OPTIMIZATION OF THE METROPOLIS ALGORITHM FOR A POTTS MODEL

Dominik Gresch, Mario Könz

Department of Physics
ETH Zürich
Zürich, Switzerland

ABSTRACT

1. INTRODUCTION

1.1. Motivation

1.2. Related work

2. BACKGROUND: POTTS MODEL AND THE METROPOLIS ALGORITHM

2.1. Potts Model

The n-state Potts model ensemble of spins on a lattice. n different states are available for each spin. These spins interact only with their nearest neighbors. The energy of the entire system is given by:

$$E = J \sum_{<i,j>} \sigma_i^z \sigma_j^z$$

where J is an interaction constant, σ_i^z the z-component spin at lattice site i and $< i, j >$ the set of all neighbor-pairs. If $J < 0$ the spins align preferable aligned. The 2-state Potts model on a 2D square lattice is identical with the classical Ising model. For this work the 4-states Potts model on an 3D cubic lattice with periodic boundary conditions was chosen. The mean properties of the model can be measured in the following way:

$$\bar{A} = \sum_{\omega \in \Omega} A(\omega) * p(\omega)$$

where Ω denotes the configuration space, $p(\omega) = e^{-\frac{E(\omega)}{k_b T}}$ the canonical probability to find a certain configuration ω in Ω and A and observable like energy or magnetization. Due to the large configuration space (increases exponentially with the number of spins) this sum cannot be calculated directly in a fast manner. Therefor one needs the metropolis algorithm.

2.2. Metropolis Algorithm

The metropolis algorithm is a Markov chain Monte Carlo method [1] that allows the calculation of expectation values without having to generate all configurations ω. After generating a starting state ω_0 one does many local spin updates to change the system and generate the next state ω_1. One has make sure that the states are not correlated, otherwise the error estimations is too small. For the Potts model, the algorithm consists of following steps:

1. Select a random spin σ_i^z from the current configuration ω_s and change it randomly by $\Delta\sigma_i^z = \pm 1$ to the configuration ω_s^*
2. Calculate the energy difference ΔE_i and $\frac{p(\omega_s^*)}{p(\omega_s)} = e^{-\frac{\Delta E_i}{k_b T}}$
3. Accept the new configuration with the probability $p_{accept} = min(1, \frac{p(\omega_s*)}{p(\omega_s)})$
4. Repeat step 1 to 3 m times until ω_s and ω_{s+m} are decorrelated
5. Measure functions of interest on the state ω_{s+m}
6. If the error in the measurements is too large, repeat from step 1

3. MODULAR OPTIMIZATIONS AND AUTOTUNING

The following section describes the implementation of the Potts model, specifically the splitting into four modules, optimizations performed on those modules and finally autotuning to select the best-performing modules.

3.1. Modular Structure

Splitting the implementation into four modules (with their respective interfaces) allowed for an efficient workflow. It became possible to optimize parts of the implementation completely independently, which is a great benefit when working in a team. The four modules are as follows:

- **`SIM`:** Contains the high-level aspects of the metropolis algorithm: Computing probabilities, accepting or rejecting an update, and also calls to the other modules (i.e. requesting random numbers or getting / setting spins).
- **`GRID`:** Takes care of the boundary conditions, i.e. it computes the nearest neighbours of a selected spin.

- **MATRIX:** Contains the explicit data format of the system (e.g. `std::vector` or `C array`).
- **RNG:** Provides the random numbers. All implementations use a mersenne twister engine.

In the following parts, the different optimazation techniques used for each module will briefly be mentioned.

3.2. SIM optimizations

- **Probability Precomputation:** Since there is a finite number of possible energy differences ΔE, the probability $p_i = \exp\left(-\frac{\Delta E_i}{k_B T}\right)$ of accepting a spin change is precomputed. This may increase memory usage, but gets rid of any floating point computation.
- **Interleaving:** The calls to the random number generator (picking the spin location) and the actual computation is interleaved amongst two steps. This allows prefetching, i.e. the data for the next step can be loaded during the current step.
- **Explicit Prefetching:** Instead of leaving it up to the Compiler and/or Hardware, the prefetching is done explicitly.

3.3. GRID optimizations

A **lookup table** (one array for each space direction) was used to get the nearest neighbour indices efficiently. This method is useful mainly because it replaces the potentially expensive check for the boundary condition.

