Int J Parallel Prog (2013) 41:59–88
DOI 10.1007/s10766-012-0204-y

Kernel Polynomial Method on GPU

Shixun Zhang · Shinichi Yamagiwa · Masahiko Okumura · Seiji Yunoki

Received: 10 April 2012 / Accepted: 21 June 2012 / Published online: 4 July 2012

Abstract The simulation of lattice model systems for quantum materials is one of the most important approaches to understand quantum properties of matter in condensed matter physics. The main task in the simulation is to diagonalize a Hamiltonian matrix for the system and evaluate the electronic density of energy states. Kernel polynomial method (KPM) is one of the promising simulation methods. Because KPM contains a fine-grain recursive part in the algorithm, it is hard to parallelize it under the thread level parallelism such as on a supercomputer or a cluster computer. This paper focuses

S. Zhang
Department of Computer Science, Graduate School of System and Information Science, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan
e-mail: sxzhang@cs.tsukuba.ac.jp

S. Yamagiwa (✉)
Faculty of Engineering, Information and Systems, University of Tsukuba/JST PRESTO, Tsukuba, Ibaraki 305-8573, Japan
e-mail: yamagiwa@cs.tsukuba.ac.jp

M. Okumura
CCSE, Japan Atomic Energy Agency, 5-1-5 Kashiwanoha, Kashiwa, Chiba 277-8587, Japan
e-mail: okumura.masahiko@jaea.go.jp

S. Yunoki
Computational Condensed Matter Physics Laboratory, RIKEN ASI, Wako, Saitama 351-0198, Japan

S. Yunoki
JST CREST, Kawaguchi, Saitama 332-0012, Japan

S. Yunoki
Computational Materials Science Research Team, RIKEN AICS, Kobe, Hyogo 650-0047, Japan
e-mail: yunoki@riken.jp

on methods to parallelize KPM on a massively parallel environment of GPU, aiming to achieve high parallelism for more speedups than the recent CPUs. This paper proposes two implementation methods called the full map and the sliding window methods, and evaluates the performances in the recent GPU platform. To enlarge available simulation sizes and at the same time to enhance the performance, this paper also describes additional optimization techniques depending on the GPU architecture.

Keywords Kernel polynomial method · GPU · Condensed matter physics · Strong correlation lattice

1 Introduction

The modern technological advancement, which keeps improving our life more convenient, has been based mostly on discovery of a new class of materials with rich functionalities such as semiconductors, magnets, and superconductors. Researchers in condensed matter physics have been working for an ultimate alchemy to design those materials by modeling the constituent atomic elements using the theory of quantum mechanics, which is now called *materials design* [10]. The implementation for materials design requires large scale computations, and thus it is typically performed on cluster computers and supercomputers [3].

Although the quantum mechanics, which governs the electronic motion in materials, was established more than 80 years ago, there exist many properties and phenomena whose origins are yet to be understood. Such examples include copper based high temperature superconductors [1] and peculiar magnetic insulators of certain organic compounds [17]. The common feature of these materials is a strong quantum correlation between electrons, which is turned out to be crucial for determining their behavior. It is exactly this strong correlation that makes it difficult to treat these systems analytically without introducing any bias.

The best way to treat the strong quantum correlations is to solve quantum mechanical equation of motion numerically and exactly. Since the degrees of freedom increase exponentially with the number of electrons $\sim O(10^{23})$, we still need to resort some sort of approximations. However, unlike the standard analytical treatments, the numerical simulations can handle the strong correlation effects in a controllable manner. Among many, well established numerical methods thus far are exact diagonalization method [2], quantum Monte Carlo method [4,6], density-matrix renormalization group method [11,14–16], and KPM [13]. Each method is suited to particular sets of problems, and also each has severe limitations. For instance, the density-matrix renormalization group method is able to evaluate the ground state as well as the low energy excited states in high accuracy, but it is limited so far to (quasi) one-dimensional systems.

The simulation evaluates various physical quantities such as density of states (DOS) and Green's functions for electrons, which are necessary to study electronic and magnetic structures. In particular, a straightforward method to calculate the DOS by diagonalizing a Hamiltonian matrix requires computational complexity $O(D^3)$, where D is the system size. This complexity is a performance bottleneck to evaluate higher

energy excited states. In this respect, the KPM has an exceptional advantage because the KPM reduces the complexity of diagonalization to $O(D)$ at most by truncating polynomial expansions, which controls the accuracy of the approximation. Therefore, this paper focuses on the KPM to appropriately evaluates the electronic DOS and Green's function for a whole range of excitation energy including higher excited states [13].

The KPM is an approximation method based on polynomial expansions from which physical quantities are evaluated. In particular, the Chebyshev expansion is the most common and useful polynomial to be applied. To avoid the Gibbs phenomenon caused by truncation of the polynomial expansions with a finite order, modified kernel polynomials are preferably used. For example, the Dirac's delta function is well approximated by truncating Chebyshev expansion with the *Jackson kernel* [13]. Moreover, in quantum statistical mechanics, it is required to evaluate the trace of a large-dimensional Hamiltonian matrix. This trace is efficiently approximated by using random vectors [13] (we call it "stochastic trace method" in this paper). By truncating polynomial expansions and adapting random vector bases, the computational complexity is significantly reduced to evaluate the DOS and other physical quantities.

The computational cost inevitably increases with the system sizes, and with the number of polynomials kept and random vectors generated to meet the desired accuracy. It is therefore expected to reduce the simulation latency drastically by implementing the KPM in a parallel platform.

Regarding computer hardware, the graphics processing units (GPU) have become available to be used for acceleration platform as a substitute of CPU. Due to the recent drastic performance growth of GPU, it has already achieved the performance up to TFLOPS order and also has implemented the double precision floating point engines. Therefore, it is applied to various scientific fields to solve the grand challenge applications equipped to a personal computing environment [8].

This paper proposes designs and implementations of the KPM on GPU that achieve much higher performances than the recent CPUs. This paper will show two implementations called the *full map* and the *sliding window* methods. As explained in the next section, vectors (higher order polynomials) are generated recursively. This characteristic is suffered to parallelize the KPM effectively in a CPU-based large system based on thread level parallelism. Applying GPU resources, this paper will challenge to overcome the performance limitation caused by the fine-grain recursive operations. This paper also addresses a potential issue on the memory size which becomes explosively enormous for a larger system in the KPM.

This paper is organized as follows. Section 2 describes the detailed explanation of KPM and the overview of the general purpose computing on GPU. Section 3 proposes the designs and implementations of the KPM on GPU. Section 4 analyzes the performances of typical sets of input parameters used in condensed matter physics and discusses the programming methods proposed in this paper. Section 5 applies additional architectural and application specific optimization techniques for improving the performance discussed in Sect. 4. Finally, Sect. 6 concludes this paper and describes the future directions.

2 Backgrounds and Definitions

2.1 Simulations in Condensed Matter Physics

Employing the theory of quantum mechanics, the simulations in condensed matter physics define models with atoms arranged in one to three spatial dimensions (D) as illustrated in Fig. 1. For example, Fig. 1a shows a 1D model with four sites, where each site mimics an atom of a target material. The simulation analyses the quantum mechanical motion of electrons, which are described by "hopping" from one atom to another in the model. The hopping amplitude $t_{ij} (= t_{ji})$ is given by the overlap between wave functions of electrons locating at site i and its nearest neighbor site j. The potential energy on site i is denoted by ε_i. Using these definitions, the Hamiltonian matrix of the model is organized, e.g., as shown in Fig. 1b for a 1D system. The characteristics of the matrix are i) the number of rows or columns (called H_SIZE in this paper) is determined by the total number of sites in the model system, ii) the number of non-zero matrix elements is given by summing the number of non-zero hoppings (t_{ij}) and the number of non-zero on-site potentials (ε_i), and iii) other elements are zeros because there are no hoppings and thus correlations between farther sites. For example, in the case of Fig. 1a (see also Fig. 1b), $H\_SIZE = 4$ (the total number of matrix elements: 4×4) and the number of non-zero and zero matrix elements are 12 and 4, respectively. For this 1D example, the matrix dimensions are very small. Note, however, that most of the target materials in condensed matter physics are modeled by 3D lattices, as depicted in Fig. 1c for one of the simplest 3D models. When we consider a small 3D model, e.g., a $10 \times 10 \times 10$ lattice model, $H\_SIZE = 1,000$ and the number of non-zero and zero matrix elements are 7,000 and 993,000, respectively. Generally, in 3D system, the dimension of matrix and the number of non-zero matrix elements become larger quickly with respect to the number of sites.

In order to study physical properties of materials, the simulation computes the electronic DOS. If we evaluate the DOS exactly, the full diagonalization of the Hamiltonian matrix is necessary. This computation requires the complexity of $O(H\_SIZE^3)$ using

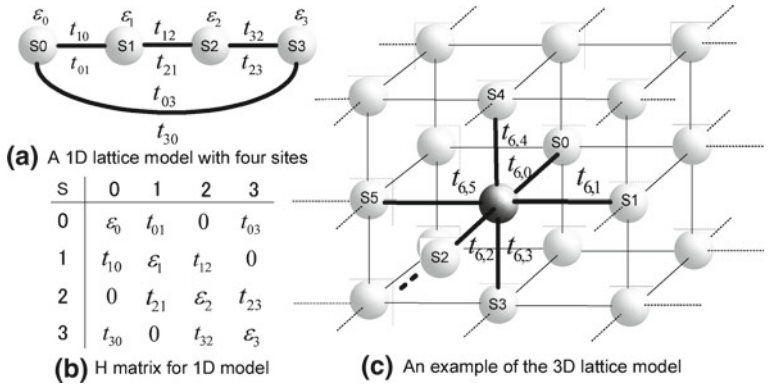


Fig. 1 Modeling electrons in **a** one and **c** three dimensional systems. Elements in the Hamiltonian matrix for the one dimensional system are tabulated in **b**

a straightforward algorithm. It is highly desirable to reduce this complexity because the increase of H_SIZE causes the computation time explosion. One of the best methods widely used to reduce the computation time is the kernel polynomial method (KPM), which gives an approximated DOS in a controlled manner [13].

