Computing and Informatics, Vol. 33, 2014, 1191–1208

GPU-ACCELERATED AND CPU SIMD OPTIMIZED MONTE CARLO SIMULATION OF φ^4 MODEL

Piotr Białas

Faculty of Physics, Astronomy and Applied Computer Science
Jagiellonian University, ul. Reymonta 4, 30-059 Krakow, Poland
&
Mark Kac Complex Systems Research Centre
Faculty of Physics, Astronomy and Applied Computer Science
Jagiellonian University, ul. Reymonta 4, 30-059 Krakow, Poland
e-mail: piotr.bialas@uj.edu.pl

Jakub Kowal, Adam Strzelecki

Faculty of Physics, Astronomy and Applied Computer Science
Jagiellonian University, ul. Reymonta 4, 30-059 Krakow, Poland
e-mail: {jakub.kowal, adam.strzelecki}@uj.edu.pl

Abstract. In this contribution we describe an efficient GPU implementation of the Monte-Carlo simulation of the Ginzburg-Landau model. We achieve the performance close to 50 % of the peak performance of the used GPU. We compare this performance with a parallelized and vectorized CPU code and discuss the observed differences.

Keywords: GPU computing, vectorization, SIMD, Monte-Carlo simulations

Mathematics Subject Classification 2010: 65Y05, 65Y10, 82B99

1 INTRODUCTION

The Graphical Processing Units (GPU) emerge these days as the most efficient devices for numerical calculations. To ascertain this it is sufficient to look at the list

of 500 supercomputers and count how many of them have GPU accelerators; and this trend is still increasing [2]. This was made possible not only by the accessibility of the hardware but also by the realization from the side of the vendors that those units can be used also for computations, and providing user with suitable software interface to do this. Technologies like CUDA and OpenCL make the GPU relatively easily accessible to C/C++ programmers. In the meantime also most of the numerical packages vendors have added GPU support to their tools.

However, like with most of the computer languages there is a learning gap between using them and using them efficiently. The GPU are not only *massively parallel* machines containing thousands of computing cores but can be also considered as *vector machines*. While the NVIDIA GPU we are using does not have explicit vector instruction, the scalar kernel code is implicitly distributed over thousands of threads, each thread executing the same program. NVIDIA calls this Single Instruction Multiple Threads (SIMT) architecture. Moreover, the smallest execution unit is one *warp* currently containing 32 threads. From the point of view of the programmer all threads in a warp execute in Single Instruction Multiple Data (SIMD) fashion. That is why it can be advantageous to consider NVIDIA GPUs as vector machines with vector length equal to 32 [3]. This is a trend visible also in the CPU market: the SSE extensions introduced four floats wide vector (SIMD) instructions, AVX eight floats wide and *Xeon Phi* 16 floats wide[1]. To get most out of the hardware it became crucial to vectorize the algorithm we want to implement.

In this contribution we present an efficient implementation, both on GPU and CPU, of the Monte-Carlo simulations of the discretized version of the Ginzburg-Landau model. The Ginzburg-Landau model is probably after the Ising model – the second most important model of the statistical physics (for an excellent introduction to those topics [4]). It is the basis for the modern theory of the phase transitions. Under the name of the φ^4 theory it is a model for the scalar field that describes among other things the much publicized Higgs particle. It is a simple model to formulate, but in spite of its simplicity it is unsolvable. While many approximate methods exist to study its properties, Monte-Carlo simulations are used extensively and their importance is growing as the speed and the accessibility of the new hardware increases.

Our motivation was to develop a tool to accompany the lecture on Statistical Field Theory given by one of the authors. The idea was to create a kind of "virtual lab" where students could study the effects of various parameters on the properties of the model. This requires a fast implementation working on the easily accessible commodity hardware, so the results can be obtained in reasonably short time. For the educational reasons we also included a "cut-off" term (see next section) in the model, which, while not strictly necessary, allows for greater clarity of the presentation. This term, however, introduces the interactions between the sites that are not nearest neighbors (see Figure 1) complicating the parallelization/vectorization

[1] Strictly speaking, the *Xeon Phi* is not a CPU but an accelerator. However, its cores are based on *Intel Pentium* CPU micro-architecture.

of the algorithm. We have obtained a hundredfold speedup compared with our old CPU version. That drew our attention to the inefficiency of the CPU code. We have completely rewritten the code taking advantage of the many cores and SIMD instructions of the CPU; paradoxically it took more effort than implementing the GPU version.

We believe, however, that our contribution is relevant to a wider audience. Monte-Carlo simulations of this type play an important role in statistical physics, solid state physics and field theory. Those simulations are usually very time consuming, so the speed of the algorithm is of a paramount importance. Our implementation can be easily adapted to many variants of the model as we describe in more detail later. The GPU implementation is targeted for small desktop machines with CUDA enabled commodity graphics cards and is up to 20 times faster than CPU program running on the same machine[2]. Such "small-scale" computing plays a very important role in science as it provides fast feedback right at the workplace, without resorting to more laborious and/or expensive bigger machines.

