Acta Univ. Sapientiae, Informatica, **5,** 2 (2013) 184–211

# Yang-Mills lattice on CUDA

Richárd FORSTER
Eötvös University
email: forceuse@inf.elte.hu

Ágnes FÜLÖP
Eötvös University
email: fulop@caesar.elte.hu

**Abstract.** The Yang-Mills fields have an important role in the non-Abelian gauge field theory which describes the properties of the quark-gluon plasma. The real time evolution of the classical fields is given by the equations of motion which are derived from the Hamiltonians to contain the term of the SU(2) gauge field tensor. The dynamics of the classical lattice Yang-Mills equations are studied on a 3 dimensional regular lattice. During the solution of this system we keep the total energy on constant values and it satisfies the Gauss law. The physical quantities are desired to be calculated in the thermodynamic limit. The broadly available computers can handle only a small amount of values, while the GPUs provide enough performance to reach out for higher volumes of lattice vertices which approximates the aforementioned limit.

## 1 Introduction

In particle physics there are many fundamental questions which demand the GPU, from both theoretical and experimental point of view. In the CERN NA61 collaboration one of the most important research field is the quark-gluon plasma's critical point examination. The topics of theoretical physics includes the lattice field theory which is a crossover phase transition in the quark gluon plasma and SU(N) gauge theory with topological lattice action in the QCD.

We present an algorithm which determines the real time dynamics of Yang-Mills fields which uses the parallel capabilities of the CUDA platform. We

**Computing Classification System 1998:** I.1.4
**Mathematics Subject Classification 2010:** 81T25
**Key words and phrases:** lattice gauge field, parallel computing, GPU, CUDA

compare this and the original sequential CPU based program in terms of efficiency and performance.

The real time evolution of these non-Abelian gauge fields is written by the Hamiltonian lattice SU(2) equation of motions [9, 1]. A lattice method was developed to solve these systems on the 3 dimensional space which satisfies Noether theory [2]. This algorithm keeps the Gauss law, the constraint of total energy and the unitary, orthogonality of the suitable link variable. It enables us to study the chaotic behavior as full complex Lyapunov spectrum of SU(2) Yang-Mills fields and the entropy-energy relation utilizing the Kolmogorov-Sinai entropy which was extrapolated to the large size limit by this numerical algorithm [6].

In the parallel algorithm all the links are computed concurrently by assigning a thread to each of them. By taking advantage of the GPUs highly parallel architecture this increases the precision of the calculation by allowing us to utilize more dense lattices. Just by adding a few more points to the lattice, the link count can increase drastically which makes this problem a very parallel friendly application which can utilize the high amount of computational resources available in modern day GPUs [8].

By extending the available CPU based implementation we were able to achieve a 28 times faster runtime compared to the original algorithm. This approach does not involve any special optimization strategies which will impose more performance boost in future releases.

In the original concept the calculations on the GPU were using only single precision floating point numbers to make the evaluations work on older generation cards too, but with the possibility to utilize double precision values, it is possible to achieve higher precision in energy calculation on the GPU as well.

There are more kind of open questions in the high energy physics which can be interesting for the GPU like studying the Yang-Mills-Higgs equations and the monopoles in the lattice QCD. The action function permits to use the thermalization of the quantum field theory in the non-equilibrium states. The solution of these problems requires high calculation power and efficiency.

This paper is constructed as follows. First we review basic definitions dealing with the Yang-Mills fields on lattice, then introducing the basic principles of the GPU programming in CUDA and finally we present numerical results providing comparisons between the CPU and GPU runtimes, extrapolate them for large N values, show the ratio between the sequential and parallel fraction of the algorithm, concluding with the thermodynamic limit of the total energy.

## 2 Homogeneous Yang-Mills fields

The non-Abelian gauge field theory was introduced as generalizing the gauge invariant of electrodynamics to non-Abelian Lie groups which leads to understand the strong interaction of elementary particles. The homogeneous Yang-Mills contains the quadratic part of the gauge field strength tensor [4, 10].

The $F^a_{\mu\nu}$ forms the component of an antisymmetric gauge field tensor in the Minkowski space, it is expressed by gauge fields $A^a_\mu$:

$$F^a_{\mu\nu} = \partial_\mu A^a_\nu - \partial_\nu A^a_\mu + g f^{abc} A^b_\mu A^c_\nu, \tag{1}$$

where $\mu, \nu = 0, 1, 2, 3$ are space-time coordinates, the symmetry generators are labeled by $a, b, c = 1, 2, 3$, $g$ is the bare gauge coupling constant and $f^{abc}$ is the structure constant of the continuous Lie group. The generators of the Lie group fulfills the following relationship $[T^b, T^c] = i f^{bcd} T^d$.

The equation of motion can be expressed by covariant derivative in the adjoin representation:

$$\partial^\mu F^a_{\mu\nu} + g f^{abc} A^{b\mu} F^c_{\mu\nu} = 0. \tag{2}$$

We use Hamiltonian approach to investigate the real time dynamics of these systems SU(2). The lattice regularization of such theories were studied numerically.

## 3 Lattice Yang-Mills theory

The real time coupled differential equations of motion are solved by numerical method for basic variables which form matrix-valued link in the 3 dimensional lattice with lattice elementary size $a$ (Figure 1). These are group elements which are related to the Yang-Mills potential $A^c_i$:

$$U_{x,i} = \exp(a A^c_i(x) T^c), \quad \text{where} \quad T^c \quad \text{is a group generator.}$$

For SU(2) symmetry group these links are given by the Pauli matrices $\tau$, where $T^c = -(ig/2)\tau^c$. The indices $x, i$ denote the link of the lattice which starts at the 3 dimensional position $x$ and pointing into the nearest neighbor in direction $i$, $x + i$.

In this article we study the Hamiltonian lattice which can be written as a sum over single link contribution [1, 3]:

$$H = \sum_{x,i} \left[ \frac{1}{2} \langle \dot{U}_{x,i}, \dot{U}_{x,i} \rangle + \left( 1 - \frac{1}{4} \langle U_{x,i}, V_{x,i} \rangle \right) \right], \tag{3}$$

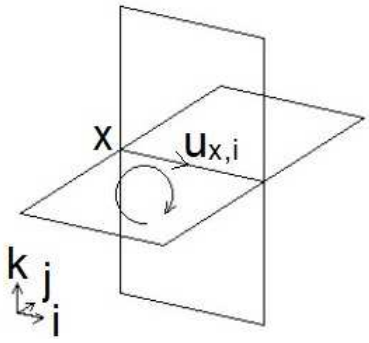


Figure 1: Wilson loop

where $\langle A, B\rangle = tr(AB^\dagger)$ and $V_{x,i}$ are complement link variables, these are constructed by products of $U_{x,i}$-s along all link triples which close with given link $(x, i)$ an elementary plaquette. The canonical variable is $P_{x,i} = \dot{U}_{x,i}$ and a dot means the time derivative.

The non-Abelian gauge field strength can be expressed by the oriented plaquette i.e. product of four links on an elementary box with corners $(x, x + i, x + i + j, x + j)$:

$$U_{x,ij} = U_{x,i}U_{x+i,j}U_{x+i+j,-i}U_{x+j,-j},$$

where $U_{x,-i} = U^\dagger_{x-i,i}$.

