Journal of Computational Physics 231 (2012) 1209–1215

Contents lists available at SciVerse ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

GPU-based single-cluster algorithm for the simulation of the Ising model

Yukihiro Komura *, Yutaka Okabe

Department of Physics, Tokyo Metropolitan University, Hachioji, Tokyo 192-0397, Japan

ARTICLE INFO

Article history:
Received 18 August 2010
Received in revised form 30 September 2011
Accepted 30 September 2011
Available online 10 October 2011

Keywords:
Monte Carlo simulation
Cluster algorithm
Ising model
Parallel computing
GPU

ABSTRACT

We present the GPU calculation with the common unified device architecture (CUDA) for the Wolff single-cluster algorithm of the Ising model. Proposing an algorithm for a quasi-block synchronization, we realize the Wolff single-cluster Monte Carlo simulation with CUDA. We perform parallel computations for the newly added spins in the growing cluster. As a result, the GPU calculation speed for the two-dimensional Ising model at the critical temperature with the linear size $L = 4096$ is 5.60 times as fast as the calculation speed on a current CPU core. For the three-dimensional Ising model with the linear size $L = 256$, the GPU calculation speed is 7.90 times as fast as the CPU calculation speed. The idea of quasi-block synchronization can be used not only in the cluster algorithm but also in many fields where the synchronization of all threads is required.

1. Introduction

The Ising model is a simple and standard model of statistical physics. The two-dimensional (2D) Ising model, which was exactly solved by Onsager [1], shows a phase transition at the critical temperature $T_c/J = 2.269\ldots$ The Ising model has been studied by using many Monte Carlo techniques, including the Metropolis algorithm [2], cluster algorithm [3,4], extended ensemble algorithm such as the Wang–Landau method [5]. Since the Ising model is simple and its exact solution is available, it is sometimes used to test a new Monte Carlo algorithm.

Recently the speed-up of computation with graphics processing unit (GPU) has captured a lot of attention in many fields including computational biology and chemistry [6], molecular dynamics simulation of fluids [7,8], CT image reconstruction [9,10], finance [11] and much more. GPU has been developed as a supplementary arithmetic device of image processing. GPU has many streaming multiprocessors (SM). Each SM is composed of eight streaming processors (SP) with a multi-threaded instruction unit and shared memory. GPU has realized the acceleration of image processing by using many SMs in parallel. Since GPU is specialized for highly parallel computation, it is designed such that many transistors are devoted to data processing. The common unified device architecture (CUDA) released by NVIDIA makes it possible to make a general purpose computing on GPU. CUDA allows us to implement algorithms using standard C language with CUDA specific extensions. The host program launches a sequence of kernels. A kernel is organized as a hierarchy of threads. Threads are grouped into blocks, and blocks are grouped into a grid. The organization of a grid is determined when the GPU "kernel" function is invoked as

$$kernel\_function <<< grid, block >>> (arguments); \quad (1)$$

Here, *grid* and *block* are integer variables, which specify the number of blocks and that of threads, respectively. For simplicity, we have used one-dimensional representation for the dimensions of *grid* and *block*. Threads in a single block will be executed

* Corresponding author.
E-mail addresses: y-komura@phys.se.tmu.ac.jp (Y. Komura), okabe@phys.se.tmu.ac.jp (Y. Okabe).

0021-9991/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2011.09.029

on a single SM, sharing the data cache, and can synchronize and share data with threads in the same block. There are several types of memories, and there is a significant difference in the access speed of memories. The access speed of global memory, which is accessible from either all the threads or the host, is much slower than that of shared memory, which is accessible from the threads in one block. The access speed of register, which is only accessible by the thread, is nearly equal to that of shared memory.

