A GPU-Accelerated Worm algorithm

Ydalia Delgado
Karl-Franzens University Graz
Report - HPC Seminar
Graz, February 2011

Abstract

I discuss the implementation on GPUs of a worm algorithm for a classical spin system. I show that although the worm algorithm is not well suited to parallelization, it is possible to achieve a very good improvement on the performance using CUDA.
1 Introduction

To make calculations of Quantum Chromodynamics (QCD) at non-vanishing quark density is a very challenging task because the weight factors used as probabilities in the Monte Carlo simulations become complex when hadron density is taken into account. Therefore new algorithmic concepts are necessary. In order to test these new ideas we start with a simplified model of QCD [1], not only to be sure that these new methods work but also because these effective theories are much easier to handle with. We are interested to use the model where QCD is reduced to a classical spin system and study its phase transition as a function of the temperature (as in the Ising model) and chemical potential.

In order to find the phase transition line we perform the simulations using the worm algorithm [3] for being the most suitable algorithm to deal with non-vanishing quark density. We have to explore the parameter space and generate millions of worms at every parameter point, which takes a lot of computation time. This problem gets worse when we go to larger lattices and we use more complicated models. Therefore, I want to use CUDA [5] and the compute capability of the GPUs to improve the performance of the simulation.

I have applied CUDA in two different ways:

• Using CURAND to speed up the generation of random numbers, while the complete simulation is performed on the host.

• Running the complete algorithm on the device.

Here I will show the implementation, the results and the future work.

1.1 Effective Model of QCD

This model has been studied by [2], where QCD is reduced to a system formed by a 3d lattice with a classical spin, that can take 3 different states, sitting at every site of the lattice. The gluonic part is represented by the spin-spin interaction term, where the strength of the interaction (parameter $\tau$) is an increasing function of the temperature and the quarks are represented by the term that corresponds to an external magnetic field multiplied by the chemical potential. For the sake of simplicity of the CUDA implementation, I will not consider the external magnetic field term.
The model is described by the Hamiltonian:

\[ H[P] = -\sum_n \tau \sum_{\nu=1}^3 \left[ s(n)s(n+\hat{\nu})^* + \text{c.c.} \right] \tag{1} \]

where \( \sum_n \) runs over all spatial sites \( n \). The spin variables \( s(n) \in \{1, e^{2\pi i/3}, e^{-i2\pi/3}\} \).

And the coupling parameter \( \tau \) is real and non-negative.

The partition function of the model is given by:

\[ Z = \sum_{\{s\}} e^{-H[s]} \tag{2} \]

### 1.1.1 Flux representation

To use the worm algorithm we map the theory to a set of new variables known as flux representation[4]. For setting the notation, I briefly present the derivation of this representation.

For the neighbor interaction term of (1) we use the ansatz:

\[ e^{\tau[s(n)s(n+\hat{\nu})^*+\text{c.c.}]} = C \sum_{b_{n,\nu}=-1}^{+1} B|b_{n,\nu}| \left( s(n)s(n+\hat{\nu})^* \right)^{b_{n,\nu}} \tag{3} \]

In this expression the term living on the link \( (n, \nu) \) is written as a sum over dimer variables \( b_{n,\nu} \in \{-1, 0, +1\} \). And \( C \) and \( B \) are real and positive functions of the parameter \( \tau \).

The final form of the partition function is, after summing out over the spins:

\[ Z \propto \sum_{\{b\}} \prod_{n,\nu} B|b_{n,\nu}| \prod_{n} T \left( \sum_{\nu} |b_{n,\nu} - b_{n-\hat{\nu},\nu}| \right) \tag{4} \]

\( T(m) \) is equal to 1 if \( m \) is a multiple of 3 and vanishes otherwise. This constraint reduces the allowed configurations to those where the total sum of dimers at every site is a multiple of 3.

### 1.2 Worm Algorithm

The worm algorithm is a Monte Carlo simulation of closed-path configurations of dimers from the flux representation, called worms.

- Advantages:
  - Most suitable algorithm.
– Small autocorrelation time in critical regions.

• Disadvantages:

– The creation of each worm is a serial process. Thus, we will not be able to explode the complete compute capability of the GPUs.

The algorithm uses 3 different steps to produce admissible configurations, illustrated in Fig.1. The worm starts at a random position (1). Then, it may decide to insert dimer fluxes (positions 2) until the worm closes (3).

![Figure 1: Illustration of the worm algorithm.](image)

For each parameter we need to generate some worms and then perform a “measurement”, that consists on counting the total number of non-zero dimers on the lattice. The CPU implementation has the following scheme:

```cpp
int main() {
    for (i = 1; i = n_T; i++) {
        do n_equi worms; // create n_equi equilibrate the system
        for (imeas=0; imeas < n_meas; imeas++) {
            do n_skip worms // before each measurement skip n_skip worms
            for decorrelation
                measure();
        }
    }
}

void doOneWorm() {
    do {
        add_dimer();
    }
}
```
} while (head != tail);
}

2 Realization by CUDA

As mentioned in the introduction, I have tried two different implementations with CUDA. Here, I explain them in more detail.

2.1 Generation of random numbers on the device:

The most straightforward, but not the best, application of CUDA is to generated all the random numbers on the device and copy them to the host for the simulation. Fig.2 shows the performance of RANLUX [6], the random number generator (RNG) used in the original simulations. CURAND [7] on the host and on the device. I generated $10^6$ random numbers in blocks of $N$ numbers. We can observe that in the case of RANLUX, the time needed for the generation remains almost constant along $N$. On the other hand, CURAND becomes faster for blocks larger than 4000 numbers. As expected, CURAND on the device is even faster than the host.

