

more work of course. For example when calculating f we need at least 4 bits per site to generate an index. For a full description of how we have solved this we reffer to the source code[2]

4. *Reduced data load by reusing the rng:* This optimization is the kind which could break the actual results, but luckilly it works for us, as it is based on trying to get the lowest quality random numbers we can get away it. Now LCGs are already highly correlated and low-quality random numbers. What we ended up doing is keeping one (32 bit) state per work item, which updates 64 sites, leading to a half-a-bit per site. We try to reduce the correlation by creating 4 streams by XOR-ing the state which randomly chosen constants and then looping over using each of these streams and writting back the non XOR-ed result to the state when we are done.

All of these optimizations do come at the price of a minimum size of 256×256, which is relatively large. Even larger systems could be optimized more by using larger block sizes as each block needs all the border points in addition to its interior points and due to the bit-packing we have used these borders are a significant part of the total memory loads. (roughly 1/3).

3. *Results*

The result of our work can be summarized in two graphs, figure 1 which show the performance increase of our methods, and figure 2 which shows the consistency of our method. Note how the results for $k > 1$ are inconsistent with the others. This might be an issue of convergence (the system has trouble reaching equilibrium with the reduced loop count) or due to a subltelty with the PRNG.

In terms of performance we see a clear increase by using our optimized method, save for one point at a system size of 2^8 which seems to be due to the reduced number of work-items not saturating the device. Using $k > 1$ does give a significant speed-up still however significantly less than what we gained in previous steps.

4. *Conclusion*

We can conclude that we have successfully optimized our Ising model implementation, and demonstrated the importance of both reducing transfers between host and device and from memory to compute-unit. We applied a novel bit-packing method to increase the ratio of computation to memory access and greatly reduced the memory requirements for both the spin-states as well as the PRNG. Further improvements could perhaps be made by increasing the block size and/or improving the data layout to improve data-traffic for the dual-red-black coloring scheme.

5. *A note on measuring performance.*

It is common to measure the performance of Ising Models using either flips-per-second or time-per-flip and hence this was used in this report. However, for practicall purposes we want sample N independent states at a certain equilibrium (due to the stochastic nature these will vary, and in fact the variance is of physicall importance) and therefore what we should be measuring is the real-time autocorrelation time of the system. The autocorrelation time is a measure of how long it takes for a state to become independent of its initial configuration. Using a stronger PRNG for example might lead to a much lower autocorrelation time (in terms of iterations) at the cost of a much longer iteration time.

[2] It's a little big magic-y, but also very hard to explain in text.