Then the local magnetic field strength $B^c_{x,k}$ is related to the plaquette:

$$U_{x,ij} = \exp(\epsilon_{ijk}a^2B^c_{x,k}T^c), \tag{4}$$

where $\epsilon_{ijk}$ is +1 or -1 if $i, j, k$ is an even or odd permutation of 1,2,3 and vanishes otherwise. The electric field $\mathcal{E}^c_{x,i}$ is given in the continuum limit:

$$\mathcal{E}^c_{x,i} = \frac{2}{ag^2}tr(T^c\dot{U}_{x,i}U^\dagger_{x,i}). \tag{5}$$

We use the SU(2) matrices which can be expressed by the quaternion representation $(u_0, u_1, u_2, u_3)$ for one link, where the components $u_i$ $i = 0, \ldots, 3$ are real numbers, it can be written:

$$\begin{aligned} U &= u_0 + i\tau^a u^a \\ U &= \begin{pmatrix} u_0 + iu_3, iu_1 + u_2 \\ iu_1 - u_2, u_0 - iu_3 \end{pmatrix}. \end{aligned} \tag{6}$$

The determinant of the quaternion is:

$$\det U = u_0^2 + u_1^2 + u_2^2 + u_3^2 = 1.$$

The length of the quaternion $\det U = \|U\|$ is conserved, because $\dot{u}_0 u_0 + \dot{u}_a u_a = 0$. The three electric fields $E^a_{x,i}$ which are updated on each link by the next form:

$$\dot{\mathcal{E}}^a_{x,i} = \frac{i}{ag^2} \sum_j \mathrm{tr}\left[\frac{1}{2}\tau^a (U_{x,ij} - U^\dagger_{x,ij})\right], \tag{7}$$

where the value of $j$ runs over four plaquettes which are attached to the link $(x, i)$.

The time evolution of the electric fields constraints the Gauss law:

$$D^{ab}_i \mathcal{E}^b_{x,i} = 0. \tag{8}$$

This means charge conservation.

The Hamiltonian equations of motion are derived from expression (3) by canonical method and these can be solved with $dt$ discrete time steps. The algorithm satisfies the constraints of total energy and the Gauss law which is detailed in the next Section 3.1. Update of link variables is derived from the following implicit recursion forms of the lattice equation of motions:

$$\begin{aligned} U_{t+1} - U_{t-1} &= 2dt(P_t - \epsilon U_t), \\ P_{t+1} - P_{t-1} &= 2dt(V(U_t) - \mu U_t + \epsilon P_t), \end{aligned} \tag{9}$$

$$\epsilon = \frac{\langle U_t, P_t\rangle}{\langle U_t, U_t\rangle}, \qquad \mu = \frac{\langle V(U_t), U_t\rangle + \langle P_t P_t\rangle}{\langle U_t, U_t\rangle}, \tag{10}$$

where $\varepsilon, \mu$ are the Lagrange multipliers and the symmetry SU(N) fulfills the unitarity $\langle U_t, U_t\rangle = 1$ and the orthogonality $\langle U_t, P_t\rangle = 0$ conditions.

### 3.1 Constraint values

This algorithm fulfills the constraints of the system's total energy and the Gauss law.

The total energy $E_{tot}$ is determined by the sum over each link of lattice for every time steps. The energy is defined for a single link at the time step $t$:

$$E_l = \frac{1}{2} < P, P > + 1 - \frac{1}{4} < U, V >, \tag{11}$$

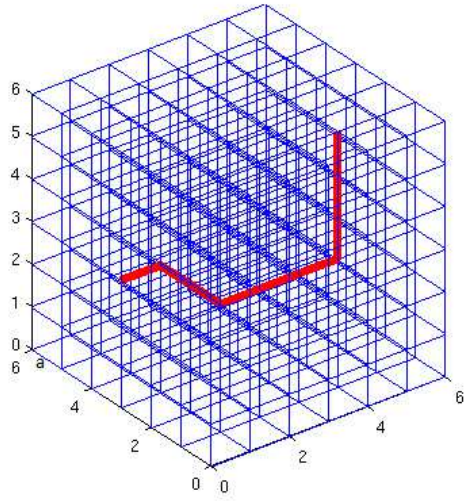


Figure 2: Flux line on the three dimensional lattice, where $a$ means the size of the box

where $U = U_t$, $V = V_t$ and $P = P_t$. The value $E_{tot}$ does not change during the time evolution. This is a constraint to use the Noether theorem [2]. The Gauss law is a constraint quantity:

$$\Gamma = \sum_{l^+} PU^\dagger - \sum_{l^-} U^\dagger P = 0, \tag{12}$$

where the sum is performed over links $l^+$ which is started at the box on the lattice and the $l^-$ means the links to end at that side of the grid. This is conserved by the Hamiltonian equations of motion:

$$\dot{\Gamma} = \sum_{l^+} VU^\dagger - \sum_{l^-} U^\dagger V = 0. \tag{13}$$

Corresponding to the quantum electrodynamics' law the charge and flux line initialization (Figure 2) occur with the following recursion expressions on the lattice:

$$\begin{aligned} P_1 &= QU_1, \\ P_n &= U_{n-1}^\dagger P_{n-1} U_n \quad (1 \le n \le N), \end{aligned}$$

where the starting value of charge is $Q$ and the end of this quantity equals to $-F^{\dagger}QF$. Flux line ordered product is written by the expression

$$F = \prod_{i=1}^{N-1} U_i. \tag{14}$$

The condition of neutrality is expressed by these equations:

$$\left.\begin{array}{rl} Q - F^{\dagger}QF & = 0 \\ \mathrm{tr}Q & = 0 \end{array}\right\} \Rightarrow Q = \frac{q}{2}(F^{\dagger} - F).$$

In the next section we discuss the connection between the Hamiltonian expression and Wilson action, because this plays an important role in the strong interaction.

### 3.2 Relation between Wilson action and Hamiltonian lattice

The Wilson action should be summed over all elementary squares of the lattice $S = \sum_{p_{x,i,j}} S_{p_{x,i,j}}$. The action function of the gauge theory on lattice is written over plaquette sum to use the nearest neighbour pairs. Because in the continuous time limit the lattice spacing $a_t$ becomes different for the time direction, therefore the time-like plaquettes has other shape than the space-like ones. Therefore the coupling on space-like and time-like plaquettes are no longer equal in the action:

$$S = \frac{2}{g^2}\sum_{p_t}(N - \mathrm{tr}(U_{p_t})) - \frac{2}{g^2}\sum_{p_s}(N - \mathrm{tr}(U_{p_s})). \tag{15}$$

The time like plaquette is denoted by $U_{p_t}$ and the space like is $U_{p_s}$.

