International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 2, Number 1, pages 41–55, January 2012

Parallelizing Kernel Polynomial Method Applying Graphics Processing Units

Shixun Zhang, Shinichi Yamagiwa
School of Information, Kochi University of Technology/JST PRESTO
Kami, Kochi 782-8502 Japan

Masahiko Okumura, Seiji Yunoki
Computational Condensed Matter Physics Laboratory
RIKEN ASI
Wako, Saitama, 351-0198 Japan,
JST CREST
Kawaguchi, Saitama 332–0012, Japan, and
Computational Materials Science Research Team
RIKEN AICS
Kobe, Hyogo, 650-0047 Japan

Received: July 23, 2011
Revised: October 28, 2011
Accepted: December 15, 2011
Communicated by Akihiro Fujiwara

Abstract

The Kernel Polynomial Method (KPM) is one of the fast diagonalization methods used for simulations of quantum systems in research fields of condensed matter physics and chemistry. The algorithm has a difficulty to be parallelized on a cluster computer or a supercomputer due to the fine-grain recursive calculations. This paper proposes an implementation of the KPM on the recent graphics processing units (GPU) where the recursive calculations are able to be parallelized in the massively parallel environment. This paper also describes performance evaluations regarding the cases when the actual simulation parameters are applied, where one parameter is applied for the increased intensive calculations and another is applied for the increased amount of memory usage. Moreover, the impact for applying the Compress Row Storage (CRS) format to the KPM algorithm is also discussed. Finally, it concludes that the performance on the GPU promises very high performance compared to the one on CPU and reduces the overall simulation time.

Keywords: GPGPU, Kernel Polynomial Method, Condensed Matter Physics, CUDA, CRS

1 Introduction

Today's technological achievement in our everyday life is based on years of fundamental research for a wide variety of materials with fascinating functionalities such as semiconductors, magnets, and superconductors. Researchers in condensed matter physics revealed long ago that those different properties of materials result from different behaviors of electrons, which are described by quantum

mechanical equation of motion. Although it has been more than 80 years since quantum mechanics was established, there are still many properties of matters whose origins are yet to be understood. Such examples include copper based high temperature superconductors [2] and some of magnetic insulators of organic compounds [13]. The common feature of these systems is a strong quantum correlation between electrons, which is turned out to be crucial for determining their properties. It is precisely this strong correlation that makes it difficult to treat these systems analytically without introducing any bias in theory.

The best way to treat the strong quantum correlations is to solve quantum mechanical equation of motion numerically exactly. Because of exponential increase of degrees of freedom with the number of electrons $\sim O(10^{23})$, we must still resort some sort of approximations. However, unlike analytical treatments, numerical simulations can handle the strong correlation effects with controllable approximations. Among many, well established numerical methods thus far are exact diagonalization method [3, 11], quantum Monte Carlo method [4], density-matrix renormalization group method [9, 10, 7, 12], and kernel polynomial method (KPM) [8]. Each method is suited to particular sets of problems and at the same time each has severe limitations. For instance, the exact diagonalization method is able to evaluate the ground state (and low energy excited states) in high accuracy, but it is limited to a small size of 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 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 in turn controls the accuracy of the approximation. Thus, this paper focuses on the KPM which appropriately evaluates the DoS and Green's function including higher energy excited states [8].

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 due to truncated 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 [8]. Moreover, in quantum statistical mechanics, it is required to evaluate the trace of large-dimensional Hamiltonian matrices. This trace is efficiently approximated by using random vectors [8] (we call it "stochastic trace method" in this paper). Therefore, combining these two methods, truncated polynomial expansions and random vector bases, allows us to evaluate the DoS and other physical quantities with significantly reduced complexity.

The computational cost inevitably increases with system sizes considered, 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 parallel platform.

Regarding computer hardware, the graphics processing units (GPU) have become available to be used for acceleration platform as a substitute of CPU. This is due to the recent drastic performance growth of GPU. The recent GPU has already achieved the performance up to TFLOPS order. Therefore, it is applied to various scientific fields to solve the grand challenge applications under a personal computing environment [6].

The program on the GPU is called stream-based program which processes each data unit contained in input data streams, and generates the corresponding data unit forming output data streams. This computing style has benefits of 1) eliminating memory access bottleneck, which is seen in the von Neumann style architecture, and 2) data parallelism because each data unit does not have any dependency in the data streams. The recent challenges to speedup intensive computations enforce algorithms to be redesigned to fit to GPU and to receive the benefit of especially the data parallelism characteristics assigning small operations to the enormous number of the stream processors. This computing style would become a typical computing style in the next supercomputing generation.

