Performance Acceleration of Lattice Model Simulations for Interacting Electrons Applying Kernel Polynomial Method on GPUs

Graduate School of Systems and Information Engineering
University of Tsukuba

July 2013

Shixun Zhang

Abstract

Material research plays a role of great importance for the technology advancement. Based on quantum mechanics, in condensed matter physics various materials are studied through examining the behaviors of their constituent microscopic elements such as lattices, atoms and electrons. Among many, there exists a class of materials in which the interactions among electrons are considered as decisive factors for some unusual properties, e.g. superconductivity. Numerical study is a crucial approach to simulate such materials through solving Schrodinger equation.

Kernel Polynomial Method (KPM) is one the of most competitive numerical methods due to its low time complexity and high flexibility. However, KPM relies on the polynomial expansion technique which involves intensive recursive matrix vector operations. It is these recursive operations that prevent us from achieving very high performance through massive parallelism on a large cluster or supercomputer via MPI. Besides, as KPM is a highly memory bounded algorithm, the relatively low memory bandwidth on a CPU-based machine severely limits the performance.

Motivated by the fast growth of GPU's performance in recent years, this study focus on the GPU implementation of KPM to solve several fundamental physics quantities, namely density of states, local density of states and Green's function. The performance evaluations indicates in most cases GPUs could significantly accelerate KPM's performance for all three applications, suggesting that GPU is a promising tool to help break through the parallelization difficulty brought by the fine-grain parallelism and the memory bound characteristics.

Finally, to ease the difficulties of implementing GPU programs in physics research, a GPU-based library is proposed. The material research will benefit from the friendly user interface provided in the KPM library.

***Keywords*:** *Condensed Matter Physics, Numerical Material Research, Kernel Polynomial Method, Massively Parallel Computing, GPU, Density of States, Local Density of States, Green's function, Quantum Monte Carlo*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Numerical Study in Material Science and Kernel Polynomial Method

The modern technological advancement that makes 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 [1–3]. For example, Giant Magneto Resistance (GMR) [4, 5] found in 1988 let us be able to develop large-capacity hard drive through highly improving the data density on a silicon substrate. In the last decades GMR has resulted in a revolution of data storage technology for computer system [6]. Now the hard disks inside almost every personal computer and server are based on GMR. Therefore, synthesis of new special materials has a significant meaning for development of the modern technology and a more convenient society.

In addition to synthesis of new materials in laboratory, researchers in physics also have been working for an ultimate alchemy to design those materials by modeling the constituent atomic elements using the theory of quantum mechanics [7]. Although the quantum mechanics that governs the electronic motion in materials was established over 80 years ago, there still exist properties and phenomena whose origins are not well understood. Such examples include copper based high temperature superconductors [8] and peculiar magnetic insulators of certain organic compounds [9]. The common feature of these materials is the strong quantum correlation between electrons, which is turned out to be crucial for determining their behaviour. It is exactly this strong quantum correlation that makes it difficult to treat these systems analytically without introducing any bias.

One of the best ways to treat such system is to solve quantum mechanical equations numerically. As an exact numerical method with very high accuracy, full diagnolization [10] has been widely applied to study the spectral properties of various systems with time complexity $O(N^3)$, where N represents the system size. However, the degrees of freedom that increase exponentially with the number of electrons, $O(10^{23})$, may lead to huge computation amount, which makes the full diagnolization method fail to handle the relatively larger system even on a large cluster or supercomputer.

Given this situation, we have to resort to some sort of approximations. Unlike the exact method, the approximation method could simulate much larger systems in an expected accuracy within shorter computational time. Among many, well established approximate numerical methods thus far are *Quantum Monte Carlo (QMC)* method [11, 12], *Density-Matrix Renormalization Group (DMRG)* method [13–16], and *Kernel Polynomial Method (KPM)* [17]. Each method is suited to particular sets of problems, and also each has drawbacks [17].

Quantum Monte Carlo method has been widely applied to simulate the quantum process. Through importance sampling, QMC can estimate the properties such as phase transitions for quite large systems[18]. However, QMC method has limitations as well. In order to reduce the time complexity from $O(N^3)$ to $O(N)$, for example, a QMC method, which is applied to electron systems coupled to classical degrees of freedom, requires truncated expansion moments that may reduce the accuracy of the calculation [18]. Regarding DMRG, it is able to evaluate the ground state properties as well as the low energy excitations in high accuracy, but it is limited so far to (quasi) one-dimensional systems [15].

Comparing with QMC Method and DMRG, KPM is a more basic and general method [17], since it is essentially created just for examining the spectral properties of a given Hamiltonian, it provide users with many flexibilities such as choosing appropriate Hamiltonian space and basis to optimize the calculation. Due to this high flexibility, KPM is able to be combined with other numerical methods such as QMC and DMRG to improve the computing performance and accuracy. KPM also has some advantageous features such as low time complexity ($O(N)$) and controllable accuracy achieved by truncating the expansion moments. In some occasions, for example, when the partition function is calculated, a truncated-free spectrum in high accuracy could be obtained through extrapolation [19].

However, being involved in KPM, CPE is known as an ill-conditioned problem for algorithm implementation on CPUs using parallel programming techniques such as OpenMP and MPI, as CPE relies on the series of recursive moments which can not be calculated in a parallel manner [20], the performance suffers in a multi-core system such as cluster or supercomputer. Fortunately, in the calculation for each moment we could exploit the fine-grain parallelism introduced by matrix and vector operations. However, on a traditional CPU-based multi-core system, the scale of parallelism is strictly limited to the number of CPU cores within a node. Moreover, considering the fact that KPM is an memory-bounded algorithm, the relatively low bandwidth between CPU and DRAM dramatically reduce the performance can be reached even on fast CPUs.

1.2 Parallel Computing in Computational Science

Due to the performance limitation of a single processor as a result of limited frequency and number of transistors on a microchip, the parallel computing was naturally introduced to computational science in which the workloads usually are able to be divided into small pieces and distributed to many processors [21, 22]. In the past decades numerous supercomputers have been built to address the sharply increasing needs for high performance computing [23]. In a modern supercomputer, hundreds of thousands processor cores [24] have been integrated to provide petascale performance.

In the past decades, most of the supercomputers have been build based on CPU processors varying from IBM's Power series processor to Intel and AMD's x86 based CPUs [24]. Since the multi-core CPU had not been emerged in the market until 90s, at the early stage the supercomputers have to rely on single core CPUs and therefore the number of parallelism can be reached was severely limited [24]. On such systems the message passing interface (MPI) [25], whose prototype began to emerge on the early 90s, plays an important role for making parallel programs [26].

In MPI if a threads needs to communicate with an another thread, it has to explicitly call the runtime function to send the data, called *message*, to another thread [27]. Since MPI lacks support of shared memory among threads to encourage the memory locality [25], the memory space for different threads are independent, the "message sending mechanism" inevitably involves keeping multiple copies of the same data in different threads even they are running on a shared memory architecture. This not only result in more memory consumption but also decrease the memory accessing efficiency. At the late of 90s, another technique to parallelize the scientific computation called OpenMP was officially released [28], unlike MPI, OpenMP focus on the parallelization on shared memory architecture (SMA), in which multiple processors share the same memory area [29]. Therefore, OpenMP usually is considered as a complementary programming approaches [25, 28, 29].

The multi-core CPUs that have began to be popular on supercomputers after 2000 use a similar architecture with desktop computers [24]. However, these CPUs are actually designed for general purpose such as multimedia processing, documents editing or gaming [30]. The capability of general purpose CPU to run various tasks requires a balance of its performance in terms of different aspects such as floating point (FP) and integer operations [31–33]. However, for scientific computing, the requirement is much more dedicated, which usually is high FP operation performance and large parallelism [34, 35].

Being considered an solution to address the needs for high performance computing (HPC), the scientific computing orientated co-processors, also known as accelerators, emerged in the past few years are now attracting more and more attentions not only from the users but also from the processor manufacturers [36, 37]. In addition to the Tesla series GPU released by NVIDIA from 2008 [38], Intel also pushed to the market its co-processor called Xeon Phi that integrates 60 x86 CPU cores on 2012 [39]. In the cluster market many extra large supercomputers have already resorted to the co-processors to the break through peak performance record [24]. As for users, programming on co-processors becomes a necessary skill for the researchers working on computational science [36, 40].

In this study, focusing on NVIDIA's Tesla GPU and CUDA (Compute Unified Device Architecture) [41] platform and employing hybrid programming techniques, we apply KPM to implement high performance program to evaluate some important physics quantities including density of states (DOS), local density of states (LDOS)(for Anderson Localization study [42, 43]) and Green's function combined with Monte Carlo method to simulate double exchange model [44].

It should be noted that on a cluster although the co-processors may execute much more workload than CPUs, the parallelism over CPUs among compute nodes should not be ignored since it could provide different granularity of parallelism [45, 46]. i.e. the overall workload could be firstly divided and distributed to each node, for each node, co-processors is then invoked for fine-grain parallelism. Therefore, programming on accelerator-enabled system should follow a hybrid manner combining with traditional parallel techniques such as OpenMP and MPI.

1.3 Research Objectives

In this study, based on CUDA platform and employing hybrid programming techniques, we aim to solve the difficulty brought by the fine-grain recursion in KPM through applying NVIDIA's Tesla C2050 GPU, which has a peak bandwidth of 144GB/s and peak performance of 500 GFLOPS for double precision operations [47].

Another objective is to find out an appropriate method to combine the MPI and GPU by exploiting different granularity of parallelism, as the optimization over a heterogeneous cluster is also a crucial issue [48, 49]. Moreover, we also aim to develop optimization techniques to reduce memory usage, decrease the memory bus traffic and achieve high memory bandwidth through compressing large sparse matrix and coalescing memory access.

In order to help researchers in condensed matter physics perform simulations using KPM, currently we have established applications to evaluate three fundamental physics quantities. Our final target is to build a GPU-based programming library that can provide high performance, high stability and friendly user interface. The library should be able to run on large cluster or supercomputer. Moreover, considering the future extension, the library should be designed to provide enough flexibilities.

1.4 Originality and Contributions

On one hand, due to the interactions among numerous number of electrons and nuclei in materials as well as quantum nature of electron motion, the numerical simulation always require huge amount of computation [50]. On the other hand, the larger scale simulations are able to reveal to researchers more detailed electronic structures and various quantum phases [51]. In many occasions, one simulation costs days or weeks even for small systems. Therefore, it comes as no surprise that higher performance for material simulations is always desired eagerly for the physics researchers.

To address the needs for high performance, the co-processors such as GPU have the potential to be one of the main choices for scientific computing in the coming years. With massive parallelism and high memory bandwidth, it is very advantageous to implement KPM that could help to solve the fine-grain parallelism problem.

However, programming and optimizing a GPU program usually require a good knowledge of the GPU hardware architecture and programming model [40]. Indeed, some well established mathematics libraries such as BLAS [52] and Lapack [53] has been ported to GPU platform and make utilizing GPU less complicated. However, as these mathematics libraries are designed for general purpose, dedicated optimizing techniques for KPM such as unrolling the loops for Chebyshev expansion are unable to be applied if we use these general libraries rather than establishing a dedicated KPM implementation.

Currently, there is no public KPM implementation on GPU even using the general libraries. Building a high performance library has significant meanings for physics researcher, it is mainly because the GPU-based KPM library could shorten the simulation time significantly. With speedup of tens of times, a computational task that takes several weeks can be shorten to several hours. And also, with speedup impact, physics researchers could study much larger systems that may reveal more detailed results with much better accuracy.

This research has a meaning of significance for the computer science as well, as the traditional parallel programming techniques such as OpenMP and MPI may have difficulty for fine-grain parallelism. This work is a productive attempt to overcome the parallel granularity problem through hybrid programming combining MPI and CUDA. In addition, the discussion of the architectural aspects in a heterogeneous cluster such as memory bandwidth between CPU and GPU, communication overhead among threads and nodes could be a helpful guideline for implementing other similar algorithms.

1.5 Organization of This Thesis

This thesis is organized as follows: Chapter 2 gives an introduction to the background of material research and mathematical principle of kernel polynomial method and its applications to evaluate two of the most fundamental physics quantities, namely density of states (DOS) and local density of states (LDOS) [54–56]. After a comprehensive explanation of KPM algorithm, the numerical complexity is examined and algorithm profiling is performed. In addition, matrix compression techniques are also introduced in order to reduce the memory consumption of large Hamiltonian matrix. Given the significant importance of the multiplication between sparse matrix and vector (SpMV), this chapter also demonstrates the latest research on SpMV and shows the difficulties in implementing SpMV on conventional CPU-based cluster.

Chapter 3 shows the GPU hardware architecture and CUDA programming environment. Several optimization techniques are explained. At the end of this chapter, the specifications of *Yamgiwalab* GPU clusters that used for KPM implementations and performance evaluations in this thesis is introduced.

Chapter 4 demonstrates the an application of KPM that is applied to solve the density of states (DOS) of a given Hamiltonian matrix in a straightforward manner. Several implementation methods are demonstrated and evaluated.

In Chapter 5, a relatively more complicated KPM application is implemented on GPU to solve local density of states. The performances of single GPU as well as GPU cluster are examined.

Chapter 6 demonstrates Monte Carlo simulation of Double Exchange model [44] using KPM to solve the Green's function [19].

The topics regarding implementing KPM library is discussed in Chapter 7, the conclusions and future works are addressed in Chapter 8.

Chapter 2

The Kernel Polynomial Method

2.1 Definition

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

$$f(x) = \frac{1}{\pi\sqrt{1-x^2}}\left[\mu_0 + 2\sum_{n=1}^{\infty}\mu_n T_n(x)\right] , \tag{2.1}$$

where

$$\mu_n = \int_{-1}^{1} dx\, f(x) T_n(x) , \tag{2.2}$$

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

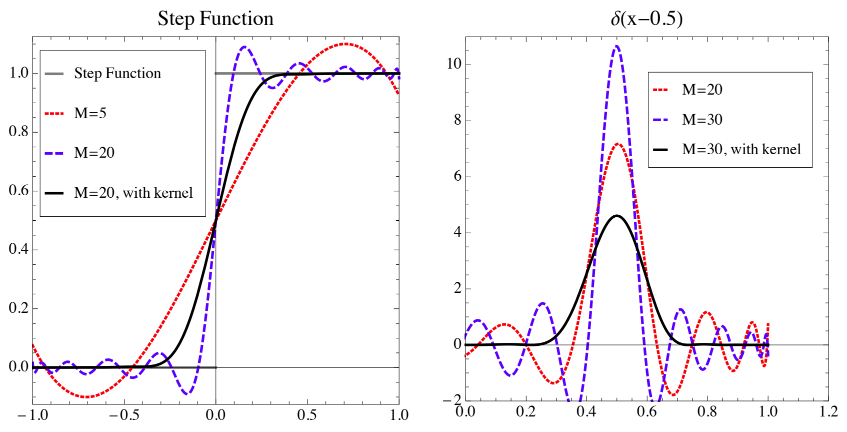
$$T_n(x) = \cos\left[n \arccos(x)\right] . \tag{2.3}$$

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

$$T_0(x) = 1 , \quad T_1(x) = x , \tag{2.4}$$

$$T_{n+2}(x) = 2xT_{n+1}(x) - T_n(x) . \tag{2.5}$$

$$y(x) = \begin{cases} 0, x < 0 \\ 1, x >= 0 \end{cases} \qquad \delta(x) = \begin{cases} 0, x \neq 0 \\ \infty, x = 0 \end{cases}$$



(a) (b)

Figure 2.1: Chebyshev expansion for (a) step function and (b) Dirac delta function, M represents the expansion order, larger M leads to higher accuracy. A kernel factor could be applied to generate a oscillation-free function denoted by the black thick curve

KPM is defined as

$$f_{\mathrm{KPM}}(x) = \frac{1}{\pi\sqrt{1-x^2}} \left[g_0\mu_0 + 2\sum_{n=1}^{N-1} g_n\mu_n T_n(x) \right] , \tag{2.6}$$

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

$$\| f - f_{\mathrm{KPM}} \| \xrightarrow{N\to\infty} 0 , \tag{2.7}$$

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

Discrete cosine transformation

Given the expansion coefficients μ_i, in a naive approach one should pick within [-1, 1] a series of $\{x_1, x_2, ..., x_S\}$, for each of which function $f(x_i)$ should be constructed using Equation 2.6. If the number of collection x_i is S, the time complexity for construction $f(x), x \in \{x_1, x_2, ..., x_S\}$ is $O(NS)$, it may result in large performance cost if function $f_{\mathrm{KPM}}(x)$ is repeatedly and intensively constructed.

Taking advantage of the Chebyshev basis function (Equation 2.3), if use the sampling method

$$x = \cos(\frac{\pi k}{N-1}) \; k = 0, 1, ..., N-1 \; , \tag{2.8}$$

Equation 2.6 becomes

$$\frac{1}{\pi\sqrt{1-x^2}} \left[g_0\mu_0 + 2\sum_{n=1}^{N-1} g_n\mu_n \cos(\frac{\pi kn}{N-1}) \right] \; , \tag{2.9}$$

the formula in [...] can be evaluated using discrete cosine transformation (DCT) of time complexity $O(Nlog(N))$

Normalization of Hamiltonian matrix

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 $[-1, 1]$,

$$\tilde{H} = (H - \alpha_+)/\alpha_- \, , \tag{2.10}$$

where

$$\alpha_\pm = (E_{\text{upper}} \pm E_{\text{lower}})/2 \, , \tag{2.11}$$

The parameters E_{upper} and E_{lower} are the upper and lower limits of the eigenvalues of H obtained by the Gerschgorin theorem [57].

2.2 Applying KPM to Solve Density of States

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.

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

$$\rho(\omega) = \frac{1}{D}\sum_{k=0}^{D-1} \delta(\omega - E_k)\,, \tag{2.12}$$

where E_k is the k-th eigenvalue and $\delta(x)$ is the delta function. We apply the linear transformation (2.10) and obtain the equation

$$\rho(\tilde{\omega}) = \frac{1}{D}\sum_{k=0}^{D-1} \delta(\tilde{\omega} - \tilde{E}_k)\,, \tag{2.13}$$

where

$$\tilde{\omega} = (\omega - \alpha_+)/\alpha_-\,. \tag{2.14}$$

In order to obtain the approximated DOS using KPM, the coefficients μ_n (2.2) in this case is obtained as

$$\begin{aligned}\mu_n &= \int_{-1}^{1} d\tilde{\omega}\, \rho(\tilde{\omega}) T_n(\tilde{\omega}) \\ &= \frac{1}{D}\sum_{k=0}^{D-1} T_n(\tilde{E}_k) \\ &= \frac{1}{D}\sum_{k=0}^{D-1} \langle k|T_n(\tilde{H})|k\rangle = \frac{1}{D}\mathrm{Tr}[T_n(\tilde{H})]\,,\end{aligned} \tag{2.15}$$

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

Evaluation of Traces

In order to evaluate the trace in Equation 2.15, 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

$$\langle\langle \xi_{r,i} \rangle\rangle = 0\,, \quad \langle\langle \xi_{r,i}\xi_{r',i'} \rangle\rangle = \delta_{rr'}\delta_{ii'}\,, \tag{2.16}$$

a random vector is defined through

$$|r\rangle = \sum_{i=0}^{D-1} \xi_{r,i} |i\rangle \, . \tag{2.17}$$

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

$$\begin{aligned}\mu_n &= \frac{1}{D} \mathrm{Tr} \left[T_n(\tilde{H}) \right] \\ &= \frac{1}{D} \sum_{i=0}^{D-1} \left[T_n(\tilde{H}) \right]_{ii} \\ &\simeq \frac{1}{D} \frac{1}{R} \sum_{i,j=0}^{D-1} \sum_{r=0}^{R-1} \langle\!\langle \xi_{r,i} \xi_{r,j} \rangle\!\rangle \left[T_n(\tilde{H}) \right]_{ij} \\ &= \left\langle\!\!\left\langle \frac{1}{D} \frac{1}{R} \sum_{r=0}^{R-1} \langle r | T_n(\tilde{H}) | r \rangle \right\rangle\!\!\right\rangle \, . \end{aligned} \tag{2.18}$$

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 (2.4) and (2.5),

$$|r_0\rangle = |r\rangle \, , \quad |r_1\rangle = \tilde{H} |r_0\rangle \, , \tag{2.19}$$

$$|r_{n+2}\rangle = 2\tilde{H} |r_{n+1}\rangle - |r_n\rangle \, . \tag{2.20}$$

Then μ_n is expressed by this expression as

$$\mu_n \simeq \left\langle\!\!\left\langle \frac{1}{D} \frac{1}{R} \sum_{r=0}^{R-1} \langle r_0 | r_n \rangle \right\rangle\!\!\right\rangle \, . \tag{2.21}$$

Performance Enhancement

In addition, the number M of coefficients is obtained with $M/2$ iterations if the following equation is used,

$$T_{2m-i}(H) = 2\, T_{m-i}(H)\, T_m(H) - T_i(H), \ i = 0, 1$$

applying $\langle r|$ and $|r\rangle$ on the left and right side, respectively,

$$\langle r|T_{2m-i}(H)|r\rangle = 2\langle r|T_{m-i}(H)\ T_m(H)|r\rangle - \langle r|T_i(H)|r\rangle,\ i = 0, 1$$

With the correspondence $|r_n\rangle := T_n(\tilde{H})|r\rangle$, it becomes

$$\begin{aligned} \langle r|r_{2m-i}\rangle &= 2\langle r_{m-i}|r_m\rangle - \langle r|r_i\rangle,\ i = 0, 1 \\ \mu_{2m-i} &= 2\langle r_{m-i}|r_m\rangle - \mu_i. \end{aligned} \tag{2.22}$$

2.3 Applying KPM to Solve Local Density of States

The local density of state (LDOS) $\rho_i(\omega)$ of the D-dimensional Hamiltonian matrix H is defined by

$$\rho_i(\omega) = \frac{1}{D}\sum_{k=0}^{D-1} |\langle i|k\rangle|^2 \delta(\omega - E_k)\,, \tag{2.23}$$

where E_k is the k-th eigenvalue and $\delta(x)$ is the delta function. We apply the linear scale transformation (2.10) and the LDOS is given by

$$\rho_i(\tilde{\omega}) = \frac{1}{D}\sum_{k=0}^{D-1} |\langle i|k\rangle|^2 \delta(\tilde{\omega} - \tilde{E}_k)\,, \tag{2.24}$$

where

$$\tilde{\omega} = (\omega - \alpha_+)/\alpha_-\,. \tag{2.25}$$

In order to calculate the LDOS using KPM, we need the coefficients μ_n^i defined in Equation (2.2), which is obtained as

$$\begin{aligned}
\mu_n^i &= \int_{-1}^{1} d\tilde{\omega}\, \rho_i(\tilde{\omega}) T_n(\tilde{\omega}) \\
&= \frac{1}{D}\sum_{k=0}^{D-1} |\langle i|k\rangle|^2 T_n(\tilde{E}_k) \\
&= \frac{1}{D}\sum_{k=0}^{D-1} \langle i|k\rangle T_n(\tilde{E}_k) \langle k|i\rangle \\
&= \frac{1}{D}\sum_{k=0}^{D-1} \langle i|T_n(\tilde{H})|k\rangle \langle k|i\rangle \\
&= \langle i|T_n(\tilde{H})|i\rangle
\end{aligned} \tag{2.26}$$

where $|k\rangle$ is the k-th eigenvector of H and $\langle k| = |k\rangle^{\dagger}$.

As a similar way that we used in DOS application, here we define $|i_n\rangle := T_n(\tilde{H})|i\rangle$, with the relations (2.4) and (2.5), the following recursion is obtained,

$$|i_0\rangle = |i\rangle\,, \quad |i_1\rangle = \tilde{H}|i_0\rangle\,, \tag{2.27}$$

$$|i_{n+2}\rangle = 2\tilde{H}|i_{n+1}\rangle - |i_n\rangle\,. \tag{2.28}$$

Therefore, the coefficients of the expansion for LDOS can be expressed by

$$\tilde{\mu}_n = \langle i|i_n\rangle \tag{2.29}$$

2.4 KPM Algorithm Analysis

KPM is quite a common numerical method that can be used to evaluate many functions for many physics applications. However, it involves a recursive operations to produce the expansion coefficients as shown in Equation 2.20. It is this recursion that is considered as ill-condition problem for parallelization on cluster, because the different expansion coefficients could not be calculated simultaneously by a large number of threads via MPI. Therefore, it is necessary to review the algorithm and

Algorithm 1 Calculation of expansion moments μ_i using KPM

```
Input: Vector $\vec{r}$
Input: Expansion order $M$
Input: Normalized Hamiltonian matrix $H$ of dimension $D \times D$
Output: The expansion coefficients $\mu[i], i \in (0, M-1)$
Require: Vector $\vec{r}$ , $\vec{r}_1$, $\vec{r}_2$
 1: for $i = 0 \rightarrow M/2$ do
 2:     if $i = 0$ then
 3:         $\vec{r}_1 \leftarrow \vec{r}$
 4:         $\mu[0] \leftarrow \vec{r}_1^* \cdot \vec{r}_1$
 5:     else if $i = 1$ then
 6:         $\vec{r}_2 \leftarrow H \times \vec{r}_1$
 7:         $\mu[1] \leftarrow 2\vec{r}_1{}^* \cdot \vec{r}_2 - \mu[1]$
 8:         $\mu[2] \leftarrow 2\vec{r}_2{}^* \cdot \vec{r}_2 - \mu[0]$
 9:     else
10:         $\vec{r}_1 \leftarrow 2H \times \vec{r}_2 - \vec{r}_1$
11:         Swap pointers of $\vec{r}_1$ and $\vec{r}_2$
12:         $\mu[2i-1] \leftarrow 2\vec{r}_1^* \cdot \vec{r}_2 - \mu[1]$
13:         $\mu[2i] \leftarrow 2\vec{r}_2^* \cdot \vec{r}_2 - \mu[0]$
14:     end if
15: end for
```

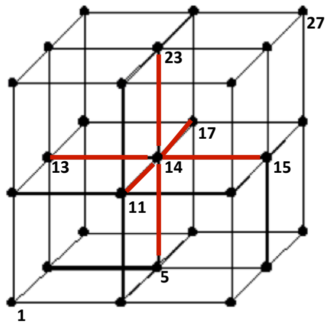
exploit potential parallelism.

At the beginning, let us examine the characteristics of the general KPM algorithm to calculate the expansion coefficients μ_i. Applying Equation 2.22, Algorithm 1 demonstrates the subroutine for calculating the Chebyshev expansion coefficients, it requires three input parameters: an initial vector $\vec{r}$, expansion order M (Integer) and the Hamiltonian matrix H of size $D \times D$. The output is the expansion coefficients μ_i. To ensure the convergence, Hamiltonian matrix should be normalized before being passed to this subroutine.

Here should note that vector $\vec{r}$ consists of either real numbers or complex numbers depending on different physics models, in the algorithm the notion $\vec{r}^*$ is used to represent the conjugate vector of $\vec{r}$.

2.4.1 Matrix Compression Techniques

Using KPM, one's main task is to evaluate the expansion coefficients μ_n, which involves a recursive relation between the vectors. Especially, implied by Equation 2.20



(a) Cubic lattice

$$\begin{array}{c|ccccccccccccccc}
 & 1 & \dots & 5 & \dots & 11 & \dots & 13 & 14 & 15 & \dots & 17 & \dots & 23 & \dots & 27 \\
\hline
1 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
\vdots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
14 & 0 & 0 & t_{14,5} & 0 & t_{14,11} & 0 & t_{14,13} & \epsilon_{14} & t_{14,15} & 0 & t_{14,17} & 0 & t_{14,23} & 0 & 0 \\
\vdots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
27 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots
\end{array}$$

(b) Hamiltonian matrix

Figure 2.2: An example of construction of Hamiltonian matrix

and Equation 2.28, inside the recursion the matrix and vector multiplication takes the most percentage of computational workload, therefore, according to Amdahl's law [58], a high performance implementation of the recursion, especially the matrix vector multiplication, is considered the key to boost the performance of KPM.