3.4. MATRIX optimizations

The main concern with the `MATRIX` module storing the system in a memory - efficient way. Also, since the access to spins is random, the `MATRIX` is the only way we can hope to achieve locality (between one position and its nearest neighbours). The following optimizations have been done:

- **Compression:** Since each spin can only be in one of four states, it can be described using only 2 `bits`. This can be used to shrink the system's size in memory by a factor of 4 (compared to using one `byte` per spin).
- **Z - order:** Bit interleaving in the three indices is used to increase locality amongst a spin and its nearest neighbours.

3.5. RNG optimizations

- **Economic use:** Since many of the random numbers used only are a few `bits` long, the `std::mt19937` implementation (which uses 32`bit` for each random number at least) can be optimized greatly by using all the bits in each generated random number (with some overhead for splitting up the random numbers).
- **MKL:** The underlying Mersenne twister engine was exchanged for the MKL implementation [2] (whilst still using the method of economic use described before).

3.6. Autotuning

Many of the optimizations seen in sections 3.2 - 3.5 perform well only under certain conditions (i.e. at certain system sizes and temperatures). For example, using Z - order might be great at large sizes, but the overhead for computing the index is too large for smaller ones.
The modular structure of the code provides the ideal grounds for tackling this problem using an autotuning approach:
For a fixed number of sizes, the optimized combination of modules is searched (either by doing a full sweep over all the combinations, or - much faster - by iteratively exchanging one module whilst keeping the others fixed). This installation routine creates a header file (`install.hpp`) containing template specialisations, as seen in Listing 1.

Listing 1. example for a template specialisation

```
template<>
struct opt<150> {
   template<int S>
   using impl = greschd_v3_sim::impl<S, S,
       S, addon::mkl_mt_rng, msk_v1_pbc,
       msk_v2_dynamic_zip>;
};
```

For an arbitrary size N, the implementation will then choose the appropriate specialisation (that is, the one with size closest to N) by use of type traits, as indicated in Listing 2.

Listing 2. size_helper, part of the type traits

```
template<int S, int N, bool B>
struct size_helper {
   static constexpr int size =
       size_helper<S, N - 1, S >= (sizes[N]
       + sizes[N - 1])/ 2 >::size;
};
```

4. EXPERIMENTAL RESULTS

This section describes the different measurements that were performed and their results. After introducing the platforms that were used and some notation, the impact each optimisation has on runtime is studied. Finally, the results of screening all module combinations and the performance of the auto - tuned code is shown.

4.1. Experimental setup

The two platforms used are shown in Table 1.

	Haswell	Wolfdale
Architecture	Intel Core i7	Intel Core 2
Frequency	2.4 GHz	2.4 GHz
OS	Ubuntu 13.10	Ubuntu 14.04
Compiler	gcc 4.8.1	gcc 4.8.1
Flags	-std=c++11 -DNDEBUG -O3 -march=core-avx2	-std=c++11 -DNDEBUG -O3 -march=core2

Table 1. Platforms used for the measurements

For all measurements, the rdtsc command was used to determine the runtime. In the roofline measurements, the memory traffic was measured with perfplot [3]. Additionally, VTune, perf and gprof were used for profiling during development. also used.
Measurements showing the splitting of runtime amongst the different modules is achieved by starting / stopping the time measurement between different tasks in the algorithm. This may to some extent influence the total runtime by inhibiting out-of-order computation.
Runtime of `GRID` and `MATRIX` is measured together, but split into accessing selected spin itself (single spin, s.s.) and accessing its nearest neighbours (n.n.).
The exact configuration of modules used in a measurement is noted using a four - digit index corresponding to (SIM, GRID, MATRIX, RNG) as listed in Table 2 (with optimisations as described in Sec. 3):

4.2. Results: SIM optimization

Figures 1 and 2 show the runtime partition of the single spin update. All modules except SIM are the same. The baseline-module computes the Boltzmann factors each time, while the second module precomputes these probabillities. The amount of cycles needed for the SIM part is reduced on both systems. On Haswell, this shows a 2x speedup for a system with side length $N = 20$.

	index	optimisations
SIM:		
	0	baseline
	1	probability precomputation
	2	1 & interleaving
	3	2 & explicit prefetching
GRID:		
	0	baseline
	1	boundary lookup table
MATRIX:		
	0	baseline (`std::vector`)
	1	C array
	2	compressed
	3	Z - order
	4	compressed & Z - order
RNG:		
	0	baseline (STL mt)
	1	economic use
	2	1 & MKL engine