Apart from providing ready to use code, which is available online [1], we hope to contribute also to general methodology of adapting Monte-Carlo to SIMD architectures. This clearly is a trend in modern hardware but much of the knowledge remains tacit in nature. We have found several published GPU implementations of similar model; however, they are discrete spin models with nearest neighbor interactions [5, 6, 7]. We use a continuous field model with extended neighborhood that to our knowledge was not yet ported to the GPU. The literature on the SSE/AVX implementation of such models is also scarce. Actually, filling this particular gap is even more important as we often did find that scientists in our surrounding are more knowledgeable in GPU computing than in vectorizing computations on the CPU. This is to some extent due to the publicity that the GPUs are getting recently; but we also believe that NVIDIA CUDA programming environment, and likewise OpenCL, provide a cleaner interface to the parallel/vector features of the processor; this comes of course at the price of greater specialization. The GPUs are inherently parallel machines and provide only parallel mode of programming. The CPUs, on the other hand, have a long history as (super)scalar machines with parallel and vector capabilities put in later on, so it is usual to mix the scalar and vector code in a single CPU program. Additionally, accessing the vector instruction of the CPU in a way that is portable across many compilers is still essentially possible only by using the so called *intrinsics* or inline assembly, that is changing fast with new architectures (Haswell and Xeon Phi) and new compilers; but it also requires the paradigm shift on the part of the users that are used to view the CPU as consisting of many scalar cores and exploiting only the part of the parallelism present in the processor. We hope with this contribution to address some of this issues and increase the explicit knowledge about vectorization of Monte-Carlo simulations.

[2] The speedup is of course highly dependent on the machine configuration used.

2 GINZBURG-LANDAU MODEL

Ginzburg-Landau model in k dimensions that we use is in its discretized form defined by the Hamiltonian [8]:

$$\begin{aligned} H[\varphi] \;=\; \sum_{\vec{i}} \Bigg(& \frac{1}{2}\sum_{\mu=1}^{k}(\varphi(x_{\vec{i}}+\vec{a}_\mu)-\varphi(x_{\vec{i}}))^2 \\ & + \frac{\mu^2}{2}|\varphi(x_{\vec{i}})|^2 + \frac{g}{24}(|\varphi(x_{\vec{i}})|^2)^2 \\ & + \frac{1}{2\Lambda}\left(\sum_{\mu=1}^{k}(\varphi(x_{\vec{i}}+\vec{a}_\mu)-2\varphi(x_{\vec{i}})+\varphi(x_{\vec{i}}-\vec{a}_\mu))\right)^2\Bigg) \end{aligned}$$

where φ is a scalar[3] field defined on the sites of a k dimensional lattice (we only consider $k = 2$ and 3). x_i denotes a site of the lattice where i is a multidimensional index and $\vec{a}_\mu$ denotes the unit vector in the direction μ. We use periodic boundary conditions, i.e.

$$\varphi(x + L_\mu \vec{a}_\mu) = \varphi(x) \tag{1}$$

where L_μ is the number of lattice sites in direction μ.

The Monte-Carlo simulations consist of generating the field configurations with probability proportional to $\exp(-H[\varphi])$. The generation is done by the mean of the Metropolis algorithm (the original idea is presented in [9], but since then it has been explained in many textbooks, see e.g. [4]). This amounts to sequentially updating all the points of the lattice. During each update we change the value of the field φ in a single lattice x_i (φ_i is a shorthand for $\varphi(x_{\vec{i}})$):

$$\varphi_i \to \widetilde{\varphi}_i = \varphi_i + \eta_i. \tag{2}$$

The η_i is a pseudo-random number. In principle the only restriction on η distribution is that it has to be symmetric: the probability of generating η and $-\eta$ must be the same – $P(\eta) = P(-\eta)$. In practice we draw η from the uniform distribution on the interval $(-\varepsilon, \varepsilon)$.

Then we calculate the difference of the Hamiltonian of the new and the old field:

$$\Delta H = H[\varphi|\varphi_i \to \widetilde{\varphi}_i] - H[\varphi]. \tag{3}$$

If $\Delta H < 0$ we accept the change and substitute $\widetilde{\varphi}_i$ for φ_i, otherwise we accept the change with the probability $\exp(-\Delta H)$. In other words if we denote by $P(\phi_i \to \phi_i + \eta_i)$ the probability of updating the field at lattice site x_i, then it is given by

$$P(\varphi_i \to \varphi_i + \eta_i) = \min\left(1, \exp(-\Delta H)\right). \tag{4}$$

[3] In general this can be a vector, but in this contribution we restrict ourselves to scalar case only.

The probability that an update will be accepted depends implicitly on ε. Large ε leads to large values of η and hence large values ΔH leading to small acceptance rates. Conversely, a small ε will produce high acceptance rates but each update will introduce only a small change in configuration, and so more updates are needed to obtain the same effect. In practice one aims at maintaining the acceptance rate of the order of 50 %.