Another very important feature in KPM algorithm is that the size of Hamiltonian matrix could reach to a very large order in realistic simulation due to the fact that physical system usually contains very large amount of atoms or molecules which result in huge degrees of freedom [59, 60]. However, Hamiltonian matrix is a very sparse matrix (e.g. the sparsity is in order of $O(10^{-6})$) and therefore can be greatly compressed to reduce memory usage and improve computational performance. Choosing which compression format usually depend on the matrix sparse pattern and the types of operations (e.g. matrix-vector multiplication or matrix-matrix multiplication) that the matrix will be involved [61–63].

Construction of Hamiltonian Matrix

Before discussing the matrix compression techniques, let us start with how to construct the Hamiltonian matrix, and therefore the matrix sparse pattern could be determined. Here we take Anderson disorder model as an example, which describe the electron's motions, called "hoping", from one site to one of its nearest neighbors (nn) in the lattice. Figure 2.2 (a) depicts a 3^3 cubic lattice that consists of 27 sites, each of which is labeled by a number from 1 to 27. The electron motion in this lattice was described by the Hamiltonian matrix shown by Figure 2.2 (b), in which

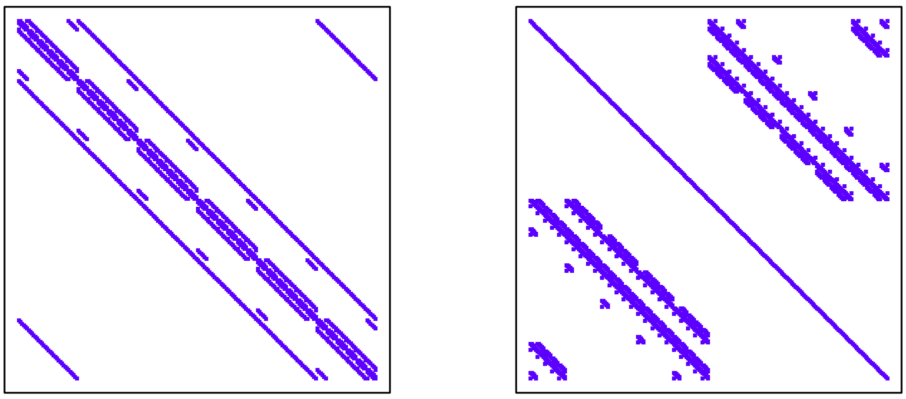


Figure 2.3: Distribution of non-zero values in the sparse Hamiltonian matrix for cubic and bcc, respectively. The two matrices are constructed based on Anderson disorder model with periodical boundary condition.

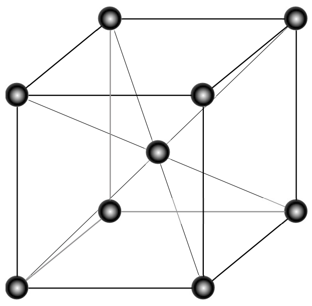


Figure 2.4: A unit cell of body centered cubic lattice, each site is surrounded by 8 nearest neighbors

the matrix element $t_{i,j}$ represents the "hoping" integral from site i to j. For each matrix element, $t_{i,j} \neq 0$ only happens if site j and i are nearest neighbors with each other, e.g. i is 14 and j is 5. Otherwise, $t_{i,j}$ is set to 0.

With *Periodical Boundary Condition*(PBC), each site is surrounded by 6 nearest neighbors, e.g. nearest neighbors of site 14 include site 5, 11, 13, 15, 17 and 23. Together with the on-site potential ($t_{14,14} = \epsilon_{14}$), each matrix row contains only 7 none-zero values no matter how large the lattice is. It should be noted here that for the sites on the boundary of a lattice, the number of its nearest neighbors are also seen as 6, since, for example, site 23 can be seen as nn of site 5.

Figure 2.3 visualizes the distributions of non-zero values, also known as *sparse pattern*, of Hamiltonian matrices for cubic and bcc lattice, respectively. As for bcc lattice shown by Figure 2.4, each site has 8 nearest neighbors and therefore each matrix row only contains $8 + 1 = 9$ none-zero values.

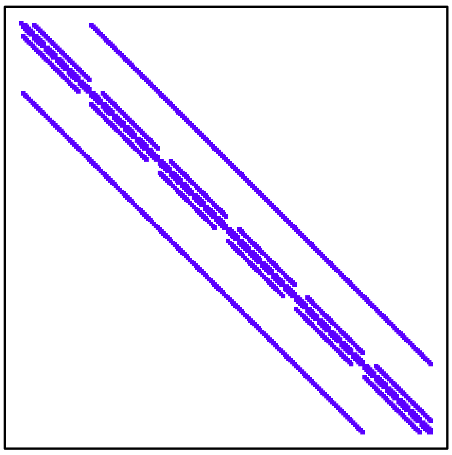


Figure 2.5: Distribution of non-zero cubic lattice applying open boundary condition

It should be noted that the number of non-zeros in each row are not always same, it may varies in some occasions, e.g. when *Open Boundary Condition*(OBC) is applied. With open boundary condition, in Figure 2.2 (a), site 23 is no longer the nn of site 5. Therefore, the site 14 has 6 nearest neighbors while site 5 only has 5 nn. Figure 2.5 shows the sparse pattern for a cubic lattice with open boundary condition. Comparing with Figure 2.3 (a) that is based on periodical boundary condition, the number of non-zeros in the matrix varies between $4 \sim 7$.

Matrix compression techniques: CSR and ELL

CSR (Compressed Sparse Row) [64] and ELLPack [65] are both effective formats to compress a sparse matrix for SpMV. However, both CSR and ELL have pros and cons, choosing which format in a given application, for maximum compression ratio and computational performance, depends on the sparse matrix pattern and also the operation type, e.g. multiplication or addition. In this study, we apply both CSR and ELL format.

CSR format

Figure 2.6 (a) shows a 6×6 matrix in dense format that contains many zero elements. Figure 2.6 (c) shows the matrix compressed in CSR format, CSR requires three one-dimensional arrays: A, CA, RA, where A stores the values of the non-zero

$$G=\begin{array}{|c|c|c|c|c|c|}\hline 1&2&0&0&0&0\\\hline 3&4&5&0&0&0\\\hline 0&6&7&8&0&0\\\hline 0&0&9&10&11&0\\\hline 0&0&0&12&13&14\\\hline 0&0&0&0&15&16\\\hline\end{array}$$

(a) Dense matrix

$$A=\begin{array}{|c|c|c|}\hline 1&2&*\\\hline 3&4&5\\\hline 6&7&8\\\hline 9&10&11\\\hline 12&13&14\\\hline 15&16&*\\\hline\end{array}\qquad JA=\begin{array}{|c|c|c|}\hline 0&1&*\\\hline 0&1&2\\\hline 1&2&3\\\hline 2&3&4\\\hline 3&4&5\\\hline 4&5&*\\\hline\end{array}$$

Non-zero values *Column indices*

(b) ELL format

$$\begin{array}{rl}
\textit{Non-zero values}\quad A= & \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\\hline\end{array}\\
\textit{Column indices}\quad CA= & \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 0&1&0&1&2&1&2&3&2&3&4&3&4&5&4&5\\\hline\end{array}\\
\textit{Row indices}\quad RA= & \begin{array}{|c|c|c|c|c|c|c|}\hline 0&2&5&8&11&14&16\\\hline\end{array}
\end{array}$$

(c) CSR format

Figure 2.6: Matrix compression techniques. figure (a) shows a 6×6 dense matrix, figure (b) and (c) demonstrate how to compress the dense matrix by ELL and CSR, respectively.

elements in the matrix, CA stores the column indices of the non-zero elements in A, and RA stores the indices of which element in array A starts a new row. For the sake of convenience for SpMV, the number of all values in A, i.e. 16, is appended to the end of array RA.

Assuming all values of the dense matrix are stored in double precision, the memory consumption ratio of CSR is given by

$$\begin{aligned}\frac{\text{CSR}}{\text{Dense}} &= \frac{\overbrace{P\times 8}^{\text{A}} + \overbrace{P\times 4}^{\text{CA}} + \overbrace{(D+1)\times 4}^{\text{RA}}}{\underbrace{D\times D\times 8}_{\text{Dense matrix}}}\\ &= \frac{32P+4(D+1)}{8D^2},\end{aligned}\tag{2.30}$$

in which P denotes the number of non-zeros in the matrix and D represents the matrix size. In the expression the two vectors CA and RA in CSR are assumed to be stored in four-bytes integers.

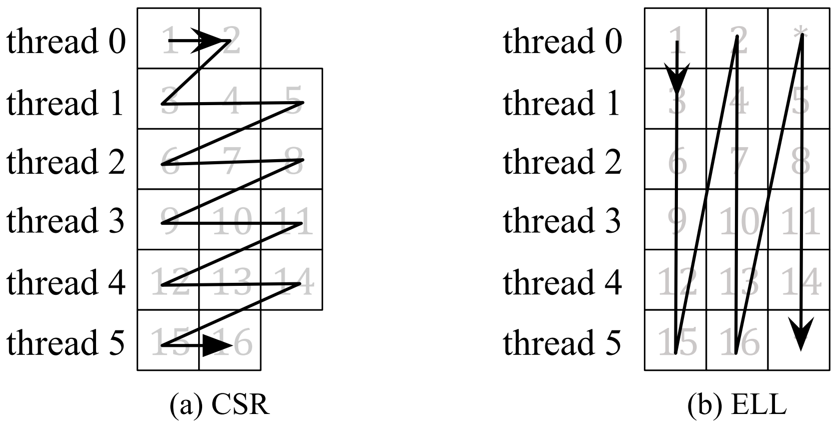


Figure 2.7: row/column-major storage format for ELL and CSR

ELL format

ELL format is an another effective compression format for sparse matrix in SpMV. Figure 2.6 b) demonstrates how to compress the dense matrix in Figure 2.6 a) using ELL format. As shown in the figures, ELL requires two 2D arrays to store the non-zeros and their column indices respectively. In many cases, since the number of non-zeros in every row varies, additional data (e.g. both 0 for A and CA in SpMV) should be padded to ensure that A and CA are 2D matrices. Therefore, the compression efficiency of ELL largely depends on the sparse pattern of the matrix, which here refer to the difference among the numbers of non-zero in every row.

Using double precision as well, the compression ratio of ELL format is as follows

$$\frac{\text{ELL}}{\text{Dense}} = \frac{\overbrace{D \times W \times 8}^{\text{A}} + \overbrace{D \times W \times 4}^{\text{JA}}}{\underbrace{D \times D \times 8}_{\text{Dense matrix}}} = \frac{3W}{2D}, \tag{2.31}$$

in which W represents the width of matrix A and JA after padding the extra data.

Discussion for ELL and CSR

CSR and ELL are both compression formats widely applied in various applications, according to the detailed explanation in the last section, we could compare them

```
void spMV_CSR(float *A, float *CA, float * RA, size_t dim, float *V, float *R)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;
    for(int i=0; i<dim, i += num_threads){
       int row_start = RA[i];
       int row_end = RA[i + 1];
       float sum = 0;
       for(int j=row_start; j < row_end; j++){
           sum += A[j] * V[CA[j]];
       }
       R[i] = sum;
    }
}
```

Figure 2.8: SpMV in CSR format

```
void spMV_ELL(float *A, float *CA, int num_row, int num_col, float *V, float *R)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    int num_threads = blockDim.x * gridDim.x;
    for(int i = 0; i < num_row, i += num_threads){
       float sum = 0;
       for(int j=0; j < num_col; j++){
           int index_1d = i * num_col + j;  //for row-major
           //int index_1d = i + j * num_row;  //for column-major
           sum += A[ index1d ] * V[CA[index_1d]];
       }
       R[i] = sum;
    }
}
```

Figure 2.9: SpMV in ELL format

with each other in terms of the memory consumption and performance.

When the number of non-zeros in each row varies sharply in the matrix, CSR is a more appropriate choice since there is no requirement of extra padding data. In this case ELL is less efficient since it requires large amount of padding data not only for matrix A but also for JA. In addition, CSR's performance is also superior to ELL format, since ELL may include unnecessary memory access and calculations for the padding data.

However, when the number of non-zeros in each row is a constant, just as the periodical condition case, ELL becomes a better choice due to: 1) Higher compression ratio due to no requirement for padding data and no requirement for storing the row indices, i.e. vector RA in CSR format. 2) Higher performance due to less memory access.

It should be noted that ELL provides us with an opportunity to choose row or column-major storage format to keep the matrix in global memory while the CSR format does not. This feature may become crucially important when we try to implement the program using massive parallel threads running on a shared memory architecture such as GPU.

Let us take the sparse matrix-vector (SpMV) multiplication as an example. Figure 2.8 and Figure 2.9 show two naive implementations of GPU-based SpMV ($A \times V = R$) for CSR and ELL, respectively. In a parallel environment with many concurrently running threads, the multiplication usually is assigned to threads by rows, e.g. each thread calculates the multiplication between a row and the vector as shown in Figure 2.7. In CSR case, Figure (a), no consecutive data is accessed, however, using ELL and column-major storage thread $0 \sim 5$ read the consecutive data, which may greatly improve the cache performance. In addition, GPU may combine the multiple memory access request to consecutive data into one instruction [41] to help programmers reach very high bandwidth, sometimes the realistic throughput could be very close the theoretical peak bandwidth.

2.4.2 Numerical Complexity

This algorithm mainly consists of three types of operations, SpMV, vector-vector multiplication and abstraction. It is easy to understand the most intensive calculation exists in line 10.

For a given dense matrix G of $D \times D$, the time complexity of matrix-vector multiplication is represented by $O(D^2)$, however, in our case, the Hamiltonian matrix is compressed in sparse format and maximum number of non-zero values in each row does not scale with the matrix size D. Again, taking the cubic lattice as an example, in Hamiltonian matrix each row contains 7 non-zeros values, which result in 14 floating point operations for the multiplication between one row and the vector, the total operations for $H \times \vec{r}$ is 14D, therefore the time complexity of SpMV is $O(D)$ rather than $O(D^2)$. Regarding the vector abstraction in line 10 and vector multiplication in line 12 and 13, they both contribute time complexity of O(D). Therefore, for M/2 loops the total time complexity is presented by $O(MD)$.

Indeed, larger D requires a lager expansion order M, but usually M is much smaller than D, comparing with the full diagnolization of complexity O(D^3), KPM has a great advantage in terms of the computational performance.

2.4.3 Algorithm Profiling

It is easy to understand that the recursion relation in line 10 (Algorithm 1) is most time consuming part, in which the SpMV occupies very high percentage of floating point operations. In order to reveal and analysis more features of the recursion, here we use a simple profiling model [66], in which the criteria to determine if an algorithm is memory or computational bounded regarding a given hardware can be expressed by

$$c = \frac{\text{peak performance (GFLOPS)}}{\text{peak memory bandwidth(GB/s)}} = \frac{\text{number of floating point operations}}{\text{1 byte}}, \tag{2.32}$$

The value of c represents that for every byte read from memory, how many floating point operations is supposed to be executed to cover the memory latency. For example, the theoretical peak performance of Tesla GPU C2050 is 500GFLOPS while its peak bandwidth is 144GB/s, so c is about $500/144 = 3.47$.

For a given algorithm, we calculate p to determine if the algorithm is memory/computation bounded, if

$$p = \frac{\text{number of floating point operations}}{\text{number of bytes loaded from memory}} \tag{2.33}$$

is smaller than c, the algorithm generally considered as memory bound, otherwise, it can be recognized as computational bounded.

Here assuming using the real number of double precision, for the recursion in line 10, there are $14D$ floating point operations for $H \times \vec{r}_2$, D operations for multiplying the result of $H \times \vec{r}_2$ by 2, and another D operations caused by the substraction of vector $\vec{r}_1$. Therefore the total floating point operations is $16D$.

As for the memory read in line 10, assuming we use ELL format and double precision, the recursion needs to access

$$\underbrace{D \times 7 \times 8}_{\text{matrix values (read)}} + \underbrace{D \times 7 \times 4}_{\text{values' column indices (read)}} + \underbrace{D \times 8}_{\text{vector } \vec{r}_2 \text{(read)}} \tag{2.34}$$

$$+ \underbrace{D \times 8}_{\text{vector } \vec{r}_1 \text{(read)}} + \underbrace{D \times 8}_{\text{vector } \vec{r}_1 \text{(write)}} = 108D \tag{2.35}$$

bytes. Divided $16D$ by $108D$, the p value for the recursion is obtained as 0.148, which is much smaller than 3.47, indicating this algorithm is highly memory bounded.

Hardware Limitations

Therefore the memory bandwidth would be the bottleneck for KPM. The recent CPU achieves the following theoretical peak I/O bandwidth applying dual channel DDR2-800 memory modules:

$$400MHz \times (2channels) \times (64bits/channel) \times (2bits/clock)$$

$$= 12.5GB/s$$

Actually in our system the Intel's Core i7 processor achieves about 9 GB/s given by the *stream benchmark* [67], which is only less than 10% of the peak bandwidth of Tesla C2050 (144GB/s).

An another disadvantage of CPU is that the memory bus is shared among multiple cores, this may lead to poor performance scaling in terms of number of CPU cores. This analysis is consistent with our experimental result that will be discussed in the following chapters.

2.4.4 Limitations of Using Third-party Library

In order to ease the difficulties of making high performance program, some well optimized math libraries such as Lapack [53] and BLAS [52] were established and serving as important tools for computational science researchers. As the GPU computing becomes more and more popular in recent years, the GPU-based math libraries such as CULA [68] and CUBLAS [69] have been established in correspondence to CPU version of Lapack and BLAS library. There are also some libraries designed mainly focusing on sparse matrix operations such as CUSP [70].

However, since these third-party libraries are designed for very general purpose such as a matrix-vector multiplication, using these libraries to implement KPM algorithm sometimes may bring difficulties for us to make custom optimizations. For example, if we use the third-party library to implement the recursion (line 10 in Algorithm 1), we have to follow two steps: First, call a function to calculate the matrix-vector multiplication and store the produced vector, say V, into global memory. Second, call another function to perform vector-vector subtraction. In the second step we have to read the vector V from memory again, which is not necessary if we combine the two steps into one kernel.

Therefore, implementing a dedicated KPM library with high performance kernels is very necessary and important.

2.5 Sparse Matrix Vector Multiplication (SpMV)

Since SpMV plays a role of great importance in solving the linear algebra equations [71, 72] that widely exists in physics numerical simulations, it has been intensively studied both on CPUs and GPUs [73–80].

Among many researches on SpMV, Table 2.1 lists two papers [75, 77] focusing the sparse matrix vector multiplication on many core architecture. The reason to choose these two papers is that they include the SpMV performance evaluation of sparse matrix *atmosmodd* (in UFL Sparse Matrix Collection[1]) and *Laplace 7pt*, these two matrices have a similar sparse structure/pattern to our Hamiltonian matrix introduced in Section 2.4.1. The matrix size for atmosmodd and Laplace 7pt [77] are $10^6 \times 10^6$ and $1,270,432 \times 1,270,432$, respectively. The number of non-zeros in each row for both of the two matrices is near to 7, which is close to our Hamiltonian matrix. The last column represents the percentage of the reached SpMV performance

[1]http://www.cise.ufl.edu/research/sparse/matrices/

Table 2.1: SpMV performance in related research

Paper	Matrix Format	Storage Major	FP [a]	Hardware	Theo. Peak Perf.	Reached Perf.	Percentage
E. Saule, et. al., 2012[b]	CSR	row	double	C2050 (448 cores)	500GFlops	6.5GFlops	1.3%
	CSR	row	double	Dual X5680 (12cores)	160GFlops	3GFlops	1.9%
	CSR	row	double	Xeon Phi(61cores)	1TFlops	7GFlops	0.7%
Nathan Bell, et. al., 2008[c]	ELL	column	single	GTX280(240cores)	933GFlops	23Gflops	2.4%
	CSR	row	single	GTX280(240cores)	933GFlops	5GFlops	0.53%
	ELL	column	double	GTX280(240cores)	78GFlops	12.5GFlops	16.0%
	CSR	row	double	GTX280(240cores)	78GFlops	2.5GFlops	3.2%

[a]floating point precision, single or double
[b]using atmosmodd which is a structured sparse matrix.
[c]using Laplace 7pt structured matrix.

comparing to the peak performance shown in column 6.

A common observation in Table 2.1 is that, for all three processors, namely Tesla C2050 GPU, Xeon X5860 CPU and Xeon Phi, the obtained performances take very small percentage (mostly less than 4%) of the theoretical peak performance no matter for CSR or ELL format.

Besides, the performance difference between CSR and ELL format can be also observed in Table 2.1. For example, for double floating point operation, the performance of applying ELL (12.5Gflops) is about 5 times of CSR performance(2.5GFlops) on GTX280, mainly because the ELL is stored in column major to trigger coalesced memory access while CSR is restricted to row-major format.

In order to improve the performance of SpMV, various new storage formats have been proposed. *G. Jeswin et. al.* use a revised DIA format [81] that improves the performance of SpMV for structured sparse matrix by over 75% on GTX580 GPU. *F. Vázquez, et. al.* proposed ELLPACK-R [82] format to achieve an acceleration factor of over 80 comparing to the CSR on Intel's Dual Core E8400 CPU. *G. Jeswin, et. al.* [78] proposed a new C-DIA compression format and improves the SpMV performance by 75% in maximum.

The bandwidth research between global memory (DRAM) and L2 cache is given by Table 2.2. On the contrary to the low floating performance shown in Table 2.1, the realistic memory bandwidth reach much higher percentage of the theoretical bandwidth. Especially, applying column-major storage format to ELL matrix, the

Table 2.2: SpMV bandwidth in related research

Paper	Matrix Format	Storage Major	FP [a]	Hardware	Theo. Peak Bandwidth	Reached Bandwidth	Percentage
Nathan Bell, et. al., 2008[b]	ELL	column	single	GTX280(240cores)	141GB/s	110GB/s	78.0%
	CSR	row	single	GTX280(240cores)	141GB/s	20GB/s	14.1%
	ELL	column	double	GTX280(240cores)	141GB/s	110GB/s	78.0%
	CSR	row	double	GTX280(240cores)	141GB/s	17GB/s	12.1%
Nathan Bell, et. al., 2009[c]	ELL	column	single	GTX285(240cores)	159.0 GB/s	120GB/s	75.4%
	CSR	row	single	GTX285(240cores)	159.0 GB/s	25GB/s	15.7%

[a]floating point precision, single or double
[b]using Laplace 7pt structured sparse matrix.
[c]using Laplace 7pt structured sparse matrix.

bandwidth can be drastically boosted. For example, for single floating point operation, the reached memory bandwidth is as high as 110GB/s on GTX 280, suggesting an over 75 % of the peak bandwidth (141GB/s).

2.6 Discussion and Summary

At the beginning of this chapter we reviewed the mathematical formulation of kernel polynomial method and its applications to evaluate DOS and LDOS, which is followed by a comprehensive introduction to KPM algorithm and characteristics through detailed analysis.Through performance profiling, it can be learned that KPM is a highly memory bounded algorithm due to the heavy sparse matrix-vector multiplication in the recursion.

As the Hamiltonian matrix is a sparse matrix, two matrix compression formats, namely CSR and ELL, are applied in this study. The two formats are reviewed in terms of several aspects such as compression ratio and storage format.

Since SpMV plays such an important role in KPM, the end of the chapter focus on introducing related works on this topic that has been intensively studied. The conclusions is consistent with our analysis through algorithm profiling in Section 2.4.3, which is that the memory accessing speed is the bottleneck to achieve high performance. The related research also indicates that GPU performs much better than CPU regarding SpVM. Therefore, implementing KPM on GPU has promising perspective.

Chapter 3

GPU Architecture and CUDA Programming Model

In the past few years remarkable advancement has been made not only on GPU hardware performance but also on programmability. The number of processing units and size of memory is increasing steadily. For example, the latest Nvidia's Tesla K20x has embed around 2500 processing units, which is almost 19 times more than the first generation GPU released on 2008. Another important improvement of GPU is memory bandwidth, which is crucial for memory bounded applications. With latest DDR5 memory and 384-bit wide memory bus, K20x could provide up to 250 GB/s bandwidth between the processing units and memory.

As co-processor, GPU could not work independently without CPU. GPU requires a programming model that is bounded with the hardware architecture, understanding the programming model and the corresponding hardware architecture is very necessary for users.

3.1 GPU Architecture

A video adapter that includes a GPU and a Video RAM (VRAM) which is connected to a CPU's peripheral bus such as PCI Express. The video adapter works as a peripheral device of the CPU, the GPU is controlled by the CPU to perform a part of tasks in

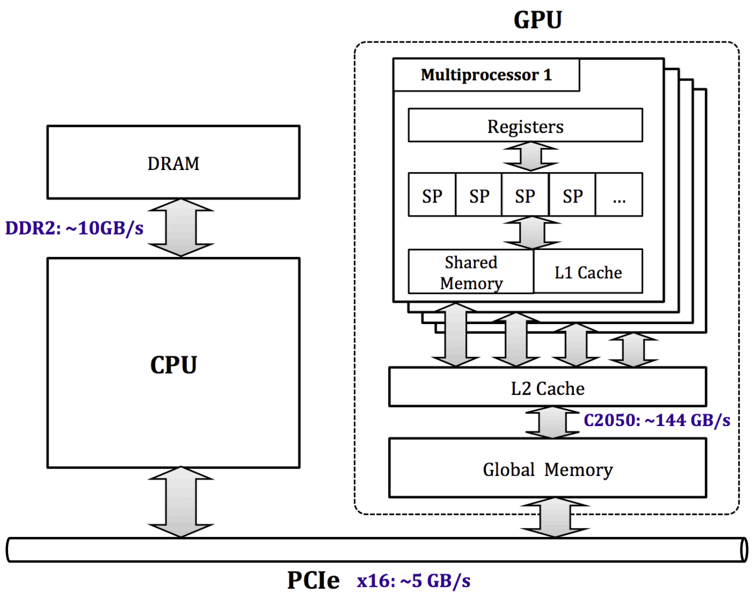


Figure 3.1: NVIDIA's GPU architecture

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 one kind of processor called the *stream processor* (SP). Hundreds of the stream processors are massively integrated in an LSI chip and work together concurrently fetching the SIMD style program, these SPs are grouped into several *multiple processors* (MP), in each of which SPs share the registers and L1 cache, besides, in each MP there exists some amount of fast accessing cache, called *shared memory*, exposed to programmers for performance optimization. All the MP share the same L2 cache, which connect to the GPU's global memory. As the GPU originally designed for processing graphics, even the general purpose computing follows the stream-based style.

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 used in this study provides its peak memory bandwidth up to 144GB/s, according to its profiler *computeprof* given by NVIDIA. For the memory bounded applications, the large I/O performance is attractive to fully exploit the potential ability of the multiple stream processors working concurrently.

```
int main(){
    float *h_a, *h_b, *h_c; //pointers to host memory

    . . .   //initialise host vector h_a, h_b, h_c

    float *d_a, *d_b, *d_c; //pointers to GPU memory

    //allocate memory on GPU
    cudaMalloc((void**)&d_a, sizeof(float) * num);
    cudaMalloc((void**)&d_b, sizeof(float) * num);
    cudaMalloc((void**)&d_c, sizeof(float) * num);

    //copy the input data from host memory to GPU memory
    cudaMemcpy(d_a, h_a, sizeof(float) * num, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(float) * num, cudaMemcpyHostToDevice);

    //invoke the kernel function
    //num_blocks: number of blocks
    //num_threads: number of threads in a block
    VectorAdd<<<num_blocks, num_threads>>>(d_a, d_b, d_c);

    //copy the calculation result from GPU to host memory
    cudaMemcpy(h_c, d_c, sizeof(float) * num, cudaMemcpyDeviceToHost);
}

//this is kernel function executed as a thread on GPU
__global__ void VectorAdd(float *a, float *b, float *c)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    c[thread_id] = a[thread_id] + b[thread_id];
}
```

Figure 3.2: CUDA programming model

3.2 CUDA Programming Model

Regarding the programming environment for GPU-based computing, the Compute Unified Device Architecture (CUDA) has been proposed by NVIDIA corporation [41]. CUDA provides an easy way to access to the computational resource and transparent interface for the numerical computation. In CUDA programming model, a thread is described as a stream-based function written in C, also called a *kernel function*. All of the threads are grouped into many *thread blocks*, each of which corresponds to a MP in Figure 3.1. Thread block can be tiled into one, two or three dimensions, the number of threads in a thread block also called *block size*.