2.2 Kernel Polynomial Method

2.2.1 Definition

The basis of KPM is the following (Chebyshev) polynomial expansion of a function $f(x)$ defined in $[-1, 1]$,

$$f(x) = \frac{1}{\pi\sqrt{1-x^2}}\left[\mu_0 + 2\sum_{n=1}^{\infty}\mu_n T_n(x)\right], \tag{1}$$

where

$$\mu_n = \int_{-1}^{1} dx\ f(x)T_n(x), \tag{2}$$

and $T_n(x)$ is the Chebyshev polynomial defined as

$$T_n(x) = \cos\left[n \arccos(x)\right]. \tag{3}$$

It should be mentioned that the Chebyshev polynomials satisfies the following recursion relations,

$$T_0(x) = 1\,, \quad T_1(x) = x, \tag{4}$$

$$T_{n+2}(x) = 2xT_{n+1}(x) - T_n(x). \tag{5}$$

KPM is defined as

$$f_{\mathrm{KPM}}(x) = \frac{1}{\pi\sqrt{1-x^2}}\left[g_0\mu_0 + 2\sum_{n=1}^{N-1} g_n\mu_n T_n(x)\right], \tag{6}$$

where the additional coefficients g_n given by a kernel which satisfies the limit

$$\|f - f_{\mathrm{KPM}}\| \xrightarrow{N\to\infty} 0\,, \tag{7}$$

where $\|\cdot\|$ is suitable well-defined norm.

2.2.2 Application to Quantum Systems

In quantum physics, we need to expand functions of the Hamiltonian matrix. In this paper, we focus on the DOS. Then, we show an example of application of KPM for calculation of DOS.

We consider the system described by the Hamiltonian matrix H. First, we apply the following linear transformation in order to fit the spectrum of H to $[-1, 1]$,

$$\tilde{H} = (H - \alpha_+)/\alpha_-, \tag{8}$$

where

$$\alpha_\pm = (E_{\text{upper}} \pm E_{\text{lower}})/2, \tag{9}$$

The parameters E_{upper} and E_{lower} are the upper and lower limits of the eigenvalues of H obtained by the Gerschgorin theorem [12].

The density of state (DOS) $\rho(\omega)$ of the D-dimensional Hamiltonian matrix H is defined by

$$\rho(\omega) = \frac{1}{D}\sum_{k=0}^{D-1} \delta(\omega - E_k), \tag{10}$$

where E_k is the kth eigenvalue and $\delta(x)$ is the delta function. We apply the linear transformation (8) and obtain the equation

$$\rho(\tilde{\omega}) = \frac{1}{D}\sum_{k=0}^{D-1} \delta(\tilde{\omega} - \tilde{E}_k), \tag{11}$$

where

$$\tilde{\omega} = (\omega - \alpha_+)/\alpha_-. \tag{12}$$

In order to obtain the approximated DOS using KPM, the coefficients μ_n (2) in this case is obtained as

$$\begin{aligned}
\mu_n &= \int_{-1}^{1} d\tilde{\omega}\, \rho(\tilde{\omega}) T_n(\tilde{\omega}) \\
&= \frac{1}{D}\sum_{k=0}^{D-1} T_n(\tilde{E}_k) \\
&= \frac{1}{D}\sum_{k=0}^{D-1} \langle k|T_n(\tilde{H})|k\rangle = \frac{1}{D}\mathrm{Tr}[T_n(\tilde{H})],
\end{aligned} \tag{13}$$

where $|k\rangle$ is the kth eigenvector and $\langle k| = |k\rangle^\dagger$.

2.2.3 Stochastic Evaluation of Traces

In order to evaluate the trace in Eq. (13), we introduce the stochastic evaluation method of traces, which estimates μ_n by average over only a small number $R \ll D$ of randomly chosen vector.

First, we introduce an arbitrary basis $\{|i\rangle\}$ a set of independent identically distributed random variables $\{\xi_{r,i}|\xi_{r,i} \in \mathbb{R}\}$ which in terms of the statistical average $\langle\langle\cdot\rangle\rangle$ fulfill

$$\langle\langle \xi_{r,i} \rangle\rangle = 0\,, \quad \langle\langle \xi_{r,i}\xi_{r',i'} \rangle\rangle = \delta_{rr'}\delta_{ii'}, \tag{14}$$

a random vector is defined through

$$|r\rangle = \sum_{i=0}^{D-1} \xi_{r,i}|i\rangle. \tag{15}$$

Using them, we can approximately evaluate the trace as follows,

$$\begin{aligned}
\mu_n &= \frac{1}{D}\mathrm{Tr}\left[T_n(\tilde{H})\right] \\
&= \frac{1}{D}\sum_{i=0}^{D-1}\left[T_n(\tilde{H})\right]_{ii} \\
&\simeq \frac{1}{D}\frac{1}{R}\sum_{i,j=0}^{D-1}\sum_{r=0}^{R-1}\langle\langle \xi_{r,i}\xi_{r,j} \rangle\rangle \left[T_n(\tilde{H})\right]_{ij} \\
&= \left\langle\!\!\left\langle \frac{1}{D}\frac{1}{R}\sum_{r=0}^{R-1}\langle r|T_n(\tilde{H})|r\rangle \right\rangle\!\!\right\rangle.
\end{aligned} \tag{16}$$

In order to make $\langle r|T_n(\tilde{H})|r\rangle$, we use the following recursive relations for the vectors $|r_n\rangle := T_n(\tilde{H})|r\rangle$ derived from the relations (4) and (5),

$$|r_0\rangle = |r\rangle\,, \quad |r_1\rangle = \tilde{H}|r_0\rangle, \tag{17}$$

$$|r_{n+2}\rangle = 2\tilde{H}|r_{n+1}\rangle - |r_n\rangle. \tag{18}$$

Then μ_n is expressed by this expression as

$$\mu_n \simeq \left\langle\!\!\left\langle \frac{1}{D}\frac{1}{R}\sum_{r=0}^{R-1}\langle r_0|r_n\rangle \right\rangle\!\!\right\rangle. \tag{19}$$

2.2.4 Numerical Complexity

The numerical complexity of the KPM is $O(SRND)$ if the $\tilde{H}$ is sparse matrix, where S is the number of the realization of the set of random variables $\{\xi_{r,i}\}$. The process

costing $O(D)$ is the making part of $|r_n\rangle$ shown in Eq. (18), which is the heaviest part in KPM. When the $\tilde{H}$ is considered as a dense matrix, the complexity of the part becomes $O(D^2)$. The $O(SR)$ comes from the average and summation in Eq. (19) and $O(N)$ from the number of recursive iterations in Eqs. (17) and (18). This numerical cost $O(SRND)$ is very effective against the full diagonalization which costs $O(D^3)$ if $SRN \ll D^2$, and the $\tilde{H}$ is a sparse matrix. However, when it is a dense matrix, the numerical cost becomes $O(SRND^2)$ due to all multiplications for all elements in the $\tilde{H}$ and the $|r_n\rangle$ must be performed straightly without considering the CRS (*Compressed Row Storage*) format as used for storing a sparse matrix efficiently.

2.3 Compression Techniques for Sparse Matrix

The multiplication between a Hamiltonian matrix and a vector takes the most computation in KPM algorithm. Additionally, Hamiltonian matrix is usually huge (for example, correlation among one hundred atoms in a 3D cubic lattice model is presented by a one million by one million matrix) due to the fact that physical system usually contains huge amount of atoms or molecules. Therefore, in order to simulate huge physical systems and improve the performance of simulation, compression techniques are almost inevitably introduced to Hamiltonian matrix. In this paper, in addition to the discussion of the dense matrix implementation, we also discuss the impact of both compression row format (CRS) and ELL format [5].

2.3.1 CRS Format

The CRS format is a well-known expression technique that reduces memory usage in computer applying to a sparse matrix. Figure 2 shows an example of 6×6 sparse matrix. The A is an array of non-zero elements. The arrays IA and JA contain the indices of row and column respectively of elements in A. The IA can be simplified to IA' by translating the elements to the positions where each element in each row appears. Thus, represented the original matrix by A, IA' and JA, the total memory usage can be reduced. In the example shown in Fig. 2, the original matrix can be compressed to about 80 %.

Applying the CRS format to the Hamiltonian matrix of 3D cubic lattice model, the number of bytes is calculated as follows:

$$H\_SIZE \times 7 \times 8 + H\_SIZE \times 7 \times 4 + H\_SIZE \times 4 = 88 \times H\_SIZE$$

Fig. 2 CRS format

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 5 & 6 & 0 & 0 \\ 0 & 0 & 0 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 9 & 10 \\ 0 & 0 & 0 & 0 & 0 & 11 \end{bmatrix}$$

$A = [1,2,3,4,5,6,7,8,9,10,11]$
$IA = [1,1,2,2,3,3,4,4,5,5,6]$
$JA = [1,2,2,3,3,4,4,5,5,6,6]$
$IA' = [1,3,5,7,9,11]$
$Ratio = (11+11+6)/36 \approx 0.8$

because each row in the matrix includes 7 non-zero elements. Here, elements in A consists of 8 byte double precision floating point numbers. Elements in JA and IA' consists of 4 byte integer numbers.

When the CRS format is applied to the sparse matrix used in the simulation of the three dimensional lattice illustrated in Fig. 1c, the larger the number of sites is, the higher the compression ratio is achieved. Thus, it is very effective to apply the CRS format to a large sparse matrix used in the computation of the KPM.

2.3.2 ELL Format

ELL format is another effective compression format for sparse matrix. Using the same sparse matrix used in the case of CRS format, ELL format compress it to a non-zero element matrix A and a column index one JA as shown in Fig. 3. The A consists of non-zero elements in corresponding rows. The number of rows of A is equal to the one of the original sparse matrix. If the number of columns does not match to the maximum number of non-zero elements in a row of the sparse matrix, A is padded by $*$ to indicate zero-elements. To implement the $*$, 0 or -1 is applied depending on the application. In our case, we apply 0 to it to ignore multiplication between a non-zero number and a zero because KPM performs matrix multiply mainly. On the other hand, JA consists of column numbers of non-zero elements in the original sparse matrix. The number of rows corresponds to the original matrix. The $*$ is also applied to JA as the same definition of A. Thus ELL format generates two matrices of the same size with the number of rows that equals to the original sparse matrix. The sparse matrix of Fig. 3 can be compressed to about 66 %.