The crucial feature of this algorithm is that the update is local, i.e. ΔH depends only on the values of fields in the immediate neighborhood of the updated point

$$\begin{aligned}\Delta H &= -(\widetilde{\varphi}_i - \varphi_i)\, c_i \\ &\quad + (\widetilde{\varphi}_i^2 - \varphi_i^2)\left(k + \frac{\mu^2}{2} + k(1+2k)\frac{1}{\Lambda}\right) \\ &\quad + \frac{g}{4!}\left(\widetilde{\varphi}_i^4 - \varphi_i^4\right)\end{aligned} \tag{5}$$

where

$$c_i = \left(c_i^{01} - \frac{1}{\Lambda}\left(c_i^{02} - 4k c_i^{01} + 2c_i^{11}\right)\right) \tag{6}$$

and

$$\begin{aligned}c_i^{01} &= \sum_{\mu=1}^{k} \left(\varphi(x_{\vec{i}} + \vec{a}_\mu) + \varphi(x_{\vec{i}} - \vec{a}_\mu)\right) \\ c_i^{02} &= \sum_{\mu=1}^{k} \left(\varphi(x_{\vec{i}} + 2\vec{a}_\mu) + \varphi(x_{\vec{i}} - 2\vec{a}_\mu)\right) \\ c_i^{11} &= \sum_{\mu=}^{k}\sum_{\nu=1}^{\mu-1}\big(\varphi(x_{\vec{i}} + \vec{a}_\mu + \vec{a}_\nu) + \varphi(x_{\vec{i}} - \vec{a}_\mu - \vec{a}_\nu) \\ &\quad + \varphi(x_{\vec{i}} + \vec{a}_\mu - \vec{a}_\nu) + \varphi(x_{\vec{i}} - \vec{a}_\mu + \vec{a}_\nu)\big).\end{aligned} \tag{7}$$

The term c_i is referred to as *corona* of the site x_i. The corona is a weighted sum of the values of the field φ on the sites in the neighborhood of the site x_i. Figure 1 provides a geometrical interpretation of corona in two dimensions. In our case, because of the last term in (1), the corona is extended as compared to more commonly used nearest neighbors neighborhood (squares in Figure 1). As we will show in the next section this considerably complicates the parallelization of the algorithm.

3 GPU IMPLEMENTATION

The Metropolis algorithm lends itself naturally to parallelization, however some caution must be taken. Notably, grid points that lie in the same neighborhood cannot be updated together; that is if we update the red (dark in print) central point in Figure 1 we cannot update any of the gray points in the same time. Basically this

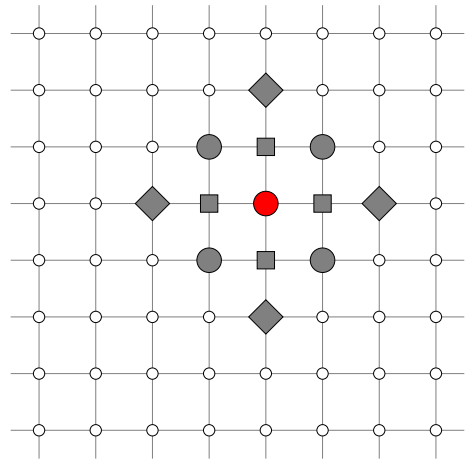

Figure 1. The neighborhood used to calculate the ΔH. The values of the field ϕ on the gray marked lattice sites contribute to the corona c_i with geometrical symbols marking the contribution to different terms: c^{01} – squares, c^{11} – circles and c^{02} – diamonds.

stems from the requirement that in order to process two sites x_a and x_b in parallel those two operations must be independent. That means that the joint probability $P(\phi_a, \phi_b \to \phi_a + \eta_a, \phi_b + \eta_b)$ must factorize:

$$P(\phi_a, \phi_b \to \phi_a + \eta_a, \phi_b + \eta_b) = P(\phi_a \to \phi_a + \eta_a) P(\phi_b \to \phi_b + \eta_b). \tag{8}$$

This can be only achieved if $P(\phi_a \to \phi_a + \eta_a)$ does not depend on the value of ϕ_b and vice versa. To ensure that this condition is satisfied we decompose the lattice into several sub-lattices such that points belonging to each sub-lattice are far enough from each other and can safely be updated in parallel.

In case of the more commonly encountered nearest-neighbor interaction (squares in Figure 1) it is enough to divide the lattice in two sub-lattices by labeling each side "black" or "white" as on the checkerboard (sometimes this is also referred to as "even-odd" decomposition) [7]. It is easy to see that in that case no site has a nearest neighbor of the same color, so all the sites of the same color can be updated in parallel. This would be the case of the Ginzburg-Landau model without the last term in (1).

As we consider in here a model with much bigger corona this simple decomposition is not sufficient any more. We have been unable to find a decomposition requiring less than eight sub-lattices in two dimensions, and 16 in three dimensions. The two-dimensional decomposition is depicted in Figure 2 where we have marked each site (square) with the index of the sub-lattice it belongs to.

