

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:

$$E = J \sum_{\langle i,j \rangle} \sigma_i^z \sigma_j^z$$

where J is an interaction constant, σ_i^z the z -component spin at lattice site i and $\langle i,j \rangle$ 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 a 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. Therefore 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 to make sure that the states are not correlated, otherwise the error estimations are too small. For the Potts model, the algorithm consists of the 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 and interfaces to the `MATRIX` module.

- **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 optimization 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 32bit 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 optimisations 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.

That means one starts with a certain combination of modules and then tries all possible combinations by only exchanging the first one. The fastest version amongst those is kept for the next step, in which the second module is being exchanged, and so on.

This process is repeated until convergence is reached, that is until a full iteration (trying to exchange each modules) does not yield any faster combinations. However, this does **not guarantee** that the process has found the **global minimum**, since it might be stuck at a local one.

This installation routine creates a header file containing template specialisations describing the best combination for each measured size. Listing 1 shows a snippet of this header.

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;
```

```
};
```

Making an installation that also varies according to temperature was also discussed, but dismissed because the temperature of the system may be varied during one measurement cycle. This could lead to a large overhead for switching between versions.

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 for measurements are shown in Table 1. They will be referenced by their architecture code-name (Wolfdale / Haswell).

	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
	-std=c++11	-std=c++11
Flags	-DNDEBUG	-DNDEBUG
	-O3	-O3
	-march=core-avx2	-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.

In the following sections, combinations of modules where one module is being exchanged will be discussed. For those measurements, one can look at both decrease in total runtime and decrease in **runtime caused** by that **specific module**. However, the former is always influenced by the choice of all the other modules, which makes the latter a better measure for the effectivity of an optimisation.

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).

As an example, (0132) would correspond to a baseline SIM with boundary lookup table in GRID, Z-order MATRIX and economic use & MKL RNG.

	digit	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

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 probabilities. The amount of cycles needed for the SIM part is reduced on both systems. On Haswell, this shows an overall speedup of 2x for a system with side length $N = 20$. Considering only the part of the runtime caused by the SIM module, however, the speedup is roughly 6x.

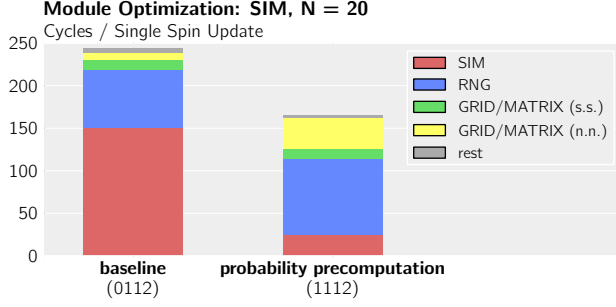


Fig. 1: Speedup by probability precomputation, on Wolfdale

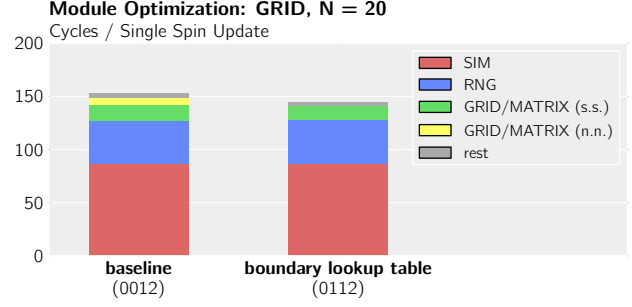


Fig. 4: Influence on runtime by using a boundary lookup table (GRID optimisation), on Haswell

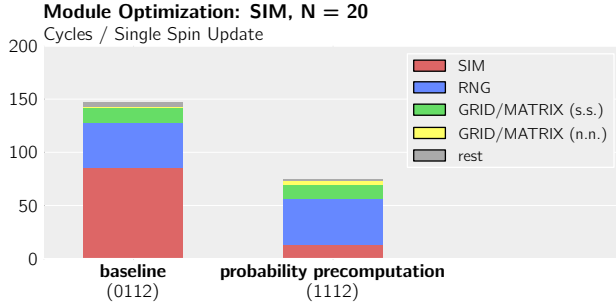


Fig. 2: Speedup by probability precomputation, on Haswell

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).

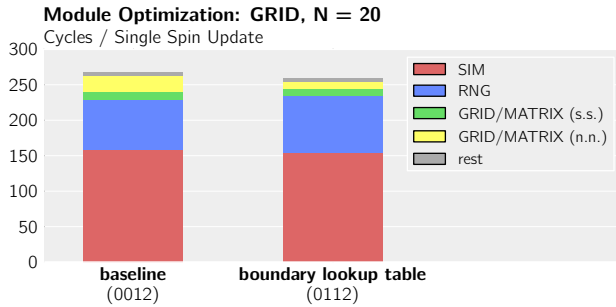


Fig. 3: Influence on runtime by using a boundary lookup table (GRID optimisation), on Wolfdale

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).

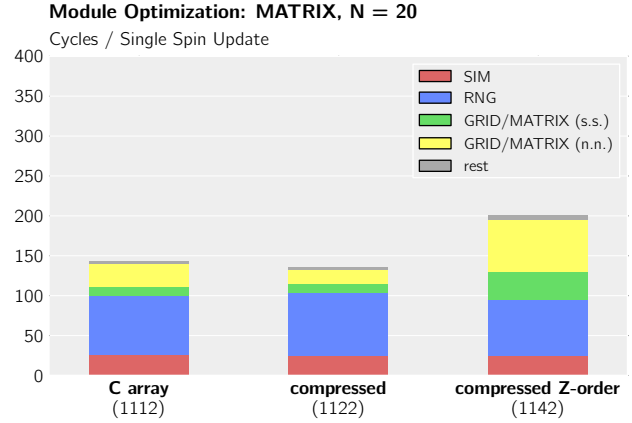


Fig. 5: Different MATRIX optimisations for $N = 20$, on Wolfdale

At larger sizes, as the impact of accessing the spins increases, compression 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.