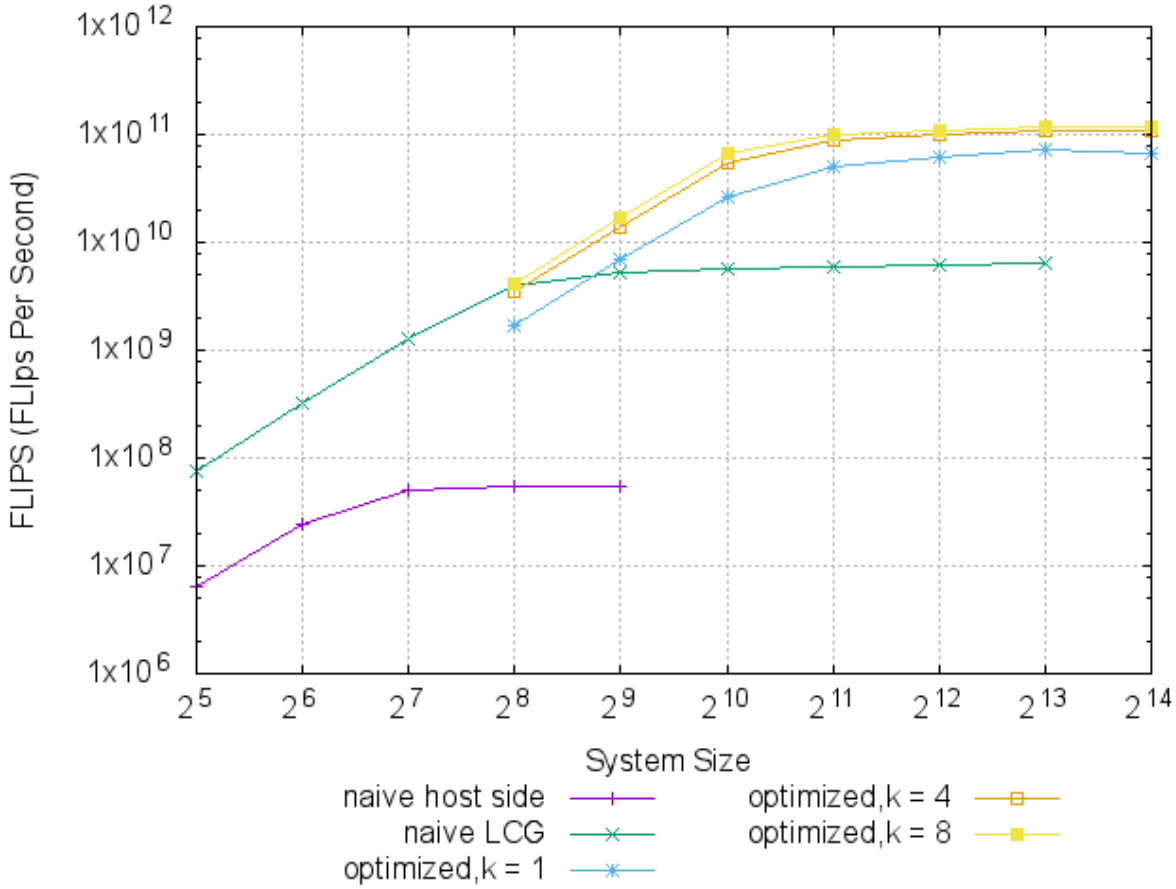

Figure 1: Performance of various methods as a function of system size

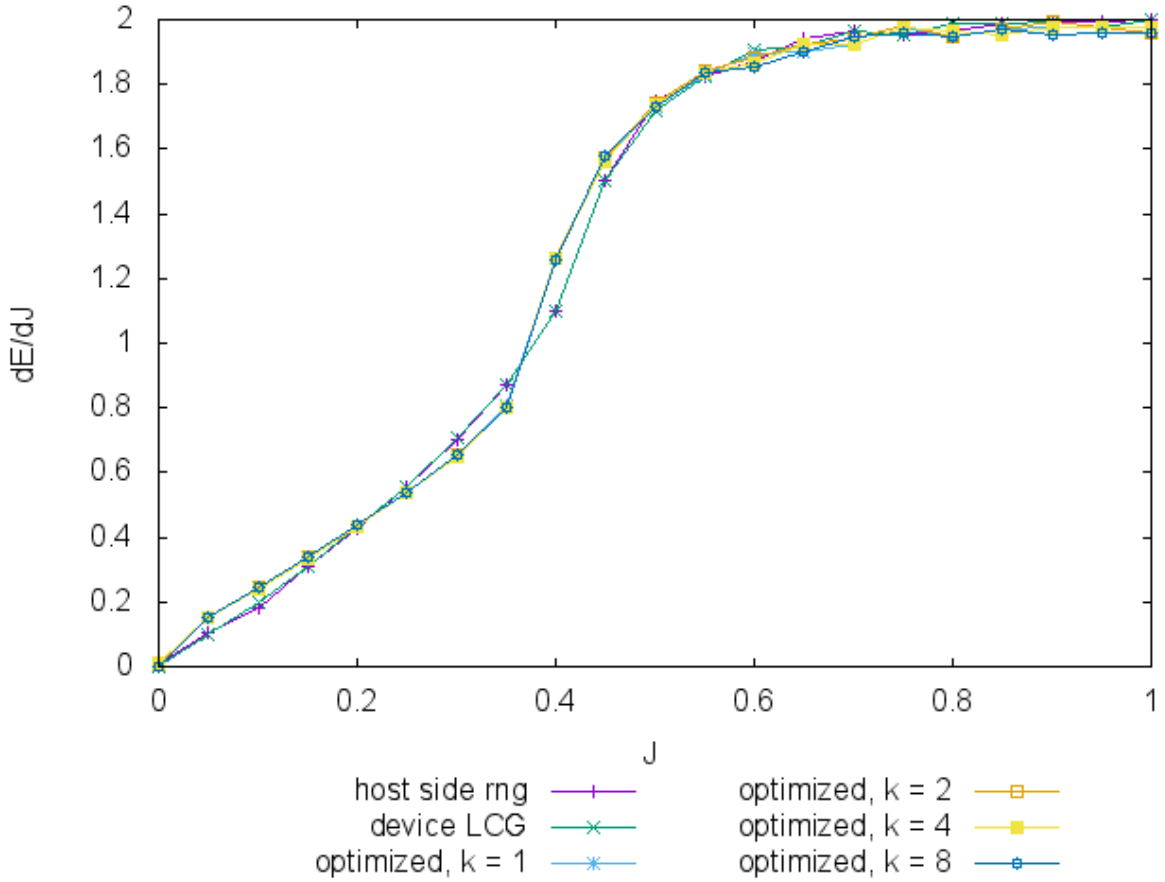

Figure 2: Gradient of the energy for the various methods.