The comprehensive programming model of CUDA is illustrated by Figure 3.2, in which it performs addition of two vectors. The overall program can be seen as two parts targeted to CPU and GPU, respectively. The "main" function is executed by CPU while the kernel function "VectorAdd", with prefix "__global__", is executed on GPU. As co-processor, GPU usually only execute the intensive operations, e.g. vector addition. CPU is in charge of preparing the data and controlling the program flow.

Because the CPU and GPU have independent memory space, memory copy operation is necessary. One typical procedure of the CUDA program execution is

$$\begin{tikzcd}
\boxed{\textit{copy the data from host memory to GPU memory}} \arrow[d, Rightarrow] \\
\boxed{\textit{call kernel function to perform calculation}} \arrow[d, Rightarrow] \\
\boxed{\textit{copy the result back to host memory}}
\end{tikzcd}$$

In Figure 3.2 the pointers "h_a", "h_b" and "h_c" point to the host memory space that is allocated using C function *malloc*. The pointers "d_a", "d_b" and "d_c" is the corresponding pointers that hold the addresses of GPU's global memory space, they are allocated using CUDA API function *cudaMalloc*.

The initial data are prepared in host side, therefore before the calculation, we have to use *cudaMemcpy* to copy the data from host to GPU memory, the option *cudaMemcpyHostToDevice* parameter is used to specify the direction of copy operation. After finishing copying the data, kernel function *VectorAdd* is involved to perform the calculation. To invoke the kernel function, we have to specify at least two parameters, here we use *num_block* and *num_theads* to represent the number of blocks and the block size (also known as number of threads in each block).

After the calculation finish, through specifying the option *cudaMemcpyDeviceToHost* we use function *cudaMemcpy* again to copy the result back to host memory.

All the copy operation between CPU and GPU must via PCIe bus as depicted in Figure 3.1. Comparing to the bandwidth between GPU's L2 cache to global memory, the current PCIe bus can not provide high memory throughput, therefore frequent or intensive memory copy between CPU and GPU may greatly decrease the performance, therefore, we should avoid frequent communication between CPU and GPU.

3.3 Optimization Techniques on GPU

Comparing to implementing the GPU program, the work to optimize the GPU code is usually considered much more challenging. Users are required to have a good knowledge not only on the programming model but also a comprehensive understanding of hardware architecture and program profiling skills. Among many, the follow are some of the most important techniques that could dramatically boost GPU's performance.

Coalescing Memory Access

Accessing the data in GPU's global memory is critical to the performance especially for the memory bound algorithms such as KPM. To increase the memory capacity and bandwidth, the DRAM consists of multiple chips to provide larger memory bus width [83]. For example, one 128-bit data bus needs sixteen 8-bit chips. In this mechanism the data are accessed in parallel, if one location is accessed, many of its adjacent locations are also transferred to processor's cache no matter if they are required by the processor. In CUDA if the consecutive locations are accessed by multiple processors at the same time, the GPU automatically combines the multiple memory read request into one read transaction, for example, the 16 requests to read 16 8-bit integers will be combined into one 128-bit transaction. This data reading style called coalesced memory accessing.

The parallel memory accessing nature is not only the motivation of coalescing access but also plays a crucial role in cache effectiveness. When a memory location is accessed, its adjacent locations are also transferred to the cache, if they are accessed in the follow short period before being flushed out the cache, it can be passed to the processor without global memory accessing [41].

Involving Shared Memory

Shared memory is a very important feature to boost GPU's performance by greatly reducing the memory read latency [41]. Actually, shared memory is a visible and programmable memory with an equal accessing speed to L1 cache, in NVIDIA's Fermi architecture, the size of shared memory and L1 cache can be exchanged, suggesting they are implemented in a similar way.

The latency of shared memory is about 20$\sim$40 clock cycles while the global memory requires 400$\sim$600 cycles for one read transaction [84]. Therefore keeping the frequently used or intermediate data in shared memory would improved the performance greatly.

Parallelization Scheme

Usually the parallelization scheme does not play an independent role to the performance, for example, for different parallelization designs, we can or can not take advantage of the shared memory. Take the multiplication between a row-major dense matrix and a vector as an example, in the first scheme, we distribute the workload by rows to each treads, that means thread i calculate the multiplication between row i and the vector. In second scheme, all the threads read the data in a row simultaneously, multiply the data with the vector and make the reduction. In the second scheme, we can take the advantage of the shared memory but the reduction operation may decrease the performance. Therefore, how to parallelize a target algorithm should be carefully designed.

Minimizing Memory Bandwidth Bottleneck

A method that could minimize the memory bandwidth bottleneck is much more useful to the memory bound algorithms such as the SpMV in KPM. The target of the optimization is to increase the "*FP instructions/byte*" ratio to perform more FP operations for every bytes read from memory. There are many techniques can help us to achieve this goal by maximizing the shared memory and cache effect. For example, we unroll the loops in SpMV to archieve high "*FP instructions/byte*" ratio and also give the compiler more hints to optimize the cache.

3.4 Experiment Environment

All the programs in this study are implemented and evaluated on Yamagiwalab's cluster. The hardware architecture and software environment are as follows:

The cluster consists of a head node and 16 compute nodes of the NEC LX series. All nodes are connected with 40GByte/sec Infiniband network. The head node contains one Intel's Core Quad i7 (3.2GHz) processor of four cores and 12GB DDR2 memory. Besides, one Tesla C2050 compute card is attached via x16 PCI express. The head node also serves as NIS [85] and NFS [86] server in order to share the user accounts and home directory, respectively.

Each node contains 12 CPU cores (Dual Xeon E5645 2.4GHz) and two NVIDIA Tesla M2050 GPUs being connected to the x8 PCI Express buses. Besides, 12GB memory is attached to each node. Totally there are 192 CPU cores and 32 GPUs on the compute nodes.

As for the software environment, The OS for all the node is CentOS 5.5 and the CUDA version is 4.0. The compiler for CPU is gcc-4.4.4. All experiments performed below use the compiler option "-O3" for CPU and GPU programs.

3.5 Discussion and Summary

This chapter briefly reviews the NVIDIA GPU's hardware architecture and CUDA programming model that provides us with high performance and friendly programming interface, respectively.

Comparing to the programming, optimizing the program on GPU is more challenging, several optimization techniques are discussed in this chapter, including coalescing memory access, involving shared memory and choosing parallelization methods. Since KPM is a highly memory bounded algorithm, given GPU's much higher bandwidth than that of CPU, it is resealable to seek a GPU-based implementation of KPM that could reach to high performance.

Chapter 4

KPM to Evaluate Density of States

Since most properties of materials are determined by the behaviour of electrons, the electron density of state is considered as one of the most important physical quantities. For instance, DOS behaves differently for metals, semiconductors, and insulators [87]. In mathematics, it represents the distribution of eigenvalues.

Comparing to Lanczos method [10], which is considered as an excellent tool for evaluating extremal eigenvalues, KPM is a more appropriate choice to evaluate distribution of eigenvalues [17]. In this chapter, applying KPM, an effective DOS implementation is proposed on GPU. This chapter also includes discussions of various aspects that play a role in the computational performance such as the format of Hamiltonian matrix, paralleization methods and memory management schemes. The performance evaluation indicates that GPU could significantly accelerate the KPM's performance [20, 88, 89].

4.1 Algorithm design

Based on the mathematical derivation of KPM and the application for DOS function in Section 2.1 and Section 2.2, Algorithm 2 illustrates the algorithm. At the beginning it constructs and normalize the Hamiltonian matrix that is compressed in CSR format here. Evaluation the trace of Hamiltonian matrix requires a large number of random vectors that are in normal distribution, here the number of the

Algorithm 2 Calculate DOS applying KPM

```
Require: Integer $RS$ to represents the number of random vectors
Require: Hamiltonian matrix $H$ of dimension $D \times D$
Require: Integer $N$ to represents the number of moments
Require: Vector $\vec{r}$ , $\vec{r}_1$, $\vec{r}_2$, $\vec{r}_3$, $\mu$
 1: Construct matrix H according to the physical model
 2: Normalize matrix H
 3: for $j = 1 \to RS$ do
 4:    Create a random vector $\vec{r}$, its elements in normal distribution
 5:    for $i = 1 \to N$ do
 6:       if $i = 1$ then
 7:          $\vec{r}_1 \leftarrow \vec{r}$
 8:          $\mu[1] \leftarrow \mu[1] + \vec{r} \cdot \vec{r}_1$
 9:       else if $i = 2$ then
10:          $\vec{r}_2 \leftarrow H \times \vec{r}_1$
11:          $\mu[2] \leftarrow \mu[2] + \vec{r} \cdot \vec{r}_2$
12:       else
13:          $\vec{r}_3 \leftarrow 2 \times H \times \vec{r}_2 - \vec{r}_1$
14:          $\mu[i] \leftarrow \mu[i] + \vec{r} \cdot \vec{r}_3$
15:          Swap pointers of $\vec{r}_1$,$\vec{r}_2$ and $\vec{r}_3$
16:       end if
17:    end for
18: end for
19: for $i = 1 \to N$ do
20:    $\mu[i] = \mu[i]/(RSD)$
21: end for
22: Construct DOS function using $\mu_i$ using Equation 2.6
```

random vectors is represented by product of two integers RS, for each of random vector $\vec{r}$, it performs the the recursive iteration, which mainly involves SpMV and dot product of two vectors. Since the expansion coefficients μ_n is an average that is evaluated using a stotistical method, in the implementation μ_n is accumulated for each loop of j and then divived by RSD (line 20). Finally, using Equation 2.6 the DOS function is re-constructed with the estimated coefficients μ_n

In this algorithm, the SpMV is the most computational intensive part, an effective parallel implementation would be crucial for the overall performance.

Time Complexity Discussion

The construction and normalization (line 1 and line 2 in Algorithm 2) of Hamiltonian matrix H is of $O(D)$ complexity and contribute little to the overall performance overhead. As the length of the vector $\vec{r}$ (in line 4) is D, creation of a random vector contributes another $O(D)$ complexity. Considering the loop from line 3 to line 18, the overall complexity for line 4 becomes $O(DRS)$. The recursion part which mainly involves matrix-vector multiplication and vector-vector abstraction (VVA), if the matrix H is compressed in CSR format and each row has 7 none-zero values, one SpMV and one VVA contributes $O(D)$ complexity. Since the recursion is iterated for N times, where N represents the polynomial expansion order, the code from line 5 to line 17 contributes $O(DRSN)$ complexity. Together with the code line 19-21 which contributes $O(N)$, the total time complexity is obtained as

$$O(D) + O(D) + O(RSD) + O(DRSN) + O(N)$$

In realistic case, as

$$RS \gg 1, D \gg N,$$

The time complexity can be briefly represented as

$$O(DRSN), \tag{4.1}$$

suggesting that the execution time will be scaling linearly with matrix size D.

Since the accuracy of the obtained DOS function largely determined by the expansion order N, an increasing D demands an increasing N to keep the resolution of DOS function unchanged[90]. Parameter RS represents how many random vectors to use for evaluation the trace.

4.2 Implementation on GPU

Implementing an algorithm is not only a problem of archiving the best performance on a given processor, but also a task that involves making the trade-offs among various aspects such as scalability and usability. As the first attempt to implement KPM on GPU, in this study several different implementations are proposed to show KPM's features from different aspects and find out an appropriate implementation.

These different implementations include

- (i) single CUDA kernel version
    - (a) full map method (using dense matrix)
    - (b) full map method (using CSR matrix)
    - (c) silding window method (using CSR matrix)
- (ii) multiple CUDA kernel version
    - (a) multiple CUDA kernel method (using CSR matrix)

All the four versions (i.a, i.b, i.c, ii.a) above can be divided into two groups, (i) the single kernel version and (ii) the multiple kernel version.

In version (i), all the code that runs on GPU is embed into a single CUDA kernel function, which means the loop from line 3 to line 21 in Algorithm 2 is placed inside a kernel function, the advantage of doing this way is that we can avoid the overhead caused by data transfer between CPU and GPU as well as frequent invocation of kernel functions. With the single kernel concept, three different implementations are made: (a) the full map method using dense matrix format, which is the most naive implementation in which the matrix H is stored without compression and there is no memory optimization at all. (b) the full map method applying CSR format to Hamiltonian matrix, and (c) sliding window method using CSR matrix. Version (c) implements a relatively complicated memory management system to reduce the memory usage and bus traffic.

In KPM algorithm, synchronizations must be placed after/before each iteration for the recursion. However, the explicit synchronization among all the threads in different blocks is not supported by CUDA [41]. To overcome this difficulty, in single kernel versions, the loops that begins from line 3 is divided to each block, so every block performs the recursion independently as shown in Figure 4.1 (1). The total memory consumption is the sum of memory usage in each block, therefore, this method usually requires large amount of memory.

In order to reduce the memory consumption, version (ii), called multiple-kernel, version is proposed by dividing the whole algorithm into several small kernel functions. In this version the code from line 4 to line 17, that will be executed by GPU, is divided into several kernel functions. In this case, the loop clause from line 3 is placed outside the kernel function and executed by CPU. Unlike version (i), this implementation do not need explicit synchronizations because there is a implicit synchronization between two kernel functions in CUDA programming model. In this implementation, since all the threads across every block share the same vector $\vec{r}$, $\vec{r}_1$, $\vec{r}_2$ and $\vec{r}_3$, the memory consumption is significantly reduced. Therefore, this version can calculate much larger Hamiltonian size, e.g. $256^3 \times 256^3$.

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.

4.2.1 The Full Map Method

Implementation

Figure 4.1 (1) shows the generation part for the $\vec{r}_n$. $\vec{r}_n$ needs $\vec{r}_{n-1}$, $\vec{r}_{n-2}$. $\vec{r}$ is randomly generated at the beginning. These four vectors are kept in the global memory and the recursion is performed by swapping the pointers without memory copy. The matrix-vector multiplication is distributed to BS threads by row. After vector $\vec{r}_n$ is generated, it is then multiplied to $\vec{r}$ by BS threads to generate μ_n. Therefore, this part will generate $\tilde{\mu}_1$, $\tilde{\mu}_2$ ... $\tilde{\mu}_N$ using $\vec{r}$ and $\vec{r}_n$ where n is from 1 to N.

Figure 4.1 (2) depicts the parallelization for generation of μ_n. The dot product result of $\vec{r}_n$ and $\vec{r}$ is stored into $\tilde{\mu}$, another memory area. As a dot product of two

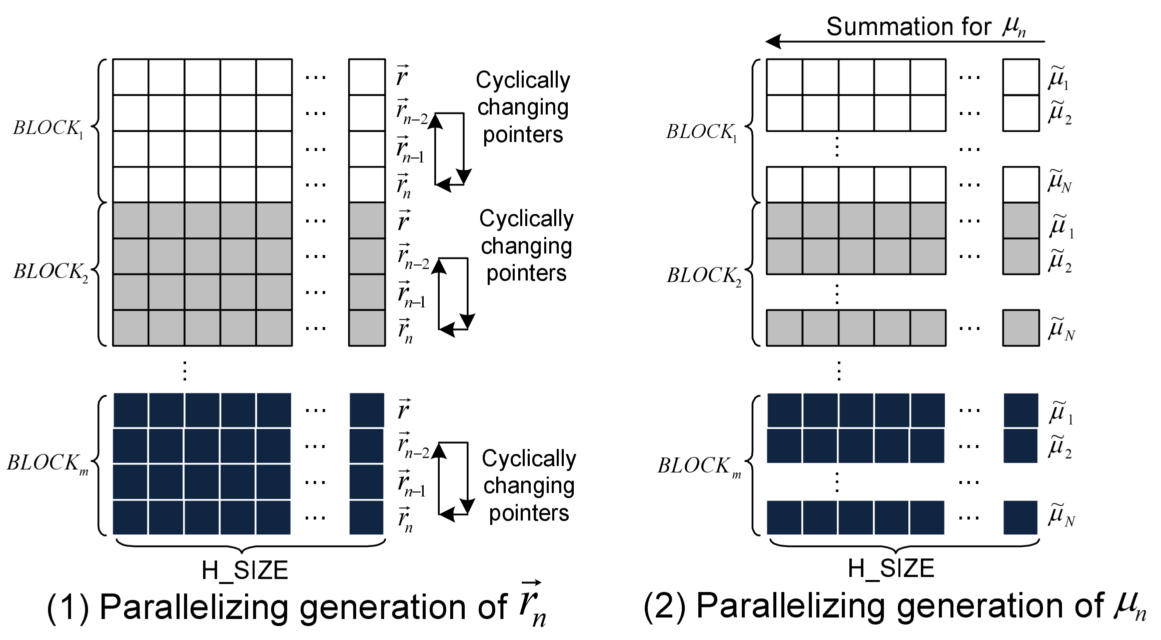


Figure 4.1: Implementation applying the full-map method.

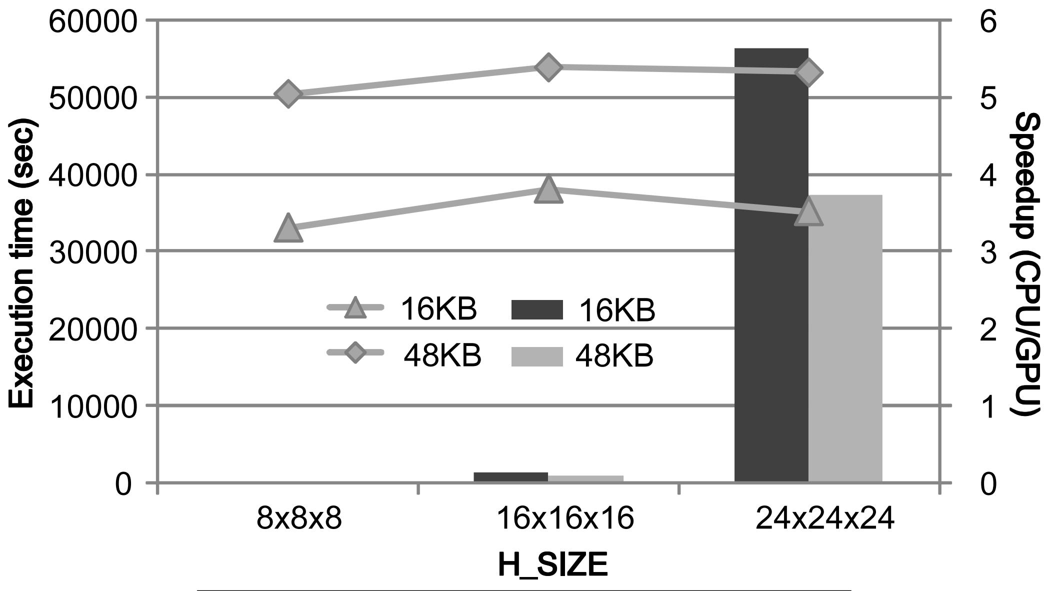
vectors, $\tilde{\mu}$ should be a scalar, but here, in order to increase the parallelism, $\tilde{\mu}$ is stored and accumulated as a vector. In the final, working in parallel, all threads in a block just make summation from left side to right side to produce a scalar $\tilde{\mu}_n$.

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 Figure 4.1, four $\vec{r}$ vectors per block are kept in the global memory. Each $\vec{r}$ vector has H_SIZE elements. Therefore, this part consumes $Number\ of\ Blocks \times 4 \times H\_SIZE \times 8$ bytes. As for 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 in dense format and being shared among all blocks, occupying $H\_SIZE^2 \times 8$ bytes. Therefore, the total number of memory needed for the full



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

Figure 4.2: Performances of the full map method comparing among 16KByte/48KByte cache and the speedup.

map method is calculated as follows:

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

This implementation is very straightforward and have an advantage of less control overhead because all variables are prepared in memory simply. However, the memory usage increases linearly by the number of thread blocks. Thus, this method will lead to low parallelism, i.e. reducing the number of thread blocks, to increase the capability for simulating larger lattice model.

Evaluation for The Full Map Method With Dense Format

The first evaluation analyses the performance of the full map method keeping the dense matrix H in global memory. Figure 4.2 shows the performances of applying dense format to full map method on Tesla C2050, the performance difference of applying different L1 cache sizes are also shown.

As we discussed in Section 2.4.2, due to the dense matrix, the performance follows the complexity of $O(SRMD^2)$. Therefore, changing H_SIZE causes exponential growth of 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 $8 \times 8 \times 8$, $16 \times 16 \times 16$ and $24 \times 24 \times 24$ to the experiments in this section.

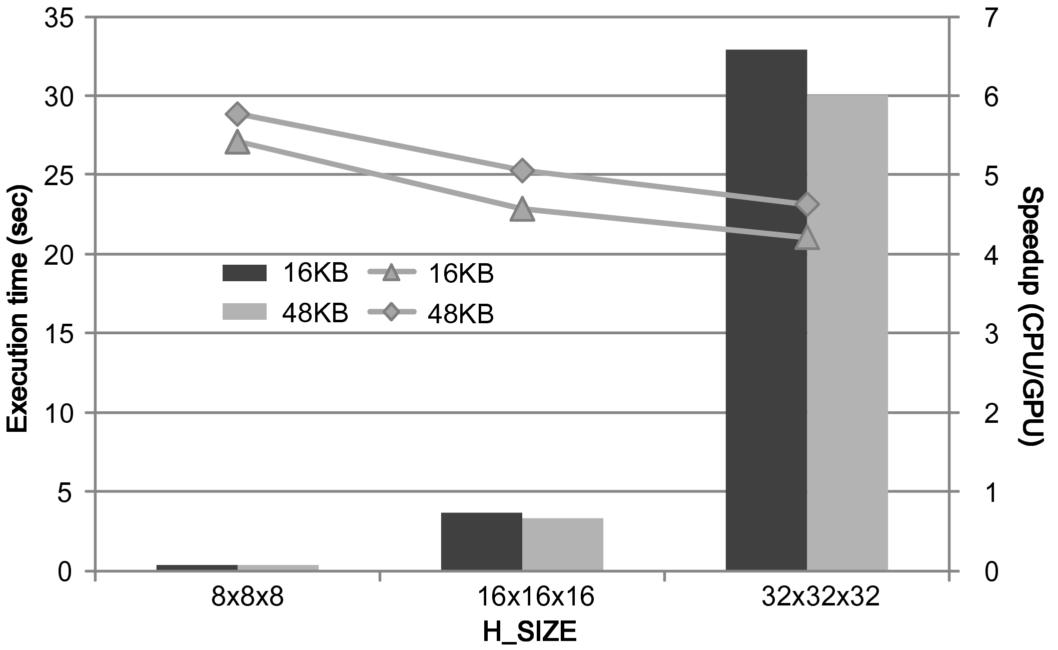
It can be learned in Figure 4.2 that the performances with 48KB cache size achieves about 30-35% better performance than the ones with 16KB cache size, suggesting that the larger cache size enhances the performance because the cached part of the vector $\vec{r}$ or matrix H can be effectively shared with all the threads in a block.

Because, with dense format, the full map method achieves only less than six times better performance than a recent single CPU core. This speedup factor does not have a big impact since, in modern systems, dual quad-core CPUs (8 cores totally) may produce higher performance than a GPU. 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 it consumes very large amount of memory.

Evaluation for The Full Map Method With the CSR Format

Let us apply the CSR format to the full map method. Figure 4.3 presents the performance evaluations. It indicates the CSR format can drastically improve the performance not only on GPU but also on CPU, because the calculation amount is much reduced by ignoring redundant calculations with zeros in the dense format matrix. It also can be learned that the time complexity becomes $O(SRMD)$ from the performance shown in Figure 4.3, as that has been discussed in Section 2.4.2.

The speedup seems to be saturated to about 4-5. CSR does not improve the speedup of GPU/CPU because applying CSR the CPU performance is also greatly improved. However, the execution time decreases drastically, indicating that the sparse matrix compression techniques, such as CSR, are indispensable for speeding



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

Figure 4.3: Performances of the full map method with the CSR format.

up the KPM. Applying CSR format, calculation for larger H_SIZE, e.g. $32 \times 32 \times 32$ becomes possible since the required amount of memory is greatly reduced.

Although CSR compression to matrix H could reduce the memory consumption greatly, the memory consumption for the calculation in Figure 4.1 (1) and (2) does not reduce. Considering there are only about 2.6GB memory on GPU, calculation of very large H_SIZE such as $128 \times 128 \times 128$ is still not possible. It is very necessary for the KPM algorithm on GPU to reduce the required amount of memory. Therefore, another implementation method, called sliding window method, is proposed targeting to reduce the total memory consumption for the operations in Figure 4.1 (2).

4.2.2 The Sliding Window Method

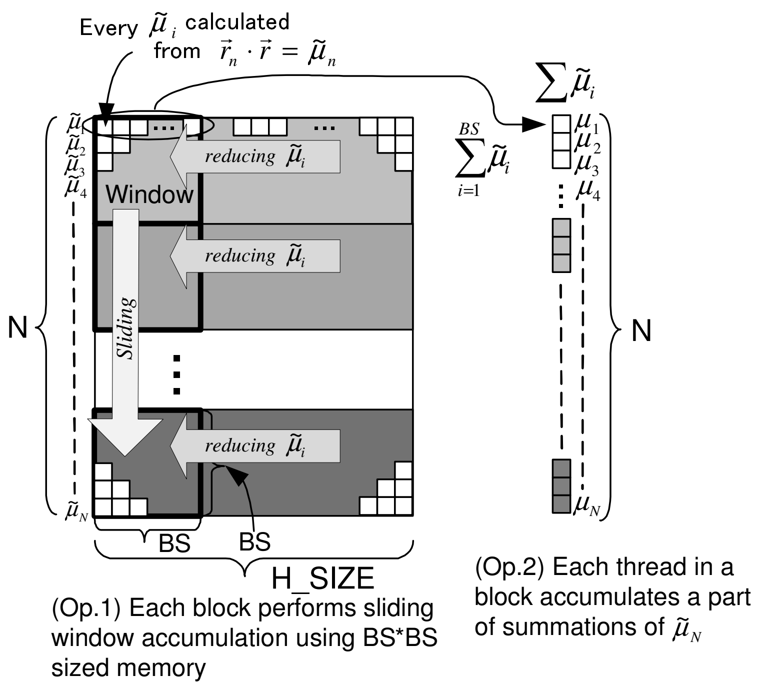
Implementation

At the beginning, Let us discuss how much large memory consumption that the full map method uses in the case when H_SIZE is $100 \times 100 \times 100$, $N = 128$ and $Number\ of\ Blocks = 8$. The operation (1) in Figure 4.1 costs 256 MBytes, and the operation (2) in Figure 4.1 costs 8 GBytes. However, the later increases explosively when we increase the expansion order N and number of blocks. This H_SIZE is not able to be simulated by the full map method because 8GBytes for the data structure of Figure 4.1(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.