This effect can be observed directly in Fig. 7, comparing C array and compressed MATRIX modules. Whilst the two modules are along the same line in the roofline plot, the system size at which the performance starts dropping is different (C array: $N \approx 100$, compressed: $N \approx 200$).

On Haswell, it can be observed that even for large sizes ($N = 1000$, 10^9 spins), accessing the nearest neighbours takes next to no time (see Fig. 8). 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).

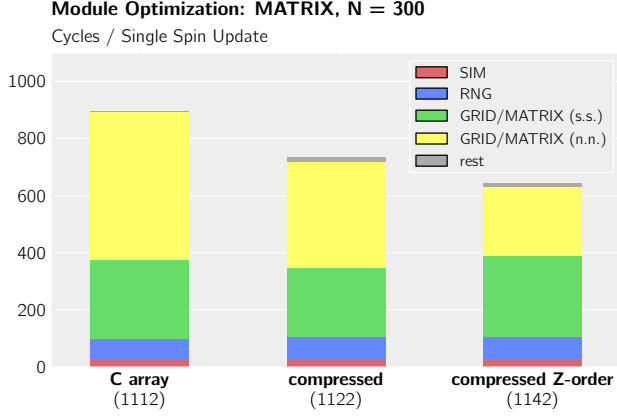


Fig. 6: Different MATRIX optimisations for $N = 300$, on Wolfdale

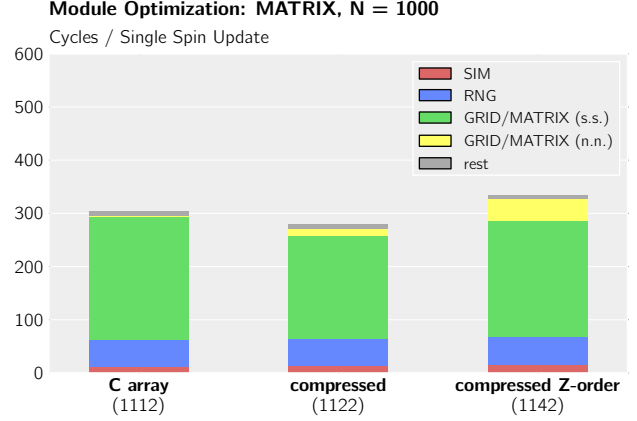


Fig. 8: Different MATRIX optimisations for $N = 1000$, on Haswell

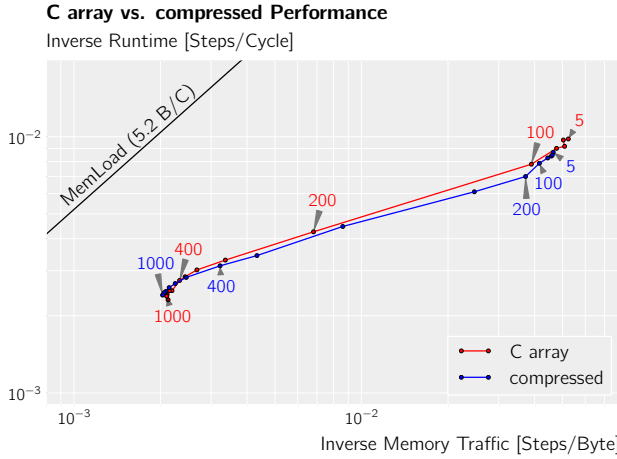


Fig. 7: Performance of C array and compressed MATRIX modules, for various sizes, on Haswell

4.5. Results: RNG optimization

The impact of economic use of random numbers and the MKL Mersenne - twister engine (as explained in Sec. 3.5) are shown in Fig. 9.

It can be seen that the main speedup comes from economic use, whilst changing to the MKL engine has virtually no effect on runtime.

For a system of size $N = 20$ (8000 spins), these optimisations amount to an overall speedup of roughly 2 – 2.5x (on Haswell). Considering only the part of total runtime caused by the RNG module, the speedup is roughly 6x, and hence similar to the speedup seen for SIM.

4.6. Results: Autotuning

A full screen of all the runtimes of all the module combinations has been done to determine which perform best at

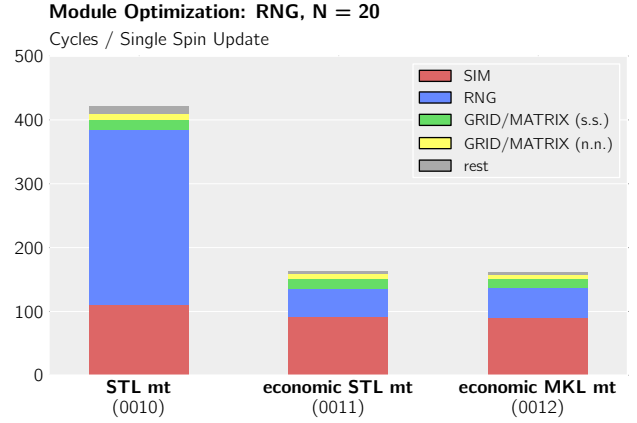


Fig. 9: Runtime using different RNG modules, on Haswell

various sizes and temperatures. The runtimes achieved by the fastest version can be seen in Fig. 10 (Wolfdale) and 11 (Haswell), which clearly shows the speedup between Wolfdale and Haswell (close to 2x).

Fig. 12 and 13 show which module combination performed best, using the indexing introduced in Sec. 4.1. A few points are worth noting:

Firstly, the module combinations used on the two platforms are sometimes quite different from each other and do not have too much of a pattern. This indicates that it might be quite difficult to actually predict which module works best. Secondly, it has to be noted that for some sizes, the difference between fastest and second - fastest module combinations is very small - sometimes even within measurement error. So it might be possible to construct a neater arrangement of modules by allowing those second - best combinations.

Thirdly, whilst the temperature does have some impact on which module is used, the dominant factor seems to be the size. This justifies not optimising w.r.t. temperature in the

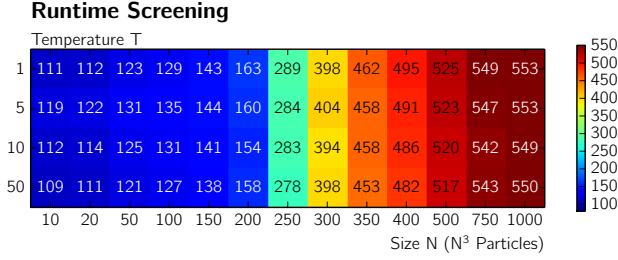


Fig. 10: Runtime of the fastest module combinations, on Wolfdale

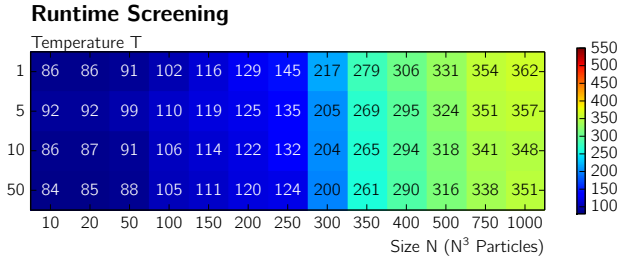


Fig. 11: Runtime of the fastest module combinations, on Haswell

installation procedure (see Sec. 3.6).

Lastly, the region $N \in [150, 400]$ (on Wolfdale) / $N \in [200, 400]$ (on Haswell) is of particular interest. This region corresponds to the system size running out of LLC. Accordingly, the MATRIX module used in this region is the compressed version (third digit of the index = 2).

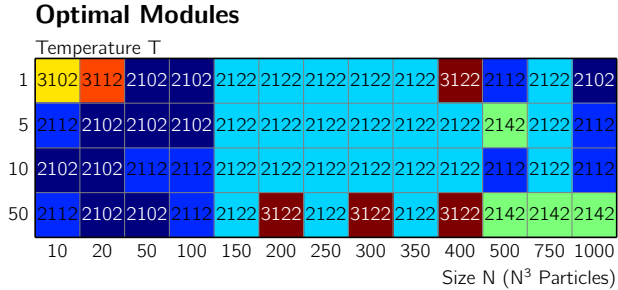


Fig. 12: Best - performing modules (marked by index) for different sizes and temperatures, on Wolfdale

The performance of the final version, using the fast installation procedure described in Sec. 3.6, is shown in Fig. 14. Compared to the baseline implementation, the overall speedup ranges between 2 – 5x, depending on system size. The speedup tends to be bigger for smaller sizes, where the vastly improved SIM and RNG modules takes up a significant part of the runtime.

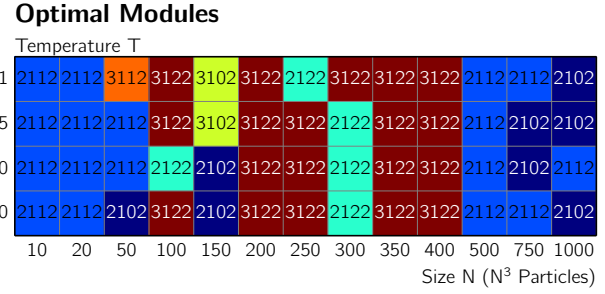


Fig. 13: Best - performing modules (marked by index) for different sizes and temperatures, on Haswell

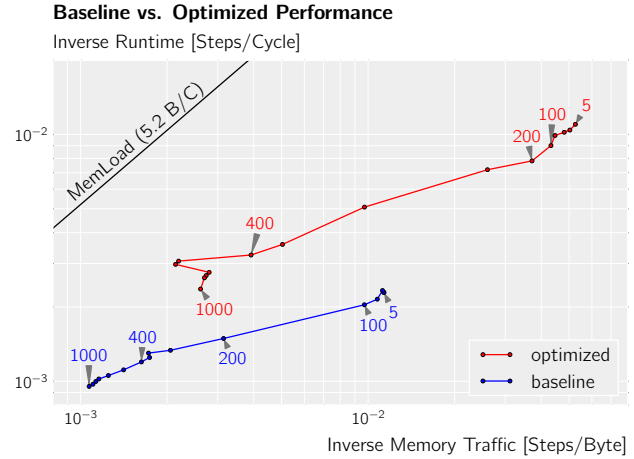


Fig. 14: Final (installed) version of the code vs. baseline implementation, on Haswell

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 Offenbeck, 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.