In the case of 3D lattice cubic model we consider, the total number of bytes in ELL format is calculated as follows:

$$H\_SIZE \times 7 \times 8 + H\_SIZE \times 7 \times 4 = 84 \times H\_SIZE$$

when the elements in A employs 8 byte double precision floating point and the ones of JA employs 4 byte integer numbers.

As we can see in the equations above, ELL format generates the smaller number of bytes in the actual use in the 3D cubic model. The efficiency is about $4/88 \approx 4.5$ %. However, this efficiency is given by the special case due to the characteristics of the correlations of 3D cubic lattice model illustrated in Fig. 1c that is expressed by a sparse matrix of seven non-zero elements in each row. Because KPM can be applied to general physics or chemistry problems to find eigenvalues with diagonalizing a correlation matrix of atoms. In those general use of KPM, the compression ratio of ELL can

Fig. 3 ELL format

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 5 & 6 & 0 & 0 \\ 0 & 0 & 0 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 9 & 10 \\ 0 & 0 & 0 & 0 & 0 & 11 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \\ 9 & 10 \\ 11 & * \end{bmatrix} \quad JA = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 5 & * \end{bmatrix}$$

$$Ratio = (12 + 12) / 36 \approx 0.66$$

become worse than CRS. Therefore, because CRS format is a compression method for any sparse matrix. Here, we first focus on the CRS format in KPM implementation.

2.4 General Purpose Computing on GPU

2.4.1 GPU Architecture

A video adapter that includes a GPU and a Video RAM (VRAM) is connected to a CPU's peripheral bus such as PCI Express. The video adapter works as a peripheral device of the CPU, and its GPU is controlled by the CPU to perform a part of visualization tasks in the system. To utilize the GPU as a computing resource for GPGPU applications, the CPU downloads the application program, called kernel program, to the GPU's instruction memory and also prepares input data for the program.

The recent GPUs have only a kind of processor called the *stream processor*. Hundreds of the stream processors are massively integrated in an LSI chip and work together concurrently fetching the SIMD style program. The processor works for general purpose processing in any kind of calculation. However, the computing style must be followed in the stream-based one that enforces the programmer to revise the original program targeted to the von Neumann style architecture to the stream processing style.

The GPU-based program fetches the data and generates the result to the memory areas. The GPU reads/writes the VRAM directly to execute the calculation for the program. In this case, the original data is prepared in the main memory. The CPU copies the data to the VRAM. During the execution of the program, the GPU generates the results to the VRAM. The CPU copies the results from the VRAM to the main memory. During the execution of the program, GPU uses two types of memory called *global* and *shared* memories. The global memory is provided by the memory placed outside of GPU such as DDR3 VRAM. The shared memory is placed besides of the stream processor that works as if a semi-automated data cache.

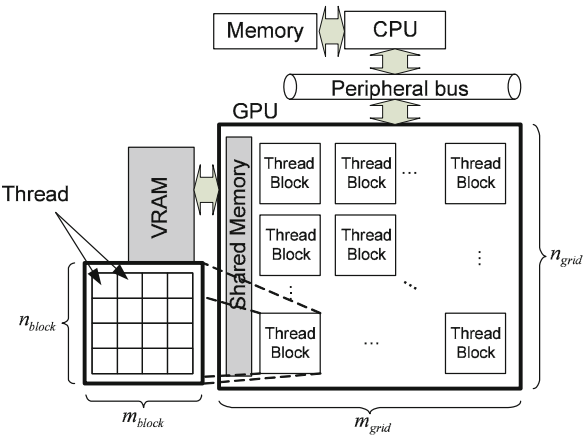
In addition to the massively parallel processing ability of the GPU, it has a large I/O capacity in the memory interface. For instance, the NVIDIA's Tesla C2050 provides its peak memory bandwidth up to 144GB/s, according to its profiler named *computeprof*. On the other hand, the recent CPU achieves the following theoretical peak I/O bandwidth applying dual channel DDR2-800 memory modules:

$$400MHz \times (2channels) \times (64\,bits/channel) \times (2\,bits/clock) = 12.5GB/s$$

Actually the Intel's Core i7 processor achieves about 9 GB/s using the *stream benchmark* [7], which is only less than 10 % of the peak bandwidth of C2050. Thus, to utilize the large I/O bandwidth of GPU, it is effective to brush up the I/O part of the program.

2.4.2 CUDA

The compute unified device architecture (CUDA) has been proposed by NVIDIA corporation [9]. The CUDA assumes an architecture model as illustrated in Fig. 4a.



