2013

GPU acceleration of the Variational Monte Carlo Method for Many Body Physics

Kaushik Ragavan Rajagopalan
Louisiana State University and Agricultural and Mechanical College, kaushikragavan89@gmail.com

GPU ACCELERATION OF THE VARIATIONAL MONTE CARLO METHOD FOR MANY BODY PHYSICS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Rajagopalan Kaushik Ragavan
B.ENG in Electronics and Communication,
Anna University, Chennai, May 2010
May 2013

Acknowledgments

My sincere thanks to Dr. David Koppelman for his valuable teachings on GPU, CPU architecture and on High Performance computing, which motivated me to pick my thesis in this domain. He was a great mentor and guided me on every step of my thesis.

I would like to thank Dr.Mark Jarrell, Dr.Juana Moreno and the LA-Sigma group for funding this research.

I would like to thank Dr.Xin Li and Dr.Juana Moreno for accepting my request to be a part of the thesis committee.

I would also like to thank other colloborators: Dr.Ka Ming Tam, Dr.Zhifeng Yun, Niladri Sengupta and Dr.Sandeep Pathak for their valuable contributions and sharing of thoughts.

Finally, I would like to thank my parents for supporting my education till now. It was their motivation and trust which made my degree possible.

Table of Contents

APPENDIX

List of Tables

List of Figures

Abstract

High-Performance computing is one of the major areas making inroads into the future for large-scale simulation. Applications such as 3D nuclear test, Molecular Dynamics, and Quantum Monte Carlo simulations are now developed on supercomputers using the latest computing technologies. As per the TOP500 supercomputers rating, most of today‘s supercomputers are now *heterogeneous*: with massively parallel Graphics Processing Units (GPU) equipped with Multi-core CPU(s) to increase the computational capacity.

The Variational Monte Carlo(VMC) method is used in the Many Body Physics to study the ground state properties of a system. The wavefunction depends on some variational parameters, which contain the physics for a better prediction. In general, the variational parameters are chosen to realize some sort of order or broken symmetry such as *superconductivity* and *magnetism*.

The variational approach is computationally expensive and requires a large number of trajectories to obtain convergence. The Markov chains (MCs) exhibit abundant *data parallelism* and parallelizing across CPU clusters will prove to be expensive and does not scale in proportion to the system size. Hence, this method will be a suitable candidate on a massively parallel Graphics Processing Unit (GPU).

In this research, we discuss about the various optimization and parallelization strategies adopted to port the VMC method to a NVIDIA GPU using CUDA. We obtained a speedup of nearly 3.85 X compared to the MPI implementation [4] and a speedup of upto 19 X compared to an object-oriented C++ code.

Chapter 1
Variational Monte Carlo (VMC)

1.1 Trial Wavefunction

Variational Monte Carlo (VMC) method is a direct application of Monte Carlo integration to strongly correlated systems. The variational approach has been used widely in different areas of condensed matter physics, in particular the d-wave superconducting state of the high T_C cuprates at T=0 [1]. In quantum mechanics, variational principle can be derived by expanding a normalized trial wavefunction, Ψ_T, in terms of the exact normalized eigenstates of the Hamiltonian [9].

$$\Psi_T = \sum_{i=0}^{\infty} c_i \Psi_i, \tag{1.1}$$

where c_i is given by,

$$\sum_{i=0}^{\infty} |c_i|^2 = 1 \tag{1.2}$$

The many-body Hamiltonian, $\hat{H}$, evaluated by,

$$\left\langle \Psi_T \middle| \hat{H} \middle| \Psi_T \right\rangle = \left\langle \sum_i c_i \Psi_i \middle| \hat{H} \middle| \sum_j c_j \Psi_j \right\rangle = \sum_i |c_i|^2 \epsilon_i, \tag{1.3}$$

where $\epsilon_i = \left\langle \Psi_i \middle| \hat{H} \middle| \Psi_i \right\rangle$

From the above equations [9], the expectation value of a trial wavefunction with the Hamiltonian must be greater than or equal to the true ground state energy. The variational method depends mostly on the trial wavefunction used. Wavefunctions are normally obtained by Hartree-Fock or similar methods and additional parameters added to build in additional physics such as, known limits and derivatives of the many-body wavefunction. The wavefunction is then further optimized by the variational parameters.

From [4], an example of a trial wavefunction is then taken to be,

$$\Psi(R) = D(R)exp\left[\sum_{i<j}^{N} -u(r_{ij})\right], \tag{1.4}$$

where D is a determinant of Hartree-Fock or meanfield solutions. The variational parameters for constructing D is used to optimize the trial wavefunction. The extra projection factor u is included in the above wavefunction.

1.2 The VMC Algorithm

The VMC algorithm consists of two distinct phases [9, 5]: Equilibration and Accumulation (Measurement). In the first phase, the system is made to equilibrate and sampled for $|\Psi|^2$. In the second phase, the energies and other observables are accumulated. Thus we perform random walk across configurations.

1. **Equilibration:**

   (a) Generate an initial random configuration.

   (b) For each electron in the configuration:

      i. Propose a move from m_r to m_r'

      ii. Compute the ratio $R = |\Psi(m_r{}')/\Psi(m_r)|^2$

      iii. Perform metropolis acceptance comparison $min(1, R)$

      iv. If the move is accepted, update the configuration.

      v. Else restore the old configuration.

   (c) Repeat the above steps until the system equilibrates.

2. **Accumulation:**

   (a) Repeat the same procedure from Equilibration.

   (b) Accumulate the local energy and other observable parameters at m'_r and m_r,

   (c) Perform metropolis acceptance comparison $min(1, R)$

   (d) Repeat the above steps until energies are accumulated

1.2.1 Difficulty of VMC

The energy function [5] is defined as,

$$E(\alpha_i) = \frac{\sum_C \Psi^*_\alpha(C) H \Psi_\alpha(C)}{\sum_C \Psi^*_\alpha(C)\Psi_\alpha(C)} \tag{1.5}$$

In many of these cases, the wavefunction in real space is given by, $\Psi_\alpha(r_{1\uparrow}, \ldots, r_{N\uparrow}, r_{1\downarrow}, \ldots, r_{N\downarrow})$, where $r_{i\sigma}$ are the coordinates of the electrons on a lattice and $C \equiv (r_{1\uparrow}, \ldots, r_{N\uparrow}, r_{1\downarrow}, \ldots, r_{N\downarrow})$, a configuration of electrons.

To sum over all configurations, for example: a lattice with 100 sites and 50 $\uparrow$ and 50 $\downarrow$ electrons, we need to visit over 10^{60} configurations. To overcome this difficulty, we use Monte Carlo method to perform the sum[2].

$$E(\alpha_i) = \sum_C P(C) \frac{H\Psi_\alpha(C)}{\Psi_\alpha(C)} \tag{1.6}$$

where $P(C)$, *the probability of the configuration*, is given by,

$$P(C) = \frac{|\Psi_\alpha(C)|^2}{\sum_C \Psi^*_\alpha(C)\Psi_\alpha(C)}$$

For any operator O

$$\langle O \rangle = \sum_{C} P(C) \frac{O\Psi_\alpha(C)}{\Psi_\alpha(C)} \tag{1.7}$$

where, $\sum_C P(C) = 1$.

To avoid visiting all the configurations, we will visit the "most important" configurations [5] and add up the corresponding contribution. The configurations with "high probability" are considered to be important. This is called as *importance sampling* [5].

Using the importance sampling, accurate results for various quantites can be obtained by a smaller number of Monte Carlo sweeps, N_{MC}.

$$E(\alpha_i) = \frac{1}{N_{MC}} \sum_{k=1}^{N_{MC}} \frac{H\Psi_\alpha(m_r)}{\Psi_\alpha(m_r)} \tag{1.8}$$

For other operators,

$$\langle O \rangle_\alpha = \frac{1}{N_{MC}} \sum_{k=1}^{N_{MC}} \frac{O\Psi_\alpha(m_r)}{\Psi_\alpha(m_r)} \tag{1.9}$$

For every Monte Carlo (MC) step, we need to evaluate the ratio of $\frac{|\Psi(m_{r'}|^2}{|\Psi(m_r|^2}$ which is of complexity $O(N^3)$. In order to optimize with respect to α_i, we need a complexity of $O(N)$.

1.2.2 Faster VMC

Consider the spinless fermions [5, 4] with a configuration k given by a wavefunction Ψ and a configuration l given by a wavefunction Φ. Both the configurations differ only by a position of a electron e, e_l and e_l'

$$\begin{bmatrix} \psi_{a1}(e_1) & \dots & \psi_{a1}(e_l) & \dots & \psi_{a1}(e_N) \\ \vdots & \dots & \vdots & \dots & \vdots \\ \psi_{aN}(e_1) & \dots & \psi_{aN}(e_l) & \dots & \psi_{aN}(e_N) \end{bmatrix} \tag{1.10}$$

Since Ψ and Φ differ by only one column, the ratio can be determined as[2],

$$\frac{det[\Phi]}{det[\Psi]} = \sum_k \Psi^{-1}_{kl} \Phi_{kl} \tag{1.11}$$

which of $O(N)$

The calculation of Ψ^{-1} is reduced to the order of $O(N^2)$ using Sherman-Morrison-Woodbury (SMH) method.

1.2.3 Summary of VMC method

1. Start with a random configuration
2. For $k = 0, N_{MC}$
   (a) Pick an electron at random for the configuration m_r and move to a random position. Name this configuration m'_r.

   (b) Check for Probability, accept the configuration if the ratio is greater than a uniform random number: $min\left\{1, \frac{|\Psi(m'_r)|^2}{|\Psi(m_r)|^2}\right\}$, set $m_{r+1} = m'_r$ if accepted, else set $m_{r+1} = m_r$.

   (c) Perform the update of Ψ^{-1} using SMH formula.

   (d) Wait until the system Equilibriates.

   (e) Repeat the above steps (a - c).

   (f) Accumulate the energy contributions.

$$\frac{O\Psi_\alpha(m_{r+1})}{\Psi_\alpha(m_{r+1})}$$

3. Determine necessary averages for $E(\alpha_i)$.

4. Optimize over α_i to obtain $|\Psi_G\rangle$.

5. Study the ground state $|\Psi_G\rangle$.

The VMC method is tested on a tilted square lattice [4, 1]. The Resonance Valance Bond model is used to represent the high T_C. The number of sites, N_S is given by $N_S = L^2 + 1$, where L is odd. The number of electron pairs, N_P is given by, $N_P = N_e/2$ where N_e, is the number of electrons. The number of electrons can be calculated based on a given hole doping (x), $Nsites \times (1.0 - x)$ [5].

The Hubbard model for this method is given by,

$$-\sum_{ij} t_{ij} c_{i\sigma} c_{j\sigma} + U \sum_{i} n_{i\uparrow} n_{i\downarrow} \tag{1.12}$$

where $U = 0$ (free electrons) and $U = \infty$ (Extremely correlated liquid) and the model is a $t - t' - t''$ model to study the material dependencies.

Chapter 2
Prior Work

2.1 CPU Implementation

In [1, 4], the VMC method is used to study the competition between Antiferromagnetic and Superconducting states in High-T_C superconductors. A Fortran implementation of VMC was done by [4] by simulating identical systems, called *Markov chains (MCs)*, each initialized randomly. Since the initial implementation was a sequential version and in order to exploit the parallelism involved in the MCs, the programming model was converted to a MPI version [5].

2.2 MPI Implementation

A MPI Fortran implementation was done by [5, 4] to study the d-correlated systems using the variational approach. The programming model designates a MPI rank (or processor) per Markov chain and requires inter-processor and inter-node communication to calculate the average energy and error distribution. This implementation does not utilize cache blocking for regions within the code which require *locality of reference*. Since, designating a CUDA thread per MC will result in a poor performance due to the working set exceeding the device limits, a better MPI implementation can still yield a good performance. However, the MPI version will be computationally expensive and we need to find a cost effective approach for the VMC algorithm.

2.3 GPU Implementation

The first step in porting VMC code to CUDA was done by Byron Tasseff, an REU student at our research group. A *naive* GPU implementation was done by [17] which laid the foundation for our research on tuning this algorithm. In his implementation, a CUDA thread performs a MC and the number of MCs correspond to the number of threads running in parallel. The implementation lacks enough *thread-level parallelism* and does not utilize the fast memory resources on a GPU, such as, shared and constant memory. Functions such as update of a configuration and ratio of determinant of the configurations have different memory access pattern and certain caching techniques can yield a better performance. In our GPU implementation, we address these issues and optimize the memory usage.

A GPU and a FPGA implementation was done by [18]. This implementation compares the performance of Quantum Monte Carlo calculations between a CPU, GPU and a FPGA. Atoms ranging from 256 to 8192 were tested on a Dual-Core-Dual-processor AMD Opteron @ 2.2 GHz, NVIDIA C1060 GPU and a Virtex-4 XC4VLX160 FPGA. Results show a speedup of ≈ 2 for the GPU implementation compared to the CPU version. The implementation does not effectively utilize the memory resources on a GPU and we address these issues in our code.

Chapter 3
Graphics Processing Units

3.1 Increasing Trend in Parallel Computing

The semiconductor industry has now settled on two main trajectories for microprocessor design [8]. The *Multicore* trajectory strives to improve the performance of sequential program, while doubling the number of cores with each generation. As an example, $Intel^{©}$ $Core^{TM} i7$ microprocessor has four processor cores, each of which is an out-of-order, multiple instruction issue processor. The processor implements the full x86 instruction set and supports hyperthreading with two threads per core [8]. On the contrary, a GPU increases the throughput of parallel applications. As an example, the NVIDIA Tesla $M2090^{©}$ supports upto 512 cores, Dual Warp Scheduler, which simultaneously schedules and dispatches instructions from two independent warps and NVIDIA *GIGATHREAD* for faster application context switching.

Figures 3.1, 3.2 show that NVIDIA GPU(s) have better FLOPS and Memory Bandwidth compared to their CPU counterpart [7]. Data-parallel appplications, when ported to GPU, are observed to get a performance boost compared to the CPU. This discrepancy is maily due to more number of transistors dedicated for data processing rather than data caching and flow control [8].

Figure 3.3 compares the architecture of a Multicore CPU and a GPU model. CPU dedicates more transistors for caching and control, whereas GPU dedicates it for computational units.

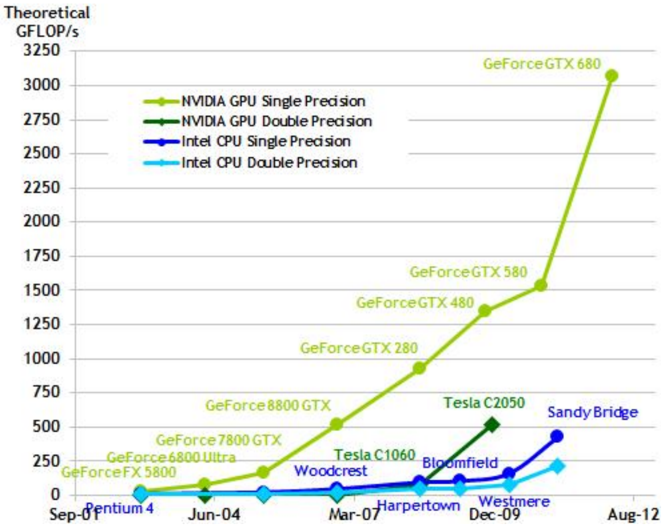


Figure 3.1: Floating Point operations per second for CPU vs GPU, Source: CUDA Programming guide

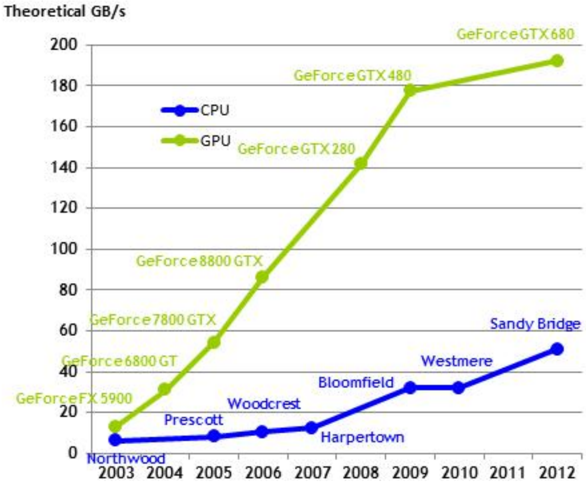


Figure 3.2: Memory Bandwidth for CPU vs GPU, Source: CUDA Programming guide

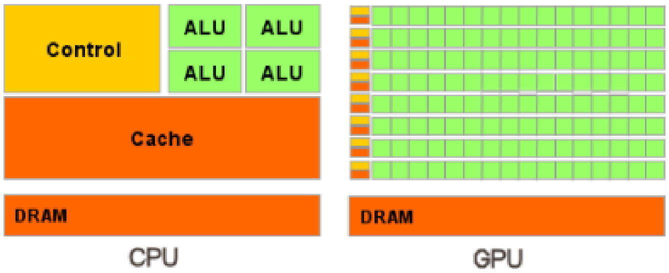


Figure 3.3: GPU vs CPU architecture, Source: CUDA Programming guide

3.2 CUDA Programming model

Introduced by NVIDIA in November 2006, $CUDA^{TM}$ is a new parallel programming model that leverages the compute engine in NVIDIA GPU(s) to solve complex computational problems efficiently than a traditional CPU [7].

