THE UNIVERSITY OF
WARWICK

AUTHOR: **Thomas Edwards** DEGREE: **MSc**

TITLE: **Parallelising the Transfer-Matrix Method using Graphics Processors**

DATE OF DEPOSIT: .....

I agree that this thesis shall be available in accordance with the regulations governing the University of Warwick theses.

I agree that the summary of this thesis may be submitted for publication.

I **agree** that the thesis may be photocopied (single copies for study purposes only).

Theses with no restriction on photocopying will also be made available to the British Library for microfilming. The British Library may supply copies to individuals or libraries. subject to a statement from them that the copy is supplied for non-publishing purposes. All copies supplied by the British Library will carry the following statement:

> "Attention is drawn to the fact that the copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's written consent."

AUTHOR'S SIGNATURE: .....

USER'S DECLARATION

1. I undertake not to quote or make use of any information from this thesis without making acknowledgement to the author.
2. I further undertake to allow no-one else to use this thesis while it is in my care.

DATE SIGNATURE ADDRESS

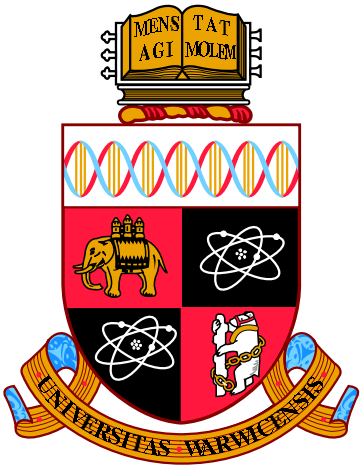
.....

.....

.....

.....

.....



Parallelising the Transfer-Matrix Method using Graphics Processors

by

Thomas Edwards

Thesis

Submitted to the University of Warwick

for the degree of

Master of Sciences

Centre for Scientific Computing, Department of Physics

December 2011

THE UNIVERSITY OF WARWICK

Contents

Acknowledgments

I would like to thank my supervisor, Prof. Rudolf Römer, for being supportive and helping me through my Masters of Science, for his patience in reading my thesis drafts and offering feedback, and for helping me feel welcome in the Disordered Quantum Systems research group. With his help I have gained a lot of useful skills and a good feel for research in general, for which I am very grateful. I would like to thank Dr. Marc Eberhard for his valuable CUDA expertise, for helping me debug my CUDA code and for marking my thesis. I would also like to thank Dr. David Quigley for allowing me to use his Tesla C1060 GPU as well as offer some CUDA advice. I would like to thank the system administrators at the Centre for Scientific Computing, for going out their way to sort out technical problems arising during my research. I would like to thank the cleaners working in the Physical Sciences building for keeping my office clean and for their friendly smiles and conversation. I would like to thank my work colleagues in the Disordered Quantum Systems research group for helping me get aquainted with the technical, administrative and scientific aspects of my Masters, and for their social company throughout the year. In particular I would like to thank Sebastian Pinski for helping me get started with some of the technical sides of research (such as using ssh to remotely login to the HPC systems) and Dr. Andrea Fischer for offering very important advice and helping me get through my thesis in the last couple of months. I would also like to thank my friends and work colleagues in room PS001 (and other nearby offices) for their never-ending supply of jokes and office banter. Last but not least, I would like to extend my thanks to my friends and family for offering much needed social and personal support.

List of Tables

List of Figures

Abstract

We study the disorder-induced Anderson localisation of a d-dimensional solid, computing the localisation lengths using the Transfer-Matrix Method (TMM) and aiming to develop an efficient parallel implementation to run on Graphics Processing Units (GPUs). In the TMM, a quasi one-dimensional bar of length $L >> M$ is split into slices of size M^{d-1}. The Schrödinger equation is reformulated into a $2M \times 2M$ transfer matrix T_n, which is recursively applied at each slice to propagate the wavevectors through the solid. NVidia's programming architecture for GPUs, CUDA, is used to develop the GPU implementation of the TMM, the CUDA-TMM. Two schemes are developed, the Multi-Parameter Scheme (MPS) and the Single-Parameter Scheme (SPS). In this thesis, various advantages and limitations of both schemes as well as using CUDA in general are discussed.

Abbreviations

CoW - Cluster of Workstations

CPU - Central Processing Unit

CSC - Centre for Scientific Computing

CUDA - Compute Unified Device Architecture

CUDA-TMM - CUDA implementation of the TMM carried out on GPUs

DES - Distributed Element Scheme

DVS - Distributed Vector Scheme

GPU - Graphics Processing Unit

GPGPU - General Purpose Graphics Processing Unit

HBC - Hardwall Boundary Conditions

MIT - Metal-Insulator Transition

MPS - Multi-Parameter Scheme

PBC - Periodic Boundary Conditions

Serial-TMM - Serial implementation of the TMM carried out on CPUs

SM - Streaming Multi-Processor

SPS - Single-Parameter Scheme

TMM - Transfer-Matrix Method

Chapter 1

Introduction

1.1 Anderson Localisation and the Metal-Insulator Transition

Anderson localisation is the absence of diffusion of waves in a disordered medium [1]. It applies to any sort of wave where disorder can occur, such as electromagnetic [2, 3, 4], water [5], sound [6, 7] and quantum waves [8]. It was first suggested in the context of electrons in disordered semi-conductors [9], as a possible mechanism for the metal-insulator transition (MIT). The absence of electron transport for these systems is due to the failure of the energies of neighbouring sites (atoms) to match sufficiently [1]. The disorder in semi-conductors can take many forms such as random impurities, vacant atoms, and abnormal lattice spacings [8]. In disordered semi-

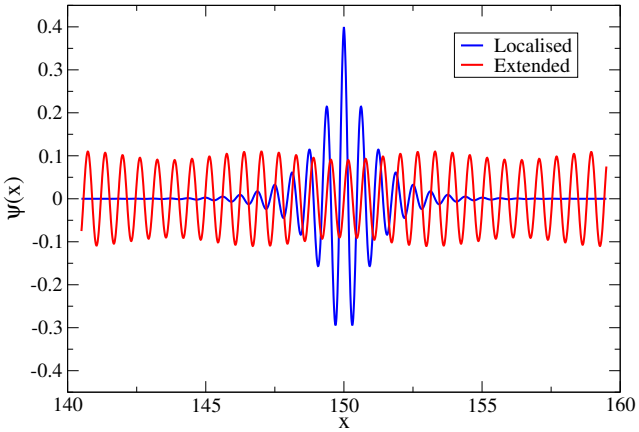


Figure 1.1: Plot of wavevector amplitude against position (lattice site). Blue: exponentially localised electron eigenstate in the insulating phase. Red: extended electron state corresponding to the metallic phase.

conductors, transport of electrons occurs via quantum-mechanical jumps from site to site [1]. At low temperature T and when the disorder of a metal increases to a certain amount, a phase transition occurs which causes the electrons to exponentially

localise. This means that the wavefunction for an electron at a particular site in the disordered system will decay exponentially away from that site. This is caused by the electron wavefunction interfering with itself due to disorder scattering [1], such that electrons are no longer spread out across the system like the extended states of a metal, and thus the system becomes insulating. The 'localisation length' describes the characteristic decay length for the electron wavefunction [8]

$$\psi(r) = f(r)e^{\frac{-r}{\lambda}},$$

where λ is the localisation length, and r is the distance of the electron from a particular site. This localisation effect prevents the diffusion of electrons at $T = 0$ (i.e. the system is an insulator). This is known as the disorder-induced MIT [8, 10, 11].

1.2 Conductance and the One-Parameter Scaling Theory

The disorder-induced MIT has been approached by Abrahams et al in 1979 using 'one-parameter scaling theory' [9]. In this study, the MIT was found to exist in three-dimensional systems with no electron-electron interactions, no magnetic field and no spin-orbit coupling. The theoretical approach that explains this is called the 'scaling hypothesis of localization'. Essentially the scaling hypothesis states that there is only one relevant scaling variable which describes the critical behaviour of the conductivity or localisation length at the MIT [8]. In the localised regime, conductivity becomes vanishingly small and is no longer a useful quantity to describe electron transport in finite systems [8, 12]. Instead of looking at the conductivity, one starts by investigating the conductance of an L^d sized metallic cube [13, 8]

$$G = \sigma L^{d-2} = g\frac{e^2}{\hbar},$$

where σ = conductivity, d = dimensionality, e = electron charge, $\hbar$ = Planck's constant and g = dimensionless conductance. For metals, this means that

$$g \propto L^{d-2}.$$

For insulators, i.e. when the disorder is strong and the wavefunction is exponentially localised, the conductance decays with system size [13]

$$g \propto e^{-L/\lambda}.$$

To see how the conductance behaves as the size changes, one then defines the logarithmic derivative [13, 8]

$$\beta = \frac{d\log g}{d\log L}.$$

and looks at how this behaves asymptotically to determine the onset of the metallic and insulating phases for different dimensions d [8]. For large conductance, one has

$$\beta = \frac{d}{d\log L}(d-2)\log L = d-2,$$

and for small conductance

$$\beta = \frac{d(\frac{-L}{\lambda})}{\frac{1}{L}dL} = -\frac{L}{\lambda} = \log g.$$

The β curves plotted in figure 1.2 show that β is always negative for $d \leq 2$ which

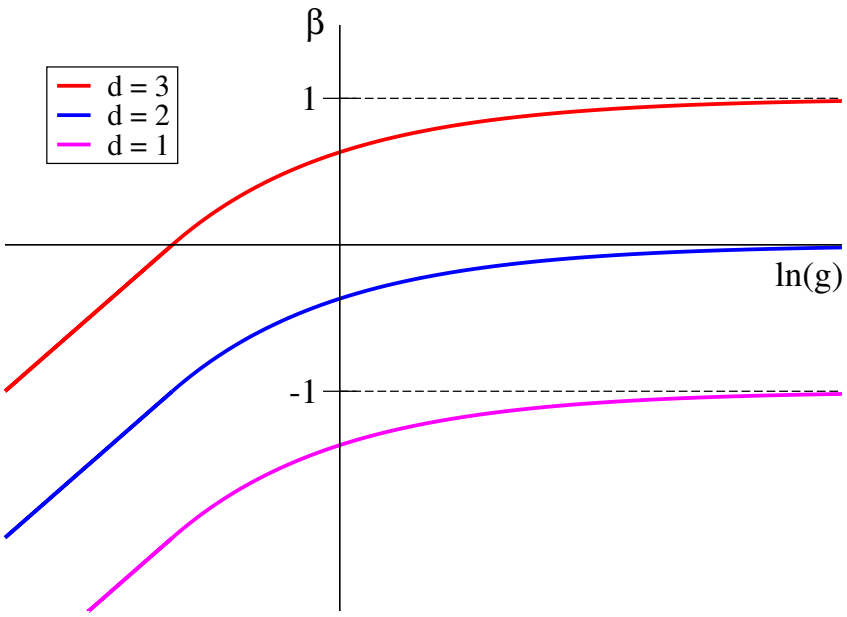


Figure 1.2: Schematic of the β curves showing the conductance of disordered systems for dimensionality $d = 1, 2, 3$. In the 1D and 2D cases, $\beta < 0$. For a 3D system, there is a critical conductance at $\beta = 0$ where the MIT occurs.

implies that an increase in the system size L will drive it to an insulator [13]. Therefore for 1D and 2D systems, there are no extended states and hence these systems are insulators. For $d = 3$, β is negative for small g and positive for large g, showing that there is a MIT at the critical conductance g_c where $\beta = 0$, so an increase of L will either drive the system to a metallic or insulating phase [13]. Thus the main result of the one-parameter scaling theory is that an MIT can only exist in three dimensions. The behaviour of the conductance for different signs of β is summarised below:

- $\beta < 0$ conductance decreases with system size (insulator)
- $\beta > 0$ conductance increases with system size (metallic)

- $\beta = 0$ conductance independent of system size (MIT).

The scaling hypothesis implies a continuous second-order quantum phase transition [10, 13]. Near the critical energy E_c, the conductivity and localisation length scale as

$$\begin{aligned} \lambda(E) &\propto (E_c - E)^{-\nu}, \quad E \leq E_c \quad \text{(metallic phase)}, \\ \sigma(E) &\propto (E - E_c)^{s}, \quad E \geq E_c \quad \text{(insulating phase)}, \end{aligned}$$

where $s = (d-2)\nu$ is known as Wegner's Scaling Law [11].

The scaling hypothesis has been verified numerically by Mackinnon and Kramer using a recursive method to calculate the localisation lengths [14], where they show that only localised states are found in 2D and that there exists an MIT in 3D systems.

1.3 Numerical Approaches to Anderson Localisation

Randomness in the Anderson model of localisation makes analytical treatment difficult, thus many numerical methods have been applied [8]. As $d = 2$ is the lower critical dimension of Anderson localisation, the 2D problem is in a sense close to 3D. States are only slightly localised for weak disorder, so a small magnetic-field or spin-orbit coupling can lead to extended states and thus an MIT [8]. Near the MIT, large systems are required due to divergence of the localisation lengths, so computing time and memory increase dramatically. This problem therefore requires specially adapted algorithms. One can diagonalise the Hamiltonian and obtain eigenvectors which give the localisation lengths. One way of doing this is to use the Cullum-Willoughby Implementation (CWI) of the Lanczos algorithm [15, 16]. The method we use is the Transfer-Matrix Method (TMM), which considers the system as a quasi-1D bar of length L and width M such that $L >> M$. The reason why this approach is advantageous is discussed fully in Chapter 2.

1.4 Finite-Size Scaling Theory

After computing the localisation lengths for a quasi-1D system, one can extrapolate to a fully 2D/3D system using Finite-Size Scaling (FSS). This is motivated by one-parameter scaling theory, developed in 1979 by Abrahams et al [9]. As described earlier in section 1.2, it states that only one characteristic dimensionless quantity is needed to describe the critical behaviour of the system. In other words, close to the MIT at temperature $T = 0$, the conductance only depends on the dimensionless conductance g and the scaled factor b [17], like so

$$g(bL) = f(b, g(L)).$$

The idea of FSS is to scale the reduced localisation length $\lambda(M)/M$ for different disorder parameters onto a scaling curve

$$\frac{\lambda(M)}{M} = f\left(\frac{\zeta}{M}\right),$$

where ζ is a scaling parameter which is a function of the disorder [8]. By obtaining highly accurate data and using the FSS technique, one can effectively extrapolate the localisation length to infinitely sized systems (i.e in the thermodynamic limit).

1.5 Discretisation of the Single-Electron Disordered System

To find the localisation lengths numerically, it makes sense to work with a lattice model of the disordered quantum system [18]. First one takes the dimensionless Schrödinger equation in continuous form

$$H\psi(r) = V(r)\psi(r) + \nabla^2\psi(r),$$

where ψ is the wavefunction of the electron, r is the position, V is the potential energy and H is the Hamiltonian operator representing the total energy of the electron. By discretising (1.5), one can derive the Hamiltonian in lattice form [8, 19],

$$\mathbf{H} = \sum_i V_i|i\rangle\langle i| - \sum_{ij} t_{ij}|i\rangle\langle j|,$$

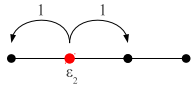
where i and j denote lattice sites, V_i is the potential energy at site i which is random and uniformly distributed in the range $[-\frac{W}{2}, \frac{W}{2}]$, where W is the strength of the disorder. The t_{ij} are the transition rates for the electron to go from one site to another, and represent the kinetic energy part of the Hamiltonian. In this model, we have nearest-neighbour hopping so that $t_{ij} = 1$ for adjacent sites and $t_{ij} = 0$ otherwise.

1.6 Construction of the Hamiltonian in the Anderson model

Starting with the 1D Hamiltonian, in figure 1.3 one can logically see how to construct the higher-dimensional analogues, by treating each diagonal disorder element as a Hamiltonian of the lower-dimensional system. For this example we have system size of $M = 4$. The 1D Hamiltonian, $H_i^{(1)}$, replaces the diagonal disorder in the 2D analogue, where the subscript i represents the 1D row. The unit transition rates are replaced with 4×4 identity matrices, $\mathbb{1}_4$. The same analogy is used to construct the 3D Hamiltonian, but this time each element is a $4^2 \times 4^2$ matrix and the Hamiltonian subscripts represent different 2D planes.

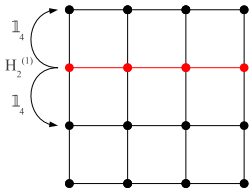
The number of lattice points in the 3D Anderson model grows as $N = L^3$, where L is the short linear dimension of the system. This means that the Hamiltonian has a size of $L^3 \times L^3 = L^6$. For a system of linear size 100, the Anderson

1D:



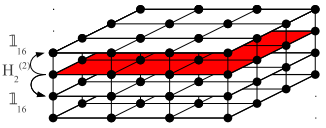
	1	2	3	4
1	V_1	1	0	1
2	1	V_2	1	0
3	0	1	V_3	1
4	1	0	1	V_4

2D:



	1	2	3	4
1	$H_1^{(1)}$	$\mathbb{1}_4$	$\mathbb{0}$	$\mathbb{1}_4$
2	$\mathbb{1}_4$	$H_2^{(1)}$	$\mathbb{1}_4$	$\mathbb{0}$
3	$\mathbb{0}$	$\mathbb{1}_4$	$H_3^{(1)}$	$\mathbb{1}_4$
4	$\mathbb{1}_4$	$\mathbb{0}$	$\mathbb{1}_4$	$H_4^{(1)}$

3D:



	1	2	3	4
1	$H_1^{(2)}$	$\mathbb{1}_{16}$	$\mathbb{0}$	$\mathbb{1}_{16}$
2	$\mathbb{1}_{16}$	$H_2^{(2)}$	$\mathbb{1}_{16}$	$\mathbb{0}$
3	$\mathbb{0}$	$\mathbb{1}_{16}$	$H_3^{(2)}$	$\mathbb{1}_{16}$
4	$\mathbb{1}_{16}$	$\mathbb{0}$	$\mathbb{1}_{16}$	$H_4^{(2)}$

Figure 1.3: Construction of the Anderson Hamiltonian from 1D to 3D, with width $M = 4$ and periodic boundary conditions. The elements on the diagrams corresponding to the ones on the tables are highlighted in red. In 1D, the hopping of the electron between sites is represented by off-diagonal unit transitions. The potential energies at each site are denoted by the diagonal elements V_i. In 2D/3D, the interactions between rows/planes are represented by $\mathbb{1}_4$ and $\mathbb{1}_{16}$, which are 4×4 and $4^2 \times 4^2$ identity matrices respectively. The $\mathbb{0}$ are zero matrices. $H_i^{(d)}$ represents the Hamiltonian for the i^{th} d-dimensional element.

matrix has size $100^6 = 10^{12}$. If one byte is used to store each element of the matrix on a computer, then one terabyte would be required. This is already larger than most modern hard-drives. Instead of attempting to solve the Hamiltonian, iterative methods such as TMM (Transfer-Matrix Method) are generally used, followed by FSS (Finite-Size Scaling). This is because in the TMM, one simulates a quasi-1D bar, using much smaller matrices than would be required for the extended system.

Chapter 2

Transfer-Matrix Method

The TMM is a numerical technique which is used for computing the localisation lengths of a disordered system. In the TMM, a quasi one-dimensional bar of length $L >> M$ is split into slices of size M^{d-1}. The Schrödinger equation is recursively applied such that the wave function at the $(n+1)^{\text{th}}$ slice, ψ_{n+1}, is computed from the $(n-1)^{\text{th}}$ and n^{th} slices, ψ_{n-1} and ψ_n. Reformulating the Schrödinger equation into a transfer matrix T_n and repeating multiplications of these matrices at each slice gives the 'global transfer matrix', Γ_n, which maps the wave functions from one side of the bar to the other. The minimum eigenvalue computed from this matrix gives the localisation length. To obtain the minimum eigenvalue and prevent numerical instabilities resulting from the exponential increase in the eigenvalues, the eigenvectors must be re-orthonormalised after every few matrix multiplications. This takes a considerable amount of time, making it crucial to efficiently parallelise the TMM code. As the disorder decreases in fully extended 2D and 3D systems, the localisation lengths become very large. One of the advantages of the TMM is that in simulating a quasi-1D system, the problem is close to that of a true 1D system (i.e. the localisation lengths are small compared to L) and the matrices used are only the size of the bar cross-section, much smaller than the full Anderson Hamiltonian of an extended 2D or 3D system. In this thesis, two types of boundary conditions for the TMM are explored. For hardwall boundary conditions (HBC), the wavefunction vanishes at the long edges of the quasi-1D bar. For periodic boundary conditions (PBC), instead of the wavefunction vanishing at one of the long edges, the value of the wavefunction amplitude at the opposite long edge is taken as the adjacent wavefunction amplitude.

2.1 The Transfer-Matrix Method: 1D

In 1D, the single particle Schrödinger equation in the lattice model is

$$\psi_{n+1} = (E - V_i)\psi_n - \psi_{n-1}.$$

In the TMM we re-arrange this equation into matrix form like so

$$\begin{pmatrix}\psi_{n+1}\\ \psi_n\end{pmatrix} = \begin{pmatrix}E-V_n & -1\\ 1 & 0\end{pmatrix}\begin{pmatrix}\psi_n\\ \psi_{n-1}\end{pmatrix} = \mathbf{T_n}\begin{pmatrix}\psi_n\\ \psi_{n-1}\end{pmatrix}.$$

The transfer matrices $\mathbf{T_n}$ transfer the wavevector amplitudes between sites $(n, n-1)$ and $(n+1, n)$. By multiplying these transfer matrices together, one can evolve the wavevectors from one end of a chain of sites to the other, as visualised in figure 2.1.

$$\begin{pmatrix}\psi_{L+1}\\ \psi_L\end{pmatrix} = \mathbf{T_L}\dots\mathbf{T_2}\mathbf{T_1}\begin{pmatrix}\psi_1\\ \psi_0\end{pmatrix} = \mathbf{\Gamma_L}\begin{pmatrix}\psi_1\\ \psi_0\end{pmatrix}.$$

Due to the symplecticity of the transfer matrices, Oseledec's theorem [20] states that the eigenvalues of $\Gamma = (\Gamma_L^\dagger\Gamma_L)^{1/2L}$ converge toward $e^{\pm\gamma}$ as $L\to\infty$, where γ is known as a Lyapunov exponent [18]. The localisation length is then determined by the inverse of the Lyapunov exponent

$$\lambda = \frac{1}{\gamma}.$$

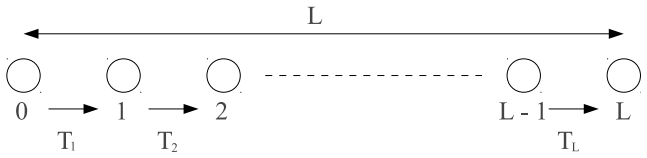


Figure 2.1: Propagation of the 1D TMM along a chain of sites.

The perturbative expansion of the localisation length for a weakly disordered 1D system is [8],

$$\lambda(E) = \frac{24(4t^2-E^2)}{W^2},$$

where transition rates are $t = 1$ for the Anderson model. This means that one should obtain $\lambda(0) = 96V^2/W^2$ except in actuality one obtains $\lambda(0) = 105V^2/W^2$. This is due to an anomaly in the band centre caused by a breakdown of second-order perturbation theory [21]. This discrepancy can be seen later in figure 6.1, where the numerical results of localisation length have been plotted against energy for constant disorder $W = 1$. The localisation lengths against disorder for constant energy $E = 0$ have also been plotted in figure 6.2. Anomalies in the localisation length are seen for disorders with low and high orders of magnitude ($W \sim 0.01$ and $W \sim 10$).

2.2 The Transfer-Matrix Method: 2D and Greater

For the 2D TMM, we consider a quasi-1D strip consisting of M chains of L sites, where $L >> M$. Using FSS [9], one can then extrapolate this quasi-1D strip to a

fully 2D system. The Schrödinger equation for a single particle in 2D is

$$\psi_{n+1,m} = (E - V_{n,m})\psi_{n,m} - \psi_{n,m+1} - \psi_{n,m-1} - \psi_{n-1,m},$$

where chain number $m = 1, \ldots, M$ and slice number $n = 1, \ldots, L$. This equation can written in vector vector form,

$$\mathbf{\Psi_{n+1}} = (E\mathbb{1} - \mathbf{H_n})\mathbf{\Psi_n} - \mathbf{\Psi_{n-1}}.$$

Analogous to the 1D TMM, this can be rearranged into matrix form

$$\begin{pmatrix} \Psi_{n+1} \\ \Psi_n \end{pmatrix} = \begin{pmatrix} E\mathbb{1} - \mathbf{H_n} & -\mathbb{1} \\ \mathbb{1} & \mathbb{0} \end{pmatrix} \begin{pmatrix} \Psi_n \\ \Psi_{n-1} \end{pmatrix} = \mathbf{T_n} \begin{pmatrix} \Psi_n \\ \Psi_{n-1} \end{pmatrix},$$

where Ψ_n is an $M \times M$ matrix. The Hamiltonian for a width $= M$ system with PBC is

$$\mathbf{H_n} = \begin{pmatrix} V_{n1} & 1 & 0 & 0 & 1 \\ 1 & V_{n2} & 1 & 0 & 0 \\ 0 & 1 & V_{n3} & 1 & 0 \\ & & & & \\ 0 & 0 & 1 & \ddots & 1 \\ 1 & 0 & 0 & 1 & V_{nM} \end{pmatrix}.$$

As in the 1D case, one takes a product of a large number transfer-matrices to obtain the localisation length.

In 2D, the eigenvalues of $\Gamma = (\Gamma_L^\dagger \Gamma_L)^{1/2L}$ converge to $e^{\pm\gamma_m}$, where γ_m are the Lyapunov exponents (one for each m). The localisation length is defined as the longest decay length given by the minimum Lyapunov exponent

$$\lambda = \frac{1}{\gamma_{\min}},$$

since this is the length within which the wave function must eventually decay.

According to Römer and Schulz-Baldes [22], a good approximation for the localisation length of a 2D (quasi-1D) system with PBC is given by

$$\lambda \approx \frac{96M}{W^2} M_e^2 \left(\sum_{l,k} \frac{2 - \delta_{k,l}}{\sin\eta_l \sin\eta_k} \right)^{-1},$$

where $\eta_l =$ rotation phase, $M =$ system width (number of channels), $M_e =$ number of elliptic channels and the sum runs over elliptic channels only. The channels (the 1D chains in the quasi-1D bar) are elliptic when $|\mu_l| < 2$, where

$$\mu_l = e^{i\eta_l} + e^{-i\eta_l} = -2\cos\frac{2\pi l}{M} - E,$$

are the eigenvalues of Δ_M, the discrete laplacian, for channels $l = 0, \ldots, M-1$ [22]. The theoretical values together with the numerical results for the 2D localisation

length have been plotted against energy in section 6.2.1.

2.2.1 Numerical Instabilities of the Transfer-Matrix Method

The problem with the TMM is that it is numerically unstable. As $L \to \infty$, Γ_L will converge towards the largest eigenvalue times its eigenvector if Ψ_1 and Ψ_0 are arbitrary. However, to compute the localisation length one must find the minimum eigenvalue. The problem is that the ratio of the smallest eigenvalue to the largest eigenvalue of Γ_L becomes comparable to machine accuracy after few matrix multiplications, meaning that the smallest eigenvalue gets lost very soon. This is because each matrix multiplication will amplify the largest eigenvalue. One can find all the eigenvalues if the orthogonality of the wavevectors is maintained, and prevent numerical instability by normalising the wavevectors. This can be achieved by starting with $\Psi_1 = \mathbb{1}$ (identity matrix), $\Psi_0 = \mathbb{0}$ and re-orthonormalising the wavevectors after every few matrix multiplications [18]. This is done using the Gram-Schmidt process (described in section 2.2.2). As the matrices are repeatedly multiplied along each step, the eigenvectors move around in the $M \times M$-dimensional symplectic space and they eventually converge towards the eigenvectors with the correct Lyapunov exponents, provided that orthonormality is retained [18].

2.2.2 Gram-Schmidt Re-orthonormalisation

The Gram-Schmidt procedure is an algorithm designed to orthonormalise a set of vectors [23]. Using the algorithm in figure 2.2a, each of the M columns vectors of $(\Psi_{n+1}, \Psi_n)^T$, v_i, is orthonormalised with respect to all the previous columns. Each vector is first normalised, and then the overlap between that vector and the previous vectors is subtracted. This process is repeated for all vectors until they become orthogonal to each other (and normalised). A visualisation of the Gram-Schmidt procedure is shown in figure 2.2b.

(a)

for $i = 1 \to M$ **do**
  for $j = 1 \to i - 1$ **do**
    $v_j \leftarrow v_j - v_i^* v_j v_i$
  end for
  $v_i \leftarrow \frac{v_i}{\|v_i\|}$
end for

(b)

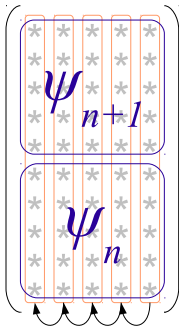


Figure 2.2: Classical Gram-Schmidt procedure. (a) shows the pseudo-code of the procedure [23], where v_i is the i^{th} column vector out of the M column vectors of $(\Psi_{n+1}, \Psi_n)^T$. (b) is a visual demonstration of the procedure implemented in the TMM.

By using this algorithm, the first column v_1 converges towards the eigenvector

with the largest e^{γ_m} (where $m = 1, \ldots, M$), the 2nd column converges toward the eigenvector with the second largest e^{γ_m}, and so on, the last column converging towards the eigenvector with the eigenvalue closest to unity, $e^{\gamma_{\min}}$.

$$\begin{aligned} 1^{\text{st}} \text{ column} &\rightarrow \text{ eigenvector with largest eigenvalue } e^{\gamma_{\max}} \\ 2^{\text{nd}} \text{ column} &\rightarrow \text{ eigenvector with } 2^{\text{nd}} \text{ largest eigenvalue } e^{\gamma_{\text{pre-max}}} \\ &\vdots \\ M^{\text{th}} \text{ column} &\rightarrow \text{ eigenvector with smallest eigenvalue } e^{\gamma_{\min}} \end{aligned}$$

$$\text{Localisation length, } \lambda = \frac{1}{\gamma_{\min}}.$$

The idea of the TMM is to perform N_{orth} transfer-matrix multiplications, followed by re-orthonormalisation. As shown later in table 6.1, the Gram-Schmidt procedure dominates computations for large M, thus N_{orth} should be as large as possible. This can be adjusted during calculation by comparing the norm of v_i leading the $\lambda_{\min}$ before and after the renormalisation [24]. If the change in norm is greater than a specific number (defined by machine precision) then N_{orth} is decreased by 1. If the change in norm is less than a specific number then it is increased. By following this procedure, N_{orth} converges fast to a number roughly in the range of 5-30 [24]. After that it only fluctuates slightly. For the computations performed for this thesis, N_{orth} has mainly been kept fixed at 10.