This paper focuses on a GPU-based implementation of the KPM applying the stream-based computing style [14]. We propose an effective implementation of the KPM on the GPU to accelerate its

performance faster than the recent CPU. As seen 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. Applying GPU resources and a stream-based programming style, this paper will challenge to overcome the performance limitation caused by the recursive operation. In addition, KPM has intensive matrix operations such as matrix-vector multiplication in which the format of the matrix has a great impact on the performance and memory consumption. Therefore, we will discuss the impact of applying a compressed format to the matrix in the algorithm.

This paper is organized as follows. Section 2 describes the detailed explanation of KPM and the overview of the general purpose computing on the GPU. Section 3 proposes the design and implementation of KPM on the GPU. Section 4 analyzes the performances of typical sets of input parameters used in condensed matter physics and discusses the program behaviors when the parameters change to increase resource usage regarding processor and memory. Finally, section 5 concludes this paper.

2 Background and definitions

2.1 Kernel polynomial method

2.1.1 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 N, also known as *truncation number*, defines the number of Chebyshev polynomials and controls the accuracy of the function. The additional coefficients g_n is given by a kernel function to eliminate Gibbs effect.

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

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

2.1.2 Application to quantum systems

In quantum physics, we need to expand functions of the Hamiltonian matrix. In this paper, we focus on the density of state (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.

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 k-th 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 k-th eigenvector and $\langle k| = |k\rangle^{\dagger}$.

2.1.3 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,

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

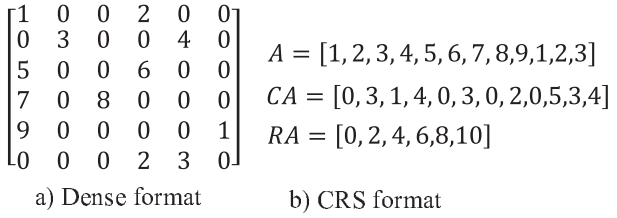


Figure 1: Compressed Row Storage format

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 Compressed Row Storage format

Compressed Row Storage format (CRS) is a well-known matrix compression technique that significantly reduces memory usage needed to store sparse matrices. Figure 1 a) shows a 6×6 matrix that contains several zero elements. Figure 1 b) shows the matrix in CRS format. In order to store the matrix in CRS format, three arrays are needed: A, CA, RA. A stores the value of the non-zero elements of the matrix, CA stores the column indices of the non-zero elements in A, RA stores the locations in array A that indicate each index for the corresponding to the next row. In Figure 1 example, CRS format reduces about 19% memory usage when the elements are stored in 4 bytes integer.

The number of non-zero elements in Hamiltonian matrix depends on the physical lattice model. In a 3D cubic lattice model for double exchange[1] simulation, each site has six nearest neighbours, adding the on-site potential, we have seven non-zero elements per row. Thus, the total number of the non-zero element is $H\_SIZE \times 7$, where the H_SIZE equals to D mentioned in section 2.1. Since the elements are stored in double floating point format to achieve good accuracy of the result, the memory needed to store the vector A becomes $H\_SIZE \times 7 \times 8$ bytes. The memory for the vector CA is $H\_SIZE \times 7 \times 4$ bytes when CA is stored in 4 bytes integers. The memory for RA is $H\_SIZE \times 4$ bytes. By adding together the memory of the three vectors, the total memory usage for storing a Hamiltonian matrix in the CRS format can be obtained as:

$$H\_SIZE \times 88 \tag{20}$$

bytes. Comparing with the memory usage in the case of the dense format, the compression ratio of the CRS format can be presented as:

$$ratio = \frac{11}{H\_SIZE} \tag{21}$$

which shows that the larger the matrix size is, the higher compression ratio is obtained.

2.2.1 Numerical complexity

The numerical complexity of the KPM is $O(RSND)$ 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

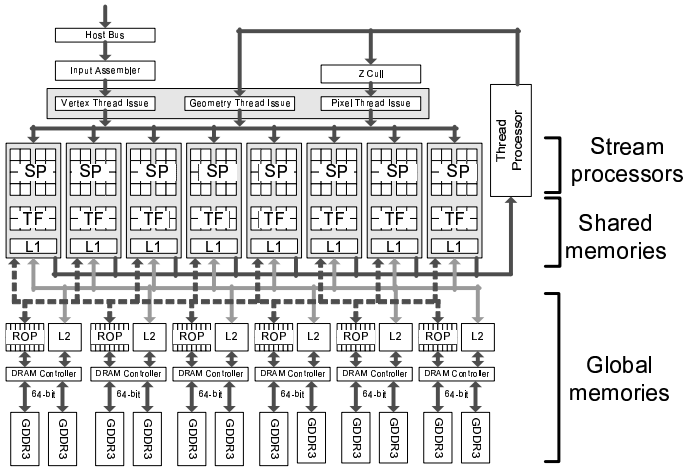


Figure 2: A GPU architecture.