The CUDA programming model is designed for an easy transition from the C code by a minimal set of language extensions, thereby providing a low learning curve for the programmer. The core has three key abstractions [7] - a hierarchy of thread groups, shared memories and barrier synchronization. These abstractions enable the programmer to partition the application to fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism.

Figure 3.4 shows the scaling model based on the Multiprocessor count (SM) on a GPU. The **threadIdx** can be of one, two or three-dimensions, hence forming a one, two, or three dimensional *thread block*. Each Streaming Multiprocessor (SM) is capable of hosting 1536 CUDA threads, with limited memory resources shared between them. The total number of resident *thread blocks* per SM is given by 8 for a fermi architecture [11]. Blocks are organized into a one, two or a three-dimensional *grid* of thread blocks.

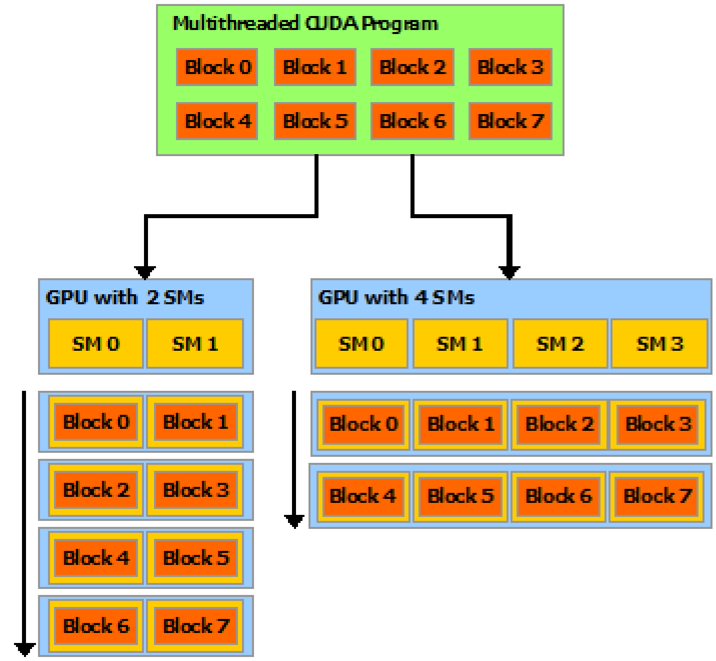


Figure 3.4: Automatic Scalability of Applications, Source: CUDA Programming guide

Each thread has a private local memory, and a thread block has a shared memory with read/write capability for all threads within the block [7]. Finally, all threads have access to the same global memory, two read-only memory spaces, constant and texture. The global, constant and texture memory spaces are optimized for different memory usage [7, 8]. *Global memory coalescing* should be followed to avoid longer memory latency. The fermi architecture offers the flexibility of partioning the shared/L1 memory space according to the application. The 64 KB memory space can be partioned into 16 KB shared/64 KB L1 or vice-versa. Figure 3.5, 3.6 shows the memory hierarchy and their access pattern.

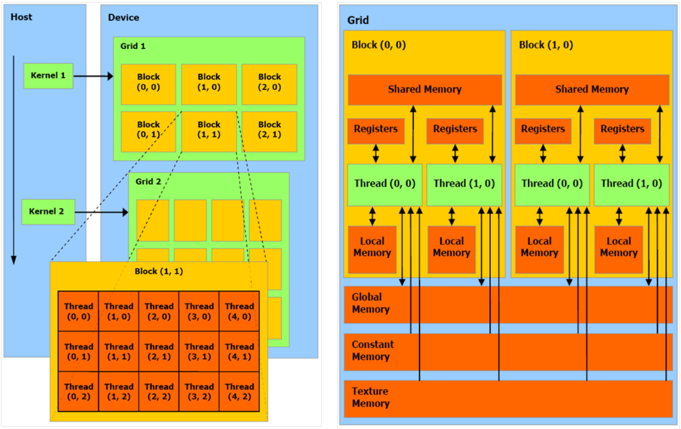


Figure 3.5: CUDA Thread hierarchy, Source: Programming Massively Parallel Processors- A hands-on Approach

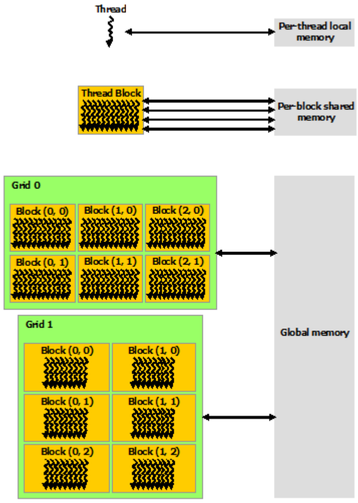


Figure 3.6: CUDA Memory Hierarchy, Source CUDA Programming guide

Chapter 4
CPU Implementation of VMC

VMC has been implemented targeting a cluster level system by [4, 5], a CPU, GPU and a reconfigurable Field Programmable Gate Array (FPGA) by cite [18]. In this chapter, we present a straightforward implementation of the VMC method using C++ based on an object-oriented approach. The following sections describe the pseudocode for the VMC method and its core components, structure of the CPU code, algorithm analysis and the results obtained for a single-core implementation. Based on our results and taking factors such as computational cost into consideration [17] and [4], we claim a cost-effective and efficient model can be implemented on a GPU.

4.1 Pseudocode for the VMC Method

In this section, we show the pseudocodes for the VMC method and its core stages: Equilibration and Accumulation given by Algorithms 4, 3. The VMC method, given by Algorithm 1 has four main stages: Electron move, calculation of ratio of determinants(DetPByDet), update of a configuration (UpdateConfig), and copy of a configuration (Copyconfig), if the move is rejected [5, 9]. The electron move procedure will pick a pair, spin at random and an oldsite from the electron occupancy list (plist). Now, a newsite is picked based on the neighbor probability and an electron is moved to this site. The spinflip is determined and then DetPByDet, UpdateConfig and DetPByDet are called in-order to determine the product of the ratio of determinant of the configurations ($dpbd1 \times dpbd2$). The norm of this product is compared to a real uniform random number and the move is accepted/rejected based on this. [5].

Algorithm 1 Overview of the VMC Method

```
begin vmc
  Initialization()        ▷ Generate lattice object, wavefunction and pairfunction
  Equilibration()         ▷ Perform the Equilibration procedure
  Accumulation()          ▷ Accumulate the energy and study the groundstate
end vmc
```

Algorithm 2 Equilibration

```
procedure Equilibration( )
    for $i \leftarrow 0, nsweeps$ do
        Electron move()                  ▷ Call the electron move procedure
        Perform MonteCarlo Sweep()       ▷ Determine the acceptance of the move
                                           and update the Configuration
    end for
end procedure
```

Algorithm 3 Accumulation

```
procedure Accumulation( )
    for $i \leftarrow 0, navesweeps$ do    ▷ Change the loop bounds to navesweeps and npsweep
        for $j \leftarrow 0, npsweep$ do   ▷ Repeat the procedures from Equilibration
            Electron Move()
            Perform MonteCarlo Sweep()
        end for
        $energy \leftarrow$ EnergyofConfig()        ▷ Call the energy of configuration
        $eneloc \leftarrow eneloc + energy$         ▷ Accumulate the energy
    end for
end procedure
```

Algorithm 4 Electron Move

```
procedure ELECTRON MOVE( )
    $ipair \leftarrow rand(Npairs)$                ▷ Pick a pair and spin at random
    $ispin \leftarrow rand(2)$
    $oldsite \leftarrow plist(ipair, ispin)$
    while $(newsite == 0)$ do                      ▷ Find a site for the electron move
        if $neiprob > 0.0$ then
            $newsite \leftarrow rand(Nsites)$
        else
            $newsite \leftarrow neiblist()$        ▷ neiblist maintains the neighbour information
        end if
        if $(latocc(newsite)\,!= (BT\,or\,HL))$ then   ▷ latocc will determine the spin at the newsite
            $spinflip \leftarrow true$
            $jpair \leftarrow whichpair(newsite * 2 + 1 - ispin)$   ▷ whichpair(ilat,spin), “spin” at lattice site ilat
        else
            $newsite = 0$
        end if
    end while
end procedure
```

Algorithm 5 Perform MonteCarlo Sweep

```
procedure PERFORM MONTECARLO SWEEP( )
                                        ▷ Determine the spinflip from the electron move stage
    if $spinflip = true$ then
                          ▷ Calculate the Ratio of determinants and Update the Configuration
        $dpbd1 \leftarrow$ DetPByDet($ipair, 2 * ispin - 1, newsite$)
        UpdateConfig($ipair, 2 * ispin - 1, newsite, dpbd1$)
        $dpbd2 \leftarrow$ DetPByDet($jpair, 1 - 2 * ispin, oldsite$)
        $dpbd \leftarrow dpbd1 * dpbd2$
    else
        $dpbd \leftarrow$ DetPByDet($ipair, 2 * ispin - 1, newsite$)
    end if
    $norm2 \leftarrow norm(dpbd)$                                  ▷ Determine $\frac{\Psi'^{-1}}{\Psi^{-1}}$
    if $norm2 \geq uniformrand(0, 1)$ then                         ▷ Move is accepted
        if $spinflip = true$ then
            UpdateConfig($jpair, 1 - 2 * ispin, oldsite, dpbd2$)
        else
            UpdateConfig($ipair, 2 * ispin - 1, newsite, dpbd$)
        end if
    else                              ▷ Restore the configuration: Move is not accepted
        Copyconfig()
    end if
end procedure
```

Algorithm 6 Energy of a configuration

```
procedure EnergyofConfig( )
    saved = config()                                   ▷ Make a copy of the configuration
    tempconf = config()                                ▷ Make a temporary configuration to
                                  calculate dpdbd and update the configuration
    for $ispin \leftarrow 0, 2$ do              ▷ Calculate the Kinetic energy part of the Hamiltonian
        $spin \leftarrow 2 * ispin - 1$
        for $ipair \leftarrow 0, Npairs$ do
            $isite \leftarrow plist(ipair * 2 + ispin)$
            for $jn \leftarrow 0, nneibs$ do
                $jsite \leftarrow neiblist(jn + nneibs * isite)$
                if $latocc(jsite) = HL$ then
                    $dpbd \leftarrow \mathrm{DetPByDet}(ipair, spin, jsite)$
                    $enekloc \leftarrow enekloc + thop(ishell(jn)) * real(dpbd)$
                    $spkeloc(ispin) \leftarrow spkeloc(ispin) + thop(ishell(jn)) * real(dpbd)$
                end if
            end for
        end for
    end for
                                     ▷ Compute the Exchange term of the Hamiltonian
    $otheloc \leftarrow 0$                           ▷ To hold the energy from the exchange term
    for $inn \leftarrow 0, nearnsets$ do
        $isite \leftarrow nearnp(inn * 2)$; $jsite \leftarrow nearnp(inn * 2 + 1)$
        $ispin \leftarrow latocc(isite)$; $jspin \leftarrow latocc(jsite)$
        $ipair \leftarrow whichpair(isite * 2 + (1 + ispin)/2)$
        $jpair \leftarrow whichpair(jsite * 2 + (1 + jspin)/2)$
        if $ispin * jspin < 0$ then
            $tempconf \leftarrow \mathrm{CopyConfig}()$
            $dpbd \leftarrow \mathrm{DePByDet}(ipair, ispin, jsite)$; UpdateConfig()
            $dpbd1 \leftarrow \mathrm{DetPByDet}(jpair, jspin, isite)$
            $otheloc = otheloc - real(dpbd * dpbd1) + 1.0$
        end if
    end for
    $otheloc \leftarrow (Jij * otheloc)/(2)$          ▷ Jij represents the Antiferromagnetic exchange
    $enektot \leftarrow enekloc$
    $othetot \leftarrow otheloc$
    $energy \leftarrow (enektot + othetot)/Nsites$
    $kinenergy \leftarrow (spkeloc(0) + spkeloc(1))/Nsites$
    $othenergy \leftarrow othetot$
    return $energy$
end procedure
```

The programming model follows the object-oriented approach and the major data structures, such as Ψ^{-1} and pairfunction are members of the classes defined in Table 4.2. By varying the input parameters such as hole doping and the lattice size, data structures such as Ψ^{-1} and pairfunction become the point of interest due to increasing memory consumption. Table 4.1 shows the memory consumption for different lattice models.

Table 4.1: Data Structures and Memory requirements

Data structure	**Formula**	**Data Type**	**L = 5**	**L = 9**	**L = 15**
Nsites(N_S)	$L^2 + 1$	**int**	26	82	226
Nelecs(N_e)	$Nsites * (1.E0 - x)$ where x is hole doping	**int**	24	74	204
Npairs(N_P)	$Nelecs/2$	**int**	12	37	102
PsiInv	$(Npairs^2)$	**double**	1.152 kB	10.952 kB	83.232 kB
Pairfunction	$(Nsites^2)$	**double**	5.408 kB	53.792 kB	408.608 kB
Plist	$(2 * Npairs)$	**int**	96 B	296 B	816 B

Table 4.2: Classes used in the Implementation of VMC

S.No	**Class**	**Functionality**
1	Sqlat	Holds Lattice parameters such as Nsites, Nelecs, Npairs and hopping, etc
2	Config	Contains the energy arrays, and functions for calculating the Ratio of Determinants and Update of Ψ^{-1}
3	Wavefunction	Contains the variational parameters required to generate the pairfunction or wavefunction
4	Montecarlo	Contains the lattice object, pairfunction and randomseeds for a corresponding Markov chain

4.1.1 Structure of the CPU code

The implementation uses an object-oriented approach for the individual functions mentioned in the previous section. Each class is used to create an object and the final executable is obtained by compiling and linking the individual objects. Table 4.2 describes the classes and the corresponding functionality.

4.2 Calculation of the Ratio of Determinants using SMH Formula

The SMH formula, as explained in Section 1.2.2, is used to calculate the ratio of determinant of the configurations. This is implemented using a function called DetPByDet.

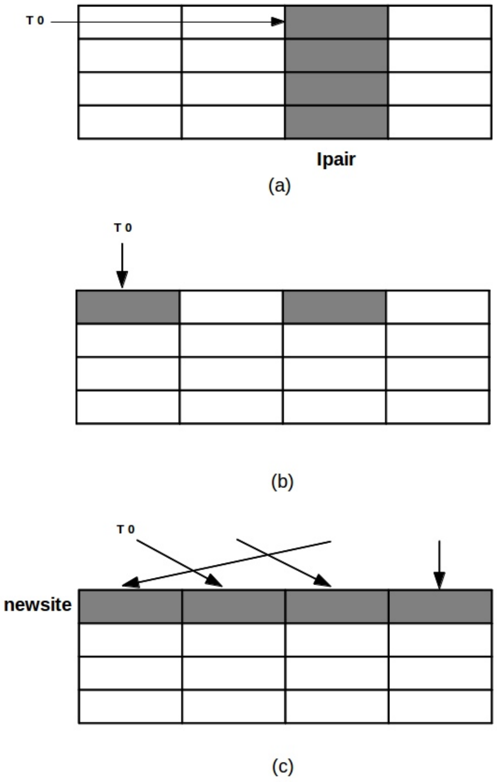


Figure 4.1: Sequential Memory Access Pattern for the up spin: (a) Ψ^{-1}, (b) Plist, (c) Pairfunction

Figure 4.1 shows the access pattern for the up spin. A site is picked from an electron list and referred to as an oldsite. This site corresponds to the column of pairfunction. A newsite given by an input parameter corresponds to the row of pairfunction. The *Ipair* represents the electron pair and corresponds to the column of Ψ^{-1}. Thus, a dot product of a row of pairfunction and a column of Ψ^{-1} is calculated (dpbd). There will be N_P cache misses for the Ψ^{-1} array and one cache miss, to load the first pairfunction element on the **L1 cache**. For the given lattice size, L, there will be a total of N_P fused multiply-add floating point operations (FLOPS) to compute a dot-product.

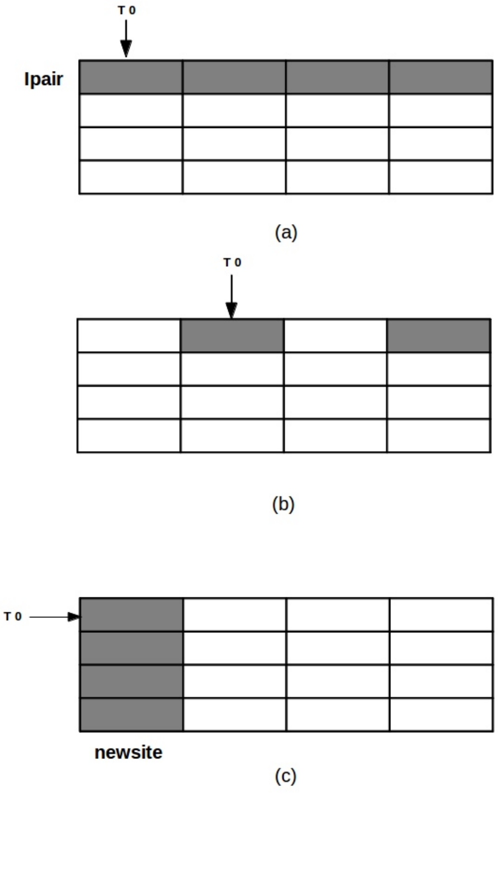


Figure 4.2: Sequential Memory Access Pattern for the down spin: (a) Ψ^{-1}, (b) Plist, (c) Pairfunction

Figure 4.2 shows the access pattern for the down spin. In this case, the access pattern is reversed, where a column of the pairfunction and a row of Ψ^{-1} are accessed. There will be N_S of cache misses for the pairfunction array and one cache miss for the Ψ^{-1}. From section 1.2.2, the ratio of determinants can be calculated in the order of $O(N_P)$ using SMH formula [5].