2.2.3 Modified Gram-Schmidt

The Gram-Schmidt procedure described above is known as the Classical Gram-Schmidt algorithm. It turns out to be numerically unstable due to the sensitivity of rounding errors on a computer [23]. A more stable version called the 'Modified Gram-Schmidt Procedure' [23] is detailed in figure 2.3. Both algorithms are equivalent.

for $i = 1 \rightarrow n$ **do**
  $v_i \leftarrow \frac{v_i}{\|v_i\|}$
  for $j = i + 1 \rightarrow n$ **do**
    $v_j \leftarrow v_j - v_i^* v_j v_i$
  end for
end for

Figure 2.3: Pseudo-code of the Modified Gram-Schmidt procedure.

However, since the Modified Gram-Schmidt procedure is more numerically stable than the Classical Gram-Schmidt, this is the one chosen for the CUDA-TMM.

2.3 Implementation of the Transfer-Matrix Method

The main program of the TMM is described in pseudo-code in figure 2.4. $N_{\max}$ sets the maximum number of iterations (matrix-multiplications) for the algorithm.

For the TMM subroutine, the actual Hamiltonian matrix is not stored in memory but encoded into the algorithm itself. The algorithm doesn't directly multiply matrices together. It calculates the bare minimum needed to effectively carry out a matrix multiplication by avoiding computing and multiplying with zeroes. The wavefunction matrix Ψ is divided into 2 arrays, Ψ_A and Ψ_B. This is so that the wavefunction at both the present and the past slices can be processed without having to swap arrays. The matrix multiplication subroutine is split into 2 steps so that at first $\Psi_n \leftarrow T_{n-1}\Psi_{n-1}$ then $\Psi_{n+1} \leftarrow T_n\Psi_n$.

The error of the Lyapunov exponent, σ, is computed so that if it is less than a specified number, σ_ϵ, the program stops (the localisation length has converged).

```
for Width = Width0 → Width1 do
  for Iter1 = 1 → N_max/N_orth (stride 2) do
    for Iter2 = 1 → N_orth do
      Ψ_B ← T_n Ψ_A
      Ψ_A ← T_{n+1} Ψ_B
    end for
    Re-orthonormalise columns of Ψ_A and Ψ_B
    Compute Lyapunov exponents γ_i
    if σ < σ_ε then
      Exit Iter1 loop
    end if
  end for
  Write data
end for
```

Figure 2.4: Pseudo-code of the main program.

The TMM was initially coded in Mathematica. This turned out to be far too slow for practical purposes. So we used FORTRAN instead, but came across many problems trying to compute the localisation lengths due to numerical instabilities. The localisation lengths are computed from the eigenvalues like so

$$\lambda = \frac{1}{\gamma} = \frac{1}{\log(e_\Gamma)},$$

where e_Γ is any particular eigenvalue of the global transfer-matrix Γ_L. Initially we kept track of the eigenvalues, but after a few hundred transfer-matrix multiplications they grew far too large/small and numerically overflowed/underflowed. So the trick was to keep track of the logarithms of the eigenvalues, instead of the eigenvalues themselves. After each renormalisation, the Lyapunov exponents γ_i were stored in

memory so that the final localisation length could be acquired by summing together all the stored gammas,

$$e^{L\gamma} = e^{\Delta\gamma_1+\Delta\gamma_2+\ldots+\Delta\gamma_M}.$$

This method was implemented in the TMM code written by Rudolf Römer [24]. The M^{th} normalisation constants of the Gram-Schmidt procedure have to be multiplied together to determine the overall normalisation of the M^{th} eigenvector, thus yielding the eigenvalue closest to unity and smallest Lyapunov exponent, which gives the localisation length. In practise, the logarithms are summed together as it is more efficient than performing multiplications.

In 3D, the critical disorder at which the MIT occurs is approximately $W_c = 16.5$ [24]. The TMM code was verified by running various disorders in both HBC and PBC, yielding critical disorders around that region (from about $W = 15.25$ to $W = 16.5$). The results are in section 6.3.

Chapter 3

Parallelisation of the Transfer-Matrix Method

3.1 Parallel Computing in General

The TMM requires highly accurate data in order for the FSS technique to faithfully represent macroscopic systems. The ever increasing need for more accurate data comes with orders of magnitude more computation. One could satisfy this need by simply creating faster processors. However, processors are already approaching their fundamental limit to clock speed. A promising approach is to use massively parallel computing, where one increases the number of processors working together on a problem instead of trying to build more powerful processors. The computational task has to be split into parts which can be done simultaneously. The subtasks on different processors might take different amounts of time, but further steps require their results, so some processors may have to wait. The subtasks usually have to exchange data. This introduces an overhead for organisation, such as starting a job on all the processors, transferring input and output to the nodes, etc. The idea is to optimise the algorithm in such a way that it can be split up into equal subtasks to be carried out on multiple processors with minimal communication between them.

3.2 Synchronisation and Race Conditions

There are times when threads (processors) will need to stop and wait for other threads to catch up, especially when they need to communicate with each other. A 'barrier' would need to be coded in the algorithm so that any thread that enounters it will stop until all other threads have encountered it.

Another problem that can occur in parallelisation is a race condition between different threads. A race condition occurs when more than one thread tries to update the same memory location. For example, say that two threads have the instruction to increment the value stored in memory location x by one. The desired net result is $x \leftarrow x + 2$. While the first thread updates the value of x, the second thread may still see the old value of x before being updated by the first thread. The net result

could either be $x \leftarrow x + 1$ or $x \leftarrow x + 2$, depending on the order and times the threads read/write to x. This problem can be solved if the threads can operate on the memory 'atomically', meaning that only one thread can operate on x at a time.

3.3 Parallelisation of the Transfer-Matrix Method

As visualised in figure 3.1, there are two simple parallelisation schemes one could adopt for the TMM: The Distributed Element Scheme (DES) and the Distributed Vector Scheme (DVS) [24]. In the DES, different elements of each column vector of Ψ are stored on different processors. During the matrix multiplication part of the TMM, adjacent processors need to communicate with each other due to site hopping in the direction perpendicular to the TMM propagation, which significantly decreases the speedup gained from running multiple processors. However, the re-orthonormalisation process can be done quite quickly as each dot-product needed for the Gram-Schmidt procedure can be calculated locally.

In the DVS, each column vector is stored on a different processor. No communication is required for the matrix multiplication part, so the speedup is proportional to the number of processors used. On the other hand, the re-orthonormalisation requires each vector to be sent to every other vector on separate processors, incurring a communication overhead.

A direct comparison of both methods has shown that the DVS scheme is faster [24]. An implementation of the TMM using the DVS scheme carried out on

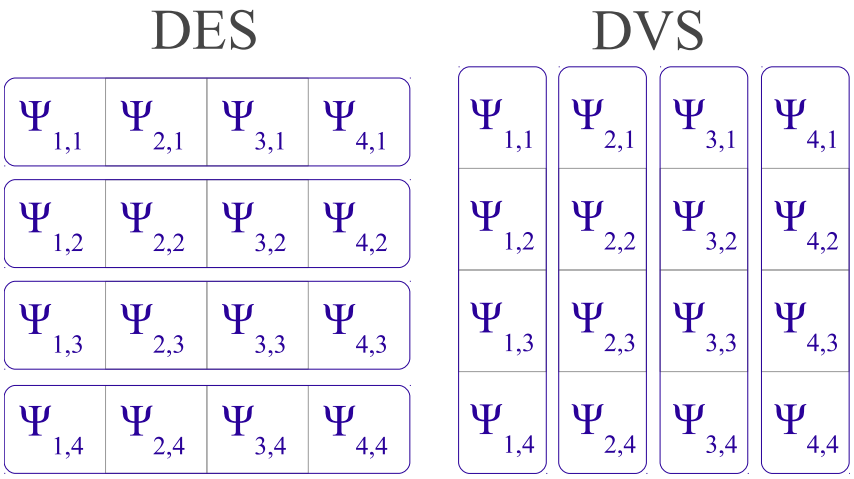


Figure 3.1: DES and DVS schemes.

a GCPP parallel computer by Römer [24] shows that while there is a net reduction of computing time for large system sizes, it doesn't scale well past 8 processors. A better parallel implementation of the TMM is desired, one such that the speedup scales close to linearly for the number of processors used.

3.3.1 Parallelised Transfer-Matrix Multiplication

In the TMM, the wavefunction at the future slice Ψ_{n+1} is given by

$$\Psi_{n+1} \leftarrow V\Psi_n - \Psi_L - \Psi_R - \Psi_{n-1},$$

where Ψ_L and Ψ_R are the adjacent wavefunctions to the left and right of Ψ_n in the present slice. During the matrix multiplication procedure, each component of Ψ only requires three components from the previous step in the multiplication process, as shown in figure 3.2. For example, to calculate $\Psi_n(i)$, one would need only $\Psi_{n-1}(i)$, $\Psi_{n-1}(i-1)$ and $\Psi_{n-1}(i+1)$ (in other words, itself and adjacent wavefunctions in the previous step). The separate column vectors j are independent of each other in this procedure, so if using the DVS scheme all calculations can take place in local memory.

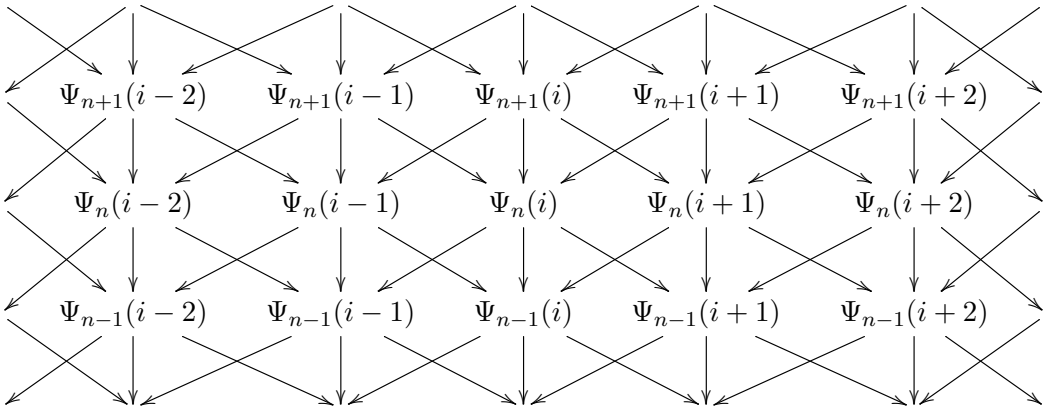


Figure 3.2: Diagram showing the dependencies of the components of Ψ throughout the transfer-matrix multiplication procedure. Since each column vector j of Ψ is independent, we simply denote $\Psi(j,i)$ as $\Psi(i)$ for clarity. The subscript denotes the step (or slice) along the TMM procedure ($n-1$ is past, n is present and $n+1$ is future). Each horizontal row represents a slice in the quasi-1D bar of the TMM (the matrix multiplication propagates upwards).

3.3.2 Parallelised Gram-Schmidt Algorithm

Figure 3.3 shows a partially parallelised implementation of the Gram-Schmidt method on the column vectors of $(\Psi_{n+1}, \Psi_n)^T$. At the j^{th} step in the algorithm, the j^{th} vector is normalised. This vector is then passed to each i^{th} vector where $i > n$. The overlap between the j^{th} and i^{th} vector is then subtracted from the i^{th} vector.

Due to the nature of the Gram-Schmidt algorithm, it is not obvious how to fully parallelise, in the sense that all parallel components run independently without waiting for each other. At the first step, all vectors are parallelised. However, after

each subsequent normalisation, the number of vectors being processed in parallel decreases by 1. As each vector resides on a different processor, the number of idle processors increases by 1 at each step, so on average only half of the processors are running in parallel at any one time.

One benefit of parallelising the Gram-Schmidt procedure is that compared to the completely serial implementation, it is more accurate because there are more numbers of the same magnitude being summed together [23].

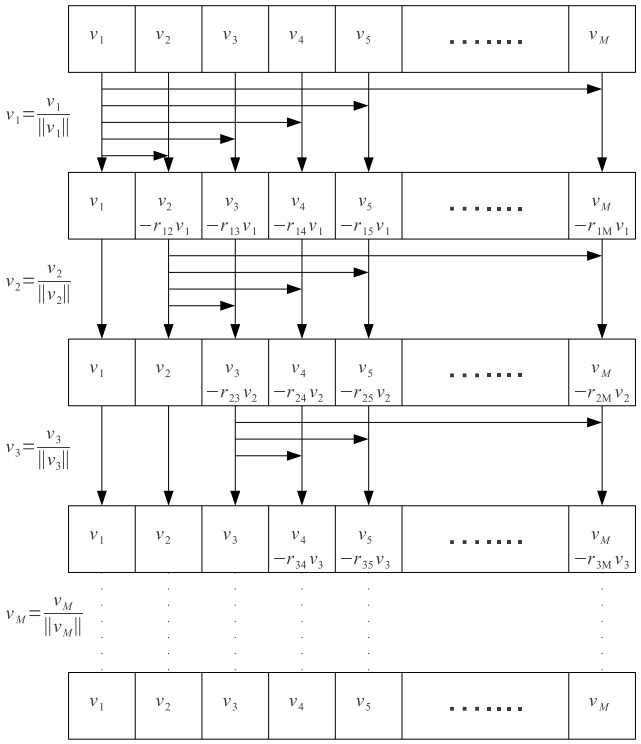


Figure 3.3: Parallel implementation of the Gram-Schmidt method with M column vectors v_i.

Milde and Schneider [25] have developed a parallel implementation of the Gram-Schmidt procedure on CUDA. However, while their method is good for orthonormalising very large vectors a single time, it is bad for repeatedly orthonormalising small vectors millions of times, which is required for the TMM. This is because their method makes extensive use of slow global memory and requires launching a new kernel for each renormalisation.

Chapter 4

NVidia GPUs and CUDA

4.1 GPUs in Scientific Computing

One method of parallelising algorithms involves using GPGPUs (General Purpose Graphics Processing Units). GPUs are specialised processing circuits built in such a way to accelerate the building of images in a computer. They have a large number of processor cores, and are ideally suited for running highly parallelised code. Unlike CPUs, GPUs have a largely parallel throughput architecture that allows many threads (processes) to be executed simultaneously. OpenCL (Open Computing Language) is an open-source programming framework for executing programs across various computing architectures such as CPUs and GPUs [26]. CUDA (Compute Unified Device Architecture) is a proprietary programming framework developed by NVidia for use on NVidia GPUs. In the CUDA model of programming, the GPU runs a 'kernel', an instance of code which each thread (processor) executes. NVidia have developed specialised GPUs for scientific computing. These are built with ECC (error correction codes) and have more double-precision cores than the standard GPUs, giving them the accuracy required by numerical scientists. For this reason, CUDA is chosen for developing a GPGPU implementation of the TMM.

4.2 The CUDA Programming Model

The CUDA programming model is based on the concept of the kernel. This is a program of which an identical copy is executed by each thread in the GPU. Each thread has its own local memory and its own set of variables and thread ID. The threads are are organised into blocks, where each block has a block ID and shared memory for inter-thread communication. Transferring data between blocks requires reading and writing to global memory which has a much larger latency than shared memory, and thus should generally be avoided if possible. The idea of CUDA is to launch as many threads as possible with little communication between them, most of the communication taking place within a block via shared memory. The algorithm must be split up into independent parts that can run separately in order to take advantage of the large number of cores in a GPU. Finally, all the blocks are

organised into a single grid as shown in figure 4.1, for which there is one per kernel launch (the latest NVidia GPUs can run more than one kernel at a time, meaning more than one grid).

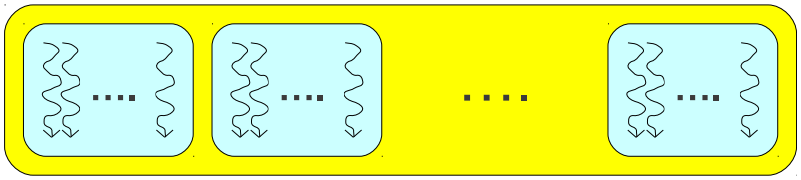


Figure 4.1: The CUDA programming model. A kernel is launched, consisting of a grid (yellow rectangle) of blocks (pale blue squares), with each block consisting of a group of threads (arrows). Each block of threads has its own shared memory, to allow fast inter-thread communication within the block. Ellipses represent repeating units.

The GPU has its processor cores organised into streaming multi-processors (SMs). The SMs of NVidia GPUs execute threads in groups of 32 called warps. Threads in the same warp run concurrently and it is advised to have threads in the same warp execute the same conditional branches of code. This means no `if` statements should be encountered by a warp of threads unless all the threads satisfy the same condition. This is because having different threads running different code will cause asynchronicity as they take different amounts to time to complete their tasks, which in turn will cause threads to wait longer at thread barriers. Putting a `syncthreads` barrier within a conditional branch is also not recommended, as it could cause the code to crash and produce erroneous results. `syncthreads` is a subroutine used to enforce synchronisation of threads within a block, and is described in section 4.2.2.

4.2.1 Advantages

As a consequence of their large parallel throughput, GPGPUs have a couple of advantages over CPUs:

- High performance per watt (servers using NVidia's Tesla M2050 GPUs consume $\frac{1}{20^{\text{th}}}$ the power of CPU based servers) [27]
- Low price to performance ratio (as above, but with $\frac{1}{10^{\text{th}}}$ the cost) [27]

4.2.2 Disadvantages

Not all algorithms will benefit from the usage of GPGPUs. One of the disadvantages is that most algorithms require a logical rethink to be parallelised in a way which is conducive to the GPU. Careful consideration will need to be made in order to take advantage of the GPU architecture. This includes the memory hierarchy described in section 4.3 (cache, registers, local, shared, constants and global memory), as well as the logical structure of the kernel described in section 4.2. There are limitations

to each type of memory that need to be considered. GPUs have a low amount of shared-memory and cache (just 16 kB for most GPUs and up to 48 kB for the latest range), and the global memory has high latency (400-800 clock cycles [28], compared to on-chip shared memory which takes roughly 30 clock cycles to access, according to micro-benchmarking tests carried out on the GPU summarised in table 4.2), so these factors will need to be considered in the algorithm development. The inherently parallel nature as well as the various programming peculiarities in CUDA make it very difficult to debug. For example, it is impossible to carry out input/output operations during the execution of the kernel. In order to print values to standard output, data has be transferred from the local/shared memory to the device global memory, then outside the kernel it needs to be transferred to the host (CPU) memory. There is no error output while a kernel is being executed, so a segmentation fault could occur without the user knowing. Ultimately some algorithms just cannot be parallelised in a way to make good use of the GPU architecture, as they are inherently serial, require a lot of inter-processor communication or are data intensive (large memory requirement). One has to be careful from the outset of parallelising an algorithm with CUDA, as it is often fraught with problems as well as the fact that it might not yield any significant speedup in the end.

Race Conditions in CUDA

In parallelisation one also has to be careful of race conditions, as discussed earlier in section 3.3. To solve these problems, CUDA has an in-built thread barrier `syncthreads`. When this subroutine is invoked, it will cause the thread to wait until all other threads in the same block have reached `syncthreads`. The problem of two threads updating the same value at the same time can be solved by using atomic operations. Such operations make sure that only one thread can update a value at a time. The drawback of using thread barriers and atomic operations is that they slow down the algorithm, since when threads are waiting at thread barriers they are not doing any useful work, and atomic operations require the usage of global memory which has a high latency.

4.3 CUDA Architecture

CUDA devices have a rich hierarchy of memory types, as shown in figure 4.2. The fastest type of memory are the registers, which are attached to each core. Each thread also has its own private local memory, which acts as a spill over for registers. The local memory is part of the device memory, having the same high latency and low bandwidth as global memory, so it is advisable to keep per-thread variables/arrays small enough to fit into the registers. For the latest NVidia GPGPUs, this is less of a problem since local memory accesses are cached like global memory is [28]. The device memory consists of DRAM in the range of 1-10 GB. Each block has its own shared memory which allows fast inter-thread communication and data sharing within a block. In most GPU devices this is limited to 16 kB, but in the latest NVidia GPU series 'Fermi', the maximum is 48 kB. Fermi devices also have a

cache hierarchy consisting of an L1 cache per SM (streaming multi-processor) and an L2 cache shared amongst all SMs. The L1 cache is accessible only to the threads in that SM, while the L2 cache can be accessed by all threads in the GPU. Data transfers from global memory to host memory (on the CPU motherboard) cannot occur during kernel execution, so the kernel must be stopped if data is to be printed or processed with the CPU. The capabilities of NVidia GPUs are summarised in table 4.1.

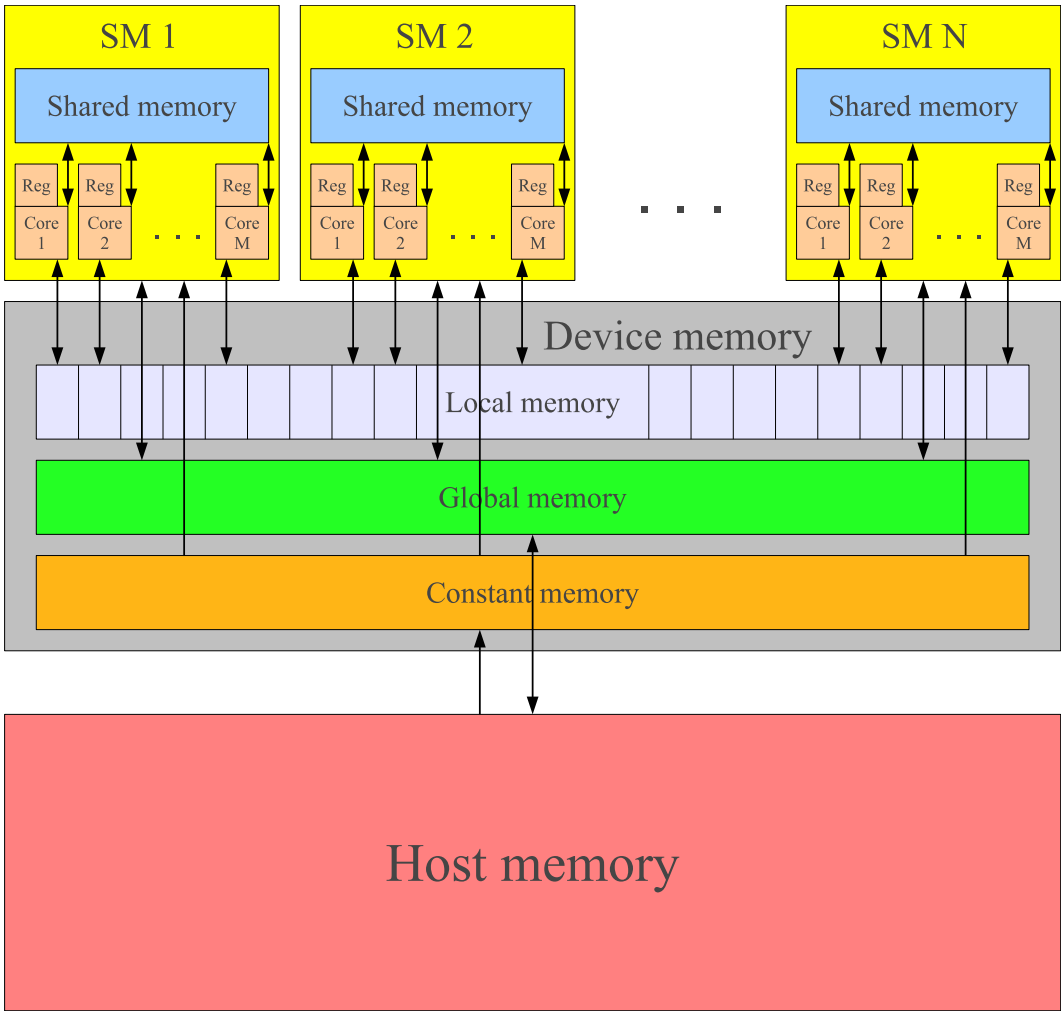


Figure 4.2: Architecture of a CUDA device with N streaming multiprocessors (SMs) each with M processor cores. The squares labeled 'Reg' are the registers. The ellipses denote repeating units. This diagram assumes a 1-to-1 mapping of SMs to blocks, however, each SM can run up to 8 blocks simultaneously each with their own shared memory.

Table 4.1: Specifications of CUDA devices with different compute capabilities [28]. The compute capability is essentially a 'version' of CUDA features that the GPU supports.

4.4 A Review of GPU Resources at the Centre for Scientific Computing

NVidia has a range of graphics cards, some made especially for scientific computing purposes. The features and specifications of the different GPUs at the CSC used to develop the CUDA-TMM are summarised in table 4.2. Most of the data for this table is taken from running the `pgaccelinfo` utility from the PGI module and from NVidia card specifications [27, 29]. The shared and global memory latencies were calculated using GPU microbenchmarking utilities written by Wong et al [30, 31]. The global memory latency is given as a range of latencies from testing different array sizes and strides. The peak single precision and double precision performances take into account all operators in all cores being used simultaneously. For single precision, this includes the multiply, addition and special function operators (these consist of transcendental functions such as sin, cosine, reciprocal and square-root [32]), which contribute to three operations per flop (floating point operation) and commonly abbreviated as MUL+ADD+SF. For double precision, fused multiply-add operations (FMA) are taken into account [32]. These are operations than can simultaneously perform a multiplication and addition, so thus contribute to two operations per flop. The error correction codes (ECC) reduce the DRAM device memory by 12.5%, so this is taken into account in the table.

4.4.1 Geforce Series

The Geforce series of graphics cards are designed for gamers. These excel at accelerating 3D graphics and in-game physics. Most of the initial debugging and testing of the CUDA-TMM code was carried out on a Geforce 9300 GE in a Linux desktop workstation.

4.4.2 Tesla 10-Series

The Tesla range of NVidia GPUs are the first dedicated GPGPU devices for use in scientific computing. The main difference between these GPUs and the standard GPUs is that they don't have a graphics port on them to use for display, being entirely used for high performance computing. The Tesla 10-series GPU used for CUDA-TMM development was a Tesla C1060.

4.4.3 Tesla 20-Series 'Fermi'

The standard 'Fermi' series GPU consists of 512 CUDA cores, organised into 16 SMs of 32 cores each (see figure 4.3). The GPUs used in the University of Warwick's

	GPU		
	Tesla M2050	**Tesla C1060**	**Geforce 9300 GE**
Compute Capability	2.0	1.3	1.1
Number of Cores	448	240	8
Number of SMs	14	30	1
Single Precision Performance (Peak)	1288 Gflops	933.12 Gflops	31.2 Gflops
Double Precision Performance (Peak)	515.2 Gflops	77.76 Gflops	None
Clock Rate	1147 MHz	1296 MHz	1300 MHz
Global Memory Size	2.62 GB (with ECC on)	4.0 GB	255 MB
Shared Memory (per SM)	48 kB/16 kB configurable	16 kB	
Max Threads per Block	1024	512	
L1 Cache (per SM)	16 kB/48 kB configurable	None	
L2 Cache	768 kB	None	
ECC Memory	Yes	No	
Concurrent Kernels	Up to 16	No	
Shared Memory Latency (Clock Cycles)	44	38	36
Global Memory Latency (Clock Cycles)	No data	505-510	551-606

Table 4.2: Summary of GPU resources available at the University of Warwick's Centre for Scientific Computing. Single/double precision performance is measured in flops (floating point operations per second). The micro-benchmarking tool [30, 31] to calculate global memory latency crashed for the Tesla M2050, which is why there is no data here.

supercomputer, Minerva, are Tesla M2050's based on the Fermi series, though they only have 448 cores instead of 512. Due to the demand from scientists for double precision, the Fermi series offers much more double precision capability than previous GPUs, with 16 double precision fused multiply-add operations per SM per clock. Tesla 20-series GPUs feature more than 500 gigaflops of double precision peak performance and 1 teraflop of single precision peak performance [27].

A new development in the Fermi series is the L1/L2-cache hierarchy, illustrated in figure 4.3. The L1-cache resides on the SM chip. Each SM has 64 kB which is configurable to either 48 kB shared memory and 16 kB L1-cache, or vice versa. The 768 kB L2-cache is shared across the whole GPU. All global memory accesses go through this cache, serving as a high speed global data-share. The Fermi can also run up to 16 kernels in parallel, which is ideal for multiple users sharing a GPU device. Another new development is that Fermi cards now have ECC (error correction codes) which are used to correct mistakes in computation caused by random bit flips.

4.5 How to Get the Most Out of GPUs

This thesis mainly discusses how the TMM can be effectively parallelised, and whether there would be a significant performance boost in using GPUs as opposed to CPUs. The challenge is in working out how to rewrite the TMM algorithm to take advantage of the high degree of parallelisation and the memory hierarchy of NVidia GPUs.

The high performance of GPUs comes from the large number of cores and low latency shared memory, so to get the most out of a GPU one needs to be launching thousands of threads. Most modern CPUs run with a clock rate of $\sim$3 GHz whereas GPUs have a clock rate of $\sim$1 GHz. The floating point unit (FPU) in a GPU also takes roughly 3 times as many clock cycles to operate as as the FPU in CPUs. So this makes GPUs roughly $\sim$10 times slower than CPUs per floating point operation. Assuming an algorithm which is compute intensive and uses negligible memory, one would need to launch at least 10 threads in a GPU kernel to get better performance than in a CPU. Unfortunately for the TMM, threads need to repeatedly communicate with each other (especially during the renormalisation routine), so in reality the number of threads required to get a performance boost over the serial implementation is much higher.

One wants to avoid 'divergent branching' which is when threads in the same warp follow different code paths. This is because it is less efficient for threads to be running at different speeds. For example, one thread may complete a task before the other and reach a thread barrier in which it must wait until the other threads in the same warp have finished. The longer threads have to wait, the more computing time is wasted.