Therefore, an another method for the operation (2) is proposed, called *sliding window* method. The former part of the sliding window method corresponds to the operation (1) in Figure 4.1. Figure 4.4 summarizes the operation in the sliding window method that corresponds to the operation (2) of the full map method. The operations are performed by two parts: one is accumulation of $\tilde{\mu}_i$ partially and another is final reduction of $\tilde{\mu}_i$.

For the first part, it prepares a memory area where the square is $BS \times BS \times 8$ bytes in each thread block, where BS represents the block size. Each block performs generations of $\tilde{\mu}_i$ where $1 \leq i \leq N$ from the dot product $\vec{r_n} \cdot \vec{r}$ according to the operation (1). Each multiplication performed in the dot product (i.e. $\vec{r_n}[i] \times \vec{r}[i]$ where $1 \leq i \leq H\_SIZE$), which is reduced to a $1 \times BS$ array, is stored and accumulated into the window memory. After the window is full filled, BS threads within a block concurrently accumulating the data horizontally to produce scalar μ_i. 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.

Figure 4.4 shows how the calculation is performed using the sliding window. The thread block needs to prepare only the window memory. Making summation of the $\tilde{\mu}$s from the dot products of $\vec{r_n} \cdot \vec{r}$, and reducing all $\tilde{\mu}$s with summations into



Every $\tilde{\mu}_i$ calculated from $\vec{r}_n \cdot \vec{r} = \tilde{\mu}_n$
$\sum \tilde{\mu}_i$
$\sum_{i=1}^{BS} \tilde{\mu}_i$
reducing $\tilde{\mu}_i$
Window
Sliding
N
BS
BS
H_SIZE
(Op.1) Each block performs sliding window accumulation using BS*BS sized memory
(Op.2) Each thread in a block accumulates a part of summations of $\tilde{\mu}_N$

Figure 4.4: Implementation applying the sliding window method. Each thread block manages this operations using the memories.

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 $w[\,][\,]$, to calculate the vector product $\vec{r_n} \times \vec{r}$, The thread i produces

$$\sum_m \vec{r_n}[m \times BS + i] \times \vec{r}[m \times BS + i] \tag{4.2}$$

where m represents the number of sliding windows in horizon, $0 \leq m \leq 2$, and saves it to $w[j\%4][i]$ where $1 \leq i \leq 4$ and $1 \leq j \leq 24$. Thus, using Equation 4.2, the dot product is reduced to a $1 \times BS$ array.

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 Figure 4.4 (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 Figure 4.4 (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 $\vec{r}_n$ in the KPM needs the same memory size as the one in the full map method, which is 256MBytes. In addition, the generation of μ_n becomes $Number\ of\ Blocks \times (BS \times BS \times 8 + N \times 8)$. When apply the actual parameters to this equation, it becomes about 1MB. Thus, the sliding window method reduces the memory usage drastically, and makes the large size simulation available.

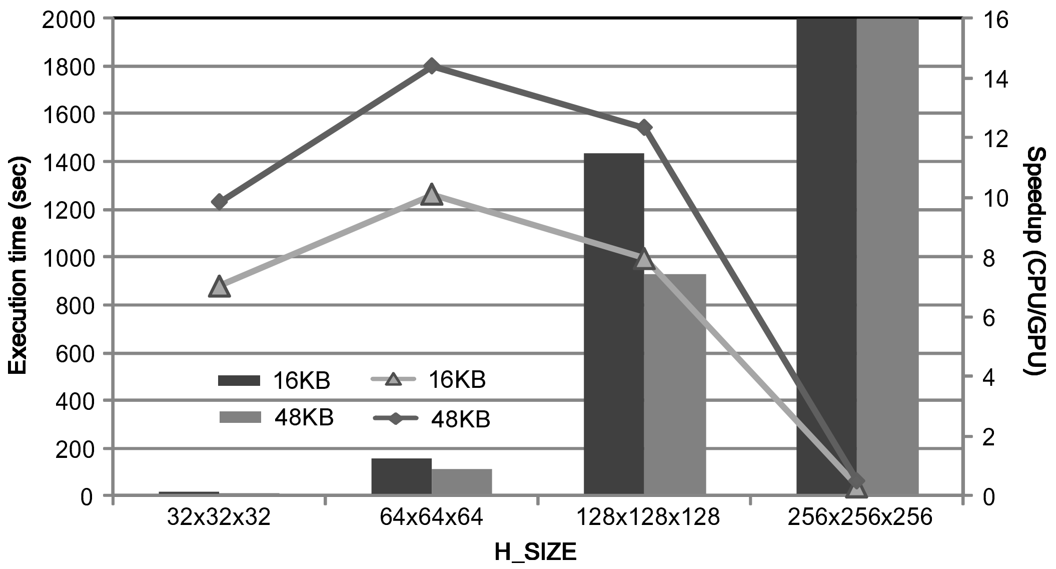
According to the discussion about the memory usage of the sliding window method above in the case of double precision, the memory cost is estimated using the equation below:

$$\underbrace{H\_SIZE \times 88}_{\text{Matrix H}} + \underbrace{Number\ of\ Blocks \times H\_SIZE \times 4 \times 8}_{\text{Figure 4.1 (1)}} \\ + \underbrace{Number\ of\ Blocks \times (BS \times BS \times 8 + N \times 8)}_{\text{Figure 4.4}}$$

4.2.3 Discussion on Full Map And Sliding Window Method

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. When we apply the CSR 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 non-zero elements, therefore the sizes of A, IA' and JA is $H\_SIZE \times 7 \times 8$ 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 CSR format needs only 1 GBytes. Thus, the size problem can be moderated by the CSR 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



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

Figure 4.5: Performances of the sliding window method with the CSR format comparing among 16KByte/48KByte cache and the speedup.

of NVIDIA C2050 case, the GPU assigns 16Kbytes to the L1 cache memory and 48Kbytes to the shared memory accessed by the threads within a block. 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.

Evaluation for The Sliding Window Method With CSR Format

We have confirmed that the CSR format is very effective for the performance in the previous section. Therefore, this section additionally applies the format to the sliding window method. Moreover, the method 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.

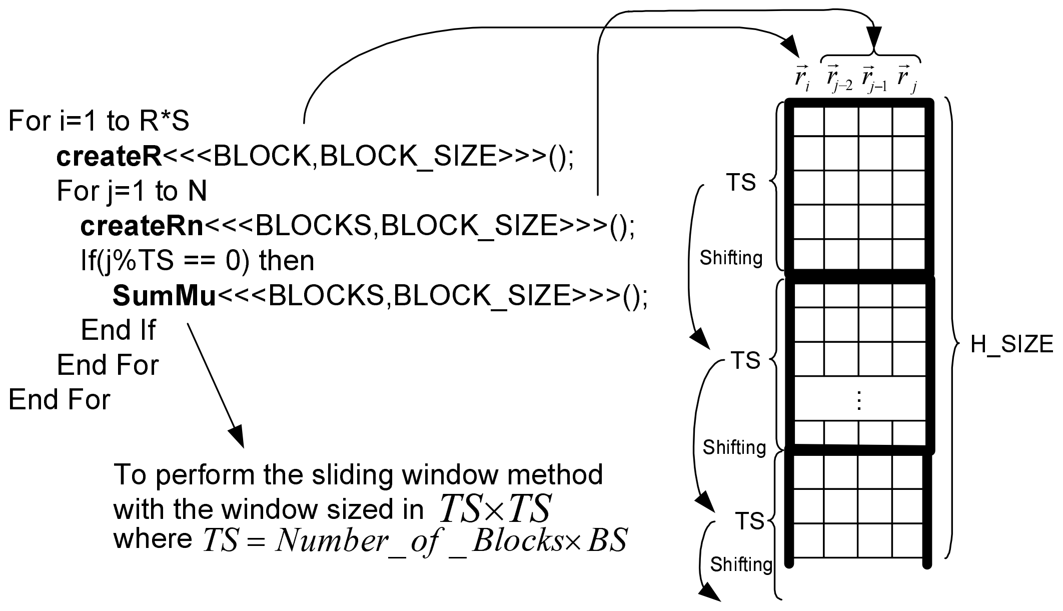
Figure 4.5 shows the performances and the speedups of the sliding window method applying the CSR format. Although the sliding window method includes many control code to reduce $\tilde{\mu}_i$ to the window memory, it achieves better performance than the full map method comparing the execution times of $H\_SIZE = 32 \times 32 \times 32$. The sliding 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 as this speedup factor is usually larger than the maximum number of processing cores in a recent CPU. 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.

4.2.4 Implementing KPM With Multiple CUDA Kernels

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 Figure 4.5, the speedup degrades very much because in order to reduce the required amount of memory we have to reduce 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 through distributing the program into several kernel functions.

The part where the sliding window method consumes memory at most is the operation (1) in Figure 4.1. To reduce the required amount of memory for the operation (1), we propose a technique that divides the operations to multiple kernel



```
For i=1 to R*S
    createR<<<BLOCK,BLOCK_SIZE>>>();
    For j=1 to N
      createRn<<<BLOCKS,BLOCK_SIZE>>>();
      If(j%TS == 0) then
        SumMu<<<BLOCKS,BLOCK_SIZE>>>();
      End If
    End For
End For
```

To perform the sliding window method with the window sized in $TS \times TS$ where $TS = Number\_of\_Blocks \times BS$
$\vec{r}_i$ $\vec{r}_{j-2}$ $\vec{r}_{j-1}$ $\vec{r}_j$
TS
Shifting
TS
Shifting
TS
Shifting
H_SIZE

(a) Pseudo code Used in the divided kernel version invoked in CPU

(b) Memory consumption and thread assignment for generating $\vec{r}_n$

Figure 4.6: Implementation of the multiple kernel version.

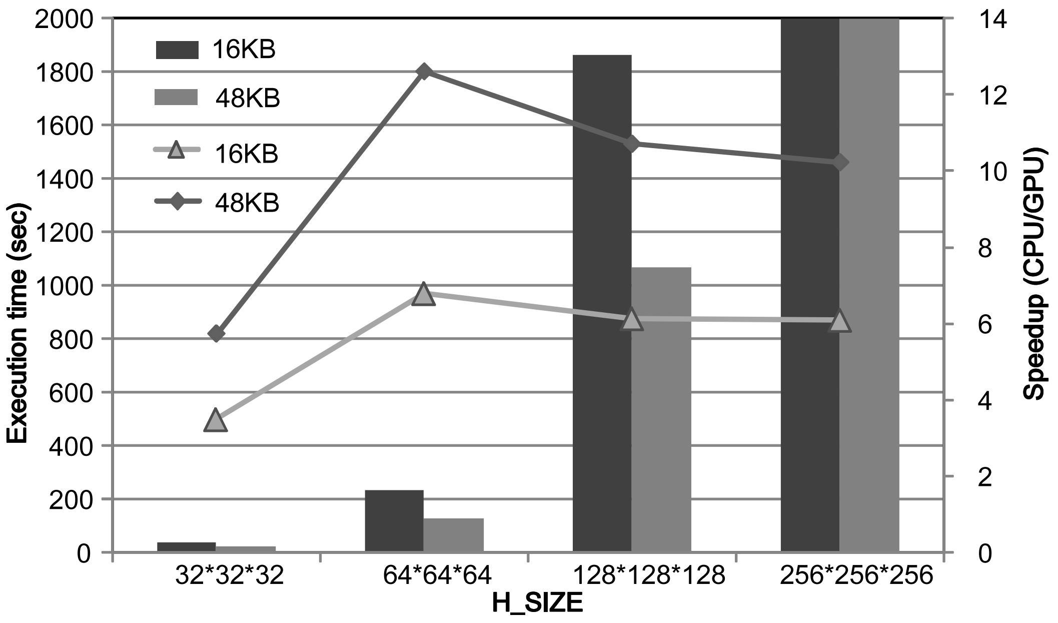
programs and calls the kernels repeatedly from the CPU. Here, we divide the operations into three kernels as shown in Figure 4.6 (a); the *createR* randomly generates $\vec{r}$, the *createRn* calculates $\vec{r}_n$ using $\vec{r}_{n-2}$, $\vec{r}_{n-1}$ and $\vec{r}$ and then also saves $\vec{r}_n \cdot \vec{r}$ into 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 area in GPU's global memory with size of $H\_SIZE \times 4 \times 8$ bytes. The elements of the $\vec{r}_n$ are calculated in parallel based on the thread numbers ($TS = NumberofBlocks \times BS$) as illustrated in Figure 4.6. The *createR* is called once per N iterations of *createRn*. The *createRn* kernel also saves a part of μ_i into the sliding window used by the *SumMu* kernel. 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, through dividing 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 Figure 4.1(1) and the parallelism become 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 for eath iteration. The overhead may 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 Figure 4.7 when 32 thread blocks and 128 BS are used for all kernels. Therefore, the total number of threads working concurrently is 4096. The sliding window is 4096×4096. The data cache size is also exchanged between 16Kbytes and 48Kbytes. As we have expected, the overhead for loading/discarding the kernel to/from GPU causes performance degradation. 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 of the massive 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 4.8 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 1.10-1.65 times. On the other hand, the effect of the CSR format shown in (b) is remarkable because it achieves a drastic performance improvement both for CPU and GPU versions. The speedup increases as the problem size increases. Therefore, it is very clear that the CSR 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 H_SIZE of $32 \times 32 \times 32$, the performance speedup increases if the comparison is performed with a larger



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

Figure 4.7: Performances and speedup of the divided sliding window method with the CSR format with 16KByte/48KByte data cache.

H_SIZE. 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 2600 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 of a single thread. It can be concluded that the GPU-based implementation achieves much higher performance than the CPU-based one because of the high parallelism and high memory bandwidth.

Here, let us compare the performances in floating point operations per second (FLOPS). The total FP operations in double precision consist of a) the generation of random vector $\vec{r}$, b) the generation of vector $\vec{r}_n$ recursively, c) the vector dot products for $\tilde{\mu}_i$. Part a) occupies very few percentage in the total execution time using the CUDA's random number generation library. Ignoring the part a), the computation amount consists of part b) and part c) can be expressed by

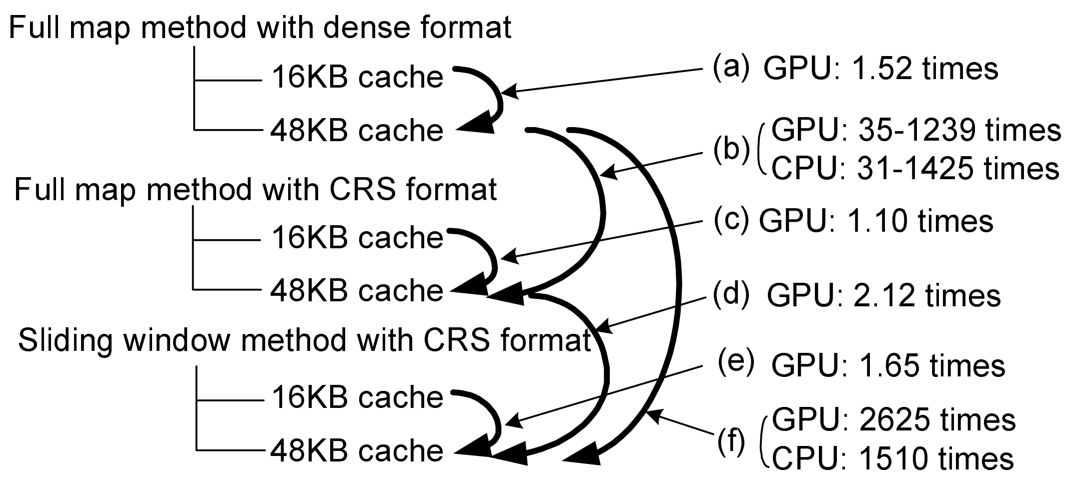


Figure 4.8: Performance comparisons between each implementation of KPM on GPU and CPU

$$\underbrace{H\_SIZE \times 16 \times N \times R \times S}_{\text{Part b)}} + \underbrace{H\_SIZE \times 2 \times N \times R \times S}_{\text{Part c)}}$$
$$= H\_SIZE \times 18 \times N \times R \times S \qquad (4.3)$$

When we apply the actual parameters ($N = 128$, $R = 14$ and $S = 128$) to the multiple kernel version, the performance achieved on Tesla C2050 is about 7.7 GFLOPS for $H\_SIZE = 256 \times 256 \times 256$. 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. Using the same parameters, the CPU achieves 0.9 GFLOPS, only about 3.0% of the peak performance of the Core i7 processor. The low GFLOPS values comes as no surprise as it has explained in Chapter 2 that KPM is a highly memory bound algorithm. It is also consistent with other research results on SpMV shown in Section 2.5. Therefore, the key technique that improves the entire performance of the KPM is to reduce the number of I/O operations for read/write the compressed sparse matrix in the global memory.

4.3 Discussion and Summary

This chapter demonstrates how to calculate DOS function using KPM. Because of the approximation nature of KPM, it can be used to simulate very large Hamiltonian, e.g. up to $256 \times 256 \times 256$. Because KPM contains a fine-grain recursive part, which is hard to parallelize using thread level parallelism on a supercomputer or a cluster computer, here we focuses on parallelizing KPM on a massively parallel environment, i.e. GPU, aiming to achieve high parallelism for more speedups than the recent CPUs.

In order to evaluate KPM's features in terms of parallelization methods, in this chapter several implementations on GPU has been demonstrated, namely, the full map method with/without CSR format, the sliding window method and the multiple kernel implementation. Their performances are also evaluated and analyzed comparing with each other. To enlarge available simulation sizes and at the same time to enhance the performance, this chapter also describes additional optimization techniques and proposed an implementation of multiple kernel version. The performance evaluation indicates that using multiple kernel implementation, the performance of KPM could be accelerated by over 10 times for Hamiltonian matrix size of $256 \times 256 \times 256$.

Chapter 5

KPM to Evaluate Local Density of States

Local Density of States (LDOS), known as the spatially resolved density of states, is another fundamental quantity in quantum mechanics since it provides important information to study the effects of randomness and crystal imperfection that might exist in materials. Namely, a transition between conductive metal and non-conductive insulator driven by randomness (called Anderson transition [91–93]) is manifested by a change of behaviour for LDOS [94, 95]. Regarding the numerical methods to solve the LDOS, KPM is effective due to its reduced complexity and adaptive accuracy.

In this chapter, applying KPM, a high performance GPU-based LDOS implementation is proposed. Unlike the DOS application in Chapter 4, which is implemented only on a single GPU, for LDOS application here we have to resort to MPI to parallelize the task since evaluating LDOS involves much more computation amount than DOS.

5.1 Algorithm for LDOS

Based on mathematical principles introduced in section 2.3, Algorithm 3 demonstrates how to evaluate LDOS function using the KPM. The output of this algorithm

Algorithm 3 Calculate LDOS applying KPM

```
Require: Hamiltonian matrix $H$ of dimension $D \times D$
Require: Set of vectors $\rho_j(E)$ to store ldos
Require: Integer S to represents the sampling number
Require: Integer $M$ to represents the number of moments
Require: Vector $\vec{r}$ , $\vec{r}_1$, $\vec{r}_2$, $\mu$
 1: Construct matrix H according to the Anderson disorder model
 2: Normalize matrix H
 3: for $j = 1 \to S$ do                                   ⎫
 4:    Create a random vector $\vec{r}$                    ⎪
 5:    for $i = 1 \to N$ do                                ⎪
 6:       if $i = 1$ then                                  ⎪
 7:          $\vec{r}_1 \leftarrow \vec{r}$                ⎪
 8:          $\mu[1] \leftarrow \mu[1] + \vec{r} \cdot \vec{r}_1$   ⎪
 9:       else if $i = 2$ then                             ⎪  Part 1
10:          $\vec{r}_2 \leftarrow H \times \vec{r}_1$     ⎬  (line 3 to 17)
11:          $\mu[2] \leftarrow \mu[2] + \vec{r} \cdot \vec{r}_2$   ⎪
12:       else                                             ⎪
13:          $\vec{r}_1 \leftarrow 2 \times H \times \vec{r}_2 - \vec{r}_1$   ⎪
14:          $\mu[i] \leftarrow \mu[i] + \vec{r} \cdot \vec{r}_1$   ⎪
15:          Swap pointers of $\vec{r}_1$ and $\vec{r}_2$  ⎪
16:       end if                                           ⎪
17:    end for                                             ⎭
18:    for $s = 1 \to L$ do                                            ⎫  Part 2
19:       Construct function value $\rho_j(E_s)$ applying Equation 2.1 ⎬  (line 18 to 20)
20:    end for                                                         ⎭
21: end for
```

is a set of LDOS functions $\rho_j(E)$, where $j \in \{1, 2, ..., S\}$. At the beginning it defines three integer parameters: N, S and D, where S represents the number of random vectors. Integer N denotes the number of moments of Chebyshev expansion, it is used to control the accuracy (truncation) of eigenvalue spectrum. S denotes the number of samples in energy interval, which fit into (-1, 1) as required by the Chebyshev expansion. D is the size of matrix H (i.e. the number of elements in H is $D \times D$), here H is a sparse matrix compressed using the ELL format and $D \equiv H\_SIZE$.

At first Hamiltonian matrix H has to be constructed and normalized, here Anderson disorder model is applied. Then it (line 4) chooses a random vector $\vec{r}$ of the form $(0, ..., 0, r_k, 0, ..., 0)$ in which $r_k \equiv 1$ and k is randomly chosen. For each $\vec{r}$, a new vector $\vec{r}_i$ is generated at every iteration of the recursive part illustrated by lines from

6 to 15. Here only two vectors, $\vec{r}_1$ and $\vec{r}_2$, are used to perform the recursion through swapping the pointers after each iteration as noted by line 15. A dot product of $\vec{r}$ and $\vec{r}_i$ is performed and produces Chebyshev expansion moment coefficient μ_i.

Given the μ_i generated, lines from 18 to 20 reconstructs LDOS function (Equation 2.23) through the similar recursive operations using Equation 2.1.

Time Complexity Discussion

According to the algorithm above, let us discuss the complexity. The algorithm can be roughly divided into two parts: 1) calculation of the coefficient (from line 4 to line 17) μ and 2) reconstruction of LDOS function (line 18 to line 19).

In Part 1), due to the ELL format, the number of non-zero elements in a row of the matrix H is fixed to seven in any case of H_SIZE, the number of operations for the recursion denoted by line 13 is $16 \times D$, thus the time complexity is O(D). Considering the outside loops between the line 3 to 17, the total complexity of Part 1) is $O(DSN)$.

As for Part 2), the time complexity is becomes O(LSN), where L represents the number of energy samples between (-1, 1). Therefore, the total time complexity is presented by $O(DSN) + O(LSN)$. Note that the latter part may be ignored when integer D is large enough, e.g. $D = 40^3$ and $L = 256$. Therefore,in this study we use the approximate time complexity for LDOS, which is

$$O(DSN) \tag{5.1}$$

Let us recall the time complexity of DOS, which is $O(DRSN)$ as derived in section 4.1. Unlike DOS application in which RS and D is not bounded, suggesting RS can be kept as constant for increasing D, LDOS requires that S should be increased for a large D, e.g. in this study we choose $S = D$. This bound relation results in much larger computational amount to evaluate LDOS functions than DOS function. This is an important reason why in this chapter MPI is employed to extend the parallelism to cluster scale.

5.2 Implementation on CPU

Using the algorithm mentioned above, we implemented the algorithm on a CPU-based platform that has two Xeon E5645 2.4GHz 6-core CPUs and 12 GByte memory. 12 cores can work concurrently sharing the memory. We implemented Algorithm 3 in a straightforward manner applying the row major dynamic assignment for vectors and matrices. Considering that the vector $\vec{r}$ is chosen randomly and no relationship exists among the different vectors, the most outside loop for S can be divided among CPU cores and therefore each thread assigned to a CPU core calculates $S/num\_of\_threads$ loops. For the computations among CPU cores within a node, the matrix H is shared through the threads without explicit communications.

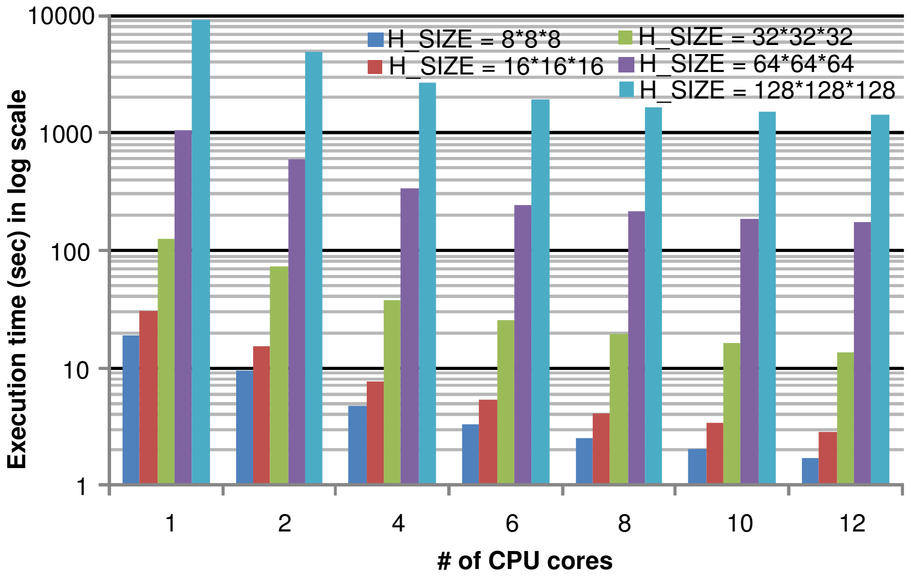
Figure 5.1 shows the performance of our implementation by varying the number of CPU cores (i.e. the number of threads) from one to twelve. When the H_SIZE is small (e.g. 16^3), the performance linearly increases with the number of parallelism, for example, the doubled number of threads leads to half execution time. However, when H_SIZE becomes large, the linearity is broken.

We can further examine the performance in FLOPS. Here, ignoring the Part 2, FLOPS value can be roughly calculated by the following expression

$$(\underbrace{16 \times D \times N \times R}_{\text{Part 1}})/T \tag{5.2}$$

where T is the execution time in second. As we can see in Figure 5.1, the best performance achieved at 32^3 for 12 cores, which is 10.3GFLOPS. For larger H_SIZE, the GFLOPS begins to drop significantly. Probably because the memory begins to be saturated as increasing the number of CPU cores. The cache miss rate also may increase for larger H_SIZE.

This represents the KPM for solving LDOS has memory intensive computational characteristics. To avoid this performance saturation, we have two methods to improve the performance: one is to reduce the number of memory accesses in the algorithm and another is to apply a processor resource that has a large memory I/O bandwidth. Since GPU can provides much higher bandwidth than CPU, it is



Core	1	2	4	6	8	10	12(GFLOPS)
8^3	19	9	5	3	2	2	2 (2.2)
16^3	30	15	8	5	4	3	3 (6.7)
32^3	127	73	37	26	19	16	14 (10.3)
64^3	1045	591	334	245	213	184	173 (6.4)
128^3	9300	4874	2657	1926	1686	1537	1448 (6.1)

Figure 5.1: Performance scaling on the multicore CPUs for evaluating LDOS.

reasonable for applying GPU to solve LDOS.

5.3 The Design and Implementation on GPU

In this section, we firstly review the parallelization design for Algorithm 3, which is followed by the implementation accordingly. To exploit high parallelism, we also extend the implementation to GPU cluster.

