Quantum Transport Modelling with GPUs

Mohammed Harb
Center for the Physics of Materials
Department of Physics
McGill University
Montreal, Quebec
2012

A Thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

To my parents Zeinab and Aziz,

Contents

List of Figures

Abstract

In this thesis, we have developed a parallel GPU accelerated code for carrying out transport calculations within the Non-Equilibrium Green's Function (NEGF) framework using the Tight-Binding (TB) model. We also discuss the theoretical, modelling, and computational issues that arise in this implementation. We demonstrate that a heterogenous implementation with CPUs and GPUs is superior to single processor, multiple processor, and massively parallel CPU-only implementations.

The GPU-Matlab Interface (GMI) developed in this work for use in our NEGF-TB code is not application specific and can be used by researchers in any field without previous knowledge of GPU programming or multi-threaded programming. We also demonstrate that GMI competes very well with commercial packages.

Finally, we apply our heterogenous NEGF-TB code to the study of electronic transport properties of Si nanowires and nanobeams. We investigate the effect of several kinds of structural defects on the conductance of such devices and demonstrate that our method can handle systems of over 200,000 atoms in a reasonable time scale while using just 1-4 GPUs.

Résumé

Dans cette thèse, nous présentons un logiciel qui effectue des calculs de transport quantique en utilisant conjointement la théorie des fonctions de Green hors équilibre (non equilibrium Green function, NEGF) et le modèle des liens troits (tight-binding model, TB). Notre logiciel tire avantage du parallélisme inhérent aux algorithmes utilisés en plus d'être accéléré grâce à l'utilisation de processeurs graphiques (GPU). Nous abordons également les problèmes théoriques, géométriques et numériques qui se posent lors de l'implémentation du code NEGF-TB. Nous démontrons ensuite qu'une implémentation hétérogène utilisant des CPU et des GPU est supérieure aux implémentations à processeur unique, à celles à processeurs multiples, et même aux implémentations massivement parallèles n'utilisant que des CPU.

Le GPU-Matlab Interface (GMI) présenté dans cette thèse fut développé pour des fins de calculs de transport quantique NEGF-TB. Néanmoins, les capacités de GMI ne se limitent pas à l'utilisation que nous en faisons ici et GMI peut être utilisé par des chercheurs de tous les domaines nayant pas de connaissances préalables de la programmation GPU ou de la programmation "multithread". Nous démontrons également que GMI compétitionne avantageusement avec des logiciels commerciaux similaires.

Enfin, nous utilisons notre logiciel NEGF-TB pour étudier certaines pro-

priétés de transport électronique de nanofils de Si et de nanobeams. Nous examinons l'effet de plusieurs sortes de lacunes sur la conductance de ces structures et démontrons que notre méthode peut étudier des systèmes de plus de 200 000 atomes en un temps raisonnable en utilisant de un à quatre GPU sur un seul poste de travail.

Statement of Originality

My goal in this thesis work is to identify and solve the issues that cripple computational performance in the study of the electronic transport properties of nanoscale systems, and to use highly specialized state-of-the-art techniques to simulate systems consisting of a very large number of particles. Specifically, my main contributions to this work are:

- I developed a scalable, multi-threaded interface for connecting Matlab to the GPU to accelerate linear algebra operations. My software is called the GPU-Matlab Interface (GMI) and is written in C. It is built upon CUDA, CULA and POSIX Threads and is capable of handling several GPU devices in parallel and without requiring the Matlab Parallel Computing Toolbox (PCT) or other proprietary toolboxes. GMI's performance also competes very well with commercial software and can be readily used by users with no previous experience with GPUs or multi-threaded programming. Some benchmarks, applications and results are were included in a recent review article [1].

- I used GMI to develop a parallelized heterogenous CPU/GPU code for carrying out transport calculations within the Non-Equilibrium Green's

Function (NEGF) framework using the Tight-Binding (TB) model. Using this code, I investigated the transport properties of pure Si nanowires and nanobeams and demonstrated that my method can handle structures containing over 200,000 atoms on a very reasonable timescale of several hours.

Matlab and PCT are trademarks of Mathworks. CUDA, CULA, MAGMA, POSIX Threads, and Jacket are trademarks of Nvidia, EM Photonics, The Computational Algebra Group at the School of Mathematics and Statistics at the University of Sydney, The Lawrence Livermore National Laboratory, and Accelereyes respectively.

Acknowledgments

I thank my supervisor Prof. Hong Guo not only for his guidance and support, but also for introducing me to the field of nanoelectronics, which I have become very passionate about. His enthusiasm and drive have enabled me to discover the joy of scientific research. I also sincerely appreciate the genuine interest he takes in his students and the valuable wisdom he imparts on what it takes to succeed as a scientist.

Interacting with the talented graduate students, post-docs, and research associates in the group on a daily basis has taught me more than any classroom could. I thank Dr. Dongping Liu, Dr. Yibin Hu, Dr. Jinghzhe Chen, Vincent Michaud-Rioux, Dr. Lei Liu, Dr. Yu Zhu, and Dr. Kevin Zhu for their assistance at various stages of the project.

Lastly, I thank my friends and family for their warmth and endearment. I have had an extraordinary experience at McGill, and formed many fond memories. I'm lucky to have shared it with such outstanding individuals.

1

Introduction

The first microprocessor was invented by Intel in 1971 and contained a few thousand transistors[2]. A few years later, Gordon Moore made his famous observation that the number of components in integrated circuits was doubling approximately every two years [3]. Since then, electronic devices have been scaled down relentlessly. Today's microprocessors are crammed with billions of transistors. Figure 1.1 shows the number of transistors in microprocessors vs the year of introduction since 1971.

The components of today's electronics are on the order of tens of nanometers and can truly be regarded as quantum mechanical devices. Their electronic behaviour is best understood with quantum mechanical modelling using first principles theories like Density Functional Theory (DFT)[5, 6, 7, 8]. However, the computational cost of these parameterless *ab initio* techniques makes them impractical for large systems. Empirical methods, on the other hand, are several orders of magnitude faster but not sufficiently accurate. Tight-binding modelling is the intermediate solution. It has the advantage of retaining quantum

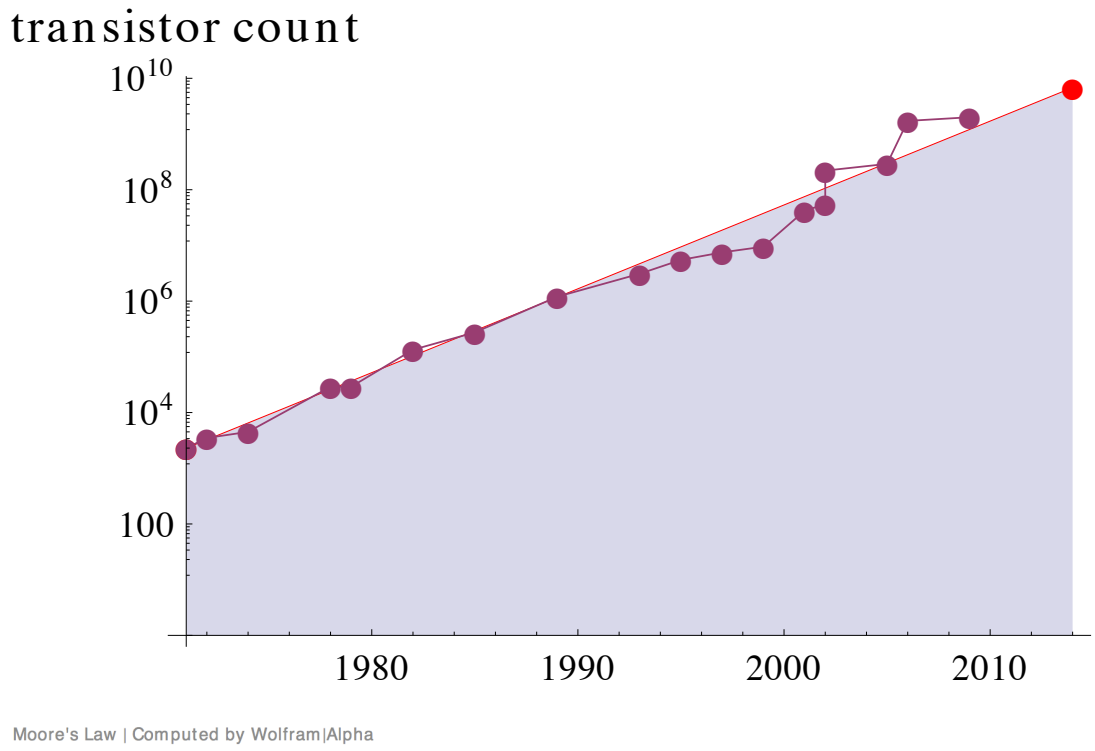


Figure 1.1: The number of transistors in microprocessors has doubled every two years. Data retrieved from Wolfram|Alpha Pro[4].

mechanical details, while being 2-3 orders of magnitude faster when compared to *ab initio* methods [9].

Both techniques, first principles modelling using DFT and tight-binding modelling, can be combined with the Non-Equilibrium Green's Function (NEGF) formalism to form a solid foundation for the study of electronic transport theory in meso-scale systems[10, 11, 12, 13, 14]. The basic idea is to calculate the Hamiltonian self-consistently with DFT or non-self-consistently using tight-binding modelling, and apply NEGF to study the electronic transport properties of the system.

Both approaches, however, share a common persistent bottleneck. The size of the matrices involved in NEGF-DFT and NEGF-TB calculations scales non-linearly with the number of simulated particles and is usually the factor most responsible for limiting the size of systems that can be studied. One can simply utilize more cores in a supercomputer, but this is excessive and inelegant. Typical massively parallel supercomputers also consume enormous amounts of power. Japan's K Computer, consisting of 705,024 cores, consumes 9.89 MW of power - about half of the peak power output of a nuclear submarine - most of which goes into cooling the components. It is clear that CPU-only supercomputers cannot be scaled much more and that more specialized, application specific hardware needs to be utilized if computing power is to continue growing at the current rate.

GPUs, once used only for video processing applications, have recently gained attention for their impressive floating point capabilities and low power consumption [15, 16, 17]. They contain hundreds of cores and are ideal for performing computationally intensive problems that have crippled performance in the past. Typically, these highly specialized devices are used alongside general purpose processors as part of a heterogenous computing scheme, where the sequential parts of the application are run on traditional processors and the numerically intensive parts are sent to the GPU for processing.

To tap into this promising computational resource and overcome the ever-present linear algebra bottleneck, we developed a package to reroute linear

algebra calls by Matlab to the GPU for acceleration. We also developed a parallel GPU accelerated code that combines the tight-binding model and the non-equilibrium Green's function (NEGF) formalism to study the transport properties of nanoscale systems such as silicon nanowires and nanobeams.

In chapter 2, we discuss the length scales at which a quantum mechanical treatment is necessary to understand device physics and briefly review the theoretical formalisms, such as the Landauer formalism and the non-equilibrium Green's function formalism, used in this work.

Chapter 3 addresses the modelling techniques used in the study of electronic transport in mesoscopic systems such as the Principal Layer (PL) algorithm and the iterative Transfer Matrix algorithm for calculating self-energies. Two block-tridiagonal matrix inversion algorithms, the Recursive Green's Function (RGF) algorithm and the Generalized Green's Function (GGM), algorithm are also presented.

Chapter 4 addresses the computational details of the work. We argue in favour of a heterogenous computing scheme instead of a massively parallel CPU-only implementation and describe how it fits in the context of our NEGF-TB code. We also introduce the GPU-Matlab Interface (GMI) developed in this work and compare its performance to several types of CPU implementations and several commercial GPU interfaces for Matlab.

In chapter 5, we use the NEGF-TB code developed in this work in combination with GMI to study the transport properties of Si nanowires and nanobeams.

The effects of periodic and random vacancies are also considered.

Finally, chapter 6 presents a summary of the thesis, future developments, and some concluding remarks.

The appendices contain a reference manual for GMI as well as a sample code for one of GMI's functions, with some comments.

2

Quantum Transport

2.1 The Mesoscopic Regime

The current through a macroscopic conductor is proportional to A the cross sectional area, and L, the length of the conductor. The conductance of such a macroscopic conductor can be written as:

$$G = \sigma \frac{A}{L} \tag{2.1}$$

Here, σ, the electrical conductivity, depends specifically on the material the conductor is composed of. As the conductor is scaled down to small length-scales, the discrete properties of the material need to be taken into account in order to describe the electronic transport behaviour of the conductor. Quantum effects become important if the dimensions of the conductor are on the same length scale as several characteristic lengths. These length scales are:

1. λ_F: If the conductor has the same length scale as the de Broglie wavelength of current carrying electrons, then the wave like nature of the elec-

trons must be taken into account. At low temperatures, current is carried mainly by electrons having an energy close to the Fermi energy E_F or those lying near the Fermi surface. The de Broglie wavelength of these electrons is defined as the Fermi wavelength:

$$\lambda_F = \frac{2\pi}{k_F} \tag{2.2}$$

2. L_m: The elastic mean free path is the average distance an electron travels before suffering an elastic collision. Since the Fermi electrons are mainly responsible for carrying current, the mean free path can be obtained as:

$$L_m = v_F \tau_M. \tag{2.3}$$

where $v_F = \frac{\hbar k_F}{m}$ is the Fermi velocity and τ_M is the momentum relaxation time; $\frac{1}{\tau_M}$ is the rate at which the electron loses momentum. Generally, $\frac{1}{\tau_M} = \frac{\alpha}{\tau_C}$ where τ_C is the mean time interval between collisions and the factor α ranges from $0 < \alpha < 1$ depending on how effective the collisions are in scattering the electron's momentum. If the length scale of the conductor is on the order of the mean free path, the transport becomes ballistic. A more detailed discussion on scattering times and the factor α can be found in [18].

3. L_ϕ: If the conductor is on the same length scale as the distance an electron travels before the phase of its wave function is lost, the electrons can no

longer be modelled classically but rather as wave packets possessing a quantum phase. This length scale is the phase coherence length L_ϕ and can be expressed in terms of the phase relaxation time τ_ϕ:

$$L_\phi = v_F \tau_\phi. \tag{2.4}$$

As with the momentum relaxation time, the phase relaxation time τ_ϕ can be defined in terms of the time between inelastic collisions : $\frac{1}{\tau_\phi} = \frac{\alpha_\phi}{\tau_C}$ where $0 < \alpha_\phi < 1$.

The above length scales are intermediate between atomic length scales (microscopic) and bulk conductor length scales (macroscopic). Although, they vary from one material to another, the *mesoscopic transport regime* typically ranges from roughly 10nm, the de Broglie wavelength in semiconductors and mean free path in polycrystalline metal films, up to 100μm, the mean free path/phase relaxation length in high mobility semi-conductors at low temperature [19]. Today's commercial semi-conductor devices are in these meso-scales.

2.2 The Landauer Formalism

The transport of charge in mesoscopic systems cannot be modelled using Ohm's law alone because this treatment does not account for quantum mechanical effects. In this section, an expression for the conductance of a mesoscopic conductor will be derived.

Consider a conducting device connected to two electron reservoirs at chemical potentials μ_L and μ_R through two leads as shown in figure 2.1. It is assumed that the leads conduct ballistically and that electrons injecting from the leads into the contacts have zero probability of reflection. For example, a $+k$ electron originating from the left contact can be reflected in the device region, but must inject into to the right contact with no reflection if it is transmitted through the device into the right lead. In our considerations, *reflections can only occur in the device region.*

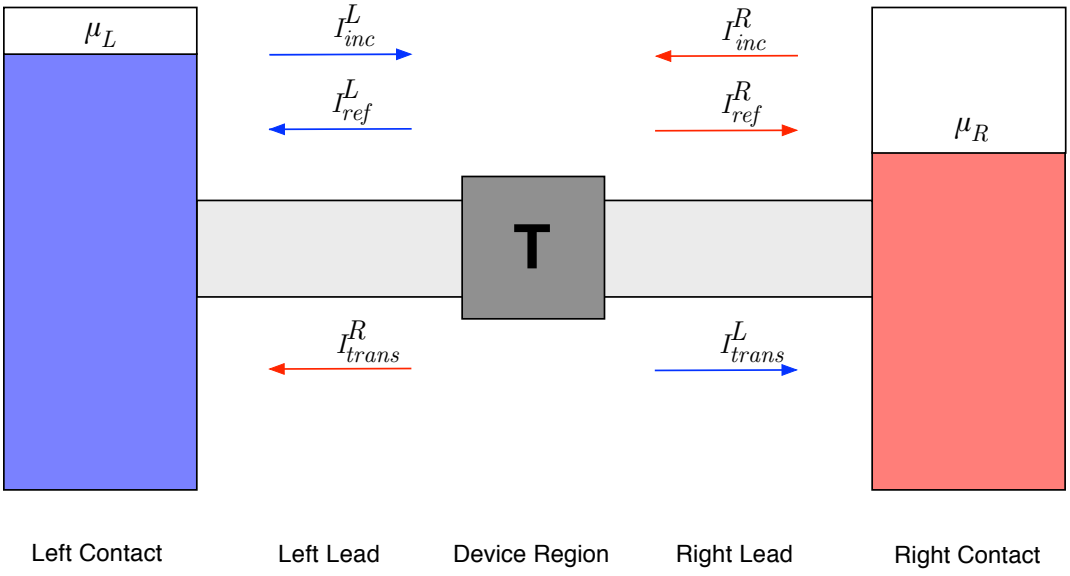


Figure 2.1: A simple two-probe system consisting of a conducting device connected to electrical contacts at μ_L and μ_R by the means of two leads. It is assumed that the leads conduct ballistically and that the contacts are reflectionless.

If the leads are narrow relative to the contacts, it is reasonable to assume that that the probability of reflection is negligible at the contact/lead interface.

A direct consequence of a model with reflectionless contacts is that the distribution of electrons originating from left contact, f_L, is independent from the distribution of electrons originating from the right contact, f_R. The current incident on the device region from the left lead can be written:

$$I^L_{inc} = env = \frac{e}{L}\sum_k \left(\frac{1}{\hbar}\frac{\partial E}{\partial k}\right) f_L(E)M(E) \tag{2.5}$$

where $M(E)$ is the number of transverse modes available at energy E. Converting the sum to an integral:

$$I^L_{inc} = \frac{2e}{h}\int f_L(E)M(E)dE \tag{2.6}$$

An expression for the reflected current can be written in terms of the probability of transmission $T(E)$:

$$I^L_{ref} = -\frac{2e}{h}\int f_L(E)M(E)(1-T(E))dE \tag{2.7}$$

Similarly, an expression for the current from the right lead that has transmitted through the device to the left lead can be written:

$$I^R_{trans} = -\frac{2e}{h}\int f_R(E)M(E)T(E)dE \tag{2.8}$$

By charge conservation, the total current in the left lead and anywhere else in

the circuit can be written:

$$\begin{aligned} I =& I^L_{inc} + I^L_{ref} + I^R_{trans} \\ =& \frac{2e}{h} \int [f_L(E) - f_R(E)] M(E) T(E) dE \end{aligned} \tag{2.9}$$

If $M(E)$ and $T(E)$ are constant in the range $\mu_L > E > \mu_R$, the electrochemical potentials of the left and right leads respectively, then the the current at zero temperature can be written:

$$I = \frac{2e}{h}[\mu_L - \mu_R]MT \tag{2.10}$$

and the conductance of the device is given by:

$$G = \frac{2e^2}{h}MT \tag{2.11}$$

Equation 2.11 is the Landauer formula. In this picture, the current through a conducting device is expressed in terms of the probability that an electron can transmit through it. Several reviews of the Landauer formalisms are available [19, 20, 21].

2.3 The Non-Equilibrium Green's Function Formalism

Another widely used approach is the Non-Equilibrium Green's Function (NEGF) formalism. Several detailed reviews of NEGF are available [19][22][23][24][25]. In this section, the key results of NEGF and its application to the transport through a two-probe system are outlined.

2.3.1 1-D Wire with Constant Potential

It is useful to first study a simple case to understand the physical interpretation of the Green's function and demonstrate the usefulness of the technique. Here, a 1-D wire with a constant potential U_0 is considered. The Hamiltonian is given by:

$$\hat{H} = -\frac{\hbar}{2m}\frac{\partial^2}{\partial x} + U_0 \tag{2.12}$$

and the corresponding Green's function $G(x)$ is defined as:

$$\left[E - U_0 + \frac{\hbar}{2m}\frac{\partial^2}{\partial x}\right] G(x) = \delta(x) \tag{2.13}$$

Comparing with the 1-D Schrödinger equation,

$$\left[E - U_0 + \frac{\hbar}{2m}\frac{\partial^2}{\partial x}\right] \psi(x) = 0 \tag{2.14}$$

we find that equations (2.13) and (2.14) are identical except for the non-homogenous term $\delta(x)$; $G(x)$ can simply be viewed as the wave function at x resulting from a unit excitation applied at the origin. It is easy to find the solutions to equation (2.13). They can be written:

$$G^r(x) = -\frac{i}{\hbar\nu}e^{+ik|x|} \qquad\qquad G^a(x) = +\frac{i}{\hbar\nu}e^{-ik|x|} \tag{2.15}$$

where $k = \sqrt{2m(E-U_0)}/\hbar$ and $\nu = \frac{\hbar k}{m}$. The first solution, the retarded Green's function, corresponds to waves moving outwards from the excitation point. The second, the advanced Green's function, corresponds to waves moving towards the excitation point. They are generally related by the relation:

$$G^r = [G^a]^\dagger \tag{2.16}$$

The Green's functions $G^r(x)$ and $G^a(x)$ contain all the information in the wave function $\psi(x)$ and are usually more convenient to work with. Once the Green's function of a system is obtained, all experimentally relevant quantities can be calculated [25].