Consider the path is a closed contour i.e. Wilson loop (Figure 1), where the trace of the group element corresponding to such a contour which is invariant under gauge changes and independent of the starting point. The product of such group elements along connected lines is a gauge covariant quantity, the trace over such products along a closed path is invariant. This lattice system is very suitable for describing gauge theories. Because the $U_{p_t}$ can be series expansion by $a_t$:

$$U_{p_t} = U(t)U^{\dagger}(t + a_t) = UU^{\dagger} + a_t U\dot{U}^{\dagger} + \frac{a_t^2}{2}U\ddot{U}^{\dagger} + \ldots$$

$$N - \mathrm{tr}(U_{p_t}) = -\frac{a_t^2}{2}\mathrm{tr}(U\ddot{U}^{\dagger}) \qquad \text{up to } O(a_t^3) \text{ correction.}$$

We investigate the unitarity of the expression $UU^{\dagger} = 1$ at the beginning of Section 3, therefore this implies the following:

$$U\dot{U} + U\dot{U}^{\dagger} = 0 \quad \text{and} \quad \ddot{U}U^{\dagger} + 2\dot{U}\dot{U}^{\dagger} + U\ddot{U}^{\dagger} = 0.$$

The homogeneous non-Abelian gauge action can be written in the next form:

$$\Delta S_H = \frac{2}{g^2}\left(\frac{a_t^2}{2}\sum_i \mathrm{tr}(\dot{U}_i\dot{U}_i^{\dagger}) - \sum_{ij}(N - \mathrm{tr}(U_{ij}))\right). \tag{16}$$

The first sum is over all links and the second one goes over space-like plaquettes. The scaled Hamiltonian was derived in the next form. General discretized ansatz can be written as:

$$S = a_t \sum_t a_s^3 \sum_s L, \tag{17}$$

than the scaled Hamiltonian becomes:

$$a_t H = \frac{2}{g^2}\left(\frac{a_t^2}{2}\sum_i \mathrm{tr}(\dot{U}_i\dot{U}_i^{\dagger}) + \sum_{ij}(N - \mathrm{tr}(U_{ij}))\right). \tag{18}$$

In the next section we derived the algorithm in the explicit form.

## 4 Lattice field algorithm

In this section we introduce the numerical solving of the coupling differential equations of motion by CPU [2]. The initial condition is uniformly random in the SU(2) group space to fulfil the constraints unitarity, orthogonality and Gauss law. The update algorithm satisfies the periodic boundary.

### 4.1 Implicit-explicit-endpoint algorithm

First we determine the forms $\mu$ and $c$ corresponding to orthogonality and unitarity conditions which were introduced in Section 3. We denote:

$$P' = P_{t+1} \qquad P = P_t.$$

The equations of motion (9) are written:

$$P' = P + (V - \mu U + \varepsilon P'), \tag{19}$$

$$U' = U + (P' - \varepsilon U). \tag{20}$$

The Lagrange multiplicators $\mu, \varepsilon$ are given in the next form to satisfy the symmetry SU(2):

$$\begin{aligned} (1-\varepsilon)P' &= P + (V - \mu U), \\ U' &= (1-\varepsilon)U + P', \end{aligned}$$

where $c = 1 - \varepsilon$. First we obtain the value $\mu$ from orthogonality:

$$\begin{aligned} c\langle U', P'\rangle &= \langle cU, P\rangle + \langle P', P\rangle + c\langle U, V - \mu U\rangle + \langle P', V - \mu U\rangle, \\ 0 &= 0 + c(\langle U, V\rangle - \mu) + c\langle P', P'\rangle, \\ \mu &= \langle U, V\rangle + \langle P', P'\rangle. \end{aligned}$$

On the next step we obtain $c$ from unitarity:

$$\begin{aligned} \langle U', U'\rangle &= c\langle U', U\rangle + \langle U', P'\rangle, \\ 1 = c\langle U', U\rangle &= c(\langle cU, U\rangle + \langle P', U\rangle), \\ 1 &= c^2 + c\langle P', U\rangle, \end{aligned}$$

$$c\langle U, P'\rangle = \langle U, P\rangle + \langle U, V - \mu U\rangle = \langle U, V\rangle - \mu = -\langle P', P'\rangle,$$

$$1 = c^2 - \langle P', P'\rangle \Rightarrow c = \sqrt{1 + \langle P', P'\rangle}.$$

$(c > 1, \quad \varepsilon < 0)$.

In the next section we express the explicit and implicit form of the algorithm.

### 4.1.1 Algorithm

The method is written in implicit form. The final expressions of these equations of motion are the following:

$$\begin{aligned} V^\dagger &= V - \langle U, V\rangle U, \\ \tilde{P} &= P + V^\dagger, \\ cP' &= \tilde{P} - (c^2 - 1)U, \\ U' &= cU + \tilde{P}. \end{aligned}$$

The resolution of implicit recursion is:

$$\begin{aligned} c(P' + U') &= \tilde{P} + c^2 U + (1 - c^2)U + cP', \\ cU' &= \tilde{P} + U, \end{aligned}$$

but $U' = cU + P'$, so

$$\begin{aligned} P' &= U' - cU = \frac{1}{c}(\tilde{P} + U) - cU, \\ cP' &= \tilde{P} + (1 - c^2)U. \end{aligned}$$

The length of $cP'$ becomes:

$$\begin{aligned} c^2\langle P', P'\rangle &= \langle \tilde{P}, \tilde{P}\rangle + 2(1-c^2)\langle \tilde{P}, U\rangle + (1-c^2)^2\langle U, U\rangle, \\ c^2(c^2-1) &= \langle \tilde{P}, \tilde{P}\rangle + (1-c^2)^2, \\ (c^4 - c^2) - (c^4 - 2c^2 + 1) &= \langle \tilde{P}, \tilde{P}\rangle, \\ c^2 - 1 &= \langle \tilde{P}, \tilde{P}\rangle, \end{aligned}$$

$\Rightarrow \quad c = \sqrt{1 + \langle \tilde{P}, \tilde{P}\rangle}.$

Finally the algorithm explicitly:

$$\begin{aligned} V^\dagger &= V - \langle U, V\rangle U, \\ \tilde{P} &= P + V^\dagger, \\ c &= \sqrt{1 + \langle \tilde{P}, \tilde{P}\rangle}, \\ P' &= \frac{1}{c}(\tilde{P} + U) - cU, \\ U' &= cU + P'. \end{aligned}$$

This algorithm was applied on GPU in Section 6. These processes are compared with the original sequential method on the CPU against the parallel version on the GPU.

## 5 Compute unified device architecture

In the last decade the performance increase of the central processing units have slowed down drastically compared to a decade earlier. At the same time the graphical processing units are showing a very intense evolution booth in performance and architecture thanks to their origin from the graphical computations and thanks to the never-ending need for more computational power (values on Figure 3 were taken from [12]).

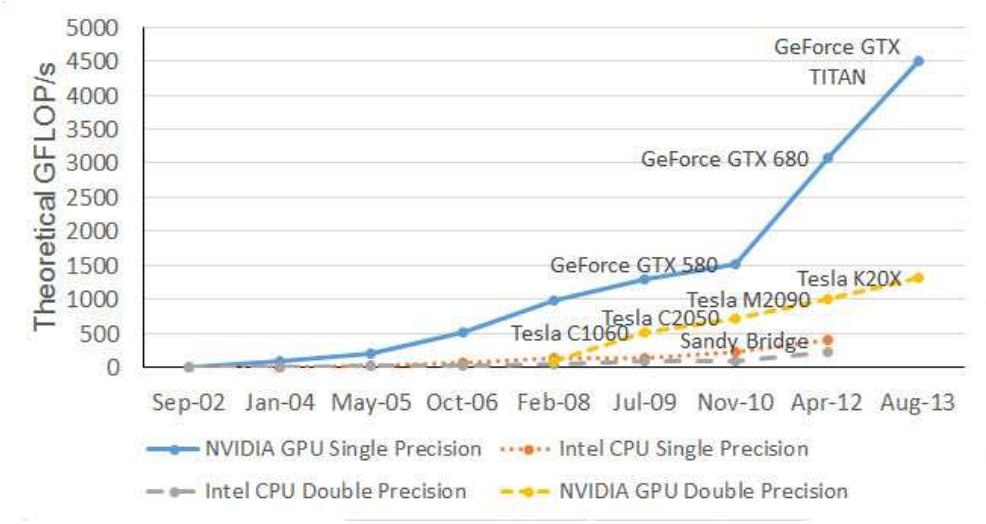


Figure 3: Performance increase of the CPU and the GPU

Our idea is to process the Yang-Mills model's high volume data on the GPU. This way we can use bigger lattices for calculation, achieving higher precision and faster runtime. With the many core architecture through the CUDA it is now possible to evaluate the actual status of the lattice by checking the individual link values in parallel.