Given this decomposition it would be relatively easy to implement the described algorithm in a "naive" way by assigning one sub-lattice point to one thread. Then

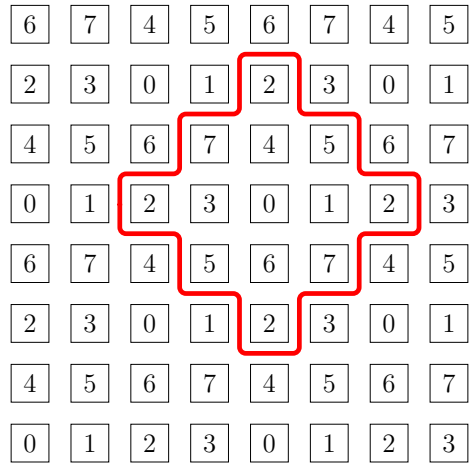

Figure 2. Partition of the lattice into eight independent sub-lattices. Squares with same number on them belong to the same sub-lattice and can be updated in parallel.

each thread would perform the Metropolis update step on that site. However, the GPU is no exception as to the speed of the memory access. While being very fast, the throughput from global memory is still almost two orders of magnitude slower than needed to sustain the maximal instruction throughput. The *NVIDIA* cards we use contain so called *Scalar Multiprocessors* (SM). On Fermi architecture each SM is equipped with up to 48 kB shared memory and 32 kB of 32 bit registers which act as user managed cache [10]. While starting from the compute capability 2.0 *NVIDIA* cards are also equipped with L1 and L2 caches, the use of automatic cache is, as we will show later, not advantageous in terms of performance[4]. The naive algorithm would heavily use the global memory resulting in quite poor performance.

To efficiently use the shared memory we adopt the hierarchical scheme from [6] suitably modified to account for bigger neighborhood. We first divide the whole lattice into chunks of $B_x \times B_y$ points (see Figure 3) or $B_x \times B_y \times B_z$ in three dimensions.

To process each point inside a block we also need the neighboring points – the so called *hallo*. In our case the hallo is two points wide. We divide the grid of chunks into sub-grids in such a way that chunks together with the hallo in each sub-grid do not overlap (see gray points in Figure 3). In 2D we need four of such sub-grids and eight in 3D case.

We then start a kernel that processes all chunks in one sub-grid. Each chunk in this sub-grid is assigned to a block of N^{th} threads (CTA – Cooperative Threads Array). First we fetch the values of the fields from global to shared memory. In 3D we use $B_x = B_y = B_z = 16$ which, including the hallo, gives altogether 20^3

[4] Actually the NVIDIA has renounced the use of the L1 cache for global reads in the Kepler architecture.

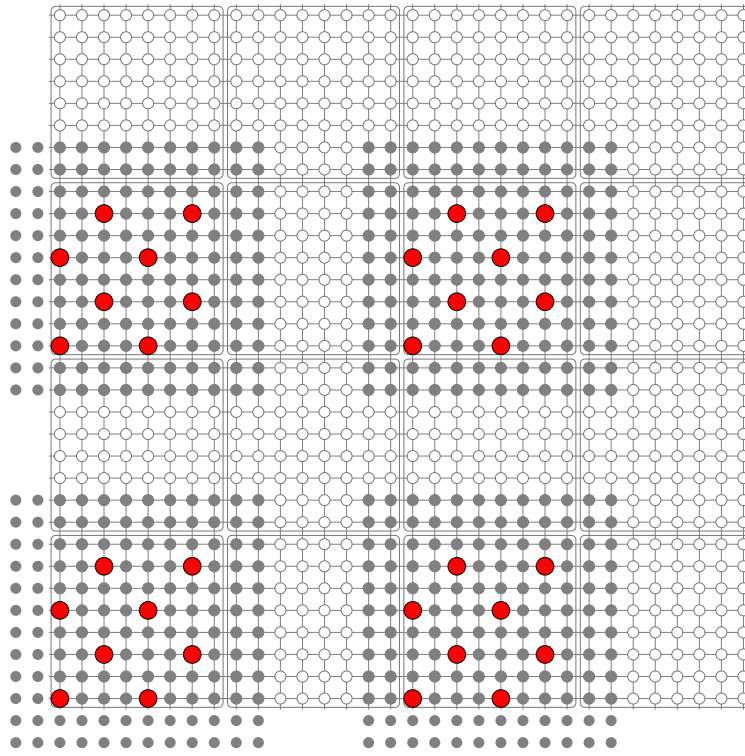

Figure 3. Partition of the lattice into chunks. Each gray chunk is processed by one thread block (Cooperative Thread Array – CTA). The red (dark in print) points belong to one sub-lattice as defined in Figure 2 and each of them is processed in parallel by one thread. The value of the φ field for all points marked in gray is loaded into shared memory. That includes the necessary border points, the so called *hallo*.

field values to be loaded. This gives 32 kB which fits nicely into 48 kB shared memory.