2.3.2 Application to the Two-Probe Geometry

A two-probe system as shown in figure 2.2 is considered. The left/right lead and device region Hamiltonians are given by:

$$\hat{H}_L = \sum_k \epsilon_k \mathbf{c}_k^\dagger \mathbf{c}_k \tag{2.17}$$

$$\hat{H}_C = \sum_n \epsilon_n \mathbf{d}_n^\dagger \mathbf{d}_n \tag{2.18}$$

$$\hat{H}_R = \sum_m \epsilon_m \mathbf{b}_m^\dagger \mathbf{b}_m \tag{2.19}$$

Here, $\mathbf{c}_k$ destroys a left lead electron in the state $|k\rangle$, $\mathbf{b}_m$ destroys a right lead

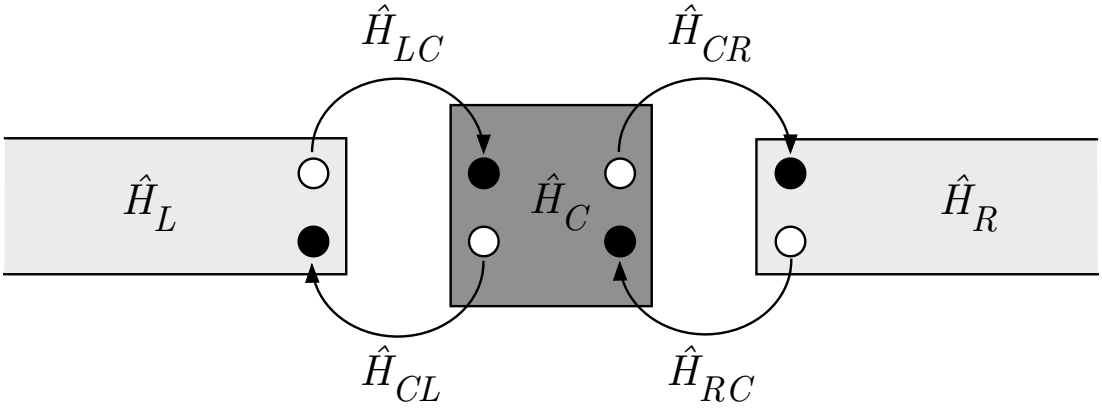


Figure 2.2: A simple two-probe system consisting of a conducting device connected to electrical contacts at μ_L and μ_R by the means of two leads. It is assumed that the leads conduct ballistically and that the contacts are reflectionless.

electron in the state $|m\rangle$, and $\mathbf{d}_n$ destroys an electron in the device region that is in the state $|n\rangle$. Tunnelling events between the leads and central region can

be described by the coupling Hamiltonians as:

$$\hat{H}_{LC} = \sum_{k,n} (t^L_{k,n})^* \mathbf{d}^\dagger_n \mathbf{c}_k \qquad \hat{H}_{CL} = t^L_{k,n} \mathbf{c}^\dagger_k \mathbf{d}_n \tag{2.20}$$

$$\hat{H}_{CR} = \sum_{n,m} t^R_{n,m} \mathbf{b}^\dagger_m \mathbf{d}_n \qquad \hat{H}_{RC} = (t^R_{n,m})^* \mathbf{d}^\dagger_n \mathbf{b}_m \tag{2.21}$$

where t^L and t^R are coupling constants and the coupling Hamiltonians are related by $\hat{H}_{LC} = \hat{H}^\dagger_{CL}$ and $\hat{H}_{CR} = \hat{H}^\dagger_{RC}$. The Hamiltonian of the whole system can be expressed as the sum of the non-interacting and coupling Hamiltonians:

$$\hat{H} = \hat{H}_R + \hat{H}_C + \hat{H}_L + \hat{H}_{LC} + \hat{H}_{CL} + \hat{H}_{CR} + \hat{H}_{RC} \tag{2.22}$$

Separating the wave function into the left (L), right (R), and central (C) regions, the Schrödinger equation takes the following form:

$$\begin{pmatrix} E - H_L & -H_{LC} & 0 \\ -H_{CL} & E - H_C & -H_{CR} \\ 0 & -H_{RC} & E - H_R \end{pmatrix} \begin{pmatrix} \psi_L \\ \psi_C \\ \psi_R \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{2.23}$$

where the $\psi_{\{L,R,C\}}$ are the wave functions in different regions of the system and E are the corresponding diagonal energy eigenvalues. The Green's function of

the system satisfies:

$$\begin{pmatrix} \epsilon - H_L & -H_{LC} & 0 \\ -H_{CL} & \epsilon - H_C & -H_{CR} \\ 0 & -H_{RC} & \epsilon - H_R \end{pmatrix} \begin{pmatrix} G_L & G_{LC} & G_{LR} \\ G_{CL} & G_C & G_{CR} \\ G_{RL} & G_{RC} & G_R \end{pmatrix} = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & & I \end{pmatrix} \tag{2.24}$$

By solving the above matrix equation for G_C, the retarded Green's function G_C^r can be obtained[22]:

$$G_C^r = [\epsilon - (H_C - \Sigma_L^r - \Sigma_R^r)]^{-1} \tag{2.25}$$

where $\epsilon = (E + i0^+)I$ adds an infinitesimal imaginary part to an energy point. The infinitesimal 0^+ is necessary to avoid singularities in case E happens to lie at an energy eigenvalue. The retarded self energies $\Sigma_{\{L,R\}}^r$ are defined as:

$$\Sigma_L^r = H_{CL} g_L^r H_{LC} \qquad\qquad \Sigma_R^r = H_{CR} g_R^r H_{RC} \tag{2.26}$$

where $g_{\{L,R\}}^r$ are the retarded surface Green's function for the left/right leads:

$$[\epsilon - H_{\{L,R\}}] g_{\{L,R\}}^r = I_{\{L,R\}} \tag{2.27}$$

Once G_C^r is obtained, the transmission function $\mathcal{T}(E)$ can be obtained using the Fischer-Lee relation [1]:

$$\mathcal{T}(E) = \mathrm{Tr}[\Gamma_L G_C^r \Gamma_R G_C^a] \tag{2.28}$$

where,

$$\Gamma_{\{L,R\}} = \Sigma_{\{L,R\}}^r - \Sigma_{\{L,R\}}^a \tag{2.29}$$

and $\Sigma_{\{L,R\}}^a = [\Sigma_{\{L,R\}}^r]^\dagger$ The current can be calculated simply by integrating the transmission function over all energies:

$$I = \frac{2e}{h} \int [f_L(E) - f_R(E)] \mathcal{T}(E) dE \tag{2.30}$$

Equation 2.30 is a more general case of equation 2.9 - the transmission function $\mathcal{T}(E)$ includes all the information contained in $M(E)$ and $T(E)$ expressed in terms of the internal states of the system.

2.4 Contour Integration

At zero bias, the central region density matrix ρ at zero temperature can be constructed in terms of the retarded Green's function.

$$\rho = \frac{2}{\pi} \mathrm{Im} \left[\int_{-\infty}^{E_F} G_C^r(E) dE \right] \tag{2.31}$$

and the number of states N can be counted from tracing the density matrix:

$$N = \mathrm{Tr}\,[\rho] \tag{2.32}$$

For closed systems, the Fermi level is obtained by numerically solving equations (2.31 and 2.32) for E_F. For open systems, E_F can be determined from the electronic structure of the leads. Referring to equation 2.25 makes it clear that G_C^r has poles in the lower complex plain near the real axis, specifically at $E = E' - i0^+$ where E' is an eigenvalue of $(H_C - \Sigma_L^r - \Sigma_R^r)$.

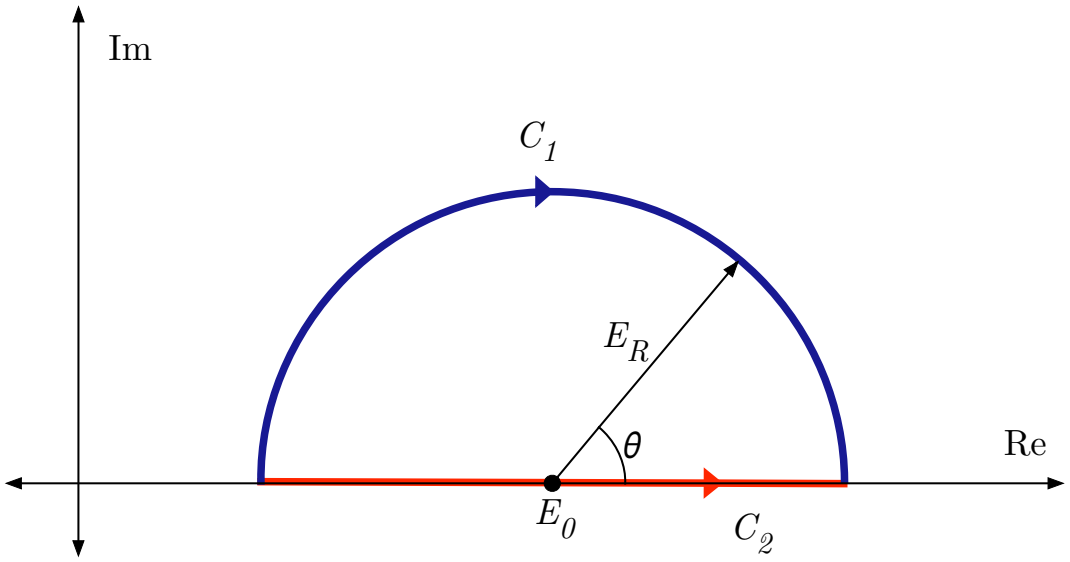


Figure 2.3: Contour integration of the Green's function: The integration is performed in the upper complex plane to avoid poles in the lower plane near the real axis. The values of the integrals over contours C_1 and C_2 are identical.

The integration must be steered clear of this region to avoid running into a pole, so the integral is performed along a semi-circle in the upper complex plain

by parametrizing $E = E_0 + E_R e^{i\theta}$ and equation 2.31 as

$$\rho = \frac{2}{\pi}\mathrm{Im}\left[\int_{\pi}^{0} G_C^r(E_0 + E_R e^{i\theta}) i E_R e^{i\theta} d\theta\right] \tag{2.33}$$

where E_0 and E_R correspond to the center and radius of the semi-circle contour respectively, as in figure 2.3.

In this chapter, we first started with the expression for the conductance of a macroscopic conductor, well known as Ohm's law, and discussed why this relation breaks down as the conductor is scaled down to small length scales. To study the behaviour of meso-scale systems, a quantum mechanical treatment is necessary. We reviewed the theoretical formalisms used in the study of such systems such as the Landauer formalism and the Non-Equilibrium Green's Function (NEGF) formalism. In the following chapter, we present the modelling tools used.

3

Device Modelling

On a technical level, the simplest nano-electronic device can be regarded as consisting of a device scattering region, such as the channel region of a transistor, contacted by several electrodes. The electrodes extend to electron reservoirs far away where bias voltages are applied and electric current is collected[1]. Figure 3.1 shows a schematic of a such a device.

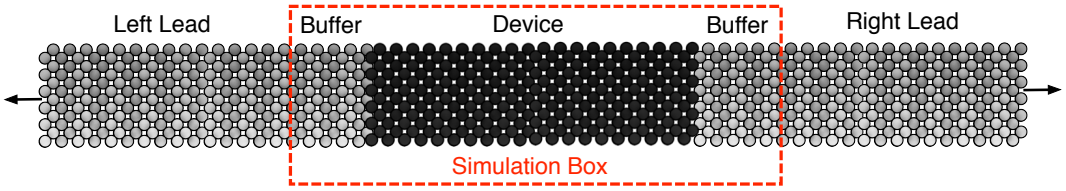


Figure 3.1: A typical atomic two-probe system. A portion of the leads is included in the device region as a 'buffer'.

The left and right leads are modelled as periodic and infinite while the scattering region, or simulation box, is finite and, in general, not periodic. A portion of the left and right leads are included in the simulation box as a buffer layer. Once the Hamiltonian of the system is specified, the transport properties of the system can, in principle, be computed using the techniques outlined in

section 2.3. However, this is a costly computational endeavour and care needs to be taken into choosing proper algorithms.

Section 3.1 outlines the Principal Layer (PL) algorithm and its application to the two-probe geometry. Section 3.2 describes the computation of the lead surface Green's functions $g_{\{L,R\}}$ and self energies $\Sigma_{\{L,R\}}$. Since the leads are semi-infinite, this needs be done iteratively. Section 3.3 discusses the techniques used in the calculation of the central region Green's function G_C and finally, the calculation of the transmission function $\mathcal{T}(E)$ is addressed in section 3.4.

3.1 Principal Layer Ordering

The prescription in [13, 26] is followed and the system is partitioned into several principal layers (PL) along the direction of transport. Interactions within a PL can be described by the matrices $h_{i,i}$ while interactions between layers i and j are described by the interaction matrices $h_{i,j}$.

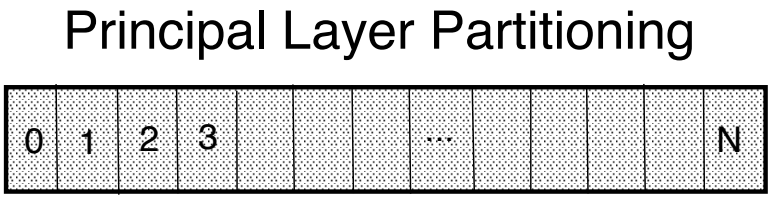


Figure 3.2: The device is partitioned into several principal layers. The atoms in each layer interact only with atoms in the same or nearest neighbouring layer.

The layers are chosen thick enough so that that the atoms in a PL interact only with atoms in the same PL or the nearest neighbouring PLs ($h_{i,j} = 0$ if $|i-j| > 1$). Figure 3.2 shows a schematic. The nearest neighbour interactions result

in the full Hamiltonian taking a block-tridiagonal form. For example, the full Hamiltonian $\hat{H}$ for a device such as the one shown in figure 3.2 can be written:

$$\begin{array}{c} \text{PL} \\ 0 \\ 1 \\ 2 \\ 3 \\ \vdots \\ \vdots \\ N \end{array} \begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & 3 & \cdots & \cdots & N \end{array} \\ \begin{pmatrix} h_{00} & h_{01} & 0 & 0 & 0 & \dots & 0 \\ h_{10} & h_{11} & h_{12} & 0 & 0 & \dots & 0 \\ 0 & h_{21} & h_{22} & h_{23} & 0 & \dots & 0 \\ 0 & 0 & h_{32} & h_{33} & h_{34} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & h_{NN-1} & h_{NN} \end{pmatrix} \end{array} \tag{3.1}$$

The principal layer ordering lends itself well to the two-probe geometry. Here, the semi-infinite leads are modelled as consisting of an infinite number of principal layers extending to the left and right. The periodicity of the structures provides a simplification - the leads' self-interaction and coupling matrices exhibit translational symmetry:

$$h_{0,0} = h_{1,1} = h_{2,2} = ... \tag{3.2}$$

$$h_{0,1} = h_{1,2} = h_{2,3} = ... \tag{3.3}$$

$$h_{1,0} = h_{2,1} = h_{3,2} = ... \tag{3.4}$$

As for the central region, it is finite and in general, not periodic. The full Hamiltonian (eqn. 2.22) of the system can now be organized in the following form:

$$\begin{pmatrix} \ddots & & & \\ h_{L10} & h_{L00} & h_{L01} & \\ & h_{L10} & h_{L00} & h_{L01} \\ & & h_{L10} & h_{L00} \end{pmatrix} \begin{pmatrix} & & & \\ & & & \\ & & & \\ h_{LC} & & & \end{pmatrix}$$

$$\begin{pmatrix} & & & h_{CL} \\ & & & \\ & & & \\ & & & \end{pmatrix} \begin{pmatrix} h_{C00} & h_{C01} & & \\ h_{C10} & h_{C11} & h_{C12} & \\ & & \ddots & \\ & & h_{C_{N-1N}} & h_{C_{NN}} \end{pmatrix} \begin{pmatrix} & & & \\ & & & \\ & & & \\ h_{CR} & & & \end{pmatrix}$$

$$\begin{pmatrix} & & & h_{RC} \\ & & & \\ & & & \\ & & & \end{pmatrix} \begin{pmatrix} h_{R00} & h_{R01} & & \\ h_{R10} & h_{R00} & h_{R01} & \\ & h_{R10} & h_{R00} & h_{R01} \\ & & & \ddots \end{pmatrix}$$

In practice, a portion of the leads are included in the central region as a buffer so that $h_{L00} = h_{C00}$ and $h_{C_{NN}} = h_{R00}$. Consequently, $h_{L01} = h_{LC}$ and $h_{CR} = h_{R01}$.

3.2 Iterative Calculation of the Self-Energies

Referring to equation (2.27), we construct the Green's function equation for the semi-infinite right lead:

$$\begin{pmatrix} h_{R00} & h_{R01} & & \\ h_{R10} & h_{R00} & h_{R01} & \\ & h_{R10} & h_{R00} & h_{R01} \\ & & & \ddots \end{pmatrix} \begin{pmatrix} g_{R00} & g_{R01} & g_{R02} & \cdots \\ g_{R10} & g_{R11} & g_{R12} & \cdots \\ g_{R20} & g_{R21} & g_{R22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} = \begin{pmatrix} I & & & \\ & I & & \\ & & I & \\ & & & \ddots \end{pmatrix} \tag{3.5}$$

A similar equation can be constructed for the left lead. Dropping the $\{L, R\}$ subscripts, both equations yield a set of equations for the surface Green's function:

$$\begin{aligned} (\epsilon - h_{00})g_{00} &= I + h_{00}g_{10}, \\ (\epsilon - h_{00})g_{10} &= h_{10}g_{00} + h_{01}g_{20}, \\ &\dots, \\ (\epsilon - h_{00})g_{N,0} &= h_{10}g_{N-1,0} + h_{01}g_{N+1,0} \end{aligned} \tag{3.6}$$

These equations can then be iterated to convergence and $\Sigma_{\{L,R\}}$ may then be calculated using equation (2.26). However, a more highly convergent scheme is presented in [27]. The chain in equation (3.6) can be transformed to express to Green's function of each layer in terms of the Green's function of the preceding layer with the introduction of the transfer matrices T and $\tilde{T}$ [13]. These are

defined:

$$T = t_0 + \tilde{t}_0 t_1 + \tilde{t}_0 \tilde{t}_1 t_2 + \tilde{t}_0 \tilde{t}_1 \tilde{t}_2 ... t_n,$$
$$\tilde{T} = \tilde{t}_0 + t_0 \tilde{t}_1 + t_0 t_1 \tilde{t}_2 + t_0 t_1 t_2 ... \tilde{t}_n, \tag{3.7}$$

where t_i and $\tilde{t}_i$ are defined recursively:

$$t_i = (I - t_{i-1}\tilde{t}_{i-1} - \tilde{t}_{i-1}t_{i-1})^{-1} t_{i-1}^2,$$
$$\tilde{t}_i = (I - t_{i-1}\tilde{t}_{i-1} - \tilde{t}_{i-1}t_{i-1})^{-1} \tilde{t}_{i-1}^2, \tag{3.8}$$

and

$$t_0 = (\epsilon - h_{00})^{-1} h_{10}$$
$$\tilde{t}_0 = (\epsilon - h_{00})^{-1} h_{01} \tag{3.9}$$

The nth term is $2^{n+1}-1$ order in h_{01} and vanishes rapidly[27]. The equations are iterated repeatedly until convergence to some tolerance ($t_n, \tilde{t}_n < \delta$). The self-energy terms in equation (2.25) are then computed directly using the transfer matrices. Convergence is usually achieved in a reasonable number of iterations (usually $\sim$10-20 for the systems we studied).

$$\Sigma_L = h_{LC} T \qquad\qquad \Sigma_R = h_{CR} \tilde{T} \tag{3.10}$$

Conceptually, the self-energies $\Sigma_{\{L,R\}}$ describe the effect of the leads on the central region. The infinite open conductor/leads system is replaced by an equivalent one consisting of a finite conductor with self-energy terms[19]. The calculation is a major computational bottleneck and highly specialized hardware and software (section 4.2) are employed to accelerate the process.