Preis et al. [12] studied the 2D and three-dimensional (3D) Ising model by using Metropolis algorithm with CUDA. They used a variant of sublattice decomposition for a parallel computation on GPU. The spins on one sublattice do not interact with other spins on the same sublattice. Therefore one can update all spins on a sublattice in parallel when making the Metropolis simulation. As a result they were able to accelerate 60 times for the 2D Ising model and 35 times for the 3D Ising model compared to a current CPU core. More recently, the GPU acceleration of the multispin coding of the Ising model was reported [13].

Metropolis algorithm [2] has flexibility which allows its application to a great variety of physical systems. However, this algorithm is sometimes plagued by long autocorrelation time, named critical slowing down. The cluster update algorithms are efficient for overcoming the problem of critical slowing down. The autocorrelation time drastically decreases by using the Wolff single-cluster algorithm [4]. It is highly desirable to apply parallel computations to cluster algorithms. But only a limited number of attempts have been reported so far along this line. The message passing interface (MPI) parallelization technique was employed by Bae et al. [14] for the Wolff algorithm. Quite recently, Kaupuzs et al. [15] studied the parallelization of the Wolff algorithm by using the open multiprocessing (OpenMP). For the 3D Ising model with linear size $L = 1024$, they reached the speed-up about 1.79 times on two processors and about 2.67 times on four processors, as compared to the serial code.

In this paper, we present the GPU calculation with CUDA for the Wolff single-cluster algorithm of the Ising model. The rest of the paper is organized as follows. In Section 2, we briefly describe the standard way of implementing Wolff single-cluster algorithm on a CPU. In Section 3, we explain the idea of quasi-block synchronization on GPU. In Section 4, we show the idea and techniques of GPU calculation for the Wolff single-cluster algorithm. In Section 5, we compare the performance of GPU calculation with that of CPU calculation. The summary and discussion are given in Section 6.

2. Wolff single-cluster algorithm

Our Hamiltonian is given by

$$\mathcal{H} = -J \sum_{\langle ij \rangle} S_i S_j. \tag{2}$$

Here, J is the coupling and S_i is the Ising spin on the lattice site i. The summation is taken over the nearest neighbor pairs $\langle i,j \rangle$. Periodic boundary conditions are employed.

Wolff proposed a Monte Carlo algorithm in which only a single cluster is flipped at a time [4]. The spin-update process of the Wolff single-cluster algorithm on a CPU can be formulated as follows [16,17]:

(i) Choose a site i randomly.
(ii) Look at each of the nearest neighbors j. If S_j is parallel with S_i, add S_j to the cluster with probability $p = 1 - e^{-2\beta}$, where β is the inverse temperature J/T. Then, flip S_i.
(iii) If all the nearest neighbors j have been checked, look at each of the nearest neighbors k of site j. If S_k is parallel with S_j and it was not a member of the cluster before, add S_k to the cluster with probability $p = 1 - e^{-2\beta}$. Then, flip S_j.
(iv) For all the added spins S_k's, repeat step (iii) until no more new bonds are created.
(v) Go to (i).

Slightly different ways of formulation are possible, for example, for the timing of the flip of spins. To check the condition that no more new bonds are created in (iv), we may use two variables `ic` and `in`, where `ic` is the total number of members of cluster, and `in` is the total number of sites which have been already checked. We repeat the step (iii) while the condition `in` < `ic` is satisfied. We note that the check of site j is done sequentially on a standard CPU.

3. Quasi-block synchronization

In parallel computing, the best performance is obtained if calculation on each thread is done independently. At the same time, barrier synchronization is needed at a certain point. CUDA provides a barrier synchronization function `syncthreads()`. When a kernel function calls `syncthreads()`, all threads in a block will be held at the calling location until everyone else in the block reaches the location. But CUDA is not equipped with an inter-block synchronization function. Since the synchronization for different blocks is required in our GPU calculation of cluster algorithm, we here propose an idea of a quasi-block synchronization.

In CUDA, the organization of a grid is determined by two special parameters provided during kernel launch as mentioned in Section 1. They are the size of the grid in terms of numbers of blocks, *grid*, and the size of each block in terms of numbers of