The points in each chunk are decomposed into sub-lattices as described at the beginning of this section (see Figure 2). After that each thread updates one site from the first sub-lattice. This is done N_{hit} times resulting in what is know as *multi-hit* metropolis algorithm. The advantage is that the corona c_i does not need to be recalculated again and most of the needed variables resides already in registers. Then after synchronization, next sub-lattice is updated and so on. This whole procedure is repeated N_{loc} times, again taking advantage of the fact that the field values are already in the shared memory. After that the kernel writes the shared memory back into global one and new kernel is started processing next sub-grid of chunks (see Algorithm 1).

```
1:  for every global sweep do
2:      for every block sub-grid do
3:          for every block in sub-grid do
4:              load block into shared memory
5:              for every local sub-lattice do
6:                  for every site i in local sub-lattice do
7:                      calculate corona c_i
8:                      for k = 1 ... N_hit do
9:                          φ̃_i ← φ_i + η
10:                         ΔH ← H[φ̃_i] − H[φ_i]
11:                         if exp(−ΔH) > r then
12:                             φ_i ← φ̃_i
13:                         end if
14:                     end for
15:                 end for
16:                 __syncthreads()
17:             end for
18:         end for
19:     end for
20: end for
```

Algorithm 1: The GPU algorithm. The CPU algorithm differs by the absence of the loops over the blocks (lines 2–4).

This whole procedure is called a *global sweep* and is repeated $N_{glob.}$ times.

Monte-Carlo calculations require a good source of random numbers. We use the Tausworthe pseudo-random number generator [11] with one copy of the generator per thread. The performance could be increased by the use of simpler generator, but we have not attempted this in this work.

One of the problems in lattice simulations is the treatment of the boundary conditions. The algorithm requires access to the site neighbors during each update. Index of neighbors can be calculated using simple arithmetics, except at the boundaries. Indexing of neighbors requires checking for the boundary at each update or an expensive modulo operation[5]. Another technique is to use the precalculated and stored indices: for every site we store the index of four or six nearest neighbors. This is also very flexible but requires a large amount of memory. This can have a negative impact on the performance and is not well suitable for the GPUs which usually have less RAM than CPUs. In our algorithm the checking for boundary is needed only when loading to shared memory. Then we can use simple index arithmetic.

[5] An exception are the powers of two, where modulo can be calculated cheaply.

4 CPU IMPLEMENTATION

As already mentioned in the beginning, to make a fair comparison of the GPU implementation with the CPU we have to use both multicore and vector (SSE/AVX) programming. The parallelization on many cores is relatively easy using the OpenMP. The same restriction applies on GPU, so we partition the lattice in the same way, but we use only one level, i.e. we do not partition the lattice into chunks. Then the SIMD instructions are used to process four (SSE) or eight (AVX), as we use the single precision updates in parallel (similarly to threads on GPU). At present the only truly portable way to use vector instructions is to use compiler intrinsic [12]. In order to avoid obscurity caused by direct use of intrinsics and to facilitate development, we have implemented a *"drop-in"* scalar type replacement class that wraps vector type declared with `__attribute__((vector_size(N)))` directive and implements missing operations using intrinsics.

Unfortunately, not all scalar *x86* instructions in modern CPUs have vector counterparts, which is a crucial requirement for vectorizing scalar code. In particular, AVX instruction set misses direct vector *XMM/YMM* gather and scatter instructions (load/save vector register data from/to memory indexed by other vector register), unlike GPU where gather and scatter is the key feature. This is partially addressed by AVX2, providing masked gather instruction. Another drawback in lack of 256-bit wide integer operations in AVX, which makes random number generator utilize 128-bit SSE only, even on 256-bit AVX capable CPUs. Again, this has changed in AVX2, which is unfortunately available only in *Haswell* desktop processors.

In our CPU implementation we use a small enhancement exploiting the out of order instruction execution capabilities of the CPU. The innermost loop of the algorithm looks as follows:

```
1: for n = 1 ... N_hit do
2:     calculate ΔH
3:     if exp(−ΔH) > rand() then
4:         accept the move
5:     end if
6: end for
```

The exponent is an expensive, high latency operation. The processor has to wait for the ΔH calculations to finish before it can start the exponent and then it has to wait for exponent to finish.

We have rewritten this part as

```
1: for n = 1 ... N_hit do
2:     lr ← log(rand())
3:     calculate ΔH
4:     if −ΔH > lr then
5:         accept the move
```

6: **end if**
7: **end for**

The advantage is now that the argument of `log` is independent of ΔH and the processor is free to reorder the instructions allowing to hide some of the latency. This implementation is safe even in case when `rand()` returns a zero as `log` then returns `-Inf`. Actually the code above is vectorized and calculations of the four or eight logs is done in parallel using the *Intel*'s SVML library [14].

As already mentioned in the previous section, indexing the neighbors can be quite costly operation requiring expensive modulo operations or large amount of memory. However if we restrict ourselves to the lattice dimensions that are powers of two the nearest neighbor calculations can be made using addition and bitwise operations. This is quite a restrictive approach, not suitable for real applications; but we have decided to use it for comparison with the GPU version. This is in some respect the "upper bound" of the CPU performance.

Porting conditional (branching) instructions to SIMD also deserves some attention. We use the same technique as NVIDIA does implicitly in their SIMT programming model. All branching (conditional jump) instructions and instructions within condition blocks are replaced by masked vector instructions. Fortunately, AVX and AVX2 provide masked instructions for this purpose.