4.3 Update of a configuration

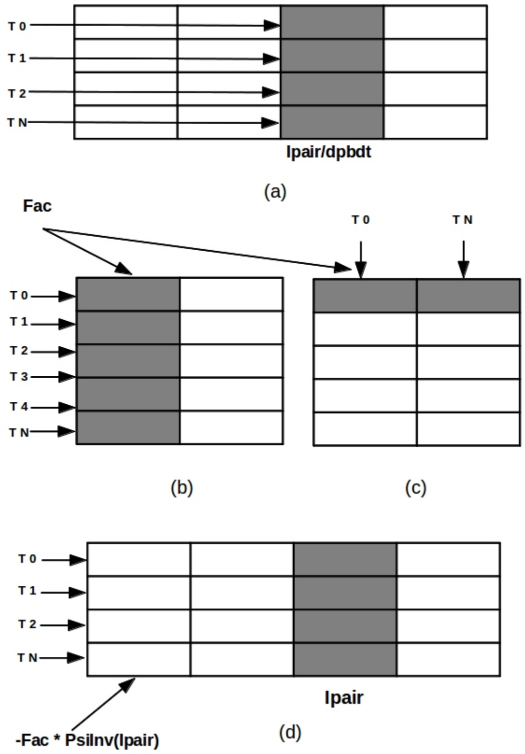


Figure 4.3: Update of a configuration for the up spin

Figure 4.3 shows the update of a configuration for the up spin of an electron. The $Ipair^{th}$ column of Ψ^{-1} is first updated using dpbd.

The remaining columns of Ψ^{-1} are updated using the dot product between Ψ^{-1} and pairfunction. The update of Ψ^{-1} is of order $O(N_P^2)$.

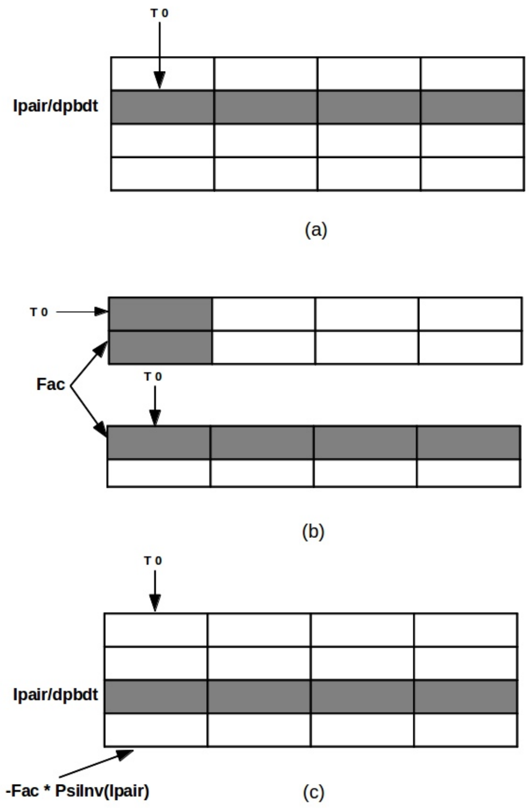


Figure 4.4: Update of a configuration for the down spin

For the case of a down spin, the access pattern is reversed, where an electron pair represents the $Ipair^{th}$ row in Ψ^{-1}. The remaining rows are updated by using the dot product between Ψ^{-1} and pairfunction. Figure 4.4 shows the update of a configuration for the down spin of an electron.

4.4 Algorithm Analysis

The execution time of the individual functions is dependent on parameters such as, N_P, N_S, *neqlsweeps*, *npsweep* and *navesweeps*. Table 4.3 gives the Floating point operations per second (FLOPS) calculation for different functions of the VMC method. The FLOPS are calculated based on a *speculation* that a spinflip will occur 50% of the time in both the Equilibration and Accumulation stage. Thus, the FLOPS are an estimate to different lattice models based on N_P and may vary depending on the occurrence of a spinflip.

Table 4.3: FLOPS calculation for the VMC method

Function	**FLOPS**	**Operation**
DetPByDet	$Npairs * MUL$	Either case of spinflip
UpdateConfig	$Npairs * DIV$	For $Ipair^{th}$ row or col of Ψ^{-1}
	$+Npairs^2 * (MUL + ADD)$	For dot product calculation
	$+Npairs^2 * (MUL + SUB)$	For updating other rows or cols of Ψ^{-1}
CopyConfig	NIL	NIL
Energy of a Configuration	$2 * Npairs * Nneibs$ $*(DetPByDet + MUL + ADD)$	Kinetic energy calculation
	$+Nearnsets*$(DetPByDet) $+Nearnsets*$Updateconfig $+Nearnsets*$(DetPByDet)	
	$+Nearnsets * (ADD + SUB)$	Accumulation

4.5 Cache Behavior

The implementation utilizes a single core of the CPU and further testing and optimization of the code are decided upon the cache behavior. We consider the Intel(R)Core(TM)i7-2600 CPU architecture for determing the cache behavior.

From [10], the cache hierarchy is given by: 32 kB data + 32 kB instruction **L1 cache**, 256 kB **L2 cache** per core.

4.5.1 Function DetPByDet

Referring to Section 4.1.1, one dot product requires a total of N_P fused multiply-add floating-point operations. Since a cache line is 64 B wide, we find that for lattice size of $L = 5$, a row of pairfunction or a column of Ψ^{-1} does not fit within a cache line. There will be only one miss penalty to load the first element for pairfunction. There will be N_P cache misses for Ψ^{-1} since we access a column. The penalties are reversed for the down spin, where there will be N_S of cache misses for the pairfunction and one miss penalty for Ψ^{-1}.

4.5.2 Function UpdateConfig

Referring to section 4.3, the update of Ψ^{-1} is of order $O(N_P^2)$. For the case of an up spin, there will be N_P cache misses for every element of the $Ipair^{th}$ of Ψ^{-1}. To perform a dot product, the penalty levels will be the same as mentioned in 4.5.1. To update the remaining columns there will be N_P cache misses for every element per column. The penalties are reversed for the down spin, where there will be one cache miss per row of Ψ^{-1}, while N_S of cache misses per element of pairfunction. The pairfunction access is permuted in either case of the spin and hence the latency will increase if the data falls outside the cache.

4.5.3 Function Pairfunction

For the case of an up spin of an electron, a row of pairfunction is accessed. Though the access pattern within a row is permuted, the data is still cached and will have a lower latency. The case is reversed for the down spin, where there will be several cache misses as explained in Section 4.1.1. From Figures 4.3, 4.4, the pairfunction will suffer a mimimum cache miss for the up spin, while there will be several cache misses for the down spin. All these cache misses are viewed from the **L1 cache** level. Since, the pairfunction is accessed more than 10^4 in both the Equilibration and Accumulation stage and the size for any lattice model is less than that of **L2 cache**, we can speculate the hardware to move it to the **L2 cache**.

4.6 Results

The VMC method has been tested on Intel(R)Core(TM)i7-2600 CPU @ 3.40 GHz. The method performs one MC and from the timing results, we find the *hot spots* for futher parallelism and optimization using GPU. More work on porting VMC to GPU using CUDA are discussed in Chapter 5. Tables 4.4-4.6 give the timing results for individual functions and for the overall VMC method for $L = 5, 9, 15$.

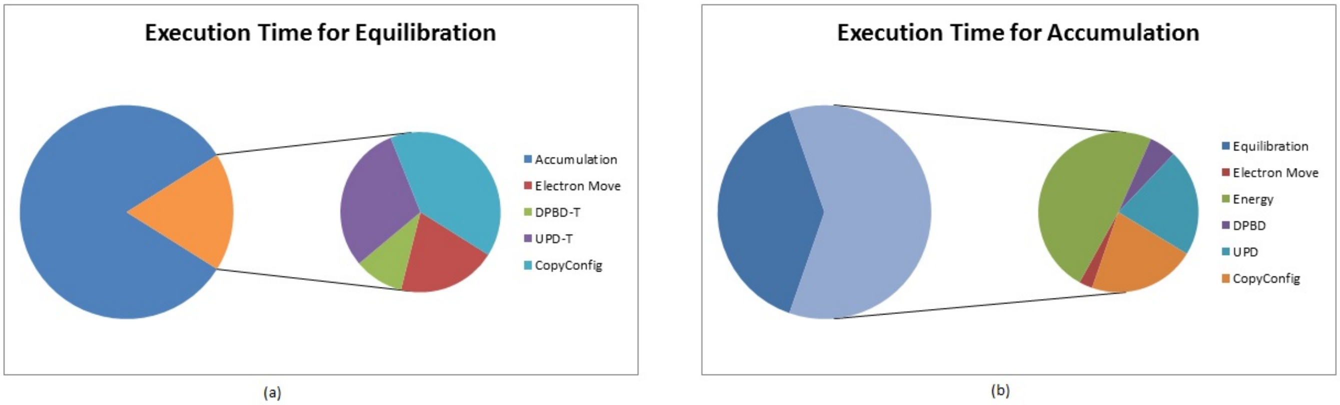


Figure 4.5: Pie chart for $L = 5$:(a) Split of Equilibration, (b) Split of Accumulation

Table 4.4: Execution time for $L = 5$

Lattice Size	**Function**	**Execution Time(s)**
$L = 5$	**Equilibration**	
	1.Electron Move	0.02
	2.Spinflip = true:	
	DetPbyDet	0.01
	UpdateConfig	0.03
	3.Spinflip = false:	
	DetPbyDet	0.00
	UpdateConfig	0.00
	4.Copyconfig	0.04
	Move rejected	0.02
	5.**Total**	**0.24**
	Accumulation	
	1.Electron Move	0.01
	2.Spinflip = true:	
	DetPbyDet	0.01
	UpdateConfig	0.07
	3.Spinflip = false:	
	DetPbyDet	0.01
	UpdateConfig	0.02
	4.**Energy calculation**	
	DetPbyDet	0.02
	UpdateConfig	0.08
	Total	**0.18**
	5.Copyconfig	0.08
	Move rejected	0.03
	6.**Total**	**0.46**
	Total VMC	**0.70**

Table 4.5: Execution time for $L = 9$

Lattice Size	**Function**	**Execution Time(s)**
$L = 9$	**Equilibration**	
	1.Electron Move	0.01
	2.Spinflip = true:	
	DetPbyDet	0.11
	UpdateConfig	1.32
	3.Spinflip = false:	
	DetPbyDet	0.02
	UpdateConfig	0.04
	4.Copyconfig	0.74
	Move rejected	0.68
	5.**Total**	**3.04**
	Accumulation	
	1.Electron Move	0.07
	2.Spinflip = true:	
	DetPbyDet	0.07
	UpdateConfig	1.44
	3.Spinflip = false:	
	DetPbyDet	0.00
	UpdateConfig	0.09
	4.**Energy calculation**	
	DetPbyDet	0.17
	UpdateConfig	1.85
	Total	**2.91**
	5.Copyconfig	0.62
	Move rejected	0.64
	6.**Total**	**6.03**
	Total VMC	**9.07**

Table 4.6: Execution time for $L = 15$

Lattice Size	**Function**	**Execution Time(s)**
$L = 15$	**Equilibration**	
	1.Electron Move	0.19
	2.Spinflip = true:	
	DetPbyDet	0.51
	UpdateConfig	25.15
	3.Spinflip = false:	
	DetPbyDet	0.07
	UpdateConfig	0.96
	4.Copyconfig	12.29
	Move rejected	11.99
	5.**Total**	**51.74**
	Accumulation	
	1.Electron Move	0.18
	2.Spinflip = true:	
	DetPbyDet	0.44
	UpdateConfig	25.17
	3.Spinflip = false:	
	DetPbyDet	0.03
	UpdateConfig	0.98
	4.**Energy calculation**	
	DetPbyDet	0.80
	UpdateConfig	36.50
	Total	**52.67**
	5.Copyconfig	12.30
	Move rejected	11.61
	6.**Total**	**104.05**
	Total VMC	**156.02**

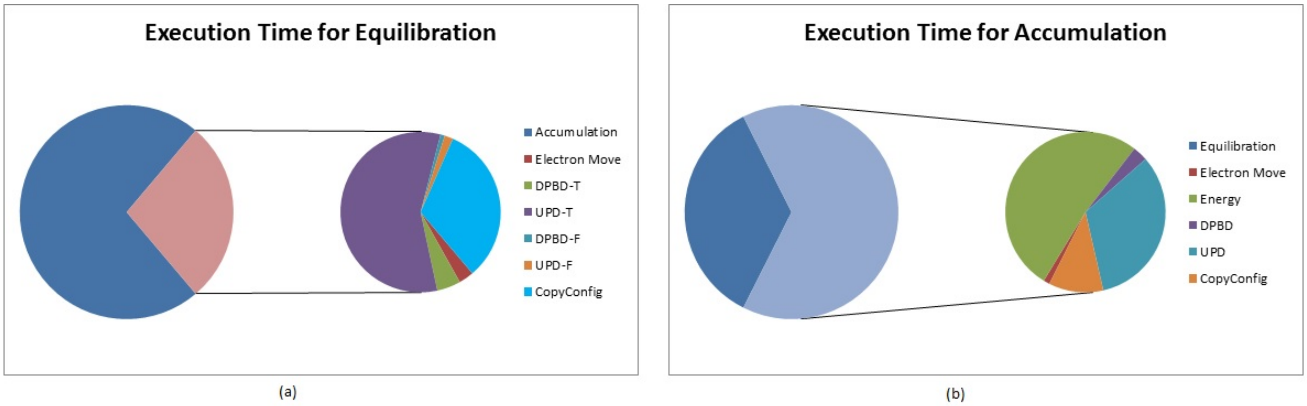


Figure 4.6: Pie chart for $L = 9$:(a) Split of Equilibration, (b) Split of Accumulation

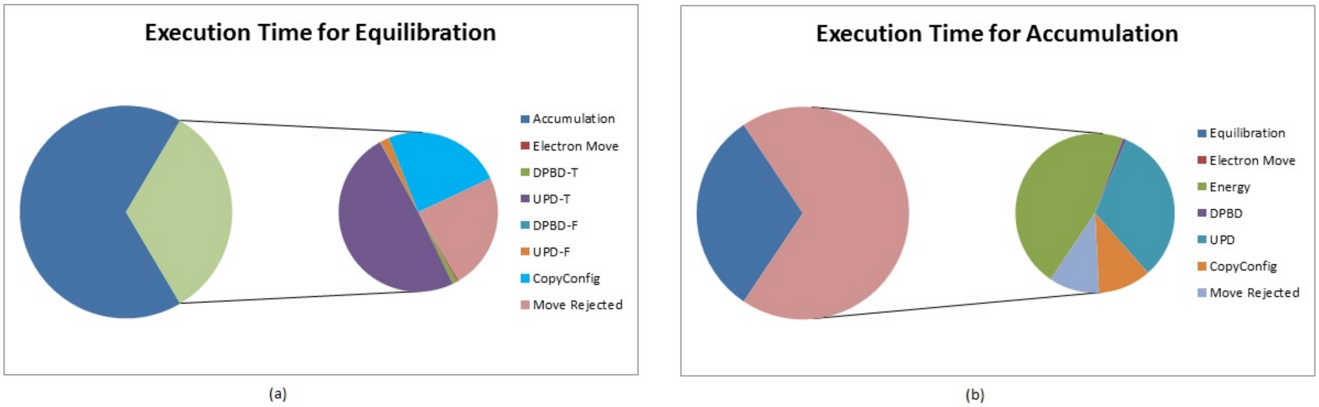


Figure 4.7: Pie chart for $L = 15$:(a) Split of Equilibration, (b) Split of Accumulation

Figure 4.5 gives the execution time for $L = 5$ using a pie-chart. The upper half represents the execution time taken by the accumulation procedure. From the chart, we determine that accumulation is the most time consuming procedure. On further profiling, we found that the execution time for calculating the energy was higher than other functions. Further breakdown revealed the most time consuming and computationally intensive part to be the update of a configuration. Chapter 5 describes the parallelization efforts on GPU for this procedure. On the lower half of the figure, variables such as, DPBD-T and UPD-T represents the execution time taken for the case when the spinflip is true.

Figures 4.6, 4.7 gives the execution time for $L = 9, 15$. From the charts, we find that the contribution is more when the spinflip is true. Variables DPBD-F and UPD-F represents the contribution when there is no spin-flip. Based on these charts, we claim that the speedup of the application will be enhanced, if the individual functions are parallelized and optimized for effective memory access.