### 5.1 The compute unified device architecture

Thanks to the modern GPUs now we can process efficiently very big datasets in parallel [7]. This is supported by the underlying hardware architecture that now allows us to create general purpose algorithms running on the graphical hardware. There is no need to introduce any middle abstractions to be able to use these processors as they have evolved into a more general processing unit (Figure 4 [14]). The compute unified device architecture (CUDA) divides the GPUs into smaller logical parts to have a deeper understanding of the platform. [12] With the current device architecture the GPUs are working like coprocessors in the system, the instructions are issued through the CPU. In the earlier generations we were forced to use the device side memory as the GPUs were not capable to accept direct memory pointers. If we wanted to

utilize the GPUs, then all the data were supposed to be copied over to the device prior the calculations.

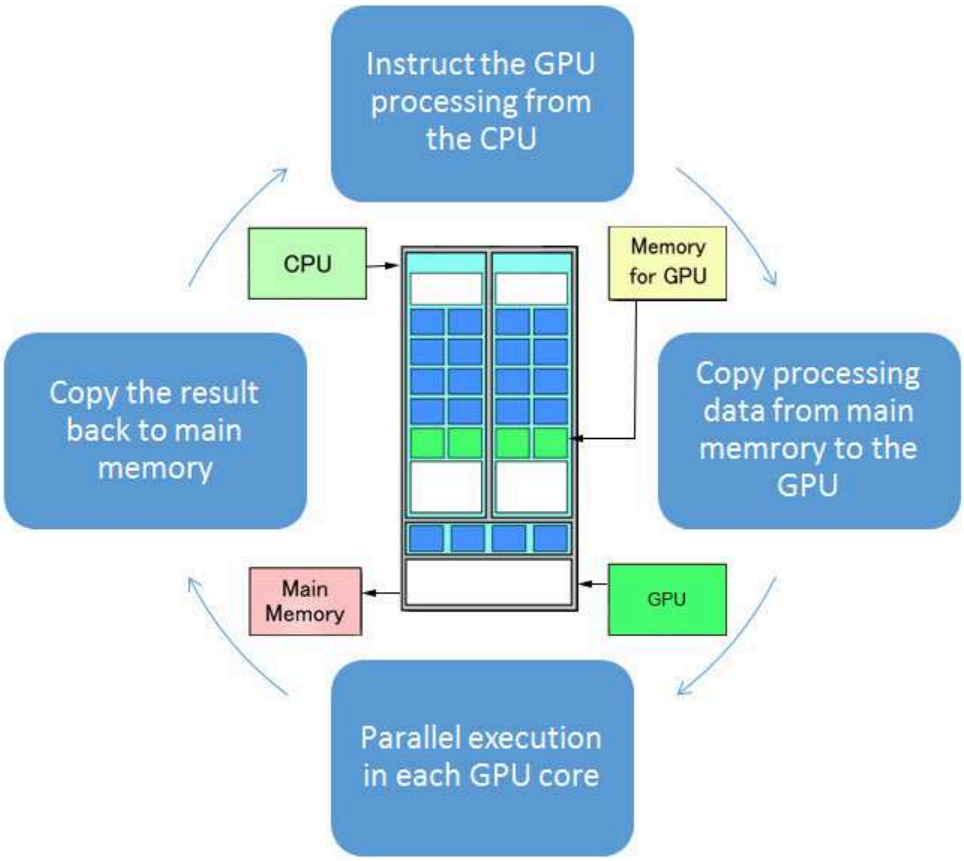


Figure 4: CUDA processing flow

While this is still the main idea behind our parallel calculations as the second generation of Compute Capability devices were released it has became possible to issue direct memory transactions thanks to the Unified Virtual Addressing [17]. This has made it possible to use pointers and memory allocations not only on the host side, but on the device as well. In earlier generations it the thread local, shared and global memories have used different address spaces, which made it impossible to use C/C++ like pointers on the device as the value of those pointers were unknown at compile time.

### 5.1.1 Thread hierarchy

CUDA is very similar to the C language, the biggest difference in its syntax is the <<< and >>> brackets which are used for kernel launches [8]. Kernels are special C like functions with void return value, that will create all the threads and that will run on the GPU. It also has the usual parameter list which contains all the variables we want to pass our input through and to receive the results of our computations. It is important, that for such inputs and outputs the memory has to be allocated on the GPU prior the kernel call. All of our threads are started in a grid which consists of many blocks which will contain the threads (Figure 5).

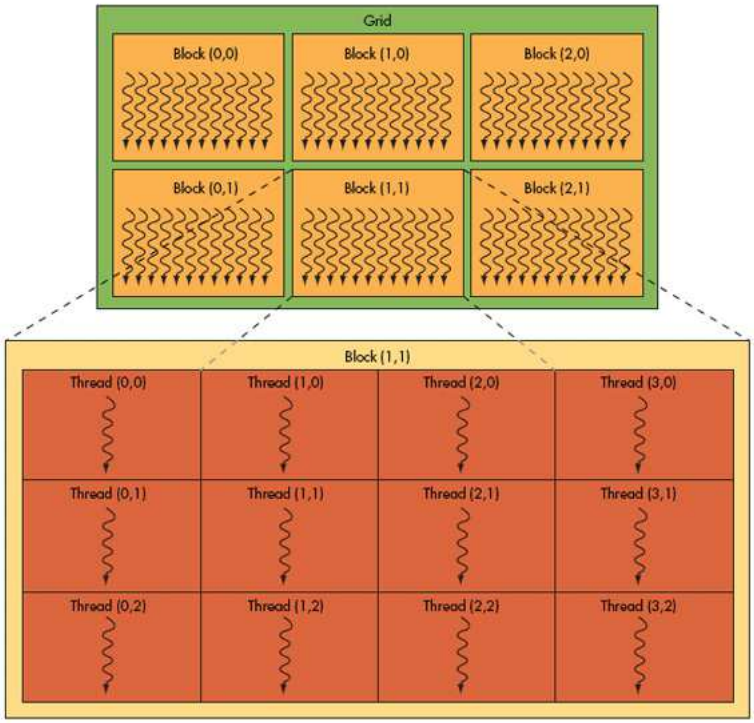


Figure 5: CUDA thread hierarchy

The maximum number of threads that we can start depends on the actual compute capability of the GPU, but it is important that this number does not equal to the actual threads being processed at the same time on the GPU. The

size of the grid and the size of the block depends on the compute capability of the hardware, but looking solely on the capability restraints we cannot show the maximum threads being processed.

A GPU can have different number of Streaming Multiprocessors (SM) and different amount of memory. The CUDA restrictions are containing occupancy restrictions as well. There are three kinds of these restrictions: resource limitations, block, and thread availability. The SMs are having the maximum limit on the number of maximum online blocks. Performance wise it is not a good practice to create algorithms that will be able to reach the resource constraints even with a very low amount of threads i.e. with high register utilization per thread [8].