This makes the following sample conditional assignment code working on x, y and z scalar variables

1: **if** $x > y$ **then**
2: $z \leftarrow$ value
3: **end if**

become the code working on $\vec{x}$, $\vec{y}$, $\vec{z}$ and $\vec{mask}$ vector variables

1: $\vec{mask} \leftarrow (\vec{x} > \vec{y})$
2: $\vec{z} \leftarrow (\vec{mask}$ `AND` $\vec{\text{value}})$ `OR` (`NOT` $\vec{mask}$ `AND` $\vec{z}$).[6]

5 PERFORMANCE

We have tested our implementations on the following hardware using single precison floating point arthmetics:

GPU *NVIDIA* GTX 470, 14 *Scalar Multiprocessors* (448 CUDA cores), 1.2 GHz, CUDA 5.0 Ubuntu Linux 12.04 `-O3 -use_fast_math -arch=sm_20`, 1 088 Gflops (single precision) [7].

[6] This step is performed by a single `blend` instruction.

[7] The peak performance it taken from `http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units`

CPU *Intel* i5-2400S *Sandy Bridge*, 4 core, 2.5 GHz, 12 GB RAM, OSX 10.8.3, GCC 4.8.0 `-O3` OpenMP, 160 Gflops (single precision)[8].

The overall performance of the algorithm expressed in Gflops or updates per sec. will depend strongly on the values of the $N_{loc.}$ and N_{hit}. The optimal values of those parameters will depend on the physics of the model, in particular on the values of parameters μ^2, g and Λ. In here we use $N_{loc} = 50$ and $N_{hit} = 8$, which would be a resonable choice near to a phase transition.

To compare the performance to that of our implementations to the peak processor performance we have counted the number of the operations needed in each step of the algorithm; this is presented in Table 1. This was done by inspection of the source code and does not take into account any optimization done by the compiler and assembler. We have counted not only floating points operations but also the integer arithmetic and bitwise logical operators.

	float	int	bit
Coronas	26		
ΔH	13		
Random Number Generator	1	2	21
$\widetilde{\varphi}_i \leftarrow \varphi_i + \eta$	3		

Table 1. The number of instructions in different steps of the algorithm. float – floating point, int – integer arithmetic, bit – shifts and bitwise logical. We do not include the instruction needed for indexing.

To describe the performance in a general way we have decided to fit the execution time T to the following model:

$$T = a + V \cdot b + N_{glob.} V \left(c + N_{loc.} \left(d + N_{hit} \cdot e\right)\right) \tag{9}$$

where V is the volume of the lattice. The interpretation of the parameters of the model together with an estimate of the number of operations performed in the correspondig part of the program is given in Table 2.

The total number of updates is given by

$$N_{up} = V \cdot N_{glob.} \cdot N_{loc.} \cdot N_{hit} \tag{10}$$

so the time per update can be estimated as

$$\frac{T}{N_{up}} = \frac{a + V \cdot b + N_{glob.} V \left(c + N_{loc.} \left(d + N_{hit} \cdot e\right)\right)}{V \cdot N_{glob.} \cdot N_{loc.} \cdot N_{hit}}. \tag{11}$$

[8] Estimated according to [13], synthetic benchmark available at `https://github.com/ujhpc/flops`

Parameter	Meaning of the parameter	n. ops	3D
$a + b \cdot V$	time for setup and closing down		
c	time per spin needed to load it to shared memory and back		
d	time per spin needed to calculate the corona	n_d	26 ops
e	time per spin needed to for update including calculation of two random numbers and exponential	n_e	64 ops + 1 exp

Table 2. Interpretation of the parameters in the model (9). The last column gives the number of numerical operations performed by the 3D versions of the GPU algorithm described in this paper. In case of the CPU the exp is changed to log.

Because we are interested in long simulations with very large value of the N_{glob} parameter the term $a + V \cdot b$ can be neglected in this limit:

$$\frac{T}{N_{up}} \approx \frac{(c + N_{loc.}\,(d + N_{hit} \cdot e))}{N_{loc.} \cdot N_{hit}}. \tag{12}$$

Using the data from Table 2 we obtain that the number of the operations performed is

$$N_{op} = V \cdot N_{glob.} \cdot N_{loc.}(n_d + N_{hit} \cdot n_e) \tag{13}$$

so the performance expressed in operations per second is estimated as

$$ops = \frac{N_{op}}{T} \approx \frac{N_{loc.}(n_d + N_{hit} \cdot n_e)}{(c + N_{loc.}\,(d + N_{hit} \cdot e))}. \tag{14}$$

We have gathered the data on the execution times of the 3D program for four different lattice sizes. The first thing we noticed is that it is not possible to fit all the data simultaneously the model (9). However, when fitting data for each lattice size separately we obtain a very good fit accuracy, better than 1 % on the average. This is understandable: the execution time can depend non-linearly on the lattice volume. One reason is that for small lattice sizes not all SMs on the GPU are used. For the 64^3 lattice we need $64^3/128 = 2\,048$ threads. With 256 threads per CTA we use only 8 thread blocks, which is much less than 14 available SMs. Even for 128^3 lattice the 64 needed blocks will not use the SM evenly. Also, the L2 cache may introduce some non-linear dependence on the lattice size. The results of the fits are included in Table 3.