Table 2. List of Modules

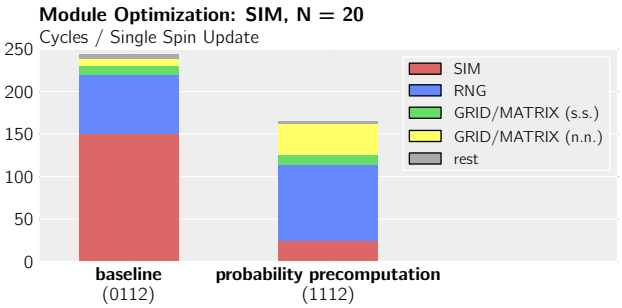

Fig. 1. Speedup by probability precomputation, on Wolfdale

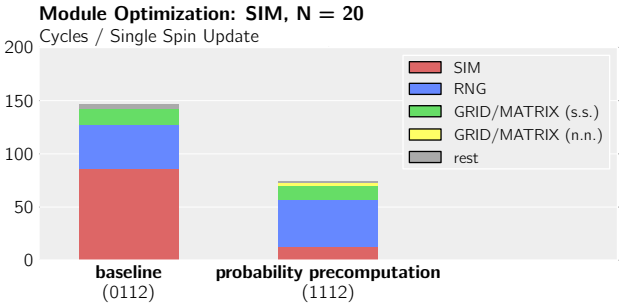

Fig. 2. Speedup by probability precomputation, on Haswell

4.3. Results: GRID optimization

Storing the boundary condition in a lookup table doesn't show a very significant speedup in terms of total runtime. However, nearest neighbour access - the part that is influenced most by the improvement - is sped up by roughly 2x for sidelength $N = 20$ (see Fig. 3 and 4).

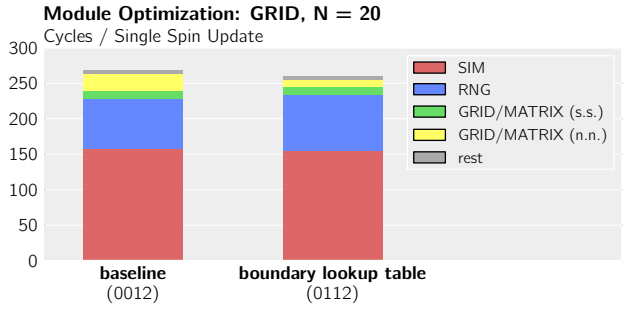

Fig. 3. Influence on runtime by using a boundary lookup table (`GRID` optimisation), on Wolfdale

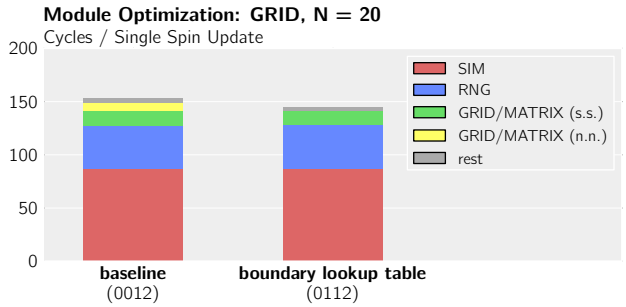

Fig. 4. Influence on runtime by using a boundary lookup table (`GRID` optimisation), on Haswell

4.4. Results: MATRIX optimization

The impact compression and Z-order have on the runtime is clearly size - dependent: For a small system ($N = 20$, 8000 spins, on Wolfdale), the influence of compression is negligible and the overhead for using Z - order is larger than the gain (see Fig. 5).
At larger sizes, as the impact of accessing the spins increases, compresssion and Z-order both become significant improvements (see Fig. 6). This improvement is most noticeable when the system starts running out of last level cache. At those sizes, the compressed version might still fit into LLC and hence perform much better.
On Haswell, it can be observed that even for large sizes