(a) Execution model of kernel threads in CUDA

```
int main(){
  float *A, *B, *C;
  ...
  dim3 dimBlock( m_block, n_block );
  dim3 dimGrid( m_grid , n_grid );
  KernelFunc<<< dimGrid , dimBlock >>>(A, B, C);
  ...
}

__global__ void KernelFunc(float *a, float *b, float *c){
  int i = blockDim.x * blockDim.y *
          (gridDim.x * blockIdx.y + blockIdx.x) + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

(b) Example CUDA code for array summation

Fig. 4 Programming model on CUDA environment

The model defines a GPU which is connected to a CPU's peripheral bus. A VRAM (the global memory) that maintains data used for calculation on the GPU is connected to the GPU. The data is copied from the host memory before the CPU commands to execute a program on the GPU. The program is executed as a thread in a thread block. The thread blocks are tiled in a matrix of from one to three dimensions. In the figure, thread blocks are tiled in two dimensions which size is $n_{grid} \times m_{grid}$. Each thread block has multiple threads in a matrix which size is varied from one to three dimensions. The figure also shows a thread block that includes $n_{block} \times m_{block}$ threads. The number of threads in a thread block is represented as BS in this paper. Each thread block has individual shared memory space where shared values accessed among threads in the block are stored temporally. Thus, the program targeted to GPU in the CUDA environment is invoked as threads. The threads are grouped by the unit of the thread block. Therefore, a large number of threads are invoked concurrently obtaining a large parallelism.

In the program on the CUDA environment, the threads are described as a stream-based function written in C called a *kernel function* as shown in Fig. 4b. The program has two parts of the codes targeted to CPU and GPU, which is initially invoked by the CPU; a main program for CPU and a kernel function called as the thread on GPU. The kernel function is defined with the `__global__` directive so that it is executed on GPU. In the function, the global variables named `gridDim`, `blockDim`, `blockIdx`, `threadIdx`, implicitly declared by the CUDA runtime, are available to be used to specify the size of the grid and the thread block, the indices of the thread block and of the thread respectively. For example, using these global valuables, Fig. 4b performs a summation of arrays A and B assigning each summation of the elements in those arrays to a thread and returns the result to the array C. The function is called by the main program specifying the sizes of the grid and the thread block with `<<<>>>`. Finally, reading data from the VRAM transferred by the main program, the kernel function is assigned to GPU, and runs as multiple threads. Thus, because programmer can just simply consider the stream-based kernel function and the calling code for the function in the main program, using the conventional C language manner, the CUDA provides an easy to access to the computational resource and transparent interface for GPGPU.

2.5 Summary

The aim of this paper is targeted to speedup the simulation for the strong correlation lattice system using parallel processing techniques. However, as we can see in the KPM algorithm the calculation of the $\overrightarrow{r}_n$ as seen in the Eq. (18) forms recursive operations. Therefore it is clear that the implementation based on a fine-grain parallelization under thread level parallelism causes fatal bottleneck of the performance such as on supercomputers and cluster computers among processor cores because either message passing or shared memory implementation needs frequent synchronizations to scatter and gather $\overrightarrow{r}_i$ and a part of the H matrix among all processors via an interconnection network. On the other hand, GPU contains the stream processors that can concurrently access to the global memory to share variables among threads. The accesses to the global memory from the threads are performed fast due to the high bandwidth local bus massively connected to the external VRAM. Therefore, the fine-grain parallelism can be extracted by enormous number of threads concurrently working with shared variables among the threads. Thus, GPU can be expected to address the performance bottleneck of the KPM.

In addition to the parallelization difficulty due to the recursive operation discussed above, the KPM has also a memory usage problem. When we consider to implement the KPM on GPU, during the $\overrightarrow{r}_i$ calculation the entire H matrix must be stored in the global memory due to accesses from all threads. However, the H matrix is very huge. We need to invent any effective implementation with reasonable memory usage that extends available problem size.

Thus, this paper focuses on implementing effective parallelization methods of the KPM invoked on GPU maintaining two aspects of the highly parallelism and the lowest memory consumption.

3 KPM on GPU

Let us introduce the GPU-based implementations applying mainly two methods to parallelize the KPM. One occupies in a large memory area to calculate directly the $\overrightarrow{r_n}$ and generate the $\tilde{\mu}_N$. Another reduces the memory consumption by applying a technique to accumulation of $\tilde{\mu}_N$ in a fixed memory square. Before describing the implementation detail, let us begin to explain the algorithm design of the KPM.

3.1 Algorithm Design

Figure 5 summarizes the KPM algorithm. The step (1) generates randomly a vector $\overrightarrow{r}$ that the number of elements is H_SIZE (this equals to the D used in Sect. 2.2). The step (2) gets $\overrightarrow{r_n}$ from $\overrightarrow{r_{n-1}}$ and $\overrightarrow{r_{n-2}}$ recursively calculating a matrix multiply of H and $\overrightarrow{r_{n-1}}$ in the step (2.1). This multiplication potentially obtains difficulty to be parallelized based on the thread level parallelism using MPI or OpenMP due to the dependencies of the recursive iteration where the most intensive calculation is needed. Then a dot product is calculated using $\overrightarrow{r_n}$ again with $\overrightarrow{r}$ at the step (2.2) and generates $\tilde{\mu}_n$. Each $\tilde{\mu}_n$, where $1 \leq n \leq N$, is calculated repeatedly for RS times. The generation of the $\tilde{\mu}_n$ is iterated for RS times. Finally, the average of all the $\tilde{\mu}_n$s is generated at the step‘(3).

Each generation of $\tilde{\mu}_n$ can be massively parallelized on GPUs, and then N μ_ns are finally generated from the RS-time iterations of the step (1) and (2). This generation of the moments achieves the objective of the KPM. The summation to generate $\tilde{\mu}_n$ can be also parallelized on GPU. Therefore, implemented on GPUs, two parallel processing parts are entirely performed during the evaluation of the moments using KPM: a) generation of $\overrightarrow{r_n}$ and b) generation of μ_n.

Here, GPU has an architectural restriction to the number of threads in a thread block referred as BS in this paper. Considering the parallelization techniques above, let us explain the implementation of a kernel program on CUDA that invokes both the a) generation of $\overrightarrow{r}$ and b) generation of $\tilde{\mu}_n$ parts.

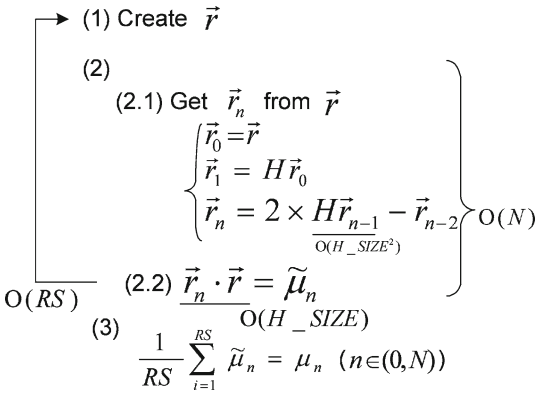


Fig. 5 Algorithm and complexity regarding H_SIZE, N, R, and S of KPM

3.2 Overall Implementation

We have implemented a kernel program for GPU using CUDA. The kernel receives the H_SIZE, N that is the number of moments and RS as the arguments. All calculations are performed based on double precision. The kernel includes two important concepts; one is how to keep high parallelism, another is an effective memory management without reducing the parallelism.

Focusing on the $\tilde{\mu}_n$ calculation, we propose two implementation techniques called the *full map* [18] and the *sliding window* methods. The former provides a simple implementation, which is a straight forward method without complex control for accessing the memory resource to calculate μ_n. On the other hand, the latter one needs a complex control for accessing the memory resource, but is expected less memory consumption. Let us explain these two methods in the following sections.

3.2.1 The Full Map Method

Figure 6(1) shows the generation part for the $\overrightarrow{r_n}$. $\overrightarrow{r_n}$ needs $\overrightarrow{r_{n-1}}$, $\overrightarrow{r_{n-2}}$ and $\overrightarrow{r}$ that is randomly generated. These four vectors are obtained in the global memory and each block will write those vectors swapping the pointers. BS threads work concurrently to generate vectors $\overrightarrow{r}$ and $\overrightarrow{r_n}$ according to the calculation of Fig. 5(2.1). Therefore, this part will generate $\tilde{\mu}_1, \tilde{\mu}_2, \ldots, \tilde{\mu}_N$ using $\overrightarrow{r}$ and $\overrightarrow{r_n}$ in the iteration regarding n from 1 to N.