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 $O(RS)$ comes from the average and summation in Eq. (19) and $O(N)$ from the recursive iteration in Eq. (18). This numerical cost $O(RSND)$ is very effective against the full diagonalization which costs $O(D^3)$ if $S, R, N \ll D^2$, and the $\tilde{H}$ is a sparse matrix. However, for a dense matrix, the numerical cost becomes $O(RSND^2)$ 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 for a sparse matrix. This paper not only discusses the case when the CRS format is not applied to the memory maintenance for the $\tilde{H}$ but also the performance and memory impact when the CRS is applied to the matrix.

2.3 General purpose computing on the GPUs

2.3.1 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 help a part of visualization tasks in the system. To utilize the GPU as a computing resource for GPGPU applications, the CPU downloads application program to the GPU's instruction memory and also prepares input data for the program. The 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.

The recent GPUs have only a kind of processor called the *stream processor*. The processor works for general purpose processes in any kind of calculation. However, the computing style must be followed in the stream-based one distributing elements included in streams into multiple stream processors. 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 cache.

2.3.2 CUDA

The Compute Unified Device Architecture (CUDA) has been proposed by NVIDIA Corporation [5]. The tools and APIs for programming on CUDA environment is now provided by the company's website.

The CUDA assumes an architecture model as illustrated in Figure 3 (a). The model defines a GPU which is connected to a CPU's peripheral bus. A VRAM (the global memory) that maintains

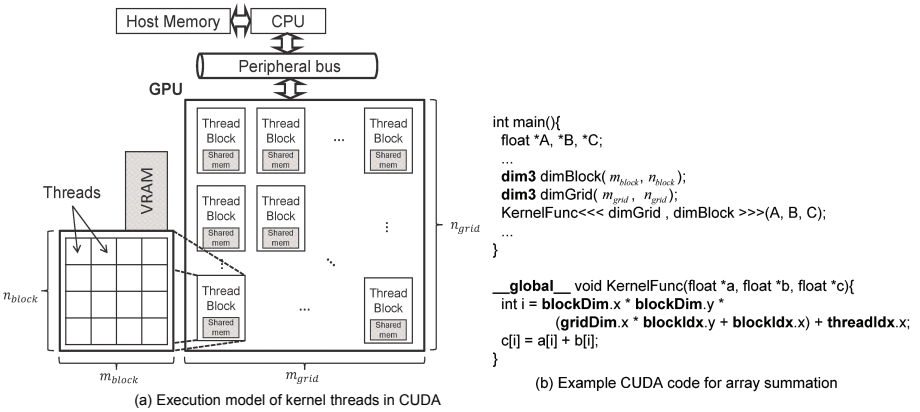


Figure 3: CUDA programming environment.

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. Each thread block has individual shared memory space where shared variables 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, obtaining a large parallelism, a large number of threads are invoked concurrently.

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 Figure 3 (b). 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 the 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 variables, Figure 3 (b) 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 and transparent interface for GPGPU.

According to the backgrounds we have mentioned above, it is important for the simulation in the quantum physics to apply a fast diagonalization method to reach the goal of the simulation quickly. However, the KPM has difficulty of fine-grain parallelization in large scale computers such as cluster computers or supercomputers due to the recursive calculation performed in the Eq. (18). Therefore, it is worth for us to implement the KPM on a GPU where the massively parallel environment is equipped with a large number of stream processors. Thus, this paper focuses on design and implementation of the KPM on the GPU that challenges to achieve the advanced performance with applying the stream-based computing style on the massively parallel environment.

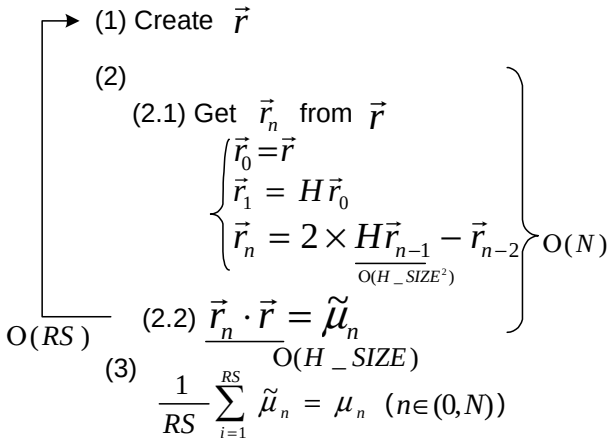


Figure 4: Design and implementaion of KPM on the GPUs.