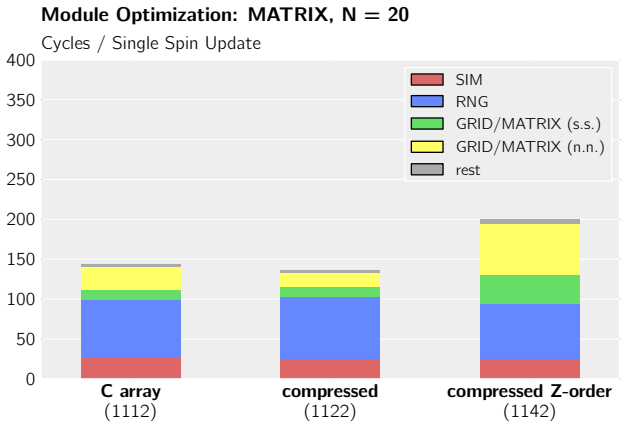

Fig. 5. Different `MATRIX` optimisations for $N = 20$, on Wolfdale

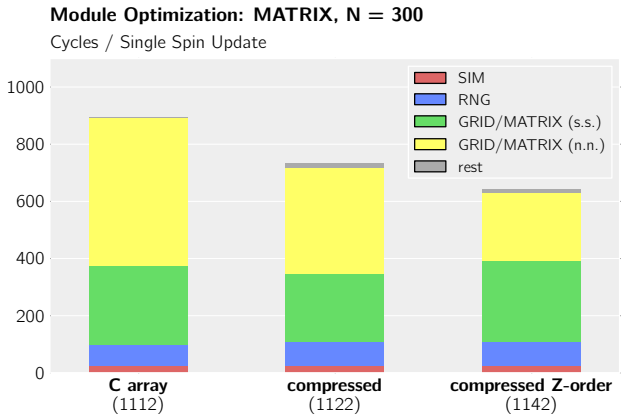

Fig. 6. Different `MATRIX` optimisations for $N = 300$, on Wolfdale

($N = 1000$, 10^9 spins), accessing the nearest neighbours takes next to no time (see Fig. 7). This might be due to the prefetching in Haswell, which could load the neighbours into Cache as soon as the position of the single spin is known. However, this would still cost memory bandwidth, but go undetected by this measuring scheme since it happens out-of-order (see Sec. 4.1).

4.5. Results: RNG optimization

Economic use of the random numbers shows significant impact ($>$ 2x for small sizes). Using the MKL engine however shows some speedup, but not a negligible one (see Fig. 8).

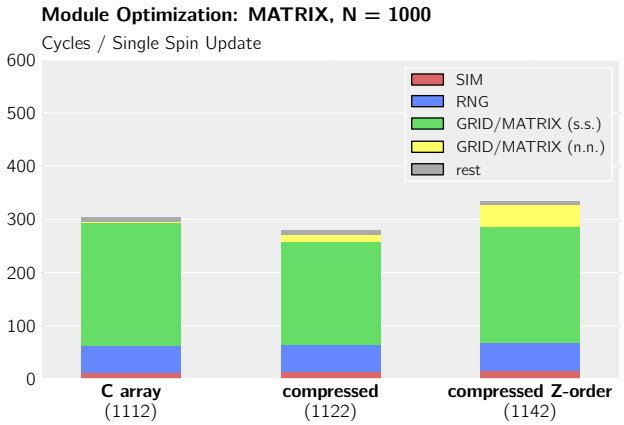

Fig. 7. Different `MATRIX` optimisations for $N = 1000$, on Haswell

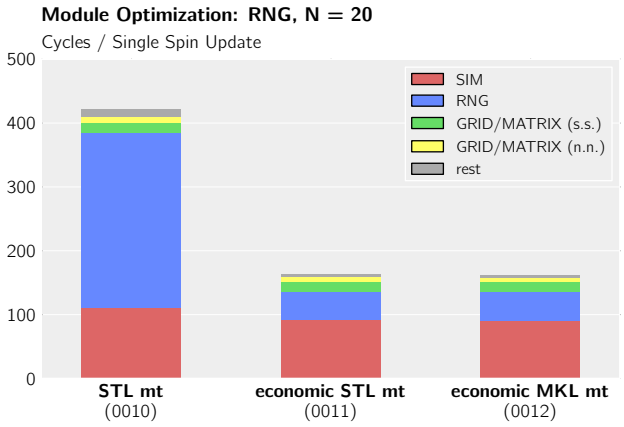

Fig. 8. Runtime using different `RNG` modules, on Haswell

4.6. Results: Autotuning

4.7. Results: Performance

5. CONCLUSIONS

6. REFERENCES

[1] Werner Krauth, "Introduction to monte carlo algorithms," arXiv:cond-mat/9612186, 2006.

[2] Intel, "Math kernel library," `http://developer.intel.com/software/products/mkl`.

[3] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Püschel, "Applying the roofline model," Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014.