In case of CPU we do not loop over the blocks and the model is:

$$T = a + V \cdot b + N_{glob.} V\,(d + N_{hit} \cdot e)\,. \tag{15}$$

V	$a + Vb$ [s]	c [ns]	d [ns]	e [ns]
GPU				
64^3	0.56	1.64	0.28	0.18
128^3	0.49	1.06	0.21	0.14
256^3	0.41	0.90	0.19	0.12
512^3	0.86	0.66	0.20	0.11
CPU				
64^3	0.12		12.7	1.51
128^3	1.17		13.4	1.45
256^3	0.93		12.6	1.43
512^3	3.72		13.5	1.44

Table 3. Results of the fit of Equations (9) GPU and (15) CPU

The time per update in this case is given by:

$$\begin{aligned} \frac{T}{N_{up}} &= \frac{a + V \cdot b + N_{glob.} V \left(d + N_{hit} \cdot e\right)}{V \cdot N_{glob.} \cdot N_{hit}} \\ &\approx \frac{d + N_{hit} \cdot e}{N_{hit}} \end{aligned} \tag{16}$$

and

$$ops \approx \frac{n_d + N_{hit} \cdot n_e}{(d + N_{hit} \cdot e)}. \tag{17}$$

Again we have found a strong nonlinear dependence on the lattice volume and we had to fit data for each lattice volume separately. The results are presented in Table 3.

	GPU			CPU			GPU/CPU
Size	Time	Gops	Gexp/s	Time	Gops	Glog/s	Speedup
128^3	0.16	412	6.1	3.1	21.4	0.32	19
256^3	0.15	456	6.8	3.1	21.8	0.32	20
512^3	0.14	490	7.3	3.1	21.6	0.32	22

Table 4. Performance of the algorithm. Time denotes the time of one spin update (in nanoseconds). $N_{loc.} = 50$, $N_{hit} = 8$.

The overall performance of the both implementations is presented in Table 4. We have decided to list the number of exponential and logarithm calculations separately because we use the hardware exponent implementation on the GPU and software library on CPU, so the two are hard to compare.

5.1 GPU Cache

The good performance of the presented GPU algorithm comes from the effective use of the shared memory. We have also checked to which extent this could be achieved

by just using the automatic cache capabilities of modern GPUs. To this end we have implemented a version of the algorithm which did not use the shared memory, however the access pattern was the same. The idea was that the first iteration of the local sweep loop would fill the L1 cache and consecutive iterations would not need to access the global memory and the performance should be comparable to our original implementation.

This turned out not to be the case. The algorithm using the L1 cache was approximately six times slower than the one using shared memory. This deteriorated performance could be traced back to large number of cache misses. Nominally the 20^3 block should fit in L1 cache as it did fit into the shared memory. However, the cache line is 32 words long (128 B) and the blocks are not aligned on the cache lines boundaries (see gray points in Figure 3). On top of that the cache is probably not fully associative. All that together implies that the cache size should be up to four or five times larger to accommodate the whole block.

6 EXTENSIONS AND LIMITATIONS

The above implementation can be easily adapted to other variants of the model. The ΔH can be arbitrary within the constraints that its calculation cannot access points outside of the neighborhood considered in this contribution (see Figure 1). Of course, when the ΔH calculation requires smaller number of lattice points the above procedure can be overly complicated and thus not optimal. In that case it may be advantageous to use e.g. the checkerboard decomposition.

The implementation can be also easily extended to the case when the field φ is a vector, not scalar. In this case the amount of shared memory required to hold the chunk and its hallo grows linearly with the vector dimension. Thus the chunk size has to be reduced accordingly to fit into available 48 kB.

However, one cannot reduce the chunk arbitrarily. In our implementation the smallest chunk size in 3D is 8^3. Cosidering that only one of the 16 sub-lattices is processed simultaneously such a small block requires only 32 threads, rather on the low side leading to degrade performance. So in 3D our algorithm is suitable for scalar or two-dimensional vector model, called also the XY model which is also very interesting (see again [4]).

The size of the lattice that we can simulate is limited by the available RAM on the graphic card. On 1 GB we can fit lattice up to 512^3 which is enough for our application. So far we have not considered a multi-GPU implementation.

7 DISCUSSION

The results presented in Table 4 show clear advantage of the GPU over CPU. This is hardly surprising, but the 20× difference is significantly higher than comparison of theoretical performance ratio of tested i5 CPU to *GTX 470* equal ≈ 7×. The

GTX is thus more efficient in the sense that it is capable of achieving performance much closer to the peak. This can be attributed to several reasons:

- The shared memory that acts as an managed cache allows to use the small cache size much more effectively. In particular, it is possible on the GPU to fit the whole 20^3 grid block into the shared memory. This is not possible using the conventional cache both on GPU and CPU leading to large number of cache misses and decrease of performance.
- Modern CPUs are not fully vector processors. Not every scalar instruction has a vector counterpart. In particular, the eight words wide integer instructions are present in AVX2 instruction set, not available to *Xeon* server CPUs, which has impact on pseudo-random number generator using only 128-bit wide AVX instructions.
- Efficient load and store vector (gather/scatter) instructions are also missing in AVX. When not loading/storing from the consecutive memory location they are effectively serialized into scalar loads/stores. On top of this GPUs provide enormous memory bandwidth comparing to CPU. These, however, are not being accounted in raw *Gops* and *Glogs* calculations above.
- GPUs provide hardware exponent, i.e. our *GTX 470* can execute 4 exponent operations in single cycle of *Scalar Multiprocessor* and recent GPUs can execute 8 exponents in single cycle. CPUs do not provide efficient hardware exponent implementation. Fastest AVX implementations [14] need minimum 20 CPU cycles, however there is no single implementation that works best for all *x86* CPUs [15].
- CPUs have less general purpose registers than GPUs, making CPU code more prone to latency problems caused by tight register dependencies. Peak performance can be reached only by synthetic benchmarks, while real computation algorithms seldom reach half of it.

Undeniably the trend of vectorization of CPUs is continuing and the software that wants to stay competitive must adapt to it. However, the current CPUs make this more difficult than the GPU, due to the lack of some crucial vector features present on GPU, incomplete vectorization support provided by compilers and complicated programming model.

The CPU manufacturers are trying to address these issues. *Intel Xeon Phi* has an extended 512-bit wide vector instruction set with hardware gather/scatter, and recently released *Haswell* CPU architecture with AVX2 instruction set containing 256-bit wide integer operations and gather instructions. However, AVX2 gather is still internally mapped to several μ-ops, yet it provides clear benefits in terms of instruction number reduction having notable impact on performance.

We already have some preliminary results that show 1.5× performance boost making or CPU implementation utilizing some of new AVX2 instructions. We are also working to port our code to recent NVIDIA *Kepler* architecture and *Xeon Phi* accelerator.

REFERENCES

[1] Source code: GPU: `https://github.com/ujhpc/phi4_cuda`, CPU: `https://github.com/ujhpc/phi4`.

[2] `http://www.top500.org/`.

[3] Volkov, V.—Demmel, J. W.: Benchmarking GPUs to Tune Dense Linear Algebra. Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08), IEEE Press, Piscataway, NJ, USA, 2008, pp. 31:1–31:11.

[4] Binney, J. J.—Dowrick, N. J.—Fisher, A. J.—Newman, M. E. J.: The Theory of Critical Phenomena: An Introduction to the Renormalization Group (Oxford Science Publications). Oxford University Press 1992.

[5] Ferrero, E. E.—De Francesco, J. P.—Wolovick, N.—Cannas, S. A.: q-State Potts Model Metastability Study Using Optimized GPU-Based Monte Carlo Algorithms. Computer Physics Communications, Vol. 183, 2012, No. 8, pp. 1578–1587.

[6] Weigel, M.: Performance Potential for Simulating Spin Models on GPU. Journal of Computational Physics, Vol. 231, 2012, No. 8, pp. 3064–3082.

[7] Block, B.—Virnau, P.—Preis, T.: Multi-GPU Accelerated Multi-Spin Monte Carlo Simulations of the 2D Ising Model. Computer Physics Communications, Vol. 181, 2010, No. 9, pp. 1549–1556.

[8] Parisi, G.: Statistical Field Theory. Chapter 5. Perseus Books Publishing 1998.

[9] Metropolis, N.—Rosenbluth, A. W.—Rosenbluth, M. N.—Teller, A. H.—Teller, E.: Equation of State Calculations by Fast Computing Machines. The Journal of Chemical Physics, Vol. 21, 1953, No. 6, pp. 1087–1092.

[10] NVIDIA's Next Generation CUDA ComputeArchitecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[11] Howes, L.—Thomas, D.: Efficient Random Number Generation and Application Using CUDA. In: Nguyen H. (Ed.): GPU Gems 3, Chapter 37. Addison Wesley, 2007.

[12] Intel Intrinsics Guide. Version 2.8.1. `http://software.intel.com/en-us/articles/intel-intrinsics-guide`.

[13] Intel 64 and IA-32 Architectures Software Developer Manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, February 2013.

[14] Overview: Intrinsics for Short Vector Math Library (SVML) Functions. `http://software.intel.com/en-us/node/462496`, February 2013.

[15] Fog, A.: Still No Library That is Optimal on All Processors. `http://www.agner.org/optimize/blog/read.php?i=209&v=t`.

Piotr Białas graduated from Jagiellonian University in mathematics and physics. In 1993 he obtained his Ph. D. in theoretical physics. Currently he is Professor of computer science at Faculty of Physics, Astronomy and Applied Computer Science at the Jagiellonian University. His current scientific interests include high performance computing using graphical processor units and modern processor with SIMD instructions.

Adam Strzelecki graduated from Jagiellonian University in computer science. Currently he is pursuing his Ph. D. in applied computer science at Faculty of Physics, Astronomy and Applied Computer Science at the Jagiellonian University. He is also a freelance iOS and Mac developer, consultant and designer running his own business.

Jakub Kowal graduated from Jagiellonian University in applied computer science. Currently he is pursuing his Ph. D. in applied computer science at Faculty of Physics, Astronomy and Applied Computer Science at the Jagiellonian University.