Figure 6(2) depicts the parallelization for generation of μ_n. It performs a dot product of $\overrightarrow{r}_n$ and $\overrightarrow{r}$, and saves all $\tilde{\mu}$ to another memory area. Finally, working in parallel, all threads in a block just make summation for a scalar $\tilde{\mu}_n$ where $1 \leq n \leq N$.

Here, let us consider the required memory amount for the full map method in the case of double precision. For the operation (1) depicted in Fig. 6, because four $\overrightarrow{r}$

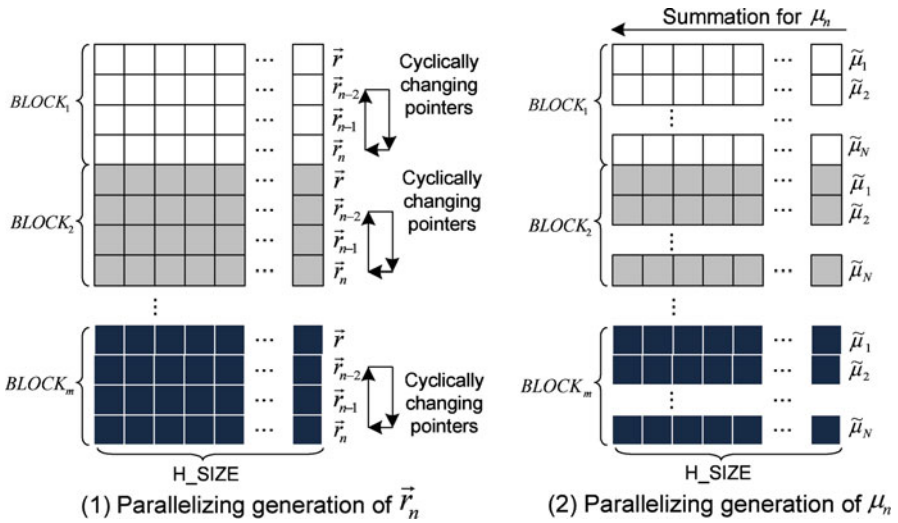


Fig. 6 Implementation applying the full-map method

vectors per block are stored in the global memory. Each $\overrightarrow{r}$ vector has H_SIZE elements. Therefore, this part consumes $Number\ of\ Blocks \times 4 \times H\_SIZE \times 8$ bytes. During the operation (2), each block performs summations to produce N $\tilde{\mu}$s. Each $\tilde{\mu}_n$ is spread horizontally to a vector of H_SIZE long. Therefore, it needs totally

$$Number\ of\ Blocks \times N \times H\_SIZE \times 8$$

bytes.

Because both operations need the H matrix, the matrix is permanently stored in the memory occupying $H\_SIZE^2 \times 8$ bytes. The operation (1) writes $\tilde{\mu}_n$ into the global memory. This needs to be kept with $\overrightarrow{r}$ vectors simultaneously. Therefore, the total number of memory needed for the full map method is calculated as follows:

$$H\_SIZE^2 \times 8 + Number\ of\ Blocks \times H\_SIZE \times (8 \times N + 32)$$

This method seems to have an advantage of less control overhead because all variables are prepared in memory. However, the memory usage increases linearly by the number of thread blocks. Thus, this method must decrease the parallelism (i.e. reducing the number of thread blocks) to calculate the larger lattice model.

3.2.2 The Sliding Window Method

Let us examine how much memory where the full map method uses in the case when H_SIZE is $100 \times 100 \times 100$, $N = 128$ and $Number\ of\ Blocks = 8$. Here we assume that the H matrix is not allocated in the memory because we will discuss it later. The operation (1) in Fig. 6 costs 256 Mbytes. The operation (2) in Fig. 6 costs 8 Gbytes that will be increased explosively when we increase the truncation N. This lattice is not able to be simulated by the full map method because 32Gbytes for the data structure of Fig. 6(2) actually does not exist technically on the recent GPU boards. Therefore, although the control overhead would increase, the operation (2) should be improved not to consume such a large memory area.

We propose another method for the operation (2) of the full map method called *sliding window* method. The former part of the sliding window method corresponds to the operation (1) in Fig. 6 of the full map method. Figure 7 summarizes the operation in the sliding window method that corresponds to the operation (2) of the full map method. The operation is performed by two parts; one is accumulation of $\tilde{\mu}_i$ partially and another is final reduction of $\tilde{\mu}_i$.

The first part prepares a memory area where the square is $BS \times BS \times 8$ bytes in each thread block. Each block performs generations of $\tilde{\mu}_i$ where $1 \leq i \leq N$ from the dot product $\overrightarrow{r_n} \cdot \overrightarrow{r}$ according to the operation (1). Each multiplication performed in the dot product (i.e. $\overrightarrow{r}_n[i] \times \overrightarrow{r}[i]$ where $1 \leq i \leq H\_SIZE$) is stored and added to variables in the window memory that correspond to divided summations assigned to the computing threads. This means that BS threads in a thread block calculates the dot products in parallel and it iterates the parallel calculation for $BS\tilde{\mu}$s. Finally summations of BS $\overrightarrow{r_n} \cdot \overrightarrow{r}$ are stored in the window memory. Thus BS threads concurrently works to implement the parallelism.

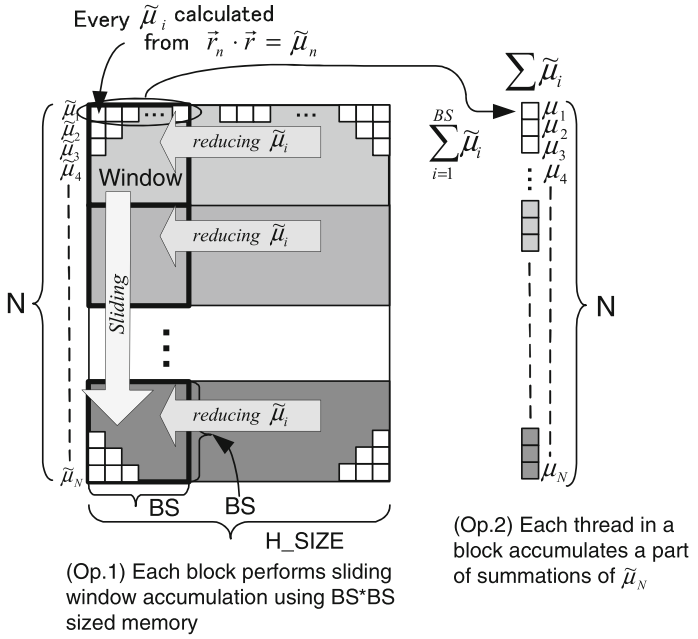


Fig. 7 Implementation applying the sliding window method. Each thread block manages this operations using the memories

Figure 7(Op.1) shows the first calculation part using the window. The thread block needs to prepare only the window memory. After making summation of the $\tilde{\mu}$s from the dot products of $\overrightarrow{r_n} \cdot \overrightarrow{r}$, and reducing all $\tilde{\mu}$s with summations into the window, the window slides to the next BS $\tilde{\mu}$s. For example, assume $BS = 4$, $H\_SIZE = 12$, $N = 24$ and the window is $w[\,][\,]$. The thread i produces

$$\sum_m \overrightarrow{r_n}[m \times BS + i] \times \overrightarrow{r}[m \times BS + i]$$

where $0 \leq m \leq 2$, and saves it to $w[j\%4][i]$ where $1 \leq i \leq 4$ and $1 \leq j \leq 24$. Thus, the window keeps the part of $\tilde{\mu}_i$.

The second part just makes summations in parallel assigning each row to a thread and reduces the final summation of $\tilde{\mu}_i$ to an array allocated in another memory area sized in $N \times 8$ bytes as depicted in Fig. 7(Op.2). Because every iteration of $R \times S$ times accumulates the summation of $\tilde{\mu}_i$ to the memory. Using the same parameters above, the second part makes summations of $w[i][j]$ for solving μ_i by the thread i, and saves the μ_i to the different memory area of Fig. 7(Op.2) where $1 \leq i \leq 4$ and $1 \leq j \leq 4$. To calculate the μ_i where $5 \leq i \leq 8$, the window is shifted below and then repeated the first and the second parts until the window includes μ_N.

Here, let us estimate the total memory size needed when H_SIZE is $100 \times 100 \times 100$, $N = 128$ and $Number\ of\ Blocks = 8$ as the same case considered for the full map method. The generation of $\overrightarrow{r_n}$ in the KPM needs the same memory size as the one in the full map method, which is 256 Bytes. On the other hand, the generation of μ_n becomes $Number\ of\ Blocks \times (BS \times BS \times 8 + N \times 8)$. When we apply the

actual parameters to this equation, it becomes about 1MByte. Thus, the sliding window method reduces the memory usage drastically, and makes the large size simulation available.

According to the discussion about the memory usage of the sliding window method above in the case of double precision, we can conclude that the memory cost is estimated using the equation below:

$$H\_SIZE^2 \times 8 + Number\ of\ Blocks \times H\_SIZE \times 4 \times 8 + Number\ of\ Blocks \times (BS \times BS \times 8 + N \times 8)$$

3.3 Discussion

This section focused on two methods that parallelize the KPM on the GPU. The KPM has a fatal bottleneck regarding the required amount of memory. Especially all threads must access the same H matrix. However, the H matrix is typically sparse in our target lattice simulation as explained in Sect. 2.1. When we apply the CRS format to the H, it is clear that we can reduce the consumed amount of memory. Moreover, each row of the H has only seven elements in the case of 3D lattice model. This mean that statically the sizes of A, IA' and JA become $H\_SIZE \times 7 \times 8$ bytes, $H\_SIZE \times 4$ bytes and $H\_SIZE \times 4$ bytes respectively when the index is stored in a 32bit integer. When we consider H_SIZE is $256 \times 256 \times 256$, the dense case needs 2 Peta Bytes for the H matrix. But the CRS format needs only 1 GBytes. Thus, the size problem can be moderated by the CRS format.

GPU has another technical optimization possibility in the architecture. GPU also has a data cache memory between the stream processors and the global memory. It is actually implemented on a thread block. In the default configuration of NVIDIA C2050 case, the GPU assigns 16Kbytes to the L1 cache memory and assigns 48Kbytes to the shared memory accessed by the threads. Calling `cudaFuncSetCacheConfig` function in CUDA API from the CPU side, it swaps the sizes between the cache memory and the shared memory. Thus, we can extend the size of the data cache memory related to a thread block. The larger the data cache memory is, the more effective threads read the H matrix allocated in the global memory.

As discussed in the sections above, the implementations on GPUs will perform highly parallelism with the enormous numbers of threads concurrently working together. Thus, it is expected that the KPM on GPU will extract the potential performance of the massively parallel platform and achieves better performance than the recent CPUs.

4 Experimental Performance Analysis

4.1 Experimental Setup

This section shows performance evaluations of the KPM implemented with the techniques discussed in the sections above. The performance based on GPU is compared

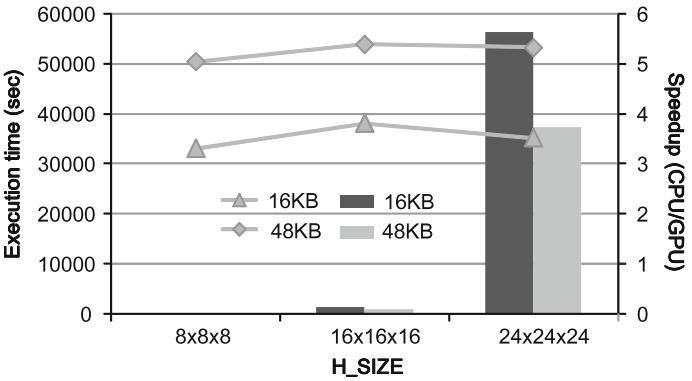