4.7 Multiple MCs and the system behavior

The sequential code can be expanded to a MPI-version performing N MCs in parallel on N CPU cores. Issues such as the total execution time, parallel and sequential regions within the equilibration and accumulation stages need to addressed. Let $T(N)$ be the total execution time to complete N MCs in parallel. Let T_e be the equilibration time on a serial implementation and let T_a be the accumulation time on a serial implementation.

If we execute N MCs on N CPUs in parallel, the total time can be theoretically, $T(N) = (T_e + T_a)(N)^{-1}$.

Practically, we cannot expect this execution time due to the follwing factors.

- Large equilibration time which will be constant for N MCs.

- Serial regions within a MC such as: Electron move, DetPbyDet and update of a configuration.

    1. For every MC we start with a random configuration and wait until the system equilibrates and start accumulating the energy.
    2. Since, certain MCs can finish well ahead, there will be a race around condition in the energy accumulation and average calculation.

- Inter-node communication delays if the number of MCs is of order 100 or more.

Hence, these factors will affect the parallelism approach and we cannot expect a direct speedup of over N for this algorithm on a MPI model.

Chapter 5
Porting VMC to CUDA

5.1 GPU Acceleration of the VMC Method

In the previous chapter, we explained the CPU implementation of the VMC method and determined the hot spots for further parallelization. From Section 4.6, we found that the CPU implementation is time consuming for lattice models $L = 9, 15$. Functions such as the ratio of determinants, update and energy of a configuration are computationally intensive and exhibit data-level parallelism, thereby becoming the best candidates for parallelization. Hence, in this chapter, we discuss the several strategies adopted to parallelize the VMC method for multiple MCs on a NVIDIA GPU using CUDA.

Figure 5.1 represents the workflow for the GPU implementation. Each MC is independent and can be executed in parallel on a GPU. Every chain has its own copy of a configuration object with members such as, Ψ^{-1}, neighbor list, energy and whichpair. The pairfunction and lattice object are shared by all MCs. A block of CUDA threads will handle a MC and hence we have N parallel CUDA blocks executing in parallel on a NVIDIA GPU. At the end of each MC, the energy per site is accumulated and the result is written to the energy array corresponding to that chain. The results are copied back to the CPU, where the mean energy and standard deviation are calculated. The groundstate, $|\Psi_G\rangle$ is then optimized with the variational parameters.

5.2 Naïve Implementation

The CUDA implementation described here is based upon a preliminary CUDA port, which will be called the *naïve implementation*, performed by [17], an intern student who worked for the La-Sigma research group at LSU. In this implementation, each MC is han-

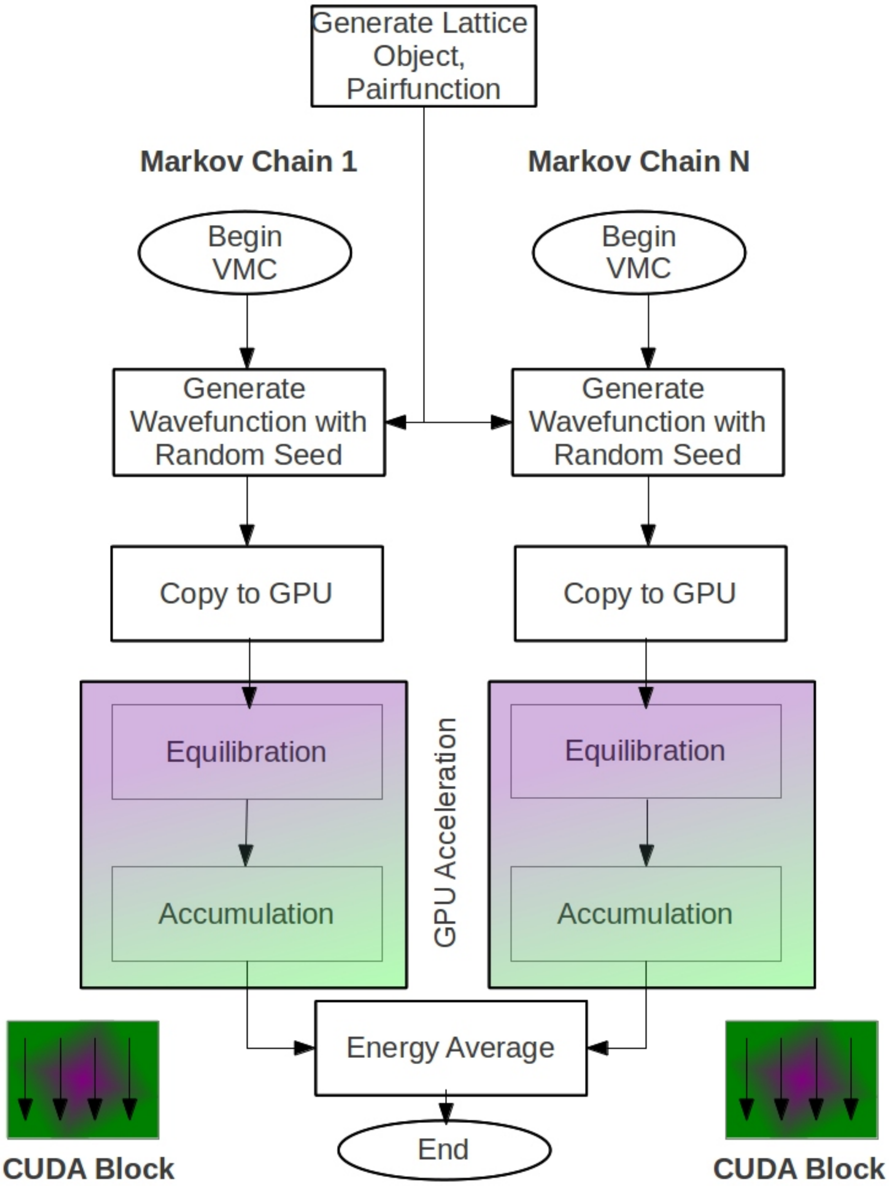


Figure 5.1: VMC Workflow for Multiple Markov Chains (MCs)

dled by a single CUDA thread and the total number of CUDA blocks were equal to the number of MCs. This does not utilize the capability of the massively parallel GPU and hence no improvement was found over the CPU code. With only one thread per block, the code cannot exceed using $\frac{1}{32}$ of the GPU‘s computing potential per warp (group of 32 threads). The other factors contributing to the slowdown being: Higher memory access latency due to global memory read/write(s) and the lack of usage of shared memory and constant memory, and unoptimized use of the L1 and L2 caches. Hence, in the following sections, we describe in detail, the parallelization and effective memory utilization efforts done to improve this naïve implementation.

5.3 Execution configuration

The number of threads required to obtain occupancy and SM utilization was the deciding factor for our implementation. The threads per MC depends upon the N_P of the given lattice.

On a Fermi device (Architecture model for a NVIDIA GPU) with compute capability 2.X [11], a SM can support 8 resident blocks of threads. However, the total number of threads per SM cannot exceed 1536 and the shared or $L1$ cache per SM is limited to 48 or 16 kB. More blocks will increase the warp occupancy per SM, while the shared memory usage shrinks to a factor of $48/8 = 6$ kB per block. Another factor affecting the performance will be the register usage per SM. On a Fermi device, the total number of registers per thread values to 63 [12]. If a thread exceeds this limit, it will result in a local memory spill, which will move the data to high latency global memory [12]. Hence, to avoid higher access latency and assign enough work for the threads, we pick the number of threads per block based on the complexity of the functions. The order of these functions are given by: DetPByDet - $O(N_P)$; Updateconfig - $O(N_P^2)$; CopyConfig - $O(2 * N_S)$. If the number of threads is set to N_P^2, there will be enough threads to cover the latency for UpdateConfig and CopyConfig. On the contrary, this will reduce the speedup due to sequential zones such as Electron move and wasteful threads in DetPByDet. If we recall from the previous chapter, the memory access pattern is distinct for either case of a spinflip of UpdateConfig. Thus, the number of threads should be in the range of ($N_P < threads/MC < N_P^2$) and rounded to the nearest multiple of a warp. We discuss more about this in Section 5.6.1.

5.4 Memory requirements and cache behaviour

Though the amount of parallelism in a MC is low, by GPU standards, as this section will show the working set, careful use of the available high-speed memory will make this code efficient.

From Section 4.1, we recall the memory requirements for different lattice models. The total device global memory on a fermi device is approximately $5 \sim 6$ GB. From the Table 4.1, we find that the total memory requirements for a lattice model of size $L = 15$ is within the GPU global memory limit. However, global memory has the highest latency compared to other memory resources on a GPU and hence we need to find effective ways to utilize the high-speed memories such as, registers, L1/shared, and constant memory [7]. In this section, we discuss about the caching behaviour on a GPU. Tables 5.1, 5.2 and 5.3 shows whether each major array used by the code fits into three different parts of the memory hierarchy for different lattice sizes.

From Table 5.1, arrays such as Ψ^{-1}, pairfunction and plist fits within the L1 cache. The repeated access of the pairfunction by different MCs will result in caching at L2 level by the hardware. Consider the lattice model, $L = 5$: From Table 5.1, we find the size of Ψ^{-1} to be 1.152 kB. The total number of threads per MC is 64. Therefore, the total number of blocks per SM will be $1536/64 = 24$. Fermi device has a limit of 8 resident blocks per

Table 5.1: Speculation of a GPU‘s cache behaviour for $L = 5$

GPU Memory	Ψ^{-1} **(double)**	Pairfunction **(double)**	Plist **(int)**
$L = 5$	1.152 KB	5.408 KB	96 B
L1 cache/ shared	Yes	Yes	Yes
L2 cache	No	Yes.Due to sharing between MC	No
Constant Memory	No	No	No

Table 5.2: Speculation of a GPU‘s cache behaviour for $L = 9$

GPU Memory	Ψ^{-1} **(double)**	Pairfunction **(double)**	Plist **(int)**
$L = 9$	10.952 KB	53.792 KB	296 B
L1 cache/ shared	Yes	No	Yes
L2 cache	No	Yes.Due to sharing between M.C	No
Constant Memory	No	No	No

SM [8]. The memory consumption per SM will be: $\Psi^{-1} = 1.152 * 8 = 9.216 * 3 = 27.648$ kB, since we have three copies of a configuration. This is still within the limit of 48 KB of L1 cache and hence will be cached by the hardware; The pairfunction will be 5.408 kB and is within the limit of 64 kB of constant memory; The plist array will be $96 * 3 = 288$ B, for three configurations and still can reside in L1 cache. All these speculations are based on the assumption that the pairfunction does not evict lines holding Ψ^{-1}. Since, there are three copies of a configuration and the cache requirements per MC is close to 27 kB, the

Table 5.3: Speculation of a GPU‘s cache behaviour for $L = 15$

GPU Memory	$\boldsymbol{\Psi^{-1}}$ **(double)**	**Pairfunction (double)**	**Plist (int)**
$L = 15$	83.232 KB	408.608 KB	816 B
L1 cache/ shared	Yes, but partly cached	No	Yes
L2 cache	Yes. Exceeds L1 cache size	Yes. Due to sharing between M.C	No
Constant Memory	No	No	No

Table 5.4: Summary of Memory prediction for $L = 5$

Lattice	**Threads per MC**	**Config**	$\boldsymbol{\Psi^{-1}}$	**Pairfunc**	**Plist**	**SM occupancy**
$L = 5$	64	Config1 Config2 Config3	L1 L1 L1	Constant-Common to all MC	L1 L1 L1	
		Total	**27.648 kB**	**5.408 kB**	**288 B**	1 MC

number of MC per SM will be limited to 1. Table 5.4 summarizes this memory prediction for $L = 5$.

For the case of $L = 15$, the number of threads per MC will be 416; the number of blocks per SM will be $1536/416 \approx 3$; and the memory consumption for $\Psi^{-1} = 83.232$ KB. Hence, it does not fit within the L1 cache and it will be cached on L2 level by the hardware. This drawback will be overcome by more threads which will hide the global memory latency due to *coalescing*.

5.5 Memory Access Pattern

In this section, we discuss about the memory access pattern and initial parallelization efforts for functions such as: DetPByDet (Ratio of determinants) and UpdateConfig (To update a configuration). Section 5.6 discusses more about the optimization strategies adopted for these functions.

5.5.1 Function DetPByDet

This function is used to calculate the dot product of Ψ^{-1} and pairfunction. We followed these steps to parallelize this function using CUDA:

- Spinflip = up

  1. The electron pair, represented by *Ipair*, will correspond to the column of Ψ^{-1}.
  2. A *newsite* is picked where the electron will be moved.
  3. From the lattice occupancy list, given by *plist*, we find the lattice site of the electron, given by *othsite*.
  4. Now, the *newsite* will correspond to the row and the *othsite* will correspond to the column of pairfunction.
  5. Thus, we perform a dot product of a column of Ψ^{-1} and a row of pairfunction using the faster VMC technique given by SMH.
  6. In this implementation, we use N_P threads to perform the dot product, whereas [17] uses one thread. Figure 5.2 explains this procedure.

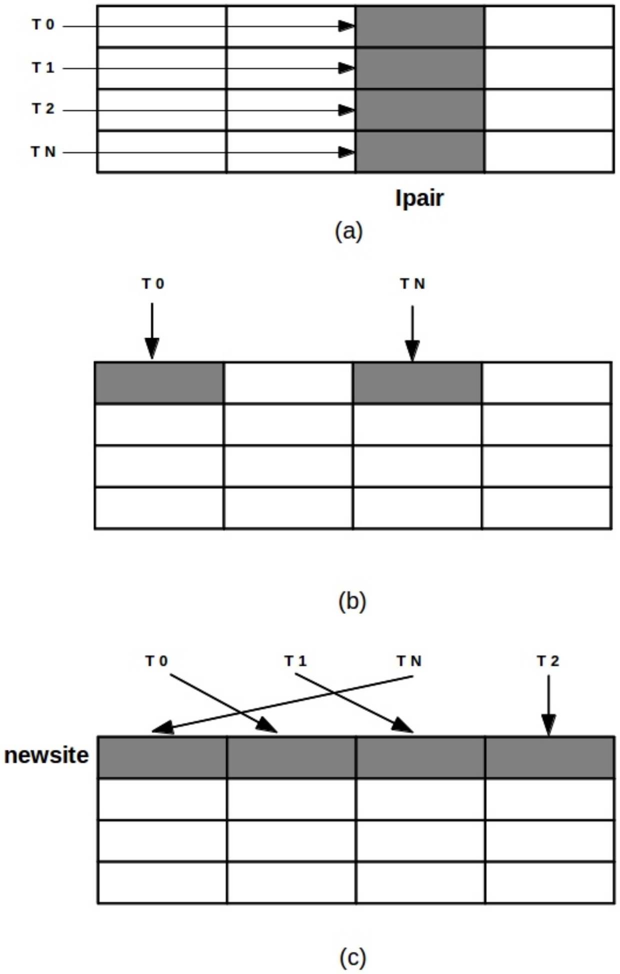
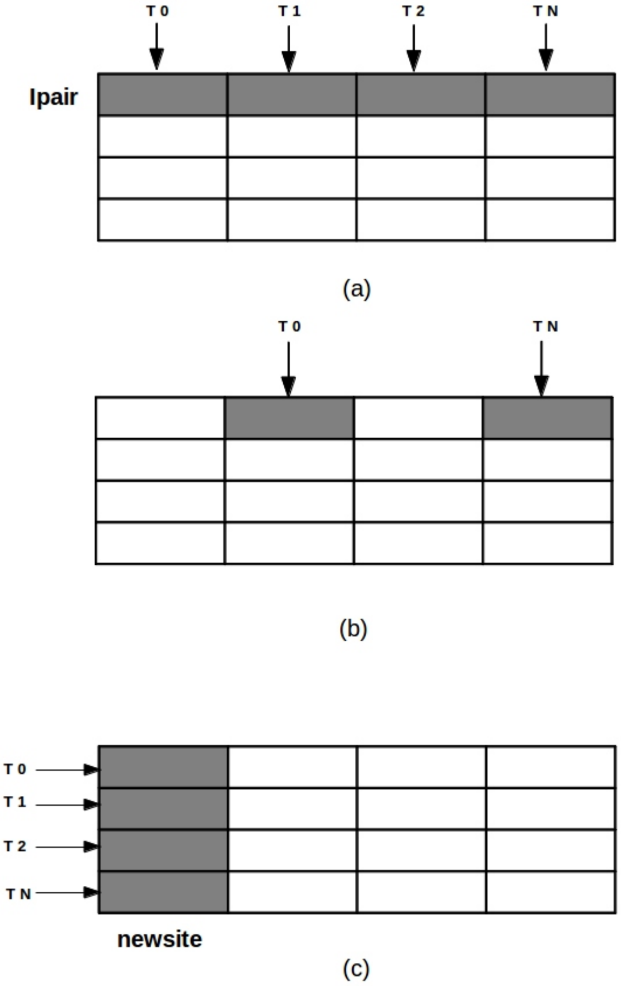


Figure 5.2: Memory Access Pattern for up spin: (a) Ψ^{-1}, (b) Picking a site from Plist, (c) Pairfunction



(a)

(b)

(c)

Figure 5.3: Memory Access Pattern for down spin: (a) Ψ^{-1}, (b) Picking a site from Plist, (c) Pairfunction

- Spinflip = down

1. The electron pair, represented by *Ipair*, will correspond to the row of Ψ^{-1}.
2. A *newsite* is picked where the electron will be moved.
3. From the lattice occupancy list, given by *plist*, we find the lattice site of this electron, given by *othsite*.
4. Now the *newsite* will correspond to the column and the *othsite* will correspond to the row of pairfunction.
5. Thus, we perform a dot product of a row of Ψ^{-1} and a column of pairfunction, thereby summing the contribution using the faster VMC technique given by SMH.
6. In this implementation, we use N_P threads to perform the dot product, whereas [17] uses one thread. Figure 5.3 explains this procedure.

5.5.2 Function UpdateConfig

- Spinflip = up

1. The electron pair, represented by *Ipair*, will correspond to a column of Ψ^{-1}.
2. The $Ipair^{th}$ column is updated using the dpbd from function DetPByDet.
3. Now, a dot product is computed between every column of Ψ^{-1} and a row of pairfunction.
4. The remaining columns of Ψ^{-1} are now updated with this dot product.
5. This implementation utilizes N_P threads for updating the Ψ^{-1} and N threads-per-col to perform the dot product. Figure 5.4 describes this procedure.

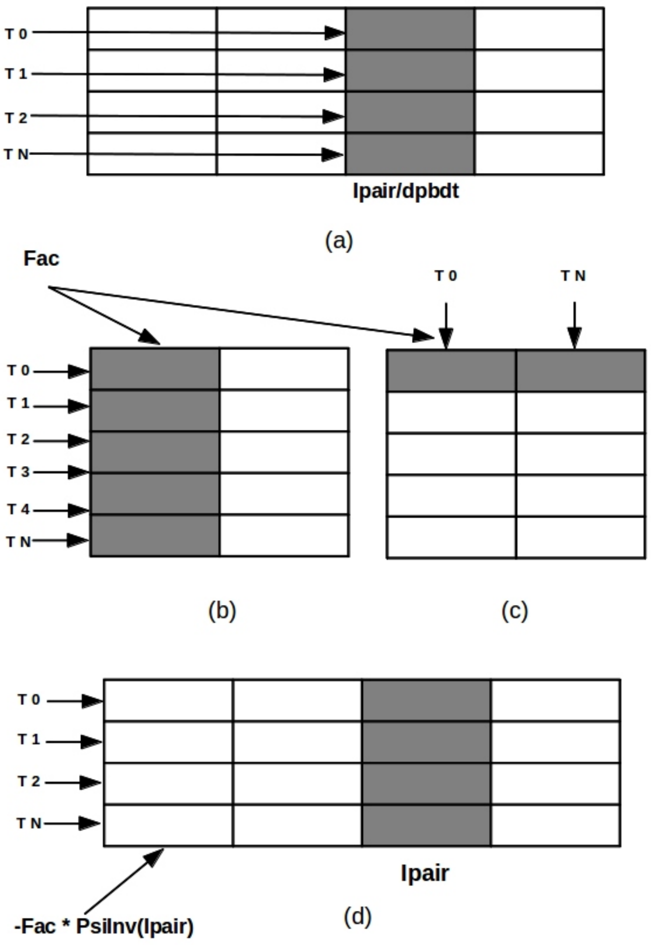


Figure 5.4: Memory Access Pattern for up spin: (a) Ψ^{-1}, (b) Picking a site from Plist, (c) Pairfunction

- Spinflip = down

1. The electron pair, represented by *Ipair*, will correspond to a row of Ψ^{-1}.
2. The $Ipair^{th}$ row is updated using the dpbd from function DetPByDet.
3. Now, a dot product is computed between every row of Ψ^{-1} and a column of pairfunction.
4. The remaining rows of Ψ^{-1} are now updated with this dot product.
5. This implementation utilizes N_P threads for updating the Ψ^{-1} and N threads-per-col to perform the dot product. Figure 5.5 describes this procedure.

5.6 Optimization efforts

In the previous Section, we mentioned the preliminary parallelization of functions using CUDA. In this section, we discuss the various memory optimization strategies adopted for these functions.

5.6.1 Thread selection in depth

Functions such as Updateconfig and Copyconfig have loop bounds on the order of $O(N_P^2)$ and $O(2 * N_S)$. An optimal thread selection range for a MC will be within ($N_P \leq blocksize \leq N_P^2$). Functions such as updateconfig have different access patterns for either case of a spinflip. Hence, we set a parameter called, *threads-per-col*, to determine the number of threads per CUDA block. The number of threads per block is given by: *Blocksize* = Round(*threads-per-col**N_P,nearest multiple of warp). Table 5.5 mentions the threads per block for different lattice models.

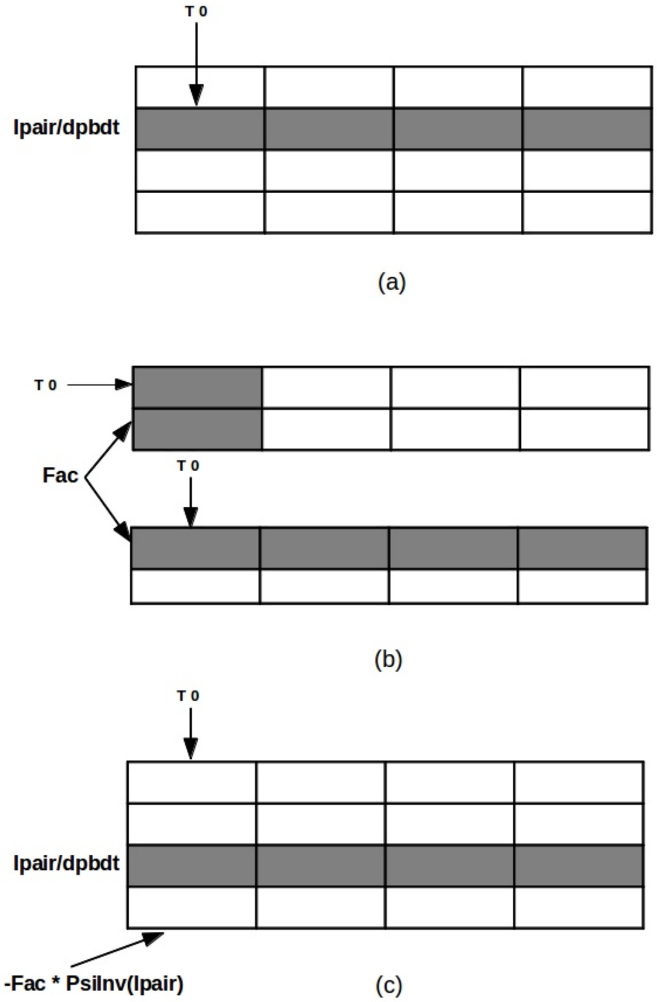


Figure 5.5: Memory Access Pattern for down spin: (a) Ψ^{-1}, (b) Picking a site from Plist, (c) Pairfunction

Table 5.5: Threads per MC and Number of MCs per SM

Lattice	**Npairs**	**Threads-per-col**	**No.threads per MC**	**No.resident MCs per SM**
$L = 5$	12	4	64	24
$L = 9$	37	4	160	9
$L = 15$	102	4	416	3

5.6.2 Elimination of redundant copies of a configuration

We predicted the memory requirements and the SM occupancy for $L = 5$ on Section 5.4. The prediction was done with three copies of a configuration per MC. In this section, we predict the SM occupancy by eliminating the redundant copies of a configuration.

Table 5.6 calculates the total memory requirements for $L = 5$ with a single configuration. The SM occupancy (Number of MCs) is given by, $48/9.312 = 5.15 \approx 5$.

Table 5.6: Occupancy calculation for $L = 5$

Lattice	**Threads per MC**	**Config**	$\mathbf{\Psi^{-1}}$	**Pairfunc**	**Plist**	**SM occupancy**
$L = 5$	64	Config1 **Total**	L1 **9.216 kB**	Constant Memory **5.408 kB**	L1 **96 B**	 5 MCs

5.6.3 Optimized memory access pattern

In the previous section, we explained the parallelization efforts for the device functions. For the function Detpbydet, the dot product can be accumulated and reduced in a shared memory. This will result in faster reduction compared to the reduction done on a global memory. For the function Updateconfig, the entire Ψ^{-1} array is accessed and updated based on the spinflip. Caching effects can be explored if this function is customized with a blocking factor for the L1 cache.

- **Function DetPByDet**

  A total of N_P threads work on this function. The dot product of each element of Ψ^{-1} and pairfunction is stored in a register for either case of a spinflip. The dot product of all elements are accumulated and reduced in a shared memory. The pairfunction has a permuted access and for the case of a down spinflip, a transposed version of the pairfunction is used. This will result lower cache misses and caching of the pairfunction at L1 level. Figure 5.6 and 5.7 explains the optimization techniques.

- **Function UpdateConfig**

  For lattice models, $L = 5, 9$, the Ψ^{-1} fits within the L1 cache and can be updated within this memory level. For $L = 15$, it does not fit within this cache and hence we need to make use of blocking factor. The following steps illustrate the optimization techniques for this function.

  - Spinflip = up

    1. Update the $Ipair^{th}$ column of Ψ^{-1} with the dpbd parameter. Here, we use N_P threads to update the column.
    2. We use the threads-per-col (T_p) parameter to set the row and column index.
    3. *col = threadIdx.x / threads-per-col*; *row = threadIdx.x % threads-per-col.*
    4. Now, these threads (4 as per the code) will work on a column of Ψ^{-1} and a row of pairfunction to compute the dot product.
    5. Perform a reduction in shared memory to compute the dot product. The dot product is then used to update the remaining columns of Ψ^{-1}. Figure 5.8 explains these optimization techniques.

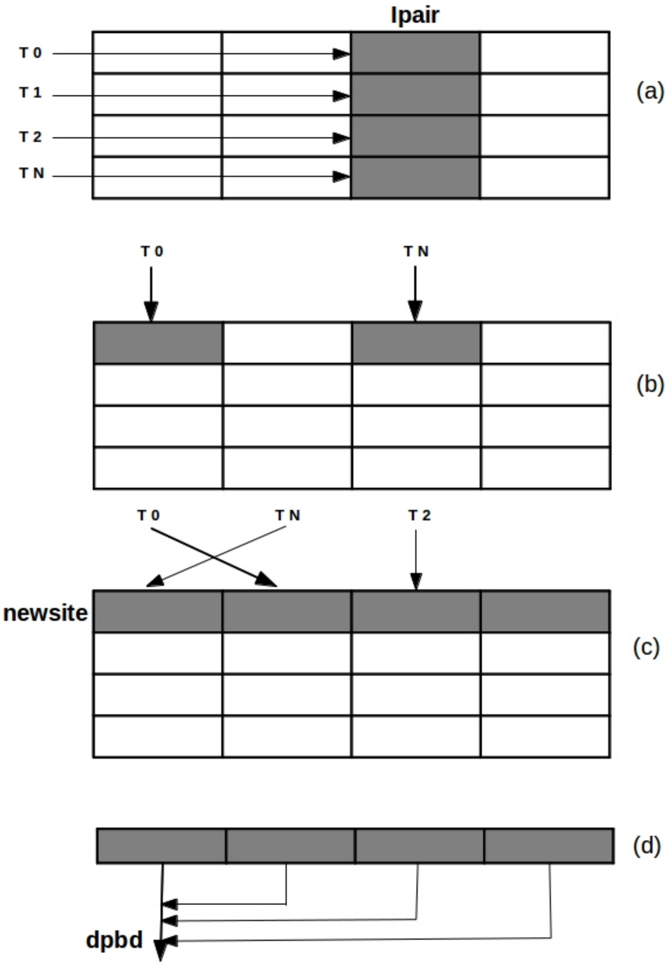


Figure 5.6: Optimization for up spin: (a) Ψ^{-1}, (b) Plist, (c) Pairfunction, (d) Reduction on a shared memory.

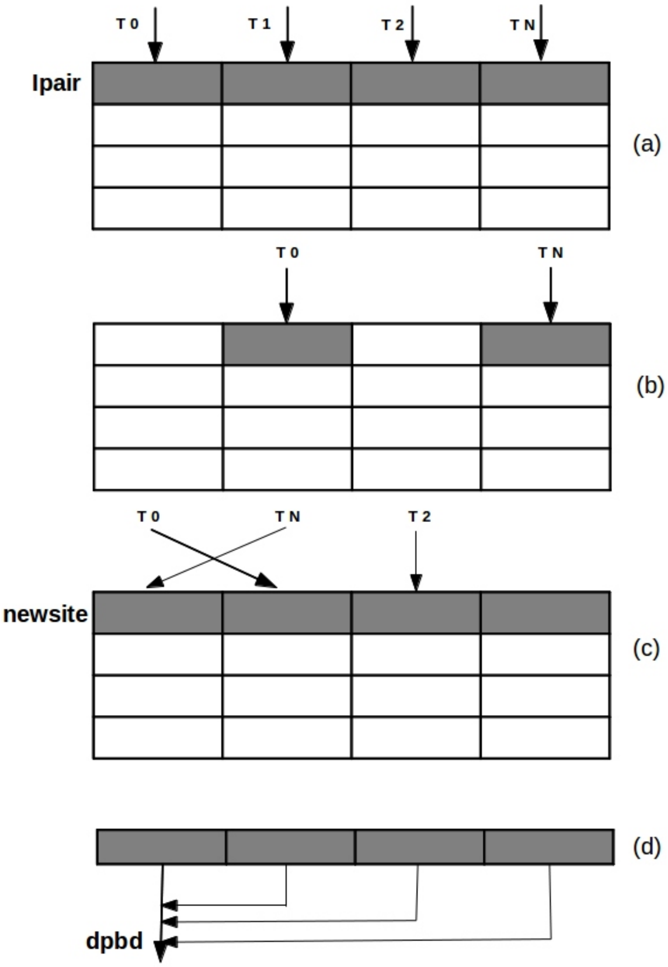


Figure 5.7: Optimization for down spin: (a) Ψ^{-1}, (b) Plist, (c) Transposed Pairfunction, (d) Reduction on a shared memory.

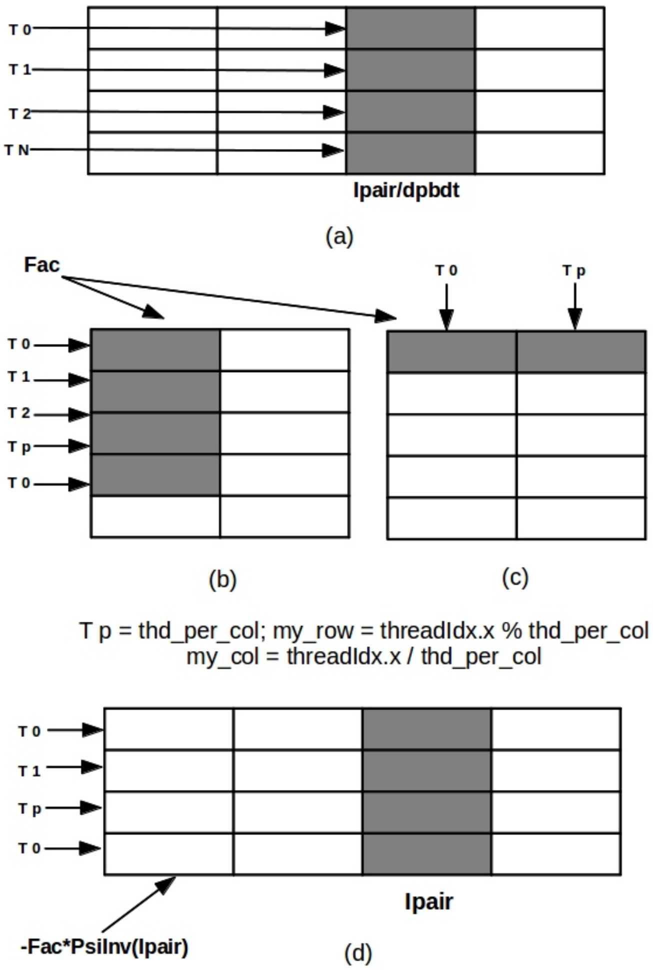


Figure 5.8: Optimization for up spin: (a) Ψ^{-1} - update of $Ipair^{th}$ column, (b) Ψ^{-1} with threads-per-col, T_P, (c) Pairfunction, (d) Ψ^{-1} - update of other columns.

- Spinflip = down

  1. We use the blocking factor, *bfactor* to set the row and column index.
  2. *row = threadIdx.x / bfactor*; *col = threadIdx.x % bfactor*.
  3. Set the number of *rows per chunk = blocksize/bfactor*.
  4. Update the $Ipair^{th}$ row of Ψ^{-1} with the dpbd parameter. N_P threads work on a row.
  5. Use the transposed pairfunction to compute the dot product. Perform a reduction in shared memory to compute the dot product.
  6. Use this dot product to update the other columns in Ψ^{-1}. Figure 5.9 shows the optimization techniques.

- **Drawbacks in the Optimization**

  In function DetPByDet, we access only a row or a column of Ψ^{-1} and pairfunction to compute a dot product. In function UpdateConfig, we first update the $Ipair^{th}$ column or row of Ψ^{-1}, then compute the dot product and finally update the remaining columns or rows of Ψ^{-1}. Thus, to reduce the amount of work, we suggest a merger of these two functions. Instead of three function calls to determine the dpbd, given by $dpbd = dpbd1 * dpbd2$, we reduce it to a single function call by calculating the *dpbd* without updating the Ψ^{-1}.

5.7 Merger of Equilibration and Accumulation

If we notice from the Algorithms 4 and 3, the steps of Electron move, Ratio of determinants (DPBD) and Update of a Configuration (UPD) are similar, except the fact that we calculate the energy of a configuration for Accumulation. Hence, in our GPU implementation we merged the two algorithms into a single function and reduced the kernel launch overhead.

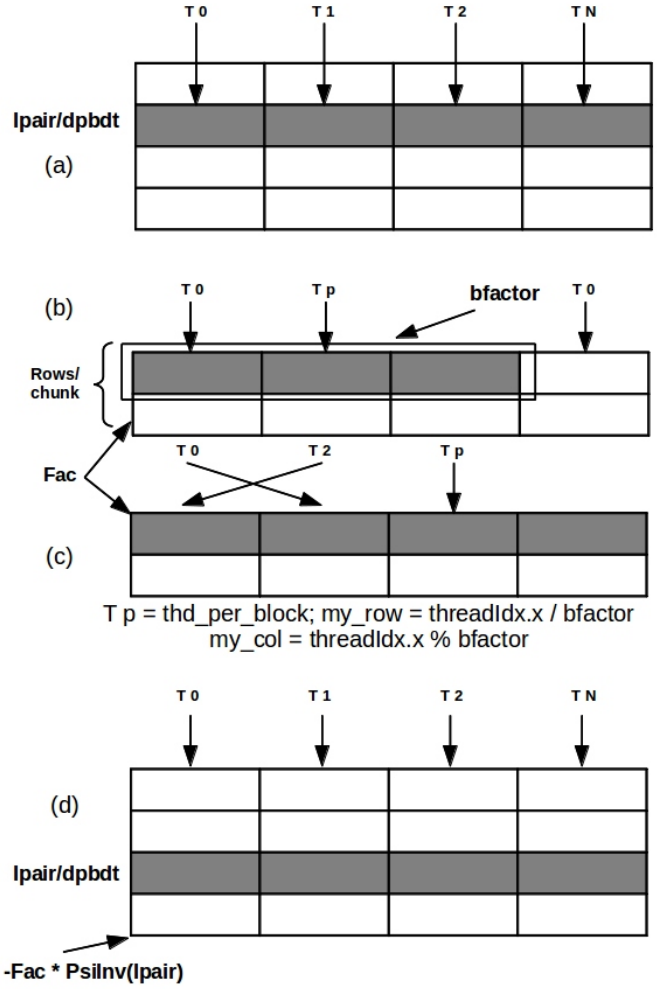


Figure 5.9: Optimization for down spin: (a) Ψ^{-1} - update of $Ipair^{th}$ row, (b) Ψ^{-1} with a blocking factor, $bfactor$ (c) Transposed Pairfunction, (d) Ψ^{-1} - update of other rows.

5.8 Streamlined function

In this function, we merged the DetPByDet and the UpdateConfig functions to calculate the dpbd(s) without updating the Ψ^{-1}. We consider either case of a spinflip and optimize accordingly.

- Spinflip = up

  1. Set the column and row index: $col = threadIdx.x \,/\, T_P$; $row = threadIdx.x \,\%\, T_P$
  2. Compute the dot product of Ψ^{-1} and pairfunction, with T_P working on a column.
  3. Store the dot product in shared memory. Perform a reduction to obtain the dpbd1.
  4. Set the column index to: $col = threadIdx.x$
  5. The dpbd is stored at the $Ipair^{th}$ index of shared memory.
  6. If the $col = Ipair$, perform the operation of a down spin from DetPByDet.
  7. If the $col \neq Ipair$, multiply the dpbd with the $Jpair^{th}$ row elements of Ψ^{-1} and subtract from the $Ipair^{th}$ column of Ψ^{-1}. Here $Jpair$ is obtained from the whichpair array (Tells which pair has an electron of spin "spin" at lattice site ilat, whichpair(ilat,spin))
  8. Multiply again with a row of transposed pairfunction. Accumulate the results in a shared memory.
  9. Perform a reduction to obtain the final dpbd: $dpbd = dpbd1 * dpbd2$. Thus, we have merged these operations: dpbd1 = DetPByDet, UpdateConfig, dpbd2 = DetPByDet into a single function without writing to Ψ^{-1}.

- Spinflip = down

    1. Set the blocking factor, $bfactor$
    2. Set the row and column index: *row = threadIdx.x / bfactor*; *col = threadIdx.x % bfactor*
    3. Set the rows-per-chunk: *rows-per-chunk = blocksize / bfactor*
    4. Now compute the dot product of every row of Ψ^{-1} with every row of transposed pairfunction. Store the dot product in a shared memory.
    5. Perform a reduction inside the shared memory to get the dpbd.
    6. Set the row index to: *row = threadIdx.x*
    7. We need to perform the operation of Spinflip = up for DetPByDet.
    8. If $row = Ipair$, obtain the product of $Ipair^{th}$ row and $Jpair^{th}$ column of Ψ^{-1} with the pairfunction.
    9. If $row \neq Ipair$, multiply the dpbd with the $Jpair^{th}$ column elements of Ψ^{-1} and subtract from the $Ipair^{th}$ row of Ψ^{-1}. Here $Jpair$ is obtained from the whichpair array (Tells which pair has an electron of spin "spin" at lattice site ilat, whichpair(ilat,spin))

- Figure 5.10 and 5.11 explain the techniques followed for either case of a spinflip.

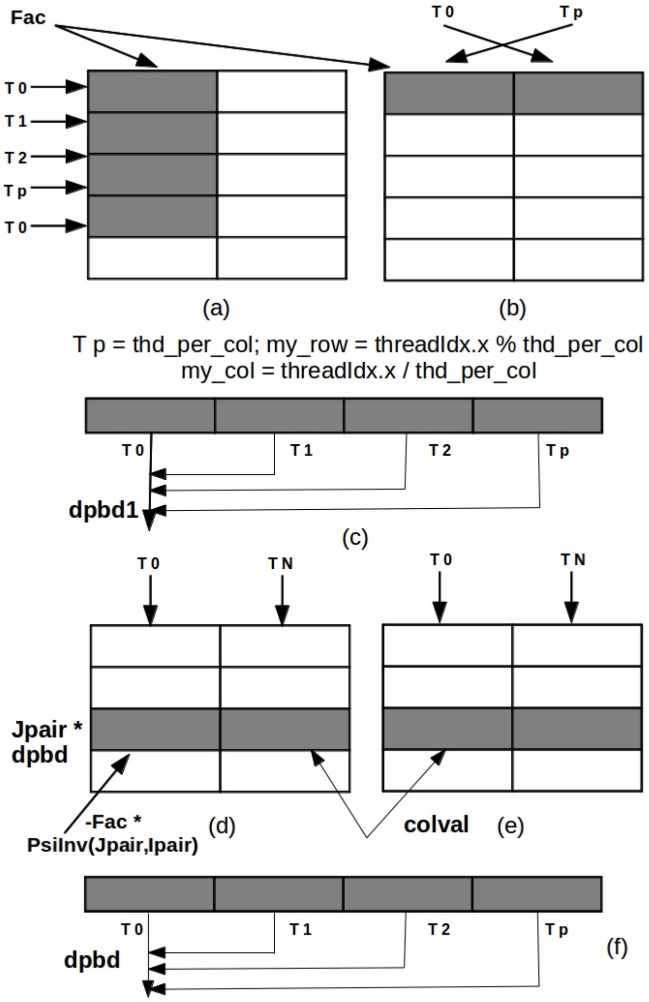


Figure 5.10: Optimization for up spin: (a) Ψ^{-1} - with T_P, (b) Pairfunction, (c) Reduction in a shared memory to obtain dpbd1, (d) Calculation of dot product using $Jpair^{th}$ row of Ψ^{-1}, (e) Calculation of dot product using Transposed pairfunction, (f) Reduction in a shared memory to obtain the final *dpbd*: $dpbd = dpbd1 * Reduction(colval)$

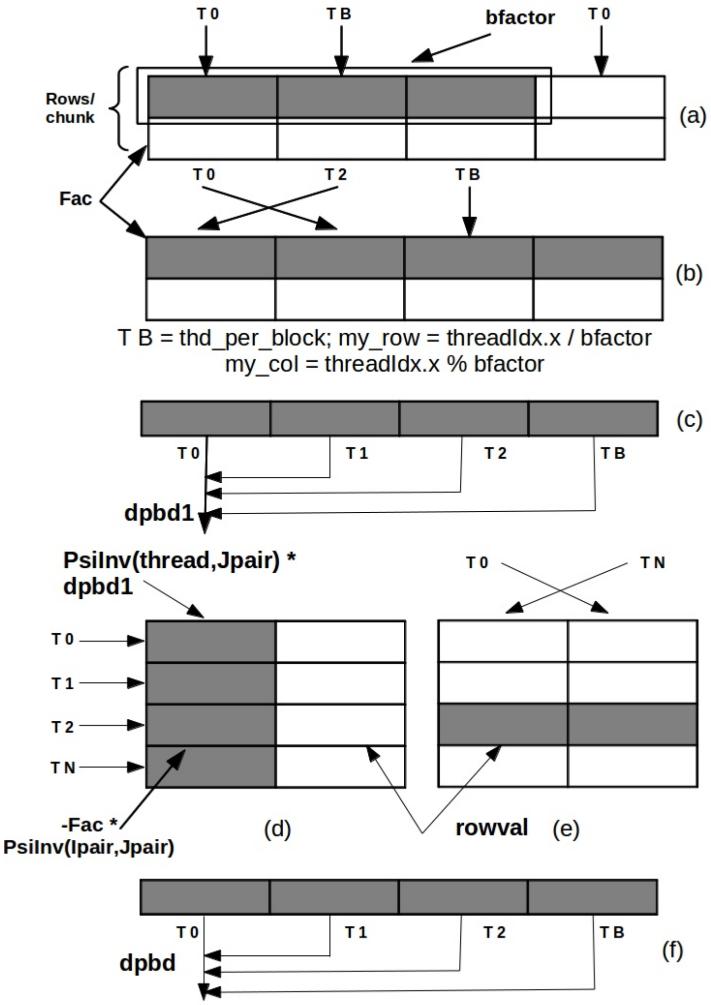

T 0
T B
bfactor
T 0
Rows/ chunk
(a)
Fac
T 0
T 2
T B
(b)
T B = thd_per_block; my_row = threadIdx.x / bfactor
my_col = threadIdx.x % bfactor
(c)
T 0
T 1
T 2
T B
dpbd1
PsiInv(thread,Jpair) * dpbd1
T 0
T N
T 0
T 1
T 2
T N
-Fac * PsiInv(Ipair,Jpair)
(d)
rowval
(e)
T 0
T 1
T 2
T B
dpbd
(f)

Figure 5.11: Optimization for down spin: (a) Ψ^{-1} - with T_B, (b) Pairfunction, (c) Reduction in a shared memory to obtain dpbd1, (d) Calculation of dot product using $Jpair^{th}$ column of Ψ^{-1}, (e) Calculation of dot product using Transposed pairfunction, (f) Reduction in a shared memory to obtain the final *dpbd*: $dpbd = dpbd1 * Reduction(rowval)$

Chapter 6
Results

In this section, we compare our results to the CPU, and MPI implementation [4].

6.1 Input Parameters

The input parameters used to run the VMC method on both CPU and GPU are given by Table 6.1.

From these parameters we obtain the N_e, N_P and N_S as per the Table 4.1.

6.2 Comparison between a CPU and GPU code

The CUDA code has been tested on a Tesla M2070 NVIDIA GPU and the results are benchmarked against a Intel(R)Core(TM)i7-2600 CPU @ 3.40 GHz. The MCs are done in parallel by a block of CUDA threads on a GPU, whereas a CPU core performs multiple MCs sequentially. We tested our GPU code by varying the blocking factor, bfactor for the UpdateConfig and the Streamlined function. Table 6.2 and 6.3 show the speedup compared to the CPU code for $bfactor = 8, 16$.

From Table 6.3 we notice a speedup of ≈ 2 compared to the GPU code with $bfactor = 8$. The speedup is mainly due to the access of 16 words (128 byte transaction) per warp compared to 8 words (64 byte transaction) for bfactor = 8.

6.3 Comparison between MPI-CPU and GPU code

We compared our CUDA code with the MPI code implemented by[5] using Fortran 90. The MPI code designates a CPU core per MC. The code has been tested on Philip supercomputer at HPC, LSU. Table 6.4 gives the nodal architecture for Philip supercomputer.

The MPI code written in Fortran 90, suffers from inter-processor and inter-node communication. The massive speedup on a GPU is due to the fact CUDA threads communicate

Table 6.1: Input Parameters

Parameter	**Function**	**Value**
t	First neighbor hopping	1.0
tp	Second neighbor hopping	0.0
tpp	Third neighbor hopping	0.0
Jex	Antiferromagnetic exchange	0.3
L	Lattice	5/9/15
x	Hole Doping	0.1
nattemptmax	Maximum generation attempts	1000000
ConditionNumberMin	Minimum condition number	1.E-07
Delta	Gap	1.5
mu	Chemical Potential	0.0
nsweeps	Number of sweeps	10000
neqlsweeps	Number of equilibration sweeps	5000
nfac	Sweep factor	1
nbins	Number of bins	10
neifrac	Neighbor fraction	0.0
naveseeds	Number of seeds to average over	1
selfrac	Selection fraction	0.75
iseedseed	Seed for random number	13213
nMarkov	Number of MCs	32

Table 6.2: Comparison of CPU vs GPU Performance for bfactor=8

Lattice	**No.MCs**	**CPU Exec Time(s)**	**GPU Exec Time(s)**	**Speedup**
$L = 5$	32	22.66	3.19	7.26
$L = 9$	32	287.13	17.37	16.72
$L = 15$	32	4992.24	309.72	16.11

Table 6.3: Comparison of CPU vs GPU Performance for bfactor=16

Lattice	**No.MCs**	**CPU Exec Time(s)**	**GPU Exec Time(s)**	**Speedup**
$L = 5$	32	22.66	2.48	9.13
$L = 9$	32	287.13	14.44	19.88
$L = 15$	32	4992.24	256.83	19.43

Table 6.4: Nodal architecture of Philip Supercomputer

Nodal Configuration	Two 2.93 GHz Quad Core Nehalem Xeon 64-bit Processors
No.Processors/node	8
Total No.Nodes	32
DRAM	24GB @ 1333MHz

Table 6.5: Comparison of MPI vs GPU Performance

Lattice	**No.MCs**	**MPI Exec Time(s)**	**GPU Exec Time(s)**	**Speedup**
$L = 5$	32	4.89	3.19	1.53
$L = 9$	32	63.66	17.37	3.66
$L = 15$	32	1193.687	309.72	3.85

in group of warp (32 threads) and all threads have a barrier synchtonization on a SM.

6.4 GPU results in depth

In this Section, we examine the individuals functions and their percentage of the total execution time.

Figures 6.1 illustrates the execution time taken by functions such as DPBD and UpdateConfig for $L = 5$. Figures 6.2 - 6.3 illustrates for $L = 9, 15$.