3 Kernel polynomial method applying GPUs

3.1 Design for massively parallel platform

Figure 4 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 in section 2.1). 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 is very hard to parallelize using MPI or OpenMP because of the dependencies due to the recursive iteration although the part needs the most intensive calculation. Then a dot product is calculated using $\overrightarrow{r}_n$ again with $\overrightarrow{r}$ at the step (2.2) and generates $\tilde{\mu}_n$. Then the generation of the $\tilde{\mu}_n$ is iterated for RS times. This means each generation of $\tilde{\mu}_n$ can be massively parallelized on the GPUs. Finally, the average of all the $\tilde{\mu}_n$s is generated at the step (3). 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. This summation to generate $\tilde{\mu}_n$ can be parallelized on the GPU. Therefore, implemented on the GPUs, two parallel processing parts are entirely performed during the evaluation of the moments using KPM: a) generation of $\tilde{\mu}_n$ and b) generation of μ_n. The maximum number of parallelism at the both a) and b) parts becomes the RS because the total number of threads executed in the stream processors is RS. Here, GPU has an architectural restriction to the number of threads in a thread block referred as BLOCK_SIZE in this paper. Therefore, the number of thread blocks becomes $RS/BLOCK\_SIZE$. Considering the parallelization techniques above, let us explain the implementation of a kernel program on CUDA that invokes both the a) and b) parts.

3.2 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 for the parallelism.

3.2.1 Parallelization of calculations

As we discussed in the last section, two heavy calculation parts (the a) and the b)) should be parallelized and it would give the largest impact for the speedup.

Figure 5 a) shows the generation part for the $\overrightarrow{r}_n$. $\overrightarrow{r}_n$ needs $\overrightarrow{r}_{n-1}$, $\overrightarrow{r}_{n-2}$ and $\overrightarrow{r}$. The vector $\overrightarrow{r}$ is randomly generated using Mersenne Twister random generator in our implementation. These four vectors are obtained in the global memory and each block, which refers to the thread block in CUDA, will write those vectors swapping the pointers without data copy. Here the number of

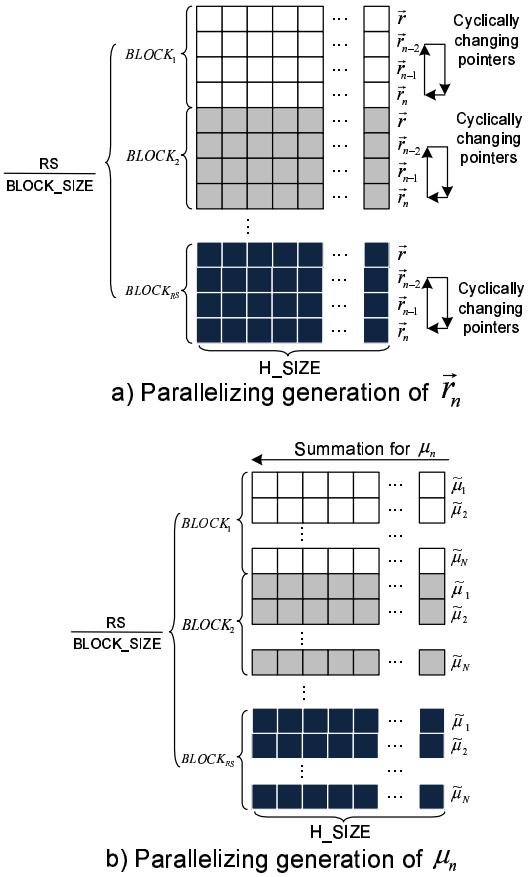


Figure 5: Parallelization of KPM.

blocks is $RS/BLOCK\_SIZE$. In each block, $BLOCK\_SIZE$ stream processors are concurrently working to generate a part of those vectors. For the generation of vector $\overrightarrow{r}$, the $i\text{-}th$ element $\overrightarrow{r}[i]$ is generated by the thread $i\%BS$, where the BS is the block size. For the vector $\overrightarrow{r}_n$, the generation of the $i\text{-}th$ element $\overrightarrow{r}_n[i]$ can be expressed:

$$\overrightarrow{r}_n[i] = 2 \times \sum_{k=0}^{D-1} H[i][k] \times \overrightarrow{r}_{n-1}[k] - \overrightarrow{r}_{n-2}[i], \tag{22}$$

where $D \equiv H\_SIZE$. Similarly, the $\overrightarrow{r}_n[i]$ is generated by the thread $i\%BS$. Therefore, this part will be fully parallelized into the total number of stream processors equipped on the GPUs. This part will generate $\tilde{\mu}_1$, $\tilde{\mu}_2$, ..., $\tilde{\mu}_N$ using $\overrightarrow{r}$ and $\overrightarrow{r}_n$ for N time iteration.

Figure 5 b) depicts the parallelization of generation for μ_n. It performs just parallel summations for generating a scalar μ_n where all thread blocks works in parallel.

3.2.2 Memory consumption

Let us consider the required memory amount for the operations in Figure 5 in the case of double precision. For the operation a), because four $\overrightarrow{r}$ vectors are stored in the global memory for a block. Each $\overrightarrow{r}$ vector has H_SIZE elements. Therefore, this part consumes $Number\ of\ Blocks \times 4 \times H\_SIZE \times 8$ bytes. The operation b) is parallelized into the number of blocks. Each block performs a part of summation using N $\tilde{\mu}$s. The length of $\tilde{\mu}$s is H_SIZE. Therefore, it needs totally $Number\ of\ Blocks \times N \times H\_SIZE \times 8$ bytes.