![Figure 2: Comparison of the different RNGs.](image)

Figure 2: Comparison of the different RNGs.
2.1.1 Performance Results

Then, we compared the performance of the worm algorithm (∼ 41K worms for $8^3$ and $16^3$ volumes) with the different RNGs (double precision). Table 1 shows the results for RANLUX, CURAND on the host, and CURAND on the device (using a GTX-280). In the generation of the worms I used blocks of 50 thousand numbers.

<table>
<thead>
<tr>
<th>V</th>
<th>Ranlux-host</th>
<th>Curand-host</th>
<th>Curand-GTX280</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8^3$</td>
<td>1</td>
<td>0.15</td>
<td>0.078</td>
</tr>
<tr>
<td>$16^3$</td>
<td>7.9</td>
<td>0.83</td>
<td>0.578</td>
</tr>
</tbody>
</table>

Table 1: Performance summary. The CPU results are for one core of an Intel i7. And the GPU results for one GPU of a GTX-280. All simulations on double precision.

2.2 Worm Algorithm on the device

The second strategy was to implement the complete algorithm on the device. First of all, we check if it is possible to allocate all the variables on the shared memory of the device.

- 6 numbers per each site:
  - 3 dimer variables, which take only 3 possible values (2 bits per dimer is enough)
  - 3 flags, to indicate if a link is occupied by a worm (only 1 bit is enough).

- Only 1 integer per site is enough to store all variables (I call it “lattice variable” from now on). Then, the total amount of memory needed to allocate one lattice variable is equal the volume of the lattice times the size of an integer, this fits only in global memory.

The structure of the new variable is shown in Tab.2, the size of each box is of 4 bits. $d_i(n)$ is the dimer variable on the direction $i$ at the site $n$ (it can be 0, 1 or 2). And $f_i(n)$ is the flag of the dimer $d_i(n)$ (1 if the link is empty and 0 if a worm is there).

Then, we proceed to “parallelize” the creation of the worms.
Table 2: Fitting all dimers and flags variables into one integer number.

2.2.1 First parallelization

One lattice variable is allocated in global memory. Then we set the parameter $\tau$ to the same value for all threads and blocks. And create many worms, as many threads can run simultaneously, in the same volume.

```c
int main() {
    for (i = 1; i = n_\tau; i + +) {
        kernel_worm<<nb_eq, nt_eq >>> (); // n_equi = nb_eq \times nt_eq
        for (imeas = 0; imeas < n_meas; imeas++) {
            kernel_worm<<nb_sk, nt_sk >>> (); // n_skip = nb_sk \times nt_sk
            kernel_measure<<nb, nt >>> ();
        }
    }
}
```

I would like to remark that this kind of parallelization is not possible with MPI due to the simultaneous access to the lattice variable by all threads.

2.2.2 Second parallelization

It is possible to make a further parallelization in the program, running different values of $\tau$ at the same time. First we allocate as many lattice variables as the number of $\tau$s we want to simulate ($n_\tau \leq$ number of blocks). Then, depending on $n_\tau$, we assign a fix number of blocks to create the worms for each lattice.

The program has the new structure:

```c
int main() {
    kernel_worm<<nb_\tau, nt_eq >>> (); // n_equi = nt_eq and n_\tau = num. of blocks
    for (imeas = 0; imeas < n_meas; imeas++) {
        kernel_worm<<nb_\tau, nt_sk >>> (); // n_skip = nt_sk and n_\tau = num. of blocks
        kernel_measure<<nb_\tau, nt >>> ();
    }
}
```
2.2.3 One problem remains

If we observe the pseudo-code of the part of the algorithm that creates one worm, there is a do-while loop that cannot be replaced. Therefore, all threads of one warp will have to wait until the last worm is finished (due to the fact that data is processed in a SIMD fashion by a CUDA multiprocessor).

\[
\text{do} \\
\quad \text{add dimer();} \\
\text{while (head } \neq \text{ tail );}
\]

After counting the number of iterations for each worms at different parameters, we observed that most of the worms are rather small (1-6 iterations are enough to be finished). While there are a few of them which need more than $10^5$ iterations. Thus, part of the threads will idle most of the time. So far, I have no ideas how to solve this problem, maybe one has to rethink the algorithm.

2.2.4 Performance Results

Tab.3 shows the comparison of the “first parallelization” with the host implementation (RANLUX and CURAND). In all cases I used single precision (to save computation time). $41K$ worms were generated on a $64^3$ lattice at a large $\tau$ in order to increase the probability to create longer worms. The GPU simulations were done on a Fermi (I tried also a GTX-280 but it is between 2 – 3 times slower).

<table>
<thead>
<tr>
<th>Ranlux-host</th>
<th>Curand-host</th>
<th>Device-GTX480</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.205</td>
<td>0.0305</td>
<td>0.0305</td>
</tr>
</tbody>
</table>

Table 3: Performance summary of the worm algorithm on the device. The CPU results are for one core of an Intel i7-4790. And the GPU results for one GPU of a GTX-480. All simulations on single precision.

The “second parallelization” case is still under checks. However, we carried out some tests and observed that it is even faster than the “first parallelization”.
3 Conclusions

We have studied an effective theory of QCD at non-zero temperature. Mapping this theory to the flux representation enables us to generate configurations efficiently in a wide range of parameters using a worm algorithm. In order to reduce the computation time I have implemented the algorithm using CUDA. Although the worm algorithm is not suitable for parallelization, we have seen that it is possible to speedup the code up to 35 times.

I will continue with the implementation of the complete algorithm, including the magnetic term of the Hamiltonian. Finally, my goal is to implement on GPUs the worm algorithm of the new models we want to study (which are more complicated and need more computation time).

4 Acknowledgments

I thank prof. Gundolf Haase for very fruitful discussions.

References