We should divide our algorithm to be called by different kernels thus decreasing the resource dependency of our code. The biggest impact on the performance is the memory utilization. It is the most important aspect to keep all of our midterm results on the device and to keep the host-device memory transactions to the minimum [11]. The data must be coalesced in the memory to provide the maximum possible throughput. At the same time the registers and the shared memory are faster by at least a factor of 100 than the global memory. This is because the global memory is on the card, while the shared memory and the registers are on the chip.

### 5.1.2 Memory access strategies

For older generation of GPUs with Compute Capability 1.x it is important to use the right access patterns on the global memory. If we will try to use the same value from thousands of threads, then the access will have to be serialized on these GPUs, while the newer ones have caching functionality to help on such scenarios. The most optimal access is the map access, where all threads will manipulate their own values, more specifically thread $n$ will access the $n$th position of the input or output array.

If the data stored in the memory can be accessed in a sequential order and it is aligned to a multitude of 128 byte address then the data fetching will be the most optimal on the devices with Compute Capability 1.x (Figure 1). The GPU can issue 32, 64 or 128 bytes transactions based on the utilization of the hardware.

If the data is in a non-sequential order (Figure 2), then additional memory transactions will be required to process everything. We mention here, by using non-sequential ordering it is possible the transactions will fetch more data, than we are really going to use at the time. This can be quite a wasteful

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.x and 3.0 |
|---|---|---|---|
| Memory transactions: | Uncached | | Cached |
| | 1 x 64B at 128<br>1 x 64B at 192 | 1 x 64B at 128<br>1 x 64B at 192 | 1 x 128B at 128 |

Table 1: Aligned and sequential memory access

approach.

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.x and 3.0 |
|---|---|---|---|
| Memory transactions: | Uncached | | Cached |
| | 8 x 32B at 128<br>8 x 32B at 160<br>8 x 32B at 192<br>8 x 32B at 224 | 1 x 64B at 128<br>1 x 64B at 192 | 1 x 128B at 128 |

Table 2: Aligned and non-sequential memory access

If the data is misaligned (Figure 3), then it will invoke more transactions as smaller ones will be required to fetch everything. This case can be problematic even on the cached devices. All tables are representing the data taken from [12].

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.x and 3.0 |
|---|---|---|---|
| Memory transactions: | Uncached | | Cached |
| | 7 x 32B at 128<br>8 x 32B at 160<br>8 x 32B at 192<br>8 x 32B at 224<br>1 x 32B at 256 | 1 x 128B at 128<br>1 x 64B at 192<br>1 x 32B 256 | 1 x 128B at 128<br>1 x 128B at 256 |

Table 3: Misaligned and sequential memory access

Overall it is important to design our data structures to be able to accommodate them to the aforementioned restrictions to be able to achieve the maximum possible memory throughput.

### 5.1.3 Other architectures and models

Our development and research was conducted on the aforementioned CUDA platform. Other architectures and programming models are available that provide high parallel performance. Currently the biggest competition for the NVIDIA GPUs are the AMD Radeon GPUs. But the biggest competition is in the discrete GPU market. In the HPC segment the NVIDIA GPUs are the mainstream solutions [18], when there is any GPU utilization in the supercomputers. For example the fastest GPU cluster based supercomputer, the TITAN incorporates NVIDIA Tesla K20X GPUs as coprocessors.

On the other hand Intel is developing it's own coprocessor for the HPC segment, the Intel Xeon Phi processor. Currently the fastest supercomputer in the TOP500 is the Tianhe-2 (MilkyWay-2) accelerated with these Xeon Phi processors, while the previously mentioned TITAN is at the second place [18]. The Linpack performance benchmark shows a $33,862.7$ TFlop/s capability for the Tianhe-2, while it shows a $17,590.0$ TFlop/s for TITAN. But if we take a look at the number of the processor cores, the Tianhe-2 uses $3,120,000$ cores, while the TITAN uses only $560,640$ cores. It is difficult to make a direct interpolation for the achieved performance in the case if we will double the cores in the TITAN, so we will take a look at the individual performance of each computers accelerator core.

The Intel Xeon Phi 3100 series of accelerators have 57 x86 cores and are capable of 1003 GFlop/s performance in double precision calculations while drawing 300 Watt of power [15]. On the other hand the NVIDIA Tesla K20X has 2688 CUDA cores, capable of 1310 GFlop/s performance also in double precision calculations while drawing only 235 Watt of power [16]. If we take the total core numbers of the supercomputers, then the Tianhe-2 has 48000 Xeon Phi processors [5], while the TITAN has 18688 Tesla K20X processors [13]. Theoretically if we double the number of K20X cards, we will have more performance than the Tianhe-2, while still utilizing less GPU, than how many MICs they are using. This shows that the Kepler GPU architecture based Tesla accelerators are more efficient than the Knights Corner MIC architecture based Xeon Phi accelerators.

Booth architectures support the OpenCL, OpenAAC programming languages which all are GPU based languages. On the NVIDIA GPUs the CUDA model is the most important as the GPUs are in connection with the programming model and as they develop the GPUs and provide new functionality, the same functionality becomes supported in the next CUDA version. This way they have the freedom to let developers utilize their GPUs how they see it the most efficient.

### 5.2 Single instruction multiple thread architecture

Our current CPUs are SIMD processors, where SIMD stands for Single Instruction Multiple Data. This implies that the multicore CPUs can evaluate a given instruction for multiple data which can achieve even higher amount of instructions per cycle with the Intel Hyper Threading Technology. In our case the test machine CPU has two physical cores, that can run up to four threads simultaneously thanks to the aforementioned technology. So in this case we will have four instructions evaluated concurrently. On the other hand the GPUs are SIMT architecture based processors. This stands for Single Instruction Multiple Thread which is very similar to the SIMD architecture. The biggest difference is in the maximum number of threads. Theoretically we are not bound by the maximum number of threads that we can launch, as the latest Kepler GPUs can initialize billions of threads at the same time. The key factor is that the performance per thread is quite low, but the GPU can handle thousands of those cores in a single clock cycle. Of course the actual number of running threads will be lower, but still bigger then what we can have on the CPUs. The basic idea behind the SIMT architecture is that we summon as many threads as many data we have for evaluation. This implies that for higher utilization we need to provide more data to work on. But even with maximum thread occupancy it doesn't directly mean we will achieve the maximum computational performance.

### 5.3 Computational architecture

In this subsection we will see what kind of technical details the GPUs have and how it affects the actual utilization of the given architecture. The actual number of threads running on the GPU comes through the term of warps. A warp is a set of 32 threads in a given Streaming Multiprocessor. Based on the actual architecture and compute capability the maximum number of warps per SM can differ (Table 4), but the size of a warp is constant 32. This means that in an optimal solution the maximum number threads running at the same time is:

$$\#\mathsf{SM} * \#\mathsf{WARP} * 32.$$

This implies that all $n$ threads are running the same instruction at the same time. But in a not so optimal scenario it is possible that the threads are diverging from the size of the warp. This means that the execution flow differs among the different threads, so it will not be possible to evaluate all the 32 threads in the warp, because they are using the same program counter. In

this case the scheduler will have to take into account that there are slower warps in which the threads are serialized, and this will decrease the overall performance. This can happen if a thread has to evaluate if statement branches or long cycles. In the case of cycles the compiler can make some optimization as it will unroll the loops, but there is no way to predict the flow among the if statements.

| | Compute Capability | | | | | | |
|---|---|---|---|---|---|---|---|
| Technical Specifications | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.5 |
| Maximum dimensionality of grid of thread blocks | 2 | | | | 3 | | |
| Maximum x-dimension of a grid of thread blocks | 65535 | | | | | $2^{31}-1$ | |
| Maximum y- or z-dimension of a grid of thread blocks | 65535 | | | | | | |
| Maximum dimensionality of thread block | 3 | | | | | | |
| Maximum x- or y-dimension of a block | 512 | | | | 1024 | | |
| Maximum z-dimension of a block | 64 | | | | | | |
| Maximum number of threads per block | 512 | | | | 1024 | | |
| Warp size | 32 | | | | | | |
| Maximum number of resident blocks per multiprocessor | 8 | | | | | 16 | |
| Maximum number of resident warps per multiprocessor | 24 | | 32 | | 48 | 64 | |
| Maximum number of resident threads per multiprocessor | 768 | | 1024 | | 1536 | 2048 | |

Table 4: Compute capability technical specifications (for description see Section 5.1.1)

# 6 Parallel Yang-Mills algorithm

In our computational problem we are calculating newer and newer states of all the links in the system. This is a great example to use the map access pattern as all threads will have a link to work on. The algorithm was implemented on the CUDA platform.