5.3.1 Parallelization Methods on Single GPU

The first approach to speedup the KPM for solving LDOS is to assign the computational intensive part to the stream processors on a single GPU. We divide the algorithm into two different kernel functions corresponding to Part 1 and Part 2 in the algorithm and the kernels are invoked in the order. Let us explain the design detail below:

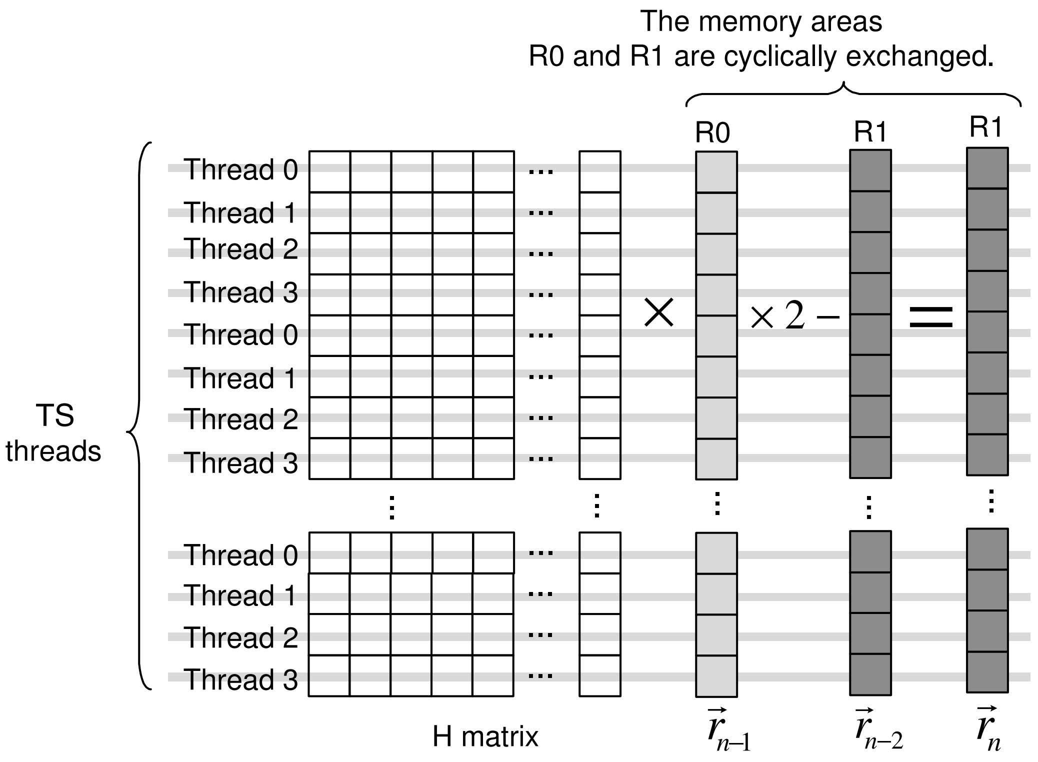


Figure 5.2: Memory allocation and parallelization method for the recursive calculation in Part 1. The memory area R0 and R1 are reused at every iteration.

Parallelization of Part 1

This part produces the Chebyshev polynomial coefficients μ_n for a given j. It is the most calculation intensive with the recursive computation for $\vec{r}_i$ that performs a matrix multiplication and a subtraction of a vector. As depicted in Figure 5.2, the multiplication between each row of H and the vector $\vec{r}_{n-1}$ are assigned to a single thread, which also subtract the corresponding element of vector $\vec{r}_{n-2}$. This result in the concurrent execution with TS threads reducing the dot products of $\vec{r}_n$ and $\vec{r}$ for μ_n. Due to the recursive computation, this parallel execution must be synchronized after the iteration. Even if this fine-grain work assignments to many threads increases the parallelism, this synchronization becomes a fatal overhead when implemented on a CPU-based cluster across its nodes via a network because the communication overhead among nodes is very large. Therefore, this part is very suitable to work efficiently on GPU architecture.

Parallelization of Part 2

This part reconstructs the LDOS function $\rho_j(E)$ applying the coefficients calculated by Part 1. Here the loop from 18 to 20 calculate the value $\rho_j(E_s)$ for each $E_s \in (-1,1)$. As a straightforward implementation, E_s is chosen in uniform distribution. Since $\rho_j(E_s)$ is independent regarding different s. we can assign a single thread s for the calculation of a $\rho_j(E_s)$ and expect the concurrent execution of L threads. Through this way, Part 2 is also fully parallelized up to L concurrent threads.

Memory Resource Assignment

Let us estimate amount of memory consumed by the KPM for solving LDOS. The amount of memory needed for the algorithm is similar to the one for the CPU-based implementation as shown in section 5.2. The required memory areas are mainly consisted of three memory areas which are: 1) the matrix H, 2) the vectors $\vec{r}, \vec{r}_1, \vec{r}_2$ for Part 1, and 3) LDOS function $\rho_i(E)$ for Part 2.

Regarding the area 1), we use the ELL format to compress the sparse matrix. The non-zero values take $H\_SIZE \times 8 \times 7$ bytes and the column indices matrix for the non-zero elements is estimated as $H\_SIZE \times 4 \times 7$ bytes because the indexes are based on integer numbers. Thus the total size of H is $H\_SIZE \times 84$ bytes. The matrix H is stored in column major order to trigger coalescing memory access by the concurrently running threads.

The area 2) employs the optimization due to the recursive operation. Unlike the DOS application which requires tree vectors of H_SIZE length as introduced in chapter 4, here it reduces to only two $\vec{r}$ vectors (i.e. $\vec{r}_1$ and $\vec{r}_2$ because the calculation is performed by exchanging the memory pointers. $\vec{r}_0$ is created by simply copying the data from $\vec{r}$ to R0. As for $\vec{r}_1$, the vector memory R0 is used as the read memory and the $\vec{r}_1$ is generated to the vector memory R1. The generation of the following $\vec{r}_n$ is illustrated by Figure 5.2, in which the pointers of R0 and R1 are exchanged. The memory for this area equals to $H\_SIZE \times 8 \times 2$ bytes as the data is stored in double precision.

```
double *r, *μ, *R0, *R1;
...
for(i=0;i<R;i++){
        int r = random();
        for(j=0;j<N;j++){
                cukpmCreateRn<<<>>>(R0, R1, r, μ,...);

                double *pTemp = R0;
                R0=R1;
                R1=pTemp;
        }
        ...
        cukpmCreateDOS<<<>>>(μ,...);
        ...
}
```

Figure 5.3: Code for GPU implementation.

Finally area 3) is allocated for S LDOS functions of $\rho_j(E)$, each of which takes L samples. Therefore when the value is based on a double precision floating point, this memory area occupies $L \times S \times 8$ bytes. Totally, the KPM for solving LDOS requires $100 \times H\_SIZE + 8 \times L \times S$ bytes. For example, when we choose $H\_SIZE = 128^3$, $S = 2048$ and $L = 1024$, about 224MBytes memory is required for the algorithm.

5.3.2 Implementation on Single GPU

Figure 5.3 demonstrates the host side code of the implementation for single GPU. It executes two CUDA kernel functions called *cukpmCreateRn* for Part 1 and *cukpmCreateLDOS* for Part 2. The advantage to divide the algorithm to multiple kernels provides easy control of memory resource management because each kernel requires Part 1 and Part 2 respectively and independently, and also the reduced synchronization timings because the result of a kernel is passed to the host side to reset the parallel execution.

cukpmCreateRn performs Part 1 that includes the recursive operations and the dot products between $\vec{r_i}$ and $\vec{r}$. The parallelism equals to $TS = BS \times num\_blocks$.

Kernel function *cukpmCreateDOS* reconstructs LDOS function using the polynomial coefficient μ generated by Part 1. Figure 5.4 shows how to parallelize LDOS

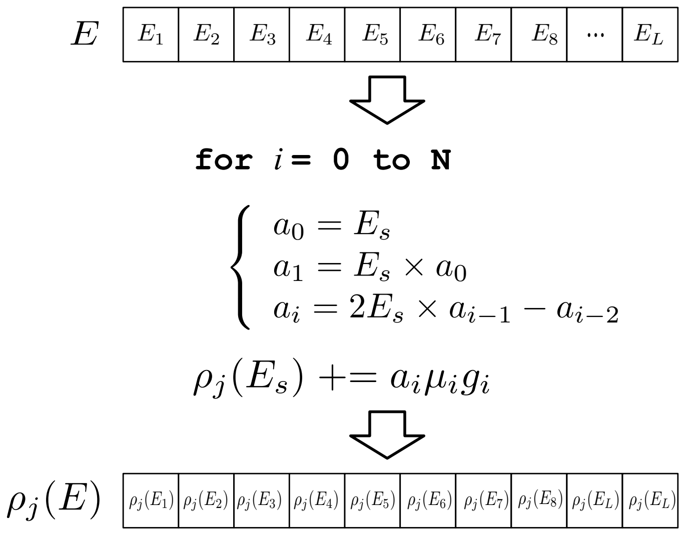

E
E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 ··· E_L
for i = 0 to N
$a_0 = E_s$
$a_1 = E_s \times a_0$
$a_i = 2E_s \times a_{i-1} - a_{i-2}$
$\rho_j(E_s) \mathrel{+}= a_i \mu_i g_i$
$\rho_j(E)$
$\rho_j(E_1)$ $\rho_j(E_2)$ $\rho_j(E_3)$ $\rho_j(E_4)$ $\rho_j(E_5)$ $\rho_j(E_6)$ $\rho_j(E_7)$ $\rho_j(E_8)$ $\rho_j(E_L)$ $\rho_j(E_L)$

Figure 5.4: Memory allocation and parallelization scheme of LDOS calculation in Part 2.

reconstruction. For each E_s, the s-th thread produces $\rho_j(E_s)$ through the recursion. Therefore, the total number of parallelism is L.

According to the design and implementation proposed above, Part 1 and Part 2 in the algorithm are separated into different kernels that each kernel has the largest number of parallelism to exploit the available concurrency of the calculations.

5.3.3 Extend to GPU Cluster

Applying the implementation for a single GPU, we parallelize the KPM for LDOS on a GPU cluster using MPI. As shown in Figure 5.5, firstly we focus on the parallelization of the outside loop of S, which is distributed to several nodes, e.g. three nodes in the figure, therefore each nodes calculate $S/3$ loops. Since each node has two Tesla GPUs, $S/3$ loops are future distributed by half ($S/6$) to each GPU which is used for the paralelization of matrix vector operations in the recursion. Since S is bounded with the matrix size D as explained in section 5.1, the resulted large number of S by a large matrix could be effectively parallelized among large number of nodes in a supercomputer.

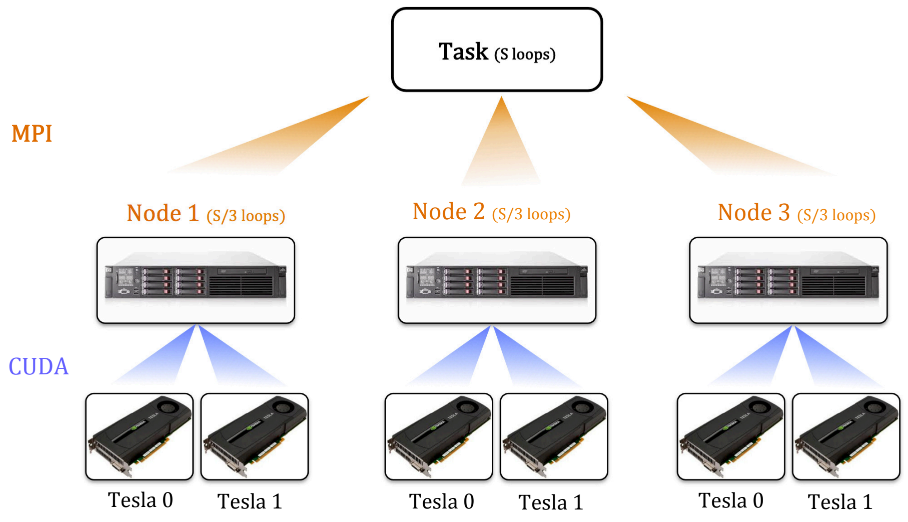


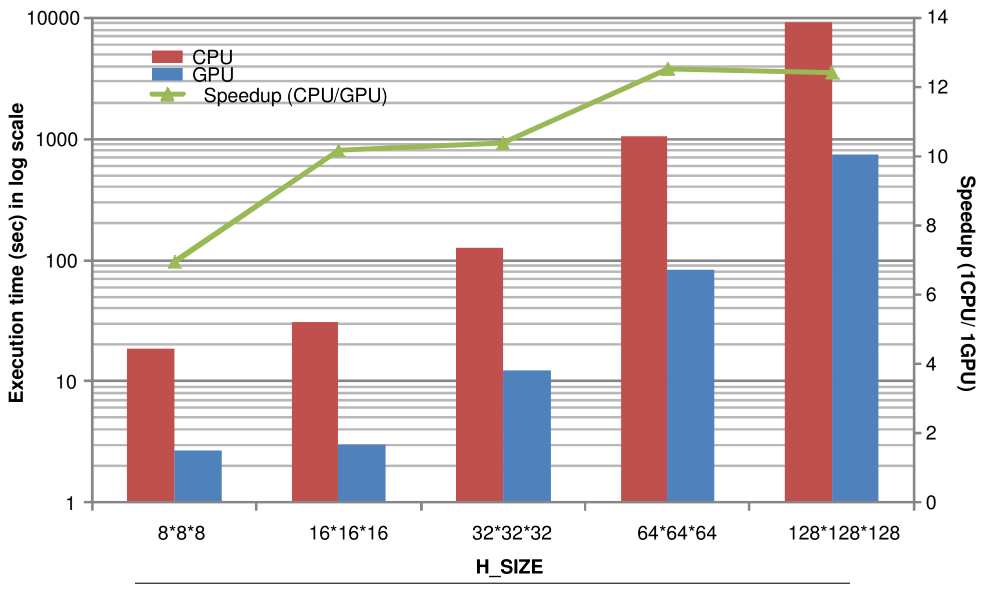
Figure 5.5: Parallelization of generation of LDOS on cluster

5.4 Experimental Performance Analysis

Let us see the performance evaluation of LDOS on both CPU and GPU, we performed two types of experimental performance evaluations. One is the performance comparison of between the single GPU and the single CPU core to investigate the impact of parallelism brought by the GPU. Another comparison investigates the scalability in the cluster environment as the CPU-based implementation is also parallelized using MPI to many computing nodes regarding the parameter S. In our experiment, $BS = 128$ and *num_blocks* $= 64$ are applied because the performance becomes the best using these parameters according to our experiment.

5.4.1 Performance Evaluation of Single CPU and Single GPU

Figure 5.6 illustrates the performance of the KPM for solving LDOS on a single CPU core and single GPU through varying H_SIZE from $8 \times 8 \times 8$ to $128 \times 128 \times 128$. The bars depict the execution times of the LDOS on the GPU and the CPU in log scale. The line shows the speedups of GPU-based implementation against the CPU one (i.e. CPU time/GPU time). In the small H_SIZE case, the execution times does not scale linearly according to the complexity $O(SDN)$, which is an approximation for large D. But when the size is large enough, the $O(SDN)$ is followed, for example, the D of system 64^3 is 8 times of 32^3, which result in $1047/126 = 8.3$ times execution



H_SIZE	8^3	16^3	32^3	64^3	128^3
CPU	19 sec	31 sec	126 sec	1047 sec	9300 sec
GPU	3 sec	3 sec	12 sec	84 sec	750 sec

Figure 5.6: Performance comparison between single CPU and single GPU.

time on CPU. The CPU-based implementation has the performance turning point at the H_SIZE between $16 \times 16 \times 16$ and $32 \times 32 \times 32$. As a result of higher bandwidth of GPU, we can see that the speedup factor increases for larger H_SIZE, over 12x better performance is achieved on GPU finally.

In order to analyze the access saturation in the memory bus, we measured the effective bandwidth (between Cache and DRAM) on memory bus. Figure 5.7 shows the percentages of GPU's memory bus usage against the peak bandwidth (about 144GB/sec) and the GPU usage percentage denoted by lines, and the actual memory bus usage with bars measured by the NVIDIA profiler called *computeprof*. For the small size of H_SIZE, the memory bus usage rate is small. This means that the calculation amount is not much against the performance capability. This is also proved by the percentage of the GPU calculation amount that is less than 40%. When H_SIZE is increased, the memory bus usage increases to nearly the peak bandwidth. In the case of $128 \times 128 \times 128$, the bus usage ratio becomes 85%.

Due to the parallelism and the large memory bus bandwidth of GPU architecture, the performance of GPU-based implementation of the KPM for solving LDOS

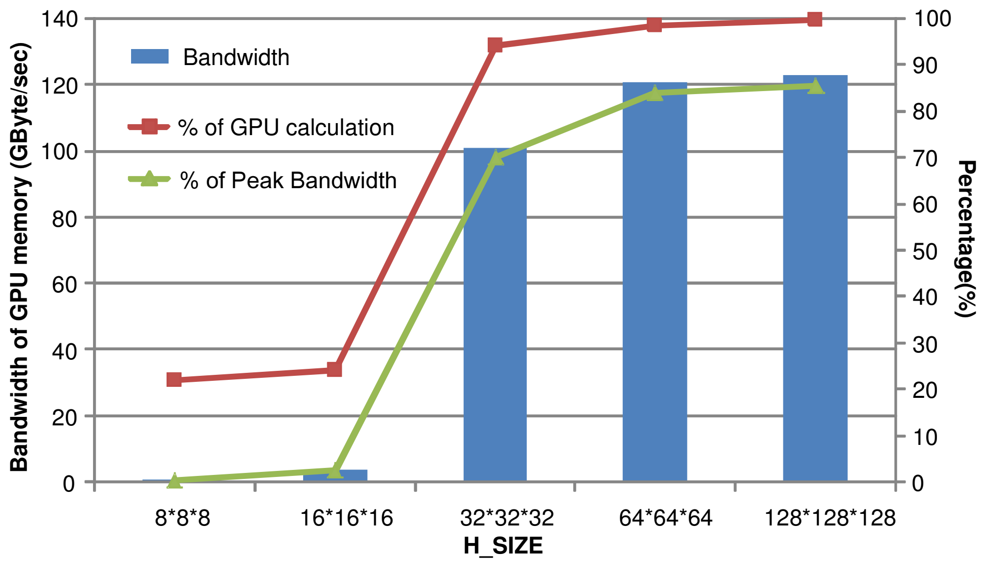
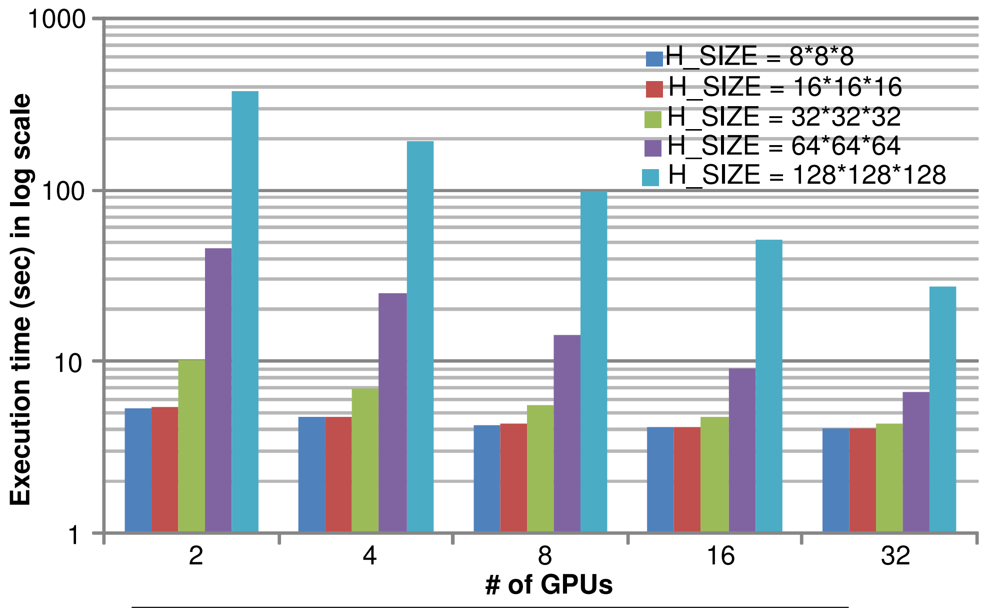


Figure 5.7: The realistic bandwidth and its effective ratio against peak (144 GByte/sec) of memory bus, the percentage of the GPU calculation comparing to the total execution time are also displayed.

shows about from 6 to 12 times better than the CPU-based one according to Figure 5.6 without considering memory bus saturation of the CPU. Actually, since the performance of 12 CPU cores is not equivalent to this characteristics (considering the case of $H\_SIZE = 128 \times 128 \times 128$, the estimated execution time on 12 CPU cores should become 9300 sec / 12 = 775 sec. However, Figure 5.1 shows 1448 sec in the case of 12 cores), providing larger bandwidth is the most important to keep high performance for the algorithm. According to the analysis above, our approach to implement the algorithm on GPU utilizes not only its highly parallel execution of the calculation, mainly of Part 1, but the large bandwidth provided by the GPU hardware.

5.4.2 Performance Evaluation on GPU cluster

We parallelized the KPM for solving LDOS to the GPU cluster to investigate the scalability of the performance according to the parallelization scheme described in section 5.3.3. In a node of the cluster, two GPUs are connected with CPU via PCI express. Therefore, the case with more than two GPUs includes communication via Infiniband using MPI among the nodes. The execution steps on GPUs includes 1) input data generated from a lattice model, 2) data distribution among nodes, 3)



# of GPUs	2	4	8	16	32
8^3	7 sec	9 sec	8 sec	5 sec	4 sec
16^3	8 sec	8 sec	9 sec	8 sec	4 sec
32^3	14 sec	12 sec	10 sec	9 sec	4 sec
64^3	61 sec	35 sec	21 sec	12 sec	7 sec
128^3	429 sec	219 sec	112 sec	60 sec	31 sec

Figure 5.8: Performance comparison of parallelizing into multiple CPUs and GPUs.

configuring GPU execution and initializing data on the GPUs, 4) invoking calculation and finally 5) gathering data from GPUs. The execution time in this experiment is measured from 2) to 5). We also parallelize the CPU-based implementation using MPI dividing the work regarding the loop of S. The execution time evaluation for CPU does not include the steps 2), 3) and 5), we measured only 4) on the CPU-based cluster.

Figure 5.8 shows the execution times of the MPI version to examine the performance scaling through varying the H_SIZE from $8 \times 8 \times 8$ to $128 \times 128 \times 128$ as well as the number of GPUs from 2 to 32. An important observation here is that: Unlike the poor performance scaling among multiple CPUs as shown in Figure 5.1, the data shown in Figure 5.8 indicates an almost ideal performance scaling with number of GPUs. For example, for 128^3 the double number of GPUs always leads to half execution time. It is one of the advantages of GPU cluster that multiple CPU cores share the same memory bandwidth while every GPU has their independent

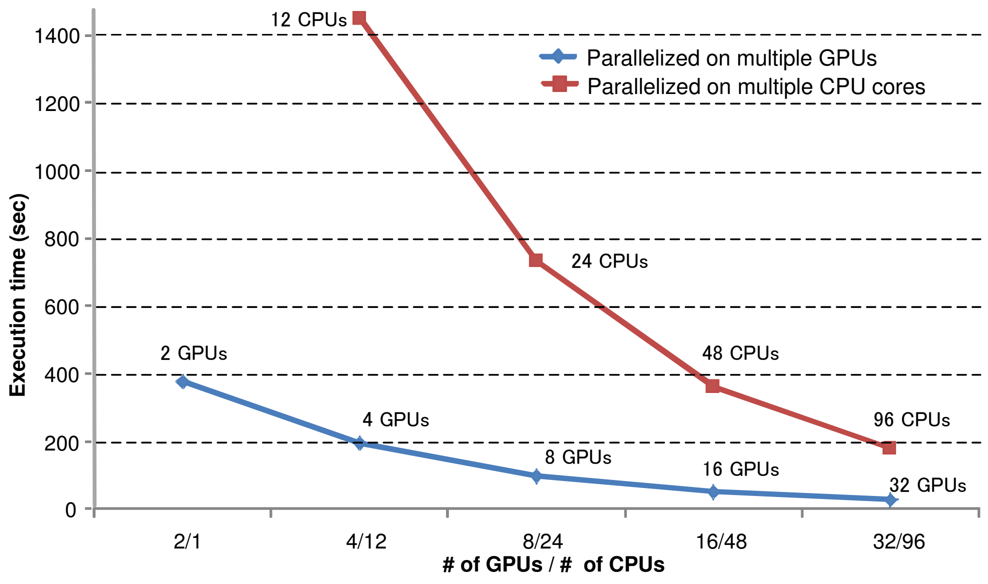


Figure 5.9: Performance comparison between multiple CPUs and GPUs in the case of H_SIZE=128^3.

bandwidth to access the global memory. This feature is specially useful for memory bounded algorithm like KPM.

In the case of $H\_SIZE = 128 \times 128 \times 128$ on 32 GPUs, the cluster achieves about 319 GFLOPS. Since 96 CPU cores achieve about only 50 GFLOPS, the performance on the GPU cluster with larger H_SIZE can provide about 6 times higher performance.

The performance scaling regarding number of GPUs and CPUs are also investigated and shown in Figure 5.9, in which the result is obtained in the case of $H\_SIZE = 128 \times 128 \times 128$. Because there are very few communications between MPI threads, the performance scaling linearly regarding the number of nodes, each of which contains 12CPUs and 2GPUs. In addition, it can be observed that two GPUs require same execution time with 24 CPU cores, this relation could be also observed for 4GPUs and 96CPUs, suggesting that for LDOS solved using KPM, a single GPU is equivalent of 24 CPU Xeon E5645 cores.

5.5 Discussion and Summary

This chapter proposes implementations of LDOS which is another essential quantity used condensed matter physics. Focusing on the memory bounded characteristics of the KPM, we parallelize the algorithm to a single GPU as well as extend it to the GPU cluster using MPI. Through comparing the performance of multiple CPUs and GPUs, It can be observed that the single GPU version shows about 12 times higher performance comparing to a single CPU core. The MPI version on the GPU cluster shows the equivalent performance of 24 times computational capacity of CPU-based implementation.

Chapter 6

KPM for Monte Carlo Simulations of Double Exchange Model

In addition to solve the DOS and LDOS, KPM can be also combined into other numerical methods such as quantum Monte Carlo method to boost the performance through evaluating the correlation functions, e.g. Green's function [17, 19]. In this chapter based on the Green's-function-based Monte Carlo method (GFMC), a high performance GPU implementation is proposed. The performance evaluation indicates that applying GPU the computational performance can accelerated by over 30 times comparing to a single CPU core [96].

6.1 Model and Method Formulation

6.1.1 Double Exchange Model

The double exchange (DE) model [44, 97] describes electrons interacting the classical spins via Hund's rule coupling. The Hamiltonian is given by

$$H = -t \sum_{\langle i,j \rangle} \left(c_{i\sigma}^{\dagger} c_{j\sigma} + \text{H.c.} \right) + J_H \sum_{i,\alpha\beta} \vec{S}_i \cdot c_{i\alpha}^{\dagger} \vec{\sigma}_{\alpha\beta} c_{i\beta}, \tag{6.1}$$

where $c^{\dagger}_{i\sigma}$ is a creation operator of electron at site i and with spin $\sigma = (\uparrow, \downarrow)$, $\langle i, j\rangle$ runs over a pair of nearest neighbor sites i and j, $\vec{S}_i$ is the classical spin at site i with its normalization $|\vec{S}_i| = 1$, and J_H denotes the Hund's rule coupling with $J_H > 0$. In the limit of infinite J_H ($J_H \to \infty$), the Hamiltonian becomes

$$H = -\sum_{\langle i,j\rangle} t_{ij} \left(c^{\dagger}_{i\sigma} c_{j\sigma} + \text{H.c.} \right) \tag{6.2}$$

with the hopping amplitude

$$t_{ij} = \cos\frac{\theta_i - \theta_j}{2} \cos\frac{\phi_i - \phi_j}{2} + i \cos\frac{\theta_i + \theta_j}{2} \sin\frac{\phi_i - \phi_j}{2}, \tag{6.3}$$

$$\tag{6.4}$$

where θ_i and ϕ_j denote the polar and the azimuthal angles of the classical spin $\vec{S}_i$, respectively.