with the one based on CPU and the ratio (CPU time/GPU time) is shown as the speedup. The experimental environment is a PC that consists of an Intel's Core i7 930 processor at 2.80 GHz with 12GB DDR3 memory, and the NVIDIA Tesla C2050 with 3GB memory that is connected to the PCI Express bus. The OS of the PC is the CentOS of the Linux Kernel 2.6.18. The driver version of the GPU is 3.2. All KPM calculations are performed with double precision floating point. The CPU version is implemented straightly employing arrays and a matrix dynamically allocated, is brushed to achieve the best performance in an execution thread, and is compiled with GCC 4.4.1 with O3 option. We also compiled the CPU version by the Intel C compiler. However, the performance changes less than 5 % better/worse than the GCC version because the I/O overhead for the H matrix dominates the total execution time. Therefore, to compare the general performance on Linux, we use the performance of the GCC version to compare with the one of GPU.

We evaluate performances between the full map and the sliding window methods applying 16 or 48 Kbyte L1 cache and also with/without the CRS format. All evaluations in this section apply $N = 128$, $R = 14$ and $S = 128$ for the KPM parameters. In the aspects of the architectural parameters on GPU, we apply $BS = 128$, $Number\ of\ Blocks = 32$ for all the evaluations.

4.2 Evaluation for the Full Map Method

The first evaluation analyzes the performance of the full map method applying an allocation of whole H matrix in the global memory. This case becomes equivalent to the one when a dense matrix is applied to the H matrix. In Fig. 8, the performances are shown as bars, the speedups are depicted in lines with points.

As we discussed in Sect. 2.2, due to the dense matrix, the performance follows the complexity $O(SRND^2)$. Therefore, changing H_SIZE causes sensitive increase of



H_SIZE	8x8x8	16x16x16	24x24x24
16KB cache	19 sec	1332 sec	56258 sec
48KB cache	13 sec	942 sec	37208 sec

Fig. 8 Performances of the full map method comparing among 16 KByte/48 KByte cache and the speedup

the execution time. Moreover, the required amount of memory on the GPU is exhausted by the full map method when H_SIZE is $32 \times 32 \times 32$. Therefore we applied $8 \times 8 \times 8$, $32 \times 32 \times 32$ and $24 \times 24 \times 24$ to the experiments in this section.

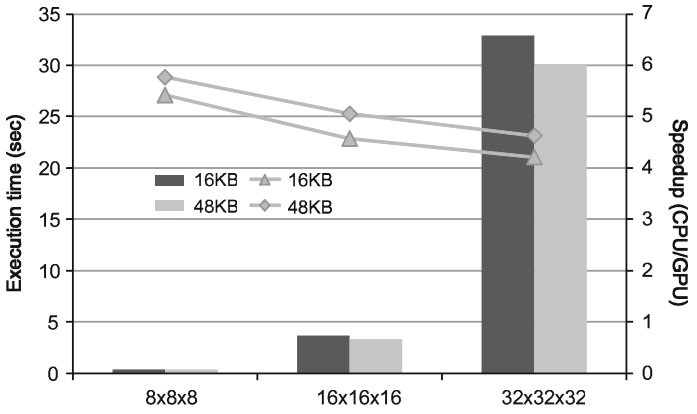
We confirmed that the performances with 48Kbyte cache size achieve almost 30–35 % better performance than the ones with 16 Kbyte cache size. This means clearly that the larger cache size enhances the performance because the cached part of the $\overrightarrow{r}$ or the H matrix can be effectively shared with all the threads in a block.

Because the full map method achieves only less than six times better performance than the GPU. This means that it is better to use the recent CPU which has eight processor cores assigning eight threads to the processor to achieve better performance than the GPU's. Therefore, although the full map method does not include much control code, it can not achieve reasonable performance comparing to the recent CPU and also it is very hard to increase the problem size because the required amount of memory is large. Thus we have confirmed that the full map method does not achieve good performance.

4.3 Evaluation Applying the CRS Format

Let us apply the CRS format to the full map method. Figure 9 illustrates the performances. We confirmed that the CRS format drastically improves the performance in both GPU and CPU versions because the calculation amount is much reduced by ignoring redundant calculations with zeros. We also confirmed that the complexity becomes $O(SRND)$ from the performance shown in Fig. 9 as we discussed in Sect. 2.2.

The speedup seems to be saturated to about 4–5. In the aspect of the speedup, the CRS format is not effective because the CPU version also has large improvement of the



H_SIZE	8x8x8	16x16x16	32x32x32
16KB cache	0.39 sec	3.69 sec	32.95 sec
48KB cache	0.36 sec	3.34 sec	30.03 sec

Fig. 9 Performances of the full map method with the CRS format

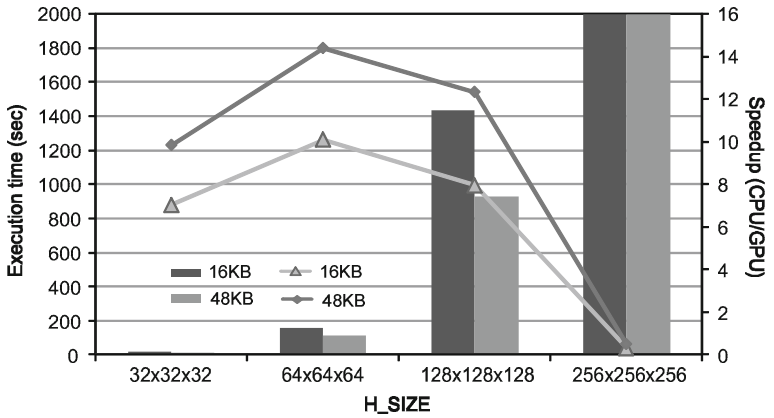
performance. On the other hand, the execution time become drastically small. Therefore, we have confirmed that the CRS format is indispensable technique for speeding up the KPM. The full map method with the CRS format can executes the case with H_SIZE of $32 \times 32 \times 32$ because the required amount of memory is reduced.

Although the problem size seems too small for GPU, it is not able to become larger than $32 \times 32 \times 32$ because the required amount of memory explosively increases, and becomes larger than 3GBytes (i.e. actual memory resource of C2050). Therefore, it is very important for the KPM algorithm on GPU to reduce the required amount of memory. Thus, we can expect that the sliding window method reduces the require amount of memory, and increases the available problem size.

4.4 Evaluation for the Sliding Window Method with the CRS Format

We have confirmed that the CRS format is very effective for the performance in the previous section. Therefore, this section additionally applies the format to the sliding window method, which can increase the problem size drastically due to reduction of the required amount of memory for summation for $\tilde{\mu}_i$. Applying large size H_SIZE, we can expect that the KPM is fully parallelized on the stream processors and the method will extract the potential high performance of GPU.

Figure 10 shows the performances and the speedups of the sliding window method applying the CRS format. Although the sliding window method includes many control code to reduce $\tilde{\mu}_i$ to the window memory, it achieves better performance than the full map method comparing the execution times of $H\_SIZE = 32 \times 32 \times 32$. The sliding



H_SIZE	32x32x32	64x64x64	128x128x128	256x256x256
16KB cache	18 sec	157 sec	1435 sec	307499 sec
48KB cache	13 sec	110 sec	926 sec	103122 sec

Fig. 10 Performances of the sliding window method with the CRS format comparing among 16 KByte/48 KByte cache and the speedup

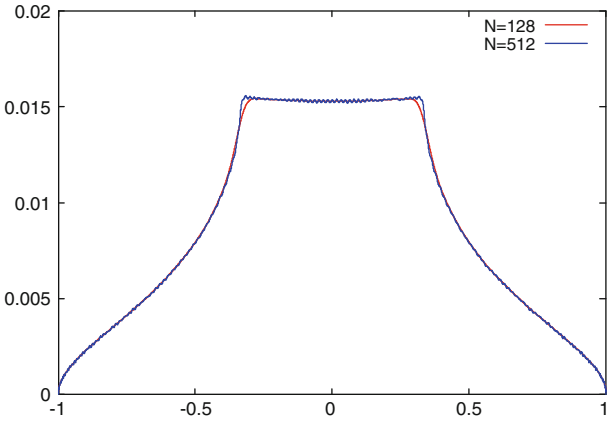


Fig. 11 The DOS comparison with truncation between N = 128 and N = 512 when the lattice is 128 × 128 × 128, $R = 14$ and $S = 128$

window method has an advantage in the required amount of memory. Therefore, it can accept the problem size of $H\_SIZE = 128 \times 128 \times 128$.

The speedup reaches about 14.4 times due to the optimized memory access that only occurs after accumulating a part of $\tilde{\mu}_n$ into a register. This is reasonable to use GPU as the advanced processing platform for the KPM because the speedup is larger than the maximum number of cores in the recent CPU (i.e. eight on such as Corei7). Thus, we have confirmed that the sliding window method invokes the KPM effectively although it includes much larger control code than the full map method.

Applied the sliding window method with $H\_SIZE = 128 \times 128 \times 128$, Fig. 11 plots two DOS data combinations when $N = 128$ and $N = 512$. When N is the smaller number, the truncation reduces to the resolution of the DOS.

4.5 Discussion

Let us consider additional possibilities to improve the performance of the sliding window method. The sliding window method can execute the problem size when $H\_SIZE = 256 \times 256 \times 256$. However, as seen in Fig. 10, the speedup degrades very much because to reduce the required amount of memory the experimental case needed to decrease the $Number\ of\ Blocks$ to one. Therefore, only a thread block is working for all KPM operations, and others are idle. To avoid the decrease of the number of active thread blocks, we finally propose another technique to reduce the required amount of memory.

The part where the sliding window method consumes memory at most is the operation (1) in Fig. 6 calculated with the memory of $Number\ of\ Blocks \times H\_SIZE \times 4 \times 8$ bytes as discussed in Sect. 3.2.1. To reduce the required amount of memory for the operation (1), we propose a technique that divides the operations to multiple kernel programs and calls the kernels for many times from the CPU side. Here, we divide the operations into three kernels as shown in Fig. 12a; the createR randomly generates $\overrightarrow{r}$, the createRn calculates $\overrightarrow{r_n}$ using $\overrightarrow{r_{n-2}}$, $\overrightarrow{r_{n-1}}$ and $\overrightarrow{r}$ and also saves $\overrightarrow{r_n} \cdot \overrightarrow{r}$ into

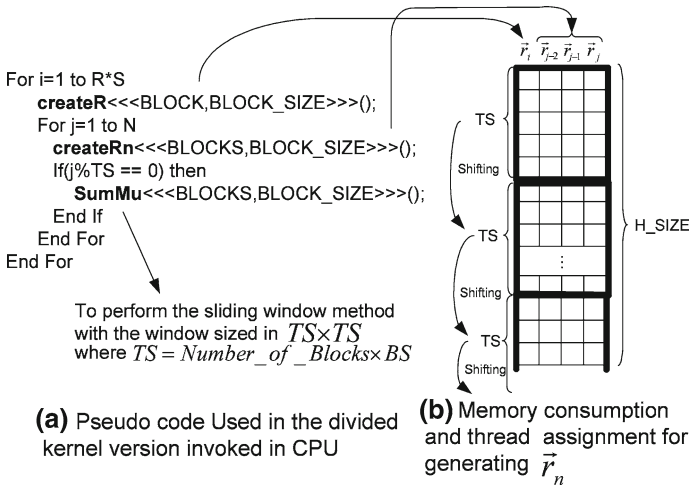
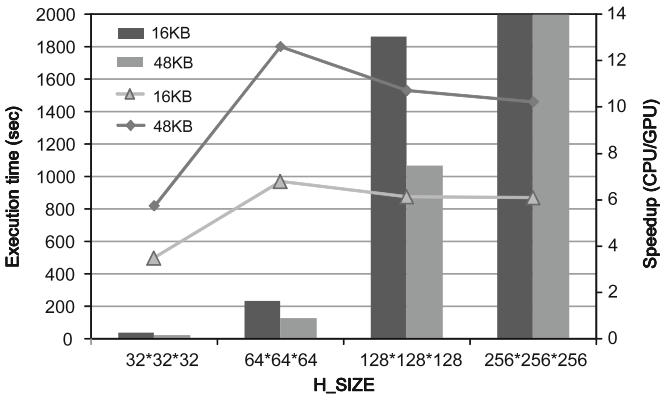


Fig. 12 Implementation of the divided kernel version

the window memory. Finally the SumMu performs the operation (2) in the sliding window method for μ_N.

The createR and createRn kernels share a single memory block sized in $H\_SIZE \times 4 \times 8$ bytes for recursive calculation. The elements of the $\overrightarrow{r}$ s are calculated respectively in parallel based on the thread size ($TS = Numberof\,Blocks \times BS$) as illustrated in Fig. 12. The createR is called once per N time iterations of createRn. The createRn kernel also saves a part of μ_i into the sliding window used in the SumMu kernel. Therefore, every TS times of calling createRn kernel the SumMu kernel is invoked because the window size is $TS \times TS$. In the other word, all threads are working concurrently for calculating μ_i in the SumMu kernel. Therefore the maximum number of parallelism of createR and createRn corresponds to the number of threads that do not related to the required amount of memory. Finally, the SumMu kernel needs a size of window memory of $TS \times TS \times 8$ Bytes. Thus, divided a kernel program to several small ones, the required amount of memory can be reduced because the total memory size does not have any relationship to the number of thread blocks as seen in Fig. 6(1) and the parallelism becomes available to be controlled flexibly.