## 6.1 Main idea

In every given timestep a kernel generates the new values for the links. The kernel call itself is in a cycle that will move until a given step count. The real difficulty arise from the sanity checks of the system. After a given timestep the algorithm checks if the system is still valid and such if any further developments are possible. This will require the actual results on the GPU to be checked if are still valid. To copy them back to the host side is just a waste of memory bandwidth and time. To check it on the device requires to have parallel algorithms for the subsequent operations. To check the systems validity we have to summarize the energy values of the links, thus we need to make a parallel summation. Thankfully the NVIDIA Thrust library already has this algorithm implemented, so we have used this approach. For this to work we have to give the values to the algorithm in the form of Thrust defined vectors. It is possible to cast raw memory pointers to vector containers, so there is no incompatibility between the Thrust defined vectors and the user defined global memory allocations. The result will be only one value which will be much easier to be transferred to the host side for evaluation and this will be done only once in every check. This isn't necessarily a good practice, because to achieve the highest memory throughput we should copy high amount of data instead of little fragments, but currently we only have to copy just these summation, so its really just one value per iteration, without implying any throughput problems.

## 6.2 Restrictions

The current implementation provides a direct port of the original Yang-Mills algorithm. As such, it does not provide any GPU specific optimizations which should further improve the already high amount of performance gain over the original CPU based version. The calculations are running on the three dimensional space with varying amount of precision, or varying dense of the lattice. The more dense it is, the more links it will generate, thus heavily increasing the inputs and the required calculations. Because this is a direct

port we do not separate the algorithm based on different functionalities, just applying the same algorithm for all the links in parallel. This implies that for the actual implementation the biggest restrain the available memory is. In this sense how dense an actual lattice can be for processing depends on how much free memory we have on the GPU, as all the links will have to be stored on the device.

## 6.3 Implementation

For implementation and testing we have used a GeForce GTX 580 with compute capability 2.0.

| | GeForce GTX 580 |
|---|---|
| Technical Specifications | Compute Capability 2.0 |
| Transistors (Million) | 3000 |
| Memory (MB) | 1536 |
| SM Count | 16 |
| Memory Bandwidth (GB/s) | 192.4 |
| GFLOPs | 1581.1 |
| TDP (watts) | 244 |

Table 5: GeForce GTX 580 technical specifications

In our case the maximum number of executed threads is 24576 (Table 5). This means that there will be this amount of instructions which evaluated at every given clock cycle in parallel. To be really efficient though it is important to do not introduce diverging threads. In our case the Yang-Mills algorithm doesn't provide any instructions that will result in diverging threads. The links of 3 dimensional lattice are aligned into an array which are distributed into a 1 dimensional grid. We compute a state of lattice through a grid of CUDA threads by giving a link to a thread for computation. As a new state is reached we use the Thrust reduction algorithm to do the required summation on the new values to check the actual properties while keeping the whole dataset on the GPU.

### 6.3.1 Compute unified device architecture based algorithm

Essentially there is no real difference between the original (see Section 4) and the CUDA based algorithm, the equations (19), (20) are the same after all.

But powerful distinction between the two is the indexing of the given links $U_{x,i}$ (Section 4.1.1) which are stored in a row ordered array.

The index of an actual thread can be calculated with the next statement, assuming that we are using a one dimensional grid (Section 5.1.1) with one dimensional blocks.

```
int idx = blockDim.x*blockIdx.x+threadIdx.x;
```

A simple optimization is the including of shared memory (Section 5.1.1).

We are accessing the links many times during an evaluation, so it is a good optimization strategy to load the links into the shared memory.

```
__shared__ float s_aux[1000];
__shared__ float s_U[1000];
__shared__ float s_V[1000];
```

The most compute heavy parts of the algorithm are the subroutines to upgrade the links and to calculate the complement of the lattice variable. As an implication of this the GPU based algorithm contains the accelerated versions of the aforementioned functions.

These functions are CUDA kernels so they are required to be __global__ functions. We are starting these kernels with 512 threads for each thread block with as many blocks as much we need to have the same amount of threads as much links we have.

Naturally this can give us more threads than the number of the available links, so a condition check is given to do not utilize the unnecessary threads.

This way we can evaluate our algorithm (Section 4.1.1) on all the maximum allowed threads at the same (Section 6.3), alas on the same amount of links.

## 6.4 Numerical results

We introduced a method to solve the Yang-Mills equations in time by expressions (19), (20) (see Section 4). The link variables were expressed by quaternion representation and due to the SU(2) symmetry three dimensional polar coordinate system was applied.

We numerically computed the differential equation of motion by real-time implicit-explicit algorithm (Section 4.1) to choose random initial configurations on any finite lattice SU(2). This process fulfils the constraints of total energy $E_{tot}$ by Lagrange multiplicators, unitarity and orthogonality of the

SU(2) symmetry conditions and Gauss law. We applied periodic boundary conditions and the nearest neighbor intersection on finite lattice.

The physical quantities required the high precision calculation and the size of elementary lattice box expected the smallest value as possible i.e. to achieve the extrapolation of the thermodynamic limit. An efficient algorithm was achieved on the GPU by parallelism in Section 6. This process is much more effective on the three dimensional lattice.

For overall testing the following system was used (Table 6):

| CPU | GPU | OS | Compiler | CUDA version |
|---|---|---|---|---|
| Intel Core i5 650 | GeForce GTX 580 | Windows 8 Pro | Visual C++ 2010 | 5.0 |

Table 6: The used system's specification

We compared the runtime of the CPU to the GPU (Figure 6), the GPU gives substantially better results considering the same lattice size which shows acceleration of a magnitude of 28 in single precision and 11 in double precision.