The grand partition function of the DE model is written as

$$Z = \prod_i^N \int d^3\vec{S}_i \text{Tr}_\text{e} \left[\exp(-\beta H(\{\vec{S}_i\}) - \mu N \right], \tag{6.5}$$

where $\beta = 1/T$ is inverse of temperature T, μ is the chemical potential, N is the total number operator of electrons, and $\text{Tr}_\text{e}[\cdots]$ indicates the trace over the electron degrees of freedom in the Fock space. The trace over classical spin degrees of freedom, i.e., $\prod_i^N \int d^3\vec{S}_i P(\{\vec{S}_i\})$, is evaluated by the Monte Carlo importance sampling with its weight $P(\{\vec{S}_i\})$ for a given spin configuration $\{\vec{S}_i\}$,

$$P(\{\vec{S}_i\}) = \text{Tr}_\text{e} \left[\exp(-\beta H(\{\vec{S}_i\} - \mu N) \right] = \exp[-\text{S}_\text{eff}(\{\vec{S}_i\})], \tag{6.6}$$

where

$$S_\text{eff}(\{\vec{S}_i\}) = -\sum_i^N \log(1 + e^{-\beta[\epsilon_i(\{\vec{S}_i\}) - \mu]}) \tag{6.7}$$

$$= -\int \log(1 + e^{-\beta[E-\mu]}) \rho(E) dE, \tag{6.8}$$

ϵ_i is the i-th eigenvalue of Hamiltonian $H(\{\vec{S}_i\})$, and $\rho(E)$ is the electron density of state (DOS) for a given spin configuration $\{\vec{S}_i\}$. Using the Metropolis algorithm [98, 99], the possibility $P(\{\vec{S}_i\} \to \{\vec{S}'_i\})$ of accepting a new spin configuration $\{\vec{S}'_i\}$ is given by

$$P(\{\vec{S}_i\} \to \{\vec{S}'_i\}) = \frac{P(\{\vec{S}'_i\})}{P(\{\vec{S}_i\})} = e^{-\mathrm{S_{eff}}(\{\vec{S}'_i\})+\mathrm{S_{eff}}(\{\vec{S}_i\})}.$$

Therefore, the quantity Δ_{seff} define by

$$\Delta_{\text{seff}} = \mathrm{S_{eff}}(\{\vec{S}'_i\}) - \mathrm{S_{eff}}(\{\vec{S}_i\}) = -\int \log(1 + e^{-\beta[E-\mu]})(\rho'(E) - \rho(E))dE \qquad (6.9)$$

is all we need to carry out the Monte Carlo calculation. Here, $\rho'(E)$ is DOS for a given spin configuration $\{\vec{S}'_i\}$.

Using full diagonalization method, we can exactly diagonalize $H(\{\vec{S}_i\})$ with O(N^3) complexity to evaluate Δ_{seff} [100–102], or we can use Kernel Polynomial Method (KPM) with O(N^2) or O(N) complexity to estimate the DOS directly [18, 103]. However, the latter one with O(N) complexity must truncate the moment calculations. Here, we use the recently proposed GFMC method [19], as will be explained below for completeness.

6.1.2 Green-function-based Monte Carlo (GFMC) method

In this $O(N)$ GFMC method, we need to evaluate only several elements of the Green's function to calculate Δ_{seff} given in Eq. (6.9), provided that the change of spin configuration $\{\vec{S}_i\} \to \{\vec{S}'_i\}$ is local, i.e, only a single spin at site i is changed with the rest of spins unaltered. We first give the definition of the Green's function $G(z)$ in the complex plane:

$$G(z) = \frac{1}{H - z}, \;\; z = E + i\epsilon, \qquad (6.10)$$

where E is real and ϵ is a very small positive real. The Hamiltonian for a given spin configuration $\{\vec{S}_i\}$ ($\{\vec{S}'_i\}$) is denoted by H (H'), and the Hamiltonian difference Δ

is thus $\Delta = H' - H$.

The determinate of $G(z)(H' - z)$, i.e.,

$$d(z) := \mathrm{Det}\left[G(z)(H' - z)\right] = \mathrm{Det}\left[G(z)(H - z) + G(z)\Delta\right] = \mathrm{Det}\left[\mathbb{1} + G(z)\Delta\right] \tag{6.11}$$

$$= \mathrm{Det}\left[G(z)\right]\left[(H' - z)\right] = \prod_i^N \frac{1}{\epsilon_i - z} \prod_i^N \epsilon'_i - z, \tag{6.12}$$

has a special role in the GFMC method, where ϵ_i and ϵ'_i are the i-th eigenvalues of H and H', respectively. It is readily shown that

$$\begin{aligned}\frac{1}{\pi}\lim_{\epsilon\to 0}\mathrm{Im}\frac{\mathrm{d}\log(d(z))}{dz} &= \frac{1}{\pi}\lim_{\epsilon\to 0}\mathrm{Im}\left(\sum_i^N \frac{1}{\epsilon_i - z} - \sum_i^N \frac{1}{\epsilon'_i - z}\right)\\ &= \sum_i^N \delta(E - \epsilon'_i) - \sum_i^N \delta(E - \epsilon_i)\\ &= \rho(E) - \rho'(E).\end{aligned} \tag{6.13}$$

Thus, this equation in the left hand side can be used in Eq. (6.9), and Δ_{seff} is now described using $d(z)$:

$$\begin{aligned}\Delta_{\mathrm{seff}} &= \frac{1}{\pi}\lim_{\epsilon\to 0}\mathrm{Im}\int \log(1 + e^{-\beta[E-\mu]})\frac{\mathrm{d}\log(d(z))}{dz}dE\\ &= \frac{\beta}{\pi}\int \frac{1}{1 + e^{\beta(E-\mu)}}\lim_{\epsilon\to 0}\mathrm{Im}\log(d(z))dE.\end{aligned} \tag{6.14}$$

It is important to notice that we need only several elements of $G(z)$ to evaluate $d(z) = \mathrm{Det}\left[\mathbb{1} + G(z)\Delta\right]$. As an example, here we consider the simple cubic lattice with the nearest neighbor hopping, and we assume that only one spin at site o is changed: $\vec{S}_o \to \vec{S}'_o$. In the cubic lattice, site o has 6 nearest neighbors (NN) denoted by $\{n, e, s, w, t, b\}$. Then, the Δ matrix has a very simple form of

$$\Delta = \begin{bmatrix} 0 & 0 & 0 & \Delta_{n,o} & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta_{e,o} & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta_{s,o} & 0 & 0 & 0 \\ \Delta_{o,n} & \Delta_{o,e} & \Delta_{o,s} & 0 & \Delta_{o,w} & \Delta_{o,t} & \Delta_{o,b} \\ 0 & 0 & 0 & \Delta_{w,o} & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta_{t,o} & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta_{b,o} & 0 & 0 & 0 \end{bmatrix}. \tag{6.15}$$

Therefore, to evaluate $d(z)$, only the following 7×7 Green's functions have to be calculated

$$G = \begin{bmatrix} G_{n,n} & G_{n,e} & G_{n,s} & G_{n,o} & G_{n,w} & G_{n,t} & G_{n,b} \\ G_{e,n} & G_{e,e} & G_{e,s} & G_{e,o} & G_{e,w} & G_{e,t} & G_{e,b} \\ G_{s,n} & G_{s,e} & G_{s,s} & G_{s,o} & G_{s,w} & G_{s,t} & G_{s,b} \\ G_{o,n} & G_{o,e} & G_{o,s} & G_{o,o} & G_{o,w} & G_{o,t} & G_{o,b} \\ G_{w,n} & G_{w,e} & G_{w,s} & G_{w,o} & G_{w,w} & G_{w,t} & G_{w,b} \\ G_{t,n} & G_{t,e} & G_{t,s} & G_{t,o} & G_{t,w} & G_{t,t} & G_{t,b} \\ G_{b,n} & G_{b,e} & G_{b,s} & G_{b,o} & G_{b,w} & G_{b,t} & G_{b,b} \end{bmatrix}. \tag{6.16}$$

The computation can be further simplified by expanding $d(z)$ as the following:

$$\begin{aligned} d(z) &= \det(\mathbb{1} + \mathrm{G(z)}\Delta) & (6.17) \\ &= [1 + \sum_{j \in NN} \Delta_{jo} G_{oj}(z)][1 + \sum_{j \in NN} \Delta_{oj} G_{jo}(z)] & (6.18) \\ &- G_{oo}[\sum_{j,k \in NN} \Delta_{jo} \Delta_{ok} G_{kj}(z)], & (6.19) \end{aligned}$$

where

$$\Delta_{jo} = \langle j|\Delta|o\rangle, G_{oj} = \langle o|G|j\rangle. \tag{6.20}$$

Moreover, using the following state $|v\rangle$:

$$|v\rangle = \Delta|o\rangle = \sum_{j\in NN} \Delta_{jo}|j\rangle, \tag{6.21}$$

$d(z)$ can be compactly expressed as

$$d(z) = [1 + G_{ov}(z)][1 + G_{vo}(z)] - G_{oo}(z)G_{vv}(z). \tag{6.22}$$

Notice that we now need only a 2×2 Green's function to evaluate $d(z)$.

Now, a question is how to calculate efficiently the local Green's functions $G(z)$. For this purpose, we use the KPM [17], which can be efficiently implemented in a GPU cluster [89]. Using two types of Chebyshev polynomials (m: integer),

$$\begin{cases} T_m(x) = \cos\left[m \arccos(x)\right] \\ U_m(x) = \dfrac{\sin\left[(m+1)\arccos(x)\right]}{\sin\left[\arccos(x)\right]}, \end{cases} \tag{6.23}$$

the diagonal elements of the Green's function are expanded as

$$\begin{aligned} G_{ii}(\tilde{w} + i\epsilon) &= \frac{i}{\sqrt{1-\tilde{w}^2}}\left[\tilde{\mu}_0 + 2\sum_{m=1}^{M-1}\tilde{\mu}_m \mathrm{T_m}(\tilde{w})\right] + 2\sum_{m=1}^{M-1}\tilde{\mu}_m \mathrm{U_m}(\tilde{w}) \\ &= \frac{i}{\sqrt{1-\tilde{w}^2}}\left[\tilde{\mu}_0 + 2\sum_{m=1}^{M-1}\tilde{\mu}_m \mathrm{exp}\left[-im\mathrm{arccos}(\tilde{w})\right]\right]. \end{aligned} \tag{6.24}$$

Since the Chebyshev polynomials $T_m(x)$ and $U_m(x)$ requires that the argument x should be within $[-1, 1]$, we must renormalize the energy spectrum E to $\tilde{w}$. The $\tilde{\mu}_m$ represents the m-th moment (defined below) after applying a kernel function, $\tilde{\mu}_m = \mu_m g_m$, where g_m is the kernel function to eliminate Gibbs oscillations [17]. Here, we apply Lorenz kernel function which is defined as $g_m = \sinh[\lambda(1 - m/M)]/\sinh(\lambda)$

with appropriate choice of λ [17]. The m-th moment μ_m is defined as

$$\begin{aligned}
\mu_m &= \frac{1}{\pi}\lim_{\epsilon\to 0}\mathrm{Im}\int G_{ii}(\tilde{w}+i\epsilon)T_m(\tilde{w})d\tilde{w} \\
&= \int\sum_n \langle i|n\rangle\langle n|i\rangle\delta(\tilde{w}-\tilde{\epsilon}_n)T_m(\tilde{w})d\tilde{w} \\
&= \sum_n \langle i|n\rangle\langle n|i\rangle T_m(\tilde{\epsilon}_n)dE \\
&= \sum_n \langle i|T_m(\tilde{H})|n\rangle\langle n|i\rangle \\
&= \langle i|T_m(\tilde{H})|i\rangle, \qquad (6.25)
\end{aligned}$$

where $\tilde{H}$ is the renormalized Hamiltonian to fit the spectra within $[-1,1]$.

Defining $|a_m\rangle = T_m(\tilde{H})|i\rangle$ and thus $\mu_m = \langle i|a_m\rangle$, the Chebyshev series in Eq. (6.23) yeilds the recursive relation of these vectors $|a_m\rangle$, namely,

$$\begin{cases} |a_0\rangle = |i\rangle \\ |a_1\rangle = \tilde{H}|a_0\rangle \\ |a_m\rangle = 2\tilde{H}|a_{m-1}\rangle - |a_{m-2}\rangle \end{cases}. \qquad (6.26)$$

In addition, the number M of coefficients is obtained with $M/2$ iterations if the following equation is used,

$$T_{2m-i} = 2T_{m-i}T_m - T_i,\ \ i = 0,1 \qquad (6.27)$$

$$\mu_{2m-i} = 2\langle r_{m-i}|r_m\rangle - \mu_i. \qquad (6.28)$$

When the coefficients μ_m are all calculated, we can evaluate the Green's function using fast Fourier transform [17]. In Eq. (6.24), if we choose

$$\tilde{w} = \cos\frac{\pi(k+\frac{1}{2})}{M},\ \ (k = 0,1,...,M-1)$$

the Green's function becomes

$$G_{ii}(\tilde{w}+i\epsilon) = \frac{2i}{\sqrt{1-\tilde{w}^2}} \sum_{m=0}^{M-1} \mu'_m \exp\left[\frac{-im\pi(k+\frac{1}{2})}{M}\right], \tag{6.29}$$

where

$$\mu'_m = \begin{cases} \tilde{\mu}_0/2, \ m=0 \\ \tilde{\mu}_m, \ m>0 \end{cases}. \tag{6.30}$$

Let us now denote the summation part in Eq. (6.29) by χ_k:

$$\chi_k = \sum_{m=0}^{M-1} \mu'_m \exp\left[\frac{-im\pi(k+\frac{1}{2})}{M}\right].$$

It should be recalled that the following expression is required for the fast Fourier transformation [104]:

$$\gamma_n = \sum_{m=0}^{M-1} c_m \exp\left[\frac{-i2m\pi n}{M}\right], (m=0,1,\dots,M-1) \tag{6.31}$$

where

$$c_m = \mu'_m \exp\left[\frac{-im\pi}{2M}\right] \tag{6.32}$$

Using the following correspondence between χ_j and γ_j

$$\begin{cases} \chi_{2j} = \gamma_j, \\ \chi_{2j+1} = \gamma^*_{M-j-1} \end{cases} \quad j=0,1,...,M/2-1,$$

χ_j can be evaluated using the fast Fourier transformation, and thus the time complexity of calculating Eq. (6.29) reduces to O($M\log(M)$), where M is the number of moments kept.

The off diagonal elements of the Green's function, G_{ov} and G_{vo}, can be evaluated similarly if we use the follow mixed elements of the Green's function (note that i

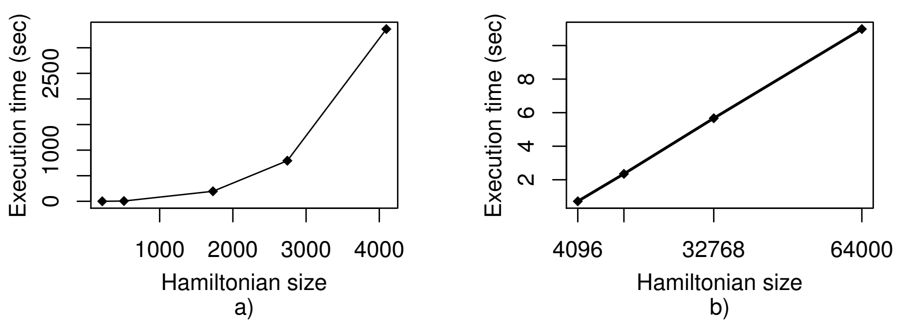


Figure 6.1: Execution time for (a) the exact diagonalization method [100–102] and (b)the GFMC method. Here, for simplicity, we performed only 10 Monte Carlo trial flips of spins, and M is fixed for different Hamiltonian sizes. We can clearly see that the time consumption for the exact diagnolization method roughly follow the complexity of $O(N^3)$, but for the GFMC method it is almost linear.

below is the imaginary unit):

$$G_{o+v,o+v} = (\langle v| + \langle o|)G(|v\rangle + |o\rangle) = G_{oo} + G_{vv} + G_{ov} + G_{vo} \tag{6.33}$$

$$G_{o+iv,o+iv} = (-i\langle v| + \langle o|)G(|o\rangle + i|v\rangle) = G_{oo} + G_{vv} + iG_{ov} - iG_{vo}, \tag{6.34}$$

G_{ov} and G_{vo} are now expressed by the diagonal elements of the Green's functions:

$$G_{ov} = \frac{1}{2}[G_{o+v,o+v} - G_{oo} - G_{vv} + i(G_{oo} + G_{vv} - G_{o+iv,o+iv})], \tag{6.35}$$

$$G_{vo} = \frac{1}{2}[G_{o+v,o+v} - G_{oo} - G_{vv} + i(G_{o+iv,o+iv} - G_{oo} - G_{vv})]. \tag{6.36}$$

Using these four elements of the 2×2 Green's function, we can readily calculate $d(z)$ and thus Δ_{seff} can be evaluated.

Finally, it should be noted that if Hamiltonian matrix H is stored in a compression format, the time complexity of Equation (6.26) is $O(N)$, where N is the dimension of H. As mentioned above, the time complexity to calculate Equation (6.24) is $O(M\log(M))$, and thus the total time complexity is expected to be $O(N) + O(M\log(M))$. When M is fixed, the time complexity should scale linearly with the dimension of H. Indeed, we find, as shown in Figure 6.1, that the execution time is approximately proportional to the Hamiltonian size N.

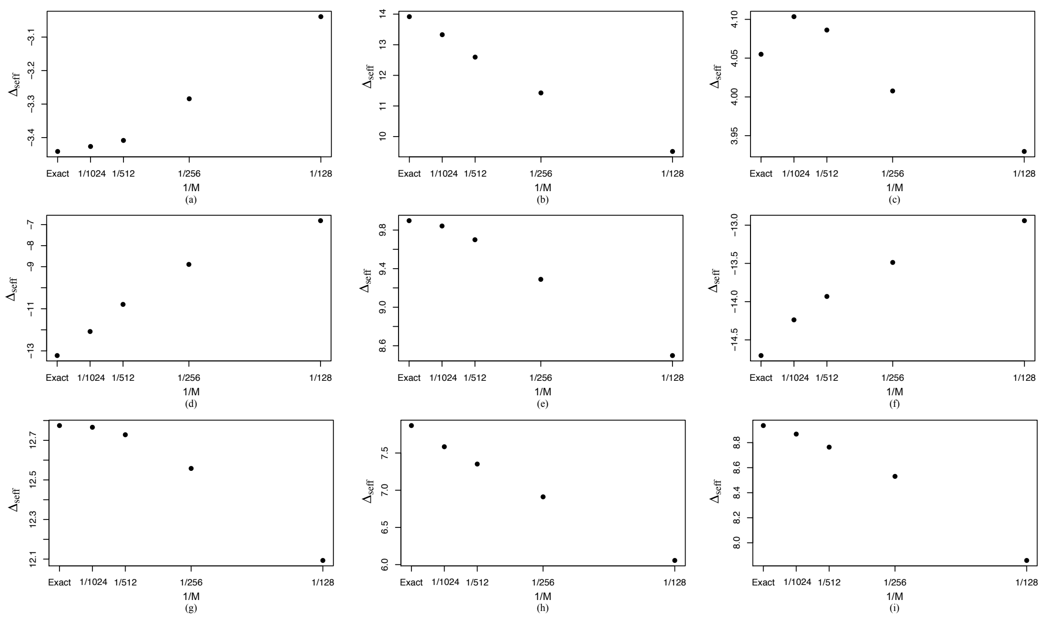


Figure 6.2: Linearity of Δ_{seff} in terms of $1/M$, where M is Chebshev expansion order

Truncated-free formulation of Δ_{seff}

Since the Green's function $G_{oo}, G_{vv}, G_{o+v}, G_{o+iv}$ is obtained by KPM which inevitably involves truncated Chebyshev expansion with a finite order M, errors are naturally exists in Δ_{seff}. Indeed, one can use a larger M to achieve high accuracy, but the computational amount will also increase sharply as a result.

In order to improve the computational accuracy without a significant increase of the computational amount, one can use 2-point fit method [19] to extrapolate a truncated-free result if the following relation could be noticed

$$\Delta_{\text{seff}} \propto \frac{1}{M} \tag{6.37}$$

The truncated-free result, i.e. the result obtained when $M \to \infty$, which implies

$$\frac{1}{M} \to 0 \tag{6.38}$$

If we know the value of $\Delta_{\text{seff}}^{(M)}$ and $\Delta_{\text{seff}}^{(2M)}$, which correspond to the expansion order M and 2M respectively, the exact value can be expressed

$$\Delta_{\text{seff}}^{(\infty)} = 2\Delta_{\text{seff}}^{(2M)} - \Delta_{\text{seff}}^{(M)} \tag{6.39}$$

To verify this relation, several Δ_{seff} of low temperature (T=0.01) are evaluated for expansion order 128, 256, 512 and 1024, respectively.

Figure 6.2 plot these values of Δ_{seff}. The exact values of Δ_{seff} given by full diagnolization (using Equation 6.7) are also shown in each figure. It can be learned that the linearity(Equation 6.37) is valid in most cases. However, it also can be seen that sometimes this method may fail to produce a result closer to the exact value, as shown in Figure 6.2 (c), probably due to the errors introduced by the numerical integral of Δ_{seff} (Equation 6.14) and Green's function (Equation 6.31).

6.2 Implementation and Parallelization Schemes

6.2.1 Algorithm Design

For a given temperature T, we use Algorithm 4 to calculate the magnetization M for the classical spins through the average over S Monte Carlo sweeps (the loop from line 1), where each sweep corresponds to N spin trail flips (the loop from line 2). Since the direction of the magnetization is trivial, the magnetization M is defined here as the length of the total spin vector.

In the implementation, we apply the Metropolis method (line 10 in the Algorithm 4) to determine whether a trail flip is accepted by comparing with a random number between 0 and 1.

Section 6.1 has illustrated how to calculate the Δ_{seff} (from line 4 to 9) using the GFMC method. Especially, the calculation of the expansion coefficients μ_m plays a curial role (line 6). This part occupies most of the execution time since the recursion [denoted by Eq. (6.26)] involves intensive matrix-vector multiplication (SpMV) with complexity of $O(N)$. Note, however, that GPU is an ideal platform to parallelize SpMV, because the multiplications between the rows and the vector could be distributed to hundreds of streaming processors. Therefore, we focus on a GPU implementation with the highly parallelism and expect a large speedup factor as compared with the CPU one.

Algorithm 4 Calculate the magnetization as a function of temperature

```
Require: Integer $S$ to represent number of Monte Carlo sampling sweeps
Require: Hamiltonian matrix $H$ of dimension $N \times N$
Require: Scalar $M$ to store accumulated magnetization
 1: for $i = 1 \rightarrow S$ do
 2:   for $j = 1 \rightarrow N$ do
 3:     Randomly choose site $o$ and change the spin randomly
 4:     Calculate the modification matrix $\Delta \leftarrow H - H'$ using Eq. (6.3)
 5:     Get vector $\overrightarrow{v} \leftarrow \Delta \overrightarrow{o}$
 6:     Calculate the coefficients $\tilde{\mu}_m^{(o)}, \tilde{\mu}_m^{(v)}, \tilde{\mu}_m^{(o+iv)}, \tilde{\mu}_m^{(o-iv)}$ applied Lorentz kernel
        function $g_m$, where $\tilde{\mu}_m^{(V)} = \langle V|T_m(\tilde{H})|V\rangle g_m$
 7:     Calculate 4 elements of the Green's function $G_{oo}, G_{vv}, G_{o+iv,o+iv}$ and
        $G_{o-iv,o-iv}$ using Eq. (6.24)
 8:     Calculate $d(z)$ using Eq. (6.22)
 9:     Calculate $\Delta_{\mathrm{seff}}$ using Eq. (6.14)
10:     if $e^{-\Delta_{\mathrm{seff}}} > \mathrm{rand}()$ then
11:       Accept the new spin configuration.
12:       Update $H$ : $H \leftarrow H + \Delta$
13:     end if
14:   end for
15:   $M = M + \frac{|\sum_i^N \vec{S}_i|}{N}$
16: end for
17: $\mathrm{Magnetization} \leftarrow M/S$
```

6.2.2 Parallelization Methods

The magnetization M as a function of temperature T is obtained through the average over a large number of Monte Carlo sweeps. If the system stays in the thermal equilibrium, the Monte Carlo sweeps are independent with each other. Therefore, we could divide the outside loop S (line 1) into many threads. However, the warm up sweeps, which are necessary to achieve the thermal equilibrium, should be abandoned for taking the average, and this condition may prevent us from dividing S to too many threads because some threads may have too few sweeps to reach thermal equilibrium.

In addition to this parallelism for the Monte Carlo sampling, the intensive matrix-vector and vector-vector multiplications in line 6 can be effectively parallelized into multi-core CPU processors using OpenMP or many-core GPU processors using CUDA. In this paper, we focus on the implementation based on GPU to archive higher parallelism than multi-core CPU.

In our algorithm, we combine two parallelizations to achieve high efficiency, i.e., the Monte Carlo sweeps are divided into several MPI threads while in each thread we employ GPU to calculate matrix-vector operations.

6.2.3 Implementation on GPU

Compression Format of Hamiltonian Matrix

Here ELL format is applied to Hamiltonian matrix, In order to trigger coalesced memory access on GPU device, the ELL matrix for storing the non-zero values and column indices are both stored in column major. Because the periodical boundary condition, which leads to a constant number of non-zeros of each row, is applied, there is no extra padding data for ELL.

Arrangement of CUDA Kernel Functions

One important issue that should be well addressed is the communication between GPU and CPU. Since the Hamiltonian matrix and the spin configurations may be updated after each iteration, if the Hamiltonian matrix are stored in CPU memory, the data transfer between CPU and GPU may significantly decrease the performance. This is especially true when the lattice size is small because the execution time for numerical calculation occupies a low percentage of the overall time consumption. According to our test, the simulation of $6 \times 6 \times 6$ cubic lattice on a 8x PCIe Tesla C2050 is about 30% slower than on a 16x PCIe C2050. Therefore, the data transfer may not only decrease the performance, but also make the program to be more dependent on the bandwidth between the CPU and GPU.

```
for $i = 1 \to S$
  for $j = 1 \to N$
    /* choose a spin randomly and flip a spin with a
    random direction and calculate matrix $\Delta$ */
    FlipSpinAndCalculateDelta <<<>>> ();

    // prepare vectors $|o\rangle, |v\rangle, |o\rangle + i|v\rangle, |o\rangle - i|v\rangle$
    PrepareVector <<<>>> ();

    /* calculate the expansion coefficients for each Green's
    function */
    for each vector in { $|o\rangle, |v\rangle, |o\rangle + i|v\rangle, |o\rangle - i|v\rangle$ }
      for m=0 to M/2
        // cacluate 2 coefficients for each iteration
        CalculateCoefficient <<<>>> ();
      end for
    end for

    // transform Eq. (6.24) to meet the requirement of FFT
    PrepareForFFT<<<>>> ();

    // perform FFT to obtain 4 Green's functions
    PerformsFFT<<<>>> ();

    // Calculate $\Delta_{\mathrm{seff}}$
    CalculateDSEFF<<<>>> ();

    // If a trial spin flip is accepted, update the status
    UpdateStatus<<<>>> ();

    // Calculate the magnetization
    MeasureSpin<<<>>> ();
  end for
end for
```

Figure 6.3: Pseudocode of GPU implementation for GFMC

To avoid the memory transfer between CPU and GPU as much as possible, in our implementation the matrix H and the spin configurations are kept in GPU memory.

With these considerations, we propose a multi-kernel GPU implemenation shown in the Figure 6.3, in which the kernel functions are indicated by "<<<>>>" and arranged as the following:

Function "FlipSpinAndCalculateDelta" flips a spin with a random direction and calculates the Hamiltonian matrix difference Δ, which is used in the function "PrepareVector" to create four vectors $\{|o\rangle, |v\rangle, |o\rangle + i|v\rangle, |o\rangle - i|v\rangle\}$. For each vector, the function "CalculateCoefficient" calculates two expansion coefficients μ_m in every iteration. After all the coefficients are ready, function "PrepareForFFT" prepares the data for fast Fourier transformation (FFT), including applying Lorentz kernel function and transform Eq. (6.24) to meet the requirement of FFT. After performing FFT by function "PerformsFFT", we obtain Δ_{seff} using function "CalculateDSEFF". If the flip is to be accepted, function "UpdateStatus" will update the spin configuration and the Hamiltonian matrix. After one MC sweep finishes, function "MeasureSpin" accumulates the spins to calculate the magnetization.

Implementation of the Recursion

The most important kernel function is "CalculateCoefficient", which involves the most intensive computation due to the recursive matrix-vector operations. Figure 6.4 demonstrates the implementation of this CUDA kernel function to calculate the coefficients μ_{2m}. In CUDA architecture, the parallel running threads are divided into several thread groups, called "Blocks". For example, in Figure 6.4 there are p thread blocks and each block has 4 threads, indexed from 0 to 3. Each thread performs the multiplication between a row and the vector $|a_{m-1}\rangle$ that is stored in a 1D array R1. Since the produced vector $|a_m\rangle$ could be placed on R0 overwriting the vector $|a_{m-2}\rangle$, we only need two 1D memory arrays, R0 and R1, to perform calculations by exchanging their pointers after each iteration.

After the vector $|a_m\rangle$ is ready, a production $\langle a_m|a_m\rangle$ is performed, shared memory is applied to store the product result. $\langle a_m|a_m\rangle$ is a dot product, for CUDA, it is a

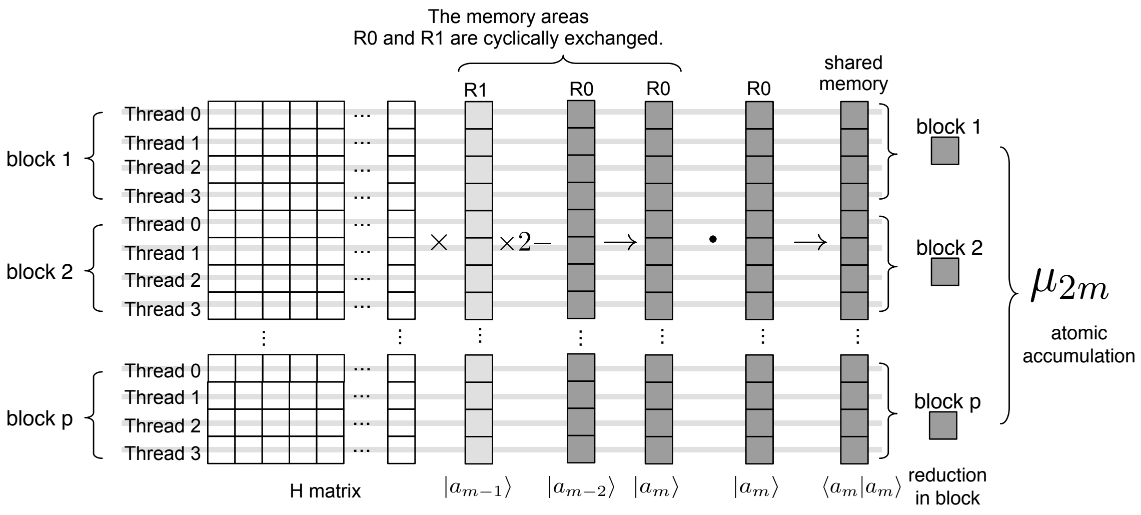


Figure 6.4: Implementation of function "CalculateCoefficient" to calculate the moments μ_{2m}

reduction problem, which will be carried out in two steps, first, the product result in each block is accumulated to produce a scalar and then the scalar in each block is further accumulated to get coefficient μ_{2m} using Eq. 6.28.

This implementation could eliminate the most data transfers during the calculation. The bandwidth test shows the performance difference between a 8x and 16x PCIe based Tesla C2050 is very small (less than 6%) while calculating a 6^3 cubic lattice.

6.3 Performance Evaluation

We evaluate the performance in two ways. The first is to measure the performance of a single CPU core and a single GPU with different Hamiltonian matrix sizes. The second is to evaluate the overall performance comparison between all CPU cores and all GPUs.

6.3.1 Performance Scaling on Multi-core CPU

Section 6.2.1 has explained that the process of calculating coefficients $\tilde{\mu}$ involves intensive SpMV operations. The performance of SpMV largely depends on the memory access speed. Therefore, the memory bandwidth plays an essential role in the over-

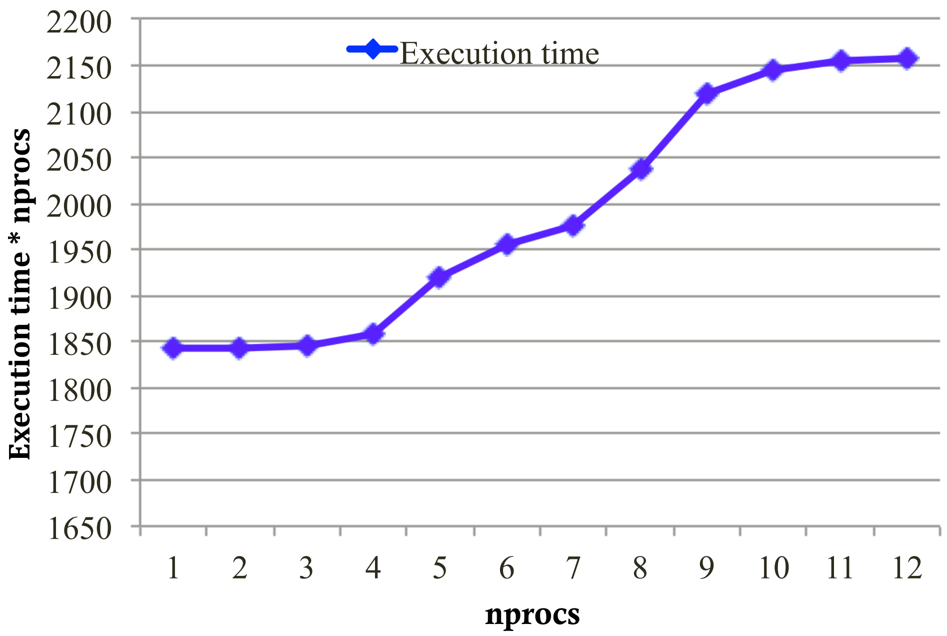


Figure 6.5: The performance scaling on multiple CPU cores within a node

all performance. This conclusion is consistent with our multi-core CPU performance scaling test shown in Figure 6.5.

This test is performed on a single node that has 2 CPUs, totally 12 cores and 12GB DDR2 800 memory with a peak bandwidth of 12.5GB/s. In order to evaluate the parallelization efficiency, we examined the quantity expressed by $y = execution\ time \times nprocs$, in which $nprocs$ represents the number of MPI threads. As the outside loop [from line 1] in Algorithm 4 is divided into multiple MPI threads and there is very little communications among these threads, in an ideal case of parallelization, y should be a constant. However, we have noticed that when the threads number becomes larger than 4, y begins to increase fast, suggesting that the efficiency become to decrease rapidly.

It indicates that the memory bandwidth becomes saturated when $nprocs$ grows to be larger than 4. Therefore, we can conclude that the bottleneck of this method is memory bandwidth. In this paper we resort to solve the problem using Tesla C2050 GPU with maximum bandwidth of 144GB/s [41].

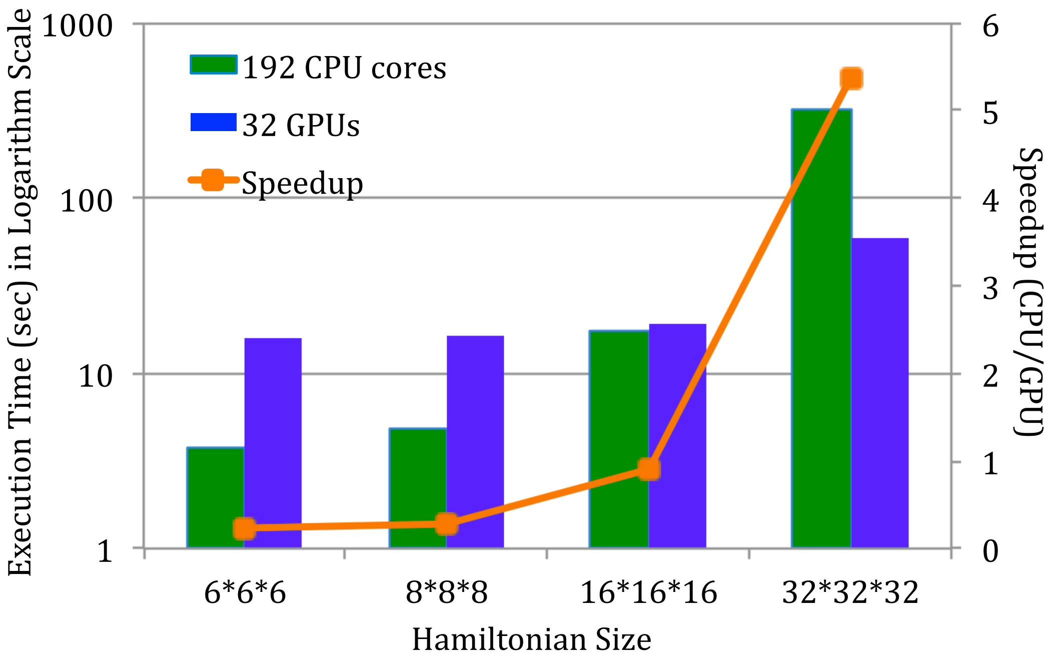


Figure 6.6: Performance scaling with Hamiltonian matrix size on cluster, the result is obtained with 38400 trail flips calculated and Chebyshev truncation number kept as 256

6.3.2 Performance Scaling for Increasing Hamiltonian Size

Figure 6.6 shows the performance scaling on the cluster while the Hamiltonian size is increasing from 6^3 to 32^3. 38400 Monte Carlo trail flips are calculated while the number of Chebyshev moments are kept as 256. It can be observed that when the H matrix is small, e.g. 6^3 or 8^3, 192 CPU cores are much faster than 32 GPUs. To explain the reason, let us divide the total time consumption of GPU program into two parts: a) the time for numerical calculation and b) the time for other works such as initializing GPU device, copying memory between CPU and GPU, and MPI communication, etc. Usually the latter part consumes very little time, but when the matrix size is small, comparing with the first part, time consumption of the latter part can not be neglected and significantly decrease the performance. However, the speedup increases rapidly with increasing H matrix size and in the best case, i.e. 32^3, 32GPU is about over than 5 times faster than 192 CPU cores. If we define the speedup factor as the equivalent number of CPU cores comparing with one GPU. For Hamiltonian with size of 32^3, this factor is about 30 (192/32 * 5).

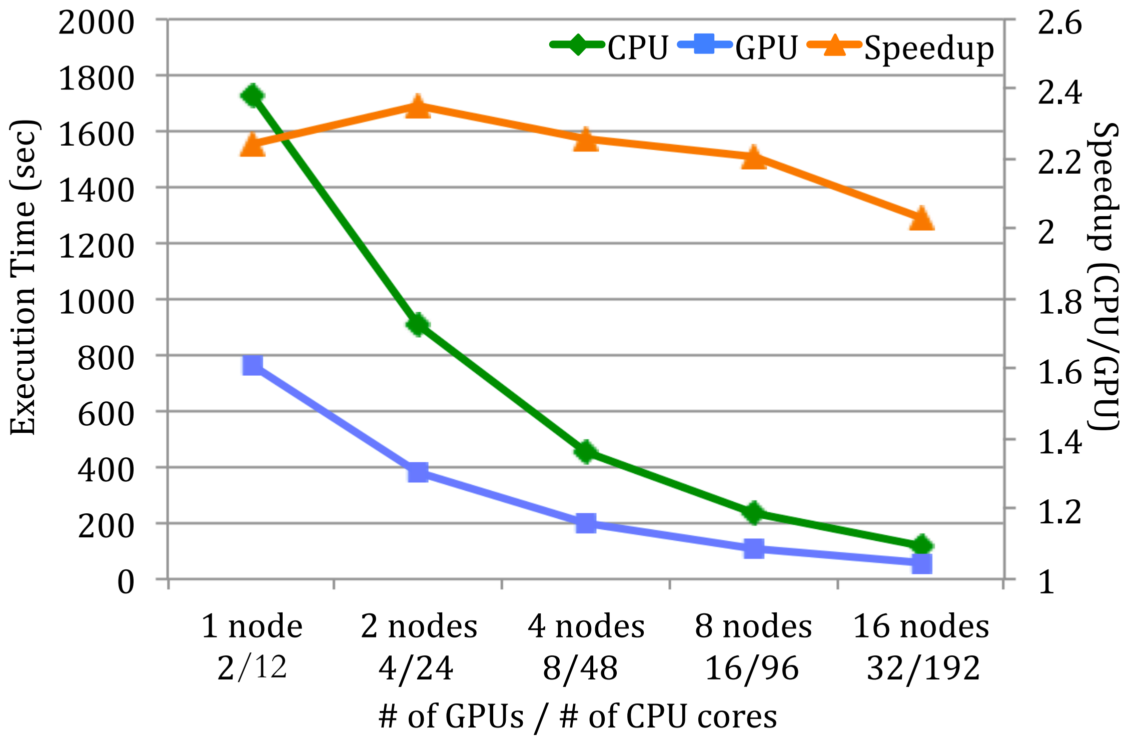


Figure 6.7: Performance scaling for increasing number of nodes, each node has 2 Tesla C2050 and 12 Xeon 2.4GHz cores. 8000 Monte Carlo trail flips of a 20^3 cubic lattice Hamiltonian are calculated with the Chebyshev truncation number kept as 256. The blue/green lines represent time consumption of CPU cores/GPUs

6.3.3 Performance Scaling for Increasing Number of Nodes

The performance scaling for increasing number of nodes is also very important reference for implementing the algorithm on a large cluster or even supercomputer. Here 8000 trail flips of a 20^3 cubic lattice Hamiltonian is calculated on our cluster that has 2 Tesla C2050 and 12 Xeon 2.4GHz cores in each node. The performance evaluation is shown in Figure 6.7. As there is very little communication among MPI threads, it can be noticed that for both CPU and GPU implementation, the time consumption always decreases by half when the number of nodes is doubled , suggesting an ideal performance scaling with number of nodes.

6.3.4 Performance Considerations

As shown in Figure 6.5, the bottleneck of this algorithm is the memory bandwidth, GPU could solve the memory bandwidth limitation very well and shows high speedup ratio. Due to the nature that there is little communication among MPI threads, this algorithm is also capable to run on many compute nodes while keeping good parallelization efficiency.

However, due to the thermal equilibrium problem explained in Section 6.2.2, the of number of MPI threads is limited, in this case, a combination of parallelization techniques such as MPI, OpenMP and CUDA should be introduced. For example, the workload is firstly distributed into different nodes using MPI, and then in each node we use OpenMP to calculate the SpMV, if the CUDA is capable, the OpenMP thread may employ GPU to calculate the SpMV. The combination can increase the parallelism and gain high performance. However, it should be pointed out that the process to combine these techniques may not be very straightforward since we have to take trade-offs in many occasions.

6.4 Discussion and Summary

In this chapter, we have provided the detailed formulation of the GFMC method for the DE model. Based on the algorithm analysis, the implementation is proposed to parallelize the GFMC method using MPI on CPU as well as CUDA on GPU. In order to eliminate the data transfer between CPU and GPU as much as possible, the program is implemented on GPU using several kernel functions. The performance evaluation indicates that the GPU implementation could effectively overcome the CPU memory bandwidth limitation and shows more than 30 speedup factor when Hamiltonian size is 32^3. The performance scaling test for increasing number of processors indicates that the MPI parallelization is very effective for this algorithm. Finally we discussed more considerations on the performance for future optimization.

Chapter 7

Library Implementation: An Introduction

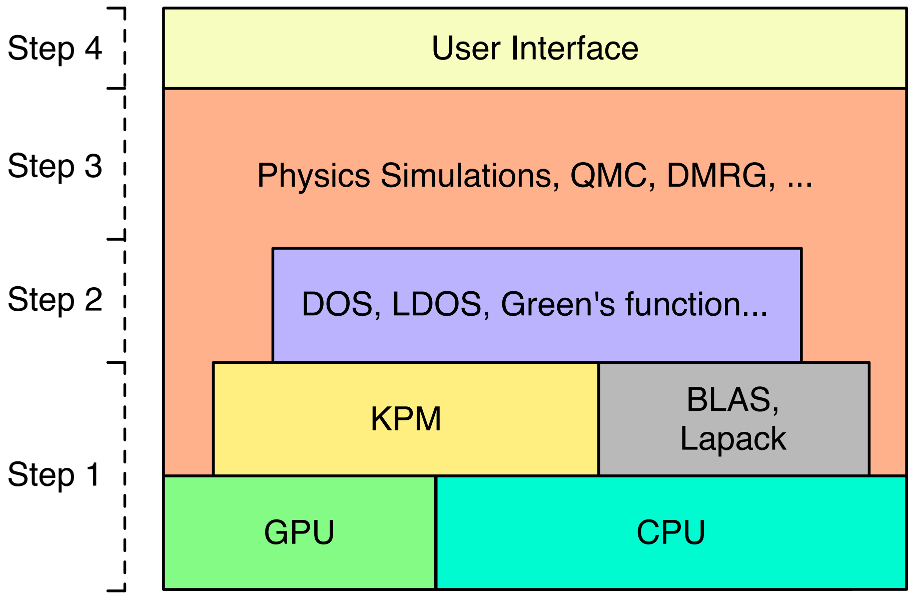
In order to ease the difficulty of utilizing the GPU in material simulations, implementing a KPM-based library for various applications with GPU acceleration capable is an important part of this work. Although currently this work is not finished yet, in this chapter we give a brief introduction to the libraries' architecture, programming interface as well as the extension targeting to address the future's needs.

7.1 Library Structure

Before introduction to the architecture, let us see the basic requirements from physics users' perspectives. The following features are considered necessary for the KPM library

- As the CPU is still the main choice for most scientific applications, in addition to the GPU-based implementation, this library should contain the CPU implementation as well.
- To meet the requirement of other numerical method, e.g. exact diagonalization that can produce high precision, a mechanism of switching between KPM and other third party libraries(e.g. Lapack) should be supported.

- As KPM is a common numerical expansion technique, it suppose to be applied to evaluate various functions other than DOS, LDOS or Green's function in this study.
- User should be able to switch operations between single precision and double precision.



Step 4
Step 3
Step 2
Step 1
User Interface
Physics Simulations, QMC, DMRG, ...
DOS, LDOS, Green's function...
KPM
BLAS, Lapack
GPU
CPU

Figure 7.1: Architecture of KPM Library and Road Map

Based on the requirements, Figure 7.1 displays the brief structure of the library. Two versions of KPM is established targeting for CPU and GPU, receptively. Some fundamental and well used functions, e.g. DOS, LDOS, Green's function, are implemented not only using KPM as well as using other third party libraries, for example, the full diagnoalization method in Lapack. These fundamental functions could be directly invoked in the simulations of QMC, DMRG, etc. In addition, the simulation is able to call the KPM and other third-party libraries directly. On the very top of the architecture an user interface is provided and can be used directly to perform simulations with few input parameters.

As shown in Figure 7.1, the project roughly consists of 4 phases. At the very beginning, we have implemented the KPM and used it to evaluate several simple functions to examine its characteristics. In the second phase, several APIs has been proposed to evaluate the DOS and LDOS function. Especially, motivated by the study of using LDOS as criteria of Anderson Localization [43], we implement the GPU based program to evaluate LDOS functions using KPM as shown in chapter 5, Appendix B visualize the LDOS functions in three dimension for a cubic lattice. At the moment this project goes into the third phase, where we make more complete simulation applications, e.g. quantum monte carlo simulation for DE model as shown in Chapter 6. Finally, a friendly user interface should be established.

7.2 Implementation Techniques

Currently, C++ is used for the CPU code and CUDA C for the NVIDIA's GPU. In order to provide a common KPM kernel not only for the fundamental functions(e.g. DOS, LDOS), C++ template is applied to provide uniform APIs.

In addition, for a given hardware platform, calculation based on double floating point operation is usually slower than operations based on single precision, however, double precision is necessary in many occasions that requires high accuracy. Thus it is important to provide mechanism to enable the users to switch between double or single floating point. In the KPM library, this feature is provided through C++ template. For example, the following API is used to create a DOS function from given expansion coefficients

```
template<typename T>
T* create(T* expan_coef, unsigned int num, T* kernel, HamiltonianInfo* hi);
```

we can switch to the single or double precision through

```
create<float>(...); // using single precision
create<double>(...); // using double precision
```

7.3 Examples of Using APIs and User Interface

Here let us take a look at two examples of how to use APIs and the user interface. The following code is an example of evaluation of DOS for a given Hamiltonian, here the Hamiltonian matrix of Anderson disorder model is used.

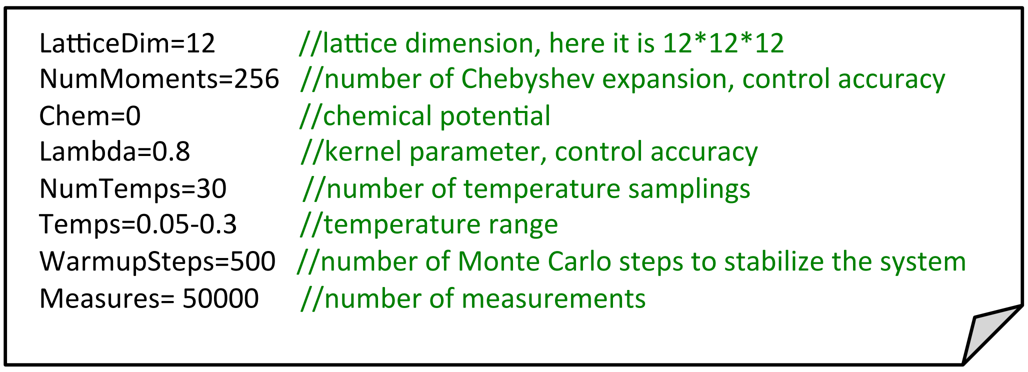
```
//create a plan for DOS, the "plan" should contain the information including expansion
order, number of random vectors, Hamiltonian matrix, etc.
DOSPlan<T> plan;

//create a HamiltonianInfo structure to store such as number lattice sites, disorder
value, etc.
HamiltonianInfo hi;

//initialize object "hi" with a 8*8*8 cubic lattice and disorder value 10.0
hi.InitialConfig(8, 8, 8, 10.0);

//initial a random vector generator
MersenneTwist mt;
mt.init_genrand(2011);

//specify number of random vectors, corresponding to RS in Algorithm 2
plan.num_r = 1792;

//Chebyshev expansion order, corresponding to N in Algorithm 2.
plan.m = 2048;

//allocate memory for storing the coefficients μ_i
plan.U = (T*)malloc(sizeof(T) * plan.m);

//create Hamiltonian matrix of a cubic lattice, the matrix is stored in ELL format.
plan.H = Hamiltonian::createCubeSingleElectron_ELL<T>(hi, mt);

//invoke this function to evaluate the expansion coefficients.
solveDOSPlan_gpu<T>(&plan); // using GPU
//solveDOSPlan_cpu<T>(&plan); //using CPU

//reconstruct DOS function
KPMDOS* dos = KPMDOS::create<T>(plan.U, plan.m, 1000, &hi);

//write result to files.
dos->writeToFile("result_dos.dat");
```

The user just need a few steps to set up a program for calculating the DOS function, as shown in the following chart.

$$\boxed{\text{Create A DOS Plan}} \rightarrow \boxed{\text{Set Parameters}} \rightarrow \boxed{\text{Create Hamiltonian}}$$
$$\rightarrow \boxed{\text{Perform Calcualtion}} \rightarrow \boxed{\text{Output Result}} \tag{7.1}$$

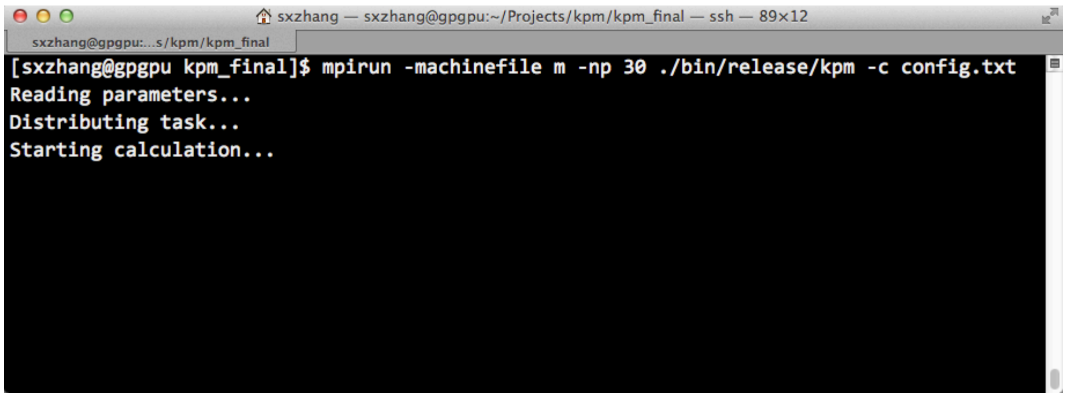
In addition to providing API functions, for QMC simulation of DE model, a simple interface has been established to ease the programming difficulty. As shown in Figure 7.2 (a), several parameters are set in a configuration file (config.txt) that is basically a plain text file. In this example, the lattice size is set to $12 \times 12 \times 12$, the expansion order is set to 256 and chemical potential is set to 0. Totally 30 temperature samples ranging from 0.05 to 0.3 are going to simulated. After 500 warm up steps where each step corresponds to 12^3 spin trail flips, 50000 measures is taken for every 12^3 trail flips.

In Figure 7.2 (b), we simply use MPI to execute the program on 30 threads, the program will distribute the work into 30 threads. As there are 30 temperature samples in this example, every thread therefore just simulate 1 temperature. The simulation result is shown in Appendix C.



```
LatticeDim=12        //lattice dimension, here it is 12*12*12
NumMoments=256       //number of Chebyshev expansion, control accuracy
Chem=0               //chemical potential
Lambda=0.8           //kernel parameter, control accuracy
NumTemps=30          //number of temperature samplings
Temps=0.05-0.3       //temperature range
WarmupSteps=500      //number of Monte Carlo steps to stabilize the system
Measures= 50000      //number of measurements
```

(a)



sxzhang — sxzhang@gpgpu:~/Projects/kpm/kpm_final — ssh — 89×12
sxzhang@gpgpu:...s/kpm/kpm_final

```
[sxzhang@gpgpu kpm_final]$ mpirun -machinefile m -np 30 ./bin/release/kpm -c config.txt
Reading parameters...
Distributing task...
Starting calculation...
```

(b)

Figure 7.2: Implementing GFMC as an executable

Chapter 8

Conclusions and Future Works

As a competitive numerical methods for solving Hamiltonians in various physics models, a high performance implementation of KPM is very necessary. However, the fine-grain and memory bounded recursion in KPM brings great difficulty for parallelization of KPM algorithm using lager number of threads on a supercomputer or cluster via MPI.