Because every kernel must be loaded into GPU's instruction memory before the execution, the divided kernels contain the potential overhead caused by loading and discarding every kernel execution at the iteration. The overhead should significantly degrade the performance when the problem size is small. We have measured the performances of the divided kernel version with the sliding window method varying H_SIZE from $32 \times 32 \times 32$ to $256 \times 256 \times 256$ as shown in Fig. 13 when 32 thread blocks and 128 BS are used for all kernels. Therefore, the total number of threads working concurrently is 4,096. The sliding window is $4,096 \times 4,096$. The data cache size is also exchanged between 16 and 48 Kbytes. As we have expected, the overhead for loading/discarding the kernel to/from GPU causes performance degradation.



H_SIZE	32x32x32	64x64x64	128x128x128	256x256x256
16KB cache	37 sec	235 sec	1862 sec	14994 sec
48KB cache	23 sec	126 sec	1067 sec	8941 sec

Fig. 13 Performances and speedup of the divided sliding window method with the CRS format with 16 KByte/48 KByte data cache

However, when H_SIZE is $256 \times 256 \times 256$, the performance has become very much better than the one of the single kernel version because different parallelism is applied to each kernel. Thus, according to the performances in the graphs, we can conclude that when H_SIZE is less than $256 \times 256 \times 256$, we should employ the single kernel version. If H_SIZE is larger than it, the divided kernel version must be selected.

Let us discuss the speedups among the different implementations proposed in this paper. Figure 14 shows the comparisons from different performance aspects. Regarding the effect of data cache size illustrated in (a), (c) and (e), any implementation achieves a performance improvement from 1.10 to 1.65 times. On the other hand, the effect of the CRS format shown in (b) is remarkable because it achieves a drastic performance improvement both in CPU and GPU versions. The speedup increases as

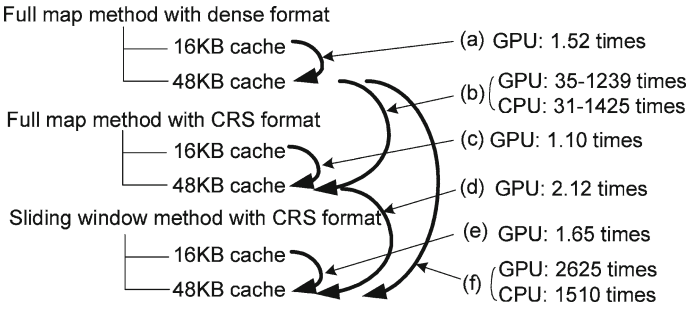


Fig. 14 Performance comparisons between each implemetaion of KPM on GPU and CPU

the problem size increases. Therefore, it is very clear that the CRS format has a large performance impact when H_SIZE becomes very large.

According to the performance improvements between the algorithms shown in (d), the sliding window method is about two times faster than the full map method. Although this comparison is performed only with the available H_SIZE measured in both algorithms (i.e. $32 \times 32 \times 32$), the performance improvement increases if the comparison is performed with a larger H_SIZE. Thus, the overall performance improvement from the full map method with a dense H matrix to the sliding window method with a sparse one has become about 2,600 times. Additionally, using the best algorithm, we are able to simulate a lattice of $256 \times 256 \times 256$ in a PC with a GPU by about 10 times shorter simulation time than the CPU-based implementation in a single thread. Therefore, we expect that the GPU-based implementation will achieve much higher performance than the CPU-based one due to the higher integration ratio of the stream processors than the one of CPU cores.

Here, let us focus on a numerical comparison of the performances calculating FLOPS values among the divided version on GPU and the CPU one. The parts of the double precision floating point operations correspond to a) the random $\overrightarrow{r}$ generation performed in the step (1), b) the recursive $\overrightarrow{r}_n$ generation done in the step (2.1), c) the vector dot products for $\tilde{\mu}_i$ done in the step (2.2) and finally d) the μ generation done in the step (3). The a) occupies a very few percentage in the total execution time using the CUDA's random number generation library. We just estimate the part takes $H\_SIZE \times w$ FLOPS where $w \ll RSN$. The w is actually very small number. The b) needs $H\_SIZE \times 14 \times R \times S$ for calculating $H\overrightarrow{r}_0$, $H\_SIZE \times 16 \times R \times S \times (N-2)$ for doing $H \cdot \overrightarrow{r}_{n-1} - \overrightarrow{r}_{n-2}$ that consists of $H\_SIZE \times 8 \times R \times S \times (N-2)$ multiplications and $H\_SIZE \times 7 \times R \times S \times (N-2)$ summations in an $H \cdot \overrightarrow{r}_i$ and $H\_SIZE \times R \times S \times (N-2)$ subtractions in the vector subtraction of $\overrightarrow{r}_i$ respectively. The c) performs $H\_SIZE \times R \times S \times N$ time multiplications and $H\_SIZE \times R \times S \times N$ summations. Finally the d) needs RS summations plus a division. Totally the FLOPS in the KPM is calculated by;

$$\begin{aligned} & H\_SIZE \times w + 18 \times H\_SIZE \times R \times S \times (N-1) + R \times S + 1 \\ & \qquad \because N \gg 1 \\ & \approx 18 \times H\_SIZE \times R \times S \times N \end{aligned} \tag{20}$$

To discuss the actual performances of the divided version on GPU and the CPU one, we apply the actual parameters to the equation above. In the case of the GPU version (i.e. where $N = 128$, $R = 14$ and $S = 128$ are applied for the parameters), the performance becomes 7.7 GFLOPS on Tesla C2050 when we apply $H\_SIZE = 256 \times 256 \times 256$. Because the peak performance of C2050 based on double precision floating point operations is 500 GFLOPS, the KPM on GPU achieves about 1.5 % of potential GPU performance. On the other hand, using the same parameters, we can calculate the performance of the CPU version as well that obtains 0.9 GFLOPS. The performance achieves 3.0 % of the peak performance of the Core i7 processor. All in all, we conclude that the KPM is a typical I/O intensive application for the computing hardware. Especially, GPU needs to perform a

large number of random accesses to large area of the global memory. Thus, the key technique that improves the entire performance of the KPM is to reduce the number of I/O operations for reading/writing the compressed sparse matrix in the global memory.

5 Further Performance Extensions

Let us consider additional performance extensions focusing on further reducing required memory for KPM. It is important to consider the I/O bandwidth during the KPM calculation because the part (2) in Fig. 5 occupies 83 % of the total execution time when the divided kernel version is applied resulting in the previous section. As the amount of required memory is reduced, the utilization ratio of I/O bus for GPU memory is also increased because redundant memory access is eliminated from the overall calculation of KPM. Thus the I/O bandwidth during the calculation of KPM will be increased applying the more compact memory usage.

Here, we will apply two optimization techniques. The first consideration for the extension is to optimize access pattern recycling the memory area that will not be used again during the same calculation phase. Moreover we also optimize the access pattern considering the memory organization of GPU system. This optimization promises to reduce the number of memory I/O and increase the total I/O bandwidth of KPM. Another optimization performs dedicated optimization to the application of KPM, which targets to the case of the 3D cubic model of the condensed matter physics. We discuss the performance increase from these two optimization techniques in the following sections.

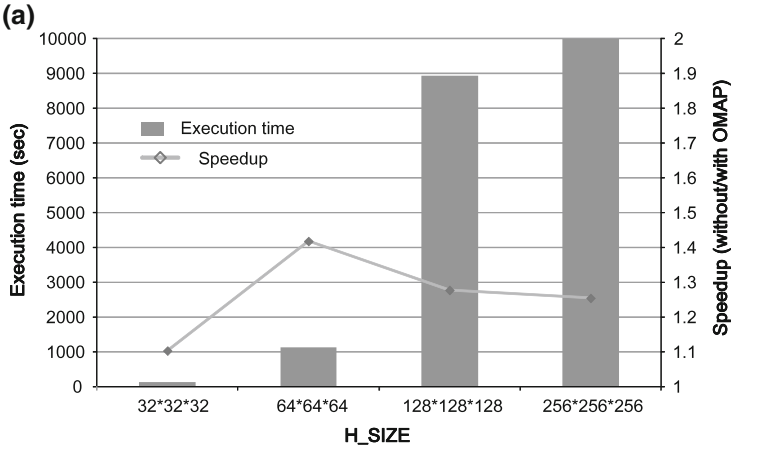
5.1 Optimization on Memory Access Pattern

The part (2) in Fig. 5 includes the recursive calculation of $\overrightarrow{r}_n$. The divided version with CRS format saves all the $\overrightarrow{r}_i$ vectors on GPU's memory. However, the $\overrightarrow{r}_{n-2}$ is not used after it is used once for the recursive calculation. Therefore, to reduce memory usage, the $\overrightarrow{r}_n$ can be saved in the memory region of $\overrightarrow{r}_{n-2}$. Therefore, we modify the equation for the calcularion to:

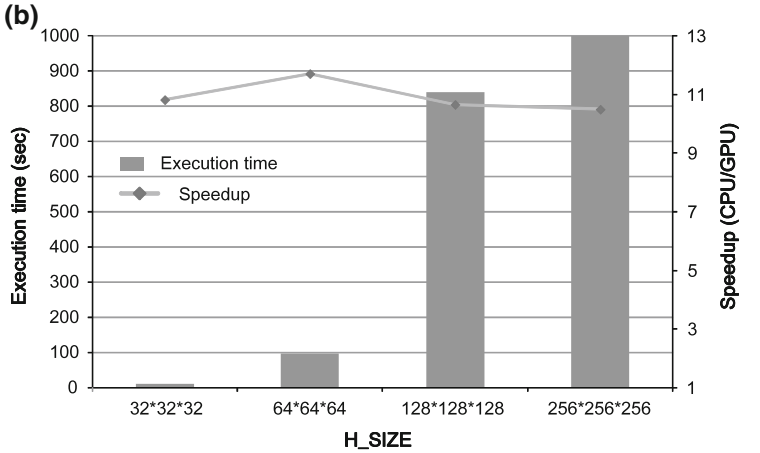
$$\overrightarrow{r}_{n-2} = H\overrightarrow{r}_{n-1} \times \overrightarrow{r}_{n-2}$$

This memory recycling technique reduces $H\_SIZE \times 8$ bytes from the required memory amount in the case of the divided version with CRS format.