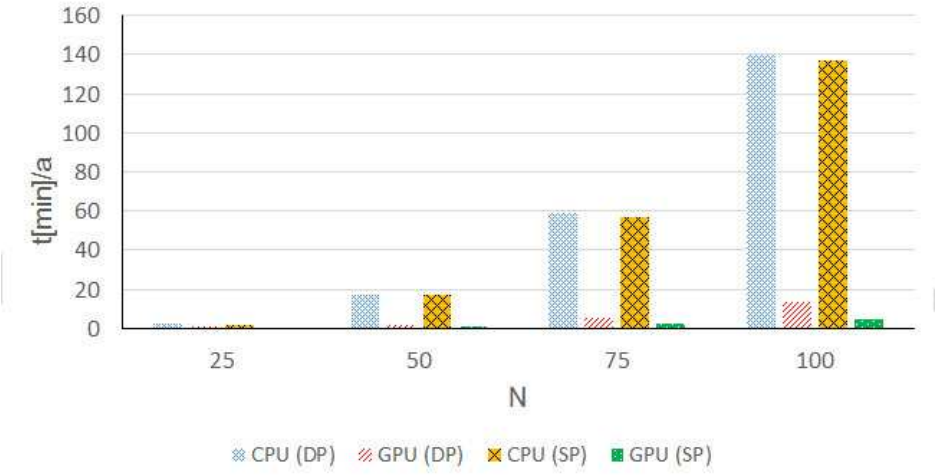


Figure 6: Runtime on the CPU and on the GPU with $N = 25, 50, 75, 100$

Even if we use single precision values for our calculations, the CPU cannot

provide any strong performance compared to the GPU because the latter has a lot more processing power.

Due to the limited resources of the CPU it is really difficult to provide a real comparison, thus we provide an extrapolation of the higher dense lattice computation runtime (Figure 7). By measuring how much time a lattice with $N^3$ vertices takes to be evaluated, we can see how much time a single link takes. Taking this into account we calculated the number of links on the three dimensional lattice and multiplied this number with that single link runtime as it follows:

$$\mathrm{const}_1 = t_1/(24N_1^3); \quad \text{extrapolated runtime} = \mathrm{const}_1 24N^3,$$

where $t_1$ is the runtime of a lattice with $N_1 = 25$ and $24N^3$ is the number of all links in a lattice with $N^3$ vertices. This value was calculated for booth single and double precision driven CPU and GPU based runtimes.

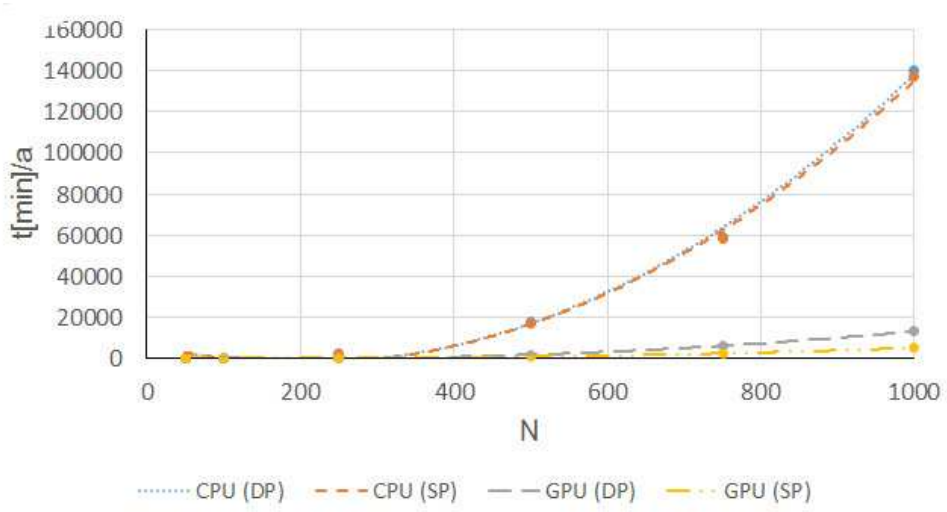


Figure 7: Extrapolated runtime of the algorithm for large N on the CPU and extrapolated runtime on the GPU. Measured range: 0 <= N <= 100; Extrapolated range: 100 < N <= 1000

From (Figure 7) it can be read that the GPU based calculations will remain faster compared to the CPU implementation even on much denser lattices. The actual measured range of the lattice size is 0 <= N <= 100 and the further

extrapolated range is $100 < N <= 1000$. (Figure 7) shows that the factor 27.68 by which the GPU is faster than the CPU on the actually calculated lattices is still valid on the extrapolated interval where this factor is 27.69. Naturally the single precision values should imply a faster runtime and the used hardware provides a significantly higher single precision peak performance, than what it gives for the double precision. Still the single precision based implementation isn't much faster than it's double precision counterpart. The reason for that is that the algorithm at hand cannot utilize the hardware efficiently.

We are mentioned it many times, that the problem at hand is very parallelization. This means that the algorithm that we are using has a very good sequential part to parallel part ratio which implies the parallelization nature of the problem. The values on (Figure 8) shows that as we increase the size of our lattice this ratio starts to grow, but very steadily. This is because the sequential part is very limited compared to the parallel part which already has a huge amount of acceleration over the original algorithm, where:

Let $t_{seq}/a$ denote the runtime of the sequential portion of the code and $t_{par}/a$ denote the runtime of the parallel portion booth values in seconds.

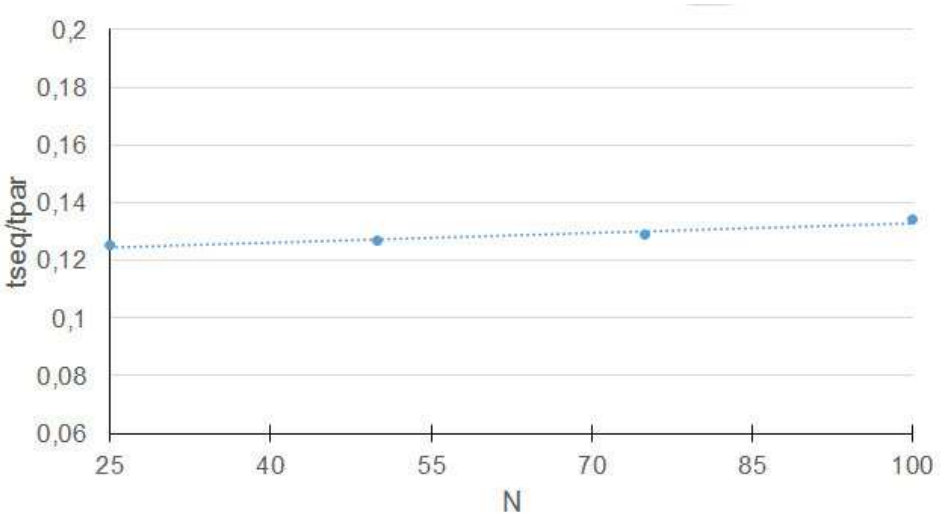


Figure 8: Ratio of the sequential and parallel runtime in the function of N, where tseq[sec]/a is the sequential fraction of the runtime and tpar[sec]/a is the parallel fraction

The fraction of the two $t_{seq}/t_{par}$ is small, because the dominant factor is the parallel part as it will take more time to be evaluated, than the lesser sequential part. The parallel routines are handling the massive datasets, updating all the link variables in the lattice, while the sequential parts are calculating the Langrange multiplicators which results in the conservation of the total energy, the unitarity and orthogonality of the SU(2) symmetry condition in this dynamical system.

The single precision calculations are considerably faster than the double precision evaluations, so it is an important question if the single precision numbers are sufficient for our needs or not. The single precision values suffers a little loss thanks to the half precision, but the two values are still equal up to the fifth decimal value, above that the deviation of the energy only exists because of the higher precision of the double values (Figure 9).

Let $E_d$ denote the energy based on the double precision values of energy and $E_s$ denote the energy based on the single precision values, and $(E_d - E_s)g^2a$ is drawn in the function of $t/a$, where runtime measured in seconds.