The operation a) writes $\tilde{\mu}_n$ into the global memory. This needs to be kept with $\overrightarrow{r}$ vectors simultaneously. Therefore, the total number of memory is $Number\ of\ Blocks \times H\_SIZE \times (8 \times N + 32)$.

Due to the recursive relationships among $\overrightarrow{r}_n$, $\overrightarrow{r}_{n-1}$ and $\overrightarrow{r}_{n-2}$, the KPM is treated generally as one of very hard parallelized algorithms. However, as we can see in this section, on the GPU, a massively parallel environment, the KPM is fully parallelized due to the stream-based computing concept. Thus, we can expect an effective speedup that will be proportional to the number of stream-processors.

4 Experimental performance analysis

This section shows performance evaluations of the KPM implemented on the GPU. The performance based on the GPU is compared with the one based on CPU. The experimental environment is a PC that consists of an Intel's Core i7 930 processor at 2.80GHz with 12GB DDR3 memory, and the NVIDIA Tesla C2050 with 448 streaming processors and 3GB Memory. The configuration of the cache in the GPU is set to 16KB. Therefore, the shared memory size is 48KB. The OS of the PC is the CentOS of the Linux Kernel 2.6.18. The driver version of the GPU is 3.0. All KPM calculations are performed with double precision floating point. The CPU version is compiled with GCC 4.4.1 with O3 option.

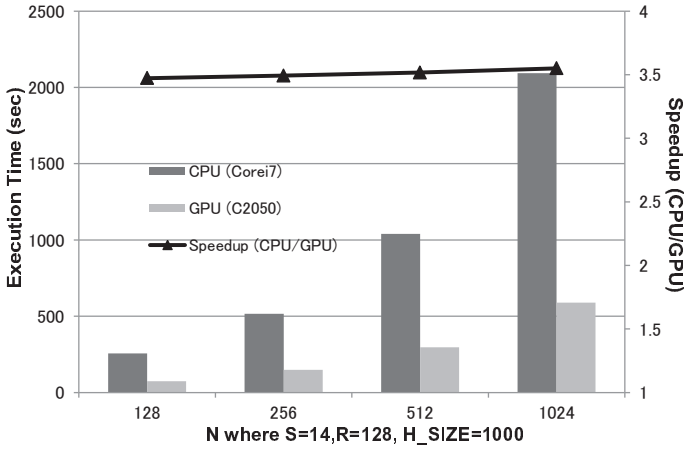
We perform four kinds of performance analysis: (1) evaluation using actual sets of parameters, (2) the one with increasing calculation size, (3) the one with increasing memory usage and finally (4) the one with applying the CRS format to Hamiltonian matrix. The first evaluation hires sets of parameters used in actual simulations of the meaningful model applied to the condensed matter physics field. The second evaluation analyses the behavior of the performances when the parameter N is increased. This means that more intensive calculation is loaded to the CPU and the GPU following the increase of N. The third one shows the performance impacts when the H_SIZE is increased. This case needs the square sized memory to store the H matrix that is increased by the impact of H_SIZE^2. The final evaluation shows the performance improvement of the CRS format compared with the dense one.

In the following performance evaluation the KPM parameters R and S are 14 and 128, respectively. For the GPU implementation, the block size is chosen 128, which gives the best performance in our experiment. Following the formula $RS/BLOCK\_SIZE$ introduced in section 3.2.1, the number of blocks is 14.

4.1 Performance analysis using actual simulation parameters

In the field of the computational condensed matter physics, the KPM is applied to a simulation to evaluate the DoS in a three dimensional lattice model. Let us consider a lattice model made of cubes in $10 \times 10 \times 10$ where an electron is placed in each corner. This model needs a Hamiltonian matrix sized in 1000×1000 due to the presentations of correlations among the electrons at each corner. The significant characteristics of the matrix include that 1) it is sparse and symmetric and 2) any row contains seven non-zero elements with the condition where all diagonal ones are zeros and the other non-zero ones are -1s. We evaluate the DoS in the case of the lattice that we assumed above using the fixed parameters of the KPM with $S = 14$ and $R = 128$. Varying N from 128 to 1024 in the steps of 2^n, Figure 6 shows the execution time and the speedup comparing the performances on the CPU with the ones on the GPU. The speedup keeps 3.5 times for all the cases.

We shall pickup two DoS data combinations from the parameter sets of Figure 6 and plot it to a graph as depicted in Figure 7. The graph shows the DoS when $N = 256$ and $N = 512$. When N is the smaller number, the truncation reduces to the resolution of the DoS. However, the processing time is smaller than the case of a large N. Therefore, although the case of $N = 512$ shows higher resolution of the DoS, it takes longer calculation time.