3.3 Calculating the Central Region Green's Function

The principal layer ordering described in the section 3.1 results in the matrix representation of $\hat{H}_C$ taking a convenient form. Computing the central region Green's function G_C (eqn. 2.25) is essentially reduced to a large block-tridiagonal matrix inversion problem. The number of diagonal blocks is equal to the number of principal layers along the transport direction, while the block size is related to the thickness of each layer (number of atoms/orbitals within the PL). For large systems, $\hat{H}_C$ can quickly reach unmanageable sizes and cannot be inverted directly. For a system where the scattering region is a roughly $(8.1^2) \times 43.5$ nm Si structure (140,000 atoms and ten orbitals per atom), $\hat{H}_C$ takes on a matrix of size of 1,400,000×1,400,000[1]. In this section, two block-tridiagonal inversion techniques, the Recursive Green's Function (RGF) and the Generalized Green Matrix (GGM) algorithms are presented. The goal is to express the blocks of G_C in terms of inverses and products of the blocks of $\hat{H}_C$. In this manner, the problem of inverting a large block-tridiagonal matrix is mapped to a sequence of several smaller, more manageable matrix inversion

and multiplication processes.

3.3.1 Recursive Green's Function (RGF) Technique

Consider a general $N \times N$ matrix A that has been partitioned into several blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{3.11}$$

The diagonal blocks are square matrices of not necessarily the same size. The inverse $\tilde{A}$ can also be partitioned in the same manner into blocks of the same size:

$$A^{-1} = \begin{pmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ \tilde{A}_{21} & \tilde{A}_{22} \end{pmatrix} \tag{3.12}$$

Using $A^{-1}A = I = AA^{-1}$, the blocks of the inverse can be expressed in terms of the blocks of the original matrix[14] as:

$$\begin{aligned} \tilde{A}_{11} &= (A_{11} - A_{12}A_{22}^{-1}A_{21})^{-1} \\ \tilde{A}_{12} &= -\tilde{A}_{11}A_{12}A_{22}^{-1} \\ \tilde{A}_{21} &= -A_{22}^{-1}A_{21}\tilde{A}_{11} \\ \tilde{A}_{22} &= A_{22}^{-1} + A_{22}^{-1}A_{21}A_{11}A_{12}A_{22}^{-1} \end{aligned} \tag{3.13}$$

If A is very large, computing A^{-1} quickly becomes impractical. Instead, the inverse can be computed using equation (3.13), which involves a chain of smaller matrix inversions.

These relations can be generalized to a block tridiagonal matrix B [14]:

$$B = \begin{pmatrix} B_{11} & B_{12} & 0 & \cdots & 0 & 0 \\ B_{21} & B_{22} & B_{23} & \cdots & 0 & 0 \\ 0 & B_{32} & B_{33} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & B_{n-1,n-1} & B_{n-1,n} \\ 0 & 0 & 0 & \cdots & B_{n,n-1} & B_{n,n} \end{pmatrix} \tag{3.14}$$

The corresponding tridiagonal blocks of the inverse B^{-1} are given by forward recursion:

$$\begin{gathered} \tilde{B}_{i,i+1} = -\tilde{B}_{i,i} B_{i,i+1} C_{i+1,i+1} \\ \tilde{B}_{i+1,i} = -C_{i+1,i+1} B_{i+1,i} \tilde{B}_{i,i} \\ \tilde{B}_{i+1,i+1} = C_{i+1,i+1}(I - B_{i+1,i} \tilde{B}_{i,i+1}) \\ (i = 1, 2, ..., n-1) \end{gathered} \tag{3.15}$$

where:

$$\begin{gathered} C_{i,i} = [B_{i,i} - B_{i,i+1} C_{i+1,i+1} B_{i+1,i}]^{-1} \\ C_{n,n} = B_{n,n}^{-1} \\ (i = n-1, n-2, ..., 1) \end{gathered} \tag{3.16}$$

or alternatively by backwards recursion:

$$\begin{aligned}
\tilde{B}_{i,i+1} &= -D_{i,i}\tilde{B}_{i,i+1}B_{i+1,i+1} \\
\tilde{B}_{i+1,i} &= -\tilde{B}_{i+1,i+1}B_{i+1,i}D_{i,i} \\
\tilde{B}_{i+1,i+1} &= D_{i,i}(I - B_{i,i+1}\tilde{B}_{i+1,i}) \\
(i &= n-1, n-2, ..., 1)
\end{aligned} \tag{3.17}$$

where:

$$\begin{aligned}
D_{i+1,i+1} &= [B_{i+1,i+1} - B_{i+1,i}D_{i,i}B_{i,i+1}]^{-1} \\
C_{1,1} &= B_{1,1}^{-1} \\
(i &= 1, 2, ..., n-1)
\end{aligned} \tag{3.18}$$

Additionally, the first column blocks of B^{-1} are given by:

$$\begin{aligned}
\tilde{B}_{i+1,i} &= -C_{i+1,i+1}B_{i+1,i}\tilde{B}_{i,1} \\
\tilde{B}_{1,1} &= C_{1,1} \\
(i &= 1, 2, ..., n-1)
\end{aligned} \tag{3.19}$$

and the last column blocks by:

$$\begin{gathered}\tilde{B}_{i,n} = -D_{i,i}B_{i,i+1}\tilde{B}_{i,1} \\ \tilde{B}_{n,n} = D_{n,n} \\ (i = n-1, n-2, ..., 1)\end{gathered} \tag{3.20}$$

The complexity of this approach scales linearly with the number of diagonal blocks in B and is approximately $O(N^3)$ in the size of the blocks[1].

3.3.2 Generalized Green Matrix (GGM) Technique

A new algorithm that competes well with RGF has been recently developed[29]. This method is currently in use in this work. We introduce the ratio matrices S_i and R_i:

$$\begin{gathered}S_1 = -B_{1,1}^{-1}B_{1,2} \\ S_i = -(B_{i,i} + B_{i,i-1}S_{i-1})^{-1} \\ (i = 2, 3, ..., n-1)\end{gathered} \tag{3.21}$$

[1] An efficient matrix inversion typically involves performing an LU decomposition, a back-substitution step, and solving the system $A^{-1}L = U^{-1}$ for A^{-1}. However, the process is still dominated by matrix multiplication which can be performed in slightly faster than $O(N^3)$. The interested reader is refered to *'Is Matrix Inversion an N^3 process?'* in [28].

$$
\begin{aligned}
R_{n-1} &= -B_{n,n}^{-1} B_{n,n-1} \\
R_i &= -(B_{i+1,i+1} + B_{i+1,i+2} R_{i+1})^{-1} B_{i+1,i} \\
&(i = n-2, n-3, ..., 2)
\end{aligned}
\tag{3.22}
$$

Now, the corresponding blocks of the inverse of (equation 3.14) can be written[29]:

$$
B^{-1} = \begin{pmatrix}
D_1 & S_1 D_2 & S_1 S_2 D_3 & \cdots & \left(\prod_{i=1}^{n-1} S_i\right) D_n \\
R_1 D_1 & D_2 & S_2 D_3 & \cdots & \left(\prod_{i=2}^{n-1} S_i\right) D_n \\
R_1 R_2 D_1 & R_2 D_2 & D_3 & \cdots & \left(\prod_{i=3}^{n-1} S_i\right) D_n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\left(\prod_{i=n-1}^{1} R_i\right) D_1 & \left(\prod_{i=n-1}^{2} R_i\right) D_2 & \left(\prod_{i=n-1}^{3} R_i\right) D_3 & \cdots & D_n
\end{pmatrix}
\tag{3.23}
$$

Here, the D_i are the diagonal blocks. They can be computed using the ratio matrices and the blocks of B:

$$
\begin{aligned}
D_1 &= (B_{1,1} + B_{1,2} R_1)^{-1} \\
D_n &= (B_{n,n} + B_{n,n-1} S_{n-1})^{-1} \\
D_i &= (B_{i,i-1} S_{i-1} + B_{i,i} + B_{i,i+1} R_i)^{-1} \\
&(i = 2, 3, ..., n-1)
\end{aligned}
\tag{3.24}
$$

On a single core, both techniques, GGM and RGF, have a similar performance. GGM has a lower complexity and is slightly faster in computing the first and last block columns while RGF performs better at computing the tridiagonal blocks. However, the real advantage is that GGM can be more parallelized to yield a speed-up factor of 1.25 to 3 over RGF. A detailed analysis of the complexity of both techniques as well as a description on parallelizing GGM can be found in [29].

3.4 Fast Calculation of the Transmission Function

Finally, $\mathcal{T}(E)$ can be computed as a trace using the Fischer-Lee relation (equation 2.28). However it is worth mentioning that because of the form of the product, not all the elements of G_C are required. Explicitly, the right hand side of (equation 2.28) is:

$$\begin{pmatrix} \Gamma_L & & \\ & & \\ & & \end{pmatrix} \begin{pmatrix} G_{C_{1,1}} & \cdots & G_{C_{1,n}} \\ \vdots & \ddots & \vdots \\ G_{C_{n,1}} & \cdots & G_{C_{n,n}} \end{pmatrix} \begin{pmatrix} & & \\ & & \\ & & \Gamma_R \end{pmatrix} \begin{pmatrix} G_{C_{1,1}} & \cdots & G_{C_{1,n}} \\ \vdots & \ddots & \vdots \\ G_{C_{n,1}} & \cdots & G_{C_{n,n}} \end{pmatrix}^{\dagger} \tag{3.25}$$

The surviving non-zero diagonal element in the result depends on the $G_{C_{1,n}}$ block only. The trace can be computed quickly as:

$$\mathcal{T}(E) = \mathbf{vec}(\left[\mathbf{\Gamma_L G_{G_{1,n}}}\right]^{\mathbf{T}}) \cdot \mathbf{vec}(\mathbf{\Gamma_R G^{\dagger}_{G_{1,n}}}) \tag{3.26}$$

For transmission calculations, blocks other than the top-right block of G_C need not be calculated, saving considerable computation time. More details are presented in section 4.4.

In this chapter, we presented some of the modelling tools used in the study of electronic transport in mesoscopic systems. We started by partitioning the mesoscopic conductor into several principal layers along the transport direction, each of which interacts only with itself or the nearest neighbouring layers. The Principal Layer (PL) ordering results in the device Hamiltonian taking a convenient block tridiagonal form. We then presented two block tridiagonal matrix inversion algorithms, the Recursive Green's Function (RGF) algorithm, and the recently developed Generalized Green's Function (GGM) algorithm, and proceeded to show how these tools can be used to efficiently calculate the self-energies, Green's function and subsequently the transmission function of a mesoscopic device.

4

Computation

A major bottleneck in computational physics is the performance of linear algebra routines. In section 3, it was shown that computing transfer matrices and subsequently the surface Green's function of the leads relies on heavily on matrix-matrix multiplication and matrix inversion. Additionally, in section 2.3, it was shown that finding the central Green's function of a two-probe system is essentially a large block-tridiagonal matrix inversion problem.

In addition to developing faster and more efficient numerical algorithms, it helps to use proper hardware. Employing massively parallel clusters of CPUs for linear algebra is an excessive approach which is of limited effectiveness at large problem sizes. Section 4.3 contains a comparison of the performance of such CPU clusters against a single GPU device.

In this section, we review the history of GPUs and present a brief overview of their unique architecture. We then introduce the GPU-Matlab Interface, designed to connect Matlab to the GPU. Finally, we show how GPUs fit into the context of quantum transport calculations and demonstrate how they can

be used to bypass severe computational bottlenecks.

4.1 Heterogeneous Computing and GPUs

GPUs have existed since the early 1980s and were traditionally employed as specialized accelerators for video games. Their potential for scientific computing was first realized by researchers in the field of computer science, however, they were limited in that they were not easily programmable compared to a general purpose processor. The high level of expertise required to use them was the major limitation that prevented them from achieving mainstream popularity, despite their impressive floating point performance.

Today, GPUs are becoming increasingly highly programmable. With the introduction of general purpose programming tools for GPU such as CUDA and OpenCL, GPU computing became much more popular and easily accessible to researchers other than highly specialized computer scientists.

OpenCL or "Open Computing Language" is an Application Programming Interface (API) initially developed by Apple and subsequently in collaboration with teams from companies such as Intel, IBM and Nvidia. The specification[30] was made public in December 2008 and the first release was made available as part of Mac OS X Snow Leopard.

OpenCL kernels are written as strings and must be incorporated inside another host code, such as C or C++, for execution. They are not device specific and are capable of running on various devices such as CPU, GPU, portable

electronic device, or any other available OpenCL capable device without any special modifications. This feature makes it very easy to port an OpenCL code from one device to another without rewriting the entire application in a different programming language. It also makes it possible to take advantage of every computational resource on the system using a single programming interface. Several excellent lectures with example code are freely available as podcasts from MacResearch [31].

CUDA (Compute Unified Device Architecture) is a general purpose GPU programming interface from Nvidia. Unlike OpenCL, CUDA is device specific and executes only on CUDA capable Nvidia GPUs. A significant effort has been made by Nvidia to increase the accessibility of their devices for scientific computation. A complete GPU-accelerated Basic Linear Algebra Subroutines library (cuBLAS) with support for all 152 standard BLAS routines and all data precision types was developed by Nvidia and researchers at The University of California at Berkeley [15] and made freely available as part of the CUDA interface. OpenCL libraries are open source and therefore take a longer time to mature. Currently, the proprietary CUDA based Lapack implementation used in this work - CULA [32] - is more extensive than its OpenCL counter part - MAGMA [33]. Some good starting points to pick up CUDA and GPU programming are [34] and [35].

A glance at GPU architecture makes it apparent that they are designed for processing highly parallel problems - something once possible only with large

CPU clusters. The original Fermi design [36] features 512 stream processors at 1401 MHz, known as CUDA cores, and up to 6 gigabytes of GDDR5 RAM. The CUDA cores are organized into 16 streaming multiprocessors (SM) of 32 CUDA cores each, shown as green vertical blocks in figure 4.1. Thread management in GPUs is done using dedicated hardware at a local level for each SM by local schedulers, shown as an orange strips, and at a global level by the dedicated GigaThread global scheduler, which is capable of managing thousands of threads in parallel.

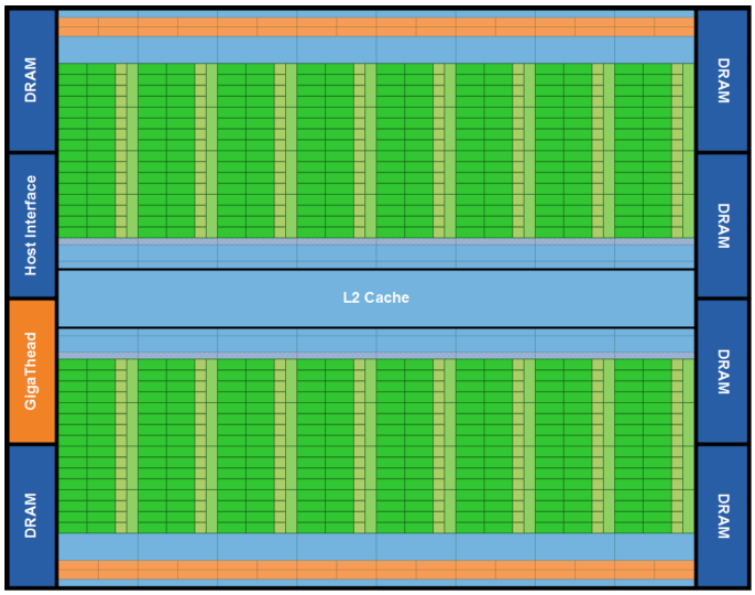


Figure 4.1: An overview of the massively parallel Fermi Architecture. The green strips represent CUDA cores. They are organized into 16 streaming processors of 32 cores each. Each streaming processor also has a local scheduler represented by an orange strip. Taken from [36].

The unique massively parallel architecture of GPUs makes them ideal for performing computations with a high arithmetic intensity, but limits their ap-

plication to sequential problems. GPUs are not used as or intended to be a stand-alone replacement to traditional processors. In general, they are employed as part of a heterogenous computing scheme where the sequential parts of the application are run on traditional processors and the highly parallelizable parts are sent to the GPU for acceleration. Heterogenous computing is cheaper, more efficient, less power consuming and presents a serious challenge to the traditional CPU-only based cluster approach. In section 4.4, we outline how NEGF calculations are performed using heterogenous computation.

4.2 The GPU-Matlab Interface

GPUs are ideal for performing computationally intensive tasks such as linear algebra routines in less time than required on CPU clusters. In order to tap into this computational resource, a Matlab-GPU Interface (GMI) has been developed in this work. The interface is multi-threaded and can also be used to manage several GPUs in parallel. The usage is very similar to the native Matlab syntax and no experience with GPU programming is required by the end-user.

GMI allows users to use multiple GPUs in parallel by starting an individual thread on each device. A thread is the smallest unit of instructions scheduled by the operating system. In this case, each thread contains a set of instructions to schedule an operation on some device, for example, scheduling a matrix multiplication operation. Thread management of the *scheduling threads* was done in-software using the freely available POSIX Threads library [1].

[1]The scheduled operation itself e.g. matrix multiplication could start new threads on the

GMI is written in C and is powered by CULA [32], a set of GPU-accelerated linear algebra routines for Nvidia cards. Recently, CULA released a feature that allows users to use the library directly from Matlab by simply changing Matlab's linking settings to point to the CULA libraries instead of a regular LAPACK library[32]. However, this is limited since it allows for only one GPU to be used at a time.

A flow diagram for GMI is presented in figure 4.2. After the user specifies the input arguments, GMI locates and counts the number of GPU devices on board, starts a number of threads equal to the number of devices, and binds each thread to a different GPU. Each thread then makes a call to the appropriate CULA routines to perform a linear algebra operation. When all the threads finish execution, the results are returned to the Matlab workspace in the host memory.

4.3 Benchmarks

In this section, we test the capabilities of the GPU-Matlab interface (GMI) developed in this work. We show that for linear algebra heavy applications, a heterogeneous computing scheme with GPUs is more suitable than any CPU-only approach. A single GPU outperforms single processors, multiple processors, and massively parallel processor grids. We also compare GMI's linear algebra performance to Jacket[37] and the Matlab Parallel Computing Toolbox (PCT)

device it executes on. These threads however are managed automatically, either in dedicated hardware if the executing device is a GPU, or in software by Matlab if the executing device is a CPU.

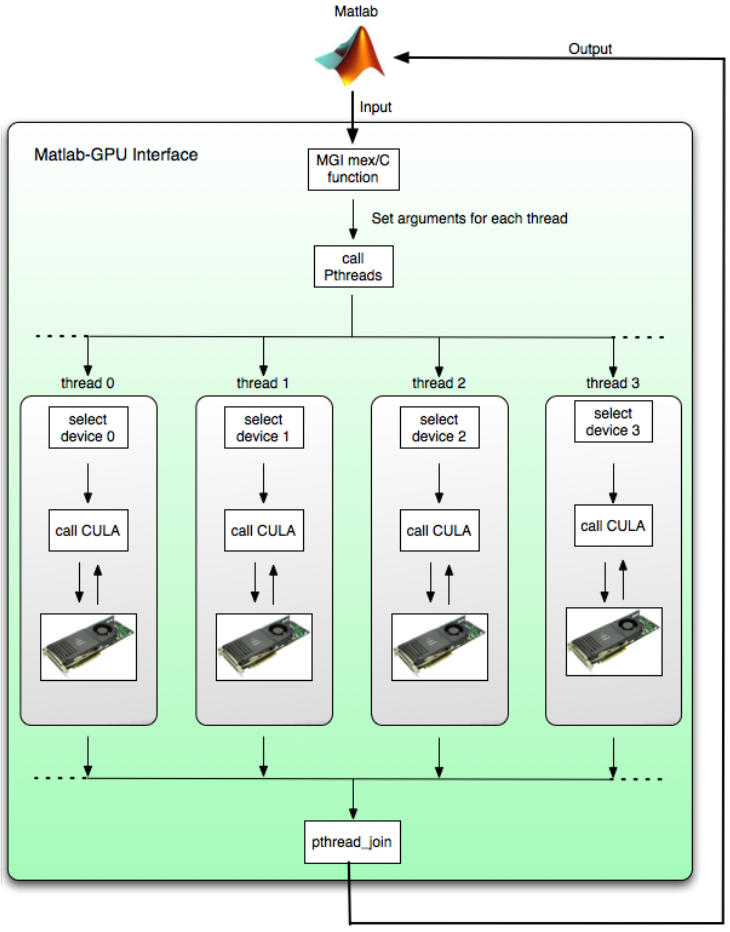


Figure 4.2: GMI flow diagram: Multiple threads with different input data but performing the same operation are executed in parallel on several GPU devices. (Single program, Multiple Data)

and demonstrate it competes very well with these commercial software. We also demonstrate that the performance scales fairly efficiently, but not perfectly, with each additional GPU added.

4.3.1 Comparison with traditional CPU approach (single-node)

We compared the linear algebra performance of a single GPU to a two single processors: (i) a mid-range general purpose processor: AMD Phenom II X6 1075T, 6 real cores and ii) a high-end general purpose processor: Intel Xeon X5650 CPU, 6 real cores. 12 virtual cores with hyper threading enabled.

The benchmark operations are matrix multiplication and matrix inversion and the test arrays were chosen to be of the same data type and shape as the large matrices involved transport calculations: double precision, complex, and square. However, the results hold for any shape, since the performance only depends on the total number of floating point operations performed, which in turn is related only to the number of elements in the array.

Matrix inversion in GMI is done with an LU decomposition with partial pivoting. The CULA routines `culaGETRF` (performs LU decomposition) and `culaGETRI` (computes matrix inverse using result of LU decomposition) are used. Suppose A can be decomposed into a product of a lower triangular matrix L and an upper triangular matrix U,

$$A = LU \tag{4.1}$$

The upper triangular matrix U can be easily inverted by back substitution and A^{-1} is computed by solving the equation

$$A^{-1}L = U^{-1} \tag{4.2}$$

for A^{-1}. The CPU operation was done in the native Matlab environment with the `inv` function. `inv` is also essentially composed of the Lapack equivalents: `GETRF` and `GETRI`. Figure 4.3 displays the matrix inversion performance on all tested platforms at various matrix sizes. The speed-up of the GPU relative to each CPU platform is also shown.

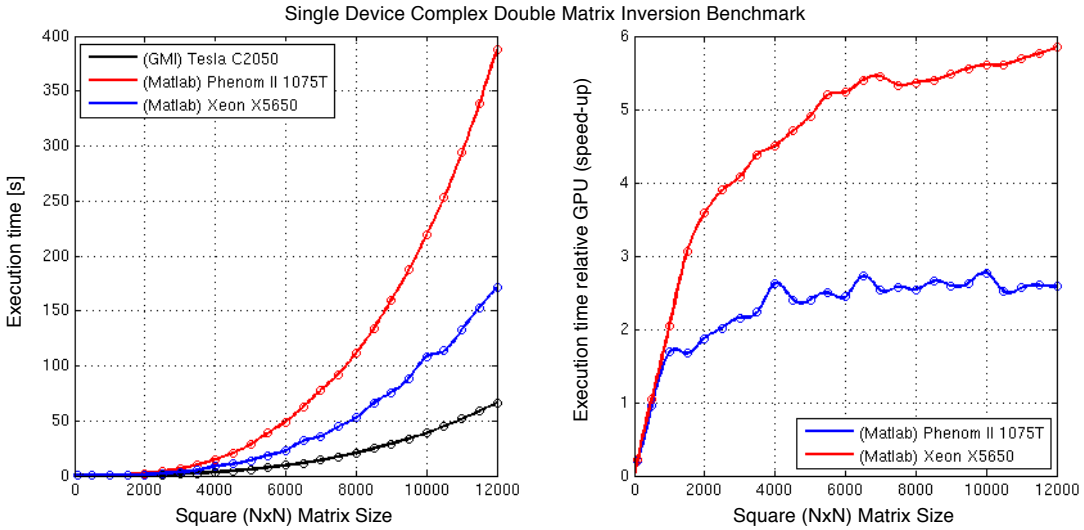


Figure 4.3: A benchmark for complex double precision matrix inversion. The horizontal axis shows the square (NxN) matrix size. The figure on the left shows the true execution time for several platforms and the right shows shows the execution time relative to the GPU.

The speed-up factor ranges from ∼2.5 to ∼6 and is not constant across all matrix sizes. The reason is that even though GPUs perform floating point operations very quickly, the overhead associated with using GPUs is greater than that of using a traditional processor. Each time the GPU is used, an

individual thread needs to be initialized, the thread is then bound to a GPU device, and the data is sent to and from to the device for processing over the machine's PCI bus port, which typically has a bandwidth of 133-533 Mbyte/s. In fact, if the total number of floating point operations (FLOPS) is not high enough, the calculation may be performed on the CPU several times before the data even reaches the GPU. This is apparent in figure 4.3: The speed-up factor relative to GPU is less than 1 for matrices smaller than 500×500. *Generally, no significant speed-up is observed unless the size of the input array exceeds at least* $\sim 10^6$ *elements.* The speed-up then scales with increasing input array size and plateaus when the computation time is very large relative to the overhead time.

Matrix multiplication in GMI is done using the general $O(N^3)$ matrix multiplication algorithm. The CULA routine used is `culaGEMM` (general matrix-multiplication). In this section, we compare the performance of the double, complex GMI matrix multiplication routine on a single GPU to a CPU-based Matlab implementation with the `mtimes` function. `mtimes` is essentially composed of the Lapack routine `GEMM` (general matrix-multiplication). The test was performed on matrices ranging from 100×100 to $10,000 \times 10,000$ with the same hardware as the previous section. Figure 4.4 displays the results on all tested platforms at various matrix sizes. The speed-up of the GPU relative to each CPU platform is also shown.

The speed-up factor ranges from 2 to 4.5 at the largest square matrix size

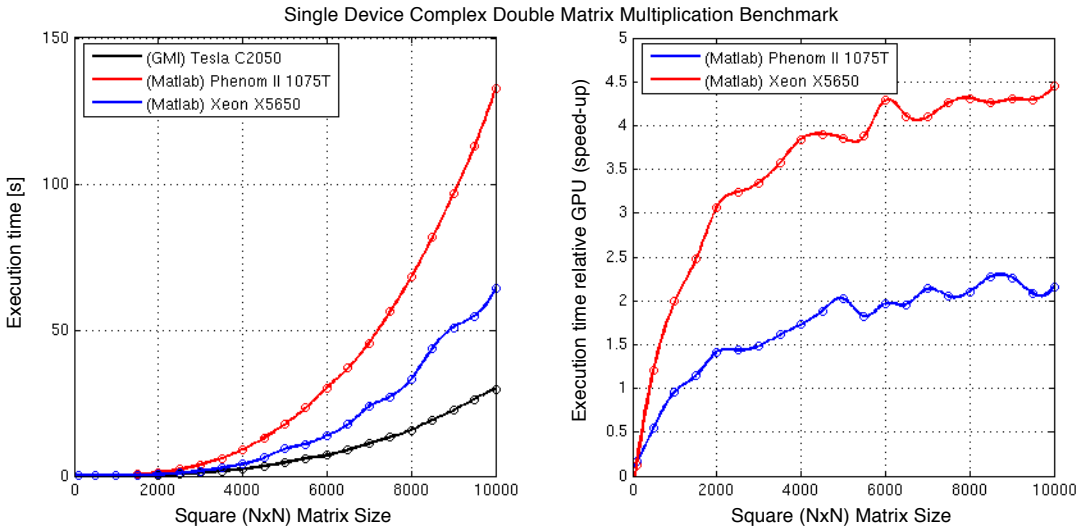


Figure 4.4: A benchmark for complex double precision matrix multiplication. The horizontal axis shows the square (NxN) matrix size. The figure on the left shows the true execution time for several platforms and the right shows shows the execution time relative to the GPU.

and for the reasons previously discussed is not constant for all matrix sizes. The speed-up is low at first and scales positively with increased square matrix size and eventually plateaus.

4.3.2 Comparison with traditional CPU approach (massively-parallel)

The linear algebra performance of a single Nvidia Tesla C2050 card was benchmarked and compared to a (i) node with two Quad-Core Xeon E5620 processors and (ii) a tuned massively parallel Scalapack implementation using the Scalapack-Matlab Interface(SMI)[38]. Scalapack is a scalable linear algebra package for use with shared memory super computing clusters. The Scalapack 'grid size' is an abstraction of the actual arrangement of individual processor cores on the supercomputer. In this case, a 5×5 grid (i.e. 25 processor cores)

was used on CLUMEQ's Guillimin supercomputer cluster equipped with Quad-Core Xeon E5620 processors. A single Quad-Core Phenom II 910 processor was also included in the benchmark to establish a baseline.

The benchmark operations were chosen to be double precision complex matrix inversions of square matrices sizes ranging from 6000×6000 to 10000×10000. Figure 4.5 shows a comparison of the execution time (and execution time relative to single GPU) on all platforms tested.

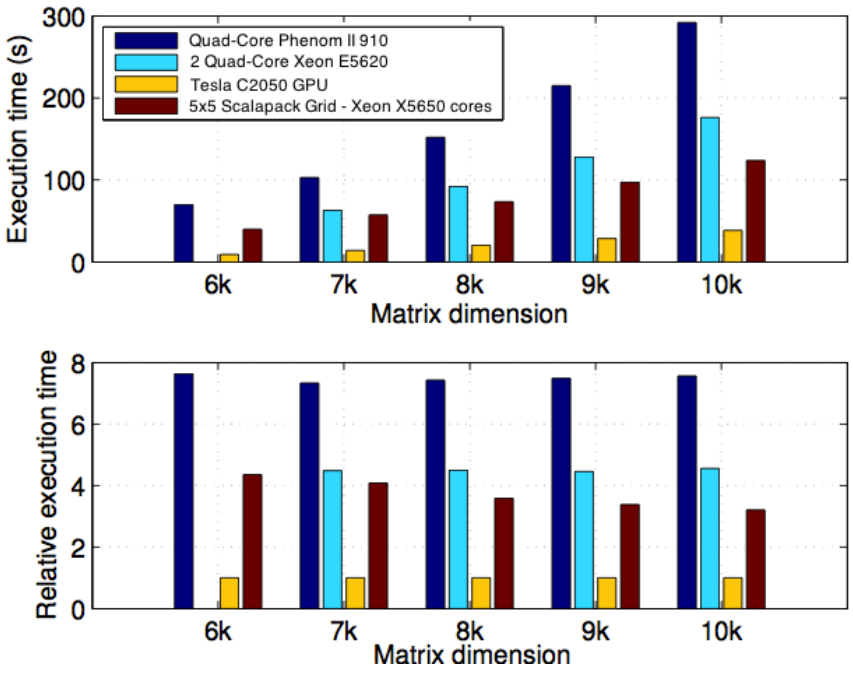


Figure 4.5: A benchmark for complex double precision matrix inversion (lower is better). The horizontal axis shows the square $(N \times N)$ matrix size . The top figure shows the true execution time for several platforms and he bottom figure shows the execution time relative to the GPU. The Nvidia C2050 GPU has the lowest execution time for all matrix sizes.

Although the massively parallel CPU implementation is a significant improvement over the single/multiple processor CPU approach, the GPU won in

each case and the speed-up ranged from a factor of 3, relative to the 25 core Scalapack grid, to a factor of 8, relative to an implementation on a typical modern quad core processor.

4.3.3 *Multiple GPU devices working in parallel*

The performance scales well with additional GPUs, but there is a non-trivial overhead associated with using additional devices. For this reason, the input data set should be reasonably large. Figure 4.6 shows the execution time for performing a computationally intense problem - 24 double precision complex matrix inversions - on 1, 2, 3, and 4 Nvidia C2050 GPUs in parallel.

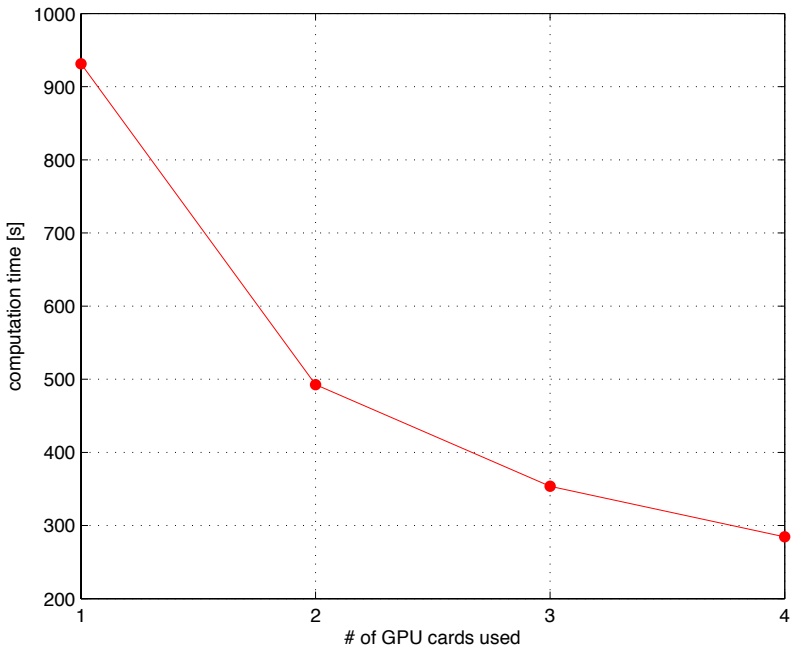


Figure 4.6: Computation time for performing 24 double precision complex 10,000×10,000 matrix inversions. The horizontal axis is the number of GPU used to perform the operation.

Since the GPUs are being controlled by a central CPU, adding an additional device incurs some overhead as a result of the devices competing for processing power. This makes scaling imperfect; according to figure 4.6, the speed-up factor relative to a single device is 1.8× for two devices (∼90% efficiency), 2.5× for three devices (∼85% efficiency) and 3.2× for four devices (∼80% efficiency). The overhead incurred by each additional device is a systematic ∼5% reduction from perfect efficiency. For this reason, it is recommended that the number of GPUs per node be limited to four cards. Additional nodes can be added to the cluster for every four extra GPUs to maintain a reasonable efficiency.

4.3.4 Comparison with Commercial GPU packages

Several proprietary GPU software for Matlab are available. In this section we compare two commercial packages, Jacket and the Matlab Parallel Computing toolbox (PCT) to the GPU Matlab Interface (GMI) developed in this work. Jacket, like GMI, relies on the CULA (GPU accelerated Lapack) package to perform linear algebra routines on CUDA capable GPUs. PCT, on the other hand, utilizes Magma, an open source GPU accelerated linear algebra library.

The Matlab Parallel Computing Toolbox (PCT) was introduced by Mathworks in 2008 and allowed users to use multicore, processors and computer clusters to solve computationally intensive problems. It also allowed users to run as many 'workers', or separate Matlab sessions, as licensing allows (up to 12) on one multicore desktop. Other key features included are parallel for-loops (`parfor`) and distributed arrays for large data set handling. As of 2010,

Mathworks introduced GPU support in the toolbox.

Jacket from Accelereyes is a third party GPU interface for Matlab. The first release appeared in 2007, 3 years before Mathwork's PCT started including GPU support. Recently, NASA utilized genetic algorithms as well as GPUs with Jacket for rover image compression in the Curiosity Mars rover mission. "With Jacket and GPUs, the researchers were able to achieve 5× speedups on the larger data sizes. [39]"

We compared the linear algebra performance of GMI to Jacket and PCT in the cases of matrix multiplication and matrix inversion. For both cases, the same operation was done using each interface over a range of square matrix sizes.

Figure 4.7 shows the general, double precision, complex square matrix multiplication computation time for both operations. Jacket closely matches GMI's matrix multiplication performance, but gradually trails behind at larger matrix sizes at matrix inversion. The version of PCT tested does not support matrix inversion [1] and trails behind both at for all sizes at matrix multiplication.

The main result that is apparent from figure 4.7 is that GMI can handle significantly larger input data sets than both other packages while maintaining an efficient performance. On our workstation equipped with Tesla C2050 cards, matrix multiplication with PCT returned out of memory errors at a maximum square (double complex) matrix size of 5700×5700, Jacket could handle matrices

[1] As of 2012, Mathworks added GPU matrix inverse support to PCT but this version was not tested in this work.

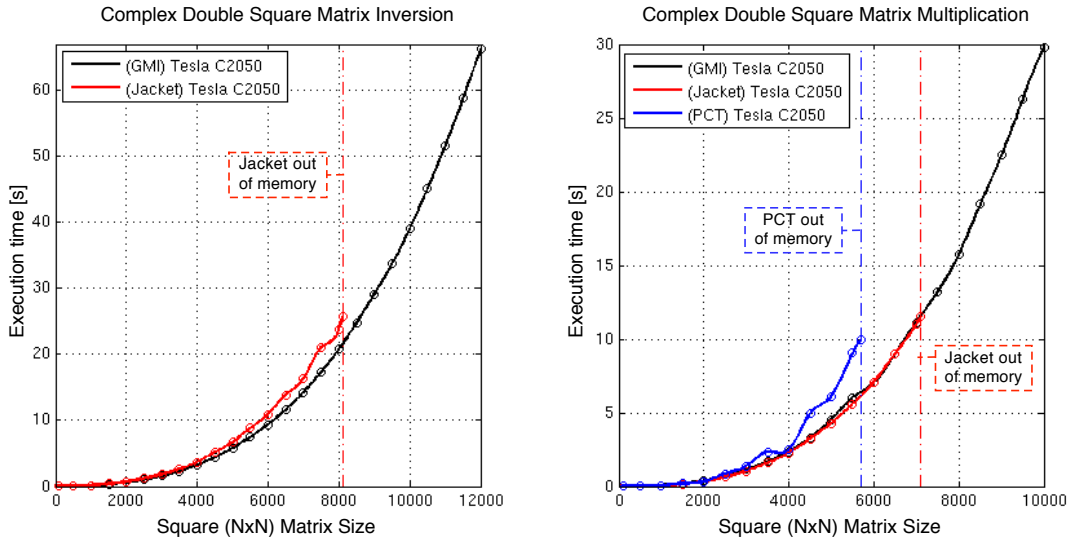


Figure 4.7: A benchmark for complex double precision matrix inversion and matrix multiplication. The horizontal axis shows the square (NxN) matrix size. The figure on the left shows the true execution time for several platforms and the right shows shows the execution time relative to the GPU. GMI matches the performance of Jacket and PCT while being able to handle significantly larger matrix sizes.

of up to 7100×7100 while GMI runs out of memory at a significantly larger matrix size of $10,300 \times 10,300$. For matrix inversion, Jacket runs out of memory at $8,100 \times 8,100$ while GMI can handle matrices of up to $12,300 \times 12,300$.

The main reason GMI can handle larger matrices is because of its unconventional 'black box' memory management. Jacket and PCT manage memory very similarly. Both create separate workspaces for the host and GPU device and allow users to send data to and from these workspaces. This is done by introducing 'gpu objects'; `gdouble` objects in Jacket and `gpuArray` objects in PCT. These objects are stored on the GPU workspace memory. Functions with gpu objects as input parameters automatically execute on the GPU. The fol-

lowing code snippets demonstrate the usage of `gdouble` and `gpuArray` objects as compared to the native Matlab environment.

-example 1: Matlab native environment:

```
>>A=rand(N)       %Create a test array in host memory.
>>C=inv(A);       %Inverts A on the CPU. Host memory now contains
                  %A and C.
```

- example 2: Jacket `gdouble` objects:

```
>>A=rand(N);      %Create a test array in host memory.
>>A_=gdouble(A);  %copy A to device memory.
>>C_=inv(A_);     %Inverts A_ on the GPU. Device memory now contains
                  %A_ and C_.
>>C=double(A_);   %Copy C_ to host memory.
```

-example 3: PCT `gpuArray` objects:

```
>>A=rand(N);      %Create a test array in host memory.
>>A_=gpuArray(A); %Copy A to device memory.
>>C_=inv(A_);     %Inverts A_ on the GPU. Device memory now contains
                  %A_ and C_.
>>C=gather(C_);   %Copy C_ to host memory.
```

In contrast, GMI does away with the concept of a 'gpu object' and separate workspaces for the host and device. Instead, GMI utilizes each GPU device as a computational black box; data cannot be stored permanently and cannot be

moved back and forth from the device unless a GPU computation is required. This has two main advantages: (i)The GPU input data set can be overwritten thus saving considerable, valuable GPU device memory. (ii)Syntax is simpler and less verbose making research code easier to port to the GPU.

-example 4: GMI:

```
>>A=rand(N);          %Create a test array in host memory.
>>C=gpuBasicDinv(A); %Copies A to the device memory. The data on
                      %the GPU is then overwritten by the result of
                      %the calculation. The results are then copied
                      %back to host memory. At any given time, only
                      %one array needs to be stored in device memory.
```

From the user's perspective, replacing `inv` with `gpuBasicDinv` simply makes the calculation run faster. The memory management and data transfer to and from the GPU device is all done under the hood.

In conclusion, it was demonstrated that the Matlab GPU Interface (GMI) competes effectively with recent, state of the art commercial packages. GMI matches or exceeds commercial software in performance and can handle significantly larger data sets. Porting CPU code to GPU code with GMI is also more straightforward because of the transparent syntax.

4.4 Pertinence to Transport

In the previous chapters, the theory and modelling tools of quantum transport were introduced. We also argued in favour of a heterogeneous GPU/CPU computing scheme as opposed to a traditional massively parallel CPU-only approach. We introduced the GPU-Matlab interface (GMI) and demonstrated some performance benchmarks. In this section, we describe concretely how heterogenous computing fits into the context of NEGF quantum transport calculations.

Figure 4.8 presents a flow diagram of the GPU-accelerated NEGF-TB code developed in this work. The code has two main parts. (i) Calculating the transfer matrices T and $\tilde{T}$ and calculating the Self-Energies Σ_L and Σ_R and (ii) Calculating the the central region Green's function G_C. In this section, we discuss our implementation on a work station equipped with four Tesla C2050 GPU devices.

The procedure in sections 3.2 and 3.3 is followed. The left column describes the operation, the next four columns represent one thread each and the final column specifies whether the operation was done on the GPU or by using sparse routines on the CPU. Sparse routines are appropriate for arrays that consist of mostly zero elements because of the special format used. In the Yale Sparse Matrix Format, an arbitrary sparse $m \times n$ matrix A is stored as three 1-D arrays. Suppose A has N_{nz} non-zero elements. The first array, `A`, is of length N_{nz} and contains all the non zero entries of A. The second array IA is of length

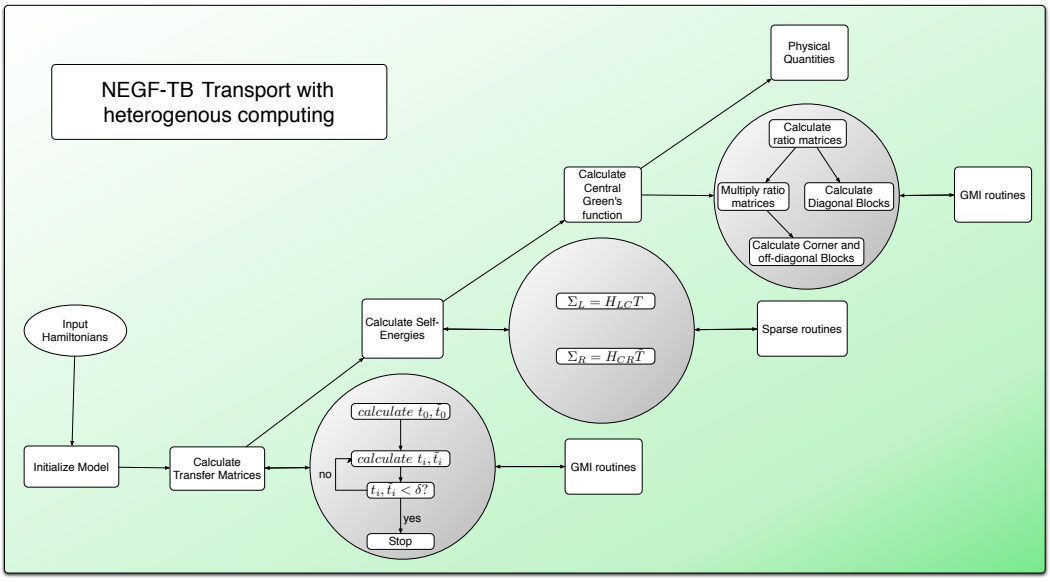


Figure 4.8: GPU Accelerated NEGF-TB code flow diagram: There are two major bottlenecks in calculating a system's Non-Equilibrium Green's Function: (i) The convergent self-energy calculation and (ii) inverting the large block-tridiagonal central region Hamiltonian with self-energy terms. Both are highly linear algebra intensive and benefit greatly from a heterogenous computing scheme. Sparse operations are done on the CPU to avoid GPU overhead, while dense, intensive calculations are sent to several GPUs working in parallel.

$(m+1)$. The first m elements are the row-indices of the first non zero element on each row of A and the last element is the value of NNZ. The third array, JA contains the column-indices of the elements of A and consists of NNZ elements. For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 3 \\ 0 & -4 & 0 \end{pmatrix} \tag{4.3}$$

can be expressed in sparse form as:

```
A= [1 2 3 -4]
IA=[1   3 2 4]
JA=[0 1 2 1]
```

For a general matrix of datatype '`type`', the sparse format requires

$$N_{nz}\,(\texttt{sizeof(type)} + \texttt{ sizeof(int)}) + (m+1) \times \texttt{sizeof(int)} \tag{4.4}$$

bytes of storage, while the usual format requies $mn \times$ `sizeof(type)`. So, the sparse format saves memory if and only if:

$$N_{nz} < \frac{mn \times \texttt{sizeof(type)} - (m+1) \times \texttt{sizeof(int)}}{\texttt{sizeof(type)} + \texttt{sizeof(int)}} \tag{4.5}$$

Note that in the special case of the integer matrix above, the sparse format is not appropriate since it contains 12 entries but the usual format contains 9 entries.

The sparse format is advantageous for matrix multiplication involving matrices with a low density of non-zero entries because (i) it reduces the amount of memory required to store the matrix - zero elements are not stored and (ii) it does not waste floating point operations on non-zero entries. Instead, a labelling scheme is used to perform the operations (the additional overhead is worthwhile if the inputs are sparse enough). However, for the same reasons, the sparse representation is severely disadvantageous for dense matrices. Additionally, it is disadvantageous for matrix inversion because the results are typically

not sparse even if the input is sparse.

The TB Hamiltonians used are typically very sparse. For example, the TB Hamiltonian of the Silicon Nanowire in section 5.2 is composed of ∼94 % zero elements. Figure 5.2 shows the structure. In our heterogenous implementation, any matrix-multiplication with a TB Hamiltonian as one of the inputs is done using the CPU with sparse routines. All other operations involve dense matrices and a large number of floating point operations - these are done on the GPU. This scheme is shown explicitly in figure 4.9 but excluded in figure 4.10 for compactness and clarity of presentation.

The first step of calculating the self-energies consists of iterating equations 3.7 to convergence. Convergence is reached when the $||t_i||, ||\tilde{t}_i|| < \delta$. The tolerance δ is typically chosen as 10^{-10} but can be increased to arbitrary precision (up to machine accuracy). The matrix multiplication operations in equations 3.7 are highly parallelizeable and this is exploited. Once the calculation converges, typically 5-10 iterations, the Self-Energies are calculated quickly using sparse routines. For the Central Region Green's function, the ratio matrices S_i are R_i are computed sequentially. Once these are determined, any diagonal block can be found quickly using equation 3.25. For transport, the top-right corner block of the Central Green's function, $G_{1,n}$ is of interest. This specific case is addressed in the pseudocode presented in figure 4.10 and the compution of any other non-diagonal block reduces to a chain of matrix multiplications (see equation 3.23) and can be easily parallalized in a similar manner. The optimal

operation	**thread 1**	**thread 2**	**thread 3**	**thread 4**	**device**
find $t_0, \tilde{t}_0$	$\text{temp1} = (\epsilon - h_{00})^{-1}$ $t_0 = \text{temp1} * h_{01}^{\dagger}$ $\tilde{t}_0 = \text{temp1} * h_{01}$ $T = C_0 = t_0$, $\tilde{T} = \tilde{C}_0 = \tilde{t}_0$				GPU CPU CPU CPU
Repeat till convergence $t_i, \tilde{t}_i < \delta$ $i = 1, 2, ...$	$\text{temp1} = t_{i-1} * \tilde{t}_{i-1}$ $\text{temp1} = (I - \text{temp1} - \text{temp2})^{-1}$ $t_i = \text{temp1} * \text{temp3}$ $T = T + C_i * t_i$ $C_{i+1} = C_i * t_i$	$\text{temp2} = \tilde{t}_{i-1} * t_{i-1}$ $t_i = \text{temp1} * \text{temp4}$ $\tilde{T} = \tilde{T} + \tilde{C}_i * \tilde{t}_i$ $\tilde{C}_{i+1} = \tilde{C}_{i+1} * \tilde{t}_i$	$\text{temp3} = t_{i-1} * t_{i-1}$	$\text{temp4} = \tilde{t}_{i-1} * \tilde{t}_{i-1}$	GPU GPU GPU GPU GPU
Calculate Σ_L and Σ_R from T and $\tilde{T}$	$\Sigma_L = -h_{01} T$ $\Sigma_R = -h_{01}^{\dagger} \tilde{T}$				CPU CPU

Figure 4.9: Self Consistent $\Sigma_{L,R}$ calculation pseudo-code parallelized (where possible) over four devices.

procedure for finding the Central Green's function depends on what blocks are of interest. The number of ratio matrices that need to be multiplied increases by 1 with every block away from the diagonal and the top-right and bottom left corner blocks, $G_{1,n}$ and $G_{n,1}$, are the most time consuming to calculate.

In this chapter, we presented the specialized computational tools we developed and applied to the study electronic transport in mesoscopic systems. We discussed briefly the history of GPUs and heterogenous computing and introduced the GPU-Matlab Interface (GMI), developed in this work. We present several benchmarks and compare GMI to several CPU approaches and several commercial GPU packages. We then proceed to explain how heterogenous computing pertains to transport calculations and present some pseudocode de-

operation	**thread 1**	**thread 2**	**thread 3**	**thread 4**	**device**
Calculate Ratio Matrices $\{S_1, S_i\}$ $i = 2, ..., n-1$	$S_1 = -(\epsilon - h_{C11})^{-1} h_{C12}$ $S_i = -(\epsilon - h_{C11} + h_{Ci,i-1} S_{i-1})^{-1}$				GPU GPU
Calculate Ratio Matrices $\{R_i, R_{n-1}\}$ $i = n-1, ..., 2$	$R_{n-1} = -(\epsilon - h_{Cnn})^{-1} h_{Cn,n-1}$ $R_i = -(\epsilon - h_{Ci+1,i+1} + h_{Ci+1,i+2} R_{i+1})^{-1} h_{Ci+1,i}$				GPU GPU
Calculate Digonal Blocks $\{D1, D_i, D_n\}$ $i = 2, ..., n-1$	$D_1 = (\epsilon - h_{C1,1} + h_{C1,2} R_1)^{-1}$ $D_i = (h_{Ci,i-1} * S_{i-1} + (\epsilon - h_{Ci,i}) + h_{Ci,i+1} R_i)^{-1}$ $D_n = (\epsilon - h_{Cnn} + h_{Cn,n-1} S_{n-1})^{-1}$				GPU GPU
Calculate $G_{1,n}$ $i = 9, ..., n-1$	$A_1 = S_1 * S_2$ $A_1 = A_1 * A_2$ $\vdots$ $A_1 = A_1 * A_2$ $A_1 = A_1 * A_2$ prod $= A_1 * A_2$ $G_{1,n} =$ prod $* D_n$	$A_2 = S_3 * S_4$ $A_2 = A_3 * A_4$ $\vdots$ $A_2 = A_3 * A_4$ $A_2 = A_3 * A_4$	$A_3 = S_5 * S_6$ $A_3 = S_i * S_{i+1}$ $\vdots$ $A_3 = S_{n-4} * S_{n-3}$	$A_4 = S_7 * S_8$ $A_4 = S_i + 2 * S_{i+3}$ $\vdots$ $A_4 = S_{n-2} * S_{n-1}$	GPU GPU $\vdots$ GPU GPU GPU GPU

Figure 4.10: Central Green's Function calculation pseudo-code. Parallelized (where) possible over four devices.

tailing our heterogenous parallel GPU NEGF-TB implementation. Here, GPUs are used for when the number of floating point operations is high (e.g. large, dense matrices) and sparse CPU routines are used for quick or sequential tasks (e.g. vector operations, sparse matrices etc.)

5

Applications

5.1 Transport properties of Silicon Nanowires

Filamentary crystals of Silicon were first fabricated about fifty years ago [40]. Traditionally, the term Silicon 'whiskers' was used to refer to these structures. In the mid 1990s, however, advances in microelectronics and fabrication techniques[41] sparked a new interest in these structures because of their applications as electrical nanowires[42]. Recently, much of the work on Silicon nanowires (SiNW) has been on their electronic properties and their applications to nanoscale electronic devices such as nanoscale-interconnects and nano-transistors[43, 44, 45].

Here, the transport properties of a hydrogen passiviated SiNW (see figure 5.1) are investigated using the NEGF-TB formalism discussed in chapter 2.3 combined with GPU acceleration. The nanowire has a diameter of 1.26 nm and is grown in the [110] direction.

The Si-Si and Si-H bonds were taken as 2.37 and 1.48 Å from the result of

an *ab-initio* DFT calculation with the Vienna Ab-Initio Simulation Package[46] (VASP) from [43]. The tight-binding Hamiltonian of the device was constructed using a 10-orbital $sp^3d^5s^*$ Hamiltonian for Si atoms and a single orbital Hamiltonian of the H atoms on the surface, as in [43] and [47]. The TB parameters and hamiltonians were obtained using 'Nanomatm' [48].

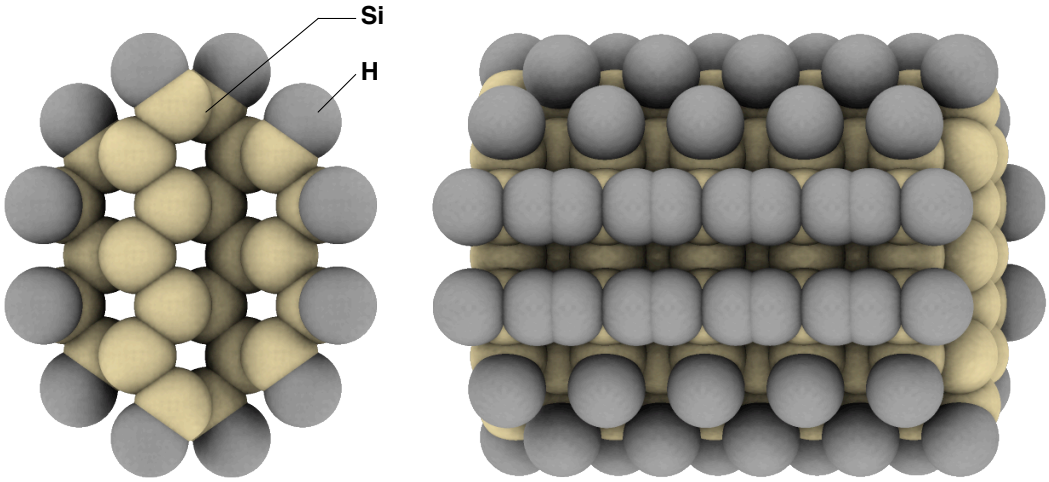


Figure 5.1: The structure investigated was a 0.46 diameter hydrogen passivated silicon nanowire grown in the [110] direction. Rendering was done with QuteMol[49]

Since the structure is periodic, the leads and central region on-site Hamiltonians are identical. The sparse, patterned structure of the on-site and coupling Hamiltonians H_{00} and H_{01} in the $sp^3d^5s^*$ basis is displayed in figure 5.2.

First, the Fermi level was be determined from a density of states calculation (see section 2.4) to lie in the band-gap. A band structure was then constructed from the tight-binding Hamiltonians and the transmission coefficient was calculated as a function of energy using the NEGF-TB formalism described in sections 2.3. The results are shown side-by-side in figure 5.3. Since the trans-

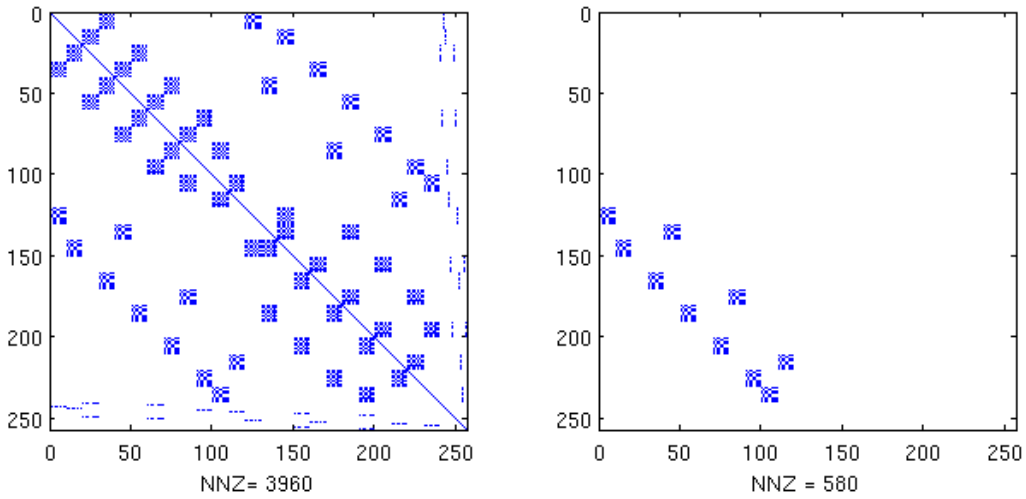


Figure 5.2: The sparse structure of the on site and coupling Hamiltonian matrices for a [110] SiNW in a ten orbital $sp^3d^5s^*$ basis set.

port is ballistic, the transmission in this case is just an integer count of the number of available bands at each energy point and vanishes at the band gap as expected.

The above structure contains 120 atoms in the simulation box. The matrices involved in the calculation are only 256×256 and this particular calculation does not fully benefit from the heterogenous NEGF code developed in this work. Its purpose is to validate the developed software. The results in transmission results in figure 5.3 closely match those obtained by Svizhenko et al [43]. In the next section, we present calculations involving vast systems with a much larger number of atoms, demonstrating the computational potential of our method.

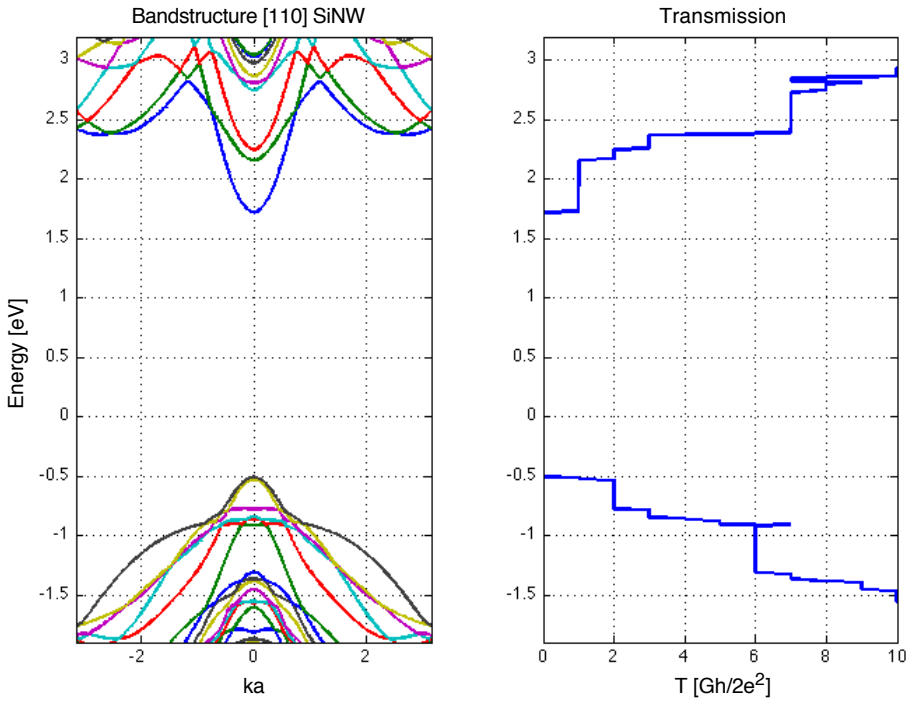


Figure 5.3: The band structure and transmission coefficient for a silicon nanowire are shown side by side with energy on the y-axis. The transmission vanishes at energy points lying in the bandgap. Since the transport is ballistic, the transmission takes on integer values equal to the number of available bands available at each energy range.

5.2 *Transport Properties of Silicon Nanobeams*

Single crystal silicon beams are among the most common structures in MEMS, usually in the form of a cantilevered beam [50, 51].

In general, the electrical properties of semiconductors are closely coupled to their mechanical properties. The field of electromechanical systems emerged with the discovery of the piezoresistive effect by C.S. Smith in 1954 [52]. Smith observed that "unaxial tension causes a change of resistivity in silicon and germanium of both n and p types", or in more utilitarian terms, that semiconduc-

tors make excellent pressure sensors.

With rapid improvements in micromachining and fabrication, the large scale production of Microelectromechanical Systems (MEMS) has become possible. Today, MEMS are ubiquitous in everyday electronic devices. Their applications include inertial sensors for game controllers, pacemakers, airbags, image stabilization in cameras, pressure sensors in altimeters and scuba gear, flow sensors for measuring engineer air intake in automobiles, IR sensors, fingerprint sensors and many others. A more complete list of applications can be found in [53, 54]

The mechanical properties of beams of single crystal silicon ranging from the nano to millimetre scale have been investigated extensively[55, 56, 57, 58]. However, these structures can contain a vast number of atoms. This makes studying their electronic properties from first principles unrealistic and semi-emperical modeling computationally arduous. The main bottleneck is linear algebra routines on large matrices.

In this section, we briefly review how silicon beams can be fabricated and apply our GPU accelerated NEGF-TB code using a work station equipped with four Nvidia Tesla C2050 GPUs to bypass the computational bottleneck and study their electronic transport properties.

5.2.1 Fabrication

Silicon nanobeams can be fabricated using field enhanced annodization with an AFM and anistropic wet etching. We summarize the fabrication procedure presented by Sunbdararajan *et al* [57]. The process starts with a Si separated

by implanted oxygen (SIMOX) wafer. From bottom to top, it is composed of thin SiO_2-Si-SiO_2 layers, a bulk Si layer and a thin SiO_2 layer. A portion of the bottom SiO_2 layer is etched with photolithography to expose the bulk Si layer on the bottom. The Si layer is then etched using anisotropic wet etching resulting in the trench shown in figure [57]. The top SiO_2 is then also etched away, exposing the thin Si diaphragm.

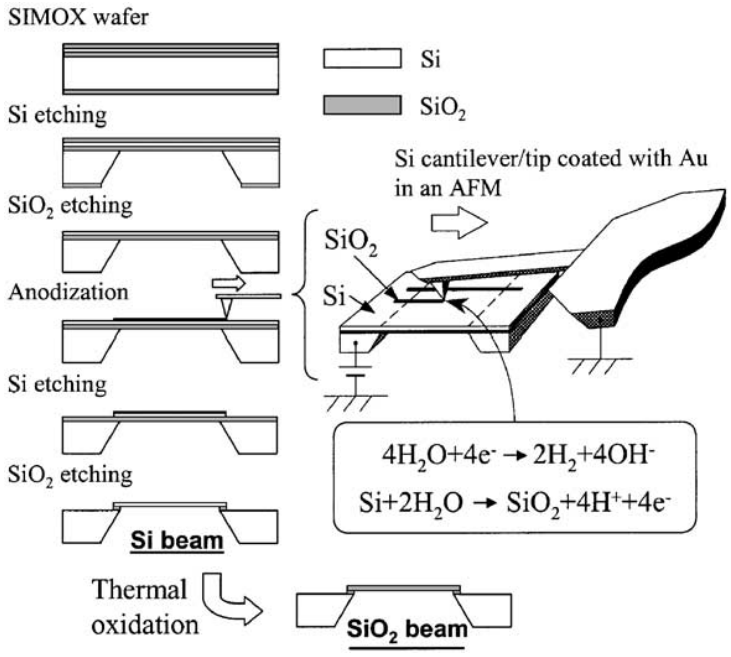


Figure 5.4: Fabrication of Si nanobeams with field enhanced annodization using AFM. Taken from [57].

A line of SiO_2 is then deposited on the Si diaphragm using field-enhanced anodization. The process works by applying a voltage bias to a cantilever tip in an AFM and moving it in a line across the diaphragm in air at room temperature. The tip oxidizes the surface of the Si diaphragm drawing a SiO_2 pattern

to be used as an etching mask. The width of the mask is proportional to the applied voltage bias. Finally, the unmasked portion of the Si diaphragm is etched anistropically resulting in a trapezoidal Si beam spanning across the trench, like a cantilever bridge.

The trapezoidal shape is due to the anistropic wet etching process because the etching rate varies profoundly depending on the crystal orientation of the exposed crystal face.

5.2.2 *The Effect of Vacancies on Transport*

Three basic [100] nanobeam structures of various sizes and shapes were studied. Nanobeam A is trapezoidal hydrogen passivated Si nanobeam containing 2,286 atoms. The cross section has a width varying from roughly $5.20 - 2.17$ nm and a height of 1.49 nm. The length of the structure in the transport direction was 4.87 nm. It is modelled as central region composed of 9 principal layers ($\sim$0.543 nm each). See figure 5.5.

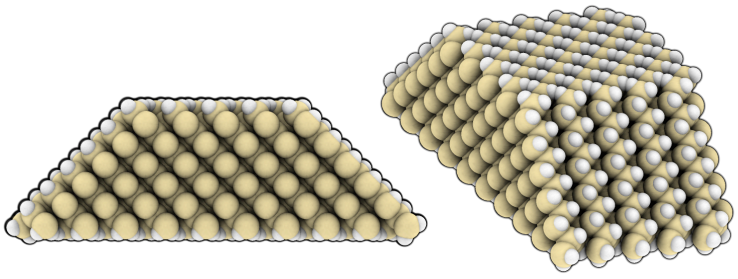


Figure 5.5: Atomic visualization of nanobeam A: Cross section and 3D lateral view.

Nanobeam B is trapezoidal hydrogen passivated Si nanobeam containing 14,059 atoms. The cross section has a width varying from roughly $10.59 - 4.61$

nm and a height of 2.99 nm. The length of the structure in the transport direction is 9.23 nm. It is modelled as central region composed of 17 principal layers (∼0.543 nm each). See figure 5.6.

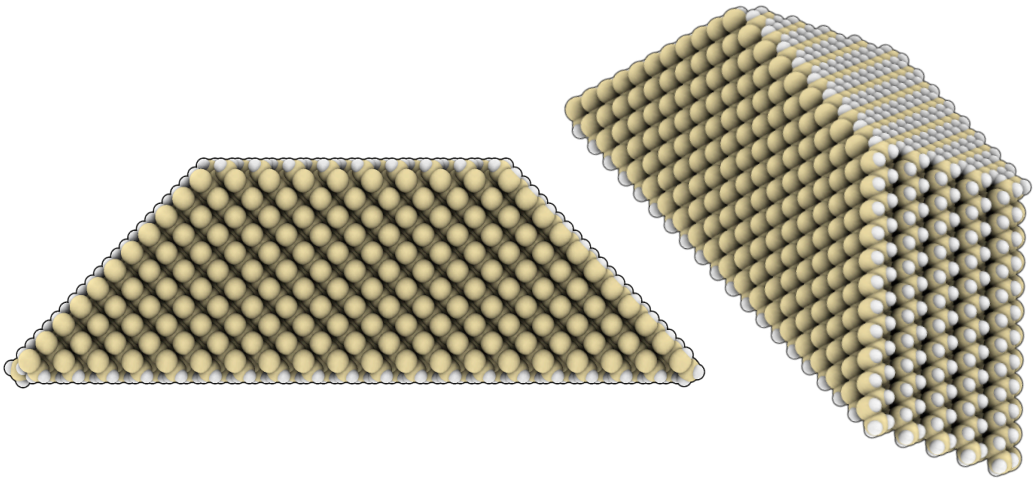


Figure 5.6: Atomic visualization of nanobeam B: Cross section and 3D lateral view.

Nanobeam C is a rectangular hydrogen passivated Si nanobeam containing 224,180 atoms. The cross section has a width and height of 8.1 nm. The length of the structure in the transport direction is 59.73 nm. It is modelled as central region composed of 110 principal layers (∼0.543 nm each). See figure 5.7.

We investigated the the effect of vacancies on the electronic transport properties. Physically, surface vacancies can arise on the edge of the structure as fabrication imperfections in the form of cracks, corners, dislocations, holes, crevices and steps[59]. Figure 5.8 shows an SEM image showing a detail of a defect on the surface of a Si Beam.

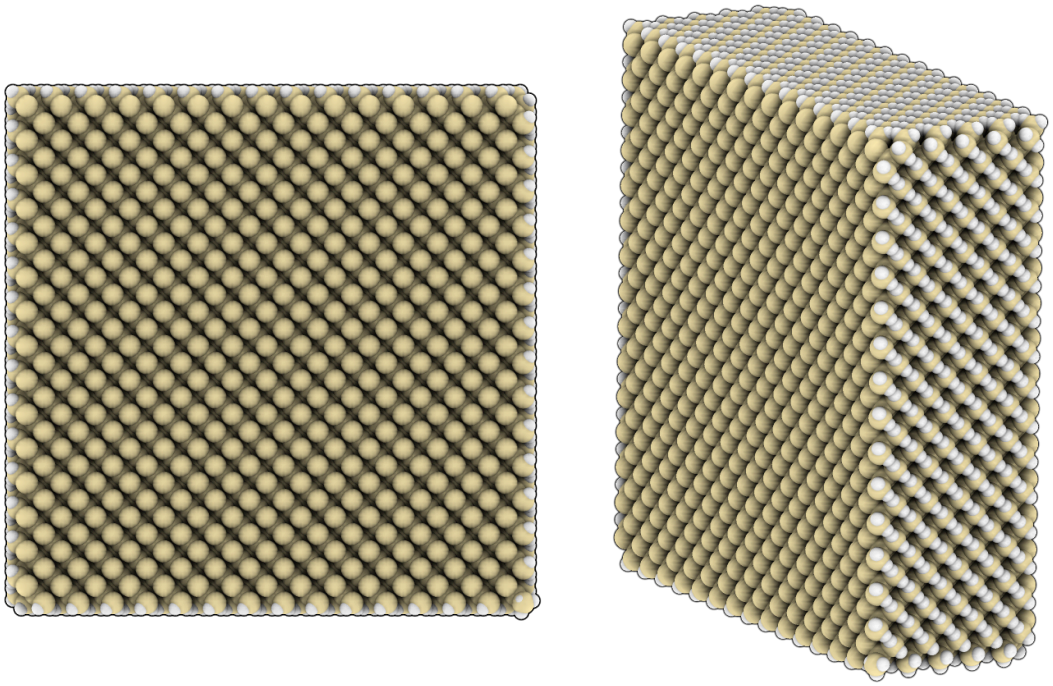


Figure 5.7: Atomic visualization of nanobeam C: Cross section and 3D lateral view.

Internal vacancy defects, on the other hand, can arise in the Si wafer itself before fabrication by vacancy diffusion. An energetic atom at the surface can spontaneously break its bonds and jump to a new surface location. Atoms from the bulk can repeatedly diffuse to fill the vacancy resulting in the a diffusion of the vacancy towards the bulk[59].

Vacancy diffusion in materials occurs for a variety of reasons. It can be well understood using Kinetic Monte Carlo techniques and can give rise to very interesting effects, such as memristance [60, 61]. Another kind of defect is a local lattice distortion as the result of a vacancy or substitutional impurity where the impurity atom is of a different size than the lattice atom[59].

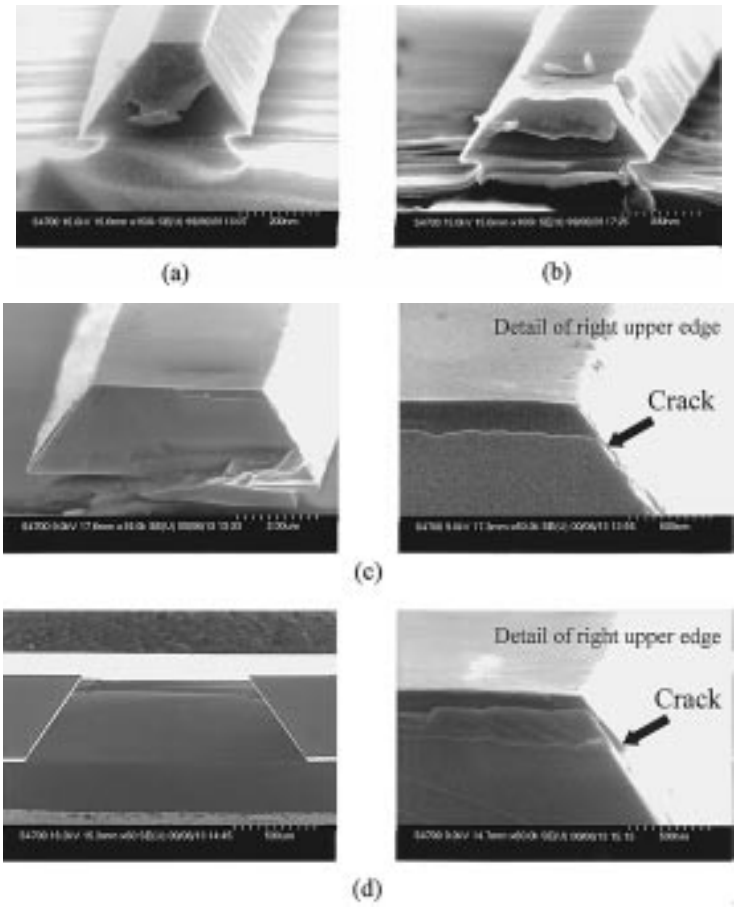


Figure 5.8: SEM image of Si nanobeam. Taken from [56].

In this work, non distortive internal and surface Si vacancy defects are considered. For structures, A and B, three cases are studied:

(i) Structure is vacancy free and periodic

(ii) Atomic vacancies distributed randomly throughout the structure.

(iii) Atomic vacancies distributed periodically in the structure to form a vacancy chain. It is believed that memristive features are due to the formation of similar

chains filaments of vacancies [61].

The leads were taken as vacancy free and periodic. The central region was taken to have vacancy concentration of 1 in 254 and 1 in 827 - one per principal layer - for structures A and for nanobeam B respectively. Only a vacancy free case was considered for nanobeam C.

The tight-binding Hamiltonians of nanobeams A and B were calculated using the same methods in section 5.2. Figure 5.9 shows the electronic band structures. As before, the Fermi level was calculated by contour integration of the Green's function, determined to lie in the band gap, and was shifted to lie at the peak of the highest valence band.

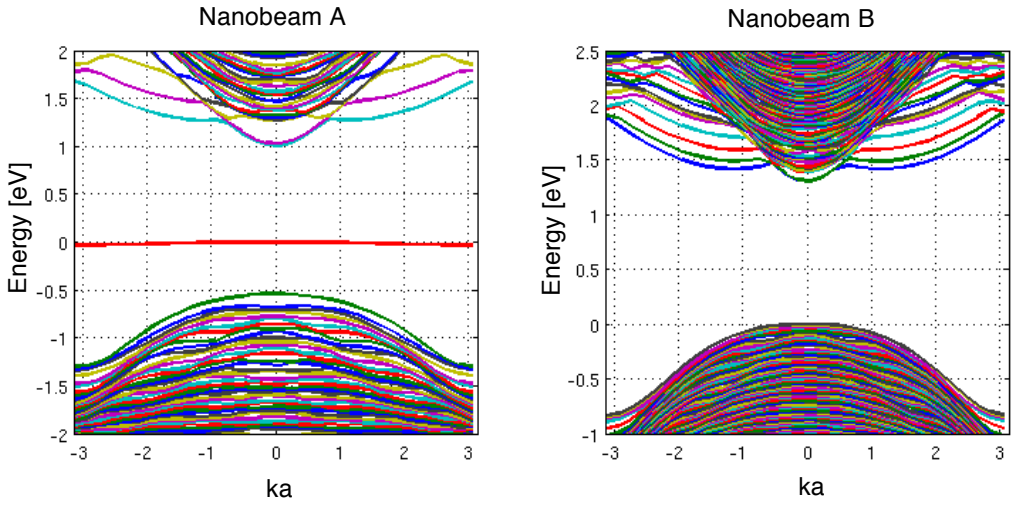


Figure 5.9: Nanobeam A and B electronic band structures.

The electronic structure of Si channels depends sensitively on several factors. The size of the channel, the group used to passivate the wire, the degree of

saturation, as well as the crystal orientation the channel is grown in all come into play. For [100] channels, First principles DFT calculations indicate that the band gap is largest when the surface is saturated with hydrogen and lowest when passivated with a hydroxyl (-OH) group [62]. For large diameter channels, not saturating the surface bonds leads to a lower and more highly indirect band gap [63]. The orientation of the channel not only affects the magnitude and position of the band gap, but even how sensitively is affected by structural defects such as roughness at an interface. [110] channels followed by [100] channels have the highest current and are the best cuts to use [64]. The band gap usually tends to be direct for small diameter channels and transitions to the bulk indirect gap as the channel diameter is increased [62, 63]. In our results, the band gaps of all considered nanowires and nanobeams were direct.

For nanobeam A, a pure system and an ensemble of 20 structures, 10 containing random vacancies and 10 with periodic vacancies, were considered. A mean value of the transmission was taken for each case. The results are presented in figure 5.10.

Since the self-energy calculation does not depend on the arrangement inside the simulation box, the calculation consisted of calculating the self-energies for all energy points first (0.01 eV mesh) first and then calculating $G(E)$ and $\mathcal{T}(E)$ for each structure at all energy points. Each self-energy calculation took $\sim$2-3 minutes (depending on convergence) and each Green's function and transmission function calculation took $\sim$20s. The total computation time including the pure

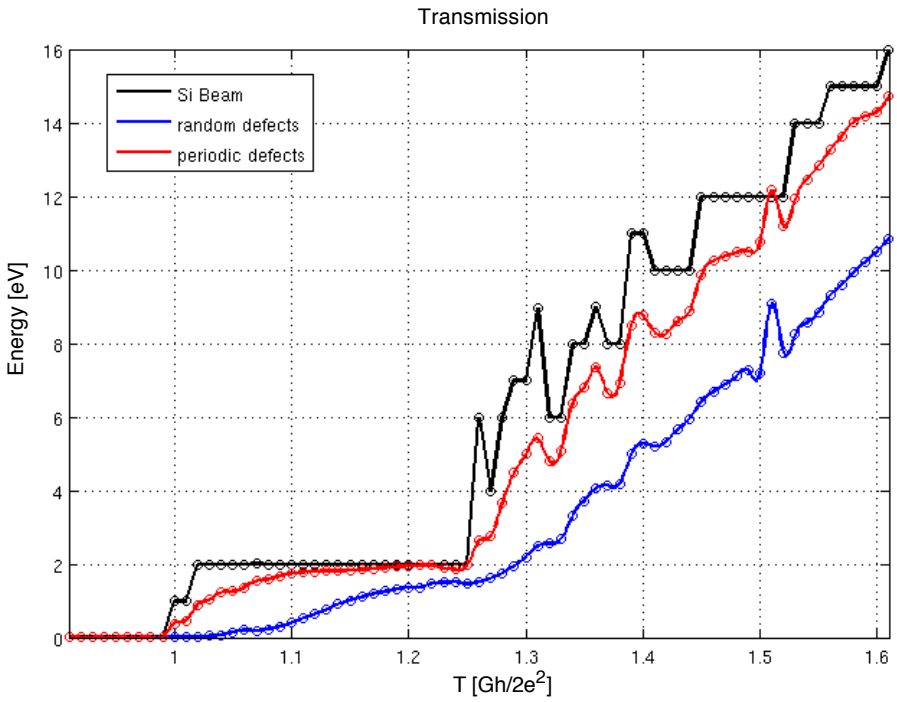


Figure 5.10: Transmission as a function of energy through nanobeam A. Random vacancies strongly degrade transmission.

structure and the ensemble average for 20 structures was ∼14.5 hours.

The transmission for periodic vacancy case follows the same behaviour as the vacancy free transmission curve, retaining the basic step pattern shape, but lags behind slightly. Additionally, the transmission is not an integer because even though the structure in the simulation box is periodic, the attached leads are vacancy free, thus breaking translational symmetry. Random defects, on the other hand, very strongly diminish the transmission in both structures considered.

We also present the unaveraged single transmission curves for periodic and

random defects in nanobeam A in figure 5.11. For periodic defects, the transmission was more strongly degraded in the case of internal defects than surface defects.

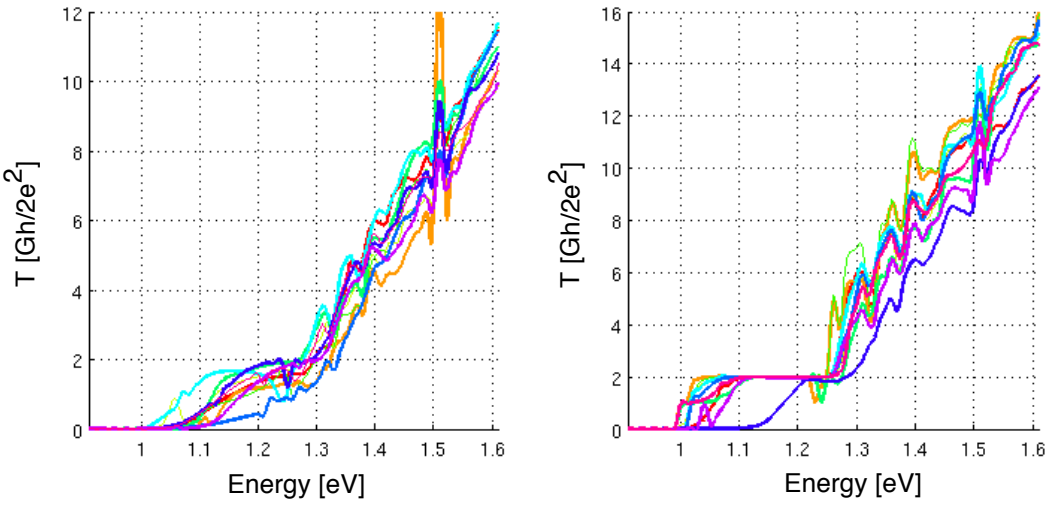


Figure 5.11: a) Transmission for an ensemble (10) of systems containing random vacancies. b) Transmission for an ensemble (10) of systems containing periodic vacancies. with basic structure A. The basic structure is nanobeam A in both cases.

Figure 5.12 displays a profile illustrating where the vacancies were located for a single principal layer (composed of four atomic planes) colour coded corresponding to each transmission curve in figure 5.11 b). The two curves with the lowest conductance both correspond to the two structures' internal vacancies. One way to interpret this would be a reduction in the effective width of the nanobeam due to defect back-scattering [65].

For nanobeam B, no spatial averaging of vacancies over an ensemble of systems was performed. We present the transmission curves for a pure structure, a structure with periodic defects, and a structure with random defects in figure

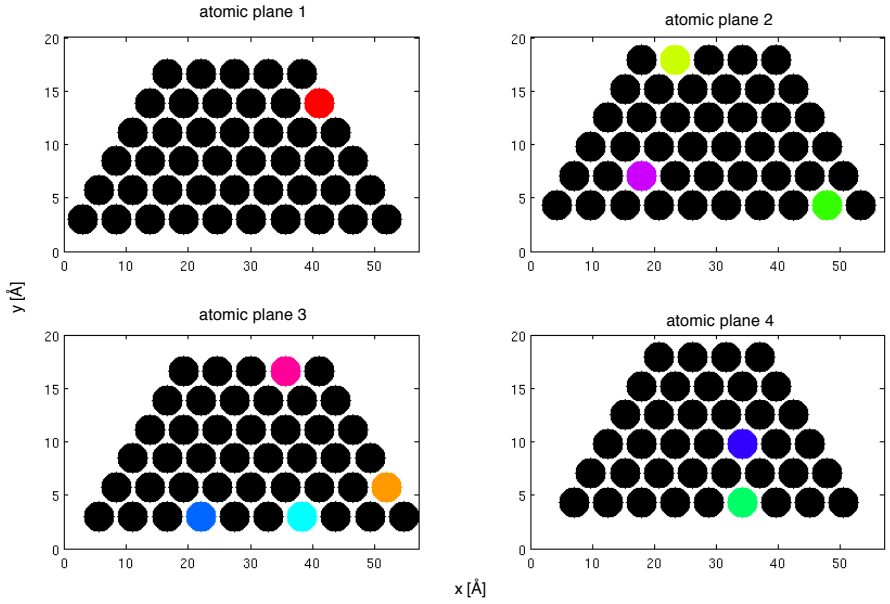


Figure 5.12: Location of vacancies in the principal layer (composed of four atomic layers) for each system with periodic vacancies considered.

5.13.

Once again, the calculation involved first calculating the self-energies at each energy point (0.01 eV mesh) and then calculating $G(E)$ and $\mathcal{T}(E)$ for each structure. The total computation time for all three arrangements was roughly $\sim$36 hours. Each self-energy calculation took $\sim$12 minutes and each Green's function and transmission function calculation took $\sim$8 minutes.

In order to test the peak computational power of our code, we computed the transmission at a single energy point through nanobeam C, shown in figure 5.7. The structure contains over 224,180 atoms. This corresponds to a block-tridiagonal central region Hamiltonian matrix H_C of $2,012,780 \times 2,012,780$ with the block diagonal consisting of 110 blocks of $18,298 \times 18,298$ each. The self-

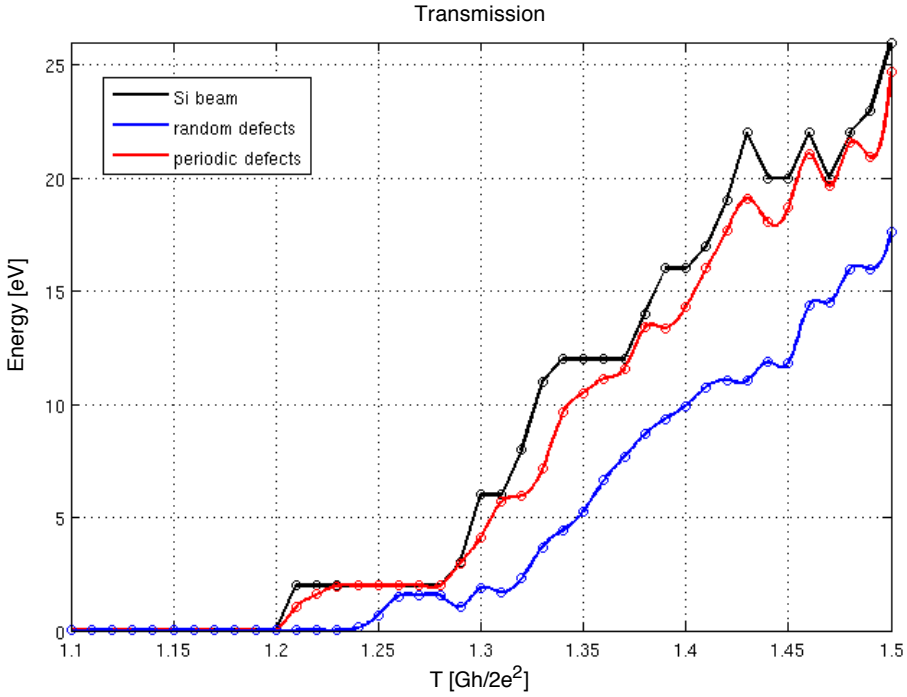


Figure 5.13: Transmission as a function of energies through nanobeam B. A pure system, a system with periodic vacancies, and a system with random vacancies are considered.

energy calculation converged to a tolerance of 10^{-8} within ∼4 hours. Computing the $G_{C1,n}$, the top-right most block of the Green's function (most difficult block), and subsequently the transmission function took ∼11 hours, an improvement on our result published in [1].

The calculation was done out of core by reading and writing calculated blocks to and from the hard disk using a special version of the NEGF-TB code. The GMI `block` functions (see appendix A) were used to achieve this. Rather than running several linear algebra operations in parallel as shown in the pseudocode in section 4.4, each individual linear algebra operation was

parallelized over all onboard devices. This approach is better suited for dealing with very large blocks.

Previously in section 3.3, we reviewed a method of mapping the large block tridiagonal matrix inversion problem to a more manageable chain of matrix inversions and matrix multiplications involving the blocks. For this reason, the computation time in our model scales as $O(n^3)$ with the number of atoms per principal layer, corresponding to the size of each on-site Hamiltonian matrix but as $O(n)$ with with the number of blocks.

Using one node equipped with 4 GPUs and the current code, we estimate that the the self-energies, $G(E)$ and subsequently $\mathcal{T}(E)$ for a system of $\sim$1,000,000 atoms can be computed in $\sim$2 days.

6

Conclusion

In this work, we have developed a parallel GPU accelerated code for carrying out transport calculations within the Non-Equilibrium Green's Function (NEGF) framework using the Tight-Binding (TB) model and reviewed the theoretical formalisms, modelling techniques, and computational tools used.

The GPU acceleration and parallelization was done using a multi-threaded GPU-Matlab Interface (GMI) developed in this work. Although GMI was originally intended to be used in the context of electronic transport calculations, it is not application specific and can be used by researchers in any field without any required knowledge of GPU programming or multi-threaded programming. Additionally, it was demonstrated that GMI's linear algebra performance competes well with commercial software and performs well when scaled to multiple GPU devices in parallel.

We validated our heterogenous parallel NEGF-TB software by studying the electronic transport properties of Si nanowires and comparing to known results. We then investigated the electronic transport behaviour of Si nanobeams and

studied the effect of random and periodic vacancies inside the beam on its conductance. We demonstrated that our method is capable of accurately simulating systems composed of over 200,000 atoms in reasonable timescales using only 1-4 GPU devices.

Several improvements, refinements, and optimizations remain to be made to our codes. Although the persistent linear algebra bottleneck was bypassed using a highly specialized implementation, the matrices we are capable of handling are so large that the calculation needs to be done out of core. Reads and writes to and from the hard disk now take a significant portion of the total computation time and need to be further minimized or eliminated.

Additionally, in the study of systems that require spatial averaging, such as systems involving point defects, vacancies and dopants, much computation time is spent on repeating the entire calculation for an ensemble of very similar systems. We are currently researching computational techniques to avoid repeating the same calculation each time.

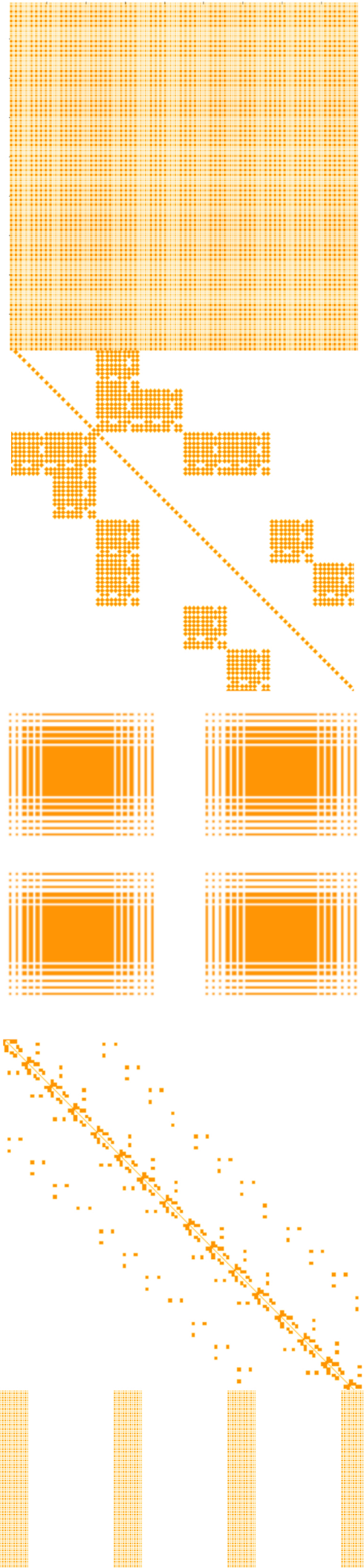
Since only the location of the defect or vacancy is altered in system, the Hamiltonians of each member of the ensemble at any given energy point differ only by row and column interchanges. A very promising implementation that we are currently investigating would be to compute the Green's function for only one system in the usual manner, and then compute only the change in the inverse for each member of the ensemble as a correction using the Sherman-Morison formula, thus greatly reducing the total computation time. However,

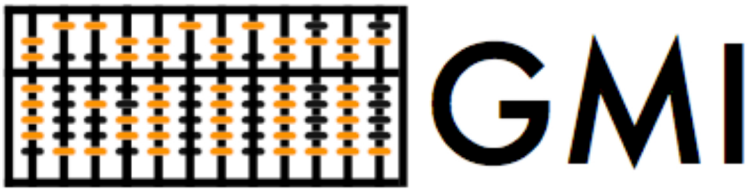
more work needs to be done before stronger claims can be made.

Scientific advancement has always been driven by increasingly sophisticated experimental techniques and the discovery of theories that describe the results. As we move forward to study increasingly complex systems, the gap between theory and experiment becomes progressively more difficult to bridge. This is a fundamental limitation that an entirely new field of physics, computational physics, has emerged to address.

Before the invention of the modern computer, scientists were limited to either using only models with closed form solutions or resorting to highly impractical and error-prone methods, such as human computers. Today and always, we are restricted to using models that can be efficiently solved by the computers and computational techniques known to us. In order to refine our theoretical models, it is very important to continuously develop strategies to circumvent bottlenecks and efficiently deal with the mathematical and computational issues that arise.

Even though the use of GPUs for solving computationally intensive scientific problems started fairly recently, accelerating applications by using specialized hardware is nothing new, and is often done *ad hoc* to boost computational performance. It is the author's impression that although it is unlikely that GPU computing is the final frontier of high performance computing (HPC), it is clear that it is by far the best currently available option.





Reference Manual

August, 25, 2012

The GPU Matlab Interface (GMI) is a set of functions that reroute linear algebra calls in Matlab to the the GPU for accelerated performance. The interface was designed with several features in mind.

1. Scalability: GMI can automatically access any number of onboard GPUs specified by the user.

2. Versatility: GMI was not developed in the context of strictly quantum transport. It can be readily used by any researcher in any field.

3. Accessibility: No knowledge GPU programming, multi-threaded programming, or CUDA is required.

4. Transparency: Users can very quickly port their CPU code to GPU code simply by replacing the native Matlab functions with GMI functions. No major changes to the application are required.

A.1 System Requirements

CUDA, CULA, and POSIX threads should be installed on the system. Please follow the instructions in their respective User Guides. At least one Nvidia CUDA (G8x series of higher) enabled GPU card must be on board. Nvidia Tesla cards are recommended. GMI was tested under Debian 'Wheezy' with Matlab 2010b on a workstation with 4 Nvidia Tesla C2050.

A.2 Preparing your Computer System

1. Make sure that the CUDA drivers/runtime versions and CULA libraries are up to date. This can be verified by running `nvcc -V` in a terminal. The CUDA compilation tool needs to be release 4.0 or higher.

```
harbm@dorothea:~$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_May_12_11:09:45_PDT_2011
Cuda compilation tools, release 4.0, V0.2.1221
```

2. Run the examples in the NVIDIA GPU Computing SDK directory, particualarly `deviceQuery.c` and `bandwidthTest.c`. These tests should return `PASSED`.

3. Make sure that all the CULA related environment variables are defined. This can be done by adding the following lines to the `.bashrc` file:

```
export CULA_ROOT="/usr/local/cula"
export CULA_INC_PATH="$CULA_ROOT/include"
export CULA_BIN_PATH_32="$CULA_ROOT/bin"
export CULA_BIN_PATH_64="$CULA_ROOT/bin64"
export CULA_LIB_PATH_32="$CULA_ROOT/lib"
export CULA_LIB_PATH_64="$CULA_ROOT/lib64"
export LD_LIBRARY_PATH=$CULA_LIB_PATH_64:$LD_LIBRARY_PATH
```

4. Run the CULA `basicUsage.c` example:

```
harbm@dorothea:/usr/local/cula/examples/basicUsage$ ./basicUsage
Allocating Matrices
Initializing CULA
Calling culaSgeqrf
Shutting down CULA
```

A.3 Installation

Download the GMI package

1. Extract the GMI folder into the home directory and define the relevant GMI environment variables by adding the following lines to the `.bashrc` file.

```
export GMI_ROOT="$HOME/GMI"
export GMI_SOURCE_PATH="$GMI_ROOT/source"
export GMI_EXEC_PATH="$GMI_ROOT/executables"
```

   Finalize these changes by typing '`source .bashrc`' into the command line.

2. Navigate to `$HOME/GMI` and run the `install.sh` script. This step will compile the GMI source code.

3. Start Matlab and add the compiled GMI executables' path:

```
>> addpath ~/GMI/executables
```

4. Try running the `gpuReveal` function to verify that the installation was done correctly. This should return some information about any GPUs onboard.

```
>> gpuReveal
Found 4 devices:
device 0: Tesla C2050 / C2070 (3 GB)
device 2: Tesla C2050 / C2070 (3 GB)
device 1: Tesla C2050 / C2070 (3 GB)
device 3: Tesla C2050 / C2070 (3 GB)
```

5. Congratulations! GMI is now ready to be used.

A.4 Data Types

Matlab labels arrays according to two complexity flags and 17 classes. The two complexity flags are `mxREAL` and `mxCOMPLEX` and they correspond to real and immaginary/complex arrays respectively. Of the 17 classes, only `mxSINGLE_CLASS` and `mxDOUBLE_CLASS`, corresponding to single and double precision, arrays are supported. Cells, logicals, strings and other classes of arrays are not supported. GMI functions are strongly typed and arguments of the appropriate complexity flag and class must be passed to each function.

GMI data type prefix	Complexity flag	Class ID
`S`	mxREAL	mxSINGLE_CLASS
`C`	mxCOMPLEX	mxSINGLE_CLASS
`D`	mxREAL	mxDOUBLE_CLASS
`Z`	mxCOMPLEX	mxDOUBLE_CLASS

Table A.1: GMI data-types

The Lapack convention is used; each function is preceded by one of the symbols `S`,`D`,`C`, or `Z` to denote its data type. For example, a function that takes a complex, single-precision array as an argument is preceded by the prefix `S`. Passing any different kind of array to that function will result in an error. The four symbols and their corresponding data types are summarized in table A.1.

A.5 Computation Types

GMI allows users to take advantage of one or several GPUs on their system with zero knowledge of multithreaded programming and GPU programming.

Users can either choose to use one GPU to work on one problem or several GPUs to work on several problems in parallel. All thread management is done under the hood and the user needs only to specify the input arguments and number of GPUs to be used. Additionally, users can take advantage of all GPUs onboard simultaneously to work on one large problem.

As with the data type symbols, one of three computation type prefixes, `Basic`, `Par`, and `Block` must precede any GMI function. A description of each computation type is presented below:

1. Basic computation: The basic computation type is labeled by the prefix

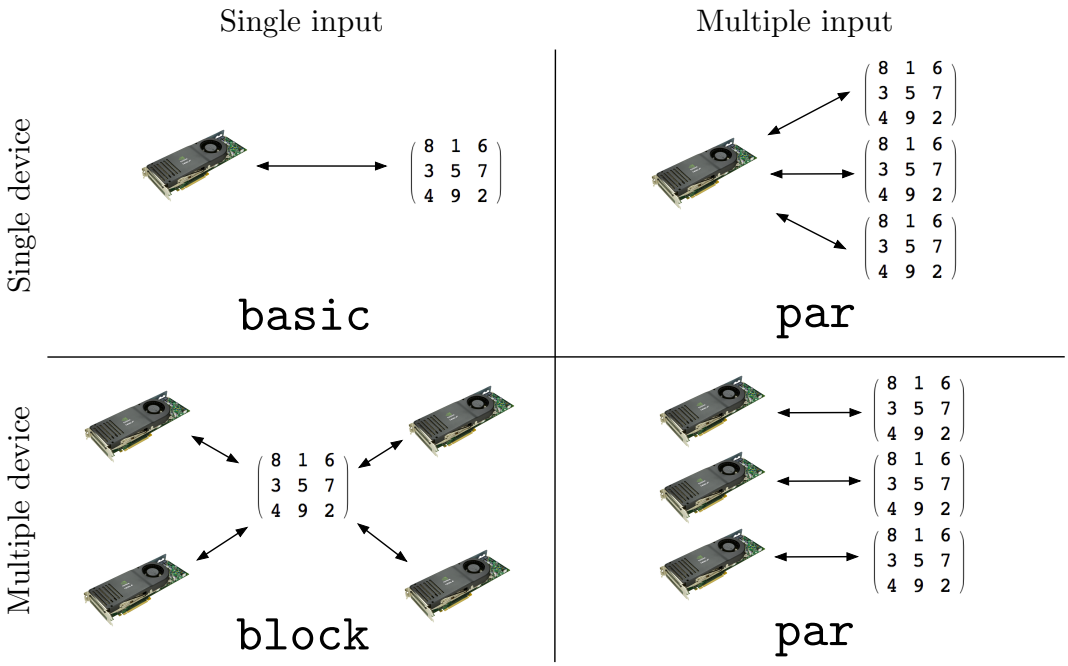


Figure A.1: GMI Computation types

`Basic` and denotes a computation involving a single algebra routine with a single input performed on one GPU. This is the simplest computation type. For example:

```
>>C=gpuBasicCinv(A);
```

performs a matrix inversion on the single precision, complex matrix `A` on the GPU and stores the result in `B`.

2. Parallel computation: The parallel computation type is denoted by the prefix `Par`. It performs a computation involving a single algebra routine with several inputs using one or several GPUs in parallel. The user simply

needs to specify the input arguments and the number of GPUs desired. For example:

```
>>[C1,C2,C3,C4]=gpuParZinv(4,A1,A2,A3,A4);
```

simultaneously inverts each of `[A1, A2, A3, A4]` on a separate GPU. If more than four arguments are passed, GMI distributes the extra arrays in the most optimal manner automatically. For example:

```
>>[C1,C2,...,CN]=gpuParZinv(4,A1,A2,...,AN);
```

automatically distributes all the data arguments over four GPUs and inverts them in parallel, four at a time. If the number of arguments is not a multiple of 4, the remaining data is simply sent to any idling devices.

3. Block computation: Unlike traditional processors, GPU memory is not shared across the devices and as a result, multiple GPU cannot be made to directly work on one large individual problem simultaneously. The `block` type was developed to address this issue and is the most advanced computation type supported. It performs a computation involving a single algebra routine with a single large input using all GPUs available onboard. This is very useful in cases where the input array is too large to fit in the memory of only one device. For example, in the case of matrix multiplication, consider the matrices A and B. Each can be subdivided into four

blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{A.1}$$

The blocks of the product $C = A \times B$ can be expressed in terms of the blocks of A and B as:

$$C = \begin{pmatrix} xA_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_12B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \tag{A.2}$$

The `gpuBlockDtimes` function computes the product $C = AB$ by computing the individual blocks of C in parallel.

```
C=gpuBlockDmtimes(A,B);
```

A block inversion is also available and uses the 2×2 block case of the RGF algorithm. See section 3.3.1. Although it seems appealing to always use the block functions instead of the basic and parallel functions, it is not recommended to do so because of the additional overhead these functions incur. The block functions are most suitable when dealing with one large array that is too large to fit on the memory of a single GPU.

A.6 Function Reference

The syntax of all GMI functions (except `gpuReveal`) follow the same easy to remember pattern: A `gpu` prefix, followed by the computation type and data

type prefixes and the linear algebra operation:

The section documents all functions available in GMI. For each function we present (i) a detailed description. (ii) A description of each of the function's input and out parameters. (iii) A description of some possible errors that may be encountered. (iv) Usage comments and guidelines if applicable.

A.6.1 `gpuReveal`

-Description

Displays the number of and some information about all onboard GPU devices.

-Input parameters

none

-Output parameters

none

- Errors

(i) No CUDA capable GPU devices are found

```
>> Found no devices onboard.
```

A.6.2 `gpuBasic{S,C,D,Z}inv`

-Description

Solves for the inverse of a square $N \times N$ matrix A using an LU decomposition. Similar to the matlab function `inv`.

-CULA Routines

- `GETRF`

- GETRI

-Input parameters

- A: Square $N \times N$ array.

-Output parameters

- C: Matrix inverse of A.

- Errors

(i) No input arguments were specified.

```
>> 1 input argument required.
```

(ii) More than one output arguments were specified.

```
>> 1 output argument required.
```

(iii) Input argument has the wrong data type or matrix is not square.

```
>> Input must be a square matrix of the appropriate data type.
```

- Example:

```
>> C=gpuBasicSinv(A);
```

A.6.3 gpuBasic{S,C,D,Z}mtimes

-Description

Performs a general matrix-matrix multiplication. Similar to the Matlab function mtimes

-CULA Routines

- `GEMM`

-Input parameters

- `A`: $M \times N$ array.
- `B`: $N \times K$ array.

-Output parameters

- `C`: $M \times K$ matrix $C = A \times B$.

- Errors

(i) Wrong number of arguments were specified.

```
>> 2 input argument required.
```

(ii) More than one output arguments were specified.

```
>> 1 output argument required.
```

(iii) A and B cannot be multiplied.

```
>> Inner dimensions must agree.
```

- Example:

```
>> C=gpuBasicCmtimes(A);
```

A.6.4 gpuPar{S,C,D,Z}inv

-Description

Inverts m square matrices using LU decomposition/backsubstitution on n GPU in parallel. This is done by distributing the matrices into n threads and starting one thread on each device. If the specified n is greater than the number of GPU available, the operation is automatically done on all available GPU. If $n > m$, then then n is automatically set to m.

-CULA Routines

- GETRF
- GETRI

-Input parameters

- n: Number of GPU to use.
- [A_1, A_2, ... ,A_m: m square arrays.

-Output parameters

- [C_1, C_2, ... ,C_m]: m square arrays containing the inverses of A_1, A_2, ... ,A_m.

- Errors

(i) No input arrays were specified.

```
>> two or more input arguments are required.
```

(ii) Input argument has the wrong data type or matrix is not square.

```
>> Input arguments must be square matrices of the appropriate data type.
```

- Example:

```
>> [C1 C2 C3 C4 ] = gpuParDinv(4,A1,A2,A3,A4);
```

A.6.5 gpuPar{S,C,D,Z}mtimes

-Description

Multiplies m *pairs* of matrices in parallel on n GPU. This is done by distributing the pairs of matrices into n threads and starting one thread on for device. If the specified n is greater than the number of GPU available, the operation is automatically done on all available GPU. If $n > m$, then then n is automatically set to m.

-CULA Routines

- `GEMM`

-Input parameters

- `n`: Number of GPU to use.
- `A_1, B_1, ... ,A_m,B_m`: m pairs of arrays.

-Output parameters

- `[ C_1,...,C_m]`: m arrays containing the products of $[A\_1 \times B\_1, ..., A\_m \times B\_m]$

- Errors

(i) No input arrays were specified.

```
>> two or more input arguments are required.
```

(ii) Input argument has the wrong data type or matrix is not square.

```
>> Input arguments must be square matrices of the appropriate data type.
```

(iii) matrices are not paired properly.

```
>> An even number of matrices is required.
```

(iv) Some matrix pairs inner dimensions do not agree.

```
>> Inner dimensions of all pairs of matrices must agree.
```

- Example:

```
>> [F1 F2 F3 F4] = gpuParZmtimes(4,A1,A2,B1,B2,C1,C2,D1,D2);
```

A.6.6 gpuBlock{S,C,D,Z}inv

-Description

Inverts a single large square matrix A using all available GPUs using the 2×2 matrix partitioning technique.

-CULA Routines

- GETRF
- GETRI

- `GEMM`

-Input parameters

- `A`: Large $N \times N$ array.

-Output parameters

- Matrix inverse of A.

- Errors

(i) No input arguments were specified.

```
>> 1 input argument required.
```

(ii) More than one output arguments were specified.

```
>> 1 output argument required.
```

(iii) Input argument has the wrong data type or matrix is not square.

```
>> Input must be a square matrix of the appropriate data type.
```

- Example:

```
>> A=rand(40000); %Test matrix. Make sure input is large.
>> C= gpuBlockDinv(A);
```

A.6.7 `gpuBlock{S,C,D,Z}mtimes`

-Description

Performs a single general matrix-matrix multiplication on two large arrays using all available GPUs.

-CULA Routines

- `GEMM`

-Input parameters

- `A`: Large $M \times N$ array.
- `A`: Large $N \times K$ array.

-Output parameters

- `C`: Matrix product $C = A \times B$.

- Errors

(i) Wrong number of arguments were specified.

```
>> 2 input argument required.
```

(ii) More than one output arguments were specified.

```
>> 1 output argument required.
```

(iii) A and B cannot be multiplied.

```
>> Inner dimensions must agree.
```

- Example:

```
>> A=rand(20000)+rand(20000)*i;  %Test matrix. Make sure input is large.
>> B=rand(20000)+rand(20000)*i;  %Test matrix. Make sure input is large.
>> C= gpuBlockZmtimes(A,B);
```

B

Source Code

The following is the C/MEX source code for GMI function gpuParZinv.c along with some comments. Other GMI functions follow a similar structure. They are not included here for compactness of this thesis. GMI can be obtained at [66]

```
1  /*****************************************************************
2  Function: gpuParZinv.c
3  Usage: Inverts matrices in parallel on several GPU.
4  Author: Harb, Mohammed
5  *****************************************************************/
6  #include "GMIframework.h"
7
8  /*Define a custom data type. Each thread will be passed on of these
       .*/
9  typedef struct{
10     int id;
11     int thisNumMatrices;
12     culaDoubleComplex** a;
```

```
13      int* n;
14      int** ipiv;
15  }threadArgType;
16
17  /*Single Process Multiple Data (SPMD) thread routine. This part of
        the code will
18   * execute on several devices in parallel each with a different
        input.*/
19  void* Zinv(void* arg){
20      int i;
21
22      /*Catch this thread's threadArg and type cast it to the
            appropriate type.*/
23      threadArgType* package = (threadArgType*) arg;
24
25      /*Define a status variable for debugging purposes.*/
26      culaStatus status;
27
28      /*Each thread will bind to a different device.*/
29      status=culaSelectDevice(package->id);
30      checkStatus(status);
31
32      /*Initialize CULA*/
33      status=culaInitialize();
34      checkStatus(status);
35
36      /*Loop over matrices assigned to the thread.*/
```

```
37      for(i=0;i<package->thisNumMatrices;i++){
38
39          /*Compute the LU factorization of a matrix using partial
                pivoting with row interchanges.*/
40          status=culaZgetrf((package->n)[i], (package->n)[i], (
                package->a)[i], (package->n)[i], (package->ipiv)[i]);
41          checkStatus(status);
42
43          /*Compute the inverse of a matrix using the LU
                factorization from GETRF.*/
44          status=culaZgetri((package->n)[i], (package->a)[i], (
                package->n)[i], (package->ipiv)[i]);
45          checkStatus(status);
46      }
47  }
48
49  /*Gateway function.*/
50  void mexFunction( int nlhs, mxArray *plhs[],
51          int nrhs, const mxArray *prhs[]) {
52
53      int i, j, k, m;
54
55      /*Make sure there is at least one matrix to operator on.*/
56      if(nrhs<2)
57          mexErrMsgTxt("gpuZparinv: Too few arguments");
58
59      /*The first argument is how many devices gpuZparinvshould use*/
```

```
60      int numThreads=(int)mxGetScalar(prhs[0]);
61
62      /*number of matrices the user passed*/
63      int numMatrices=(int)nrhs-1;
64
65      /*This is to prevent the user from using more threads than
            matrices.*/
66      if(numMatrices<numThreads)
67          numThreads=numMatrices;
68
69      /*matrixPerThread is an array of length numThreads that
            specifies how
70       *many matrices each thread will handle. e.g. if
             matrixPerThread[2]==5, this
71       *means thread #2 will handle 5 matrices.
72       */
73      int matrixPerThread[numThreads];
74      memset(matrixPerThread, 0, numThreads*sizeof(int));
75
76      /*distribute the number of matrices each thread will handle as
            evenly
77       *as possible. */
78      for(i=0;i<numMatrices;i++){
79          matrixPerThread[i%numThreads]++;
80      }
81
82      /*define an array of threadArgs of length user inputed
```

```
        numThreads.*/
83      threadArgType threadArgBundle[numThreads];
84
85      /*Allocate memory for each element of each threadArg in the
            bundle
86       * depending on how many matrices that thread is handling.*/
87      for(i=0;i<numThreads;i++){
88          threadArgBundle[i].n=(int*)mxMalloc(matrixPerThread[i]*
                sizeof(int));
89          threadArgBundle[i].a=(culaDoubleComplex**)mxMalloc(
                matrixPerThread[i]*sizeof(culaDoubleComplex*));
90          threadArgBundle[i].ipiv=(int**)mxMalloc(matrixPerThread[i]*
                sizeof(int*));
91      }
92
93      /*k starts at 1 for prhs because the first argument is the
            number of GPU
94       *being used. Use k-1 for plhs*/
95      for(i=0, k=1; i<numThreads; i++){
96
97          /*Give each threadArg a unique numeric ID and set the
                number of matrices
98           *it will handle.*/
99          threadArgBundle[i].id=i;
100         threadArgBundle[i].thisNumMatrices=matrixPerThread[i];
101
102         for(j=0;j<matrixPerThread[i];j++, k++){
```

```
103
104             /*Make sure arguments are square matrices.*/
105             m=(int)mxGetM(prhs[k]);
106             threadArgBundle[i].n[j]=(int)mxGetN(prhs[k]);
107             if(m != threadArgBundle[i].n[j])
108                 mexErrMsgTxt("gpuZparinv: Arguments must be square
                        matrices.");
109
110             /*Allocate memory for pivot variable (needed for GETRF)
                    .*/
111             threadArgBundle[i].ipiv[j]=(int*)mxMalloc(
                    threadArgBundle[i].n[j]*sizeof(int));
112
113             /*Make sure arguments are double complex matrices.*/
114             if(!mxIsDouble(prhs[k]) || !mxIsComplex(prhs[k]))
115                 mexErrMsgTxt("gpuZparinv: Arguments must be double
                        complex matrices.");
116
117             /*Allocate memory for each matrix and convert it to
                    Lapack ordering*/
118             threadArgBundle[i].a[j]=(culaDoubleComplex*)mxMalloc(
                    threadArgBundle[i].n[j]*threadArgBundle[i].n[j]*
                    sizeof(culaDoubleComplex));
119             mat2cula(threadArgBundle[i].a[j], mxGetPr(prhs[k]),
                    mxGetPi(prhs[k]), threadArgBundle[i].n[j]*
                    threadArgBundle[i].n[j] );
120
```

```
121         }
122     }
123
124     /*Create the threads, each will execute on a GPU device with
            different
125      * arguments.*/
126     pthread_t threadBundle[numThreads];
127     for(i=0; i<numThreads; i++){
128         pthread_create(&threadBundle[i], NULL, Zinv, &
                threadArgBundle[i]);
129         sleep(1);
130     }
131
132     /*Wait for all threads to finish execution before stepping
            forward.*/
133     for(i=0; i<numThreads; i++){
134         pthread_join(threadBundle[i], NULL);
135     }
136
137     for(i=0, k=0; i<numThreads; i++){
138         for(j=0;j<matrixPerThread[i];j++, k++){
139
140             /*Allocate memory for outputs and convert them to
                    mxArray ordering.*/
141             plhs[k]=mxCreateDoubleMatrix(threadArgBundle[i].n[j],
                    threadArgBundle[i].n[j], mxCOMPLEX);
142             cula2mat(threadArgBundle[i].a[j], mxGetPr(plhs[k]),
```

```
                    mxGetPi(plhs[k]), threadArgBundle[i].n[j]*
                    threadArgBundle[i].n[j]);
143
144             /*deallocate the copy used to change ordering from
                    mxArray to Lapack.*/
145             mxFree(threadArgBundle[i].a[j]);
146
147         }
148     }
149 }
```

Bibliography

[1] J. Maassen M. Harb V. Michaud-Rioux Y. Zhu H. Guo. Quantum transport modelling from atomic first principles. *Procedings of the IEEE (to appear)*, 2012.

[2] Intel Corporation. Intel 4004 Data Sheet, http://datasheets.chipdb.org/Intel/MCS-4/datashts/intel-4004.pdf.

[3] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):33–35, 1965.

[4] Wolfram | Alpha Pro. Moore's law, 2012. [Online; accessed 5-September-2012].

[5] K. Capelle. A bird's-eye view of density-functional theory. *Brazilian Journal of Physics*, 36(July):69, 2002.

[6] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, Nov 1964.

[7] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965.

[8] J. Junquera P. Ordejon D. Sanchez-Portal J.Soler E.Artacho J.D. Gale A. Garcia. The siesta method for ab initio order-n materials simulation. *Journal of Physics: Condensed Matter*, 14(11):2745–2779, 2002.

[9] C.M. Goringe D.R. Bowler and E Hernandez. Tight-binding modelling of materials. *Reports on Progress in Physics*, 60(12):1447–1512, 1997.

[10] N. Sergueev and H. Guo. Negf-dft: a first principles formalism for modeling molecular electronics. *HIGH PERFORMANCE COMPUTING SYSTEMS AND APPLICATIONS*, pages 2–4, 2003.

[11] J. Taylor. "Ab Initio *Modelling of Transport in Atomic Scale Devices*". PhD thesis, McGill University, 2000.

[12] A. Pechia G. Penazzi L. Salvucci and A Di Carlo. Non-equilibrium green's functions in density functional tight binding: method and applications. *New Journal of Physics*, 10(6):065022, 2008.

[13] Marco Buongiorno Nardelli. Electronic transport in extended systems: Application to carbon nanotubes. *Phys. Rev. B*, 60:7828–7833, Sep 1999.

[14] D. Waldron L. Liu H. Guo. Ab initio simulation of magnetic tunnel junctions. *Nanotechnology*, 18:424026, 2007.

[15] V. Volkov and J.W. Demmel. "benchmarking gpus to tune dense linear algebra". *2008 SC International Conference for High Performance Computing Networking Storage and Analysis*, (November):1–11, 2008.

[16] Vasily Volkov and James Demmel. Using gpus to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. Technical Report UCB/EECS-2007-179, EECS Department, University of California, Berkeley, Dec 2007.

[17] Vasily Volkov and James Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[18] S. Datta. *Quantum Phenomena - Modular Series on Solid State Devices, Vol 8*. Addison-Wesley, New York, 1989.

[19] S. Datta. *Electronic Transport in Mesoscopic Systems*. Cambridge University Press, 1995.

[20] S. Datta. Nanoelectronic devices: A unified view. *The Oxford Handbook on Nanoscience and Nanotechnology*, 1:1–26, 2008.

[21] A. Douglas Stone and A. Szafer. What is measured when you measure a resistance? – the landauer formula revisited. *IBM J. Res. Develop*, 32(3):384–413, 1988.

[22] Y. Ke. *"Theory of Non-Equilibrium Vertex Correction"*. PhD thesis, McGill University, 2010.

[23] D. Waldron. "Ab Initio *Simulation of Spintronic Devices"*. PhD thesis, McGill University, 2007.

[24] N. Wingreen A. Jauho Yigal Meir. "time-dependent transport through a mesoscopic structure". *Phys. rev. B*, 2006.

[25] H. Huag and A.P. Jauho. *Quantum Kinetics in Transport and Optics of Semi-Conductors*. Springer, New York, 1996.

[26] D. H. Lee and J. D. Joannopoulos. Simple scheme for surface-band calculations. ii. the green's function. *Phys. Rev. B*, 23:4997–5004, May 1981.

[27] M P Lopez Sancho J M Lopez Sancho J Rubio. Quick iterative scheme for the calculation of transfer matrices: application to mo (100). *Journal of Physics F: Metal Physics*, 14(5):1205, 1984.

[28] Cambridge New, York Port, Chester Melbourne, William T Vetterling, Saul A Teukolsky, William H Press, and Brian P Flannery. *Numerical Recipes in C*, volume 9. Cambridge University Press, 2002.

[29] V. Michaud-Rioux and H. Guo. A novel numerical method for *Ab Initio* electronic structure calculations. *unpublished*, 2011.

[30] Khronos Opencl, Working Group, and Aaftab Munshi. "opencl specification". *ReVision*, 55(2):1–385, 2011.

[31] Washington University D. Gohara, Center for Computational Biology. MacResearch OpenCL tutorials [Podcast], 1971.

[32] J. Humphrey D. Price K. E. Spagnoli A. L. Paolini E. J. Kelmelis. "cula: Hybrid gpu accelerated linear algebra routines". *SPIE Defense and Security Symposium (DSS)*, 2010.

[33] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[34] "nvidia cuda programming guide 2.0". *NVIDIA Corporation*, Version 3.(2.3.1):1–111, 2010.

[35] J. Sanders and E. Kandrot. *"CUDA by Example"*. Addison-Wesley, 2010.

[36] Whitepaper Nvidia, Next Generation, and Cuda Compute. "whitepaper nvidiaÕs next generation cuda compute architecture". *Revision*, 23(6):1–22, 2009.

[37] Accelereyes. Jacket, 2012.

[38] Guangran Kevin Zhu. Scalapack-matlab interface (smi). private communication.

[39] Accelereyes GPU Software BLOG. Jacket powers mars research, 2012.

[40] R.G. Treuting and S.M. Arnold. Orientation habits of metal whiskers. *Acta Metallurgica*, 5:598, 1957.

[41] R.S. Wagner and W.C. Ellis. Vapor-liquid-solid mechanism single of crystal growth. *Applied Physical Letters*, 4:125417, Feb 1964.

[42] Volker Schmidt, Joerg V Wittemann, Stephan Senz, and Ulrich Gosele. Silicon nanowires: A review on aspects of their growth and their electrical properties. *Advanced Materials*, 21(25-26):2681–2702, 2009.

[43] Alexei Svizhenko, Paul W. Leu, and Kyeongjae Cho. Effect of growth orientation and surface roughness on electron transport in silicon nanowires. *Phys. Rev. B*, 75:125417, Mar 2007.

[44] F. Sacconi M. P. Persson M. Povolotskyi L. Latessa A. Pecchia A. Gagliardi A. Balint T. Fraunheim and A. Di Carlo. Electronic and transport properties of silicon nanowires. *Journal of Computational Electronics*, 6(1-3):329–333, 2007.

[45] Y. Zheng C. Rivas R. Lake T. B. Boykin. Electronic properties of silicon nanowires, 2004.

[46] Vienna ab-initio simulation package.

[47] Jean-Marc Jancu, Reinhard Scholz, Fabio Beltram, and Franco Bassani. Empirical spds* tight-binding calculation for cubic semiconductors: General method and material parameters. *Phys. Rev. B*, 57:6493–6507, Mar 1998.

[48] Lei Liu. Nanomatm. private communication.

[49] M. Tarinil P. Cignoni C. Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1237–1244, 2006.

[50] R. J. Wilfinger, P. H. Bardell, and D. S. Chhabra. The resonistor: a frequency selective device utilizing the mechanical resonance of a silicon substrate. *IBM J. Res. Dev.*, 12(1):113–118, January 1968.

[51] Nader Jalili and Karthik Laxminarayana. A review of atomic force microscopy imaging systems: application to molecular metrology and biological sciences. *Mechatronics*, 14(8):907 – 945, 2004.

[52] Charles S. Smith. Piezoresistance effect in germanium and silicon. *Phys. Rev.*, 94:42–49, Apr 1954.

[53] H. Heeren and P. Salomon. Mems: Recent developments, future directions. *Technology Watch*, 2007.

[54] Robert Bogue. MEMS sensors: past, present and future. *Sensor Review*, 27(1):7–13, 2007.

[55] T. Ando K. Sato M. Shikida T. Yoshioka. Orientation-dependent fracture strain in single-crystal silicon beams under uniaxial tensile conditions . *International Symposium on Micromechatronics and Human Science*, pages 55–60, 1997.

[56] T. Namazu Y. Isono T. Tanaka. Evaluation of size effect on mechanical properties of single crystal silicon by nanoscale bending test using AFM. *Journal of Microelectromechanical Systems*, 9(4):450–459, 2000.

[57] Sriram Sundararajan, Bharat Bhushan, Takahiro Namazu, and Yoshitada Isono. Mechanical property measurements of nanoscale structures using an atomic force microscope. *Ultramicroscopy*, 91:111 – 118, 2002.

[58] H. Kahn R. Ballarini A.H. Heuer. Dynamic Fatigue of Silicon. *Journal of Microelectromechanical Systems*, 8(8):71–76, 2004.

[59] Safa O. Kasap. *Principles of Electronic Materials and Devices*. McGraw Hill Higher Education, July 2005.

[60] L. Chua. Memristor-The missing circuit element. *IEEE Transactions on Circuit Theory*, (5):507–519, September 1971.

[61] D. Li, M. Li, F. Zahid, J. Wang, and Hong Guo. Oxygen Vacancy Filament Formation in TiO_2: A Kinetic Monte Carlo Study. *J. App. Phys*, 2012.

[62] M. Nolan S. O'Callaghan G. Fagas. Silicon nanowire band gap modification. *Nano Letters*, 7(1):34–38, 2007.

[63] Y. Matsuda J. Tehir-Kheli W. A. Goddard. Surface and electronic properties of hydrogen terminated si [001] nanowires. *The Journal of Physical Chemistry C*, 115(25):12586–12591, 2011.

[64] M. Luisier A. Schenk W. Fitchner. Atomistic treatment of interface roughness in si nanowire transistors with different channel orientations. *Applied Physical Letters*, 90(10), 2007.

[65] F. Mazzamuto J. Saint-Martin V. Hung Nyugen C. Chassat P. Dollfus. Thermoelectric performance of visordered and nano structured graphene ribbons using Green's function method. *Journal of Computational Electronics*, 11(1):67–77, 2012.

[66] Mohammed Harb. Gpu-matlab interface, 2012.