Global memory has a very high latency (typically hundreds of clock cycles) and must be used as infrequently as possible. If it is to be used at all then it's advisable to coalesce the memory accesses. This means grouping global memory accesses together spatially and temporally so that a lot of data can be transferred in

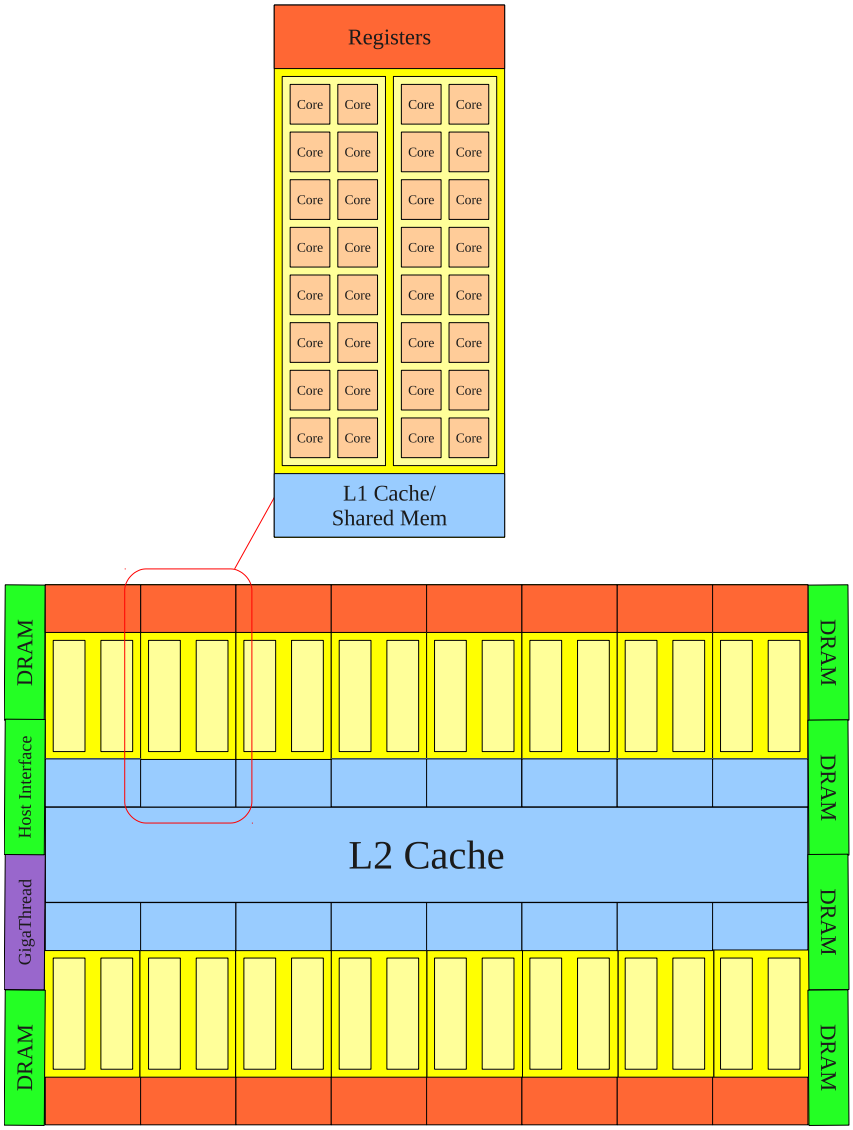


Figure 4.3: Simplified diagram of the Fermi architecture, adapted from the NVidia Fermi Compute Architecture Whitepaper [32]. The purple rectangle refers to the 'GigaThread Scheduler' which assigns blocks to SM thread schedulers.

one go, rather than fetching small portions of data regularly. One way to get around the high latency of global memory accesses is to overlap it with computational work. For example, if one warp of threads are fetching data from global memory, another warp of threads can be doing computational work to hide the latency. This insures that the GPU is always doing some computational work and not idly waiting for a few threads to finish. If a lot of operations need to be performed on data stored in global memory, then it makes sense to first copy the data into low latency shared memory, perform the bulk of the computation, then copy back to global memory.

CUDA allows up to 1024 threads per block. If there are only a few threads launched per block, one can make up for this by launching lots of blocks in parallel, in order to maximise utilisation of the GPU. It also helps to launch threads in multiples of 32 for best performance, since the GPU schedules groups of 32 concurrent threads in warps.

4.6 CUDA Algorithm Development

As the serial TMM algorithm was written in FORTRAN, it was natural to carry out code development in CUDA FORTRAN. The CUDA extension of FORTRAN was developed by The Portland Group, this means that one has to use the proprietary PGI FORTRAN compiler (`pgf90`), as opposed to the free NVidia C compiler (`nvcc`). As a consequence, there is a smaller community of CUDA FORTRAN developers than CUDA C (which makes it more difficult to find help online), and access to the CSC (Centre for Scientific Computing) at the University of Warwick is needed to compile the code.

4.6.1 First Naive Attempt at Developing the CUDA-TMM

The first attempt at converting the serial TMM code into CUDA format, using the Distributed-Vector Scheme (DVS), was naive and fraught with problems. Even running it for one dimension ($M = 1$) was on the order of 1000 times slower than running on a single CPU core. One of the reasons why it was so slow was that the wavefunctions were being stored and processed in global memory throughout the entire kernel execution, incurring a latency of hundreds of clock cycles per memory fetch.

4.6.2 Use of Shared Memory

The second version CUDA-TMM improved on the first by making use of the on-chip shared memory, which only takes a few clock cycles to access. The wavefunctions themselves were loaded from the host memory to the global device memory before the launch of the first kernel. Then during kernel execution, the wavefunctions were transferred from global memory to shared memory on which they were operated on. A kernel was launched for the transfer-matrix multiplications, then when 10 of these were finished, the wavefunctions were loaded back into global memory so that they would be saved as the kernel finished. A new kernel was then launched

to carry out the renormalisation of the wavefunctions. This algorithm was a slight improvement on the first one but still pretty slow (about 100 times slower than the CPU implementation).

4.6.3 Single Kernel launch

It became apparent that the main bottleneck was in relaunching kernels multiple times. Each kernel takes a significant amount of time to initialise, and since it takes millions of iterations for the localisation length to converge, this was contributing a huge amount of computing time. So in this version a single kernel was launched with disorder or energy parameters looped inside the kernel. Global memory was only used at the beginning and end of the kernel execution, shared memory was used for the bulk of the kernel execution. One drawback to this method is that `syncthreads` only synchronises the threads in one block. For the Tesla C1060 or Geforce, this limits the 2D system width to $M = 16$ resulting in 256 threads (one thread per element of the 16×16 matrix of Ψ), or 32 (1024 threads) for the Tesla M2050 GPUs (it will become clear in section 4.7 why only powers of 2 are permitted for the system size). This version of CUDA-TMM is significantly faster than the first version but still a lot slower than running on the CPU due to the slow clock rate and inter-thread communication.

4.6.4 Multi-Parameter Scheme

To negate some of the overhead from inter-thread communication, another level of parallelism was implemented by running separate energy/disorder parameters on each block, as shown in figure 4.4. By using a Fermi GPU, one can fit $32 \times 32 = 1024$ threads per block. In this scheme, the whole wavefunction matrix is contained within a single block, allowing widths of up to $M = 32$ in 2D systems. The TMM and renormalisation routines require synchronisation of the threads, and thus this scheme is highly efficient as threads within a block can simply be synchronised by calling `syncthreads`. There is no need to use global memory during the Gram-Schmidt procedure in this case. Part of the speedup of this scheme comes from 'naive parallelism', since each block runs as an independent system. This type of parallelism only works well for running parameters that take the same amount of time. This scheme works better if running different energy parameters for constant disorder, rather than running different disorders for constant energy. This is because changing the disorder will dramatically change the amount of time taken (due to the fact the localisation length is inversely proportional to W^2), whereas each energy takes the same time in terms of order of magnitude. By combining real parallelism across threads in one block with naive parallelism across blocks in one grid, one can achieve a speedup of up to 13.5 times that of the serial TMM, as shown in figure 6.14. Ultimately this was the scheme that was used to get the computing time results in section 6.4.2.

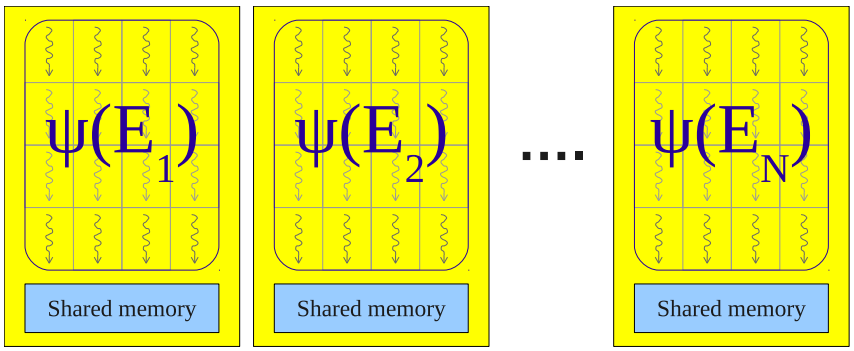


Figure 4.4: Diagram of the multi-parameter scheme (MPS). In this example, the system size is $M = 4$ and different energies E_k are run on each block where $k = 1, \ldots, N$.

4.6.5 Single-Parameter Scheme

In order to simulate 2D systems with a larger width than 32, one needs to use more than one block per wavefunction. In this scheme, as shown in figure 4.5, the wavefunction is spread out across multiple blocks such that each column vector is stored in one block. This means that the Transfer-Matrix multiplications can occur independently for each column in their separate blocks. However, during the re-orthonormalisation (the orthogonalisation part specifically) the blocks will need to communicate and synchronise with each other as each column vector is passed to the previous column vectors (Gram-Schmidt method), just like in the DVS scheme [24]. The only way to do this is to use global memory, which introduces a significant overhead to the computing time. This scheme will theoretically allow system sizes up to 1024 (i.e. 1024 blocks of 1024 threads) on Fermi devices. This is large enough to approach mesoscopic sized 2D physics (for example, graphene flakes). The global memory bottleneck can be reduced by using the fast-barrier synchronisation scheme proposed by Xiao and Feng [33].

Most CUDA forums on the internet say that one should never attempt inter-block GPU communication, because the only way to guarantee barrier synchronisation is by launching a new kernel. However, this is far too slow because a new kernel would have to be launched millions of times. According to Xiao and Feng [33], there is an effective way to implement an inter-block barrier. They have coded a `gpusync` subroutine which when combined with `threadfence` (which flushes the cache back to global memory) should act as a sufficient inter-block barrier.

The problem with this method is that it won't work unless you can guarantee that all blocks are active. You might get active blocks sitting in a loop waiting for an inactive block that will never run since there are no open SMs. What works on one device may not work on another, due to the different number of SMs. This method worked for the Tesla C1060, but it didn't work at all on the Tesla M2050 GPU, probably due to blocks waiting for inactive blocks indefinitely. For that reason, it was never implemented.

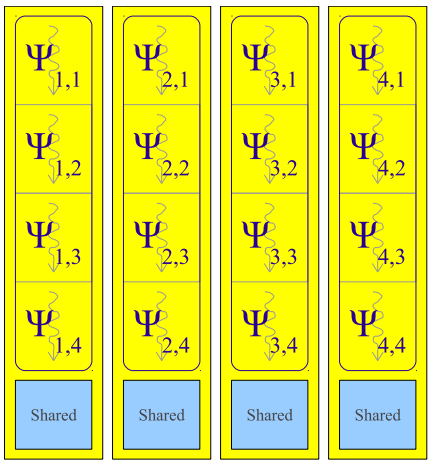


Figure 4.5: Diagram of the single-parameter scheme (SPS). In this example, the system size is $M = 4$. Different column vectors of the wavefunction are run on each block. This scheme is based on the Distributed Vector Scheme (DVS).

4.7 Shared Memory Reduction

During the normalisation and orthogonalisation routines of the TMM, each column of the wavefunction has to calculate a sum of all the elements of that column in order to compute the norm or the orthogonal overlap (in accordance to the Gram-Schmidt method described in section 2.2.2). Parallel reduction is a common and important method for summing together values of an array [34]. In this method, the sum is decomposed in a recursive tree-like way. The optimal method for performing parallel reduction in CUDA has been explained by NVidia [34] and is visualised in figure 4.6. The original array is split in half, resulting in two sub-arrays. All the elements in the second sub-array are added to the elements in the first sub-array. This process is repeated with the first sub-array, and so on until after the last step in the process, the sum of all the elements gets stored in the first element. This 'divide and conquer' approach takes $\log_2 n$ steps to complete for an array of n elements. Parallel reduction in CUDA is best performed on shared memory as it is fast and so that all the threads (within a block) can see the data. One disadvantage with this method is that it only works efficiently for system widths of $M = 2^N$ where N is an integer.

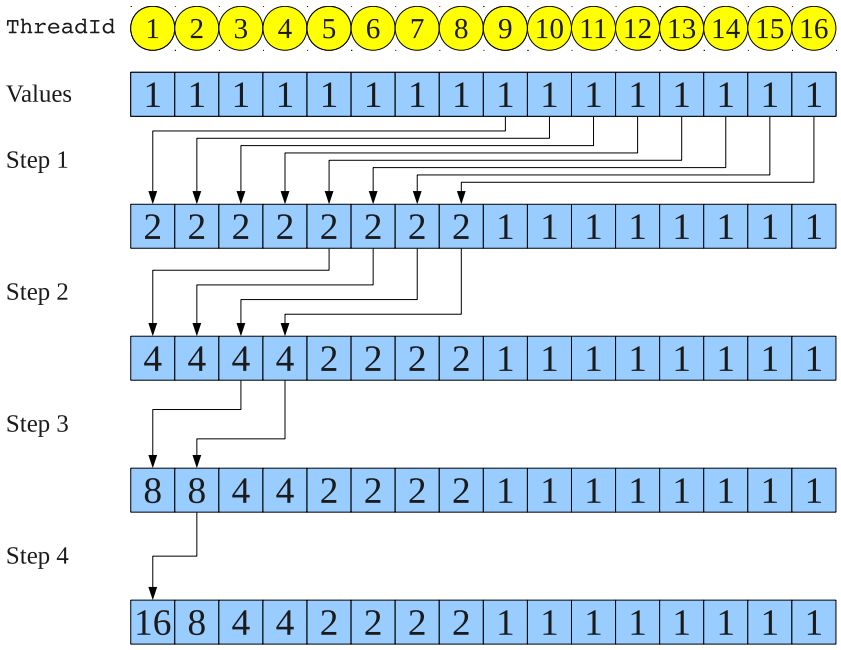


Figure 4.6: Parallel reduction using sequential addressing [34]. Arrows denote which elements of the array get summed and where the result gets stored.

Chapter 5

The CUDA-TMM Algorithm

In this chapter I will detail the inner workings of the CUDA-TMM algorithm for both MPS and SPS schemes. The Transfer-Matrix multiplication routine is fully parallelised, as explained in section 3.3.1. The re-orthonormalisation is only partially parallelised (on average $M/2$ threads run in parallel), as explained in section 3.3.2.

5.1 Master Kernel

The master kernel is the bulk of the code that each thread on the GPU executes. The pseudo-code is detailed in figure 5.1. In MPS, each block of threads executes a different energy or disorder parameter, so that each block can run independently. In this case the only inter-thread communication occurs within the blocks, via low latency shared memory. Before the kernel is launched, all the data is initialised in the host memory. Ψ_A is set to an $M \times M$ identity matrix (in the 2D case), where-as most of the other arrays are set to zero. This includes Ψ_B, the Lyapunov exponents γ, Lyapunov exponent errors σ and so on. At the beginning of the kernel execution, the indices for the shared memory array are offset, which is explained in section 5.1.1. Inside the main iteration loop (`Iter1`) another loop carries out N_{orth} matrix multiplications. For most of the results presented in this thesis, $N_{\text{orth}} = 10$. This is then followed by re-orthonormalisation of the column vectors of Ψ. After that the Lyapunov exponent (and error) is calculated by taking the logarithm of the norm. Once the error is less than a specified accuracy σ_ϵ (i.e. $\gamma_{\min}$ has converged) then the thread exits the main iteration loop. The maximum number of iterations $N_{\max}$ is set to stop the simulation for going on too long, but if σ_ϵ or $N_{\max}$ are set too low then the Lyapunov exponent may never converge under $N_{\max}$ iterations. After the data is collected, the inverse of $\gamma_{\min}$ is plotted to show the localisation length λ.

5.1.1 Array Index Offsets

In CUDA FORTRAN, only one shared memory array is allowed. Therefore, index offsets must be used to refer to different parts of the array as shown in figure 5.2, so that different objects (i.e. Ψ_A, Ψ_B, S_{norm}, etc.) can be referred to. This means that the index referring to one object must be incremented by the number of elements

offset indices for shared memory array
global memory $\rightarrow$ shared memory
for `Iter1` $= 1 \rightarrow N_{\max}/N_{\text{orth}}$ **do**
  {Carry out N_{orth} Transfer-Matrix multiplications}
  for `Iter2` $= 1 \rightarrow N_{\text{orth}}$ (stride 2) **do**
    $\Psi_A \leftarrow T_{\texttt{Iter2}}\Psi_B$
    $\Psi_B \leftarrow T_{\texttt{Iter2}+1}\Psi_A$
  end for
  {Re-orthonormalise the wavevectors column by column}
  for $v = 1 \rightarrow M$ **do**
    normalise $\Psi_{A/B,v}$
    if $j > v$ **then** orthogonalise $\Psi_{A/B,j}$ with respect to $\Psi_{A/B,j-1}$
  end for
  Calculate Lyapunov exponent
  {Exit loop if γ has converged}
  if $\sigma < \sigma_\epsilon$ **then**
    Exit `Iter1` loop
  end if
end for
shared memory $\rightarrow$ global memory

Figure 5.1: Pseudo-code of the Master Kernel. Red writing refers to comments describing what the code is doing. Global memory is downloaded into shared memory at the beginning, and at the end the shared memory is uploaded back into global memory. `Iter1` is a counter which increments each time N_{orth} matrix-multiplications followed by one re-orthonormalisation have been completed. `Iter2` is a counter which increments by two in each loop (which is what 'stride 2' refers to). The $\Psi_{A/B,v}$ refers to the v^{th} column vector of Ψ_A or Ψ_B.

taken up by the previous object. For example, if the first element of Ψ_A is referred to in the array as `shared_array(1)` and the second element as `shared_array(2)`, then the first and second elements of Ψ_B would be referred to as `shared_array(`$1+M^2$`)` and `shared_array(`$2+M^2$`)` respectively, since Ψ_A has M^2 elements. The size of the shared memory array in bytes is one of the arguments specified at the kernel launch, so this must be calculated before-hand as shown in section 5.8. The random number generator (discussed in section 5.7) uses four integers per wavevector column j, therefore $4M$ elements are required in the shared memory array.

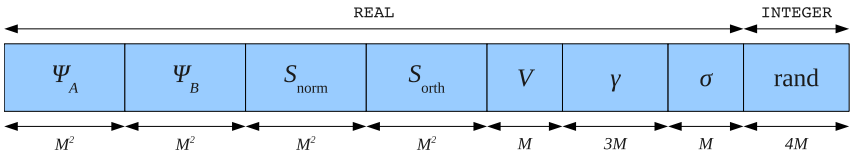


Figure 5.2: Diagram showing how the shared-memory is divided up in the MPS. Most of the array holds floating point numbers (or `REAL`s in FORTRAN), and a small part holds integers for the random number generator.

5.2 Difference Between MPS and SPS

In SPS, it is easy to map the different column vectors j of the wavefunction to different blocks and the different elements i to different threads, simply by equating j to the `blockId` and i to the `threadId`. In the MPS however, the need to contain the entire wavefunction in one block limits the use of just the `threadId` to identify the columns and rows, using the modulo and divide operations to map the `threadId`'s to different j's and i's. In this scheme, `blockId` identifies the energy or disorder parameter.

$$\text{MPS:} \begin{cases} i &= [(\texttt{threadId}-1) \mod (M)+1] \\ j &= \texttt{INT}\left(\frac{\texttt{threadId}-1}{M}\right)+1 \end{cases}$$
$$\text{SPS:} \begin{cases} i &= \texttt{threadId} \\ j &= \texttt{blockId} \end{cases}$$

For MPS, this results in the arrangement for an $M = 4$ 2D system as detailed in table 5.1.

5.3 Transfer-Matrix Multiplication Subroutine

The CUDA Transfer-Matrix Multiplication subroutine detailed in figure 5.3 is fully parallelised. The aim of this subroutine is to calculate

$$\Psi_B \leftarrow V\Psi_A - \Psi_L - \Psi_R - \Psi_B,$$

		j			
		1	2	3	4
i	1	1	5	9	13
	2	2	6	10	14
	3	3	7	11	15
	4	4	8	12	16

ThreadId

Table 5.1: Table showing the arrangement of threads in MPS.

where $\Psi_B = \Psi_{n+1}(j,i)$, $\Psi_A = \Psi_n(j,i)$, $V = V(i)$ is the potential energy of the electron, $\Psi_L = \Psi_n(j,i-1)$ is the 'left wavefunction' and $\Psi_R = \Psi_n(j,i+1)$ is the 'right wavefunction', meaning wavefunction amplitudes to the immediate left and right of $\Psi_n(j,i)$ in the present n^{th} slice. The index i corresponds to the row of the matrix Ψ, and j corresponds to the column. The `syncthreads` thread barrier is used so that each thread in a row i can see the updated value of the random potential $V(i)$. The random number generator is invoked in calculating this potential in `rand(i)` such that V is different for each i, and the same for each j. `syncthreads` is invoked to ensure that different threads in the same j^{th} column see the updated value of $V(i)$ before it is used. When calculating the left and right wavefunctions, the subroutine checks to see whether the element lies at the one of the long edges of the TMM quasi-1D bar. If HBC are being simulated, then the adjacent wavefunction amplitude is zero. Otherwise if periodic, then the adjacent wavefunction amplitude takes the value of the amplitude at the opposite edge.

5.4 Normalisation Subroutine

The CUDA Normalisation subroutine detailed in figure 5.4 is parallelised over all i. S_{norm} stores the squares of the wave function amplitudes, ready to be summed with NVidia's parallel reduction method [34] and used in the calculation of the norm. Since all the reduction is carried out within a warp of threads, no thread barriers are required here. Thread barriers are placed after the initialisation of the sum (to prepare for the parallel reduction) and before the calculation of the Lyapunov exponents γ (so that value of S_{norm} is updated before being used in the calculation).

5.5 Orthogonalisation Subroutine

The CUDA Orthogonalisation subroutine is parallelised over all i, but only partially parallelised over j. In the Master Kernel the v index is looped from 1 to M, while for each v the orthogonalisation subroutine is run for all $j > v$, so that the projections of each vector onto every previous vector can be calculated. These dot products are stored in S_{orth} ready to be summed. In accordance with the Gram-Schmidt procedure, $M/2$ columns are run in parallel on average (see section 3.3.2).

{Calculate potential energy}
$V(i) \leftarrow E - W(\texttt{rand(i)} - 0.5)$
`syncthreads`
{Calculate left wavefunction}
if $i = 1$ **then**
  if hardwall **then** $\Psi_L \leftarrow 0$
  else if periodic **then** $\Psi_L \leftarrow \Psi_A(j, M)$
else
  $\Psi_L \leftarrow \Psi_A(j, i-1)$
end if
{Calculate right wavefunction}
if $i = M$ **then**
  if hardwall **then** $\Psi_R \leftarrow 0$
  else if periodic **then** $\Psi_R \leftarrow \Psi_A(j, 1)$
else
  $\Psi_R \leftarrow \Psi_A(j, i+1)$
end if
{Calculate $\Psi_{n+1} = T_n \Psi_n$}
$\Psi_B(j, i) \leftarrow V(i)\Psi_A(j, i) - \Psi_L(j, i) - \Psi_R(j, i) - \Psi_B(j, i)$

Figure 5.3: Pseudo-code of the TMM subroutine. `rand(i)` calls the random number generator (detailed in section 5.7) and produces a `REAL` with uniform probability between 0 and 1.

{Calculate $\langle\psi_v \mid \psi_v\rangle(i)$}
$S_{\text{norm}}(v,i) \leftarrow \Psi(v,i)^2$
`syncthreads`
{Carry out sum reduction $\langle\psi_v \mid \psi_v\rangle = \sum \langle\psi_v \mid \psi_v\rangle(i)$}
if $M \geq 64$ **then** $S_{\text{norm}}(v,i) \leftarrow S_{\text{norm}}(v,i) + S_{\text{norm}}(v,i+32)$
if $M \geq 32$ **then** $S_{\text{norm}}(v,i) \leftarrow S_{\text{norm}}(v,i) + S_{\text{norm}}(v,i+16)$
$\vdots$
if $M \geq 2$ **then** $S_{\text{norm}}(v,i) \leftarrow S_{\text{norm}}(v,i) + S_{\text{norm}}(v,i+1)$
{Normalise wavevectors $|\psi_v\rangle \leftarrow \frac{\psi_v}{\sqrt{\langle\psi_v|\psi_v\rangle}}$}
$\Psi(v,i) \leftarrow \frac{\Psi(v,i)}{\sqrt{S_{\text{norm}}(v,1)}}$
`syncthreads`
{Calculate gamma}
$\gamma \leftarrow \gamma - \log(\frac{1}{S_{\text{norm}}(v,1)})$

Figure 5.4: Pseudo-code of the normalisation subroutine. This subroutine is parallelised over all i. The vertical dots represent steps in the parallel reduction, detailed in figure 4.6, where each step is of the form: **if** $M \geq 2^k$ **then** $S_{\text{norm}}(v,i) \leftarrow S_{\text{norm}}(v,i) + S_{\text{norm}}(v,i+2^{k-1})$, for integer k.

5.5.1 Orthogonalisation in the Multi-Parameter Scheme

Pseudo-code for the MPS version of the orthogonalisation subroutine is detailed in figure 5.5. The dot products for different column vectors are calculated, but because all columns reside in the same shared memory, there is only a delay of a few clock cycles (roughly 30) in fetching the wavevectors for different j's, a much lower latency than when global memory is used (roughly 500 clock cycles). This subroutine is parallelised over all i, and partially parallelised over j such that an average of $M/2$ columns are run simultaneously.

5.5.2 Orthogonalisation in the Single-Parameter Scheme

The Orthogonalisation subroutine in the SPS is a bit more complicated, as it involves inter-block communication. The pseudo-code is displayed in figure 5.6. Each column j of the wavefunction is stored on a different block. In order to calculate the projection $\langle\Psi_v|\Psi_j\rangle$ for each column, the v^{th} block needs to upload Ψ_v into global memory so that other blocks can use it. Before the other blocks can download Ψ_v into register/local memory, they must wait for the upload to finish. This requires an inter-block barrier. This is handled by the `gpusync` subroutine, using the fast-barrier inter-block synchronisation method developed by Xiao and Feng [33], which is explained in detail in section 5.6. The last block to encounter this barrier is the v^{th} block, so that after it has finished uploading Ψ_v into global memory the other blocks can carry on. As the SPS is built to handle system sizes larger than the warp

{Calculate dot product $\langle\psi_v \mid \psi_j\rangle(i)$}
$S_{\text{orth}}(j,i) = \Psi(v,i)\Psi(j,i)$
`syncthreads`
{Carry out sum reduction $\langle\psi_v \mid \psi_j\rangle = \sum \langle\psi_v \mid \psi_j\rangle(i)$}
if $M \geq 64$ **then** $S_{\text{orth}}(j,i) \leftarrow S_{\text{orth}}(j,i) + S_{\text{orth}}(j,i+32)$
if $M \geq 32$ **then** $S_{\text{orth}}(j,i) \leftarrow S_{\text{orth}}(j,i) + S_{\text{orth}}(j,i+16)$
$\vdots$
if $M \geq 2$ **then** $S_{\text{orth}}(j,i) \leftarrow S_{\text{orth}}(j,i) + S_{\text{orth}}(j,i+1)$
{Subtract projection: $|\psi_j\rangle \leftarrow |\psi_j\rangle - \langle\psi_v \mid \psi_j\rangle\langle\psi_v|$}
$\Psi(j,i) \leftarrow \Psi(j,i) - S_{\text{orth}}(j,1)\Psi(v,i)$
`syncthreads`

Figure 5.5: Pseudo-code of the orthogonalisation subroutine in the MPS. All arrays reside in shared memory.

size (32 threads), the sum reduction must use thread barriers between each step for inter-warp sums.

The main bottleneck for the SPS occurs in the `gpusync` inter-block barrier, due to the use of global memory and the fact that blocks have to wait for the global memory to finish loading. Another limitation of this scheme is that one cannot guarantee all blocks are active, meaning that during the `gpusync` subroutine, active blocks may be waiting for inactive blocks. Indeed, this method works up to $M = 32$ on the Tesla C1060 but doesn't work at all for the Tesla M2050.

5.6 The Inter-Block barrier, `gpusync`

The easiest way to communicate between different blocks in a GPU is to write the data to global memory, relaunch the kernel and then load the data back from global memory. Relaunching the kernel acts as a global barrier to enable communication between blocks. However, this incurs a significant amount of time if the kernel needs to be relaunched millions of times. When communicating between blocks on a GPU, it is important to either minimise the communication or speedup the communication itself, since this can take over 50% of the computation time [33].