Table 6.6: GPU execution time in cycles for $L = 5$

Function	**No.Cycles**
Electron move	420361934
DPBD	
DPBD-UP	95922170
DPBD-DN	97922630
Total	**193844800**
UPD	
UPD-UP	322257414
UPD-DN	259125546
Total	**581382960**
Energy	969450046
Streamlined (DPBD-UPD-DPBD)	1630512656

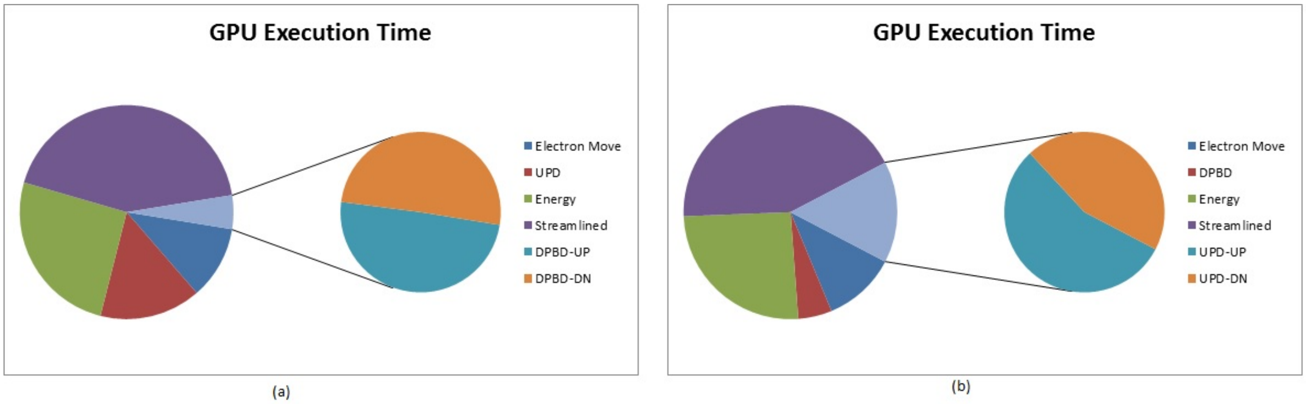


Figure 6.1: Pie chart for $L = 5$

Table 6.7: GPU execution time in cycles for $L = 9$

Function	**No.Cycles**
Electron move	1398769542
DPBD	
DPBD-UP	321308398
DPBD-DN	351500708
Total	**672809106**
UPD	
UPD-UP	1275117358
UPD-DN	1476716774
Total	**2751834132**
Energy	6105049222
Streamlined (DPBD-UPD-DPBD)	10717256244

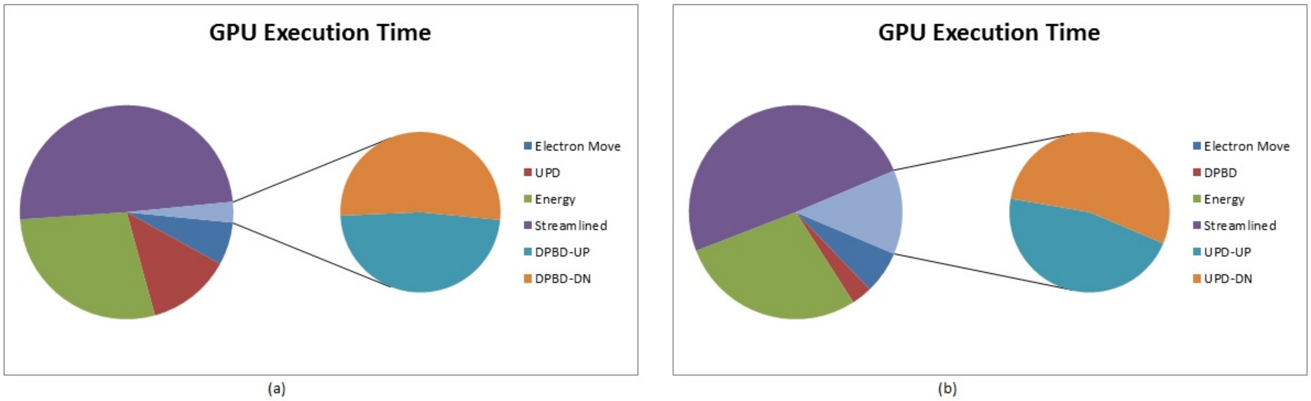


Figure 6.2: Pie chart for $L = 9$

Table 6.8: GPU execution time in cycles for $L = 15$

Function	**No.Cycles**
Electron move	4958484770
DPBD	
DPBD-UP	890322994
DPBD-DN	932234278
Total	**1822557272**
UPD	
UPD-UP	5874557324
UPD-DN	7310109920
Total	**13184667244**
Energy	48836694164
Streamlined (DPBD-UPD-DPBD)	8600610984504

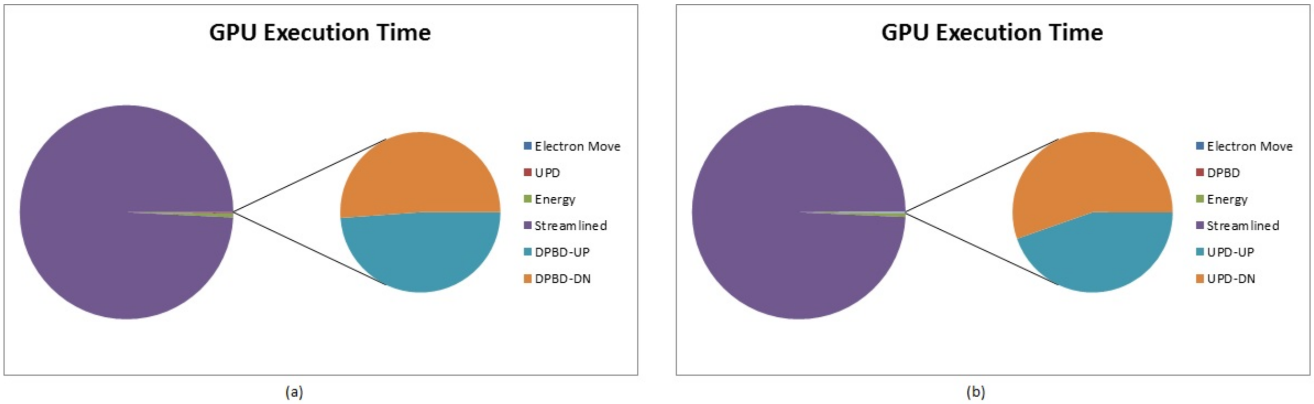


Figure 6.3: Pie chart for $L = 15$

Chapter 7
Conclusion

7.1 Conclusion

The VMC algorithm got extensive speedup compared to the MPI-Fortran implementation. We obtained nearly 114 times speedup compared to the MPI and nearly 16 times speedup compared to the C++ version. The MPI implementation suffers from extensive inter-processor and inter-node communication. Since, one Markov chain is designated to a MPI-Rank and no further optimization has been deployed, the MPI code does not scale well with the system size. From the results in the previous section, we observe that our GPU code gives a linear speedup as the lattice size increases.

7.2 Future Work

From the Algorithm‘s perspective, the ground state energy obtained by the VMC method will used be used to optimize the variational parameters and study the ground state properties of a system. The results can be compared with other approaches such as Exact Diagonalization to provide a better guess for the wavefunction. The GPU code can be enhanced by increasing the blocking factor for lattice models more than 15. Futher optimization can be done by examining the low-level assembly code to reduce the WAR, and RAW hazards. Possible expansion to multi-GPU(s) and porting it to future accelerators such as NVIDIA Kepler [13] and Intel Xeon Phi [14] will boost the performance further.

References

[1] Paramekanti, Randeria, Trivedi, "High-T_C superconductors: A variational theory of the superconducting state", *in Phys. Rev. B* 70, 054504, 2004.

[2] P. W. Anderson, P. A. Lee, T. M. Rice, N Trivedi, F. C. Zhang, "The physics behind high-temperature superconducting cuprates: the 'plain vanilla' version of RVB," *in J. Phys: Condens. Matter* 16 R755, 2004.

[3] B. Edegger, V.N. Muthukumar, C. Gros, "Gutzwiller-RVB Theory of High Temperature Superconductivity: Results from Renormalized Mean Field Theory and Variational Monte Carlo Calculations", *in Adv. Phys.* 56, 927, 2007.

[4] S. Pathak, V. B. Shenoy, M. Randeria, N. Trivedi, "Competition between Antiferromagnetic and Superconducting States, Electron Hole Doping Asymmetry, and Fermi-Surface Topology in High Temperature Superconductors", *in Phys. Rev. Lett.* 102, 027002", 2009.

[5] V. B. Shenoy, "Variational Monte Carlo Method for Fermions", in *IISC Lecture Notes, http://www.physics.iisc.ernet.in/ shenoy/LectureNotes/vmc.pdf*

[6] Nvidia, "Nvidia CUDA: Compute Unified Device Architecture: Parallel Programming model", *http://developer.nvidia.com/category/zone/cuda-zone.*

[7] Nvidia, "Nvidia CUDA Programming guide", *http://docs.nvidia.com/cuda/cuda-c-programming-guide.*

[8] W. M. Hwu, D. B. Kirk, "Programming Massively Parallel Processors - A Hands-on Approach", *Morgan Kaufmann publishers.*

[9] ORNL, “Variational Monte Carlo”, *http://www.ornl.gov/ pk7/thesis/pkthnode20.html.*

[10] Intel, “Intel $Core^{TM}$ i7-2600 Processor (8M Cache, up to 3.80 GHz)”, *http://ark.intel.com/products/52213.*

[11] Nvidia, “Nvidia‘s Next Generation CUDA Compute Architecture: Fermi”, *http://www.nvidia.com/object/fermi-architecture.html.*

[12] P. Micikevicius, “Local Memory and Register Spilling”, *http://developer.download.nvidia.com/CUDA/training/registerspilling.pdf*, 2011.

[13] Nvidia, “Nvidia‘s Next Generation CUDA Compute Architecture: Kepler GK 110”, *http://www.nvidia.com/object/nvidia-kepler.html.*

[14] Intel, “Xeon Phi: Parallel Processing, Unparalleled Discovery”, *http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.*

[15] Nightingale, Umrigar, “NATO Series Lecture Notes”, *http://www.phys.uri.edu/ nigh/QMC-NATO/webpage/abstracts/lectures.html.*

[16] D. Ceperley, G. V. Chester, M. H. Kalos, “Monte Carlo simulation of a many-femion study”, *in Phys. Rev. B* 16, 30813099, 1977.

[17] B. Tasseff, N. Setty, N. Sengupta, Z. Yun, “GPU-Accelerated Variational Monte Carlo-LSU”, *http://reu.cct.lsu.edu/documents/2012-posters/ByronTasseffURFPoster.pdf*

[18] A. Gothandaraman, “Accelerating Quantum Monte Carlo Simulations with Emerging Architectures”, *in Doctoral Dissertations University of Tennessee*, 2009.

Appendix A CUDA code for the Merged Equilibration and Accumulation stage

```
__global__ void k_sysAve
(
 Configuration d_config,
 MonteCarlo d_mcarlo, Lattice d_sqlat, Pairfunction pairfunc, double *d_enearr,
 curandState *state
  )
{
  int tid = threadIdx.x;
  int MID = blockIdx.x;
  curandState localState = state[MID];

#define dplist(i) d_config.plist[MID*2*npairs+(i)]
#define dlatocc(i) d_config.latocc[MID*nsites+(i)]
#define dwhichpair(i) d_config.whichpair[MID*2*nsites+1+(i)]

  __shared__ int idat, nbsamp;
  __shared__ int ipair,ispin,oldsite,newsite;
  __shared__ bool spin_flip;
  bool donenei;
  __shared__ double eneloc;

# ifdef COLLECT_CYCLE_COUNTS
  // These are declared shared to avoid wasting registers.
  __shared__ clock_t timer_start_thread, timer_start_eoc, timer_start_findpairs;
  __shared__ clock_t time_eoc, time_findpairs;
# endif

  if ( !tid )
    {
#     ifdef COLLECT_CYCLE_COUNTS
      timer_start_thread = clock64();
      time_eoc = time_findpairs = 0;
      time_detpbydet[0] = time_detpbydet[1] = 0;
      time_update_config[0] = time_update_config[1] = 0;
      time_update_config_cnt = time_detpbydet_cnt = 0;
      time_d_update_d[0] = time_d_update_d[1] = 0;
      time_d_update_d_cnt = 0;
#     endif

      idat = 0; eneloc = 0.0; nbsamp = 0;
    }

  // Equilibrate the system by performing a large number of Monte Carlo sweeps
  for ( int i2 = 0; i2 < navesweeps + neqlsweeps; i2++ )
    {
      for (int i = 0; i < npsweep; i++ )
{
  if(tid == 0)
    {
#             ifdef COLLECT_CYCLE_COUNTS
              timer_start_findpairs = clock64();
#             endif

      // Pick a pair and a spin at random
              ipair = curand_uniform( &localState ) * dc.npairs_randscaler;
      ispin = curand( &localState ) % 2;
      oldsite = dplist(ipair*2+ispin);

      // Find a site where this electron may be moved
      newsite = 0;
      while ( newsite == 0 )
{
                  const float neiprob = curand_uniform( &localState );
                  donenei = neiprob <= dc.neifrac;
  if ( !donenei )
    {
                      newsite = curand_uniform( &localState )
                        * dc.nsites_randscaler;
    }
  else
    {
                      const int inei = curand_uniform( &localState )
                        * dc.nneibs_randscaler;
      newsite = d_sqlat.neiblist[oldsite*nneibs+inei];
    }
  if ( dlatocc(newsite) != ispin )
    {
      if ( dlatocc(newsite) == HL )
{
  spin_flip = false;
}
      else if ( false && dlatocc(newsite) == BT )
{
                          assert( false );
  newsite = 0;
  //miscutils::InfoPrint( "****WARNING: Gutzwiller projection violation" );
  //exit( 0 );
}
      else
{
  spin_flip = true;
}
    }
  else
    {
      newsite = 0;
    }
}
#             ifdef COLLECT_CYCLE_COUNTS
              time_findpairs += clock64() - timer_start_findpairs;
#             endif
    }
  __syncthreads();

          // Note: these were complex doubles in original C++ code
          double dpbd;

  if ( spin_flip )
    {
              const int jpair = dwhichpair(newsite*2-ispin);
              dpbd = d_UpdateConfig_d
                ( ipair, ispin, newsite, MID, pairfunc, d_config,
                  jpair, oldsite );
  }
 else
  {
   dpbd = DetPByDet_parallel( ipair, ispin, newsite, MID, pairfunc, d_config.PsiInv,
  }

 __shared__ bool accept;

     if ( !tid )
      {
       const float norm2 = dpbd * dpbd;
       const float prob = curand_uniform( &localState );
       accept = norm2 >= prob;
      }
     __syncthreads();

 if ( accept )
  {
   // Accept the move
   if ( !tid ) d_mcarlo.naccept[MID] += 1;
   if ( spin_flip )
{
         const int jpair = dwhichpair(newsite*2-ispin);
         const double dpbd1 =
          DetPByDet_parallel
          ( ipair, ispin, newsite, MID,
           pairfunc, d_config.PsiInv, d_config.plist );
         // Update the configuration
         UpdateConfig_parallel
          ( ipair, ispin, newsite, MID,
           pairfunc, dpbd1, d_config );
         const double dpbd2 = DetPByDet_parallel
          ( jpair, 1-ispin, oldsite, MID,
           pairfunc, d_config.PsiInv, d_config.plist );

         // Now, move the other spin electron to the old site

 UpdateConfig_parallel( jpair, 1-ispin, oldsite, MID, pairfunc, dpbd2, d_config );

 if(tid == 0)
```

```
  d_mcarlo.nflip[MID] += 1;
}
   else
{
 UpdateConfig_parallel( ipair, ispin, newsite, MID, pairfunc, dpbd, d_config );
 if(tid == 0)
  d_mcarlo.nsingle[MID] += 1;
}

   if ( donenei )
{
 if(tid == 0)
  d_mcarlo.nneistep[0] += 1;
}
  }
}

   if ( i2 < neqlsweeps ) continue;

   __syncthreads();

   // Now, calculate the energy of the configuration
#  ifdef COLLECT_CYCLE_COUNTS
   if ( !tid ) timer_start_eoc = clock64();
#  endif
   const double energyvalue =
    EnergyOfCurrentConfiguration( d_config, d_sqlat, pairfunc);

#  ifdef COLLECT_CYCLE_COUNTS
   if ( !tid ) time_eoc += clock64() - timer_start_eoc;
#  endif
   if ( !tid )
{
 idat++;
 eneloc += energyvalue;

     if ( idat == ndat )
      {
       d_enearr[MID*nbins+nbsamp] = eneloc / double( ndat );
       nbsamp++; eneloc = 0.0; idat = 0;
```

```
            }
        }
    }

  if ( !tid )
    {
      d_mcarlo.nsteps[MID] = navesweeps * npsweep;

#     ifdef COLLECT_CYCLE_COUNTS
      clock_t time_total = clock64() - timer_start_thread;
      timers[blockIdx.x].total = time_total;
      timers[blockIdx.x].eoc = time_eoc;
      timers[blockIdx.x].detpbydet[0] = time_detpbydet[0];
      timers[blockIdx.x].detpbydet[1] = time_detpbydet[1];
      timers[blockIdx.x].detpbydet_cnt = time_detpbydet_cnt;
      timers[blockIdx.x].update_config[0] = time_update_config[0];
      timers[blockIdx.x].update_config[1] = time_update_config[1];
      timers[blockIdx.x].update_config_cnt = time_update_config_cnt;
      timers[blockIdx.x].d_update_d[0] = time_d_update_d[0];
      timers[blockIdx.x].d_update_d[1] = time_d_update_d[1];
      timers[blockIdx.x].d_update_d_cnt = time_d_update_d_cnt;
      timers[blockIdx.x].findpairs = time_findpairs;
#     endif
    }
#undef dplist
#undef dlatocc
#undef dwhichpair
}
```

Appendix B
Device Functions

```
// Energy of a configuration

__device__ double
EnergyOfCurrentConfiguration
( Configuration d_config, Lattice d_sqlat, Pairfunction pairfunc )
{
  int MID = blockIdx.x;
  __shared__ double enekloc, otheloc, spkeloc[2];

  // ***---------------------------------------------------------
  //        Kinetic energy part of the Hamiltonian
  //                          _
  //         H  = -\sum  t  ( c   c   )
  //          t    (i,j)  ij   is  js
  // ***---------------------------------------------------------

#define PsiInv(r,c) d_config.PsiInv[MID*npairs*npairs+(r)*npairs+(c)]
#define Plist(i) d_config.plist[MID*2*npairs+(i)]

  {
    const int ispin = 0;             // Down
    // The variables below need to be tuned.
    const int thd_per_jn = 1;
    const int thd_per_ipair = thd_per_col;
    const int my_1st_ipair = threadIdx.x % thd_per_ipair;
    const int my_1st_col = threadIdx.x / thd_per_ipair % thd_per_jn;
    const int jn = threadIdx.x / ( thd_per_ipair * thd_per_jn );
    double thread_sum = 0;

    if ( jn < nneibs )
      {
        for (int ipair = my_1st_ipair; ipair < npairs; ipair += thd_per_ipair )
          {
            const int isite = Plist(ipair*2+ispin);
            const int jsite = d_sqlat.neiblist[jn+nneibs*isite];
            if ( d_config.latocc[MID*nsites+jsite] == HL )
              for ( int col = my_1st_col; col < npairs; col += thd_per_jn )
                thread_sum +=
                  TPairfunc(Plist(col*2+1),jsite) * PsiInv(ipair,col);
          }

        thread_sum *= d_sqlat.thop_ishell[jn];
      }

    const double sum =
      reduce_dbl_buffer( thread_sum, nneibs * thd_per_ipair * thd_per_jn );

    if ( !threadIdx.x )
      {
        enekloc = sum;
        spkeloc[ispin] = sum;
      }
  }
  {
    const int ispin = 1;  // Up
    const int thd_per_jn = 1;
    const int thd_per_ipair = thd_per_col;
    const int my_1st_ipair = threadIdx.x % thd_per_ipair;
    const int my_1st_row = threadIdx.x / thd_per_ipair % thd_per_jn;
    const int jn = threadIdx.x / ( thd_per_ipair * thd_per_jn );
    double thread_sum = 0;

    if ( jn < nneibs )
      {
        for (int ipair = my_1st_ipair; ipair < npairs; ipair += thd_per_ipair )
          {
            const int isite = Plist(ipair*2+ispin);
            const int jsite = d_sqlat.neiblist[jn+nneibs*isite];

            if ( d_config.latocc[MID*nsites+jsite] == HL )
              for ( int row = my_1st_row; row < npairs; row += thd_per_jn )
                thread_sum +=
                  Pairfunc(jsite,Plist(row*2)) * PsiInv(row,ipair);
          }
        thread_sum *= d_sqlat.thop_ishell[jn];
      }

    const double sum =
      reduce_dbl_buffer(thread_sum, nneibs * thd_per_jn * thd_per_ipair);
    if ( !threadIdx.x )
      {
        enekloc += sum;
        spkeloc[ispin] = sum;
      }
  }
#undef PsiInv
#undef Plist


// ***---------------------------------------------------------
//        Now compute the Exchange Term
//                  _  _     1
//         H  = J ( S .S  - ---  n n  ) Term
//          J        i  j    4    i j
// ***---------------------------------------------------------
  if ( !threadIdx.x ) otheloc = 0.0;

  for ( int inn = 0; inn < nearnsets; inn++ )
    {
      const int isite = d_sqlat.nearnp[inn*2+0];
      const int jsite = d_sqlat.nearnp[inn*2+1];
      // Note ispin is used as spin here.
      const int sispin = d_config.latocc[MID*nsites+isite];
      const int jspin = d_config.latocc[MID*nsites+jsite];

      if ( sispin + jspin == 1 )
{
          const int jpair =
            d_config.whichpair[MID*2*nsites+jsite*2+jspin];
          const int sipair =
            d_config.whichpair[MID*2*nsites+isite*2+sispin];

          const double e = d_UpdateConfig_d
          ( sipair, sispin, jsite, MID, pairfunc, d_config, jpair, isite );

  if ( !threadIdx.x ) otheloc -= e + 1;
}
    }

  double energy = 0;
  if ( !threadIdx.x )
    {
      otheloc = Jij * otheloc / 2.0;
      energy = ( enekloc + otheloc ) / double( nsites );
      d_config.energy[MID] = energy;
      d_config.kinenergy[MID] = ( spkeloc[0] + spkeloc[1] ) / double( nsites );
      d_config.othenergy[MID] = otheloc;
      d_config.modified[MID] = false;
    }
  return energy;
}

// Reduction Routine
//
// Uses dbl_buffer, so cannot be mixed with other code using dbl_buffer.
//
__device__ double
reduce_dbl_buffer(double val, int size, bool pre_written = false)
{
  // Return sum of VAL for threads 0 to SIZE-1.
  // If PRE_WRITTEN is true, ignore VAL and use value already in dbl_buffer.

  const int tid = threadIdx.x;
  __shared__ double sum;
  __syncthreads(); // Don't write sum too early.
  if ( !pre_written )
    {
      if ( tid < size + 31 ) dbl_buffer[tid] = val;
      __syncthreads();
    }
  const bool one_warp = size <= WP_SZ;

  // Optimization-friendly, sync-free code for a single-warp block.
```

```
# define STEP(offset) if ( tid < offset ) \
    dbl_buffer[tid] += dbl_buffer[tid + offset];

  if ( tid < WP_SZ )
    {
      if ( !one_warp )
        for ( int i = tid + WP_SZ; i < size; i += WP_SZ )
          dbl_buffer[tid] += dbl_buffer[i];
      if ( size >= 16 ) STEP(16);
      STEP(8); STEP(4); STEP(2); STEP(1);
      if ( !tid ) sum = dbl_buffer[0];
    }
  __syncthreads();
  return sum;

#undef STEP
}

__device__ double
reduce_dbl_buffer(int size)
{
  // Return sum of values already in dbl_buffer, from index 0 to size-1

  return reduce_dbl_buffer(0,size,true);
}

// Ratio of Determinant of the configurations - DetPByDet

__device__ double DetPByDet_parallel
(
 int ipair, int spin, int newsite, int mid,
 Pairfunction pairfunc, double *d_PsiInv, int *d_plist
 )
{
#define PsiInv(r,c) d_PsiInv[mid*npairs*npairs+(r)*npairs+(c)]
#define Plist(i) d_plist[mid*2*npairs+(i)]

# ifdef COLLECT_CYCLE_COUNTS
  __shared__ clock_t time_start;
  if ( !threadIdx.x ) time_start = clock64();
```

```
# endif

  double ans = 0.0;
  const int i = threadIdx.x;

  if ( spin == UP )
    {
      if(i<npairs)
{
  int othsite = Plist(i*2); // 0 for DN
  ans = Pairfunc(newsite,othsite) * PsiInv(i,ipair);
}
    }
  else if ( spin == DN )
    {
      if(i<npairs)
{
  int othsite = Plist(1+i*2); // 1 for UP
  ans = TPairfunc(othsite,newsite) * PsiInv(ipair,i);
}
    }
#undef PsiInv
#undef Plist

  const double sum = reduce_dbl_buffer(ans,npairs);

# ifdef COLLECT_CYCLE_COUNTS
  if ( !threadIdx.x )
    {
      time_detpbydet[spin==DN?0:1] += clock64() - time_start;
      time_detpbydet_cnt++;
    }
# endif

  return sum;
}

//Update of a configuration

 __device__ void UpdateConfig_parallel
```

```
(
 int ipair, int ispin, int newsite, int mid, Pairfunction pairfunc, double dpbdt, Config
 )
{
  const int tid = threadIdx.x;

# ifdef COLLECT_CYCLE_COUNTS
  __shared__ clock_t timer_start;
  if ( !threadIdx.x ) timer_start = clock64();
# endif

# define dPsiInv(i) d_config.PsiInv[ mid * npairs * npairs + (i) ]
# define dplist(i) d_config.plist[mid*2*npairs + (i) ]
# define dwhichpair(i) d_config.whichpair[mid*2*nsites + (i)]
# define dlatocc(i) d_config.latocc[mid*nsites + (i)]

  // First, update the pair list and lattice occupancy

  if(tid == 0) {
    const int oldsite = dplist(ipair*2+ispin);
    dplist(ipair*2+ispin) = newsite;
    dwhichpair(oldsite*2+ispin) = 0;

    if ( dlatocc(oldsite) == BT )
      {
dlatocc(oldsite) = 1-ispin;
      }
    else
      {
dlatocc(oldsite) = HL;
      }
    if ( dlatocc(newsite) == 1-ispin )
      {
dlatocc(newsite) = BT;
      }
    else
      {
dlatocc(newsite) = ispin;
      }
    dwhichpair(newsite*2+ispin) = ipair;
  }
  __syncthreads();
  // Now, update PsiInv

  if ( ispin == UP )
    {
      // First, update the ipair-th column of PsiInv
      if ( tid < npairs ) dPsiInv(ipair+tid*npairs) /= dpbdt;
      __syncthreads();

      // Update the other columns

      // WARNING: block size must be >= thd_per_col * npairs.
      const int my_col = threadIdx.x / thd_per_col;
      const int my_1st_row = threadIdx.x % thd_per_col;
      if ( my_col < npairs && my_col != ipair )
        {
          double fac = 0.0;
          for ( int i = my_1st_row; i < npairs; i+=thd_per_col)
            {
              const int othsite = dplist(i*2);
              fac += dPsiInv(i*npairs+my_col) * Pairfunc(newsite,othsite);
            }
          if ( thd_per_col > 1 )
            {
              if ( my_1st_row == 0 ) dbl_buffer[my_col] = fac;
              for ( int k = 1; k < thd_per_col; k++ )
                if ( my_1st_row == k ) dbl_buffer[my_col] += fac;
              fac = dbl_buffer[my_col];
            }

          for ( int i = my_1st_row; i < npairs; i+=thd_per_col )

      dPsiInv(i*npairs+my_col) -= fac * dPsiInv(i*npairs+ipair);

        }
      __syncthreads();
    }
  // all threads
```

```
else if ( ispin == DN )
  {

    // First, update the ipair-th row of PsiInv
    if ( tid < npairs ) dPsiInv(ipair*npairs+tid) /= dpbdt;
    __syncthreads();

    // Update the other rows
    const int ispin = 1;
    const int bfactor = 8;  // Blocking factor.
    const int my_1st_row = tid / bfactor;
    const int my_1st_col = tid % bfactor;
    const int nthds = blockDim.x;  // nthds must be a mult of bfactor.
    const int rows_per_chunk = nthds / bfactor;

    for ( int my_row = my_1st_row; my_row < npairs; my_row += rows_per_chunk )
      if ( my_row != ipair )
        {
          double fac = 0.0;

          // Compute sum of "our" row elements.
          for ( int i = my_1st_col; i < npairs; i+= bfactor )
            {
              const int othsite = dplist(i*2+ispin);
              fac += dPsiInv(i+npairs*my_row) * TPairfunc(othsite,newsite);
            }

          // Get sum of entire row.
          if ( bfactor == 8 && DBL_BUFFER_SIZE >= blockDim.x + 4 )
            {
              // Optimize for popular value of bfactor.
              dbl_buffer[tid] = fac;
              fac = dbl_buffer[tid] = fac + dbl_buffer[tid+1];
              fac = dbl_buffer[tid] = fac + dbl_buffer[tid+2];
              dbl_buffer[tid] = fac + dbl_buffer[tid+4];
              fac = dbl_buffer[my_1st_row * bfactor];
            }
          else if ( bfactor > 1 )
            {
              if ( my_1st_col == 0 ) dbl_buffer[my_row] = fac;
        for ( int k = 1; k < bfactor; k++ )
         if ( my_1st_col == k ) dbl_buffer[my_row] += fac;
        fac = dbl_buffer[my_row];
       }

      // Update row using sum.
      for ( int i = my_1st_col; i < npairs; i+=bfactor )
       dPsiInv(my_row*npairs+i) -= fac * dPsiInv(ipair*npairs+i);
     }

   __syncthreads();
  }

 if ( !threadIdx.x )
  {
   d_config.modified[mid] = true;
#  ifdef COLLECT_CYCLE_COUNTS
   time_update_config[ispin==DN?0:1] += clock64() - timer_start;
   time_update_config_cnt++;
#  endif
  }
# undef dplist
# undef dPsiInv
# undef dwhichpair
# undef dlatocc

}

//Streamlined function - DetPByDet-UpdateConfig-DetPByDet

__device__ double d_UpdateConfig_d
( int ipair, int spin, int newsite,
 int mid, Pairfunction pairfunc, Configuration d_config,
 int jpair, int isite )
{
 const int tid = threadIdx.x;

# ifdef COLLECT_CYCLE_COUNTS
 __shared__ clock_t timer_start;
 if ( !threadIdx.x ) timer_start = clock64();
```

```
# endif

 double rv = 0;

# define dPsiInv(r,c) d_config.PsiInv[ mid*npairs*npairs + npairs*(r) + (c) ]
# define dplist(i) d_config.plist[mid*2*npairs + (i) ]

 if ( spin == UP )
  {
   // WARNING: block size must be >= thd_per_col * npairs.
   const int my_col = threadIdx.x / thd_per_col;
   const int my_1st_row = threadIdx.x % thd_per_col;
   double fac = 0.0;

   if ( my_col < npairs )
    {
     for ( int i = my_1st_row; i < npairs; i+=thd_per_col)
      {
       const int othsite = dplist(i*2);
       fac += dPsiInv(i,my_col) * Pairfunc(newsite,othsite);
      }
     if ( thd_per_col > 1 )
      {
       if ( my_1st_row == 0 ) dbl_buffer[my_col] = fac;
       for ( int k = 1; k < thd_per_col; k++ )
        if ( my_1st_row == k ) dbl_buffer[my_col] += fac;
       fac = dbl_buffer[my_col];
      }
    }
   __syncthreads();
   double col_val = 0;
   if ( tid < npairs )
    {
     const int my_col = tid;
     const int othsite = dplist(my_col*2+1);
     const double fac = dbl_buffer[my_col];
     const double dpbdt = dbl_buffer[ipair];

     col_val =
      my_col == ipair
```

```
       ? ( dPsiInv(jpair,ipair) * TPairfunc(newsite,isite) )
       : ( ( dPsiInv(jpair,my_col) * dpbdt - fac * dPsiInv(jpair,ipair) )
         * TPairfunc(othsite,isite) );
     }
    rv = reduce_dbl_buffer(col_val,npairs);
   }
  else
   {
    // Update the other rows
    const int ispin = 1;

    const int bfactor = 8;  // Blocking factor.
    const int my_1st_row = tid / bfactor;
    const int my_1st_col = tid % bfactor;
    const int nthds = blockDim.x;  // nthds must be a mult of bfactor.
    const int rows_per_chunk = nthds / bfactor;

    __syncthreads();

    for ( int my_row = my_1st_row; my_row < npairs; my_row += rows_per_chunk )
      {
       double fac = 0.0;

       // Compute sum of "our" row elements.
       for ( int i = my_1st_col; i < npairs; i+= bfactor )
        {
         const int othsite = dplist(i*2+ispin);
         fac += dPsiInv(my_row,i) * TPairfunc(othsite,newsite);
        }

       // Get sum of entire row.
       if ( bfactor == 8 && DBL_BUFFER_SIZE >= blockDim.x + 4 )
        {
         // Optimize for popular value of bfactor.
         dbl_buffer[tid] = fac;
         fac = dbl_buffer[tid] = fac + dbl_buffer[tid+1];
         fac = dbl_buffer[tid] = fac + dbl_buffer[tid+2];
         if ( my_1st_col == 0 )
          dbl_buffer[blockDim.x+my_row] = fac + dbl_buffer[tid+4];
        }
```

```
            else if ( bfactor > 1 )
              {
                if ( my_1st_col == 0 ) dbl_buffer[my_row] = fac;
                for ( int k = 1; k < bfactor; k++ )
                  if ( my_1st_col == k ) dbl_buffer[my_row] += fac;
                fac = dbl_buffer[my_row];
                if ( my_1st_col == 0 )
                  dbl_buffer[blockDim.x+my_row] = fac;
              }
          }

      __syncthreads();
      double row_val = 0;

      if ( tid < npairs )
        {
          const int my_row = tid;
          const int othsite = dplist(my_row*2);
          const double fac = dbl_buffer[blockDim.x+my_row];
          const double dpbdt = dbl_buffer[blockDim.x+ipair];

          row_val =
            my_row == ipair
            ? ( dPsiInv(ipair,jpair) * Pairfunc(isite,newsite) )
            : ( ( dPsiInv(my_row,jpair) * dpbdt - fac * dPsiInv(ipair,jpair) )
                * Pairfunc(isite,othsite) );
        }
      rv = reduce_dbl_buffer(row_val,npairs);
    }

# ifdef COLLECT_CYCLE_COUNTS
  if ( !threadIdx.x )
    {
      time_d_update_d[spin==DN?0:1] += clock64() - timer_start;
      time_d_update_d_cnt++;
    }
# endif
# undef dplist
# undef dPsiInv
  return rv; }
```

Vita

Rajagopalan, Kaushik Ragavan was born in Chennai, India, on 1989 to Mrs. Pushpa and Mr. T. A. Rajagopalan. After graduating from his high school with distiction, he went on to study B.ENG in Electronics and Communication at Rajalakshmi Engineering college, Anna University, India, from 2006 to 2010. He is currently a Masters student in the Department of Electrical & Computer Engineering at Louisiana State University, Baton Rouge, where he has been a graduate student since Fall 2010. During his college days, he worked as a Research Assistant at the Center for Computation and Technology, LSU for the LA-Sigma research group and contributed by tuning MonteCarlo based applications to Multi-core CPU and accelerators. With this experience, he did a summer internship at the National Institute of Computational Sciences, Oak Ridge, TN on Porting Electronic structure calculations to GPU.