N	CPU (sec)	GPU (sec)	Speedup
128	256.16	73.77	3.47
256	516.51	147.89	3.49
512	1040.05	295.77	3.52
1024	2092.78	589.44	3.55

Figure 6: Performances applying the lattice made of cubes placed in 10x10x10.

4.2 Performance analysis with increased intensive calculations

Obtaining the fixed parameters of $H\_SIZE = 128$, $R = 14$ and $S = 128$, we measure the performances with varying the N from 128 to 2048. The graph of the performances is illustrated in Figure 8. The graph shows the execution time with bars and the speedups (i.e. the CPU time is divided by the corresponding GPU time) with a line. As increasing the N, that is, as increasing the calculation amount, the speedup increases to almost 4 times. This means that the performance with the higher intensive calculations affected by the larger N causes higher effective data parallelism on the GPU when the calculation amount is increased without changing the size of the memory usage. Thus, our implementation on the GPU clearly achieves higher performance than the CPU-based KPM as increasing the calculation amount.

4.3 Performance analysis with increased memory usage

This analysis fixes $N = 128$, $R = 14$ and $S = 128$. We vary H_SIZE from 512 to 4096 with the step of 2^n. The performance presents effects caused by increasing the memory usage. The graph of the performance is depicted in Figure 9. When the amount of memory usage increases, the number of memory accesses increases. Therefore, the CPU version needs to read/write the memory as increased the size of $\tilde{H}$ matrix. On the other hand, in the matrix-vector multiplication, the vector elements can be shared among the threads, so the performance will benefit from storing a part of the vector in the fast shared memory. Thus, the execution time of the GPU version does not increase more than the complexity ($O(H\_SIZE^2)$). This causes almost four times faster performance than the CPU version.

4.4 Performance comparison with/without CRS format

Applying the CRS format explained in section 2.2 to a Hamiltonian matrix, the memory usage and memory access can be greatly reduced due to Eq. 21, where H_SIZE usually is very large. It is shown in Figure 4 that matrix-vector multiplication is the most calculation intensive part. Thus, reduced memory access can significantly increase the performance as shown in Figure 10. In addition, the speedup increases when the H_SIZE becomes larger, which can be explained by Eq. 21 (i.e. the larger H_SIZE leads to higher compression ratio). Moreover, applying the CRS format, we

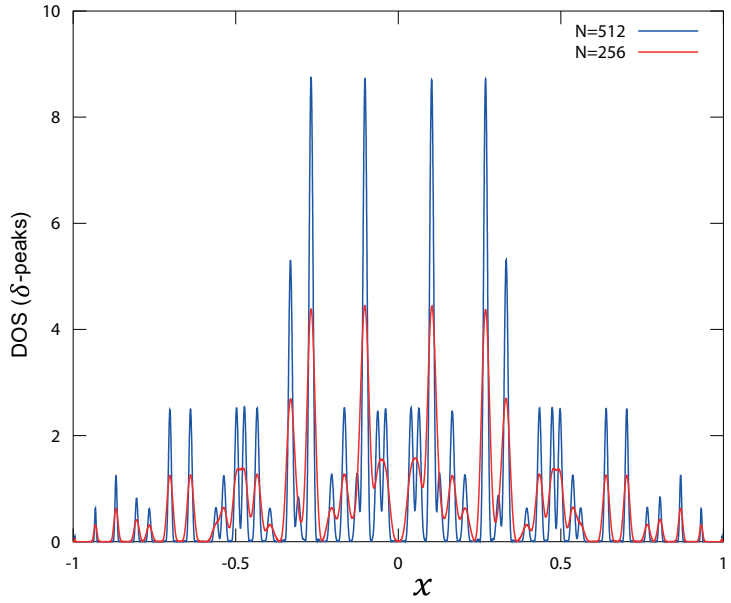
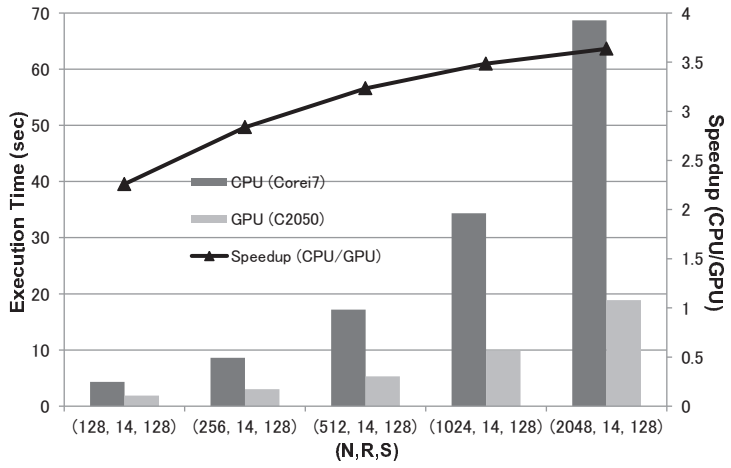
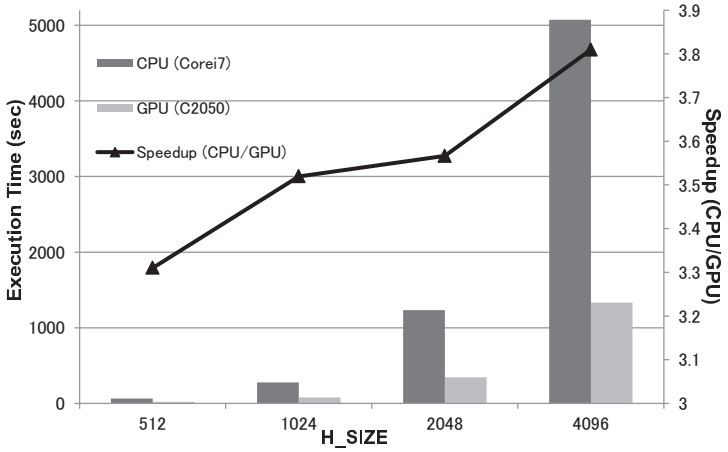