In this study, based on GPU platform, several KPM implementations are proposed and demonstrated to evaluate some fundamental quantum quantities in condensed matter physics such as DOS and LDOS. We also propose optimization techniques to enhance the performance. Through the study of the memory accessing characteristics of KPM, we mainly focus on the optimization of KPM's memory accessing.

The final performance evaluation for all the three applications indicates that, with high memory bandwidth, GPU could well handle the fine-grain parallelism to achieve higher performance than CPU. As for DOS application, one GPU achieves over 12x higher performance than a CPU core. In LDOS application, combining with MPI, GPU shows much better performance scaling, it can be learned in cluster environment, one GPU is equivalent of 24 Xeon E5645 CPU cores. In the third application, we demonstrates the quantum Monte Carlo simulation using KPM to evaluate Green's function, the performance evaluation indications that one Tesla C2050 is faster than 30 Xeon E5645 CPU cores. The material simulation is dramatically accelerated on the GPU cluster.

In order to ease the difficulty of making simulation programs on GPU, establishment of a KPM library is very necessary and helpful. An architecture of KPM library is proposed from software engineering perspective, through two examples, the using of APIs and executable are demonstrated.

However, the establishment of the library is not finished so far, currently we are considering implementing a framework that provides a uniform accessing interface and support multitask scheduling on a heterogeneous cluster. We also have room to further optimize the current KPM implementation, for example, the "roofline" model [105] could be applied as a tool to analyze the KPM algorithm's characteristics on a given hardware architecture.

Chapter 9

Acknowledgments

It is for sure that with out the guidance, help and support from my professors, friends and my family, this dissertation would not be accomplished.

First and foremost, I would like to demonstrate my deep appreciation to my supervisor in University of Tsukuba, Prof. Yamagiwa, without whom I would not get my PhD in less than three years. I will never forget that it is him who picked me from China and therefore I have a chance to make a difference to my life. It was because of his effort that I was rewarded the scholarship from China Scholarship Council. In addition, Prof. Yamagiwa provides me with excellent research platform and environment where I could use the cluster freely, which was such a great help for me to accomplish the experiment for my papers. I have also learned so many things under his supervision, such as how to conduct research in computer science, how to organize materials and write scientific papers. With no doubt, these skills gained through so much training are most precious treasures throughout my life.

I also would like to express my grateful appreciation to Dr. Yunoki, associate chief scientist in Computational Condensed Matter Physics Laboratory of RIKEN, where I was assigned an interesting project in which I began to lift my confidence and realize that I could do something different. In addition, he help me with kindness and great patience, without his insightful comments and suggestions, I would not get over the difficulties I ever met in my research. I also want to thank Dr. Okumura who used to be very kind and helpful, without his comprehensive explanations

I could not push my work forward smoothly at the beginning.

In addition, I would like to deliver my grateful thanks to Prof. Wada, chief of my dissertation committee, for his comprehensive advice and guidance to help me prepare the necessary documents for the dissertation. Without his kind advice and help, my dissertation would not proceed so smoothly.

In the pre-dissertation, I received from the committee professors Prof. Yasunaka, Prof. Sakurai and Prof. Yamaguchi, very insightful advice, suggestions and comments, which are so helpful for me improving the quality of my thesis and the final dissertation. I am so grateful for their kindness.

I also would like to thank Prof. Chaoyang Li and my friends in Kochi University of Technology as well as in University of Tsukuba, for their motivated words and sincere encouragement.

Thanks for the support and encouragement from Prof. Ronghui Luo, Prof. Qiang Sun and Prof. Fuming Zhou in Zhengzhou University, their warm reception when I was staying in Zhengzhou University will be always appreciated.

The secretaries and staffs also helped me immensely. Ms. Ajiro in RIKEN, Ms. Sakurai in University of Tsukuba and Ms. Yamanaka in Kochi University of Technology require special mention here.

Last but not the least, I must thank my enlightened parents and family for their precious support and understanding on my pursing PhD in Japan. Every year I could spend only about 10 days with them that make me feel very guilty, they have sacrificed so much that I could never pay back. For many years, their prayers and sacrifices have been always the greatest encouragement to keep me moving forward.

Appendix A: A case study of evaluation of DOS

To demonstrate the availability of KPM, here the DOS for Anderson model of disorder is evaluated,

$$H = -t \sum_{<i,j>} (c_i^\dagger c_j + H.c.) + \sum_i \epsilon_i c_i^\dagger c_i \tag{9.1}$$

where $t = 1$ and ϵ_i is set to 0 for the sake of simplicity.

Applying this model, the DOS function for a cubic lattice of size 128^3 is evaluated with expansion order N=512 and N=128, respectively. The Hamiltonian matrix is compressed using CSR format. In the calculation, the parameter R and S, which represent the number of random vectors, is chosen to be 14 and 128, respectively.

Figure 9.1 visualized the DOS function, it can be noticed that the larger N, i.e. 512, leads to higher spectrum resolution.

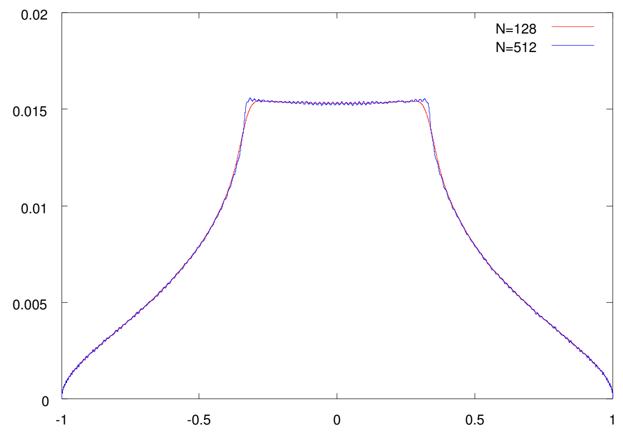


Figure 9.1: The DOS comparison with truncation between N=128 and N=512 when the lattice is $128 \times 128 \times 128$, $R = 14$ and $S = 128$.

Appendix B: Evaluation of LDOS for Anderson disorder model of a cubic lattice system

The GPU implementation proposed in section 5.3.2 is verified through calculating LDOS for Anderson model of disorder,

$$H = -t \sum_{<i,j>} (c_i^\dagger c_j + H.c.) + \sum_i \epsilon_i c_i^\dagger c_i \tag{9.2}$$

where $t = 1$ and ϵ_i is a random number in uniform distribution between $(-\omega/2, \omega/2)$, in which ω represents the magnitude of disorder.

Figure 9.2 a) and b) demonstrate spatial distribution of $\rho_i(\tilde{\omega} = 0)$ of a 40^3 lattice model with weak $(\omega = 3t)$ and strong disorder $(\omega = 18t)$, respectively. Each lattice site is represented by a sphere, with transparency corresponding to the magnitude of $\rho_i(\tilde{\omega} = 0)$ (the larger value is represented by the darker shadow). With weak disorder, the calculation shows extended states (Figure 9.2 a)), where the wave function of electron is spread over the whole system, suggesting that the system is conducting. With strong disorder demonstrated in Figure 9.2 b), electrons are confined to finite regions, suggesting that the system is insulating.

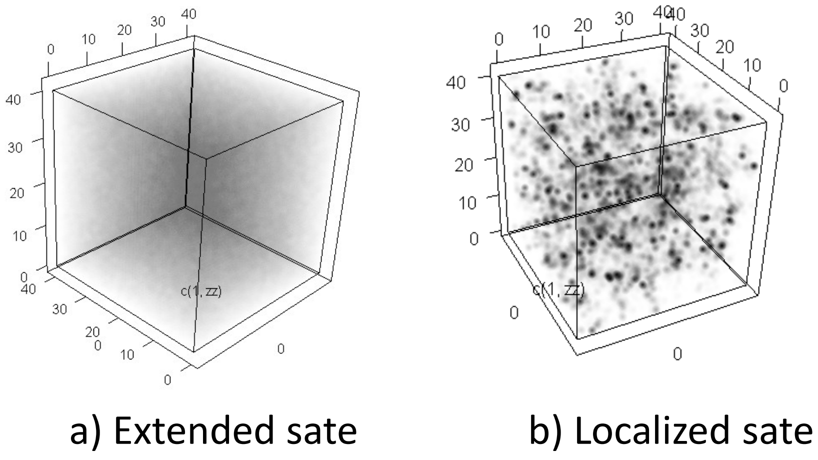


Figure 9.2: a) Extended states $(\omega = 3t)$ and 2) Localized states $(\omega = 18t)$ resulted from the KPM for solving LDOS.

Appendix C: Monte Carlo Simulation of Double Exchange (DE) Model

In order to check the availability of the GPU implementation, a DE model on the simple cubic lattice is simulated. The magnetization M as a function of temperature T is examined and the results are shown in Figure 9.3. The simulations are performed with the following parameters: Chebyshev truncation number is 256, chemical potential μ is 0, the average is taken over 5000 MC sweeps.

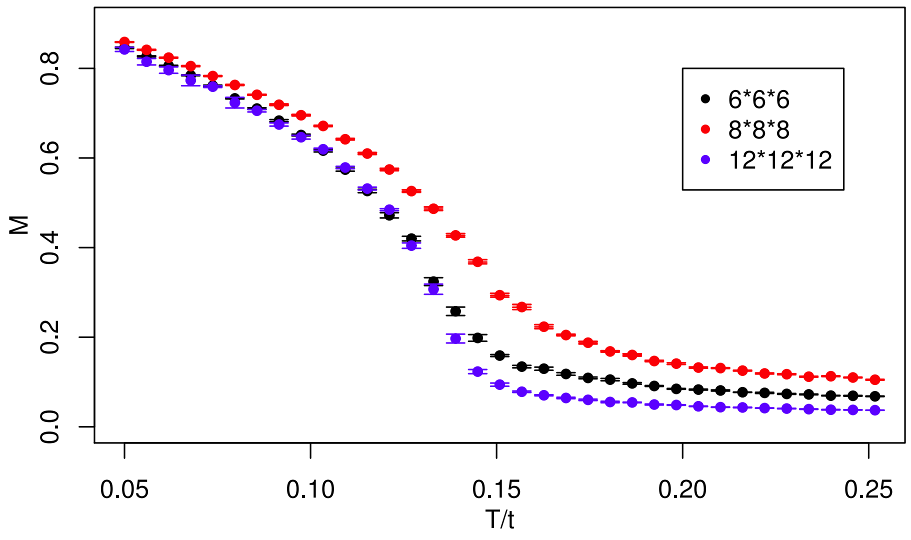


Figure 9.3: Magnetization M as a function of temperature T for different sizes (indicated in the figure) of the simple cubic lattice.

References

[1] S. M. Sze, *Semiconductor Devices: Physics and Technology*. Wiley, 2002.

[2] B. D. Cullity and C. D. Graham, *Introduction to Magnetic Materials*. Wiley-IEEE Press, 2008.

[3] Rose-Innes, A. Christopher, and E. Rhoderick, *Introduction to superconductivity*. Pergamon Press Oxford, 1969.

[4] G. Binasch, P. Grünberg, F. Saurenbach, and W. Zinn, "Enhanced magnetoresistance in layered magnetic structures with antiferromagnetic interlayer exchange," *Phys. Rev. B*, vol. 39, pp. 4828–4830, 1989.

[5] M. N. Baibich *et al.*, "Giant magnetoresistance of (001)fe/(001)cr magnetic superlattices," *Phys. Rev. Lett.*, vol. 61, pp. 2472–2475, 1988.

[6] C. Shong, S. Haur, and A. Wee, *Science at the Nanoscale: An Introductory Textbook*. Pan Stanford Publishing, 2010.

[7] K. Ohno, K. Esfarjani, and Y. Kawazoe, *Computational Materials Science*. Springer, 1999.

[8] 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*, vol. 64, no. 2, pp. 189–193, 1986.

[9] M. Yamashita *et al.*, "Highly mobile gapless excitations in a two-dimensional candidate quantum spin liquid," *Science*, vol. 328, no. 5983, pp. 1246–1248, 2010.

[10] E. Dagotto, "Correlated electrons in high-temperature superconductors," *Review of Modern Physics*, vol. 66, no. 3, pp. 763–840, 1994.

[11] W. Foulkes, L. Mitas, R. Needs, and G. Rajagopal, "Quantum monte carlo simulations of solids," *Review of Modern Physics*, vol. 73, no. 1, pp. 33–83, 2001.

[12] J. Grotendorst, D. Mark, and A. Muramatsu, *Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms*. NIC-Directors, 2002.

[13] S. R. White, "Density matrix formulation for quantum renormalization groups," *Physical Review Letters*, vol. 69, no. 19, pp. 2863–2866, 1992.

[14] S. R. White, "Density-matrix algorithms for quantum renormalization groups," *Physical Review B*, vol. 48, no. 14, pp. 10 345–1035, 1993.

[15] U. Schollwöck, "The density-matrix renormalization group," *Review of Modern Physics*, vol. 77, no. 1, pp. 259–315, January 2005.

[16] S. Yamada, M. Okumura, T. Imamura, and M. Machida, "Direct extension of the density-matrix renormalization group method toward two-dimensional large quantum lattices and related high-performance computing," *Japan Journal of Industrial and Applied Mathematics*, vol. 28, no. 1, pp. 141–151, 2011.

[17] A. Weiße, G. Wellein, A. Alvermann, and H. Fehske, "The kernel polynomial method," *Review of Modern Physics*, vol. 78, no. 1, pp. 275–306, January 2006.

[18] N. Furukawa and Y. Motome, "Order N Monte Carlo Algorithm for Fermion Systems Coupled with Fluctuating Adiabatical Fields," *Journal of the Physical Society of Japan*, vol. 73, pp. 1482–1489, June 2004.

[19] A. Weiße, "Green-Function-Based Monte Carlo Method for Classical Fields Coupled to Fermions," *Physical Review Letters*, vol. 102, no. 15, pp. 17–20, April 2009.

[20] S. Zhang, S. Yamagiwa, M. Okumura, and S. Yunoki, "Performance acceleration of kernel polynomial method applying graphics processing units," *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, vol. 0, pp. 569–576, 2011.

[21] E. F. V. de Velde, *Concurrent Scientific Computing*. Springer, 1994.

[22] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press, 1992.

[23] M. S. Susan L. Graham and C. A. Patterson, *Getting Up to Speed: The Future of Supercomputing*. The National Academies Press, 2004.

[24] Top500, "Top 500 supercomputer sites," http://www.top500.org/, 2013.

[25] Argonne National Laboratory, "The message passing interface (mpi) standard," 2013. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/

[26] I. Foster, *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison Wesley, 1995.

[27] W. Gropp, E. Lusk, and A. Skjellum, *Using Mpi: Portable Parallel Programming With the Message-Passing Interface*. The MIT Press, 1999.

[28] OpenMP Architecture Review Board, "Openmp," 2013. [Online]. Available: http://openmp.org/wp/openmp-specifications/

[29] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[30] D. Godse, *Microprocessors And Applications*. Technical Publications, 2009.

[31] Intel Corporation, *Microprocessors*. Intel Corporation, 1992.

[32] J. Evans and G. Trimper, *Itanium Architecture for Programmers: Understanding 64-Bit Processors and Epic Principles*. Prentice Hall PTR, 2003.

[33] S. Burd, *Systems Architecture: Hardware and Software in Business Information Systems*. Course Technology, 2010.

[34] G. Dahlquist and Å. Björck, *Numerical Methods in Scientific Computing: Vol. 1*. SIAM, 2008.

[35] M. Overton, *Numerical Computing with IEEE Floating Point Arithmetic: Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*. Society for Industrial and Applied Mathematics, 2001.

[36] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier Science, 2013.

[37] A. Sarje, J. Zola, and S. Aluru, *Scientific Computing with Multicore and Accelerators*. Chapman and Hall/CRC, 2010.

[38] NVIDIA Corporation, "Tesla supercomputing solutions," 2008. [Online]. Available: http://www.nvidia.com/object/tesla-supercomputing-solutions.html

[39] Intel Corporation, "The intel xeon phi coprocessor: Parallel processing, unparalleled discovery," 2012. [Online]. Available: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

[40] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2010.

[41] NVIDIA Corporation, "CUDA: Compute unified device architecture programming guide," 2008. [Online]. Available: http://developer.nvidia.com/cuda

[42] B. Kramer and A. MacKinnon, “Localization: theory and experiment,” *Reports on Progress in Physics*, vol. 56, no. 12, pp. 1469–1564, 1993.

[43] G. Schubert, J. Schleede, K. Byczuk, H. Fehske, and D. Vollhardt, “Distribution of the local density of states as a criterion for anderson localization: Numerically exact results for various lattices in two and three dimensions,” *Phys. Rev. B*, vol. 81, p. 155106, 2010.

[44] C. Zener, “Interaction between the d-shells in the transition metals,” *Phys. Rev.*, vol. 82, pp. 403–405, May 1951.

[45] R. Farber, *CUDA Application Design and Development*. Elsevier Science, 2011.

[46] S. Cook, *CUDA Programming: A Developer’s Guide to Parallel Computing With GPUs*. Elsevier Science, 2012.

[47] NVIDIA Corporation, “Tesla product literature.” [Online]. Available: http://www.nvidia.com/object/tesla_product_literature.html

[48] I. Zelinka, V. Snasel, and A. Abraham, *Handbook of Optimization: From Classical to Modern Approach*. Springer, 2012.

[49] A. Lastovetsky, *Parallel Computing on Heterogeneous Networks*. John Wiley & Sons, 2008.

[50] G. Wellein and H. Fehske, “Towards the limits of present-day supercomputers: Exact diagonalization of strongly correlated electron-phonon systems,” *High Performance Computing in Science and Engineering ’99*, pp. 112–129, 2000.

[51] D. Sénéchal, A. Tremblay, and C. Bourbonnais, *Theoretical Methods for Strongly Correlated Electrons*. Springer, 2004.

[52] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for fortran usage,” *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[53] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK’s user’s guide*. Society for Industrial and Applied Mathematics, 1992.

[54] M. Razeghi, *Fundamentals of Solid State Engineering*. Springer, 2009.

[55] L. Mihály and M. Martin, *Solid State Physics*. Wiley, 2009.

[56] R. Martin, *Electronic Structure: Basic Theory and Practical Methods*. Cambridge University Press, 2004.

[57] R. Varga, *Geršgorin and His Circles*. Springer, 2004.

[58] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” *Proceedings of the spring joint computer conference*, pp. 483–485, 1967.

[59] Y. Barlas, *Role of Electron-electron Interactions in Chiral 2DEGs*. The University of Texas, 2008.

[60] J. Chen and W. Hu, “Reproducing kernel partition of unity: from continuum to quantum,” *Computational Mechanics*, pp. 167–179, 2009.

[61] S. Pissanetzky, *Sparse Matrix Technology*. Academic Press, 2007.

[62] R. P. Tewarson, *Sparse matrices*. Elsevier Science, 1973.

[63] T. Davis, *Direct Methods for Sparse Linear Systems*. Cambridge-USA, 2006.

[64] J. Dongarra. (1995) Compressed row storage (crs). [Online]. Available: http://netlib.org/linalg/html_templates/node91.html

[65] R. Grimes, D. Kincaid, and D. Young, *ITPACK 2.0 User’s Guide*. University of Texas, 1979.

[66] P. Micikevicius, “Identifying performance limiters,” 2011. [Online]. Available: http://developer.download.nvidia.com/CUDA/training/cuda_webinars_dentifying_performance_limiters.pdf

[67] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[68] E. Photonics, “Gpu accelerated linear algebra,” 2013. [Online]. Available: http://www.culatools.com/

[69] NVIDIA Corporation, “Cublas,” 2013. [Online]. Available: https://developer.nvidia.com/cublas

[70] N. Bell and M. Garland. (2009) Cusp : Generic parallel algorithms for sparse matrix and graph computations. [Online]. Available: http://code.google.com/p/cusp-library/

[71] P. Sexton, *A Performance Analysis of Sparse Matrix-vector Multiplication in a Templated C++ Linear Algebra Library*. University of Illinois at Chicago, 2006.

[72] G. Strang, *Introduction to linear algebra*. Wellesley-Cambridge Press, 2003.

[73] M. Baskaran and R. Bordawekar, “Optimizing sparse matrix-vector multiplication on gpus,” IBM Research, Tech. Rep., December 2012.

[74] J. D. Davis and E. S. Chung, “Spmv: A memory-bound application on the gpu stuck between a rock and a hard place,” Microsoft Research, Tech. Rep., 2012.

[75] K. K. Erik Saule and Ü. V. Çatalyürek, “Performance evaluation of sparse matrix multiplication kernels on intel xeon phi,” The Ohio State University, Tech. Rep., 2013.

[76] B. Nathan and G. Michael, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11, 2009.

[77] N. Bell and M. Garland, “Efficient sparse matrix-vector multiplication on CUDA,” NVIDIA Corporation, Tech. Rep., December 2008.

[78] G. Jeswin, H. Justin, and P. Sadayappan, “High-performance sparse matrix-vector multiplication on gpus for structured grid computations,” *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pp. 47–56, 2012.

[79] F. Vázquez, G. Ortega, J. Fernández, and E. Garzón, “Improving the performance of the sparse matrix vector product with gpus,” *Computer and Information Technology (CIT)*, pp. 1146–1151, 2010.

[80] M. M. Wolf, E. G. Boman, and B. A. Hendrickson, “Optimizing parallel sparse matrix-vector multiplication by corner partitioning,” *PARA08*, 2008.

[81] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[82] F. Vázquez, E. M. Garzón, J. A. Martinez, and J. J. Fernández, “The sparse matrix vector product on GPUs,” University of Almeria, Tech. Rep., June 2009.

[83] S. Dandamudi, *Fundamentals of Computer Organization and Design*. Springer, 2003.

[84] NVIDIA Corporation, “Cuda c best practices guide,” 2012. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[85] T. Kukuk, “Homepage of the linux nis/nis+ projects.” [Online]. Available: http://www.linux-nis.org/

[86] M. T. Jones, “Network file systems and linux.” [Online]. Available: http://www.ibm.com/developerworks/library/l-network-filesystems/

[87] N. W. Ashcroft and N. D. Mermin, *Solid State Physics*. Brooks/Cole, 1975.

[88] S. Zhang, S. Yamagiwa, M. Okumura, and S. Yunoki, “Parallelizing kernel polynomial method applying graphics processing units,” *International Journal of Networking and Computing*, vol. 2, no. 1, 2012.

[89] S. Zhang, S. Yamagiwa, M. Okumura, and S. Yunoki, “Kernel polynomial method on gpu,” *International Journal of Parallel Programming*, vol. 41, no. 1, pp. 59–88, 2013.

[90] G. Schubert and H. Fehske, “Quantum percolation in disordered structures,” *Quantum and Semi-classical Percolation and Breakdown in Disordered Solids*, vol. 762, pp. 1–28, 2009.

[91] P. W. Anderson, “Absence of diffusion in certain random lattices,” *Physical Review*, vol. 109, no. 5, pp. 1492–1505, March 1958.

[92] D. J. Thouless, “Electrons in disordered systems and the theory of localization,” *Physics Reports*, vol. 13, no. 3, pp. 93–142, October 1974.

[93] P. Lee and T. V. Ramakrishnan, “Disordered electronic systems,” *Review of Modern Physics*, vol. 57, no. 2, pp. 287–337, April 1985.

[94] V. Dobrosavljević and G. Kotliar, “Mean field theory of the mott-anderson transition,” *Physical Review Letters*, vol. 78, no. 20, pp. 3943–3946, March 1997.

[95] G. Schubert, A. Weiße, and H. Fehske, “Localization effects in quantum percolation,” *Physical Review B*, vol. 71, no. 4, p. 045126, January 2005.

[96] S. Zhang, S. Yamagiwa, and S. Yunoki, “A study of parallelizing o(n) green-function-based monte carlo method for many fermions coupled with classical degrees of freedoms,” *Journal of Physics: Conference Series*, 2013.

[97] P. G. de Gennes, “Effects of double exchange in magnetic crystals,” *Phys. Rev.*, vol. 118, pp. 141–154, April 1960.

[98] N. Metropolis and S. Ulam, “The monte carlo method,” *Journal of the American statistical Association*, vol. 44, no. 247, pp. 335–341, 1949.

[99] N. Metropolis, “The beginning of the monte carlo method,” *Los Alamos Science*, January 1987.

[100] S. Yunoki, J. Hu, A. L. Malvezzi, A. Moreo, N. Furukawa, and E. Dagotto, “Phase separation in electronic models for manganites,” *Phys. Rev. Lett.*, vol. 80, pp. 845–848, January 1998.

[101] S. Yunoki and A. Moreo, “Static and dynamical properties of the ferromagnetic kondo model with direct antiferromagnetic coupling between the localized t_{2g} electrons,” *Phys. Rev. B*, vol. 58, pp. 6403–6413, September 1998.

[102] E. Dagotto, S. Yunoki, A. L. Malvezzi, A. Moreo, J. Hu, S. Capponi, D. Poilblanc, and N. Furukawa, “Ferromagnetic kondo model for manganites: Phase diagram, charge segregation, and influence of quantum localized spins,” *Phys. Rev. B*, vol. 58, pp. 6414–6427, September 1998.

[103] Y. Motome and N. Furukawa, “A Monte Carlo Method for Fermion Systems Coupled with Classical Degrees of Freedom,” *Journal of the Physical Society of Japan*, vol. 68, pp. 3853–3858, December 1999.

[104] P. Duhamel and M. Vetterli, “Fast fourier transforms: A tutorial review and a state of the art,” *Signal Processing*, vol. 19, no. 4, pp. 259 – 299, 1990.

[105] S. Williams, A. Waterman, and P. David, “Roofline: an insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, April 2009.

Publication List

Journals

[1] Shixun Zhang, Shinichi Yamagiwa, and Seiji Yunoki, "A Study of Parallelizing O(N) Green-Function-based Monte Carlo Method for Many Fermions Coupled with Classical Degrees of Freedoms," *Journal of Physics: Conference Series*, IOP, 2013

[2] Shixun Zhang, Shinichi Yamagiwa, Masahiko Okumura, and Seiji Yunoki, "Kernel Polynomial Method on GPU," *International Journal of Parallel Programming*, vol. 41, no. 1, pp. 59-88, Springer, 2013

[3] Shixun Zhang, Shinichi Yamagiwa, Masahiko Okumura, and Seiji Yunoki, "Parallelizing Kernel Polynomial Method Applying Graphics Processing Units," *International Journal of Networking and Computing*, vol. 2, no. 1, pp. 41-55, Hiroshima University, 2012

Peer-Reviewed Conferences

[1] Shinichi Yamagiwa and Shixun Zhang, "Scenario-based Execution Method for Massively Parallel Accelerators," in *Proceedings of International Symposium on Parallel and Distributed Processing with Applications (ISPA-13)*, IEEE CS, 2013

[2] Shinichi Yamagiwa and Shixun Zhang, "CarSh: A Commandline Execution Support for Stream-based Acceleration Environment," in *Proceedings of International Conference on Computational Science*, Elsevier, Nov. 2013

[3] Shixun Zhang, Shinichi Yamagiwa, and Seiji Yunoki, "GPU-based Parallelization of Kernel Polynomial Method for Solving LDOS," in *Proceedings of ScalA/Supercomputing12*, pp. 633-642, IEEE CS, 2012

[4] Shixun Zhang, Shinichi Yamagiwa, Masahiko Okumura, and Seiji Yunoki, "Performance Acceleration of Kernel Polynomial Method Applying Graphics Processing Units," in *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 569-576, IEEE CS, 2011

[5] Shixun Zhang, Shinichi Yamagiwa, Masahiko Okumura, and Seiji Yunoki, "Performance Impact Applying Compression Format to Sparse Matrix on Kernel Polynomial Method Using GPU," in *Proceedings of the Second International Conference on Networking and Computing*, pp. 337-341, IEEE CS, 2011