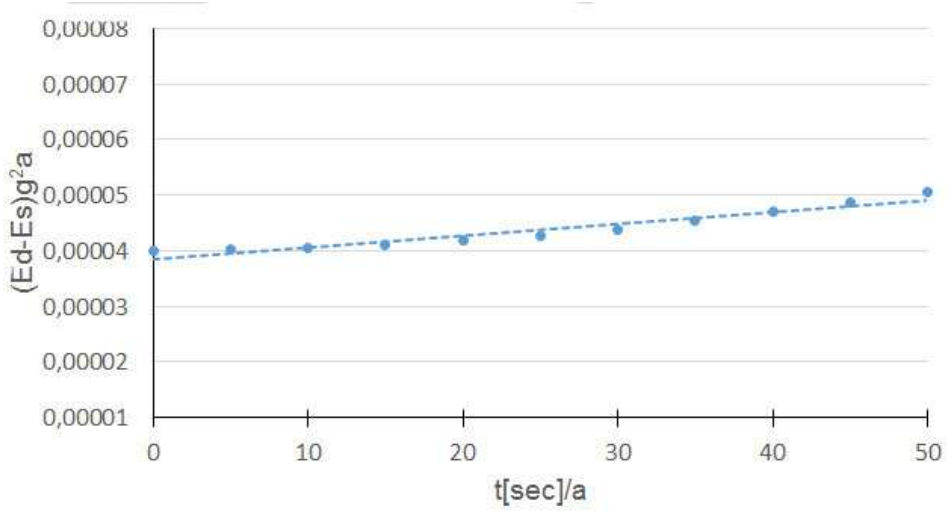


Figure 9: The energy difference in the function of the time i.e. $(E_d - E_s)g^2a$ vs $t[sec]/a$ for single precision ($E_s$) and double precision ($E_s$) values on the GPU

The fundamental value of certain physical quantities can be determined by finite-size scaling. We determined the extrapolation of the energy to the

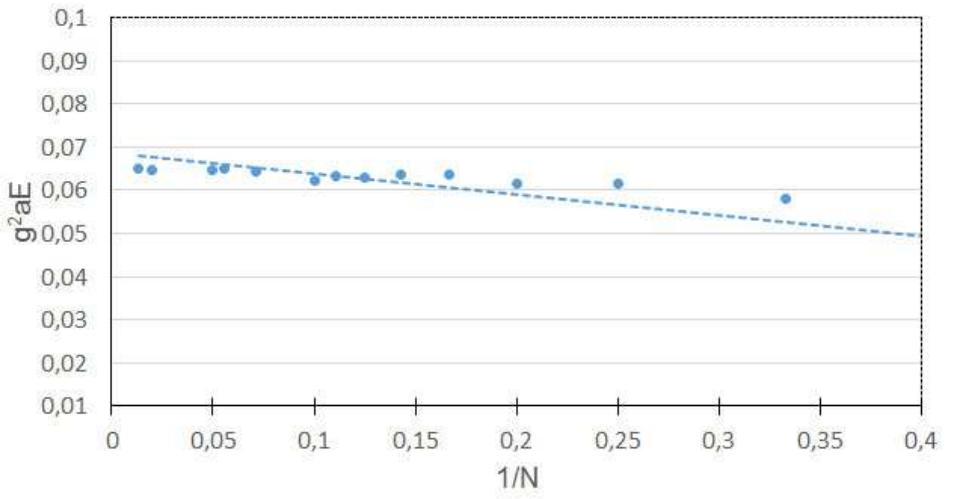


Figure 10: The thermodynamic limit of the energy ($N \to \infty$)

thermodynamic limit ($N \to \infty$) which is demonstrated on (Figure 10). The correspondence proved to be almost linear by assuming $g^2 aE \sim \frac{1}{N}$ scaling with finite-size.

In this chapter through numerical calculations we have proved that the GPU is a valuable computing platform even for the Yang-Mills algorithm, providing faster real-time performance than what the CPU has, allowing us to reach higher precision without sacrificing too much time.

## 7 Summary

We studied the time dependent behaviour of Yang-Mills fields which are expressed by coupled differential equations.

As the Fermi GPU architecture was build up from the ground to be compatible with the C++ programming standards it have became simple to port the existing applications to the GPU. In our case the direct port Yang-Mills model's algorithm was capable to achieve at least 11 times performance boost compared to the original.

Just changing the underlying hardware the algorithms still produced the

same result, thus keeping the physical principles valid. Physical constant quantities remains constraint while solving the equation of motion by parallel algorithm, such as the total energy.

The behaviour of the GPU makes it possible to solve the more complicated systems for example Yang-Mills-Higgs fields. This means more precision can be achieved on these systems. This is especially important where high precision computations in thermodynamic limit are mandatory.

## References


[1] T. S. Biró, C. Gong, B. Müller, A. Trayanov, Hamiltonian dynamics of Yang-Mills fields on a lattice, *Int. J. of Modern Phys. C* **5** (1994) 113–149. ⇒185, 186

[2] T. S. Biró, Conserving algorithms for real-time non-Abelian lattice gauge theories, *Int. J. of Modern Phys. C* **6** (1995) 327–344. ⇒185, 189, 191

[3] T. S. Biró, A. Fülöp, C. Gong, S. Matinyan, B. Müller, A. Trajanov, Chaotic dynamics in classical lattice field theories, *165th WE-Heraeus Seminar on Theory of Spin Lattices and Lattice Gauge Models 14-19 Oct 1996. Bad Honnef, Germany Lect. Notes in Physics* **494** (1997) 164–176. ⇒186

[4] M. Creutz, *Quarks, Gluons and Lattices*, Cambridge University Press, Cambridge CB2 1RP, 1983. ⇒186

[5] J. Dongarra, *Visit to the National University for Defense Technology Changsha, China*, University of Tennessee ⇒199

[6] A. Fülöp, T. S. Biró, Towards the equation of state of a classical SU(2) lattice gauge theory, *Phys. Rev. C* **64** (2001) 064902(5). ⇒185

[7] A. Iványi (ed.), *Algorithms of Informatics, Volume 3*, AnTonCom Infokommunikációs Kft. 2011, ⇒194

[8] D. B. Kirk, W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publisher, Burlington, MA, 2012. ⇒185, 196, 197

[9] B. Müller, A. Trayanov, Deterministic chaos on non-Abelian lattice gauge theory, *Phys. Rev. Letters* **68,** 23 (1992) 3387–3390. ⇒185

[10] I. Montvay, G. Münster, *Quantum Fields on a Lattice*, Cambridge University Press, Cambridge CB2 1RP, 1994. ⇒186

[11] J. Sanders, E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 2010, NVIDIA Corporation, ⇒197

[12] *CUDA C Programming Guide*, NVIDIA Corp., 2013, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. ⇒193, 194, 198

[13] *TITAN Supercomputer*, http://www.olcf.ornl.gov/titan/ ⇒199

[14] *Tosaka, CUDA*, 2008 http://en.wikipedia.org/wiki/CUDA ⇒194

[15] *Intel Xeon Phi 3100 Series Specification*, `http://www.cpu-world.com/CPUs/Xeon_Phi/Intel-XeonPhi3120P.html` ⇒199

[16] *NVIDIA Tesla K20X Specification*, http://www.nvidia.com/object/tesla-servers.html ⇒199
[17] *Whitepaper NVIDIA's Next Generation Compute Architecture: Fermi*, NVIDIA Corp., 2009, `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf` ⇒195
[18] Official list of the top 500 supercomputers, http://top500.org ⇒199

*Received: July 11, 2013 • Revised: November 25, 2013*