Moreover, we place the matrix A of CRS format for H in the column major order on the GPU's memory. Due to the burst transfer to the cache memory on GPU, massive data in some elements in the memory is transferred at an access. Therefore, accessing to the column elements of A consumed by the multiplication with $\overrightarrow{r}_{n-1}$ hit to the cache memory on the GPU. Thus, each thread will use memory I/O bandwidth of GPU effectively because the recursive $\overrightarrow{r}_{n-2}$ calculation will always access to the continuous memory region.



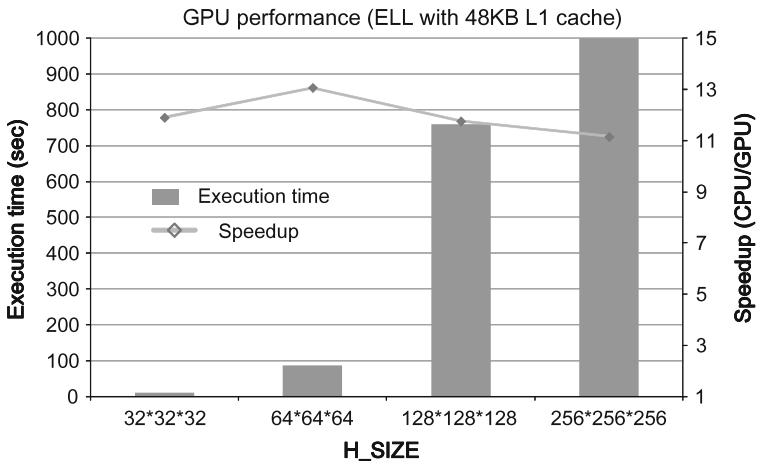
H_SIZE	32x32x32	64x64x64	128x128x128	256x256x256
CPU without OMAP	131 sec	1593 sec	11417 sec	92041 sec
CPU with OMAP	119 sec	1125 sec	8940 sec	73382 sec



H_SIZE	32x32x32	64x64x64	128x128x128	256x256x256
GPU 48KB cache	11 sec	96 sec	839 sec	6994 sec

Fig. 15 Performance comparisons with OMAP **a** CPU performance (OMAP); **b** GPU performance (OMAP with 48KB L1 cache)

Applying the techniques for optimizing memory access patterns (in short, we call it OMAP) mentioned above, we have measured the performance on GPU and CPU. Figure 15 shows (a) performance of CPU version and (b) the one of GPU version with 48KB L1 cache with OMAP. The performance of CPU version is affected by the OMAP technique because $\overrightarrow{r_i}$ is placed in a localized and continuous memory area and the cache memory is effectively used for the recursive calculation. The speedup



H_SIZE	32x32x32	64x64x64	128x128x128	256x256x256
GPU 48KB cache	10 sec	86 sec	760 sec	6581 sec

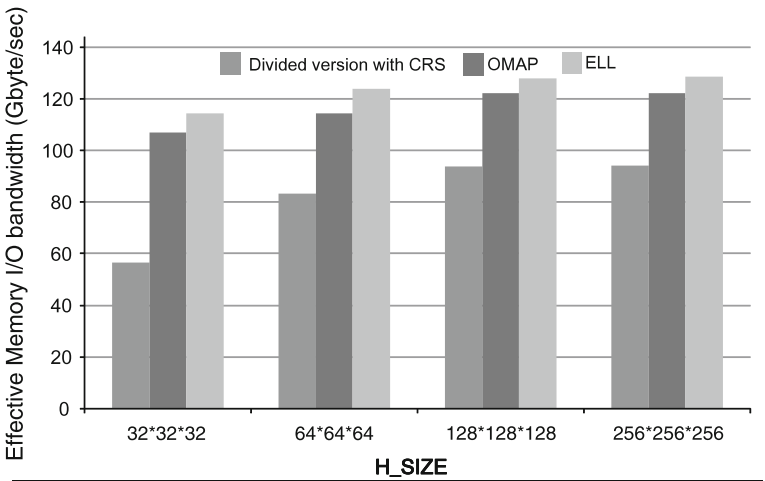
Fig. 16 Performance of ELL compression

is about 30 % comparing the performance without OMAP. On the other hand, the performance of GPU version also shows about 30 % better result comparing to the one without OMAP. The speedup from the CPU version with OMAP has become 10.5 times. Moreover, the total performance of KPM has result about 10 GFLOPS. Thus the OMAP techniques are effective for both CPU and GPU implementations because the memory access is optimized to utilize the cache memory.

5.2 Optimization Dedicated to 3D Cubic Lattice Model

According to the mathematical property of KPM, it is available to apply it to solving eigenvalue of any matrix defined by models of natural phenomenon. When the matrix is sparse, to reduce memory size for the matrix, CRS format is suitable for compressing it. On the other hand, if we limit to use KPM in the condensed matter physics, especially to the simulation for a 3D cubic model, it is clear that ELL format fits to this case because the compression ratio becomes better than CRS format as we discussed in Sect. 2.3.

Due to the smaller required memory size compressed by ELL format, both CPU and GPU versions have the benefit to the performances. Figure 16(a), (b) shows the performances on GPU and CPU versions respectively. The GPU version has been improved for about 10 % than the one with CRS format. Moreover, the CPU version has the improvement from the performance with CRS format for 25 %. Comparing the GPU version with the CPU one, the performance speedup is finally about 11.5 times. Thus, ELL format works as more effective compression method for reducing required memory amount and accelerating performance if the sparse matrix has fixed elements in each row because the 3D cubic case has 7 elements in any row in the matrix.



H_SIZE	32x32x32	64x64x64	128x128x128	256x256x256
Divided with CRS	56	83	93	94
OMAP	106	114	122	122
ELL	114	123	128	128

Fig. 17 Comparison of memory bandwidth

5.3 Performance Consideration for Additional Extensions

Using OMAP and ELL format, we have improved the performance of KPM from 7.7 GFLOPS without those techniques to 11.5 GFLOPS. During the improvements, we have focused on the memory bandwidth measured by the NVIDIA's performance profiler called *computeprof*. Figure 17 depicts a graph of the bandwidth changes among the performances of the divided versions with CRS format, with OMAP only and with OMAP and ELL format. The OMAP technique shows a large effect to achieve about 40 % higher memory bandwidth than the divided version with CRS format. This is caused by contiguous memory assignment for $\overrightarrow{r_i}$ vector that accelerates to utilize the cache memory on GPU. ELL format also accelerates the effective use of the cache memory improving the memory bandwidth for about 5 % of the one of the OMAP version. Finally, our KPM achieves 128 GByte/sec for the effective memory bandwidth during the calculation. The bandwidth has become 89 % of the peak memory I/O speed (i.e. 144 GByte/s) of the GPU. Thus, we can conclude that it is available to improve performance applying special condition dedicated to the lattice model simulation of the H matrix to KPM and the improvement is related to increasing the effective memory bandwidth during the calculation. Finally we have achieved almost the peak performance that can be the fastest performance in our environment using the Tesla C2050 GPU.

6 Concluding Remarks

Focusing on the KPM used in the simulations for strong lattice correlation model in condensed matter physics, this paper proposed methods of GPU-based implementations for the KPM. Applying typical architectural optimizations on GPU, the sliding window method has achieved up to 14.4 times better performance than the CPU-based implementation, and also is able to perform the simulation for $256 \times 256 \times 256$ lattice model at the maximum size on NVIDIA's Tesla C2050.

Moreover, under a condition used in the simulation for a 3D cubic correlation lattice model, KPM includes availability to be improved by memory accessing optimization and ELL format for sparse matrix compression. We have confirmed that the GPU performance is improved for up to 30 % using these optimizations. We have also considered the memory bandwidth on the GPU under the improved performance. The bandwidth reaches almost peak of the memory interface on the Tesla C2050. Thus, we have implemented KPM algorithm on the GPU that uses fully hardware resources.

For the future plans, we are now considering two aspects; performance acceleration in the cluster environment and the physics applications. Regarding the former plan, the KPM can be embarrassingly parallelized into multiple GPUs following the coarse grain parallelism of the parameters R, S. Therefore, we are now trying to parallelize the sliding window method using MPI and OpenMP. On the other hand, regarding the physics applications, we are now planning to actually simulate a large lattice model to find unknown state of materials using the techniques in this paper.

Acknowledgments This work is partially supported by the Japan Science Technology Agency (JST) PRESTO program. And also this work is partially supported by KAKENHI (24300020) Grant-in-Aid for Scientific Research (B).

References

1. Bednorz, J.G., Müller, K.A.: Possible high T_c superconductivity in the Ba-La-Cu-O system. Z. Phys. B Condens. Matter **64**(2), 189–193 (1986)
2. Dagotto, E.: Correlated electrons in high-temperature superconductors. Rev. Mod. Phys. **66**(3), 763–840 (1994)
3. Ferrario, M., Ciccotti, G., Binder, K.: Computater Simulations in Condensed Matter: From Materials to Chemical Biology, vol. 1, 2. Springer, Berlin (2006)
4. Foulkes, W., Mitas, L., Needs, R., Rajagopal, G.: Quantum monte carlo simulations of solids. Rev. Mod. Phys. **73**(1), 33–83 (2001)
5. Grimes, R., Kincaid, D., Young., D.: ITPACK 2.0 User's Guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas (1979)
6. Grotendorst, J., Mark, D., Muramatsu, A.: Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms. NIC-Directors (2002)
7. McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (1995)
8. Nguyen, H.: GPU Gems 3, 1st edn. Addison-Wesley Professional, Reading (2007)
9. NVIDIA Corporation: CUDA: Compute unified device architecture programming guide. http://developer.nvidia.com/cuda
10. Ohno, K., Esfarjani, K., Kawazoe, Y.: Computational Materials Science. Springer, Berlin (1999)
11. Schollwöck, U.: The density-matrix renormalization group. Rev. Mod. Phys. **77**(1), 259–315 (2005)
12. Varga, R.: Geršgorin and His Circles. Springer Series in Computational Mathematics. Springer, Berlin (2004)

13. Weiße, A., Wellein, G., Alvermann, A., Fehske, H.: The kernel polynomial method. Rev. Mod. Phys. **78**(1), 275–306 (2006)
14. White, S.: Density matrix formulation for quantum renormalization groups. Phys. Rev. Lett. **69**(19), 2863–2866 (1992)
15. White, S.: Density-matrix algorithms for quantum renormalization groups. Phys. Rev. B **48**(14), 10345–10356 (1993)
16. Yamada, S., Okumura, M., Machida, M.: Direct extension of density-matrix renormalization group to two-dimensional quantum lattice systems: studies of parallel algorithm, accuracy, and performance. J. Phys. Soc. Jpn. **78**(9), 094004 (2009)
17. Yamashita, M., Nakata, N., Senshu, Y., Nagata, M., Yamamoto, H.M., Kato, R., Shibauchi, T., Matsuda, Y.: Highly mobile gapless excitations in a two-dimensional candidate quantum spin liquid. Science **328**(5983), 1246–1248 (2010)
18. Zhang, S., Yamagiwa, S., Okumura, M., Yunoki, S.: Performance accelaration of kernel polynomial method applying graphics processing units. In: IPDPS/APDCM 2011, pp. 564–571. IEEE CS (2011)