Figure 7: The DoS comparison with truncations between N=256 and N=512 when the lattice is made of cubes placed in 10x10x10, R=14 and S=128.



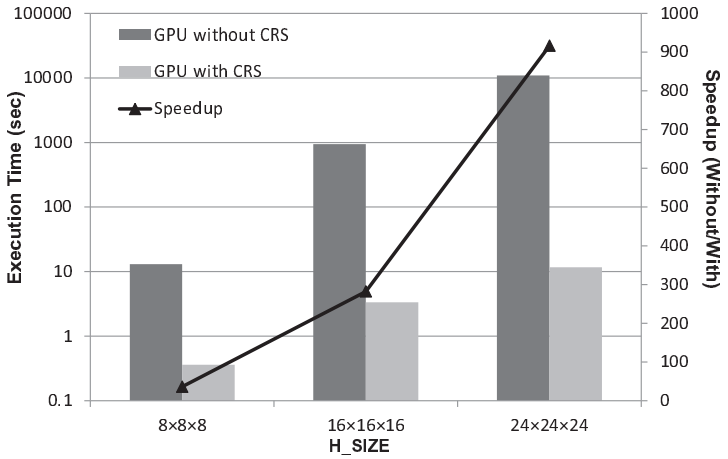
N	CPU (sec)	GPU (sec)	Speedup
128	4.31	1.91	2.25
256	8.61	3.03	2.83
512	17.18	5.31	3.23
1024	34.35	9.86	3.48
2048	68.69	18.89	3.63

Figure 8: Performance comparison increasing N.



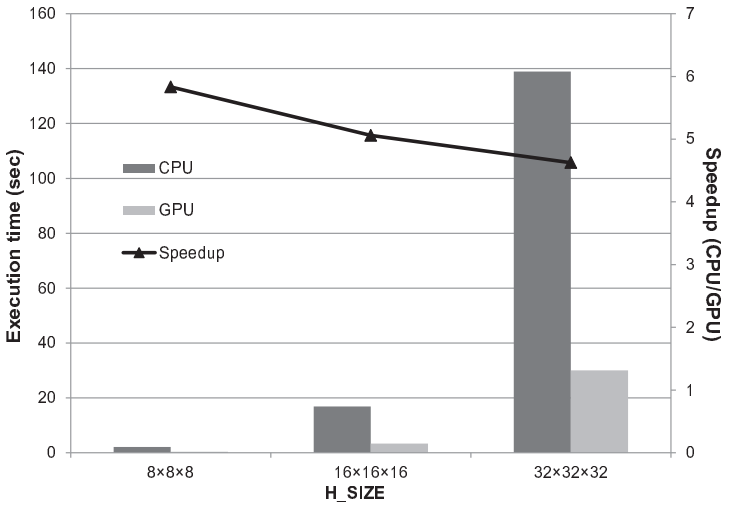
H_SIZE	CPU (sec)	GPU (sec)	Speedup
512	65.42	19.76	3.31
1024	278.18	79.04	3.51
2048	1233.11	345.76	3.56
4096	5073.31	1331.69	3.81

Figure 9: Performance comparison increasing H_SIZE.



H_SIZE	GPU without CRS (sec)	GPU with CRS (sec)	Speedup
8×8×8	13.01	0.36	36.11
16×16×16	941.89	3.34	282.01
24×24×24	10901.72	11.71	916.20

Figure 10: Execution time with/without CRS format on the GPU



H_SIZE	CPU	GPU	Speedup
8×8×8	2.1	0.36	5.8333333
16×16×16	16.9	3.34	5.0598802
32×32×32	138.9	30.03	4.6253746

Figure 11: Execution time on CPU and GPU applying CRS format

have performed comparison between the performance on the GPU and the one on CPU as shown in Figure 11, which obtains up to 4.8 times better performance. Comparing the speedup without CRS format, we can conclude that the CRS format is helpful for achieving higher speedup on the GPU and CPU [15].

As we discussed in four kinds of evaluations above, the performances on the GPU achieve better performances than the ones on CPU due to the highly parallelism caused by the GPU-based implementation explained in this paper. The implementation achieves the advanced performance even if it is applied to the actual examples from the condensed matter physics or the cases with hard conditions virtually when the amounts of the computation and the memory usage are increased. Moreover, we also confirmed that the CRS format obtains significant impacts not only on the performance but also on the memory access frequency. Thus, we can conclude that the KPM is a suitable algorithm that fits well to the GPU environment and the performance acceleration accomplishes amazingly the high performance due to the CRS format.

5 Conclusions

This paper proposed an implementation of the KPM widely used in the physics and the chemistry field to simulate various quantum states. Our GPU version shows about 5 times faster than the CPU one applying the CRS format for representing the Hamiltonian matrix. Therefore, using a GPU, productivity of the moments for a quantum state is accelerated to four times. Therefore, the GPU version is expected to be used for various grand challenge simulations to find a new quantum state that resolves unknown physical theories in the natural phenomenon.

For the future plans, we are considering to quest a method to find the best block size used in the GPU that defines the size of the stream processors' block. Moreover, the parallelization of the KPM on a message passing and a shared memory paradigm is also challenging because the recursive reference to get $\overrightarrow{r}_n$ becomes a bottleneck to be parallelized in fine-grain. Moreover, we are also planning to extend the GPU-based implementation to a GPU cluster for its parallelization.

Acknowledgment

This work is partially supported by the Japan Science Technology Agency (JST) PRESTO program.

References

[1] P. W. Anderson and H. Hasegawa. Considerations on double exchange. *Phys. Rev.*, 100:675–681, Oct 1955.

[2] J. G. Bednorz and K. A. Müller. Possible high T_c superconductivity in the Ba-La-Cu-O system. *Zeitschrift für Physik B Condensed Matter*, 64(2):189–193, 1986.

[3] E. Dagotto. Correlated electrons in high-temperature superconductors. *Review of Modern Physics*, 66(3):763–840, July-September 1994.

[4] W.M.C. Foulkes, L. Mitas, R.J. Needs, and G. Rajagopal. Quantum monte carlo simulations of solids. *Review of Modern Physics*, 73(1):33–83, January 2001.

[5] NVIDIA Corporation. CUDA: Compute Unified Device Architecture programming guide, http://developer.nvidia.com/cuda.

[6] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[7] U. Schollwöck. The density-matrix renormalization group. *Review of Modern Physics*, 77(1):259–315, January 2005.

[8] A. Weiße, G. Wellein, A. Alvermann, and H. Fehske. The kernel polynomial method. *Review of Modern Physics*, 78(1):275–306, January 2006.

[9] S.R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69(19):2863–2866, November 1992.

[10] S.R. White. Density-matrix algorithms for quantum renormalization groups. *Physical Review B*, 48(14):10345–1035, October 1993.

[11] S. Yamada, T. Imamura, T. Kano, Y. Ohashi, H. Matsumoto, and M. Machida. Ultra large-scale exact-diagonalization for confined fermion-hubbard model on the earth simulator: Exploration of superfluidity in confined strongly correlated systems. *Journal of the Earth Simulator*, 7():23–35, June 2007.

[12] S. Yamada, M. Okumura, and M. Machida. Direct extension of density-matrix renormalization group to two-dimensional quantum lattice systems: Studies of parallel algorithm, accuracy, and performance. *Journal of the Physical Society of Japan*, 78(9):094004, September 2009.

[13] M. Yamashita, N. Nakata, Y. Senshu, M. Nagata, H. M. Yamamoto, R. Kato, T. Shibauchi, and Y. Matsuda. Highly mobile gapless excitations in a two-dimensional candidate quantum spin liquid. *Science*, 328(5983):1246–1248, June 2010.

[14] S. Zhang, S. Yamagiwa, M. Okumura, and S. Yunoki. Performance Acceleration of Kernel Polynomial Method Applying Graphics Processing Units. *IPDPS/APDCM2011*, pages 569–576, 2011.

[15] S. Zhang, S. Yamagiwa, M. Okumura, and S. Yunoki. Performance Impact Applying Compression Format to Sparse Matrix on Kernel Polynomial Method Using GPU. In *Proceeding of the 2nd International Conference on Networking and Computing (ICNC11)/The First International Workshop on Challenges on Massively Parallel Processors (CMPP)*, 2011.