An alternative to relaunching the kernel is to use the inter-block synchronisation scheme developed by Xiao and Feng [33]. Their method is encapsulated into a subroutine called `gpusync`, detailed in figure 5.7 and visualised in figure 5.8. This subroutine uses two arrays, `ArrayIn` and `ArrayOut`. In the first step, only thread 1 (from each block) is used for synchronisation. This thread sets `ArrayIn(blockId) = goalVal` and then waits for `ArrayOut(blockId)` to be set to `goalVal`. In step 2, only block 1 is used. Each thread in block 1 checks the corresponding `ArrayIn(threadId)` to see if it's equal to `goalVal`. Once this condition is satisfied, a `syncthreads` barrier is invoked and then each `ArrayOut(threadId)` is

{Load Ψ from block v to global memory}
if $j = v$ **then** $\Psi_{\text{global}}(v, i) \leftarrow \Psi_{\text{shared}}(v, i)$
{Inter-block barrier to wait for global memory to finish loading}
`sync_count++`
`threadfence`
`gpusync(sync_count)`
{Load Ψ from global memory to local variable}
if $j > v$ **then**
  $\Psi_{\text{local}} \leftarrow \Psi_{\text{global}}(v, i)$
  `syncthreads`
end if
{Calculate dot product $\langle \psi_v \mid \psi_j \rangle(i)$}
$S_{\text{orth}}(j, i) = \Psi_{\text{local}} \Psi_{\text{shared}}(j, i)$
{Carry out sum reduction $\langle \psi_v \mid \psi_j \rangle = \sum \langle \psi_v \mid \psi_j \rangle(i) \ldots$}
{... across warps}
if $M \geq 512$ **then**
  if $i \leq 256$ **then** $S_{\text{orth}}(j, i) \leftarrow S_{\text{orth}}(j, i) + S_{\text{orth}}(j, i + 256)$
  `syncthreads`
end if
if $M \geq 256$ **then**
  if $i \leq 128$ **then** $S_{\text{orth}}(j, i) \leftarrow S_{\text{orth}}(j, i) + S_{\text{orth}}(j, i + 128)$
  `syncthreads`
end if
if $M \geq 128$ **then**
  if $i \leq 64$ **then** $S_{\text{orth}}(j, i) \leftarrow S_{\text{orth}}(j, i) + S_{\text{orth}}(j, i + 64)$
  `syncthreads`
end if
{... within warps}
if $M \geq 64$ **then** $S_{\text{orth}}(j, i) \leftarrow S_{\text{orth}}(j, i) + S_{\text{orth}}(j, i + 32)$
if $M \geq 32$ **then** $S_{\text{orth}}(j, i) \leftarrow S_{\text{orth}}(j, i) + S_{\text{orth}}(j, i + 16)$
$\vdots$
if $M \geq 2$ **then** $S_{\text{orth}}(j, i) \leftarrow S_{\text{orth}}(j, i) + S_{\text{orth}}(j, i + 1)$
{Subtract projection: $|\psi_j\rangle \leftarrow |\psi_j\rangle - \langle \psi_v \mid \psi_j \rangle \langle \psi_v|$}
$\Psi_{\text{shared}}(j, i) \leftarrow \Psi_{\text{shared}}(j, i) - S_{\text{orth}}(j, 1) \Psi_{\text{local}}$

Figure 5.6: Pseudo-code of the orthogonalisation subroutine in the SPS. The local memory variable Ψ_{local} is unique to each thread. S_{orth} resides in shared memory.

set to goalVal. In the third and final step, thread 1 from each block checks to see if ArrayOut(threadId) = goalVal and once this is true, the syncthreads barrier is reached and thus this thread has caught up with all the other threads. The reason why goalVal is used is so that every time gpusync is called, one can save time by simply incrementing goalVal instead of re-initialising ArrayIn and ArrayOut to their default values. Atomic operations are used within the while-loops so that the compiler doesn't 'compile out' the otherwise empty loop.

```
{Step 1:}
if tid = 1 then ArrayIn(bid) ← goalVal
{Step 2:}
if bid = 1 then
  while ArrayIn(tid) ≠ goalVal do
    perform atomic operation
  end while
  syncthreads
  ArrayOut(tid) ← goalVal
end if
{Step 3:}
if tid = 1 then
  while ArrayOut(bid) ≠ goalVal do
    perform atomic operation
  end while
end if
syncthreads
```

Figure 5.7: Pseudo-code of the gpusync subroutine [33]. threadId and blockId have been abbreviated to tid and bid. The three steps correspond to the those visualised in figure 5.8. The atomic operation performed is d_atomic(bid,tid) = atomiccas(d_atomic(bid,tid),0,1), where d_atomic is a global memory array. atomiccas is an atomic operation which compares the first argument with the second argument, and atomically stores a new value back to the first argument location if the arguments are equal [35].

5.6.1 Performance Increase Attributed to gpusync

Xiao and Feng have compared their GPU based synchronisation to CPU based synchronisation. In the CPU explicit sync, the barrier is implemented by simply terminating the current kernel execution, using the implicit CUDA function cudathreadsynchronize and then relaunching it again. According to their tests, their GPU based synchronisation is 7.8 times faster than a CPU explicit sync. They implemented their gpusync barrier in three existing algorithms: Bitonic Sorting Algorithm, Smith-Waterman Algorithm and the Fast-Fourier Transform. The Bitonic

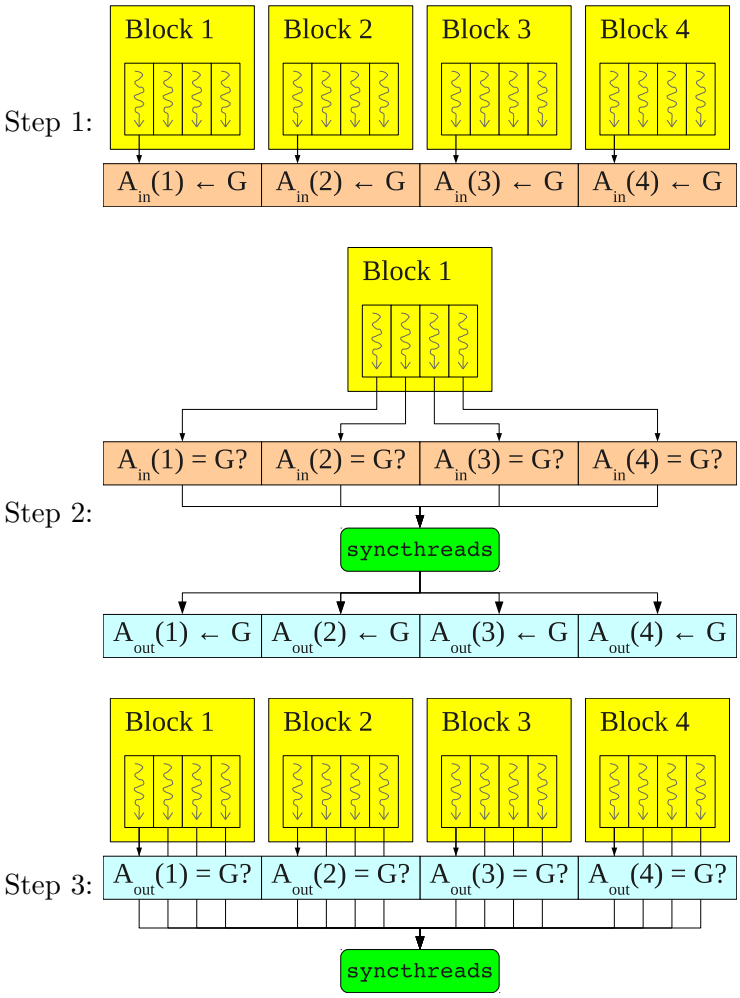


Figure 5.8: Visualisation of the `gpusync` subroutine shown in figure 5.7. `ArrayIn` and `ArrayOut` are abbreviated to A_{in} and A_{out}, `goalVal` to G. Question marks denote `IF` statements, which when satisfied allow the thread to carry on (follow the arrow).

Sorting Algorithm is a parallel sorting algorithm where $O(n \log^2 n)$ comparators are used to sort an array of n values, taking $O(\log^2 n)$ steps to complete on n processors [36]. The Smith-Waterman Algorithm is an algorithm used to find 'maximum alignment scores' between two biological sequences (e.g. protein sequences), where the segments of all possible lengths are compared to optimise this score [33, 37]. The Fast-Fourier Transform is an efficient numerical method to calculate the Discrete Fourier Transform (and inverse DFT) of a set of values, mapping values in real space to their components in frequency space (and back) [38, 39]. Compared to using the CPU implicit synchronisation, the performance increases they got were 39% in the bitonic sorting algorithm, 24% for the Smith-Waterman algorithm and 8% for the Fast-Fourier Transform [33].

5.7 Random Number Generator

For a fast and accurate TMM algorithm, one must use a pseudo-random number generator which is fast as well as reliable. The random number generator used in this case is adapted from a FORTRAN implementation [40] of an RNG suggested by Pierre L'Ecuyer in his paper of LFSR (Linear Feedback Shift Register) generators [41]. A new RNG based on this one has been written for the CUDA-TMM, fully parallelised and using shared memory for speed. A call to `rlfsr113` gives one random real in the open interval (0,1). Before using `rlfsr113` a call to `lfsrinit(seed)` must be made to initialise the generator with random integers produced with a Park/Millers minimal standard LCG (linear congruential generator). The `seed` should be any positive integer. In CUDA a separate kernel is launched before the master kernel to seed the RNG.

This random number generator is used because it's very fast (due to the use of bit-shift operations), portable (the same numbers are generated for any computer architecture) and produces random numbers with high quality statistics [41].

5.8 GPU Memory Requirements

By looking at the source code in Appendix A.3, counting the number of variables/arrays used and comparing against the memory specifications summarised in table 4.2, one can work out how much global/shared/local memory is used up as a function of system width M and number of parameters N (for the MPS scheme). Figure 5.2 also gives an indication on how much shared memory is needed in the MPS.

5.8.1 Multi-Parameter Scheme

Global Memory

The global memory required for the MPS is,

$$\begin{aligned}\text{Global memory required} = &(2M^2 + 4M + 2)N \times \texttt{SIZEOF(REAL)} \\ &+ [(4M + 1)N + M] \times \texttt{SIZEOF(INTEGER)} \\ &+ 2N \times \texttt{SIZEOF(LOGICAL)}.\end{aligned}$$

In double precision arithmetic, integers and reals are 64-bit and thus take up 8 bytes of memory (each byte is 8 bits). Logicals are truth flags, consisting of a single byte. If we take the maximum size possible to simulate (due to thread number limit) in 2D, which is $M = 32$, then the global memory requirement is,

$$\text{Max global memory required} = (17424N + 256) \text{ bytes},$$

where N is the number of energy or disorder parameters. The total amount of global memory in the Tesla M2050 device is 2.68 GB, which means that under global memory limits alone, the maximum number of parameters that can be simulated is roughly 150,000. This is greater than the maximum number of blocks that can be launched in a grid anyway, which is 65535. Therefore, this brings the number of parameters that can be simulated to 65535. It must be stressed that only a very small portion of that number of blocks can be run concurrently, due to there only being 14 SMs. Though they can't be run concurrently, the blocks are scheduled such that the GPU is constantly running jobs and not idly waiting.

Shared Memory

Shared memory is limited to 16 kB in a Tesla/Geforce GPU, or 48 kB in a Fermi GPU. This high speed memory is the most important resource to consider in CUDA-TMM, as it is much more readily used up. For MPS, the shared memory required in bytes is,

$$\begin{aligned}\text{Shared memory} = &(4M^2 + 5M) \times \texttt{SIZEOF(REAL)} \\ &+ 4M \times \texttt{SIZEOF(INTEGER)}.\end{aligned}$$

Using 32^2 threads per block in double precision, this means that the maximum shared memory required is,

$$\begin{aligned}\text{Max shared memory required} &= (4 \times 32^2 + 5 \times 32) \times 8 + 4 \times 32 \times 8 \\ &= 35072 \text{ bytes} \\ &= 34.25\text{kB}.\end{aligned}$$

As long as the Fermi GPU is configured to have 48 kB shared memory / 16 kB L1-cache instead of the other way round, then this will work. The Fermi is setup this way by default so no configuration is needed. For GPU models lower than the

Fermi range, this exceeds the maximum shared memory limit of 16 kB. However, only 512 threads can be run per block for pre-Fermi GPUs anyway, meaning that the next size down, $M = 16$, must be used, which requires only 9.13 kB of shared memory.

Local/Register Memory

In the first scheme there are 50 64-bit variables in use per thread (for double precision). For a block of M^2 threads, this means that the total memory used by local variables is,

$$\text{Register memory} = 50 \times M^2 \times 8 \text{ bytes},$$

which for the maximum of 32^2 threads, is 400 kB. This is more than 3 times the total register memory per SM, which is 128 kB, meaning that some of the local memory (which acts as register spill) will need to be used. This memory is quite slow as it resides in the device memory (i.e. DRAM). The L1-cache is too small to fit the rest of the memory, so the L2-cache will have to be used. The next size down for this scheme, $M = 16$, uses a total of 100kB, which is small enough for the registers to handle.

5.8.2 Single-Parameter Scheme

Again, looking at the code in appendix A.3 and table 4.2, one can also work out the memory requirements for the SPS.

Global Memory

$$\begin{aligned}\text{Global memory required} = (2M^2 + 4M) \times \texttt{SIZEOF(REAL)} \\ + (M^2 + 7M) \times \texttt{SIZEOF(INTEGER)}.\end{aligned}$$

Even for maximum a system size of $M = 1024$ and double precision, the total global memory used is only 24 MB. Clearly this algorithm is not memory bound for global memory.

Shared Memory

For SPS, the required shared memory is,

$$\text{Shared memory required} = 5M \times \texttt{SIZEOF(REAL)} + 4M \times \texttt{SIZEOF(INTEGER)}.$$

The maximum required shared memory for this scheme ($M = 1024$, double precision) is therefore 72 kB. This exceeds the shared memory limit on current GPU models. This leaves two options: either reduce system size by half to $M = 512$ or use single precision arithmetic instead of double precision, so that the shared memory requirement is 36 kB.

Local/Register Memory

In SPS, 41 local variables are used per thread. Therefore, the total register/local memory required is,

$$\text{Register memory required} = 41 \times M \times 8 \text{ bytes.}$$

For a maximum of $M = 1024$, this yields 328 kB. If the system size is halved to $M = 512$, the required memory is 164 kB, exceeding the register limit by just 36 kB. Taking an L1-cache of 16 kB would reduce this further to 20 kB. There are probably redundancies with the number of variables used (particularly those shared by all the threads), so with further work on the SPS version of CUDA-TMM, it should be possible to run a 2D system size of $M = 512$.

Chapter 6

Results

6.1 Plots of the Localisation Length for the 1D TMM

In figure 6.1 the localisation length has been plotted as a function of energy for a disorder of $W = 1.0$. The energy band goes from -2.5 to 2.5. This energy range is chosen due to the Gershgorin circle theorem [42] which states that the eigenvalues of a sparse, symmetric, real matrix of dimension $N^d \times N^d$ lie in $[-W/2-2d, +W/2+2d]$, where d = dimension and W = disorder strength. The numerical results are in good agreement to the 1D analytic formula from the localisation review paper [8], except near $E = 0$. The anomalous fluctuations near the centre of the energy band can be explained by the breakdown of second-order perturbation theory at $E = 0$ [21].

In figure 6.2, the localisation length is plotted as a function of disorder for an energy of $E = 0$. Both the perturbative expansion ($\lambda = 96/W^2$) and the centre band correction of the localisation length ($\lambda = 105/W^2$) have been plotted alongside for comparison. The anomaly at low disorder is due to numerical instability as the localisation length diverges. The anomaly at high disorder is due to the fact that the theoretical localisation length is only valid for weak disorder.

6.2 Plots of the Localisation Length for the 2D TMM

6.2.1 Changing Energy for Constant Disorder

In figure 6.3a, the localisation length has been plotted against energy for a disorder of $W = 1$. Both hardwall (HBC) and periodic boundary conditions (PBC) are presented, alongside the perturbative formula for the localisation length of a 2D system with PBC [22]. The numerical results for PBC are well correlated with the perturbative formula, but not so well in the centre of the band. In figure 6.3b, another simulation was carried out to see if increasing the accuracy (changing σ_ϵ from 1% to 0.1%) would allow the localisation lengths to converge closer to the theoretical values, but it made no difference. Results were computed on the Tesla C1060 and M2050.

The 2D localisation lengths for various system sizes have been plotted for HBC and PBC in figures 6.4 and 6.5 respectively.

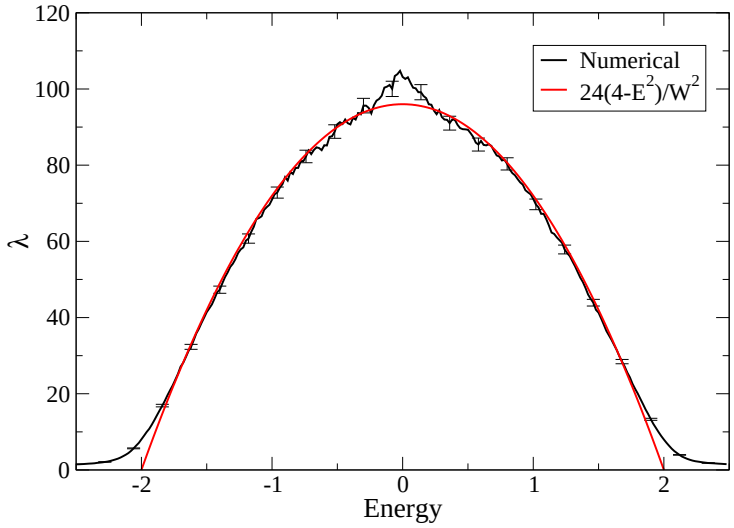


Figure 6.1: Localisation length against energy E for disorder $W = 1.0$. The red line denotes the perturbative expansion formula for the localisation length in 1D. The black line denotes the numerical results obtained from the TMM. The error bars show variations of 2 standard deviations from the average. These are shown on every 10^{th} datapoint for clarity.

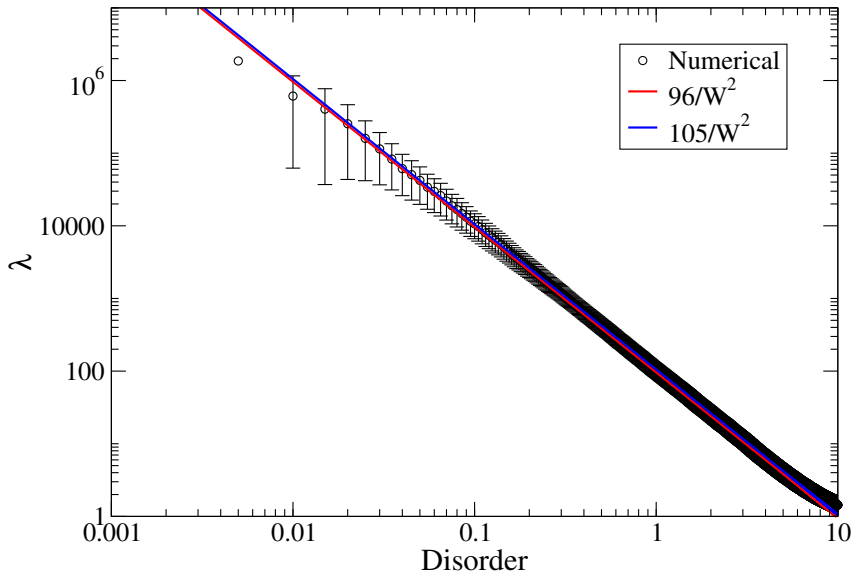
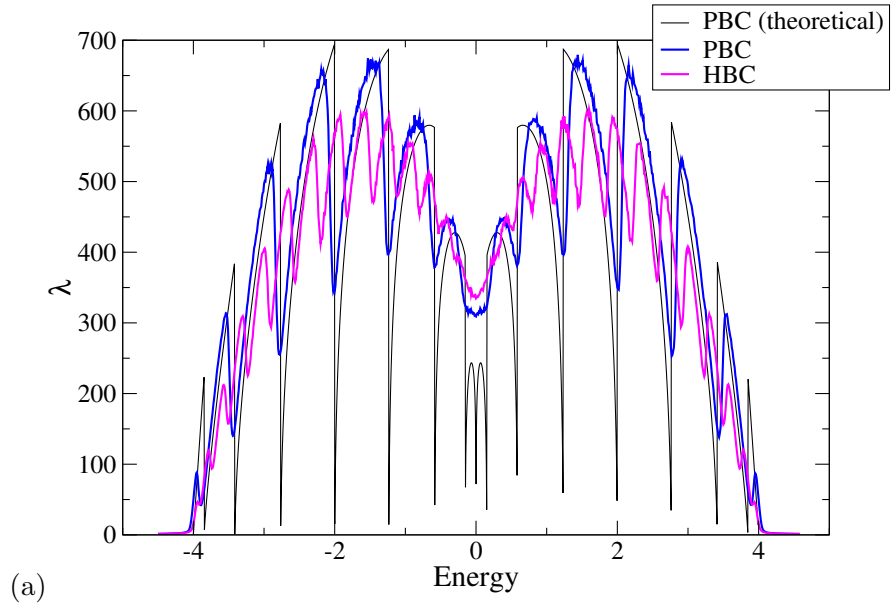
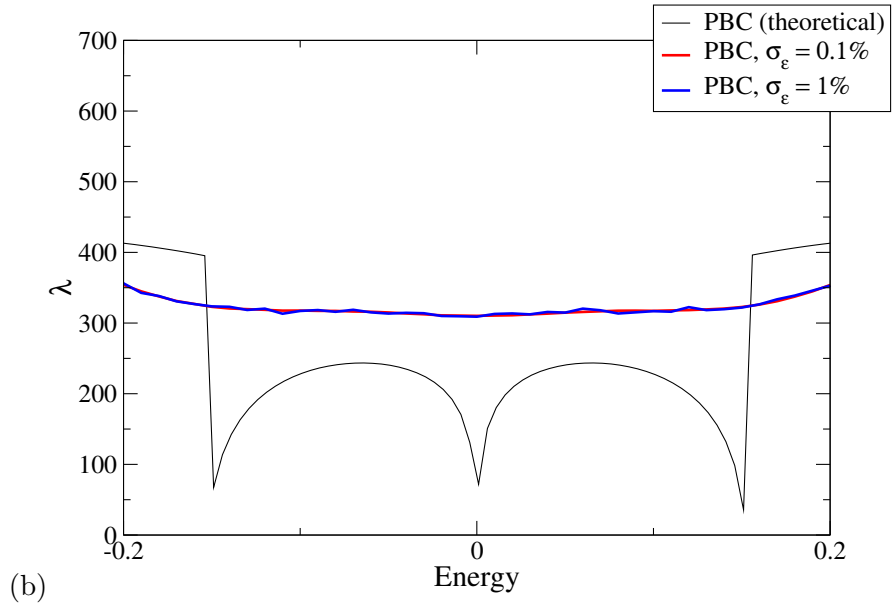


Figure 6.2: Localisation length against disorder in one dimension for energy $E = 0$. The red circles denote the perturbative expansion formula for the 1D localisation length. The blue line denotes the centre-band correction and the black line represents the numerical results from the TMM.



(a)



(b)

Figure 6.3: Localisation length against energy for a 2D system with $M = 16$, energy step $\Delta E = 0.01$, $\sigma_\epsilon = 1\%$, and $W = 1.0$. In 2D the energy band ranges from $-4 - W/2$ to $4 + W/2$. (b) is zoomed in on the central energy region $E = -0.2$ to 0.2.

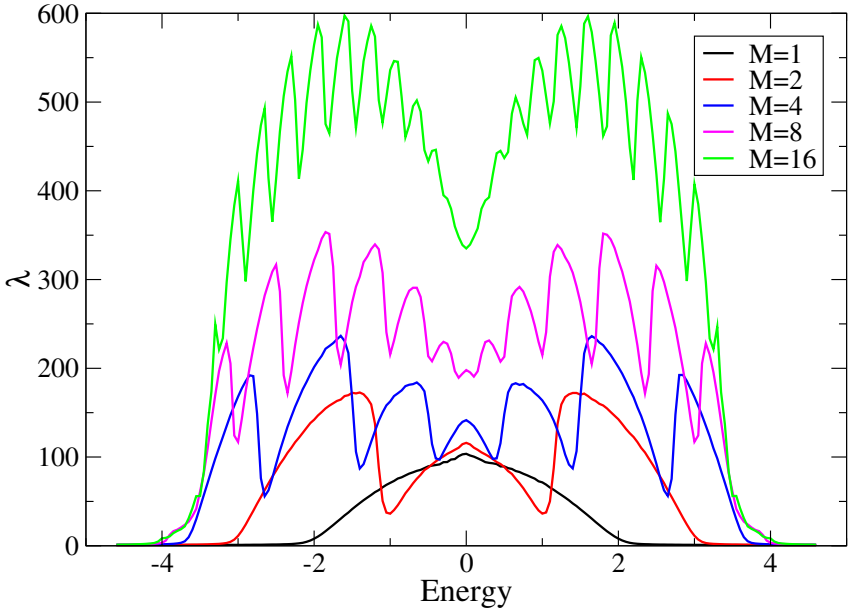


Figure 6.4: Localisation length against energy for a 2D system with various system sizes M, $\Delta E = 0.05$, $\sigma_\epsilon = 0.5\%$, $W = 1.0$, and HBC, computed on a Tesla C1060.

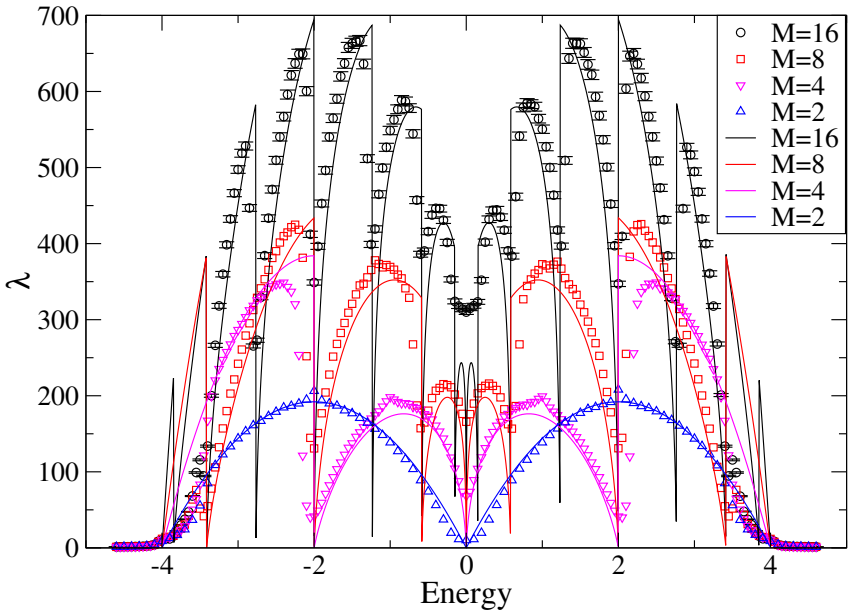


Figure 6.5: Localisation length against energy for a 2D system with $M = 2, 4, 8, 16$, $\Delta E = 0.05$, $\sigma_\epsilon = 0.5\%$, $W = 1.0$, PBC, computed on a Tesla M2050. The numerical results (symbols) are compared against the theoretical localisation lengths (lines).

6.2.2 Comparing Results of the Serial-TMM against the CUDA-TMM

The localisation length is plotted against the number of re-orthonormalisations (for each one there are 10 TM-multiplications) in the Serial-TMM and CUDA-TMM implementations, for hardwall (figure 6.6a) and periodic (figure 6.6b) boundary conditions. The disparity of results between the two implementations is due to the random numbers used by the TMM. Both serial and CUDA versions use the same random number generator, but due to the parallelisation of transfer-matrix multiplication subroutine, the order at which the random numbers get used is different. However, both implementations converge to the same localisation length (well within the error bars of 0.5% accuracy).

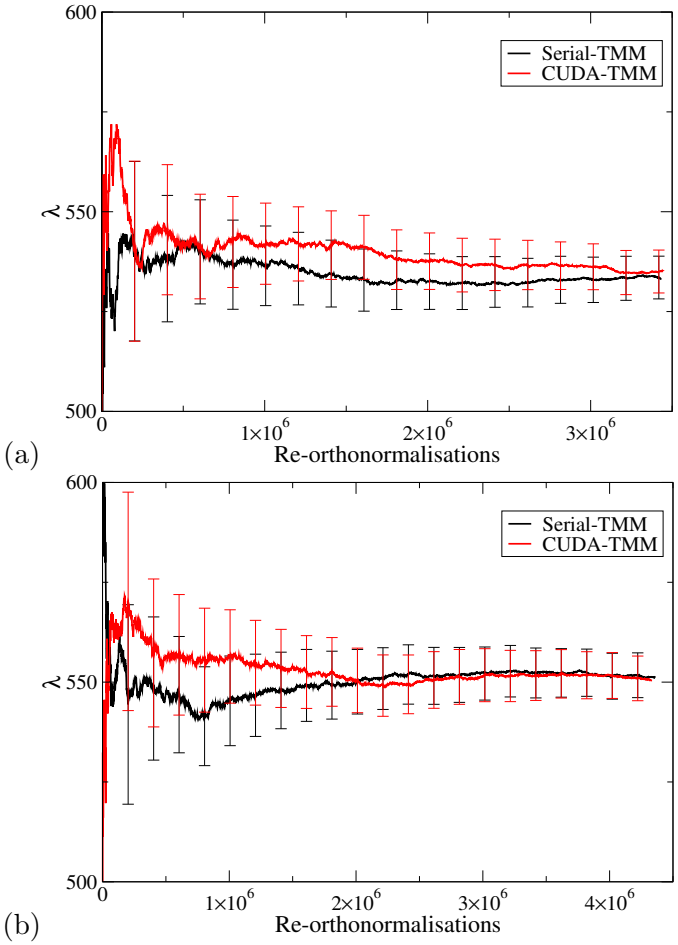


Figure 6.6: Localisation length for $M = 16$, $E = 1$, $W = 1$, $\sigma_\epsilon = 0.5\%$, with (a) HBC and (b) PBC. The black line denotes Serial-TMM runs on the CPU, and the red line denotes CUDA-TMM runs on the GPU.

The localisation lengths for a system width of $M = 8$ are plotted, with HBC in figure 6.7a and PBC in figure 6.7b. Both serial and CUDA implementations

give roughly equal results (within symbol size) except at the outermost peaks where the CUDA values are almost half the size of the serial values with HBC, and only a quarter the size of the serial values with PBC. The theoretical values are included in figure 6.7b which show that the serial implementation is more accurate. Initially it was not quite clear why the serial implementation is more accurate than the CUDA implementation. Another CUDA simulation was carried out with $N_{\text{orth}} = 2$ instead of 10, and as figure 6.8 shows, the serial values are recovered. The reason why the CUDA-TMM needs to have a smaller number of matrix-multiplications per renorm for the highly localised regime is uncertain, but it may have something to do with the difference between the double precision standard of the CPU and GPU.