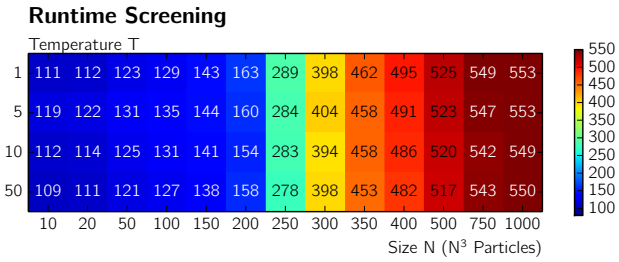

Fig. 9. Performance Screen, Wolfdale

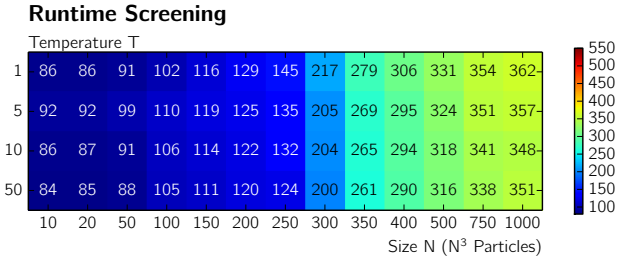

Fig. 10. Performance Screen, Haswell

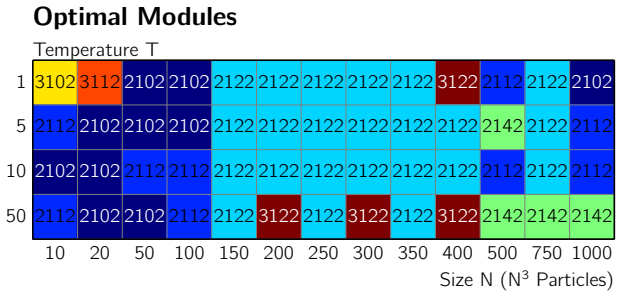

Fig. 11. Modules, Wolfdale

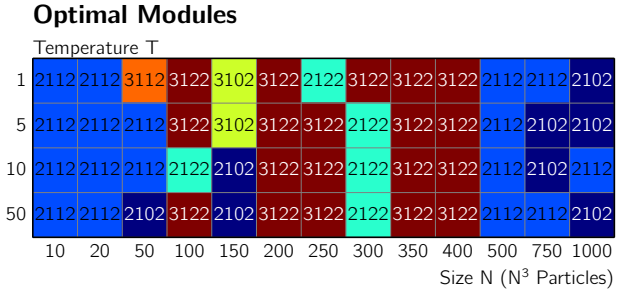

Fig. 12. Modules, Haswell

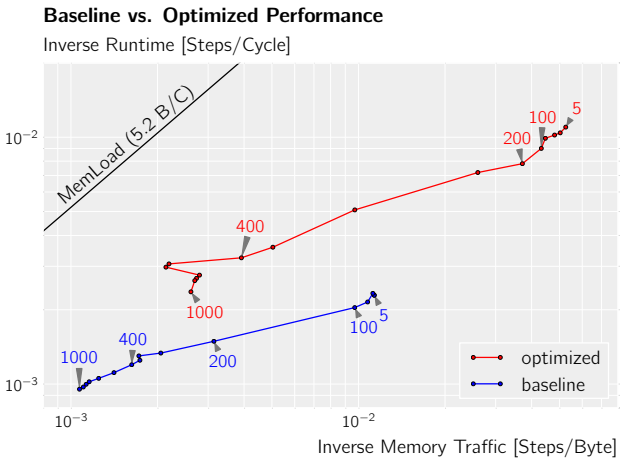

Fig. 13. Roofline, opt. vs baseline, Haswell

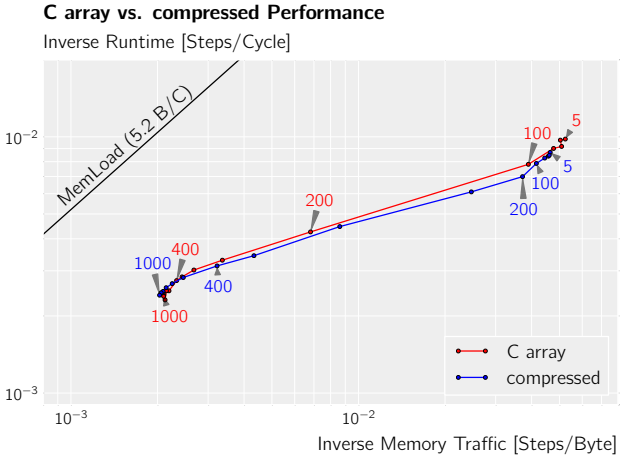

Fig. 14. Roofline, C array and compressed, Haswell