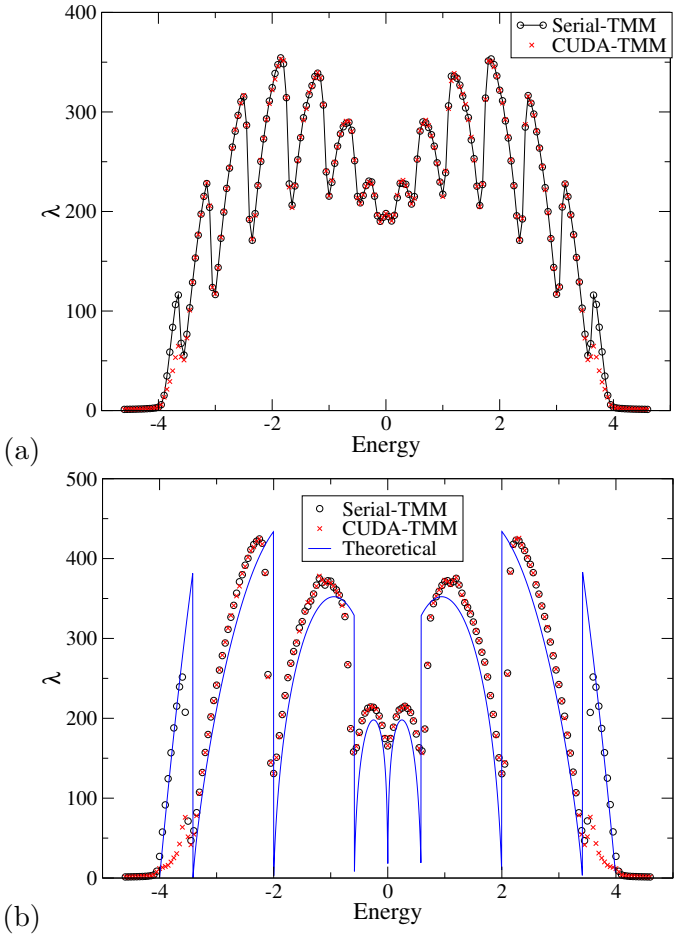


Figure 6.7: Localisation length for $M = 8$, $\sigma_\epsilon = 0.5\%$, $\Delta E = 0.05$, for (a) HBC and (b) PBC. In (b) the theoretical values are included as well (blue line). Error bars have been omitted due to being smaller than the symbol size.

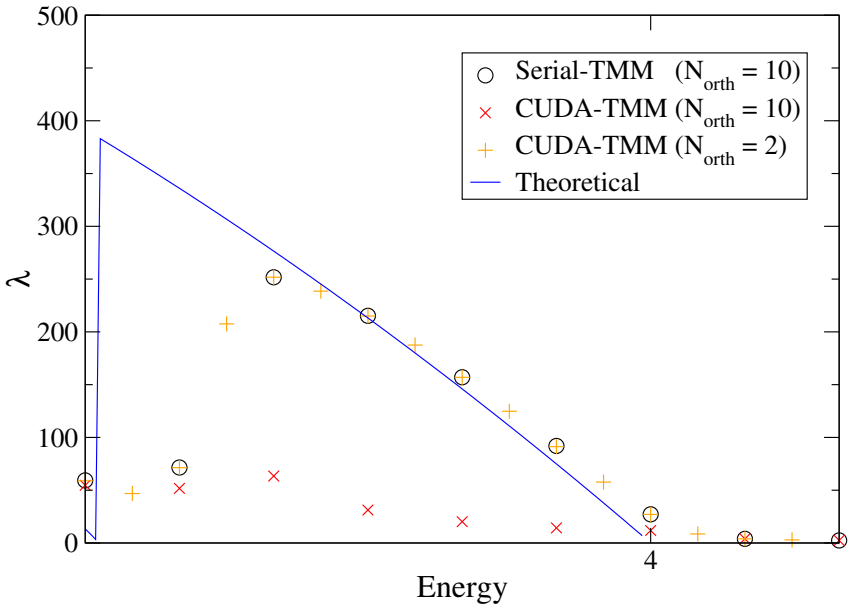


Figure 6.8: Localisation length for $M = 8$, $\sigma_\epsilon = 0.5\%$, $\Delta E = 0.05$, for PBC, as in figure 6.7b but zoomed in on the outermost peak. The black circles denote the Serial-TMM results. The red crosses denote the CUDA-TMM results. The orange pluses represent CUDA-TMM after changing N_{orth} to 2 instead of 10.

6.3 Verification of the 3D Metal-Insulator Transition

To verify the ability of the TMM to find the disorder-induced MIT, the Cluster of Workstations (CoW) was used to run the Serial-TMM code for several disorder parameters. This is a distributed computing network of all the desktops in the University of Warwick's CSC (Centre for Scientific Computing). If the reduced localisation length, λ/M, decreases for increasing M, this means that eventually the localisation length will fit within the quasi-1D bar and the system will therefore be insulating. If λ/M increases with M, this means that by increasing the system size one will drive it to a metallic state. In figure 6.9, reduced localisation length curves for different system sizes have been plotted, which demonstrate an MIT at the point where the curves cross (in the region of $W = 15$ to 16.5). The results have been grouped into odd/even system sizes and HBC/PBC. This is because the discretisation of the wavefunction has an effect on how the wavefunction gets sampled. This problem is demonstrated in figure 6.10. Here one looks at a single slice of a quasi-1D system with HBC. The wavefunction of the electron for such a system would be a sine function (so that it vanishes at each end of the slice). If we assume the wavelength is equal to M, and M is even, then the only sites at which the wavefunction vanishes are the first and last. If M is odd, then the site at the centre of the slice also samples a wavefunction value of zero.

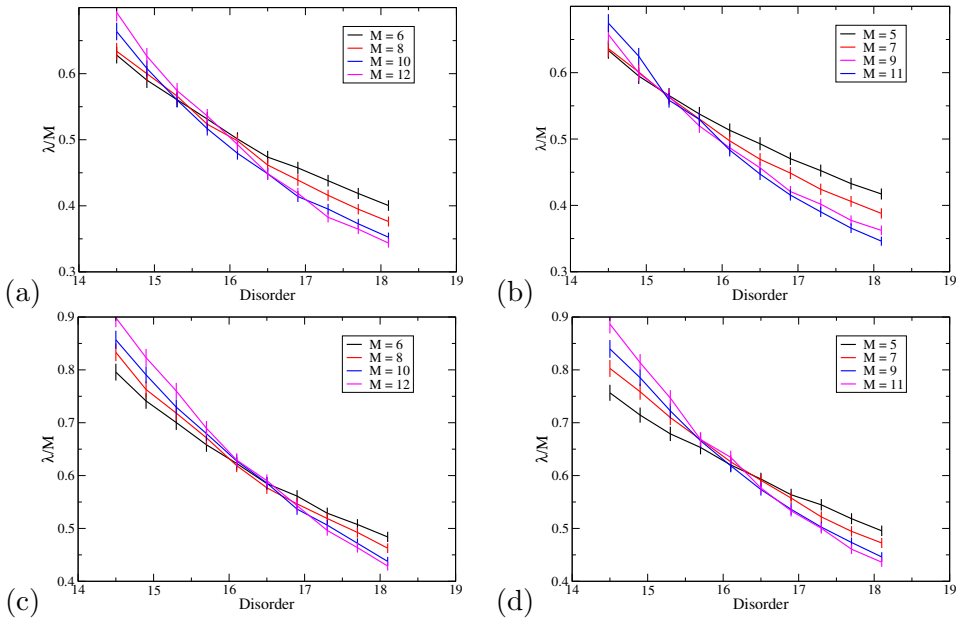


Figure 6.9: Plots of reduced localisation length at energy $E = 0$ against disorder W for different system sizes M at the 3D disorder-induced MIT. (a) and (b) use HBC, (c) and (d) use PBC. (a) and (c) have even system widths, (b) and (d) have odd system widths.

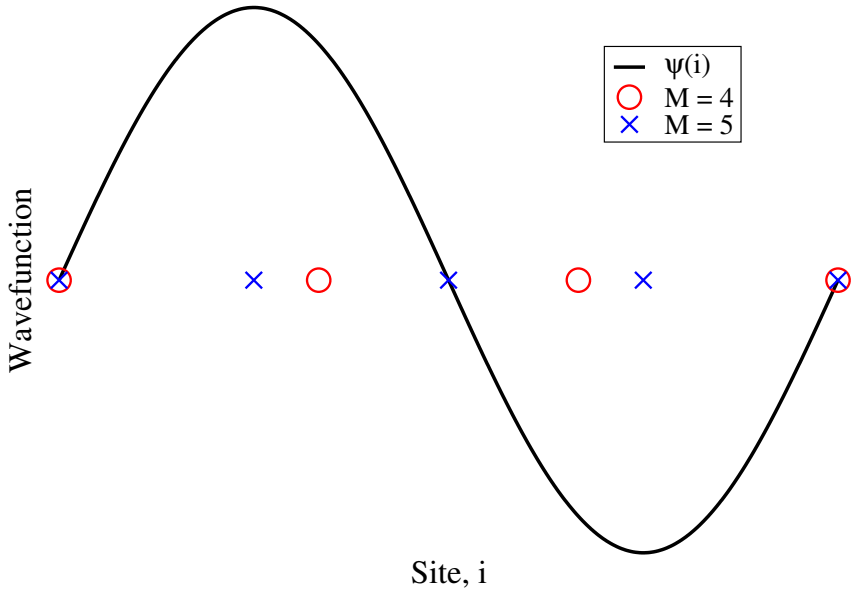


Figure 6.10: Diagram of a single slice of a quasi-1D system with HBC. The circles represents the sites of a even numbered ($M = 4$) system width, while the crosses respresent those of an odd width ($M = 5$).

6.4 Computation Times

6.4.1 Serial Scaling of Computing Time for the 3D TMM

To demonstrate the need to efficiently parallelise the TMM, serial computing times for various systems sizes and disorders have been plotted in figure 6.11. The number of iterations L needed to compute the localisation length to required accuracy scale as M^7, independent of whether the 3D system is metallic ($W = 15$), insulating ($W = 18$) or critical ($W = 16.5$). By doubling the system width, the computing time is increased 128 times.

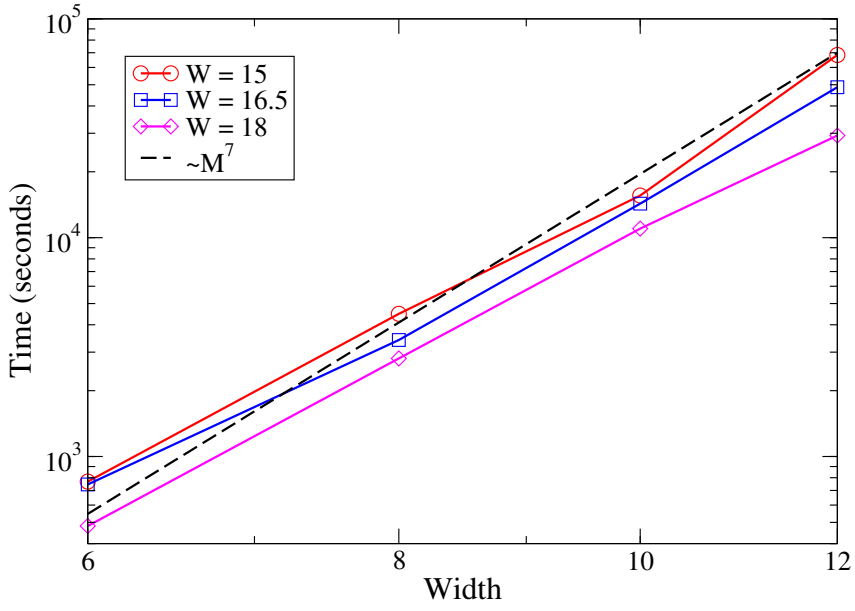


Figure 6.11: Serial-TMM computing time against system width $M = 6, 8, 10, 12$ for different disorders W. System is 3D with PBC and even widths. The critical disorder $W_c = 16.5$ is plotted alongside smaller and larger disorders $W = 15$ and $W = 18$.

6.4.2 Serial-TMM vs CUDA-TMM

The time taken to compute the 2D localisation lengths for a range of energies from $E = -4.6$ to 4.6 with disorder $W = 1$ has been recorded for different system sizes, boundary conditions and energy intervals in both Serial and CUDA implementations (using the MPS for CUDA-TMM). These computing times have been summarised in figure 6.12. For $\Delta E = 0.05$ there are 185 separate energies, for $\Delta E = 0.005$ there are 1841. To obtain the $\Delta E = 0.005$ timings for the Serial-TMM, the results for $\Delta E = 0.05$ were taken and multiplied by 10 (due to lack of time and that fact that there are 10 times as many parameters to simulate).

Another batch of HBC simulations was carried out for a weak disorder of $W = 0.1$, where the computing times have been plotted in figure 6.13. This shows

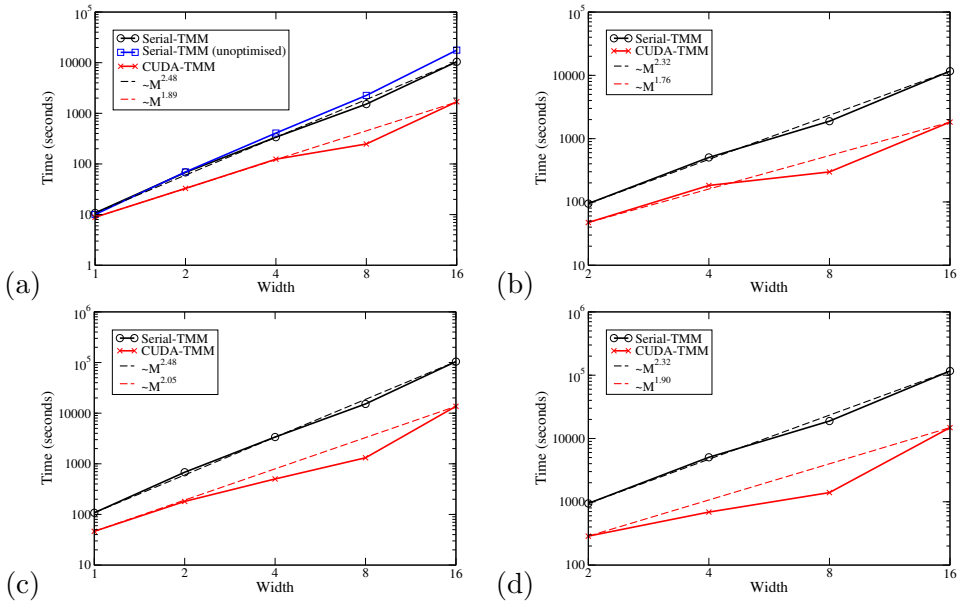


Figure 6.12: Computing time against system width M for multiple energies $E = -4.6$ to 4.6, $\Delta E = 0.05$, $\sigma_\epsilon = 0.5$ and disorder $W = 1$. System is 2D. (a) and (b) are computed with $\Delta E = 0.05$, (c) and (d) with $\Delta E = 0.005$ so there are 10 times as many energies. (a) and (c) have HBC while (b) and (d) have PBC. In (a) the unoptimised Serial-TMM runs have been included (by removing the optimisation flags when compiling the source code).

that for $M = 1$ (1D TMM) even when running 185 parameters in parallel (though not all 185 parameters can run concurrently) the Serial-TMM implementation is faster. The disorder is 10 times less than the previous simulations, so the localisation length is 100 times longer (because $\lambda \propto 1/W^2$). In this case the only parallelism at work in the CUDA-TMM is by running the separate energies in different blocks (i.e. naive parallelism), but it's not enough to make it faster than the Serial-TMM.

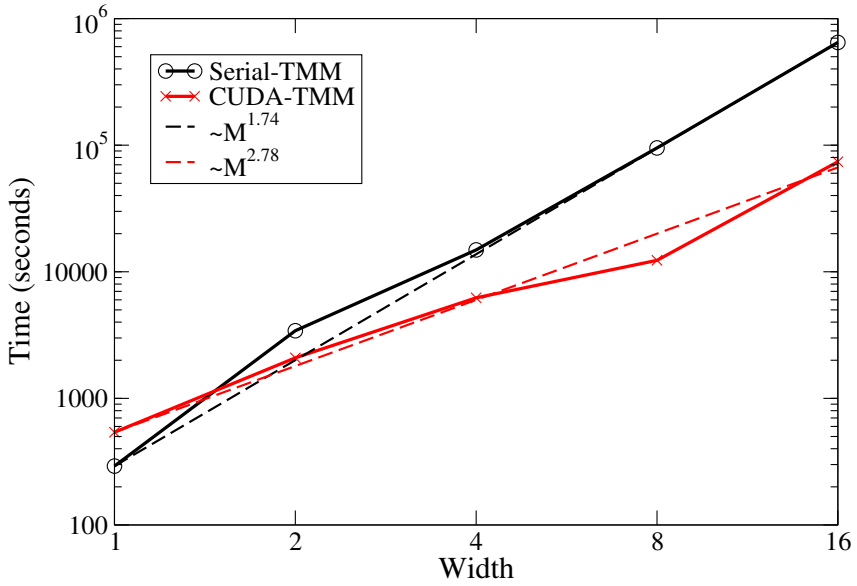


Figure 6.13: Computing time against system width M for multiple energies $E = -4.6$ to 4.6, $\Delta E = 0.05$, $\sigma_\epsilon = 5\%$ and weak disorder $W = 0.1$. System is 2D with HBC.

The speedup obtained by using the CUDA-TMM instead of the Serial-TMM has been summarised in figure 6.14. The greatest speedup achieved is for $\Delta E = 0.005$, $\sigma_\epsilon = 0.5\%$, $W = 1$, PBC, where the CUDA-TMM is about 13.5 times faster than the Serial-TMM. The reason for the large drop in the speedup for $\Delta E = 0.005$ is due to the fact that the shared memory is being saturated. This is less of a problem for $\Delta E = 0.05$ since there are 10 times less number of blocks being launched.

6.5 Profiles for the Serial-TMM

The profiles in table 6.1 show the numbers of calls and times taken to run the `TMMult3D` and `Renorm` subroutines for different widths and disorders in the Serial-TMM. In this case, HBC are used, $E = 0$ and $\sigma_\epsilon = 0.5\%$.

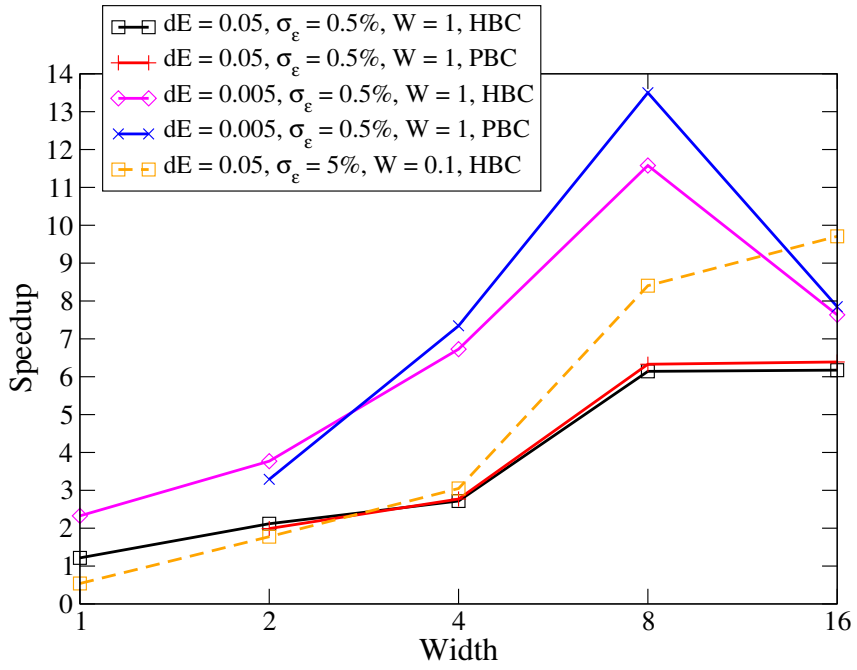


Figure 6.14: Speedup of the CUDA-TMM over the Serial-TMM for different energy intervals, accuracies and boundary conditions. All curves but one represent a disorder of $W = 1$, where as the dashed line represents a weak disorder $W = 0.1$. Open symbols represent HBC, closed symbols represent PBC. Squares and pluses denote $\Delta E = 0.05$ while diamonds and crosses denote $\Delta E = 0.005$.

Width	Disorder	Subroutine	Calls	% Time	Time (secs)
5	15	`TMMult3D`	99,550	16.97%	2.81
		`Renorm`	9,955	22.28%	3.69
	16.5	`TMMult3D`	80,100	18.92%	2.54
		`Renorm`	8,010	20.02%	2.68
	18	`TMMult3D`	64,960	16.04%	1.72
		`Renorm`	6,496	22.00%	2.36
8	15	`TMMult3D`	165,320	10.68%	31.04
		`Renorm`	16,532	38.08%	110.66
	16.5	`TMMult3D`	122,600	11.01%	23.74
		`Renorm`	12,260	38.08%	82.12
	18	`TMMult3D`	97,560	10.69%	18.32
		`Renorm`	9,756	37.86%	64.89
16	15	`TMMult3D`	367590	23.68%	1174.79
		`Renorm`	36759	52.59%	2609.23
	16.5	`TMMult3D`	240920	19.69%	771.30
		`Renorm`	24092	53.83%	2108.87
	18	`TMMult3D`	163710	9.51%	532.60
		`Renorm`	16371	60.16%	3370.06

Table 6.1: Profiles of the 3D Serial-TMM for various widths and disorders.

Chapter 7

Discussion and Conclusion

There is still much work to be done on the CUDA-TMM. For example, a 3D version of the CUDA-TMM has not yet been developed. This would probably involve using the SPS scheme. The largest width possible in this case would be $M = 32$, i.e. 32^2 blocks of 32^2 threads. The wavefunction matrix would have 1024 columns of 1024 elements each, with the same thread count as an $M = 1024$ 2D system.

Getting the SPS scheme to work on the Tesla M2050 is itself another project that needs undertaking. The various difficulties and problems debugging, testing and developing CUDA-TMM have limited the time available to get the SPS working. These problems range from code debugging, unavailability of resources, lack of coherent graphics/CUDA drivers, etc.

In CUDA FORTRAN it is impossible to have more than one shared array, which makes the programming more complicated as one has to refer to different objects in shared memory with index offsets (as shown earlier in figure 5.2). Also, CUDA FORTRAN is quite new and is still under heavy development. As a consequence, there is a much smaller community of developers than for CUDA C, making many problems encountered during code development new to the CUDA FORTRAN community.

There has been a lack of readily available GPU facilities for most of the year. This has made it difficult to debug/test CUDA code. Although each desktop workstation at the CSC has an NVidia graphics card, it cannot be used for more than roughly ten seconds at a time while the GUI is engaged, as the X windows server will time out. The Tesla C1060 is a personal GPU belonging to an academic member of staff which is not always available to use (especially for long simulations). Finally, the new array of Tesla M2050 cards installed on the Minerva supercomputer were not available to use until the last few of months of writing this thesis. These issues combined have slowed down the development of CUDA-TMM.

The act of debugging CUDA code is fraught with its own problems. Sometimes just probing the code can cause it to break. For example, if one wanted to display the values of Ψ at various stages in the algorithm, the data would have to be copied to multiple global memory arrays so that it can be transferred to host memory for printing. By turning these 'debug arrays' on or off, the code will act differently. This problem coupled with the highly parallelised nature of CUDA turns

debugging into a art that must be mastered to in order to fix just a small part of the kernel.

On top of all these problems were various system side faults that were not under my control. For example, the CSC desktops use a centralised network filesystem to ease the installation of new drivers and kernel updates to all desktops. Whenever the graphics drivers get updated, the CUDA code will stop running as the CUDA version is mismatched with the graphics driver version. Sometimes this problem can be solved by rebooting the desktop, but it is not always that simple.

There are many possible ways to speed up the CUDA-TMM that haven't been tried yet. Some are CUDA specific, and some are algorithmic. An algorithmic idea that hasn't been developed involves the MPS scheme. If some parameters are quicker to simulate than others, this will leave blocks that are finished inactive while the other blocks are still working. Instead, one could have the finished blocks working on new parameters. This negates the problem of running different disorders in parallel: since different disorders have a much larger difference in the amount of time taken to simulate, this won't be a problem if finished blocks can restart to work on other disorders. Another algorithmic idea is to investigate using a different random number generator, though this might not make much difference as the one currently being used just involves bit shifting on shared memory, so it is already pretty fast.

A more CUDA-specific idea is to have more control over the L2-cache in the Tesla M2050. One can negate the large latency of the global memory by exclusively using the L2-cache instead of the DRAM. The cache is usually hardware controlled. The only way to programme it is to use inline PTX code, assembly code for NVidia GPUs [43]. PTX is very low-level and would therefore be very difficult to implement as one would need to learn a new programming language and use it to code very detailed hardware operations.

CUDA is still under heavy development and has a long way to go before it's a well supported production compiler and programming model. It is not ready for this sort of problem. It is very well suited for problems which can be highly parallelised, i.e. split into lots of independent sub-problems with little communication between them. In the TMM, communication between cores is essential for re-orthonormalisation and requires high-latency global memory for systems larger than $M = 32$ in 2D. Despite these shortcomings, a maximum speedup of 13.5 times the serial implementation has been achieved (as shown in figure 6.14).

Using current GPGPU resources and armed with a fully debugged version of the CUDA-TMM implementation described in this thesis, one should theoretically be able to obtain an efficient 2D TMM simulation up to $M = 32$ if simulating hundreds of energy parameters simultaneously. At the current state of the CUDA-TMM, it is possible to achieve a maximum speed-up of 13.5 times the Serial-TMM for $M = 8$ and if over 1000 energy parameters are being run at once. Practically speaking, there are only very limited cases of the TMM which can be efficiently parallelised with CUDA. However, there is room for improvement. The future development of GPGPUs might prove better suited to the CUDA-TMM. Features worth looking out for in new GPGPUs that would help in the CUDA-TMM are:

- Either faster global memory or controllable L2-cache for inter-block communication
- Built-in inter-block synchronisation
- More Streaming Multiprocessors (SMs) or a way to communicate across more blocks than SMs

Appendix A

Source code

Included in this Appendix are the most important source code files. Source code for input/output subroutines, for example, was omitted for irrelevance. All code was written in FORTRAN 90 with CUDA extensions, and compiled using the PGI FORTRAN compiler. To compile `main.f90` for example (ignoring other code to link to), one would enter the following command:

```
pgf90 -Mcuda -c main.f90
```

A.1 `main.f90`

This source code contains the main program, in which subroutines from `util.f90` are called for the Serial-TMM and CUDA kernels from `cuda_util.f90` are called for the CUDA-TMM.

```
!*****************************************************************
!
! TMSEXD - Transfer matrix method for the Anderson
! model with diagonal disorder in X dimensions
!
!*****************************************************************

program tmsexd

  !---------------------------------------------------------------
  ! parameter and global variable definitions
  !---------------------------------------------------------------
  use mynumbers
  use cconstants
  use iconstants
  use ichannels
  use ipara
  use dpara

  use rng
  use cudafor
  use my_kernels

  !---------------------------------------------------------------
  ! local variable definitions
  !---------------------------------------------------------------
  implicit none

  integer :: ierr, data_version, iwidth, isize, iwidthrl, & width0_x, &
       ichannelmax, index, jndex, iter2, nofg, ig, ilayer, ivec, &
       jvec, num_bytes, int_bytes, real_bytes, iters, & num_paras, &
       old_num_paras, new_num_paras, bid, new_bid, old_bid, &
       int_store, dummy_int, devnum
  !integer(kind=4) :: iter1 !32-bit
  integer(kind=8) :: iter1 !64-bit

  real(kind=rkind) :: flux, flux0, flux1, dflux, flux0_x, &
       diagdis, energy, r_test

  logical :: kernel_finished

  ! timing variables
  real(4), dimension(1:3,1:1) :: time, timesum
  real(4) :: starttime(2), endtime(2)
  real(4) :: t, t2, etime
  external etime

  !---------------------------------------------------------------
  ! constants, date and time strings
  !---------------------------------------------------------------
  call init_numbers
  call date_time_str

  !---------------------------------------------------------------
  ! protocol feature startup
  !---------------------------------------------------------------
  RStr= "$Revision: 1.1 $ "
  DStr= "$Date: 2011/11/12 16:22:31 $ "
  AStr= "$Author: phrkaj $ "

  print*,rstr, dstr, astr
  write (*,5) datestr, timestr
5 format ("(* date ", a8, "; time ", a8, " *)" )

  ! read hostname
  call system('hostname > hostname.txt')
  open (ichhostname, file="hostname.txt", status="old")
  read (ichhostname, '(a)') hostname

  ! read project name
  call system('basename `pwd` > project.txt')
  open (ichproject, file="project.txt", status="old")
  read (ichproject, '(a)') projectname
  !print *, projectname
  print *

  !---------------------------------------------------------------
  ! input handling
  !---------------------------------------------------------------
  call input( ierr )
```

```
!print*, "dbg: ierr=", ierr
if( ierr.ne.0 ) then
   print*,"main: error in input()"
   stop
endif

!----------------------------------------------------------------------
! set cuda device
!----------------------------------------------------------------------
if(icudaflag > 0 .and. idevflag >= 0) then
   dummy_int = cudasetdevice(idevflag)
end if

!----------------------------------------------------------------------
! print flags and info
!----------------------------------------------------------------------
write(*,7) hostname
7 format(" hostname  = ", a)
write(*,8) projectname
8 format(" project   = ", a)

print*,'icudaflag = ', icudaflag
print*,'idevflag  = ', idevflag
print*,'iswapflag = ', iswapflag
print*,"----------------------------------------------------------"

!----------------------------------------------------------------------
! setup flux parameters
!----------------------------------------------------------------------
select case(icudaflag)
case(0)

   ! serial case
   if (ifluxflag.eq.0) then
      flux0     = diagdis0
      flux1     = diagdis1
      dflux     = ddiagdis
      energy    = energy0
   else
      flux0     = energy0
      flux1     = energy1
      dflux     = denergy
      diagdis   = diagdis0
   endif

case(2)

   ! cuda2 case
   select case(ifluxflag)
   case(0)

      if(diagdis1 == diagdis0) then
         num_paras = 1
      else
         num_paras = (diagdis1 - diagdis0)/real(ddiagdis) + 1
      end if

   case(1)

      if(energy1 == energy0) then
         num_paras = 1
      else
         num_paras = (energy1 - energy0)/real(denergy) + 1
      end if

   end select

case(3)

   ! cuda3 case
   if (ifluxflag.eq.0) then
      flux0     = diagdis0
      flux1     = diagdis1
      dflux     = ddiagdis
      energy    = energy0
   else
      flux0     = energy0
      flux1     = energy1
      dflux     = denergy
      diagdis   = diagdis0
   endif

case default
   stop "invalid icudaflag"
end select

!----------------------------------------------------------------------
! main parameter sweep
!----------------------------------------------------------------------
width_loop: &
do iwidth= width0,width1,dwidth

   ! ------------------------------------------------------------------
   ! get time at start of the process
   ! ------------------------------------------------------------------
   t = etime(starttime)

   !----------------------------------------------------------------------
   ! calculate isize and num_bytes needed for shared mem allocation
   !----------------------------------------------------------------------
   select case(idimenflag)
   case(3)
      isize = iwidth*iwidth
   case default
      isize = iwidth
   end select

   if(icudaflag >= 1) then

      int_bytes = 4*isize*sizeof(int(1))
      real_bytes = isize*sizeof(real(1.0))

      if(icudaflag == 2) then
         num_bytes = (4*isize*isize + 5*isize)*sizeof(real(1.0)) &
              + 4*isize*sizeof(int(1))
      else if(icudaflag == 3) then
         num_bytes = 5*isize*sizeof(real(1.0)) + 4*isize*sizeof(int(1))
      end if

   end if

   ! the # lyapunov exponents is maximally .eq. to iwidth
   nofg= min( nofgamma, isize )

   !----------------------------------------------------------------------
   ! open files
   !----------------------------------------------------------------------
   call openoutputavg( iwidth, data_version )
   print*,"data_version = ", data_version
   print*,"num_bytes = ", num_bytes
   if(icudaflag==2) print*,"num_paras = ", num_paras

   !----------------------------------------------------------------------
   ! choose between serial and cuda
   !----------------------------------------------------------------------
   select case(icudaflag)
   case(0)
      !----------------------------------------------------------------
      !----------------------------------------------------------------
      ! serial code
      !----------------------------------------------------------------
      !----------------------------------------------------------------

      !----------------------------------------------------------------
      ! flux loop
      !----------------------------------------------------------------
      flux_loop_serial: do flux= flux0,flux1,dflux

         !-------------------------------------------------------------
         ! allocate memory for the arrays
         !-------------------------------------------------------------
         num_paras = 1
         call allocatearrays(isize, ierr, num_paras)

         !-------------------------------------------------------------
         ! set values for the physical quantities
         !-------------------------------------------------------------
         if (ifluxflag.eq.0) then
            diagdis   = flux
         else
            energy    = flux
         endif

         !-------------------------------------------------------------
         ! protocoll feature
         !-------------------------------------------------------------
         write(*,1500) iwidth, diagdis, energy
1500     format("start @ iw= ", i4.1, ", dd= ", g10.3, ", en= ", g10.3)

         !-------------------------------------------------------------
         ! initialize the random number generator.
         !-------------------------------------------------------------
         call srandom(iseed)

         !-------------------------------------------------------------
         ! initialize the wave vectors and the gamma sums
         !-------------------------------------------------------------

         ! reset the wave vectors
         psia= zero
         psib= zero
         do index=1,isize
            psia(index,index)  = one
         enddo
         ! reset the gammas and error
         gamma          = zero
         gamma2         = zero
         acc_variance   = zero

         ! set convergence flag
         tmm_converged = .false.

         !-------------------------------------------------------------
         ! iteration loop
         !-------------------------------------------------------------
         tmm_loop: do iter1= 1, max( (nofiter)/(nofortho), 1)

            !----------------------------------------------------------
            ! serial tmms
            !----------------------------------------------------------
            select case(iswapflag)
```

```
      ! perform one tmmult per iteration, then swap
      case(1)

         northo_loop_swap: do iter2= 1, nofortho, 1

            call tmmult2d( psia, psib, &
                 energy, diagdis, iter1, iwidth)
            call swap( psia, psib, isize)

         enddo northo_loop_swap

      ! or perform two tmmult's per iteration
      case default

         northo_loop: do iter2= 1, nofortho, 2

            call tmmult2d( psia, psib, &
                 energy, diagdis, iter1, iwidth)
            call tmmult2d( psib, psia, &
                 energy, diagdis, iter1, iwidth)

         enddo northo_loop

      end select

      !-------------------------------------------------------------
      ! serial renorm
      !-------------------------------------------------------------
      call renorm(psia,psib,gamma,gamma2,iwidth)

      !-------------------------------------------------------------
      ! ngamma and check convergence
      !-------------------------------------------------------------
      call ngamma_calc_cpu(ngamma,gamma,gamma2,&
           acc_variance,isize,iwidth,iter1,tmm_converged)

      !-------------------------------------------------------------
      ! output
      !-------------------------------------------------------------
      if(iwriteflag.ge.1 .and. mod(iter1,nofprint).eq.0 ) then
         call writeoutput(iter1, isize, ngamma, &
              acc_variance, psia)
      endif

      !-------------------------------------------------------------
      ! check convergence
      !-------------------------------------------------------------
      if(tmm_converged) goto 4000

   enddo tmm_loop

   !----------------------------------------------------------------
   ! continue through here if convergence for a single
   ! configuration is not achieved, reset iter1
   !----------------------------------------------------------------
   print*,"no convergence in nofiter-loop:"
   iter1= iter1-1

   !----------------------------------------------------------------
   ! jump to this label if convergence for a single
   ! configuration is achieved.
   !----------------------------------------------------------------
4000  continue

   !----------------------------------------------------------------
   ! write the avg data
   !----------------------------------------------------------------
   call writeoutputavg(iwidth, diagdis, energy, ngamma, &
        acc_variance, nofg, psia, iter1, ierr )

   !----------------------------------------------------------------
   ! dump end data to standard output
   !----------------------------------------------------------------
   write(*,5010) iter1, diagdis, energy
   write(*,5012) ngamma(1), acc_variance(1)

5010  format("end @ ", i15.1, ",", g15.7, ",", g15.7)
5012  format("     ", g15.7, ",", g15.7)

   !----------------------------------------------------------------
   ! deallocate memory
   !----------------------------------------------------------------
   deallocate(psia,psib,ngamma,gamma,gamma2,acc_variance)

   !----------------------------------------------------------------
   ! end of flux loop
   !----------------------------------------------------------------

enddo flux_loop_serial

!-------------------------------------------------------------------
!-------------------------------------------------------------------
! cuda2 code
!-------------------------------------------------------------------
!-------------------------------------------------------------------
case(2)

   !----------------------------------------------------------------
   ! allocate memory for the arrays
   !----------------------------------------------------------------
   call allocatearrays(isize, ierr, num_paras)

   !----------------------------------------------------------------
   ! set dbg arrays
   !----------------------------------------------------------------
   if(idbgflag >= 1) then
      d_dbg2 = -1.0
      d_dbg2b = -1.0
      d_dbg3 = -1.0
      d_dbg4 = -1.0
      d_dbg5 = -1.0
      d_dbg6 = -1.0
      h_dbg2 = -1.0
      h_dbg3 = -1.0
      h_dbg4 = -1.0
      h_dbg5 = -1.0
      h_dbg6 = -1.0
   end if

   !----------------------------------------------------------------
   ! protocoll feature
   !----------------------------------------------------------------
   write(*,1510) iwidth, diagdis0, diagdis1, energy0, energy1
1510  format("iw= ", i4.1, ", dd0= ", g10.3, ", dd1= ", g10.3, &
        ", en0= ", g10.3, ", en1= ", g10.3)

   !----------------------------------------------------------------
   ! initialize the random number generator.
   !----------------------------------------------------------------
   if (icudaflag.ge.1) call srandom_cuda2<<<1,isize,int_bytes>>>&
        (isize,num_paras)

   !----------------------------------------------------------------
   ! initialize the wave vectors and the gamma sums
   !----------------------------------------------------------------

   ! reset the wave vectors
   d_psi_a       = zero
   d_psi_b       = zero

   h_psia        = zero
   h_psib        = zero

   do bid=1,num_paras
      do index=1,isize
         d_psi_a(bid,index,index)  = one
      enddo
   end do

   ! reset the gammas and error
   d_gamma       = zero
   d_gamma2      = zero
   d_acc_variance = zero
   d_ngamma      = zero !new

   h_gamma       = zero
   h_gamma2      = zero
   h_acc_variance = zero
   h_ngamma      = zero !new

   !----------------------------------------------------------------
   ! set convergence flags
   !----------------------------------------------------------------
   h_tmm_converged = .false.
   d_tmm_converged = .false.

   !----------------------------------------------------------------
   ! setup flux parameters
   !----------------------------------------------------------------
   if(ifluxflag==1) then
      do bid = 1, num_paras
         h_energy(bid)  = energy0  + (bid-1)*denergy
         h_diagdis(bid) = diagdis0
      end do
   else
      do bid = 1, num_paras
         h_diagdis(bid) = diagdis0 + (bid-1)*ddiagdis
         h_energy(bid)  = energy0
      end do
   end if
   d_energy = h_energy
   d_diagdis = h_diagdis

   !----------------------------------------------------------------
   ! prepare other stuff for kernel launch
   !----------------------------------------------------------------

   h_iter1 = 0
   d_iter1 = 0

   new_num_paras = num_paras
   kernel_finished = .false.
   h_finished = .false.

   !----------------------------------------------------------------
   ! iteration loop (cuda2)
   !----------------------------------------------------------------
   do while( .not.kernel_finished )

      d_finished = h_finished
      call master_kernel<<<num_paras,isize*isize,num_bytes>>>&
           (energy0,denergy,diagdis0,ddiagdis,epsilon,isize,iwidth,&
```

```
            nofortho,nofiter,&
            iwriteflag,idbgflag,nofprint,irngflag,iswapflag,&
            ibcflag,ifluxflag)

       !--------------------------------------------------------------
       ! copy data from device to host
       !--------------------------------------------------------------
       h_psia = d_psi_a
       h_psib = d_psi_b
       h_acc_variance = d_acc_variance
       h_ngamma = d_ngamma
       h_gamma = d_gamma
       h_gamma2 = d_gamma2

       ! load logicals, number of iterations
       h_tmm_converged = d_tmm_converged
       h_iter1 = d_iter1

       !--------------------------------------------------------------
       ! write data to standard output (and file)
       !--------------------------------------------------------------
       call writeoutputcuda2(num_paras, new_num_paras)

       !--------------------------------------------------------------
       ! write dbg
       !--------------------------------------------------------------
       if(idbgflag > 0) then
          h_dbg2 = d_dbg2
          h_dbg2b = d_dbg2b
          call writeoutputcudadbg2(isize)
       endif

       !--------------------------------------------------------------
       ! check to see if all blocks have finished
       !--------------------------------------------------------------
       if (new_num_paras < 1) kernel_finished = .true.

    end do

    !--------------------------------------------------------------
    ! write the avg data
    !--------------------------------------------------------------
    call writeoutputavgcuda2(iwidth, num_paras, diagdis0, energy0, &
         h_ngamma, h_acc_variance, nofg, h_psia, h_iter1, ierr)

    !--------------------------------------------------------------
    ! deallocate memory
    !--------------------------------------------------------------

    ! device arrays
    deallocate(d_idum,d_gamma,d_gamma2,d_psi_a,d_psi_b,d_z)
    deallocate(d_ngamma,d_acc_variance)
    deallocate(d_iter1,d_tmm_converged,d_finished,d_energy,d_diagdis)

    ! host arrays
    deallocate(h_gamma,h_gamma2,h_psia,h_psib,h_ngamma,h_acc_variance)
    deallocate(h_iter1,h_tmm_converged,h_finished,h_diagdis,h_energy)

    ! dbg arrays
    if(idbgflag.ge.1) then
       deallocate(d_dbg1, d_dbg2, d_dbg2b, d_dbg3, d_dbg3b)
       deallocate(d_dbg4, d_dbg4b, d_dbg5, d_dbg6)
       deallocate(h_dbg1, h_dbg2, h_dbg2b, h_dbg3, h_dbg3b)
       deallocate(h_dbg4, h_dbg4b, h_dbg5, h_dbg6)
    end if

 !---------------------------------------------------------------------
 !---------------------------------------------------------------------
 ! cuda3 code
 !---------------------------------------------------------------------
 !---------------------------------------------------------------------
 case(3)

    flux_loop_cuda3: do flux= flux0,flux1,dflux

       !--------------------------------------------------------------
       ! allocate memory for the arrays
       !--------------------------------------------------------------
       call allocatearrays(isize, ierr)

       !--------------------------------------------------------------
       ! set dbg variables
       !--------------------------------------------------------------
       h_test = 0
       d_test = 0
       if(idbgflag >= 1) then
          d3_dbg2 = -1.0
          d3_dbg3 = -1.0
          d3_dbg4 = -1.0
          d3_dbg5 = -1.0
          d3_dbg6 = -1.0
          h3_dbg2 = -1.0
          h3_dbg3 = -1.0
          h3_dbg4 = -1.0
          h3_dbg5 = -1.0
          h3_dbg6 = -1.0
       end if

       !--------------------------------------------------------------
       ! set values for the physical quantities
       !--------------------------------------------------------------
       if (ifluxflag.eq.0) then
          diagdis   = flux
       else
          energy    = flux
       endif

       !--------------------------------------------------------------
       ! protocoll feature
       !--------------------------------------------------------------
       write(*,2510) iwidth, diagdis, energy
2500   format("start @ iw= ", i4.1, ", dd= ", g10.3, ", en= ", g10.3)

       !--------------------------------------------------------------
       ! initialize the random number generator.
       !--------------------------------------------------------------
       call srandom_cuda3<<<1,isize,int_bytes>>>(isize)

       !--------------------------------------------------------------
       ! initialize the wave vectors and the gamma sums
       !--------------------------------------------------------------

       ! reset the wave vectors
       d3_psi_a        = zero
       d3_psi_b        = zero
       do index=1,isize
          d3_psi_a(index,index)  = one
       enddo

       ! reset the gammas and error
       d3_gamma        = zero
       d3_gamma2       = zero
       d3_acc_variance = zero

       ! set flags and iterations
       h3_tmm_converged = .false.
       d3_tmm_converged = .false.
       kernel_finished = .false.
       d3_kernel_finished = .false.
       d3_iter1 = 0

       !--------------------------------------------------------------
       ! start iterations
       !--------------------------------------------------------------
       !--------------------------------------------------------------

       arrayin = 0
       arrayout = 0
       d_atomic = 0

       do while (.not.kernel_finished)

          call cuda3kernel<<<isize,isize,num_bytes>>>&
               (energy,diagdis,epsilon,isize,iwidth,nofortho,nofiter,&
               iwriteflag,idbgflag,nofprint,irngflag,iswapflag,&
               ibcflag)

          !--------------------------------------------------------------
          ! copy data from device to host
          !--------------------------------------------------------------
          h3_psia = d3_psi_a
          h3_psib = d3_psi_b
          h3_acc_variance = d3_acc_variance
          h3_ngamma = d3_ngamma
          h3_gamma = d3_gamma
          h3_gamma2 = d3_gamma2

          ! load number of iterations
          iter1 = d3_iter1

          ! load convergence and kernel flags
          h3_tmm_converged = d3_tmm_converged
          kernel_finished = d3_kernel_finished

          !--------------------------------------------------------------
          ! check convergence
          !--------------------------------------------------------------
          if(h3_tmm_converged) goto 4010

          !--------------------------------------------------------------
          ! write data to standard output
          !--------------------------------------------------------------
          call writeoutputcuda3(iter1, diagdis, energy)

       end do

       !--------------------------------------------------------------
       ! continue through here if convergence for a single
       ! configuration is not achieved, reset iter1
       !--------------------------------------------------------------
       print*,"no convergence in nofiter-loop:"
       !iter1= iter1-1

4010   continue

       !--------------------------------------------------------------
       ! write the avg data
       !--------------------------------------------------------------
       call writeoutputavg(iwidth, diagdis, energy, h3_ngamma, &
            h3_acc_variance, nofg, h3_psia, iter1, ierr )

       !--------------------------------------------------------------
```

```
         ! dump end data to standard output
         !-------------------------------------------------------------
6000     write(*,6010) iter1, diagdis, energy
         write(*,6012) h3_ngamma(1), h3_acc_variance(1)

6010     format("end @ ", i7.1, ",", g15.7, ",", g15.7)
6012     format("     ", g15.7, ",", g15.7)

         if(idbgflag >= 1) then
            h3_dbg2 = d3_dbg2
            h3_dbg3 = d3_dbg3
            h3_dbg4 = d3_dbg4
            h3_dbg5 = d3_dbg5
            call writeoutputcudadbg3(isize)
         end if

         !-------------------------------------------------------------
         ! deallocate memory
         !-------------------------------------------------------------

         ! device arrays
         deallocate(d_idum,d3_gamma,d3_gamma2,d3_psi_a,d3_psi_b,d3_z)
         deallocate(d3_ngamma,d3_acc_variance)

         ! host arrays
         deallocate(h3_gamma,h3_gamma2,h3_psia,h3_psib)
         deallocate(h3_ngamma,h3_acc_variance)

         ! gpu_sync arrays
         deallocate(arrayin, arrayout, d_atomic)

         ! dbg arrays
         if(idbgflag.ge.1) then
            deallocate(d3_dbg1, d3_dbg2, d3_dbg2b, d3_dbg3, d3_dbg3b)
            deallocate(d3_dbg4, d3_dbg4b, d3_dbg5, d3_dbg6)
            deallocate(h3_dbg1, h3_dbg2, h3_dbg2b, h3_dbg3, h3_dbg3b)
            deallocate(h3_dbg4, h3_dbg4b, h3_dbg5, h3_dbg6)
         end if

         !-------------------------------------------------------------
         ! end of flux loop
         !-------------------------------------------------------------

      enddo flux_loop_cuda3

      !----------------------------------------------------------------
      !----------------------------------------------------------------
      ! end of serial/cuda code
```

```
         !-------------------------------------------------------------
         !-------------------------------------------------------------

      case default
         stop "invalid icudaflag"
      end select

      print*,"-------------------------------------------------------"

      !-------------------------------------------------------------
      ! get time at the end of the process
      !-------------------------------------------------------------
      t2 = etime(endtime)

      time(1,1) = t2 -t
      time(2,1) = endtime(1)-starttime(1)
      time(3,1) = endtime(2)-starttime(2)

      t  = time(1,1)
      t2 = time(2,1)

      write(*,'(a20,6f12.4)') &
           "time(usr,sys,diff): ", time(1,1), time(2,1), time(3,1)

      !-------------------------------------------------------------
      ! close avg file
      !-------------------------------------------------------------
      call closeoutputavg( ierr, time(1,1), time(2,1), time(3,1) )

      !-------------------------------------------------------------
      ! end of width loop
      !-------------------------------------------------------------

   enddo width_loop

   ! close other files
   close(ichhostname)
   close(ichproject)

   ! write date and time of end of run
   call date_time_str
   write (*,6) datestr, timestr
6  format ("(* date ", a8, "; time ", a8, " *)" )

   stop "tmsexd $revision: 1.1 $"

end program tmsexd
```

A.2 `util.f90`

This source code contains the subroutines used in the Serial-TMM, except for the random number generator which is contained in `random.f90`.

```
! ----------------------------------------------------------------------
! tmmult2d:
!
! multiplication of the transfer matrix onto the vector (psi_a,psi_b),
! giving (psi_b,psi_a) so that the structure of the transfer matrix
! can be exploited

subroutine tmmult2d(psi_a,psi_b, en, diagdis, iter1, M )

  use mynumbers
  use ipara
  use rng
  use dpara

  ! wave functions:
  !
  ! (psi_a, psi_b) on input, (psi_b,psi_a) on output

  implicit none

  integer M, iter1                    ! strip width

  real(kind=rkind)  diagdis,&! diagonal disorder
       en                    ! energy

  real(kind=rkind) psi_a(M,M), psi_b(M,M)

  integer isite, jstate, iseeddummy
  real(kind=rkind) onsitepot
  real(kind=rkind) new, psileft, psiright

  !print*,"dbg: tmmult2d()"

  do isite=1,M

     ! create the new onsite potential
     select case(irngflag)
     case(0)
        onsitepot= en - diagdis*(drandom(iseeddummy)-0.5_rkind)
     case(1)
        onsitepot= mod(iter1,10) * 0.1
     end select

     do jstate=1,M

        if (isite.eq.1) then

           if (ibcflag.eq.0) then
              psileft= czero              ! hard wall bc
           else if (ibcflag.eq.1) then
              psileft= psi_a(jstate,M)  ! periodic bc
           else if (ibcflag.eq.2) then
              psileft= -psi_a(jstate,M) ! antiperiodic bc
           endif

        else
           psileft= psi_a(jstate,isite-1)
        endif

        if (isite.eq.M) then

           if (ibcflag.eq.0) then
              psiright= czero             ! hard wall bc
           else if (ibcflag.eq.1) then
              psiright= psi_a(jstate,1)  ! periodic bc
           else if (ibcflag.eq.2) then
              psiright= -psi_a(jstate,1) ! antiperiodic bc
           endif

        else
           psiright= psi_a(jstate,isite+1)
        endif

        new= onsitepot * psi_a(jstate,isite) - &
             ( psileft + psiright ) - psi_b(jstate,isite)
```

```
        psi_b(jstate,isite)= new

      enddo ! jstate
    enddo ! isite

    return
  end subroutine tmmult2d

  ! ------------------------------------------------------------------
  ! tmmult3d:
  !
  ! 3d version of tmmult2d. extra boundary conditions

  subroutine tmmult3d(psi_a,psi_b, en, diagdis, M )

    use mynumbers
    use ipara
    use rng
    use dpara

    ! wave functions:
    !
    ! (psi_a, psi_b) on input, (psi_b,psi_a) on output

    implicit none

    integer M                     ! strip width

    real(kind=rkind)  diagdis,&! diagonal disorder
         en                   ! energy

    real(kind=rkind) psi_a(M*M,M*M), psi_b(M*M,M*M)

    integer isite, jstate, iseeddummy
    real(kind=rkind) onsitepot
    real(kind=rkind) new, psileft, psiright, psiup, psidown

    !print*,"dbg: tmmult3d()"

    do isite=1,M*M

       ! create the new onsite potential
       select case(irngflag)
       case(0)
          onsitepot= en - diagdis*(drandom(iseeddummy)-0.5_rkind)
       case(1)
          onsitepot= en - diagdis*(drandom(iseeddummy)-0.5_rkind)&
               *sqrt(12.0_rkind)
       case(2)
          onsitepot= en - grandom(iseeddummy,0.0_rkind,diagdis)
       end select

       do jstate=1,M*M

          ! psileft
          !if (isite.eq.1) then
          if (mod(isite,M).eq.1) then

             if (ibcflag.eq.0) then
                psileft= czero            ! hard wall bc
             else if (ibcflag.eq.1) then
                psileft= psi_a(jstate,isite+m-1)  ! periodic bc
             else if (ibcflag.eq.2) then
                psileft= -psi_a(jstate,isite+m-1) ! antiperiodic bc
             endif

          else
             !print*,"dbg: isite=",isite
             psileft= psi_a(jstate,isite-1)
          endif

          ! psiright
          !if (isite.eq.M) then
          if (mod(isite,M).eq.0) then

             if (ibcflag.eq.0) then
                psiright= czero            ! hard wall bc
             else if (ibcflag.eq.1) then
                psiright= psi_a(jstate,isite-m+1)  ! periodic bc
             else if (ibcflag.eq.2) then
                psiright= -psi_a(jstate,isite-m+1) ! antiperiodic bc
             endif

          else
             psiright= psi_a(jstate,isite+1)
          endif

          ! psiup
          if (isite.gt.(M-1)*M) then

             if (ibcflag.eq.0) then
                psiup= czero              ! hard wall bc
             else if (ibcflag.eq.1) then
                psiup= psi_a(jstate,isite-(M-1)*M)  ! periodic bc
             else if (ibcflag.eq.2) then
                psiup= -psi_a(jstate,isite-(M-1)*M) ! antiperiodic bc
             endif

          else
             psiup= psi_a(jstate,isite+M)
          endif

          ! psidown
          if (isite.lt.(M+1)) then

             if (ibcflag.eq.0) then
                psidown= czero            ! hard wall bc
             else if (ibcflag.eq.1) then
                psidown= psi_a(jstate,isite+(M-1)*M)  ! periodic bc
             else if (ibcflag.eq.2) then
                psidown= -psi_a(jstate,isite+(M-1)*M) ! antiperiodic bc
             endif

          else
             psidown= psi_a(jstate,isite-M)
          endif

          new= onsitepot * psi_a(jstate,isite) - &
               ( psileft+ psiright + psiup + psidown ) &
               - psi_b(jstate,isite)

          psi_b(jstate,isite)= new

       enddo ! jstate
    enddo ! isite

    return
  end subroutine tmmult3d

  !        ------------------------------------------------------------
  !        swap:
  !
  !        (psi_a,psi_b)= (old,new) is the incoming vector, this is
  !        swapped into (psi_a,psi_b)= (new,old)

  subroutine swap( psi_a, psi_b, M)

    use mynumbers

    integer M
    real(kind=rkind) psi_a(M,M), psi_b(M,M)

    integer jstate, index
    real(kind=rkind) dummy

    !        print*,"dbg: swap()"

    do jstate=1,M
       do index=1,M

          dummy              = psi_b(index,jstate)
          psi_b(index,jstate)= psi_a(index,jstate)
          psi_a(index,jstate)= dummy

       enddo
    enddo

    return

  end subroutine swap

  !        ------------------------------------------------------------
  !        resort:
  !
  !        sort the lyapunov eigenvalues s.t. the largest comes first.
  !        resort() is shellsort taken from numrec, shell().

  subroutine resort( psi_a, psi_b, array0, array1, n )

    use mynumbers

    integer n
    real(kind=rkind) psi_a(n,n),psi_b(n,n)
    real(kind=rkind) array0(n), array1(n)

    real(kind=rkind) aln2i, localtiny
    parameter (aln2i=1.4426950_rkind, localtiny=1.d-5)

    integer nn,m,l,k,j,i,lognb2, index
    real(kind=rkind) dummya, dummyb

    !        print*,"dbg: resort()"

    !        print*,"array0(1),array0(n)",array0(1),array0(n)

    lognb2=int(log(real(n))*aln2i+localtiny)
    m=n
    do 12 nn=1,lognb2
       m=m/2
       k=n-m
       do 11 j=1,k
          i=j
3         continue
          l=i+m
          if(array0(l).gt.array0(i)) then

             dummya   = array0(i)
             array0(i)= array0(l)
             array0(l)= dummya

             dummyb   = array1(i)
             array1(i)= array1(l)
```

```
          array1(l)= dummyb

          do 100 index=1,n
             dummya        = psi_a(index,i)
             dummyb        = psi_b(index,i)

             psi_a(index,i)= psi_a(index,l)
             psi_b(index,i)= psi_b(index,l)

             psi_a(index,l)= dummya
             psi_b(index,l)= dummyb
100       enddo

          i=i-m
          if(i.ge.1) goto 3
       endif
11   enddo
12 enddo

  !       print*,"array0(1),array0(n)",array0(1),array0(n)
  return

end subroutine resort

!-----------------------------------------------------------
! renorm:
!
subroutine renorm(psi_a,psi_b,gamma,gamma2,M)

  use mynumbers
  use iconstants
  use ipara
  use my_kernels
  implicit none

  integer M

  real(kind=rkind) psi_a(M,M), psi_b(M,M)
  real(kind=rkind) gamma(M), gamma2(M)

  integer ivec,jvec,kindex,i,j

  real(kind=rkind) sum
  real(kind=rkind) dummy2
  real(kind=rkind) dummy,norm
  equivalence (dummy,norm)

  !make the local variables static
  !save

  !print*,"dbg: renorm()"

  !-----------------------------------------------------------
  ! serial gs
  !-----------------------------------------------------------

  do ivec=1,M

     !-----------------------------------------------------------
     ! normalise
     !-----------------------------------------------------------

     ! calculation of norm
     norm= zero
     do kindex=1,M
        norm= norm + psi_a(ivec,kindex) * psi_a(ivec,kindex) &
             + psi_b(ivec,kindex) * psi_b(ivec,kindex)
     enddo

     ! normalise wavefunctions
     dummy = one/sqrt(norm)
     do kindex=1,M
        psi_a(ivec,kindex)= dummy * psi_a(ivec,kindex)
        psi_b(ivec,kindex)= dummy * psi_b(ivec,kindex)
     enddo

     !-----------------------------------------------------------
     ! orthogonalise (serial)
     !-----------------------------------------------------------

     do jvec=ivec+1,M

        sum= zero

        ! calculate projection
        do kindex=1,M

           dummy2 = psi_a(jvec,kindex)*psi_a(ivec,kindex) &
                + psi_b(jvec,kindex)*psi_b(ivec,kindex)

           sum= sum + dummy2

        enddo

        ! subtract projection from vector
        do kindex=1,M

           psi_a(jvec,kindex)= psi_a(jvec,kindex) - &
                sum * psi_a(ivec,kindex)
           psi_b(jvec,kindex)= psi_b(jvec,kindex) - &
                sum * psi_b(ivec,kindex)

        enddo

     enddo

     !-----------------------------------------------------------
     ! calculate gamma
     !-----------------------------------------------------------

     !print*,'dbg: dummy = ', dummy
     dummy      = log(dummy)
     gamma(ivec) = gamma(ivec) - dummy
     gamma2(ivec)= gamma2(ivec) + dummy*dummy

  enddo

  return

end subroutine renorm

! -----------------------------------------------------------
! ngamma_calc_cpu:
!
subroutine ngamma_calc_cpu(ngamma,gamma,gamma2,acc_variance,&
     isize,iwidth,iter1,tmm_converged)

  use mynumbers
  use iconstants
  use ipara
  use dpara
  implicit none

  integer, intent(in) :: isize, iwidth
  !integer, intent(in) :: iter1
  integer(kind=8), intent(in) :: iter1

  logical :: tmm_converged
  real(kind=rkind), dimension(isize) :: ngamma, gamma, gamma2, &
       acc_variance

  real(kind=rkind) :: thing
  integer :: ig

  do ig=1, isize

     ngamma(isize+1-ig)= gamma(ig)/real(nofortho*iter1)

     thing = gamma(ig)/real(iter1)
     acc_variance(isize+1-ig)=             &
          sqrt( abs(                       &
          (gamma2(ig)/real(iter1) -        &
          (thing)**2 )      &
          / real( max(iter1-1,1) )         &
          )) / abs( thing )
  enddo

  !-----------------------------------------------------
  ! check accuracy and dump the result
  !-----------------------------------------------------
  if( iter1.ge.iwidth  .and. &
       iter1.ge.miniter ) then

     if( acc_variance(1).le.epsilon .and. &
          acc_variance(1).ge.tiny) then
        tmm_converged=.true.
     endif

  endif

end subroutine ngamma_calc_cpu
```

A.3 cuda_util.f90

This source code contains all the subroutines (including the RNG), arrays and variables used by the GPU for the CUDA-TMM. All references to `cuda2` relate to the MPS, whereas `cuda3` relates to the SPS. `cuda1` is an old, inefficient CUDA-TMM

scheme which has been removed from the code.

```
module my_kernels
  use mynumbers
  use iconstants
  use cudafor
  implicit none

  !-----------------------------------------------------------------
  ! cuda2 + cuda3 arrays and variables
  !-----------------------------------------------------------------
  integer(kind=ikind),device,dimension(:),allocatable :: d_idum

  !-----------------------------------------------------------------
  ! cuda2 arrays and variables (multi-parameters)
  !-----------------------------------------------------------------

  ! device arrays
  real(kind=rkind),device,dimension(:,:),allocatable :: d_gamma, d_gamma2
  real(kind=rkind),device,dimension(:,:,:), allocatable :: d_psi_a, d_psi_b
  integer(kind=ikind),device,dimension(:,:),allocatable :: d_z
  real(kind=rkind),device,dimension(:,:),allocatable :: d_ngamma, &
       d_acc_variance

  integer(kind=8), device, dimension(:),allocatable :: d_iter1 !64-bit
  logical,device, dimension(:),allocatable :: d_tmm_converged
  logical,device, dimension(:),allocatable :: d_finished
  real(kind=rkind),device,dimension(:),allocatable :: d_energy, d_diagdis

  ! host arrays (page-locked memory)
  real(kind=rkind),pinned,dimension(:,:),allocatable :: h_gamma, h_gamma2
  real(kind=rkind),pinned,dimension(:,:,:), allocatable :: h_psia, h_psib
  real(kind=rkind),pinned,dimension(:,:),allocatable :: h_ngamma, &
       h_acc_variance

  integer(kind=8),pinned,dimension(:), allocatable :: h_iter1
  logical,pinned,dimension(:),allocatable :: h_tmm_converged
  logical,pinned,dimension(:),allocatable :: h_finished
  real(kind=rkind),pinned,dimension(:),allocatable :: h_diagdis, h_energy

  ! dbg arrays
  real(kind=rkind),allocatable,dimension(:,:,:,:) :: h_dbg1
  real(kind=rkind),device,allocatable,dimension(:,:,:,:) :: d_dbg1
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg2
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg2
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg2b
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg2b
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg3
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg3
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg3b
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg3b
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg4
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg4
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg4b
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg4b
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg5
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg5
  real(kind=rkind),allocatable,dimension(:,:,:) :: h_dbg6
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d_dbg6

  !-----------------------------------------------------------------
  ! cuda3 arrays and variables (cudalarge)
  !-----------------------------------------------------------------

  ! device arrays
  real(kind=rkind),device,dimension(:),allocatable :: d3_gamma, d3_gamma2
  real(kind=rkind),device,dimension(:,:), allocatable :: d3_psi_a, d3_psi_b
  integer(kind=ikind),device,dimension(:),allocatable :: d3_z
  real(kind=rkind),device,dimension(:),allocatable :: d3_ngamma, &
       d3_acc_variance

  ! host arrays (page-locked memory)
  real(kind=rkind),pinned,dimension(:),allocatable :: h3_gamma, h3_gamma2
  real(kind=rkind),pinned,dimension(:,:), allocatable :: h3_psia, h3_psib
  real(kind=rkind),pinned,dimension(:),allocatable :: h3_ngamma, &
       h3_acc_variance

  ! gpu_sync arrays
  integer, device, dimension(:), allocatable :: arrayin, arrayout
  integer, device, dimension(:,:), allocatable :: d_atomic

  ! flags and other variables
  logical,device :: d3_tmm_converged
  logical,device :: d3_kernel_finished
  logical        :: h3_tmm_converged
  integer(kind=8), device :: d3_iter1 !64-bit
  !integer(kind=8)         :: h3_iter1 !64-bit

  ! dbg arrays
  real(kind=rkind),allocatable,dimension(:,:,:) :: h3_dbg1
  real(kind=rkind),device,allocatable,dimension(:,:,:) :: d3_dbg1
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg2
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg2
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg2b
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg2b
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg3
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg3
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg3b
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg3b
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg4
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg4
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg4b
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg4b
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg5
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg5
  real(kind=rkind),allocatable,dimension(:,:) :: h3_dbg6
  real(kind=rkind),device,allocatable,dimension(:,:) :: d3_dbg6

  !-----------------------------------------------------------------
  ! serial arrays and variables
  !-----------------------------------------------------------------

  real(kind=rkind),dimension(:),allocatable :: gamma, gamma2
  real(kind=rkind),dimension(:,:), allocatable :: psia, psib
  real(kind=rkind),dimension(:),allocatable :: ngamma, acc_variance

  logical :: tmm_converged

  !-----------------------------------------------------------------
  ! other stuff
  !-----------------------------------------------------------------

  ! rng parameters
  real(kind=rkind),    parameter :: am = 4.656612873077d-10
  integer(kind=ikind), parameter :: ia = 16807
  integer(kind=ikind), parameter :: im = 2147483647
  integer(kind=ikind), parameter :: iq = 127773
  integer(kind=ikind), parameter :: ir = 2836

  ! dbg variables
  integer, device :: d_test
  integer :: h_test

contains

  !-----------------------------------------------------------
  !-----------------------------------------------------------
  ! rng kernels
  !-----------------------------------------------------------
  !-----------------------------------------------------------

  ! ----------------------------------------------------------------
  ! rlfsr113_kernel() returns random numbers of interval (0, 1)
  ! ----------------------------------------------------------------
  attributes(device) function rlfsr113_device(z1, z2, z3, z4, z) &
       result (dret)

    integer(kind=ikind), intent(in), value :: z1, z2, z3, z4
    integer(kind=ikind), shared, dimension(*) :: z
    integer(kind=ikind)          :: b
    real(kind=rkind)             :: dret

    b  = ibits(ieor(ishft(z(z1),6),z(z1)),13,19)!32-13)
    z(z1) = ieor(ishft(iand(z(z1),-2),18),b)

    b  = ibits(ieor(ishft(z(z2),2),z(z2)),27,5)!32-27)
    z(z2) = ieor(ishft(iand(z(z2),-8),2),b)

    b  = ibits(ieor(ishft(z(z3),13),z(z3)),21,11)!,32-21)
    z(z3) = ieor(ishft(iand(z(z3),-16),7),b)

    b  = ibits(ieor(ishft(z(z4),3),z(z4)),12,20)!32-12)
    z(z4) = ieor(ishft(iand(z(z4),-128),13),b)

    dret=ibits(ieor(ieor(ieor(z(z1),z(z2)),z(z3)),z(z4)),1,31)*am

  end function rlfsr113_device

  ! ----------------------------------------------------------------
  ! lfsrinit_kernel()  initialize rlfsr113_kernel (z1,z2,z3,z4)
  ! ----------------------------------------------------------------
  attributes(device) subroutine lfsrinit_device(i, z1, z2, z3, z4, z)
    implicit none
    integer(kind=ikind), intent(in), value :: i, z1, z2, z3, z4
    integer(kind=ikind) :: k
    real(kind=rkind) :: dummy
    integer(kind=ikind), shared, dimension(*) :: z

    ! initialize z1,z2,z3,z4
    if (d_idum(i).le.0) d_idum(i)=1
    k=(d_idum(i))/iq
    d_idum(i)=ia*(d_idum(i)-k*iq)-ir*k
    if (d_idum(i).lt.0) d_idum(i) = d_idum(i) + im
    if (d_idum(i).lt.2) then
      z(z1)=d_idum(i)+2
    else
      z(z1)=d_idum(i)
    endif
    k=(d_idum(i))/iq
    d_idum(i)=ia*(d_idum(i)-k*iq)-ir*k
    if (d_idum(i).lt.0) d_idum(i) = d_idum(i) + im
    if (d_idum(i).lt.8) then
      z(z2)=d_idum(i)+8
    else
      z(z2)=d_idum(i)
    endif
```

```
  k=(d_idum(i))/iq
  d_idum(i)=ia*(d_idum(i)-k*iq)-ir*k
  if (d_idum(i).lt.0) d_idum(i) = d_idum(i) + im
  if (d_idum(i).lt.16) then
    z(z3)=d_idum(i)+16
  else
    z(z3)=d_idum(i)
  endif
  k=(d_idum(i))/iq
  d_idum(i)=ia*(d_idum(i)-k*iq)-ir*k
  if (d_idum(i).lt.0) d_idum(i) = d_idum(i) + im
  if (d_idum(i).lt.128) then
    z(z4)=d_idum(i)+128
  else
    z(z4)=d_idum(i)
  endif

  ! make a single call to rand_gen() to achieve a valid state
  dummy = rlfsr113_device(z1, z2, z3, z4, z)

end subroutine lfsrinit_device

! ------------------------------------------------------------------
! srandom_cuda2()
! ------------------------------------------------------------------
attributes(global) subroutine srandom_cuda2(n, num_paras)
  implicit none
  integer, intent(in), value :: n, num_paras
  integer(kind=ikind) :: tid, bid, z1, z2, z3, z4
  real(kind=rkind) :: dummy
  integer(kind=ikind), shared, dimension(*) :: z

  ! identify threads and blocks
  tid = threadidx%x

  ! create offsets for shared array, z
  z1 = tid
  z2 = z1 + n
  z3 = z2 + n
  z4 = z3 + n

  ! create seeds
  d_idum(tid) = 1276 + tid

  ! initialise random number generator
  call lfsrinit_device(tid,z1,z2,z3,z4,z)

  ! save z to global memory
  do bid = 1,num_paras
     d_z(bid,z1) = z(z1)
     d_z(bid,z2) = z(z2)
     d_z(bid,z3) = z(z3)
     d_z(bid,z4) = z(z4)
  end do

end subroutine srandom_cuda2

! ------------------------------------------------------------------
! srandom_cuda3()
! ------------------------------------------------------------------
attributes(global) subroutine srandom_cuda3(n)
  implicit none
  integer, intent(in), value :: n
  integer(kind=ikind) :: i, z1, z2, z3, z4
  real(kind=rkind) :: dummy
  integer(kind=ikind), shared, dimension(*) :: z

  ! identify threads
  i = threadidx%x

  ! create offsets for shared array, z
  z1 = i
  z2 = z1 + n
  z3 = z2 + n
  z4 = z3 + n

  ! create seeds
  d_idum(i) = 1276 + i

  ! initialise random number generator
  call lfsrinit_device(i,z1,z2,z3,z4,z)

  ! save z to global memory
  d3_z(z1) = z(z1)
  d3_z(z2) = z(z2)
  d3_z(z3) = z(z3)
  d3_z(z4) = z(z4)

end subroutine srandom_cuda3

!---------------------------------------------------------
!---------------------------------------------------------
! tmm kernels
!---------------------------------------------------------
!---------------------------------------------------------

! ------------------------------------------------------------------
! master_kernel:
!
attributes(global) subroutine master_kernel(energy0, denergy, diagdis0, &
     ddiagdis, epsilon, isize, iwidth, nofortho, nofiter, iwriteflag, &
     idbgflag, nofprint, irngflag, iswapflag, ibcflag, &
     ifluxflag)

integer, intent(in), value :: isize, iwidth, nofortho,&
     iwriteflag, idbgflag, nofprint, irngflag, iswapflag, &
     ibcflag, ifluxflag
!integer, intent(in), value :: nofiter
integer(kind=8), intent(in), value :: nofiter

real(kind=rkind), intent(in), value :: energy0, denergy, diagdis0, &
     ddiagdis, epsilon

integer :: iter2, tid, i, j, psi_a, psi_b, z1, z2, z3, z4, v, &
     isize2, gamma, gamma2, ngamma, acc_var, norm_sum, orth_sum, &
     ivec, norm_1, orth_1, psi_a_ivec, psi_b_ivec, acc_var_1, &
     psi_a_1, psi_a_m, psi_b_1, psi_b_m, bid
!integer :: iter1, index
integer(kind=8) :: iter1, index, itersave

real(kind=rkind) :: energy, diagdis

real(kind=rkind), shared, dimension(*) :: shared_real
integer(kind=ikind), shared, dimension(*) :: shared_int

logical :: tmm_converged

! identify threads, blocks, virtual and real id's
tid  = threadidx%x
bid  = blockidx%x
i = mod(tid-1,isize) + 1 ! virtual threadid
j = (tid-1) / isize + 1 ! virtual blockid

!---------------------------------------------------------
! if parameter finished, stop here
!---------------------------------------------------------
if ( d_finished(bid) ) return

!---------------------------------------------------------
! setup flux parameter
!---------------------------------------------------------
energy = d_energy(bid)
diagdis = d_diagdis(bid)

!---------------------------------------------------------
! load number of iterations from before
!---------------------------------------------------------
itersave = d_iter1(bid)

!---------------------------------------------------------
! offset indices
!---------------------------------------------------------
isize2 = isize*isize

! shared_real indices unique to each thread in kernel
psi_a = tid
psi_b = isize2 + tid
norm_sum = 2*isize2 + tid
orth_sum = 3*isize2 + tid

! shared_real indices unique to virtual block
psi_a_1 = (j-1)*isize + 1
psi_a_m = j*isize
psi_b_1 = psi_a_1 + isize2
psi_b_m = psi_a_m + isize2
norm_1  = psi_a_1 + 2*isize2
orth_1  = psi_a_1 + 3*isize2

gamma = 4*isize2 + isize + j
gamma2 = gamma + isize
ngamma = gamma2 + isize
acc_var = ngamma + isize
acc_var_1 = 4*isize2 + 5*isize

! shared_real indices unique to virtual threads
v = 4*isize2 + i

! shared_int indices unique to virtual threads
z1 = 4*isize2 + 5*isize + i
z2 = z1 + isize
z3 = z2 + isize
z4 = z3 + isize

!---------------------------------------------------------
! copy psi, gammas and acc_var from global memory to shared memory
!---------------------------------------------------------
shared_real(psi_a) = d_psi_a(bid,j,i)
shared_real(psi_b) = d_psi_b(bid,j,i)
if (i.eq.1) then
   shared_real(gamma) = d_gamma(bid,j)
   shared_real(gamma2) = d_gamma2(bid,j)
   shared_real(acc_var) = d_acc_variance(bid,isize+1-j)
endif

!---------------------------------------------------------
! fetch random numbers from global memory
!---------------------------------------------------------
if(j.eq.1) then
   shared_int(z1) = d_z(bid, i)
   shared_int(z2) = d_z(bid, i + isize)
   shared_int(z3) = d_z(bid, i + 2*isize)
   shared_int(z4) = d_z(bid, i + 3*isize)
```

```
    end if

    ! sync threads before starting main part of kernel
    call syncthreads()

    tmm_converged = .false.

    !-----------------------------------------------------------
    ! start iterations
    !-----------------------------------------------------------
    cuda_tmm_loop: do iter1 = itersave + 1, max( (nofiter)/(noforthо), 1)

       !-----------------------------------------------------
       ! carry out transfer-matrix multiplications (cuda2)
       !-----------------------------------------------------

       select case(iswapflag)

       ! perform one tmmult per iteration, then swap...
       case(1)

          northo_loop_swap: do iter2= 1, noforthо, 1
             call tmmult_device2(shared_real, energy, diagdis, isize, v,&
                  i, j, iter1, psi_a, psi_a_1, psi_a_m, psi_b, psi_b_1, &
                  psi_b_m, z1, z2, z3, z4, shared_int, irngflag, ibcflag)
             call cuda_swap(shared_real, psi_a, psi_b)
          enddo northo_loop_swap

       ! or perform two tmmult's per iteration
       case default

          northo_loop: do iter2= 1, noforthо, 2
             call tmmult_device2(shared_real, energy, diagdis, isize, v,&
                  i, j, iter1, psi_a, psi_a_1, psi_a_m, psi_b, psi_b_1, &
                  psi_b_m, z1, z2, z3, z4, shared_int, irngflag, ibcflag)
             call tmmult_device2(shared_real, energy, diagdis, isize, v,&
                  i, j, iter1, psi_b, psi_b_1, psi_b_m, psi_a, psi_a_1, &
                  psi_a_m, z1, z2, z3, z4, shared_int, irngflag, ibcflag)
          enddo northo_loop

       end select

       !-----------------------------------------------------------
       ! carry out gram-schmidt reorthonormalisation
       !-----------------------------------------------------------
       psi_a_ivec = i
       psi_b_ivec = psi_a_ivec + isize2

       do ivec = 1,isize
          call norm_device(isize,ivec,norm_sum,norm_1,&
               gamma,gamma2,psi_a,psi_b,i,j,shared_real,iwriteflag)
          call orth_device(isize,isize2,ivec,orth_sum,orth_1,i,j,&
               psi_a,psi_b,psi_a_ivec,psi_b_ivec,shared_real,iwriteflag)
          psi_a_ivec = psi_a_ivec + isize
          psi_b_ivec = psi_b_ivec + isize
       enddo

       !-----------------------------------------------------------
       ! calculate ngamma and check convergence !maydo: i==1
       !-----------------------------------------------------------
       call ngamma_device(isize,iwidth,noforthо,iter1,epsilon,&
            gamma,gamma2,ngamma,acc_var,acc_var_1,tmm_converged,&
            shared_real,bid)

       !-----------------------------------------------------------
       ! exit if converged
       !-----------------------------------------------------------
       if(tmm_converged) goto 900

       !-----------------------------------------------------------
       ! exit and relaunch kernel if nofprint exceeded
       !-----------------------------------------------------------
       select case(iwriteflag)
       case(0)
          continue

       case default
          if (mod(iter1,nofprint) == 0) goto 900

       end select

    end do cuda_tmm_loop

    ! no convergence in nofiter-loop
    iter1 = iter1 - 1

900 continue

    !-----------------------------------------------------------
    ! save number of iterations to global memory
    !-----------------------------------------------------------
    d_iter1(bid) = iter1

    !-----------------------------------------------------------
    ! save random numbers to global memory
    !-----------------------------------------------------------
    if(j.eq.1) then
       d_z(bid, i)          = shared_int(z1)
       d_z(bid, i + isize)  = shared_int(z2)
       d_z(bid, i + 2*isize) = shared_int(z3)
       d_z(bid, i + 3*isize) = shared_int(z4)
    end if

    !-----------------------------------------------------------
    ! copy psi, gammas, acc_variance and ngamma from shared to global mem
    !-----------------------------------------------------------
    d_psi_a(bid,j,i) = shared_real(psi_a)
    d_psi_b(bid,j,i) = shared_real(psi_b)
    if (i.eq.1) then
       d_gamma(bid,j) = shared_real(gamma)
       d_gamma2(bid,j) = shared_real(gamma2)
       d_acc_variance(bid,isize+1-j) = shared_real(acc_var)
       d_ngamma(bid,isize+1-j) = shared_real(ngamma)
    endif

end subroutine master_kernel

!-----------------------------------------------------------
! transfer-matrix multiplication device subroutine
!-----------------------------------------------------------

attributes(device) subroutine tmmult_device2(shared_real, en, diagdis, &
     m, v, i, j, iter1, psi_a, psi_a_1, psi_a_m, psi_b, psi_b_1, &
     psi_b_m, z1, z2, z3, z4, z, irngflag, ibcflag)

  ! input parameters
  integer, intent(in), value :: m, psi_a, psi_a_1, psi_a_m, psi_b, &
       psi_b_1, psi_b_m, v, i, j, z1, z2, z3, z4, irngflag, &
       ibcflag
  !integer, intent(in), value :: iter1
  integer(kind=8), intent(in), value :: iter1

  real(kind=rkind), intent(in), value ::  diagdis, en

  ! local/shared variables and arrays
  integer(kind=ikind), shared, dimension(*) :: z
  real(kind=rkind), shared, dimension(*) :: shared_real
  real(kind=rkind) :: psileft, psiright
  real(kind=rkind) :: r_test

  ! create new onsite potential
  select case(irngflag)
  case(0)

     if(j.eq.1) then
        shared_real(v) = en - diagdis * &
             (rlfsr113_device(z1,z2,z3,z4,z) - 0.5)
     endif

  case(1)
     ! dbg:
     if(j.eq.1) then
        shared_real(v) = mod(iter1,10) * 0.1
     endif

  end select

  ! sync after getting random number, and before updating psileft/right
  call syncthreads()

  ! calculate left wavefunction
  if (i.eq.1) then

     if (ibcflag.eq.0) then
        psileft= 0.0                    ! hard wall bc
     else if (ibcflag.eq.1) then
        psileft= shared_real(psi_a_m)   ! periodic bc
     else if (ibcflag.eq.2) then
        psileft= -shared_real(psi_a_m)  ! antiperiodic bc
     endif

  else
     psileft= shared_real(psi_a-1)
  endif

  ! calculate right wavefunction
  if (i.eq.m) then

     if (ibcflag.eq.0) then
        psiright= 0.0                   ! hard wall bc
     else if (ibcflag.eq.1) then
        psiright= shared_real(psi_a_1)  ! periodic bc
     else if (ibcflag.eq.2) then
        psiright= -shared_real(psi_a_1) ! antiperiodic bc
     endif

  else
     psiright= shared_real(psi_a+1)
  endif

  ! update wavefunction
  shared_real(psi_b)= shared_real(v) * shared_real(psi_a) &
       - (psileft + psiright) &
       - shared_real(psi_b)

end subroutine tmmult_device2

!-----------------------------------------------------------
! orthogonalisation device subroutine
!-----------------------------------------------------------
attributes(device) subroutine orth_device(m,m2,ivec,orth_sum,orth_1,i,j,&
     psi_a, psi_b, psi_a_ivec, psi_b_ivec, shared_real, iwriteflag)
```

```
  ! local variables
  integer,intent(in),value :: m, m2, ivec, orth_sum, orth_1, i, j, &
       psi_a, psi_b, psi_a_ivec, psi_b_ivec, iwriteflag
  integer :: s

  ! shared arrays
  real(kind=rkind),shared,dimension(*) :: shared_real

  !-----------------------------------------------------------
  ! orthogonalise (cuda2)
  !-----------------------------------------------------------

  ! calculate dot product <i|j>(k)
  if(j > ivec) then
     shared_real(orth_sum) = &
          shared_real(psi_a_ivec) * shared_real(psi_a) + &
          shared_real(psi_b_ivec) * shared_real(psi_b)
  end if

  ! sync threads before performing sum
  call syncthreads()

  ! calculate sum using reduction <i|j> = sum_k{<i|j>(k)}
  if (j > ivec) then
     if (m >= 64) shared_real(orth_sum) = &
          shared_real(orth_sum) + shared_real(orth_sum + 32)
     if (m >= 32) shared_real(orth_sum) = &
          shared_real(orth_sum) + shared_real(orth_sum + 16)
     if (m >= 16) shared_real(orth_sum) = &
          shared_real(orth_sum) + shared_real(orth_sum +  8)
     if (m >=  8) shared_real(orth_sum) = &
          shared_real(orth_sum) + shared_real(orth_sum +  4)
     if (m >=  4) shared_real(orth_sum) = &
          shared_real(orth_sum) + shared_real(orth_sum +  2)
     if (m >=  2) shared_real(orth_sum) = &
          shared_real(orth_sum) + shared_real(orth_sum +  1)
  end if

  ! subtract projection from the wavevectors |j> = |j> - <i|j><i|
  if(j > ivec) then
     shared_real(psi_a) = shared_real(psi_a) - &
          shared_real(orth_1) * shared_real(psi_a_ivec)
     shared_real(psi_b) = shared_real(psi_b) - &
          shared_real(orth_1) * shared_real(psi_b_ivec)
  end if

end subroutine orth_device

!-----------------------------------------------------------
! normalisation device subroutine
!-----------------------------------------------------------
attributes(device) subroutine norm_device(m,ivec,norm_sum,norm_1,&
     gamma,gamma2,psi_a,psi_b,i,j,shared_real,iwriteflag)
  ! local variables
  integer,intent(in),value :: m, ivec, norm_sum, norm_1, gamma, gamma2,&
       i, j, psi_a, psi_b, iwriteflag
  integer :: s
  real(kind=rkind) :: dummy

  ! shared arrays
  real(kind=rkind),shared,dimension(*) :: shared_real

  !-----------------------------------------------------------
  ! normalise (cuda2)
  !-----------------------------------------------------------

  ! calculate dot product <i|i>(k)
  if(j==ivec) then
     shared_real(norm_sum) = shared_real(psi_a)*shared_real(psi_a) &
          + shared_real(psi_b)*shared_real(psi_b)
  endif

  ! sync threads before performing sum
  call syncthreads()

  ! calculate total norm using sum reduction <i|i> = sum_k{<i|i>(k)}
  if (j==ivec) then
     if (m >= 64) shared_real(norm_sum) = &
          shared_real(norm_sum) + shared_real(norm_sum + 32)
     if (m >= 32) shared_real(norm_sum) = &
          shared_real(norm_sum) + shared_real(norm_sum + 16)
     if (m >= 16) shared_real(norm_sum) = &
          shared_real(norm_sum) + shared_real(norm_sum +  8)
     if (m >=  8) shared_real(norm_sum) = &
          shared_real(norm_sum) + shared_real(norm_sum +  4)
     if (m >=  4) shared_real(norm_sum) = &
          shared_real(norm_sum) + shared_real(norm_sum +  2)
     if (m >=  2) shared_real(norm_sum) = &
          shared_real(norm_sum) + shared_real(norm_sum +  1)
  end if

  ! normalise wavevectors |i> = |i> / <i|i>
  if(j==ivec) then
     dummy= 1.0/sqrt(real(shared_real(norm_1)))
     shared_real(psi_a) = dummy * shared_real(psi_a)
     shared_real(psi_b) = dummy * shared_real(psi_b)
  end if

  ! sync threads so that norm is ready for orth
  call syncthreads()

  !-----------------------------------------------------------
  ! calculate gamma
  !-----------------------------------------------------------
  if(i==1 .and. j==ivec) then
     dummy          = log(dummy)
     shared_real(gamma) = shared_real(gamma) - dummy
     shared_real(gamma2)= shared_real(gamma2) + dummy*dummy
  endif

end subroutine norm_device

! ------------------------------------------------------------
! ngamma_device:
!
  attributes(device) subroutine ngamma_device(isize,iwidth,nofortho,&
       iter1,epsilon,gamma,gamma2,ngamma,acc_var,acc_var_1,&
       tmm_converged,shared_real,bid)
    integer, intent(in), value :: isize, iwidth, nofortho, &
         gamma, gamma2, ngamma, acc_var, acc_var_1, bid
    !integer, intent(in), value :: iter1
    integer(kind=8), intent(in), value :: iter1
    real(kind=rkind), intent(in), value :: epsilon
    real(kind=rkind) :: thing
    logical :: tmm_converged

    real(kind=rkind),shared,dimension(*) :: shared_real

    shared_real(ngamma)= shared_real(gamma)/real(nofortho*iter1)

    thing = shared_real(gamma)/real(iter1)
    shared_real(acc_var)=              &
         sqrt( abs(                          &
         (shared_real(gamma2)/real(iter1) -         &
         thing*thing )      &
         / real( max(iter1 -1,1) )          &
         )) / abs( thing )

    !-----------------------------------------------------------
    ! check accuracy and dump the result
    !-----------------------------------------------------------
    if( iter1.ge.iwidth  .and. &
         iter1.ge.miniter ) then

       if( shared_real(acc_var_1).le.epsilon .and. &
            shared_real(acc_var_1).ge.tiny) then
          d_tmm_converged(bid)=.true.
          tmm_converged=.true.
       endif

    endif

  end subroutine ngamma_device

  !-----------------------------------------------------------
  ! cuda_swap:
  !
  ! (psi_a,psi_b)= (old,new) is the incoming vector, this is swapped
  ! into (psi_a,psi_b)= (new,old)
  attributes(device) subroutine cuda_swap(psi, a, b)

    use mynumbers
    integer, intent(in), value :: a, b
    real(kind=rkind), shared, dimension(*) :: psi
    real(kind=rkind) :: dummy

    dummy = psi(b)
    psi(b) = psi(a)
    psi(a) = dummy

  end subroutine cuda_swap

  !-----------------------------------------------------------
  !-----------------------------------------------------------
  ! cuda3 kernels
  !-----------------------------------------------------------
  !-----------------------------------------------------------

  ! ------------------------------------------------------------
  ! cuda3kernel:
  !
  attributes(global) subroutine cuda3kernel(energy, diagdis, epsilon, &
       m, iwidth, nofortho, nofiter, iwriteflag, idbgflag, &
       nofprint, irngflag, iswapflag, ibcflag)

    integer, intent(in), value :: m, iwidth, nofortho,&
         iwriteflag, idbgflag, nofprint, irngflag, iswapflag, &
         ibcflag
    integer(kind=8), intent(in), value :: nofiter
    real(kind=rkind), intent(in), value :: energy, diagdis, epsilon

    integer :: iter2, i, j, psi_a, psi_b, z1, z2, z3, z4, v, &
         norm_sum, orth_sum, ivec, norm_1, orth_1, psi_a_ivec, psi_b_ivec,&
         index, psi_a_1, psi_a_m, psi_b_1, psi_b_m, sync_count
    integer(kind=8) :: iter1, itersave
    real(kind=rkind) :: gamma, gamma2, ngamma, acc_var

    real(kind=rkind), shared, dimension(*) :: shared_real
    integer(kind=ikind), shared, dimension(*) :: shared_int

    !logical :: tmm_converged
```

```
! identify threads and blocks
i  = threadidx%x
j  = blockidx%x

!-----------------------------------------------------------
! load number of iterations from before
!-----------------------------------------------------------
itersave = d3_iter1

!-----------------------------------------------------------
! offset indices used for shared memory
!-----------------------------------------------------------

! shared memory indices unique to each thread in block
psi_a    = i
psi_b    = i +   m
norm_sum = i + 2*m
orth_sum = i + 3*m
v        = i + 4*m
z1       = i + 5*m
z2       = i + 6*m
z3       = i + 7*m
z4       = i + 8*m

! other shared memory indices
psi_a_1 = 1
psi_a_m = m
psi_b_1 = 1 +   m
psi_b_m = m +   m
norm_1  = 1 + 2*m
orth_1  = 1 + 3*m

!-----------------------------------------------------------
! initialise shared memory
!-----------------------------------------------------------
shared_real(psi_a)    = 0.0
shared_real(psi_b)    = 0.0
shared_real(norm_sum) = 0.0
shared_real(orth_sum) = 0.0
shared_real(v)        = 0.0
shared_int(z1)        = 0
shared_int(z2)        = 0
shared_int(z3)        = 0
shared_int(z4)        = 0

! reset sync_count used in gpu_sync
sync_count = 0

!-----------------------------------------------------------
! copy psi, gammas and acc_var from global mem to shared/local mem
!-----------------------------------------------------------
shared_real(psi_a) = d3_psi_a(j,i)
shared_real(psi_b) = d3_psi_b(j,i)

gamma = d3_gamma(j)
gamma2 = d3_gamma2(j)
acc_var = d3_acc_variance(m+1-j)

!-----------------------------------------------------------
! fetch random numbers from global memory
! (same for each block, different for each thread)
!-----------------------------------------------------------
shared_int(z1) = d3_z(i)
shared_int(z2) = d3_z(i + m)
shared_int(z3) = d3_z(i + 2*m)
shared_int(z4) = d3_z(i + 3*m)

! sync threads before starting main part of kernel
call syncthreads()

!tmm_converged = .false.
cuda_tmm_loop: do iter1 = itersave + 1, max( (nofiter)/(nofortho), 1)

   !-----------------------------------------------------
   ! carry out transfer-matrix multiplications (cuda3)
   !-----------------------------------------------------

   select case(iswapflag)

   ! perform one tmmult per iteration, then swap...
   case(1)

      northo_loop_swap: do iter2= 1, nofortho, 1
         call tmmult_cuda3(shared_real, energy, diagdis, m, v,&
              i, j, iter1, psi_a, psi_a_1, psi_a_m, psi_b, psi_b_1,&
              psi_b_m, z1, z2, z3, z4, shared_int, irngflag, ibcflag)
         call cuda_swap(shared_real, psi_a, psi_b)
      enddo northo_loop_swap

   ! or perform two tmmult's per iteration
   case default

      northo_loop: do iter2= 1, nofortho, 2
         call tmmult_cuda3(shared_real, energy, diagdis, m, v,&
              i, j, iter1, psi_a, psi_a_1, psi_a_m, psi_b, psi_b_1,&
              psi_b_m, z1, z2, z3, z4, shared_int, irngflag, ibcflag)
         call tmmult_cuda3(shared_real, energy, diagdis, m, v,&
              i, j, iter1, psi_b, psi_b_1, psi_b_m, psi_a, psi_a_1,&
              psi_a_m, z1, z2, z3, z4, shared_int, irngflag, ibcflag)
      enddo northo_loop

   end select

   !-----------------------------------------------------
   ! carry out gram-schmidt reorthonormalisation (cuda3)
   !-----------------------------------------------------
   do ivec = 1,m
      call norm_cuda3(m,ivec,norm_sum,norm_1,gamma,gamma2,&
           psi_a,psi_b,i,j,shared_real,iwriteflag)
      call orth_cuda3(m,ivec,orth_sum,orth_1,i,j,&
           psi_a,psi_b,shared_real,iwriteflag,sync_count)
   enddo

   !-----------------------------------------------------
   ! calculate ngamma and check convergence
   !-----------------------------------------------------
   call ngamma_cuda3(iwidth,nofortho,iter1,epsilon,&
        gamma,gamma2,ngamma,acc_var,shared_real,j,m)

   !-----------------------------------------------------
   ! perform gpu sync to wait for convergence flag to be loaded
   !-----------------------------------------------------
   call threadfence()
   call gpu_sync(-1,i,j)

   !-----------------------------------------------------
   ! exit if converged
   !-----------------------------------------------------
   !if(iter1==335) goto 900 ! dbg:
   if(d3_tmm_converged) goto 900

   !-----------------------------------------------------
   ! exit and relaunch kernel if nofprint exceeded
   !-----------------------------------------------------
   select case(iwriteflag)
   case(0)
      continue

   case default
      if ( mod(iter1,nofprint) == 0 ) goto 1000

   end select

end do cuda_tmm_loop

! no convergence in nofiter-loop
iter1 = iter1 - 1

900 continue

d3_kernel_finished = .true.

1000 continue

! save number of iterations to global memory
d3_iter1 = iter1

!-----------------------------------------------------
! save random numbers to global memory
!-----------------------------------------------------
if(j.eq.1) then

   d3_z(i)       = shared_int(z1)
   d3_z(i +   m) = shared_int(z2)
   d3_z(i + 2*m) = shared_int(z3)
   d3_z(i + 3*m) = shared_int(z4)
end if

!-----------------------------------------------------
! copy psi, gammas, acc_var and ngamma from shared/local to global mem
!-----------------------------------------------------
d3_psi_a(j,i) = shared_real(psi_a)
d3_psi_b(j,i) = shared_real(psi_b)
if (i.eq.1) then
   d3_gamma(j)             = gamma
   d3_gamma2(j)            = gamma2
   d3_acc_variance(m+1-j) = acc_var
   d3_ngamma(m+1-j)        = ngamma
endif

end subroutine cuda3kernel

attributes(device) subroutine tmmult_cuda3(shared_real, en, diagdis, &
     m, v, i, j, iter1, psi_a, psi_a_1, psi_a_m, psi_b, psi_b_1, &
     psi_b_m, z1, z2, z3, z4, z, irngflag, ibcflag)

! input parameters
integer, intent(in), value :: m, psi_a, psi_a_1, psi_a_m, psi_b, &
     psi_b_1, psi_b_m, v, i, j, z1, z2, z3, z4, irngflag, ibcflag
integer(kind=8), intent(in), value :: iter1
real(kind=rkind), intent(in), value ::  diagdis, en

! local/shared variables and arrays
integer(kind=ikind), shared, dimension(*) :: z
real(kind=rkind), shared, dimension(*) :: shared_real
real(kind=rkind) :: psileft, psiright
real(kind=rkind) :: r_test

! create new onsite potential
select case(irngflag)
case(0)
   shared_real(v) = en - diagdis*(rlfsr113_device(z1,z2,z3,z4,z) - 0.5)
```

```
   case default
      shared_real(v) = mod(iter1,10) * 0.1
   end select

   ! sync after getting random number, and before updating psileft/right
   call syncthreads()

   ! calculate left wavefunction
   if (i.eq.1) then

      if (ibcflag.eq.0) then
         psileft= 0.0                    ! hard wall bc
      else if (ibcflag.eq.1) then
         psileft= shared_real(psi_a_m)   ! periodic bc
      else if (ibcflag.eq.2) then
         psileft= -shared_real(psi_a_m)  ! antiperiodic bc
      endif

   else
      psileft= shared_real(psi_a-1)
   endif

   ! calculate right wavefunction
   if (i.eq.m) then

      if (ibcflag.eq.0) then
         psiright= 0.0                   ! hard wall bc
      else if (ibcflag.eq.1) then
         psiright= shared_real(psi_a_1)  ! periodic bc
      else if (ibcflag.eq.2) then
         psiright= -shared_real(psi_a_1) ! antiperiodic bc
      endif

   else
      psiright= shared_real(psi_a+1)
   endif

   ! update wavefunction
   shared_real(psi_b)= shared_real(v) * shared_real(psi_a) &
        - (psileft + psiright) - shared_real(psi_b)

end subroutine tmmult_cuda3

!---------------------------------------------------------
! normalisation device subroutine
!---------------------------------------------------------
attributes(device) subroutine norm_cuda3(m,ivec,norm_sum,norm_1,&
     gamma,gamma2,psi_a,psi_b,i,j,shared_real,iwriteflag)
   ! local variables
   integer,intent(in),value :: m, ivec, norm_sum, norm_1, &
        i, j, psi_a, psi_b, iwriteflag
   real(kind=rkind) :: dummy, gamma, gamma2

   ! shared arrays
   real(kind=rkind),shared,dimension(*) :: shared_real

   !---------------------------------------------------------
   ! normalise (cuda3)
   !---------------------------------------------------------

   ! calculate dot product <i|i>(k)
   if(j == ivec) then
      shared_real(norm_sum) = shared_real(psi_a)*shared_real(psi_a) &
           + shared_real(psi_b)*shared_real(psi_b)
   endif

   ! sync threads before performing sum
   call syncthreads()

   ! calculate total norm using sum reduction <i|i> = sum_k{<i|i>(k)}

   ! carry out reduction across warps
   if (m >= 512) then
      if (j == ivec .and. i <= 256) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum + 256)
      call syncthreads()
   end if

   if (m >= 256) then
      if (j == ivec .and. i <= 128) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum + 128)
      call syncthreads()
   end if

   if (m >= 128) then
      if (j == ivec .and. i <=  64) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum +  64)
      call syncthreads()
   end if

   if (m >= 64) then
      if (j == ivec .and. i <=  32) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum +  32)
      call syncthreads()
   end if

   ! carry out reduction inside warp
   if (j==ivec) then
      ! if (m >= 64) shared_real(norm_sum) = &
      !      shared_real(norm_sum) + shared_real(norm_sum + 32)
      if (m >= 32) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum + 16)
      if (m >= 16) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum +  8)
      if (m >=  8) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum +  4)
      if (m >=  4) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum +  2)
      if (m >=  2) shared_real(norm_sum) = &
           shared_real(norm_sum) + shared_real(norm_sum +  1)
   end if

   ! normalise wavevectors |i> = |i> / <i|i>
   if(j == ivec) then
      dummy= 1.0/sqrt(real(shared_real(norm_1)))
      shared_real(psi_a) = dummy * shared_real(psi_a)
      shared_real(psi_b) = dummy * shared_real(psi_b)
   end if

   ! sync threads so that norm is ready for orth
   call syncthreads()  ! maydo: investigate whether this is necessary

   !---------------------------------------------------------
   ! calculate gamma
   !---------------------------------------------------------
   if(j == ivec) then ! maydo: also put if(i==1), do same for ngamma_cuda3
      dummy  = log(dummy)
      gamma  = gamma - dummy
      gamma2 = gamma2 + dummy*dummy
   endif

end subroutine norm_cuda3

!---------------------------------------------------------
! orthogonalisation device subroutine (cuda3)
!---------------------------------------------------------
attributes(device) subroutine orth_cuda3(m,ivec,orth_sum,orth_1,i,j,&
     psi_a,psi_b,shared_real,iwriteflag,sync_count)
   ! local variables
   integer,intent(in),value :: m, ivec, orth_sum, orth_1, i, j, &
        psi_a, psi_b, iwriteflag
   integer :: sync_count
   real(kind=rkind) :: psi_a_ivec, psi_b_ivec

   ! shared arrays
   real(kind=rkind),shared,dimension(*) :: shared_real

   !---------------------------------------------------------
   ! orthogonalise (cuda3)
   !---------------------------------------------------------

   ! load psi from block ivec into global memory
   if(j == ivec) then
      d3_psi_a(ivec,i) = shared_real(psi_a)
      d3_psi_b(ivec,i) = shared_real(psi_b)
   end if

   ! inter-block sync to wait for global memory to finish loading
   sync_count = sync_count + 1
   call threadfence() ! this flushes the cached global memory
   call gpu_sync(sync_count, i, j)

   ! load psi from global memory into psi_ivec
   if(j > ivec) then
      psi_a_ivec = d3_psi_a(ivec,i)
      psi_b_ivec = d3_psi_b(ivec,i)
   end if

   ! calculate dot product <i|j>(k)
   if(j > ivec) then
      shared_real(orth_sum) = &
           psi_a_ivec * shared_real(psi_a) + &
           psi_b_ivec * shared_real(psi_b)
   end if

   ! sync threads before performing sum
   call syncthreads()

   ! calculate sum using reduction <i|j> = sum_k{<i|j>(k)}

   ! carry out reduction across warps
   if (m >= 512) then
      if (j > ivec .and. i <= 256) shared_real(orth_sum) = &
           shared_real(orth_sum) + shared_real(orth_sum + 256)
      call syncthreads()
   end if

   if (m >= 256) then
      if (j > ivec .and. i <= 128) shared_real(orth_sum) = &
           shared_real(orth_sum) + shared_real(orth_sum + 128)
      call syncthreads()
   end if

   if (m >= 128) then
      if (j > ivec .and. i <= 64) shared_real(orth_sum) = &
           shared_real(orth_sum) + shared_real(orth_sum + 64)
      call syncthreads()
   end if

   if (m >= 64) then
      if (j > ivec .and. i <= 32) shared_real(orth_sum) = &
           shared_real(orth_sum) + shared_real(orth_sum + 32)
```

```
      call syncthreads()
    end if

    ! carry out reduction inside warp
    if (j > ivec) then
       ! if (m >= 64) shared_real(orth_sum) = &
       !      shared_real(orth_sum) + shared_real(orth_sum + 32)
       if (m >= 32) shared_real(orth_sum) = &
            shared_real(orth_sum) + shared_real(orth_sum + 16)
       if (m >= 16) shared_real(orth_sum) = &
            shared_real(orth_sum) + shared_real(orth_sum +  8)
       if (m >=  8) shared_real(orth_sum) = &
            shared_real(orth_sum) + shared_real(orth_sum +  4)
       if (m >=  4) shared_real(orth_sum) = &
            shared_real(orth_sum) + shared_real(orth_sum +  2)
       if (m >=  2) shared_real(orth_sum) = &
            shared_real(orth_sum) + shared_real(orth_sum +  1)
    end if

    ! subtract projection from the wavevectors |j> = |j> - <i|j><i|
    if(j > ivec) then
       shared_real(psi_a) = shared_real(psi_a) - &
            shared_real(orth_1) * psi_a_ivec
       shared_real(psi_b) = shared_real(psi_b) - &
            shared_real(orth_1) * psi_b_ivec
    end if

  end subroutine orth_cuda3

  ! ------------------------------------------------------------
  ! ngamma_cuda3:
  !
  attributes(device) subroutine ngamma_cuda3(iwidth,nofortho,&
      iter1,epsilon,gamma,gamma2,ngamma,acc_var,shared_real,j,m)
    integer, intent(in), value :: iwidth, nofortho, j, m
    integer(kind=8), intent(in), value :: iter1
    real(kind=rkind), intent(in), value :: epsilon
    real(kind=rkind) :: thing, gamma, gamma2, ngamma, acc_var

    real(kind=rkind),shared,dimension(*) :: shared_real
    !logical :: tmm_converged

    ngamma= gamma/real(nofortho*iter1)

    thing = gamma/real(iter1)
    acc_var =             &
         sqrt( abs(                        &
         (gamma2/real(iter1) -         &
         thing*thing )      &
         / real( max(iter1 -1,1) )          &
         )) / abs( thing )

    !------------------------------------------------------------
    ! check accuracy and dump the result
    !------------------------------------------------------------
    if(j == m) then
       if( iter1.ge.iwidth  .and. &
            iter1.ge.miniter ) then

          if( acc_var.le.epsilon .and. &
               acc_var.ge.tiny) then
             d3_tmm_converged=.true.
          endif

       endif
    endif

  end subroutine ngamma_cuda3

  !------------------------------------------------------------
  ! gpu_sync
  !------------------------------------------------------------
  attributes(device) subroutine gpu_sync(goalval, tid, bid)

    integer, intent(in), value :: goalval, tid, bid

    ! only thread 1 is used for synchronisation
    if(tid == 1) arrayin(bid) = goalval

    if(bid == 1) then
       do while(arrayin(tid) /= goalval)
          ! use atomic operation
          d_atomic(bid,tid) = atomiccas(d_atomic(bid,tid), 0, 1)
       end do
       call syncthreads()

       arrayout(tid) = goalval
    end if

    if(tid == 1) then
       do while(arrayout(bid) /= goalval)
          ! use atomic operation
          d_atomic(bid,tid) = atomiccas(d_atomic(bid,tid), 0, 1)
       end do
    end if
    call syncthreads()

  end subroutine gpu_sync

end module my_kernels
```

A.4 `random.f90`

This source code contains the random number generator used in the Serial-TMM. The RNG for CUDA-TMM would have been put in here, but in CUDA FORTRAN when one calls a kernel, one can only call subroutines that are contained within the same module as that kernel. So the CUDA-RNG had to be contained in `cuda_util.f90`.

```
module RNG_RLFSR113
  use MyNumbers
  implicit none

  ! accessibility
  private
  public :: rlfsr113
  public :: lfsrinit

  ! variables
  integer(kind=ikind) :: z1, z2, z3, z4

  ! parameter
  real(kind=rkind),    parameter :: AM = 4.656612873077d-10
  integer(kind=ikind), parameter :: IA = 16807
  integer(kind=ikind), parameter :: IM = 2147483647
  integer(kind=ikind), parameter :: IQ = 127773
  integer(kind=ikind), parameter :: IR = 2836

contains

  ! ------------------------------------------------------------
  ! rlfsr113()  returns a random number of interval (0, 1)
  ! ------------------------------------------------------------
  function rlfsr113() result(dRet)
    real(kind=rkind)        :: dRet

    integer(kind=ikind) :: b

    b  = ishft(ieor(ishft(z1,6),z1),-13)
    z1 = ieor(ishft(iand(z1,-2),18),b)

    b  = ishft(ieor(ishft(z2,2),z2),-27)
    z2 = ieor(ishft(iand(z2,-8),2),b)

    b  = ishft(ieor(ishft(z3,13),z3),-21)
    z3 = ieor(ishft(iand(z3,-16),7),b)

    b  = ishft(ieor(ishft(z4,3),z4),-12)
    z4 = ieor(ishft(iand(z4,-128),13),b)

    dRet=ishft(ieor(ieor(ieor(z1,z2),z3),z4),-1)*AM

  end function rlfsr113

  ! ------------------------------------------------------------
  ! lfsrinit()  initialize rlfsr113 (z1,z2,z3,z4)
  ! ------------------------------------------------------------
  subroutine lfsrinit(idum)
    integer(kind=ikind), intent(inout) :: idum

    integer(kind=ikind) :: k,c1,c2,c3,c4

    ! Check whether the FORTRAN integers can be used as unsigned long !

    ! data c1 /B'11111111111111111111111111111110'/
    !  data c1 /X'FFFFFFFE'/
    c1 = Z"FFFFFFFE"
```

```
    ! data c2 /B'11111111111111111111111111111000'/
    !  data c2 /X'FFFFFFF8'/
    c2 = Z"FFFFFFF8"
    ! data c3 /B'11111111111111111111111111110000'/
    !  data c3 /X'FFFFFFF0'/
    c3 = Z"FFFFFFF0"
    ! data c4 /B'11111111111111111111111110000000'/
    !  data c4 /X'FFFFFF80'/
    c4 = Z"FFFFFF80"

    if ((c1.ne.-2).or.(c2.ne.-8).or.(c3.ne.-16).or.(c4.ne.-128)) then
      print *,"c1,c2,c3,c4", c1,c2,c3,c4
      print *,'Nonstandard integer representation. Stoped.'
      stop
    endif

    ! Initialize z1,z2,z3,z4

    if (idum.le.0) idum=1
    k=(idum)/IQ
    idum=IA*(idum-k*IQ)-IR*k
    if (idum.lt.0) idum = idum + IM
    if (idum.lt.2) then
      z1=idum+2
    else
      z1=idum
    endif
    k=(idum)/IQ
    idum=IA*(idum-k*IQ)-IR*k
    if (idum.lt.0) idum = idum + IM
    if (idum.lt.8) then
      z2=idum+8
    else
      z2=idum
    endif
    k=(idum)/IQ
    idum=IA*(idum-k*IQ)-IR*k
    if (idum.lt.0) idum = idum + IM
    if (idum.lt.16) then
      z3=idum+16
    else
      z3=idum
    endif
    k=(idum)/IQ
    idum=IA*(idum-k*IQ)-IR*k
    if (idum.lt.0) idum = idum + IM
    if (idum.lt.128) then
      z4=idum+128
    else
      z4=idum
    endif

  end subroutine lfsrinit

end module RNG_RLFSR113

! ------------------------------------------------------------------
!
! MODULE RandomNumberGenerator                          random.f90
! RANDOM - Standard F77/F90 interface for random number generators
!
! ------------------------------------------------------------------

module RNG
  use RNG_RLFSR113
  use MyNumbers

  implicit none

  ! accessibility
  private
  public :: SRANDOM
  public :: DRANDOM
  public :: GRANDOM

  ! parameter

!  ! kind parameter for double precision
!  integer, parameter :: PRECISION = 8
  ! number of random number inside [0,1] for routine gauss
  integer, parameter :: GAUSS_N    = 20
contains

  ! ------------------------------------------------------------------
  ! SRANDOM()
  !
  ! Random number generator SEED interface for use with any old RND
  ! ------------------------------------------------------------------
  subroutine SRANDOM( ISeed )
    integer, intent(in) :: ISeed
    real(kind=rkind)    :: dummy

    integer(kind=ikind) idum

    ! change following lines to incorporate different RND generators
    idum = ISeed
    call lfsrinit(idum)

    ! Make a single call to rlfsr113() to achieve a valid state
    dummy=rlfsr113()

  end subroutine SRANDOM

  ! ------------------------------------------------------------------
  ! DRANDOM()
  !
  ! Random number generator interface for use with any old RND
  ! ------------------------------------------------------------------
  function DRANDOM( ISeed ) result(dRet)
    integer, intent(in) :: ISeed
    real(kind=rkind):: dRet

    ! change following lines to incorporate different RND generators
    dRet = rlfsr113()  ! NOTE that ISeed is never used

  end function DRANDOM

  ! ------------------------------------------------------------------
  ! GRANDOM()
  !
  ! Gaussian random number generator interface
  ! ------------------------------------------------------------------
  function GRANDOM( ISeed, avg, sigma) result (dRet)
    integer,          intent(in) :: ISeed
    real(kind=rkind), intent(in) :: avg
    real(kind=rkind), intent(in) :: sigma
    real(kind=rkind)             :: dRet


    ! change following lines to incorporate different RND generators
    call gauss(dRet, sigma, avg) ! NOTE that ISeed is never used

  end function GRANDOM

  ! ------------------------------------------------------------------
  ! GAUSS()
  !
  ! THE ROUTINE GAUSS GENERATES A RANDOM NUMBER
  ! IN A GAUSSIAN DISTRIBUTION
  !
  ! VARIABLES:
  !
  !   X     - THE OUTPUT GAUSSIAN VARIABLE
  !   sigma - standard deviation
  !   mu    - average
  !
  ! NOTE: The random number generator rlfsr113 should be
  !   initialised by calling
  !   the subroutine lfsrinit
  ! ------------------------------------------------------------------
  subroutine gauss(X,sigma,mu)
    real(kind=rkind), intent(out):: X
    real(kind=rkind), intent(in) :: sigma
    real(kind=rkind), intent(in) :: mu

    real(kind=rkind) :: Y, SUM
    integer i

    SUM=0.0
    do i=1,GAUSS_N
      Y=rlfsr113()
      Y=2._RKIND*(Y-0.5_RKIND)
      SUM=SUM+Y
    end do

    X=mu+sigma*SUM* DSQRT( 3.0_RKIND /DBLE(GAUSS_N) )

  end subroutine gauss

end module RNG
```

Bibliography

[1] P. W. Anderson. "Absence of Diffusion in Certain Random Lattices". In: *Phys. Rev.* 109 (1958), 14921505.

[2] P-E. Wolf and G. Maret. "Weak Localization and Coherent Backscattering of Photons in Disordered Media". In: *Phys. Rev. Lett.* 55 (1985), p. 2696.

[3] Y. Kuga and A. Ishimaru. "Retroreflectance from a Dense Distribution of Spherical Particles". In: *J. Opt. Soc. Am. A* 1 (1984), pp. 831–835.

[4] M. P. van Albada and A. Lagendijk. "Observation of Weak Localization of Light in a Random Medium". In: *Phys. Rev. Lett.* 55 (1985), pp. 2692–2695.

[5] J. Hanberg P. E. Lindelof J. Nørregaard. "New Light on the Scattering Mechanisms in Si Inversion Layers by Weak Localization Experiments". In: *Phys. Scr.* T14 (1986), pp. 17–26.

[6] T. R. Kirkpatrick. "Localization of Acoustic Waves". In: *Phys. Rev. B* 31 (1985), pp. 5746–5755.

[7] J. Kergomard C. Depollier and F. Laloe. "Localisation d'Anderson des Ondes dans les Réseaux Acoustiques Unidimensionnels Aléatoires". In: *Ann. Phys. Fr.* 11 (1986), p. 457.

[8] B. Kramer and A. MacKinnon. "Localization: Theory and Experiment". In: *Rep. Prog. Phys.* 56.1469 (1993).

[9] D. C. Licciardello E. Abrahams P. W. Anderson and T. V. Ramakrishnan. "Scaling Theory of Localization: Absence of Quantum Diffusion in Two Dimensions". In: *Phys. Rev. Lett.* 42 (1979), pp. 673–676.

[10] T. V. Ramakrishnan P. A. Lee. "Disordered Electronic Systems". In: *Rev. Mod. Phys.* 57.287 (1985).

[11] D. Belitz and T. R. Kirkpatrick. "The Anderson-Mott transition". In: *Rev. Mod. Phys.* 66 (1994), pp. 261–380.

[12] R. Landauer. "Electrical Resistance of Disordered One-Dimensional Lattices". In: *Phil. Mag.* 21.172 (1970), pp. 863–867.

[13] R. A. Römer and M. Schreiber. "Numerical investigations of scaling at the Anderson transition". In: *The Anderson Transition and its Ramifications – Localisation, Quantum Interference, and Interactions* (2003), pp. 3–19.

[14] A. MacKinnon and B. Kramer. "One-Parameter Scaling of Localization Length and Conductance in Disordered Systems". In: *Phys. Rev. Lett.* 47.21 (1981), pp. 1546–1549.

[15] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Volume 1: Theory*. Birkhäuser, Boston, 1985.

[16] J. K. Cullum and R. A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Volume 2: Programs*. Birkhäuser, Boston, 1985. URL: http://www.netlib.org/lanczos.

[17] D. J. Thouless. "Electrons in Disordered Systems and the Theory of Localization". In: *Phys. Rep.* 13.3 (1974), pp. 93 –142.

[18] B. Kramer and M. Schreiber. "Transfer-Matrix Methods and Finite-Size Scaling for Disordered Systems". In: *Computational Physics Selected Methods, Simple Exercises, Serious Applications*. Springer-Verlag Berlin, Heidelberg GmbH, and Co., 1996, pp. 166–188.

[19] R. A. Römer A. Eilmes A. M. Fischer. "Critical Parameters for the Disorder-Induced Metal-Insulator Transition in FCC and BCC Lattices". In: *Phys. Rev. B* 77 (24 2008).

[20] V. Oseledec. "A Multiplicative Ergodic Theorem". In: *Trans. Mosc. Math. Soc.* 19 (1968), pp. 197–231.

[21] F. Wegner M. Kappus. "Anomaly in the Band Centre of the One-Dimensional Anderson Model". In: *Z Phys. B Cond. Matt.* 45 (1981), pp. 15–21.

[22] H. Schulz-Baldes R. A. Römer. "Weak Disorder Expansion for Localisation Lengths of Quasi-1D Systems". In: *Europhys. Lett.* 68 (2004), pp. 247–253.

[23] D. Bau III L. N. Trefethen. *Numerical Linear Algebra*. Siam, 1997.

[24] R. A. Römer. "From Localization to Delocalization: Numerical Studies of Transport in Disordered Systems". PhD thesis. TU Chemnitz, 1999.

[25] M. Schneider B. Milde. "Parallel Implementation of Classical Gram-Schmidt Orthogonalization on CUDA Graphics Cards". 2009. URL: http://www.cdc.informatik.tu-darmstadt.de/~mischnei/CUDA_GS_2009_12_22.pdf.

[26] *Official Website of OpenCL*. 2011. URL: http://www.khronos.org/opencl/.

[27] *NVidia Tesla M2050 Technical Specifications*. NVidia. 2010. URL: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_M2050_M2070_Apr10_LowRes.pdf.

[28] *CUDA C Programming Guide*. NVidia. 2011. URL: http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[29] *Tesla C1060 Datasheet*. NVidia. 2010. URL: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jan10_lores_r1.pdf.

[30] M. Sadooghi-Alvandi H. Wong M-M. Papadopoulou and A. Moshovos. "Demystifying GPU Microarchitecture through Microbenchmarking". In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 235–246.

[31] M. Sadooghi-Alvandi H. Wong M-M. Papadopoulou and A. Moshovos. *Demystifying GPU Microarchitecture through Microbenchmarking: Source Code*. 2010. URL: http://www.stuffedcow.net/research/cudabmk.

[32] M. Harris. *NVidia Fermi Compute Architecture Whitepaper*. NVidia. 2010. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[33] S. Xiao and W. Feng. *Inter-Block GPU Communication via Fast Barrier Synchronization*. Tech. rep. TR-09-19. Dept. of Computer Science, Virginia Tech., 2009.

[34] M. Harris. *Optimizing Parallel Reduction in CUDA*. Tech. rep. NVidia, 2007. URL: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

[35] *Portland Group CUDA Fortran Programming Guide and Reference*. Portland Group. 2011. URL: http://www.pgroup.com/doc/pgicudaforug.pdf.

[36] *Sorting Networks and Their Applications*. Vol. 32. AFIPS. 1968, pp. 307–314.

[37] T. Smith and M. Waterman. "Identification of Common Molecular Subsequences". In: *J. Mol. Biol.* ().

[38] J. W. Cooley and O. W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Math. Comput.* 19 (1965), pp. 297–301.

[39] G. D. Bergland. "A Guided Tour of the Fast Fourier Transform". In: *IEEE Spec.* 6 (1969), pp. 41–52.

[40] T. Vojta. *Random Number Generator RLFSR113*. 1998. URL: http://www.physics.udel.edu/~bnikolic/teaching/phys660/F90/rlfsr113.f90.

[41] P. L'Ecuyer. "Tables of Maximally-Equidistributed Combined LFSR Generators". In: *Math. Comput.* 68.225 (1999), pp. 261–269.

[42] S. Gerschgorin. "Über die Abgrenzung der Eigenwerte einer Matrix". In: *Bulletin de l'Académie des Sciences de l'URSS* (6 1931), pp. 749–754.

[43] *PTX: Parallel Thread Execution ISA Version 2.0*. NVidia. 2010. URL: http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf.