Simulating Open Quantum Systems:
Computational Condensed Matter via Graphics Processing Units

An Honors Thesis
Department of Engineering Sciences and Applied Mathematics
McCormick School of Engineering
Northwestern University

Malcolm Spilka Lazarow
Completed on
June 5, 2017

supervised by
Professor Jens Koch
Department of Physics and Astronomy

Contents

1 Introduction

1.1 Project Motivation

It was Richard Feynman who first proposed the idea of a quantum computer [1]; the idea is essentially to replace a classical bit with a two-level quantum system. These two levels, denoted by $|0\rangle$ and $|1\rangle$, are called the *computational basis*. Whereas a classical algorithm manipulates classical bits, a *quantum algorithm* manipulates the computational basis with quantum mechanical dynamics.

Quantum algorithms can lead to astounding speed-up, in comparison to classical algorithms. An important example pertains to factoring an arbitrary integer into its prime factors. An open problem in computer science is whether or not it is possible to factor an arbitrary integer in polynomial time on a *classical* computer. There does exist a quantum algorithm, called Shor's Algorithm, which can factor an arbitrary integer in polynomial time [2]. This is but one example of the speed-up from a quantum algorithm, but there many more; for more examples and detailed explanations, see [2].

One of the main reasons that we do not have quantum computers today is because they are very difficult to build. In order to build a quantum computer we need to have a solid understanding of a quantum system interacts with its environment. A quantum system that is interacting with its environment is known as an *open quantum system*. The equations that describe the dynamics of open quantum systems are often very difficult, if not impossible, to solve exactly or analytically; therefore, they require the use of numerical schemes. In fact, these equations can consist of thousands of coupled differential equations, and so they can take quite a long time to solve numerically. It is not unheard of for a computation to take multiple days to solve! Of course, physicists want to speed up these computations; one way of doing so is by harnessing the power of *parallel computation*. One common method of parallel computation is the use of graphics processing units (GPUs).

GPUs were first developed for the purpose of computer graphics; without parallel computation, modern graphics would be nearly impossible because RGB levels for each pixel on a screen would have to be computed one after another. The computer on which I am typing this thesis has a screen resolution of 1366-by-768 pixels, meaning there are over one million pixels. Clearly, calculating the RGB levels for each pixel and then updating them, one after another, at a rate of 40 to 60 times per second, is an impossible task. GPUs were developed to solve this problem by using parallel computation; in other words, GPUs compute the RGB levels of each pixel simultaneously [3].

By 2003, computational scientists realized that they can "hack" GPUs so that the GPU processes data and executes custom algorithms, rather than their original purpose of computer graphics, [3]. This use of GPUs is called *General Purpose GPU Programming*. Over the years, GPUs have been updated to be more easily manipulated for scientific computation. One popular GPU model is the Nvidia Tesla [3]. All the computations for this thesis were implemented on a Tesla K40.

When I approached Professor Koch to do research, he gave me the opportunity to develop original code for a GPU to simulate open quantum systems. This thesis reports this project, which I have now completed. This thesis is arranged into five parts: this introduction, an overview of the relevant physics, a description of computation with GPUs, an application of

computation with GPUs to simulating of open quantum systems, and a conclusion.

Before diving into the material, I would like to thank Professor Koch and his Postdoctoral Fellow, Dr. Peter Groszkowski. Having started this project without any knowledge of quantum mechanics and having never heard of scientific computation with a GPU, I could not have mastered all of this material without their time, patience, and guidance. I have now completed graduate courses in quantum mechanics and am proficient in methods of parallel computation via GPUs; I am proud to present my success.

1.2 Notations and Conventions

Before beginning, I will now make a quick remark about notations:

1. $\dot{f}$ stands for the time derivative of a function f.
2. RK4 is an acronym for the numerical scheme Runge-Kutta 4.
3. We set $\hbar = 1$ throughout this entire thesis

2 The Relevant Physics: Qubits and Open Quantum Systems

2.1 Qubits

This section is dedicated to defining a qubit. After a discussion of what constitutes a good or bad choice for a qubit, we will give a somewhat detailed explanation of why the quantum harmonic oscillator is problematic as a qubit. This section will then end with a short explanation of a superconducting circuit, fluxonium, which is of interest in modern research.

Definition of a Qubit

A *qubit* is a quantum system with a two dimensional Hilbert space. Therefore, in order to realize a qubit, all we need is a quantum system that has two possible states. Qubits can be generalized to a quantum system with a d-dimensional Hilbert space, which is called a *qudit*. Although qudits are being researched, qubits are the proposed building block of a quantum computer as they are analogous to classical bits in a classical computer.

Examples of Qubits

A standard example of a qubit is the photon, wherein the photon's spin states form the computational basis. Recall that a photon, as a spin-1 particle, technically has three allowed spin states: $l = 1$ and $m \in \{-1, 0, 1\}$ (where l is orbital angular momentum and m is magnetic angular momentum). However, relativistic quantum mechanics requires that the photon can only occupy the $l = 1, m = \pm 1$ states, which is shown in more detail in [4]. Thus, we can label the $m = 1$ state as $|1\rangle$ and the $m = -1$ state as $|0\rangle$, giving us a computational basis. Of course, one can also take the two spin states of a spin one half particle, such as an electron. We will now consider other realizations of qubits and potential difficulties.

One can realize a qubit by taking any two states of a quantum system, trivially giving us a two dimensional Hilbert space; in particular, we can take the ground and first excited state of a given system. For example, we can consider the ground and first excited state of hydrogen, $|000\rangle$ and $|100\rangle$ (here, we are writing the states of hydrogen as $|nlm\rangle$ where n is the principal quantum number, l is the orbital angular momentum, and m is the magnetic angular momentum). While we can use hydrogen as a qubit, our choice for our computational basis is not aribtrary. In particular, the first and second excited states of hydrogen would be a bad choice for a computational basis for the following reasons:

1. The first excited state will decay to the ground state, so our computation basis is not "closed", in a sense.
2. The second excited state of hydrogen is degenerate, making controlled transitions between the first excited state and a *specific* second excited state possible, but potentially difficult.

Both of these facts are of paramount importance for what constitutes a good choice for the computational basis: it should be "self-contained" (meaning our system will not decay

into a state outside of our basis) and non-degenerate. Thus, while a quantum system might be a good choice for a qubit, we still have to carefully choose the states that will form the computational basis. We will now consider an example of a poor choice of a qubit: the quantum harmonic oscillator.

The Quantum Harmonic Oscillator as a Qubit

In order to see why the harmonic oscillator is problematic as a qubit, we must first recall a basic problem from quantum mechanics:

Suppose we prepare a quantum system in its ground state and then apply an external electric field, whose strength is given by $\cos(\omega t)$, to the system. If $\omega = E_1 - E_0$, the difference in the energy of the first excited state and the ground state, then how will the system evolve in time? The system will undergo so-called *Rabi oscillations*, in which the system sinusoidally oscillates between $|0\rangle$ and $|1\rangle$. In other words, if the system begins in $|0\rangle$ and we apply this external electric field, then in finite time the system will be in $|1\rangle$ with probability 1; as time progresses, the system will then oscillate between these two states. This is known as driving the system on resonance.

With this in mind, we can give an explanation as to why the quantum harmonic oscillator cannot be used as a qubit. Recall that the quantum harmonic oscillator is given by considering a quantum particle subject to a potential of the form $V(x) = \frac{1}{2}kx^2$. This results in the Hamiltonian

$$\hat{H} = \omega\left(a^\dagger a + \frac{1}{2}\right),$$

where $a^\dagger$ and a are creation and annihilation operators, respectively, which satisfy the commutation relation $[a, a^\dagger] = 1$. The energy spectrum is given by $E_n = \omega\left(n + \frac{1}{2}\right)$, where n is a non-negative integer, counting the excitation level of the system.

Note that each energy level is equidistant from its neighbors, i.e. $E_{n+1} - E_n = \omega$. Therefore, driving the system on resonance will result in the quantum harmonic oscillator's wavefunction having nonzero probability to be in state $|n\rangle$ once there is a nonzero probability that it is in state $|n-1\rangle$. Thus, if we drive the harmonic oscillator on resonance, all excited states will become partially occupied.

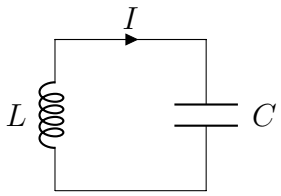
When working with a qubit, experimentalists must be able to control transitions between $|0\rangle$ and $|1\rangle$. The behavior of the quantum harmonic oscillator being driven on resonance makes the quantum harmonic oscillator difficult to control as a qubit. It is in this sense that the quantum harmonic oscillator is a poor choice for a qubit. However, the harmonic oscillator *can still* be used as a qubit, but it must be treated carefully. For example, one can couple the harmonic oscillator to another qubit; this coupling could result in a proper implementation of a quantum algorithm.

Brief Aside about Classical Circuits

In reality, physicists use numerous types of quantum systems as qubits; one such system is a superconducting circuit. Before delving into a specific example of a superconducting circuit, we will briefly review a basic classical electromagnetic circuit: an LC-circuit. The

purpose of this aside is to remind the reader that we can analyze a classical circuit by considering a fictitious *classical* particle in a potential; in the next section, when we look at a superconducting circuit, we will similarly analyze the circuit by considering a fictitious *quantum* particle in a potential.

Now, consider the LC-circuit:



where L is the inductance of the inductor, I is the current, and C is the capacitance of the capacitor.

The dynamics of the current of this circuit are governed by the differential equation

$$L\dot{I} + Q/C = 0.$$

where Q is the charge in the circuit, i.e. $\dot{Q} = I$. By dividing by L and differentiating with respect to time, we get

$$\ddot{I} + \omega^2 I = 0,$$

where $\omega = \frac{1}{\sqrt{LC}}$. Thus, we can think of the dynamics of the current of this LC-circuit as analogous to a particle whose position, x, is governed by the differential equation $\ddot{x}+\omega^2 x = 0$, which describes a harmonic oscillator. In fact, one can make a superconducting LC-circuit; this type of superconducting circuit is one way to realize the quantum harmonic oscillator.

Derivations and more detailed explanations of the above equations can be found in [5]. With this discussion in mind, we will move on to an example of a superconducting circuit.

2.1.1 Fluxonium

In this section, we will discuss fluxonium, which is a superconducting circuit of interest in modern research. Before describing specific characteristics of fluxonium, we will discuss some of the relevant aspects of superconductivity.

When a material becomes perfectly superconducting, its internal resistance is zero. Therefore, if a circuit's material becomes perfectly superconducting, a given current in the circuit will never die; moreover, superconducting circuits can be described in terms of quantum states [6]. It is for this reason that physicists seek to realize qubits via superconducting circuit: they provide a long-lived quantum system from which we can form a computational basis.

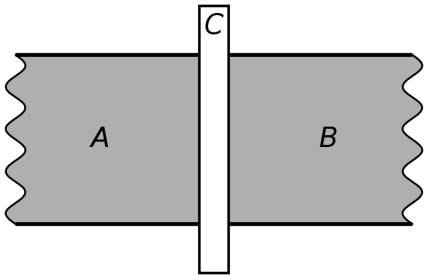
The physical theory explaining superconductivity is BCS theory, which was published in 1957 and discovered by Bardeen, Cooper, and Schrieffer [7]. BCS theory asserts that the phenomenon of *Cooper pairs* is the mechanism behind superconductivity [6]. Before being able to give a description of fluxonium, we will first describe what a Cooper pair is. This will enable us to describe the *Josephson effect*, leading into an explanation of a *Josephson junction*. This discussion is relevant and necessary because the Josephson junction is a circuit element of fluxonium.

Cooper Pairs

Electrons are spin half particles, meaning that a system of two coupled electrons behaves as a boson. Recall that if we have a group of bosons, then they can all occupy the same quantum state, as opposed to a fermion. Cooper pairs are loosely coupled electrons in a metal, thus these pairs behave as bosons. BCS theory asserts that superconductivity arises as a result of all electrons being coupled as Cooper pairs and that they *all* occupy the same state. The energy of this state has an energy lower than the Fermi energy [7].

Josephson Effect and Josephson Junction

Now, consider a superconducting circuit and suppose we interrupt the circuit with a small piece of insulating material, such as in the following image (taken from [8]):

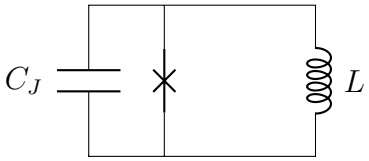


where A and B are superconducting wires and C is a piece of insulating material. This setup is known as a *Josephson junction*.

The Josephson effect is when Cooper pairs tunnel through the insulating material. In fact, a mathematical description of this phenomenon is nearly identical to the standard quantum mechanics problem of a particle tunneling across a thin barrier between two symmetric potential minima. This is described in greater detail in [6].

Fluxonium

The fluxonium qubit is a superconducting circuit in which a Josephson junction is placed in parallel with an inductor. This gives us the following circuit diagram [9]:



where the $\times$ denotes the Josephson junction. The loop on the right (consisting of just the inductor and the ideal Josephson junction) is permeated with an external magnetic field; the flux of the magnetic field through the loop is denoted by Φ_{ext}.

One property of quantum systems that arises in numerous problems is the notion of quantized, or discrete, energy. In quantum mechanics, the discretized energy spectrum is given by enforcing boundary conditions on the Schrödinger Equation. How one might apply this concept of quantization to superconducting circuits is not at all obvious. A rigorous

derivation of the quantization of the energy spectrum of superconducting circuits is simply beyond the scope of this thesis; therefore, the quantized Hamiltonian of fluxonium is given below and the curious reader is referred to [10] for more information on the theory. The Hamiltonian for fluxonium is given in [9] as

$$\hat{H} = 4E_c \left(i\frac{\partial}{\partial \varphi} \right)^2 + \frac{1}{2}E_L\varphi^2 - E_J \cos\left(\varphi - 2\pi\frac{\Phi_{ext}}{\Phi_0} \right),$$

where φ is a generalized variable related to the time integral of voltage across the junction, Φ_0 is the flux quantum, and E_c, E_L, and E_J are constants whose values are determined by the manufacturing of the fluxonium circuit. Again, more information on the physics of these terms, how they arise and what they describe, can be found in [9] and [10].

Much like the classical LC-circuit, where we can consider the current as analogous to a fictitious classical particle in a potential, we will now consider the problem of fluxonium to be analogous to that of a fictitious quantum particle in the potential

$$V(x) = \frac{1}{2}E_L x^2 - E_J \cos\left(x - 2\pi\frac{\Phi_{ext}}{\Phi_0} \right).$$

The mass of our fictitious particle completely determines the dynamics of our system. If the particle has a small mass, then the eigenenergy levels will be lower than if the particle has a large mass; therefore, decay and tunneling rates are dictated by the particle's mass.

So how do we specify the particle's mass? Recall that the momentum operator in quantum mechanics is given by $\hat{P} = i\frac{\partial}{\partial x}$ and that the Hamiltonian has the form $\hat{H} = \frac{1}{2m}\hat{P}^2 + V(x)$. Thus, it is the term E_c that determines the particle's mass. In fact, we have that $4E_c$ plays the same role as $\frac{1}{2m}$; thus, the larger the value of E_c, the smaller the mass and the smaller the value of E_c, the larger the mass.

We also note that $E_c/h, E_L/h$, and E_J/h are in units of GHz. A graph of the potential for specific values of E_c, E_L, E_J, and Φ_{ext} (given in the caption) is:

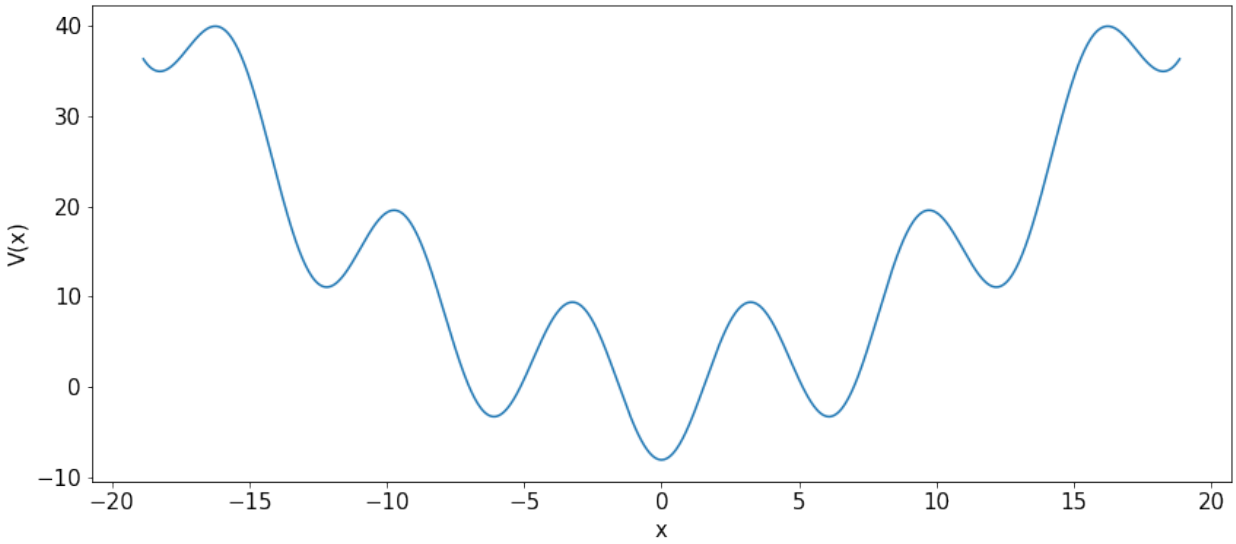


Figure 1: A graph of the potential energy of fluxonium with the values $E_L = 0.25, E_J = 8.1$, and $\Phi_{ext} = 0$. The problem of fluxonium is analogous to considering a fictitious quantum particle subjected to this potential.

We note that if we tune E_c, E_L, E_J, and Φ_{ext} properly, as is the case in Figure 1, the first wells on the left and right (with respect to the center) are at the same height, meaning they are degenerate [9]. In Section 3.4.1, we will evolve a state in the right well to see that the particle will eventually tunnel into the well on the left. Note that since the particle tunnels between these two wells, the state's wavefunction will be nonzero in the center well. Due to this degeneracy, these states are not preferable choices for our computation basis. Rather, the ground and first excited state in the center well are typically chosen to form a computational basis, assuming that the particle's mass is such that the first excited state in the center well is not degenerate.

Beyond tunneling, there is another interesting quantum effect going on in fluxonium. As noted in [11], the first well on the right and the first well on the left correspond to a currents moving in opposite directions, whereas the middle well corresponds to no current (assuming we take $\Phi_{ext} = 0$, as above). If we perform a measurement of current, then our wavefunction will collapse in the left, right, or center well. Thus, we will measure clock-wise, zero, or counter clock-wise current. The fact that the direction of the current changes as we take measurements and that our fictitious particle can tunnel between these wells are phenomena unique to the strange world of superconducting circuits. Any sense of a superposition of different directions of current would be absurd in the context of classical circuitry, but here it arises as a simple consequence of quantum mechanics.

2.2 Open Quantum Systems

In this section, we will describe how to quantitatively describe an open quantum system. An open quantum system is a quantum system which is coupled, in some sense, to the environment. For example, one might consider the environment to be a resonator, a thermal

bath, a single qubit, or even an ensemble of qubits. In order to create a quantum computer one needs to couple numerous qubits; thus, the creation of a quantum computer requires a complete understanding of how a quantum system interacts with its environment.

First, we will define the density operator, which will allow us to write the Lindblad master equation. It is the Lindblad master equation that governs the dynamics of open quantum systems. We will then give a conceptual explanation of the master equation framework, wherein we will describe how one quantifies a system and its coupling to the environment. Finally, we will conclude this chapter, and the first part of the thesis, by analyzing a qubit coupled to a thermal bath by solving the Lindblad master equation.

2.2.1 The Density Operator

Given an ensemble of quantum systems (for example, an ensemble of quantum particles), it is possible that multiple quantum states are occupied by various constituents of the ensemble. The density operator is used to describe such a statistical mixture. It is defined as:

$$\rho = \sum_j p_j \left|\phi_j\right\rangle \left\langle\phi_j\right|,$$

where p_j is the probability with which state $|\phi_j\rangle$ is measured in the ensemble.

A quantum system whose density operator is of the form $\rho = |\phi\rangle\langle\phi|$, namely a system that has probability of 1 of being in a specific state, is called a *pure state*. A system that has a nonzero probability of being in *multiple* states is called a *mixed state*. Therefore, the density operator is a natural object of study in quantum statistical mechanics.

Expectation Values with Density Operators

We will now show how to calculate expectation values in the density operator formalism. Recall that if our system is in the state $|\psi\rangle$ and we wish to find the expectation value of an operator $\hat{A}$, we compute $\langle\psi|\hat{A}|\psi\rangle$ (this quantity is denoted by $\langle\hat{A}\rangle$). Moreover, suppose that $|n\rangle$ forms a orthonormal basis for the Hilbert space of our system. We then have the identity operator, $|n\rangle\langle n|$, which has the property $\sum_n |n\rangle\langle n| = \mathbb{1}$.

We now compute:

$$\langle\hat{A}\rangle = \langle\psi|\hat{A}|\psi\rangle = \sum_n \langle\psi|\hat{A}|n\rangle\langle n|\psi\rangle = \sum_n \langle n|\psi\rangle\langle\psi|\hat{A}|n\rangle = \mathrm{tr}\Big(|\psi\rangle\langle\psi|\hat{A}\Big) = \mathrm{tr}\Big(\rho\hat{A}\Big)$$

Therefore, $\langle\hat{A}\rangle = \mathrm{tr}\Big(\rho\hat{A}\Big)$. Next, we will see that the density operator is at the core of the mathematical description of open quantum systems.

2.2.2 Lindblad Master Equation

The equation that is used to describe open quantum systems is the *Lindblad master equation*. Before beginning our discussion, it is important to mention that the Lindblad master equation is an approximation, meaning it is only valid if certain approximations are mathematically valid. More information on this can be found in the derivation in [12]. In this

section, we will denote the Hilbert space of the system as $\mathcal{H}^S$, that of the environment as $\mathcal{H}^E$, and that of the coupled system as $\mathcal{H}^{SE}$.

The Lindblad master equation describes a non-unitary time evolution of a quantum system via the time evolution of the system's density operator. The non-unitary aspect of the time evolution is due to the fact that the system is interacting with its surroundings; on the other hand, the dynamics of the Schödinger equation (describing a closed system) dictates that time evolution is given by a unitary transformation. The Lindblad master equation is given by:

$$\dot{\rho} = -i[\hat{H}, \rho] + \sum_j \gamma_j \left(\hat{L}_j \rho \hat{L}_j^\dagger - \frac{1}{2}\{\hat{L}_j^\dagger \hat{L}_j, \rho\} \right)$$

where the operators $\hat{L}_j$, which are referred to as the *Lindblad operators*, quantify the interaction between the environment and the system, γ_j is the *coupling constant* of the Lindblad operators to the system, and $\{\cdot, \cdot\}$ is the anti-commutator.

A more rigorous treatment of the following description of open quantum systems is given in [13]. Here we will give a heuristic description of how we can think of the relationship between the a quantum system and its environment. It is often useful to refer to the master equation as:

$$\dot{\rho} = \hat{\mathcal{L}}(\rho),$$

where $\hat{\mathcal{L}}$ is called the *Lindblad superoperator*. The first term in the master equation,

$$\dot{\rho} = -i[\hat{H}, \rho],$$

is referred to as the *von-Neumann equation*; in fact, it imposes a unitary time evolution and simplifies to the Schrödinger equation in the case when our system consists of a pure state. It can thus be thought of as a generalization of the Schrödinger equation to a system in a mixed state. It is the additional terms in the Lindblad master equation,

$$\sum_j \gamma_j \left(\hat{L}_j \rho \hat{L}_j^\dagger - \frac{1}{2}\{\hat{L}_j^\dagger \hat{L}_j, \rho\} \right),$$

that give us the interaction between our system and the environment, which leads to the non-unitary time evolution.

When describing an open quantum system, one distinguishes between the system itself, denoted by S, and the environment, denoted by E. Moreover, we have a density operator describing the coupled system and environment, denoted by ρ^{SE}, and a density matrix for the system itself, denoted as ρ^S. Note, since $\mathcal{H}^S$ and $\mathcal{H}^E$ are two disjoint subsystems of $\mathcal{H}^{SE}$, we can think of the Hilbert space of the coupled system as the tensor product $\mathcal{H}^{SE} = \mathcal{H}^S \otimes \mathcal{H}^E$.

As we saw in the previous section, we calculate expectation values in the density operator formalism by simply taking the trace of the density operator. Thus, we now introduce the *partial trace*. Consider ρ^{SE} as the density operator describing the coupled system and let $|\phi_i\rangle$ form an orthonormal basis for $\mathcal{H}^E$. We can then define the partial trace as:

$$tr_E : \mathcal{H}^{SE} \to \mathcal{H}^S; tr_E(\rho^{SE}) = \sum_{|\phi_i\rangle} \left(\langle \phi_i | \rho^{SE} | \phi_i \rangle \right) = \rho^S,$$

where ρ^S is called the *reduced density operator*. The partial trace allows us to describe the dynamics of the system in term of the *system's* density operator by performing a measurement on the coupled system; this is as opposed to considering the coupled density operator on the tensor product space $\mathcal{H}^{SE}$. The density operator on this tensor product space can be quite elusive to rigorously quantify because there many degrees of freedom. However, even if we could easily find ρ^{SE}, it is unnecessary since we are only interested in the dynamics of the system itself, not of the environment. Thus, it is quite advantageous to describe the dynamics of our system without having to quantify ρ^{SE}. The ability to describe the system's evolution with just the system's density operator is where the Lindblad master equation comes in.

Consider the following diagram (wherein the horizontal arrows denote the appropriate time evolution and vertical arrows denote evaluation of density matrices):

$$\begin{array}{ccc} \rho^{SE}(t) & \xrightarrow{\hat{H}_{SE}} & \dot{\rho}^{SE}(t) = i[\hat{H}_{SE}, \rho^{SE}] \\ \Big\downarrow \mathrm{tr}_E & & \approx \Big\downarrow \mathrm{tr}_E \\ \rho^S(t) & \xrightarrow[\hat{\mathcal{L}}]{} & \dot{\rho}^S(t) = \hat{\mathcal{L}}(\rho^S(t)) \end{array}$$

where $\hat{H}_{SE}$ is the Hamiltonian describing the dynamics of the coupled system and environment. Note, while taking the partial trace of $\dot{\rho}^{SE}(t)$ gives us the *exact* time evolution of our system, the environment typically has an infinite dimensional Hilbert space. Therefore, we use the Lindblad master equation to give us the time evolution of our open quantum system.

The main point of the above discussion is that it is legitimate to describe the interaction between a system and its environment by considering the system's density operator and the Lindblad operators used in the Lindblad master equation. Now that we are familiar with the master equation, we will address a specific problem. We will then move onto the computational aspect of the thesis.

Spontaneous Emission of a Qubit Coupled to a Thermal Bath

For this example, we consider the relaxation and excitation of a qubit coupled to a bath. For such a system, we define an analog of the Pauli spin matrices:

$$\sigma_1 = |e\rangle\langle g| + |g\rangle\langle e|, \sigma_2 = -i|e\rangle\langle g| + i|g\rangle\langle e|, \sigma_3 = |e\rangle\langle e| - |g\rangle\langle g|$$

We similarly define:

$$\sigma_+ = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \sigma_- = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

For this example, we consider $\hat{H}_S = \frac{1}{2}\omega_0\sigma_3$, where ω_0 quantifies the energy of the qubit. Moreover, we will take the Lindblad operators to be $L_i \in \{\sigma_-, \sigma_+\}$. The dynamic caused by σ_+ corresponds to absorption of energy *into* the system *from* the environment, whereas the dynamic caused by σ_- correspond to emission *from* the system *into* the environment.

We then have:

$$\dot{\rho} = -\frac{i\omega_0}{2}[\sigma_3, \rho] + \gamma_r \left(\sigma_- \rho(t) \sigma_+ - \frac{1}{2}\{\sigma_+\sigma_-, \rho(t)\} \right) + \gamma_e \left(\sigma_+ \rho(t) \sigma_- - \frac{1}{2}\{\sigma_-\sigma_+, \rho(t)\} \right),$$

where γ_r corresponds to the rate of relaxation and γ_e corresponds to the rate of excitation. We then make the substitution:

$$\rho = \frac{1}{2}(\mathbb{1} + \langle \vec{\sigma}(t) \rangle \cdot \vec{\sigma}) = \begin{pmatrix} \frac{1}{2}(1 + \langle \sigma_3(t) \rangle & \langle \sigma_-(t) \rangle \\ \langle \sigma_+(t) \rangle & \frac{1}{2}(1 - \langle \sigma_3(t) \rangle) \end{pmatrix}.$$

After some matrix multiplication, we receive the system of coupled differential equations (where $\gamma_\Delta = \gamma_e - \gamma_r$ and $\gamma_\Sigma = \gamma_e + \gamma_r$):

$$\dot{\rho} = \begin{pmatrix} \frac{1}{2}\langle \dot{\sigma}_3(t) \rangle & \langle \dot{\sigma}_- \rangle \\ \langle \dot{\sigma}_+ \rangle & -\frac{1}{2}\langle \dot{\sigma}_3(t) \rangle \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} -\gamma_\Sigma \langle \sigma_3(t) \rangle + \gamma_\Delta & -\langle \sigma_-(t) \rangle (\gamma_\Sigma + i\omega_0) \\ -\langle \sigma_+(t) \rangle (\gamma_\Sigma + i\omega_0) & \gamma_\Sigma \langle \sigma_3(t) \rangle + \gamma_\Delta \end{pmatrix}$$

We thus have the following differential equations:

1. $\langle \dot{\sigma}_3(t) \rangle = -\gamma_\Sigma \langle \sigma_3(t) \rangle + \gamma_\delta$
2. $\langle \dot{\sigma}_-(t) \rangle = -\frac{1}{2}\langle \sigma_-(t) \rangle (\gamma_\Sigma + i\omega_0)$
3. $\langle \dot{\sigma}_+(t) \rangle = -\frac{1}{2}\langle \sigma_+(t) \rangle (\gamma_\Sigma + i\omega_0)$

Their solutions are given by:

1. $\langle \sigma_3(t) \rangle = c_1 e^{-t\gamma_\Sigma} + \frac{\gamma_\Delta}{\gamma_\Sigma}$
2. $\langle \sigma_-(t) \rangle = c_2 e^{-\frac{t}{2}(\gamma_\Sigma + i\omega_0)}$
3. $\langle \sigma_+(t) \rangle = c_3 e^{-\frac{t}{2}(\gamma_\Sigma + i\omega_0)}$

Of course, to determine c_1, c_2, c_3, we need the initial conditions $\langle \sigma_3(0) \rangle, \langle \sigma_-(0) \rangle$, and $\langle \sigma_+(0) \rangle$; these values are given by $\rho(0)$.

Now, what do we expect if we measure σ_3 many times and average the result (this is, of course, the quantity $\langle \sigma_3 \rangle$)? Intuitively speaking, we expect one of the two cases:

1. If the qubit's energy is *lower* than the thermal energy of the bath, we would expect that the long time behavior of the qubit would be to transition to the excited state
2. On the other hand, if the qubit's energy were *higher* than the bath's energy, we would expect that the long time behavior of the qubit would be to transition to the ground state.

Below are two exact solutions (the initial conditions and constants are given in the captions):

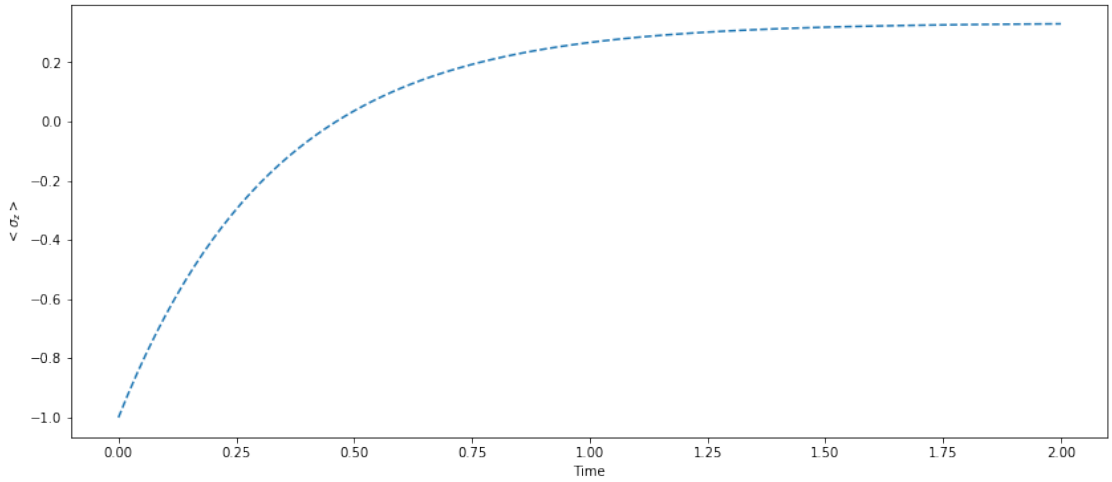


Figure 2: Exact solution to a qubit coupled to a bath; with initial condition $\langle \sigma_3(0) \rangle = -1$ and constant values $\gamma_e/\gamma_r = 2$; time is in units of γ_e^{-1}

If we increase γ_e, we expect that the expected value of σ_3 is higher. Indeed, we see:

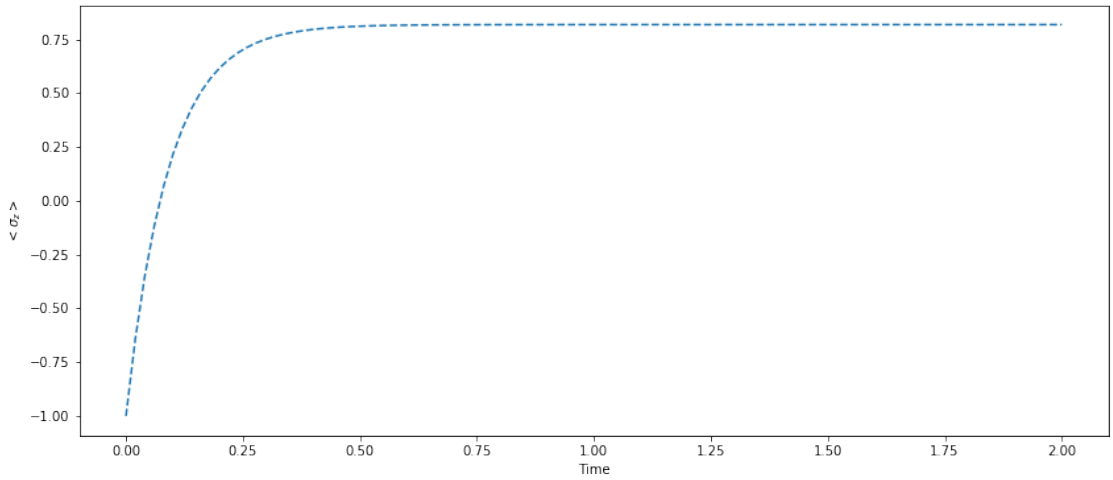


Figure 3: Exact solution to a qubit coupled to a bath; with initial condition $\langle \sigma_3(0) \rangle = -1$ and constant values $\gamma_e/\gamma_r = 10$; time is in units of γ_e^{-1}

Section 4.2.1 describes numerical solutions to

$$\langle \dot{\sigma}_3(t) \rangle = -\gamma_\Sigma \langle \sigma_3(t) \rangle + \gamma_\Delta$$

and compares it to the exact solutions. Note that the operators σ_- and σ_+ are not Hermitian, so they do not correspond to physical observables; therefore, we will not compare the exact and numerical solutions to $\langle \sigma_+(t) \rangle$ or $\langle \sigma_-(t) \rangle$.

3 Computation with GPUs

Now that we have discussed the physical concepts of interest, we move on to the computational aspect of this thesis. The primary goal of my project with Professor Koch is to efficiently solve the Lindblad master equation using a GPU. In this section, we will introduce the relevant information about a GPU that one needs in order to write code to execute a numerical scheme. In particular, we will describe how the processors on a GPU are arranged, how to write code for a GPU, the advantages and disadvantages of a GPU, and what algorithms are optimal for a GPU and why. This will culminate in the comparison of computation times of matrix multiplication and RK4 on a GPU versus on a CPU.

3.1 Introduction to GPUs

This introduction is dedicated to defining a parallelizable algorithm and giving a simple example: adding two n-tuples. This algorithm is very simple, thus it is a good example that we will use throughout Section 3 in order to understand how a GPU works. Moreover, we will explicitly see the advantage of using a GPU for this example.

List Addition: A Simple Parallelizable Algorithm

A parallelizable algorithm is an algorithm that can be implemented via parallel computation, i.e. the algorithm can be split into multiple pieces, each of which can be executed on separate processors. For example, adding two lists with n elements can be parallelized because adding the i^{th} elements of each list can be done independently of all the other elements.

Let us call the two lists $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$. If we want to compute $x+y$ (element-wise addition) on a CPU (without using parallel computation), one algorithm is:

1. Calculate $x_1 + y_1$
2. Calculate $x_2 + y_2$

   $\vdots$

n. Calculate $x_n + y_n$

If instead we use a GPU, then we will parallelize this algorithm. The algorithm would simply consist of the single step:

1. Calculate $x_i + y_i$, for each $i \in \{1, \ldots, n\}$, on the i^{th} processor of the GPU, simultaneously.

After each computation is done, the GPU collects the results from all the processors and reports the entire list. The figures below compare the computation time of a CPU versus a GPU to implement element-wise addition.

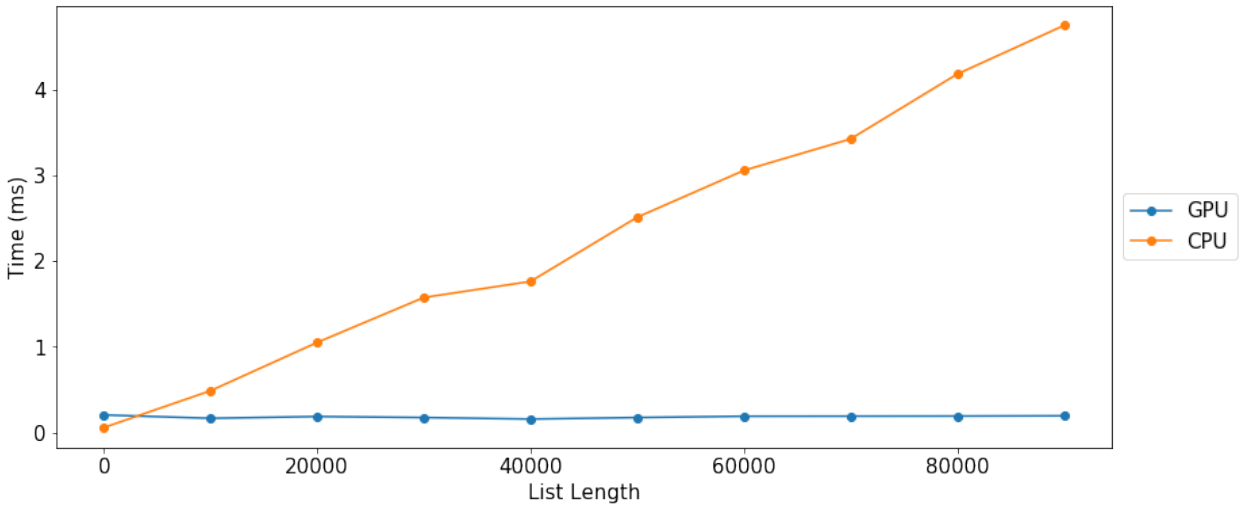


Figure 4: Computation time of element-wise list addition of single precision floating point real numbers on a CPU vs. GPU for lists of size 1 to 100001

Indeed, we see that the computation time for the GPU is approximately constant as the size of the lists increases, whereas the total computation time of the CPU gets larger as the list size increases. This is the power of parallel computation. However, we notice that for small lists, the CPU is faster. If we investigate this region, we see:

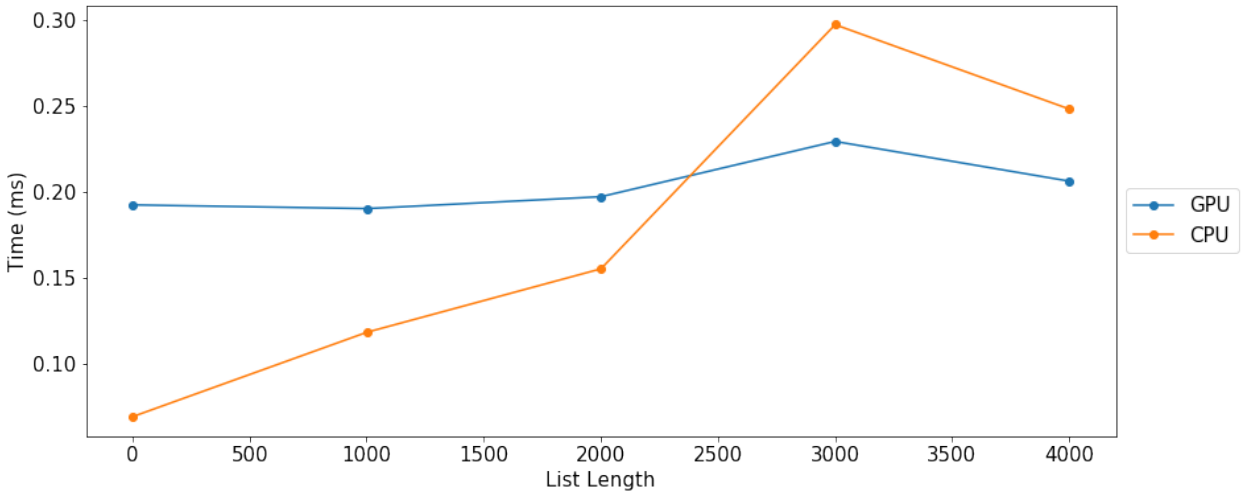


Figure 5: Computation time of element-wise list addition of single precision floating point real numbers on a CPU vs. GPU for lists of size 1 to 4001

As we can see above, the GPU is *slower* if our list size is less than ≈ 2400. The reason is that the processors on the GPU are not as powerful as standard CPU processors. Therefore, the GPU only becomes faster than the CPU if our dataset is large enough; in other words, we must be processing enough data that the total computation time of the CPU is sufficiently slow. In Section 3.3, we will see the analogous result that a GPU is advantageous over a CPU for matrix multiplication *only* when we are multiplying large matrices.

3.2 Threads, Blocks, and Grids

In this subsection, we will discuss how processors on a GPU are organized.

A *thread of execution*, typically referred to as a *thread*, is the smallest part of a processor that can execute a set of instructions. When we call the GPU to execute a task, the threads implement the algorithm and execute the computations. In our example of element-wise list addition, the threads are adding the individual elements.

Threads are organized within *blocks*. Blocks in the Tesla K40 contain 1024 threads. Therefore, to perform more than 1024 computations on our GPU simultaneously, we need to call multiple blocks. Each block is located within a *grid*, which contains 32 blocks. Thus, in order to call more than $32 \cdot 1024 = 32,768$ threads, we must call multiple grids.

Due to how the processors in the Tesla K40 are organized, computation times are optimal when threads are called in groups of 32 [14]. This requires special attention when writing functions for the GPU; this will be described in Section 3.3.1. We now move on to an important aspect of thread architecture: multidimensional threading.

Multidimensional Threading

In the case of element-wise list addition, we can think of our threads as arranged in a one dimensional line:

$$\{x_1, x_2, \ldots, x_n\}$$
$$\{y_1, y_2, \ldots, y_n\}$$
$$\{thread_1, thread_2, \ldots, thread_n\}$$

Where $thread_i$ adds x_i and y_i. This is known as *one dimensional threading*. One dimensional threading is clearly appropriate for list addition because lists are one dimensional objects; however, what if we are using a two dimensional object, such as a matrix?

If we want to add two n-by-n matrices, then one way to proceed is to think of an n-by-n matrix as a list of length n^2. We can thus use one dimensional threading and label each thread from 1 to n^2. Another way to proceed is to use *two dimensional threading*.

Suppose we are adding two matrices:

$$A = \begin{pmatrix} a_{(1,1)} & \ldots & a_{(1,n)} \\ a_{(n,1)} & \ldots & a_{(n,n)} \end{pmatrix}, B = \begin{pmatrix} b_{(1,1)} & \ldots & b_{(1,n)} \\ b_{(n,1)} & \ldots & b_{(n,n)} \end{pmatrix}$$

In the case of two dimensional threading, we think of the threads as arranged in a two dimensional array. The threads are labeled as follows:

$$\begin{pmatrix} thread_{(1,1)} & \ldots & thread_{(1,n)} \\ thread_{(n,1)} & \ldots & thread_{(n,n)} \end{pmatrix}$$

Thus, $thread_{(i,j)}$ computes $a_{(i,j)} + b_{(i,j)}$.

One can go further and arrange threads in a three-dimensional array, which might be useful for computations with rank-3 tensors. Since threads are organized within blocks, we specify the dimensionality of our threads when we initialize our blocks (we will see how this is done in the end of Section 3.3.2). We can similarly use multidimensional blocks by

specifying the size of our grids; however, we can only use one and two dimensional grids (three dimensional grids are not supported for reasons of GPU hardware, which is beyond the scope of this thesis).

In order to actually implement multidimensional threading, we need to introduce some CUDA keywords. This is discussed in Section 3.3.1 and 3.3.3.

Below is a visualization of two dimensional threading in which we use two dimensional threads and two dimensional grids [15]:

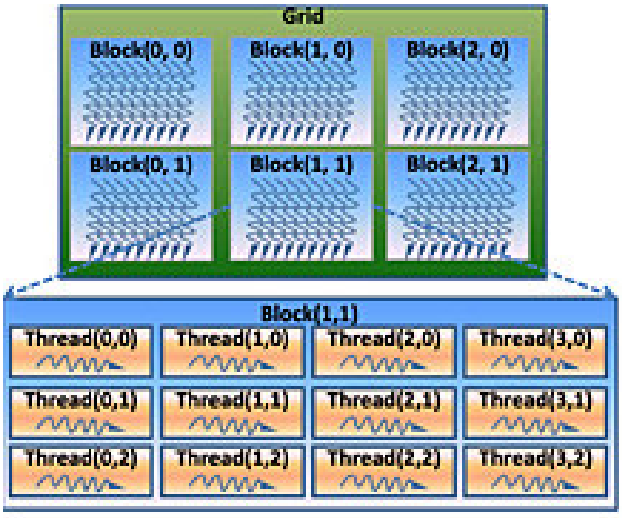


Figure 6: Organization of two dimensional blocks and two dimensional grids

An Analogy for the Thread, Block, and Grid Hierarchy

To help make sense of how threads, blocks, and grids are arranged, we can construct an analogy:

Suppose there is a warehouse, full of 1024 workers, who are numbered from 0 to 1023; here, the warehouse represents a block and the workers represent the threads within the block. If we are adding two lists, with no more than 1024 elements, then we can assign worker number i the i^{th} element of each list. Worker i will then make the computation at the same time as all the other workers.

If we are adding lists of more than 1024 elements, we will need *more* than 1024 words; therefore, we will use multiple warehouses. Suppose these warehouses are organized in regions of 32; here, the region represents a grid. Then once we use all 32 warehouses in a region, we will need to use another region, and so on and so forth. Suppose that the regions are contained within different countries. Therefore, requiring multiple grids is analogous to requiring multiple countries.

The fact that the Tesla K40 performs optimally when the threads are in groups of 32 is like saying that our workers are happiest when they work in packs of 32. Happier workers means more efficient production, so indeed we should arrange them in such a way.

This notion of multi-dimensional threads is analogous to looking at a warehouse from a birds eye view and placing the workers in a rectangular array of desks; we are simply assigning the worker at the desk in position (j, i) the corresponding element of a two-dimensional

dataset. Again, this is useful in matrix multiplication: suppose we wish to compute the product AB (where A and B are appropriately sized matrices). Worker (j, i) calculates the dot product of the j^{th} row of matrix A and i^{th} column of matrix B. Each worker computes one element in the resulting product at the same time as all other workers.

3.2.1 Streams

Whenever we want to execute a computation on a GPU, we must call a function. This function is added to a queue. The GPU then executes functions in this queue one after the other. A *stream* is a CUDA object that contains information about one queue and the location in the GPU where the functions in the queue are being executed. We can, in fact, open *multiple* streams on the GPU, which allows for another layer of parallel computation. Heuristically, multiple streams work in the following way:

Say we have 100 pairs of lists that we want to add. We have two options:

1. Use *one* stream and call a list addition function 100 times; this function will then be executed one after another on the GPU
2. Use 100 streams on the GPU and each stream will add one pair of lists

Knowledge of streams is vital to recording computation time on a GPU. This is described in Appendix A.

3.3 Writing Code for a GPU

Originally, programmers wrote code for GPUs in CUDA, a parallel computing platform created by NVIDIA [16]. CUDA is nearly identical to C, except for a number of keywords that allow the programmer to control specific aspects of threads. Today, there exists a CUDA wrapper for python, which is called PyCUDA.

In this section, we will describe the structure of a PyCUDA code. In particular, we will see how to write and use custom functions to be implemented on a GPU, which are called *kernels*. For many algorithms, it is unnecessary to write custom functions. For example, if we are programming in python for a CPU, then we can certainly write a custom function that multiplies two matrices; however, it is a much better decision to use a function from the Basic Linear Algebra Subprograms (BLAS) library because BLAS's functions are professionally optimized. Similarly, we can write a kernel to multiply two matrices, but there exist functions in cuBLAS (which stands for CUDA BLAS) that have functions that use optimized algorithms. In Section 3.3.3, we will see an example of a kernel for matrix multiplication and we will see that it is slower than the analogous cuBLAS function.

3.3.1 Kernels

Kernels are, by definition, global functions that are void valued and are written in CUDA.

It is easiest to describe how a kernel works by providing an example and explaining each aspect of the code. Consider the following kernel, which performs element-wise addition of two lists of 1000 double precision real numbers:

```
__global__ void list_kernel(double first_list[1000], double second_list[1000],
    double result_list[1000])
{
    int thread_num=threadIdx.x+blockIdx.x*blockDim.x;
    if(thread_num<1000)
    {
        result_list[thread_num]=first_list[thread_num]+second_list[thread_num];
    }
    return;
}
```

As we discussed in Section 3.2, a thread is assigned an integer when it is initialized. It is this first line:

`int thread_num=threadIdx.x+blockIdx.x*blockDim.x;`

that gives us the index that a thread is assigned.

The `if` statement takes care of the fact that there are more launched threads than elements in our datasets. In this example, we are adding lists of 1000 elements, so we are launching 1024 threads. Therefore, we need to make sure that we are not referencing elements in the lists that do not exist; without this `if` statement, we would reference elements such as `first_thread[1020]`.

Finally, the interior of the if statement is what implements the algorithm, which does not require explanation.

Notice that the arguments to the kernel function consist of:

1. A pointer to the first list to be added, entitled `first_list`.
2. A pointer to the second list to be added, entitled `second_list`
3. A pointer to the list that will contain the results, entitled `result_list`

Since kernels are required to be void functions, the easiest way to harvest data from a kernel is to use a pointer.

3.3.2 The Python Part of a PyCUDA Script

There are four main tasks that need to be done in the python part of the code:

1. Initialize the GPU; this includes copying data from the CPU to the GPU, calculating how many threads are needed, etc.
2. Compile kernels via the function `SourceModule` (from the `pycuda.compiler` library)
3. Call the kernels to execute a computation
4. Copy the data back to the CPU

How we copy data to the GPU depends on whether we are using custom kernels or pre-existing libraries. If we are using custom kernels, then we need to use the function `to_device` in the `pycuda.driver` library. On the other hand, if we are using pre-existing libraries, such as CUBLAS, then we need to copy the data as a "gpuarray", which requires the `to_gpu` function in the `pycuda.gpuarray` library. Both of these methods require that the data be in the form of a numpy array on the CPU.

In order to use a kernel, we must first compile it. This requires that the kernel is written as a PyCUDA string: we surround the kernel with *three* quotation marks on each side. We then feed the string to the `SourceModule` function and retrieve the function with the `get_function` attribute of a compiled kernel.

The kernel in the code below is generalized from the previous kernel. Instead of adding two lists of 1000 double precision real numbers, we use string substitution to replace the data type and length of each array at runtime. This allows us to add lists of arbitrary datatype and size, without having to hardcode the values into the kernel. We now give a piece of code that compiles and harvests this kernel:

```
list_length = 1000
data_type = "double"
kernel = """__global__ void list_kernel(%(T)s first_list[%(N)s], %(T)s
    second_list[%(N)s], %(T)s result_list[%(N)s])
{
    int thread_num=threadIdx.x+blockIdx.x*blockDim.x;
    if(thread_num<%(N)s)
    {
        result_list[thread_num]=first_list[thread_num]+second_list[thread_num];
    }
    return;
}"""
dictionary = {"N": list_length, "T" : data_type}
compiled_kernel = SourceModule(kernel%dictionary)
add_lists = compiled_kernel.get_function("list_kernel")
```

After running the script above, we can use `add_lists` as a function in our python script. Of course, its inputs must be pointers to the memory address on the GPU where the appropriate datasets are stored. In order to specify how many threads, blocks, and grids we are using (and their dimensionality) `add_lists` requires *two more* inputs: the number of blocks and the number of grids used. These are given as tuples:

1. The block is given as a three-tuple:
   (number of threads in the x-dimension, number of threads in the y-dimension, number of threads in the z-dimension)
2. The grid is given as a two-tuple:
   (number of blocks in the x-dimension of the grid, number of blocks in the y-dimension)

Let the block tuple be called `the_block` and the grid tuple be called `the_grid`. We call `add_lists` the following way:

```
add_lists(first_list_pointer, second_list_pointer, result_list_pointer, block =
    the_block, grid = the_grid)
```

After running the kernel as above, we copy the data in the address `result_list_pointer` back to the CPU; this is done with the function `from_device` from the `pycuda.driver` library.

3.3.3 Matrix Multiplication

In this section, we will present a kernel that multiplies two matrices on a GPU with two dimensional threading. First, let us note why a GPU is advantageous over a CPU for matrix multiplication:

If we are multiplying two n-by-n matrices, then we must compute n^2 dot products. A CPU computes each dot product one after another. On the other hand, a GPU can launch n^2 threads and each thread computes one dot product. Therefore, a GPU computes all n^2 entires in the product simultaneously.

A basic algorithm for non-parallelized matrix multiplication is:

```
for row in range(0,n):
    for col in range(0,n):
        C[row][col] = numpy.dot(A[row][:], B[:][col])
```

This algorithm is hereafter referred to as *crude multiplication*. A kernel that implements this algorithm is (note we are still assuming the use of string substitution):

```
__global__ void mult(%(T)s first_matrix[%(N)s][%(N)s], %(T)s
    second_matrix[%(N)s][%(N)s], %(T)s result_matrix[%(N)s][%(N)s])
{
    int index_x=threadIdx.x+blockIdx.x*blockDim.x;
    int index_y=threadIdx.y+blockIdx.y*blockDim.y;

    if (index_x < %(N)s & index_y < %(N)s)
    {
        result_matrix[index_y][index_x] = 0.0;
        for (int ii = 0; ii < %(N)s; ii++)
        {
            result_matrix[index_y][index_x] = result_matrix[index_y][index_x] +
                first_matrix[index_y][ii]*second_matrix[ii][index_x];
        }
    }
    return;
}
```

Note that because we are using multidimensional threading, we use `threadIdx.y+blockIdx.y*blockDim.y` as well as `threadIdx.x+blockIdx.x*blockDim.x`.

We will now compare the computation times of double precision real valued matrix-matrix multiplication for the following:

1. BLAS multiplication of dense arrays on a CPU
2. Compressed Sparse Row multiplication of compressed sparse row arrays on a CPU
3. Crude multiplication of dense arrays via custom kernels on a GPU
4. CUBLAS multiplication of dense arrays of gpuarrays on a GPU

BLAS and CUBLAS use algorithms that are more sophisticated than the crude multiplication described above. More information on matrix multiplication algorithms can be found

in [17].

We now give the data of computation time for computations described above (note for the computations reported below, all matrices except for Compressed Sparse Row are dense):

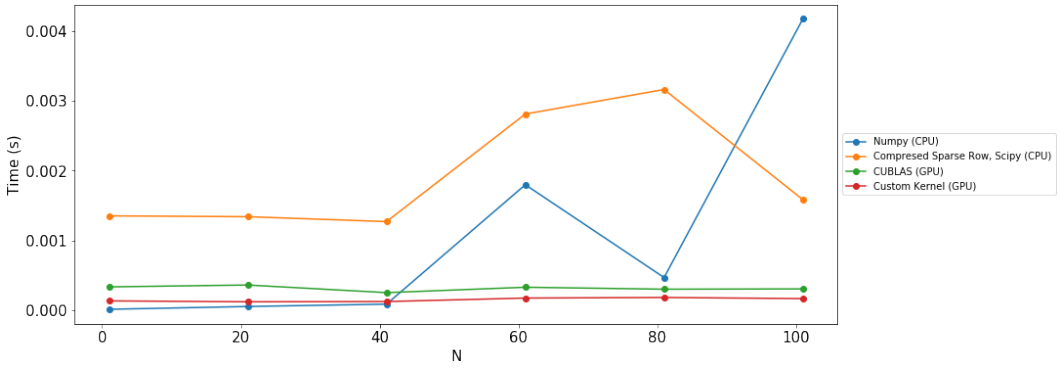


Figure 7: Computation time of matrix-matrix multiplication consisting of double precision real numbers, 1-by-1 matrices up to 101-by-101 matrices

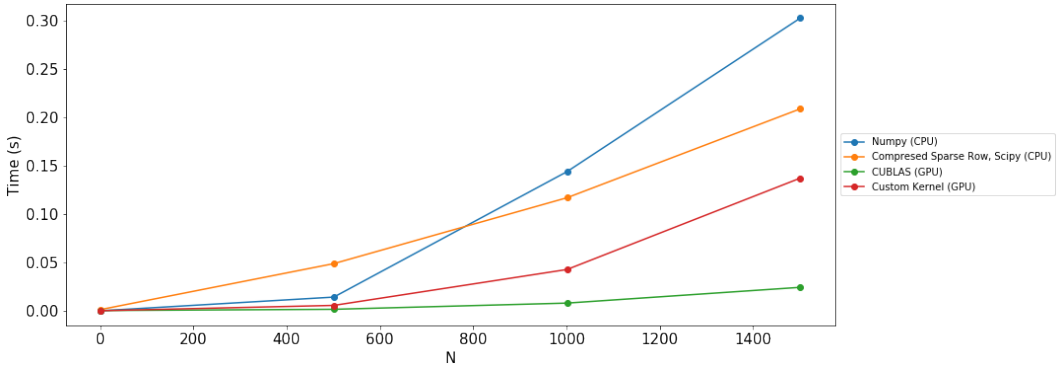


Figure 8: Computation time of matrix-matrix multiplication consisting of double precision real numbers, 1-by-1 matrices up to 1501-by-1501 matrices

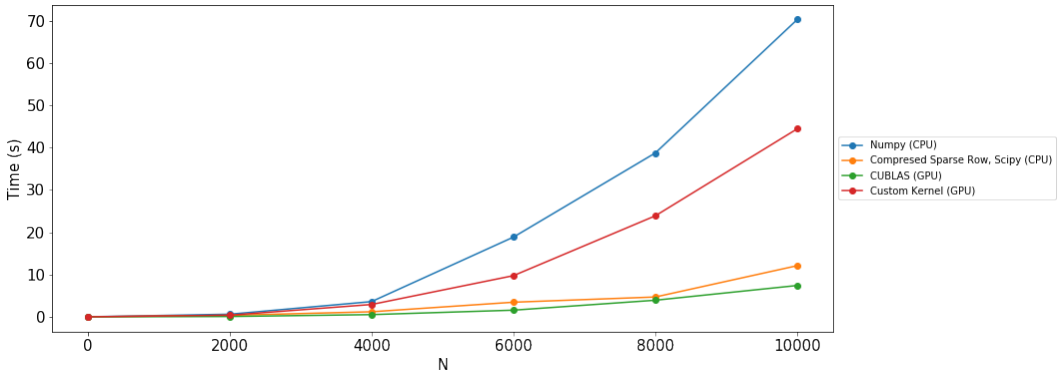


Figure 9: Computation time of matrix-matrix multiplication consisting of double precision real numbers, 1-by-1 matrices up to 10,001-by-10,001 matrices

Indeed we see that crude matrix multiplication on a GPU is highly preferable over numpy's matrix multiplication once we consider n-by-n matrices with $n \gtrapprox 40$. On the other hand, our custom kernel implementation of crude matrix multiplication is slower than Scipy's compressed sparse row matrix multiplication and CUBLAS's matrix multiplication. The clear winner here is CUBLAS's matrix multiplication; thus, for the remainder of this thesis, all computations are done with CUBLAS.

3.4 RK4 on a GPU

In this section, we will consider an implementation of RK4 on a GPU, using CUBLAS matrix multiplication, in order to evolve a wavefunction in the fluxonium potential; moreover, we will compare the computation time of RK4 on a GPU to that of a CPU.

3.4.1 Time Evolution of Fluxonium

We now solve the Schrödinger Equation with the fluxonium Hamiltonian. In particular, we will observe quantum tunneling between the first and second wells on the left and right (referring back to Figure 1). This will give us an explicit visualization of the degeneracy of the states in the left and right well.

First, we make a brief remark pertaining to numerical solutions of partial differential equations, such as the Schrödinger equation. Recall that the Schrödinger equation is given by:

$$i\partial_t \left|\psi(t)\right\rangle = \hat{H} \left|\psi(t)\right\rangle .$$

In order to solve a partial differential equation with RK4, we must discretize $|\psi(x,t)\rangle$ in space; suppose we spatially discretize $|\psi(x,t)\rangle$ to N points. Thus, we are essentially evolving $|\psi(t)\rangle$ as a vector of length N. We expect that the GPU will become faster than the CPU when we increase N to be large enough; on the other hand, for small N we expect that the overhead of the GPU, and the "weakness" of threads, makes the CPU advantageous over the GPU.

Below are a few time evolutions of fluxonium; in particular, we see the aforementioned quantum tunneling effect:

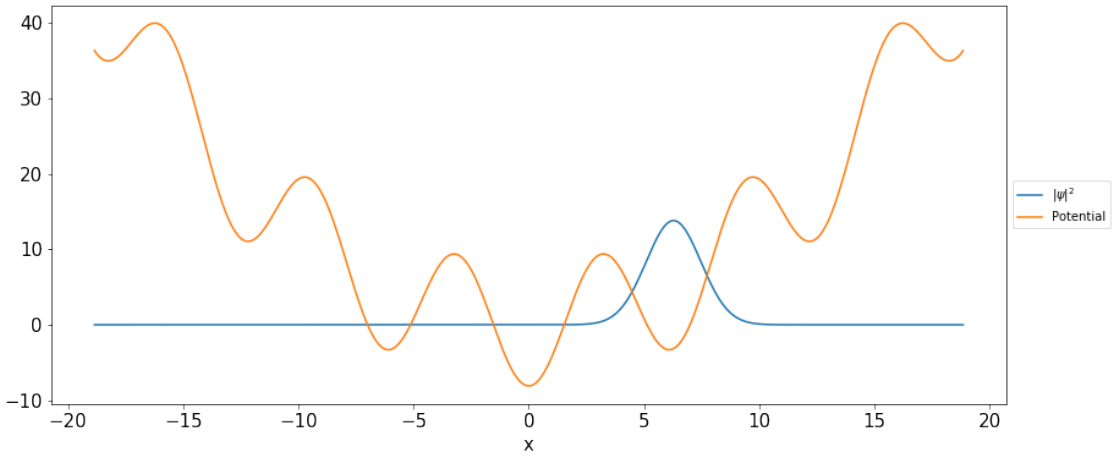


Figure 10: Modulus of a particle's wavefunction in the fluxonium potential at $t = 0$ nanoseconds; initial state is in the right well, with $E_c = 2.0$

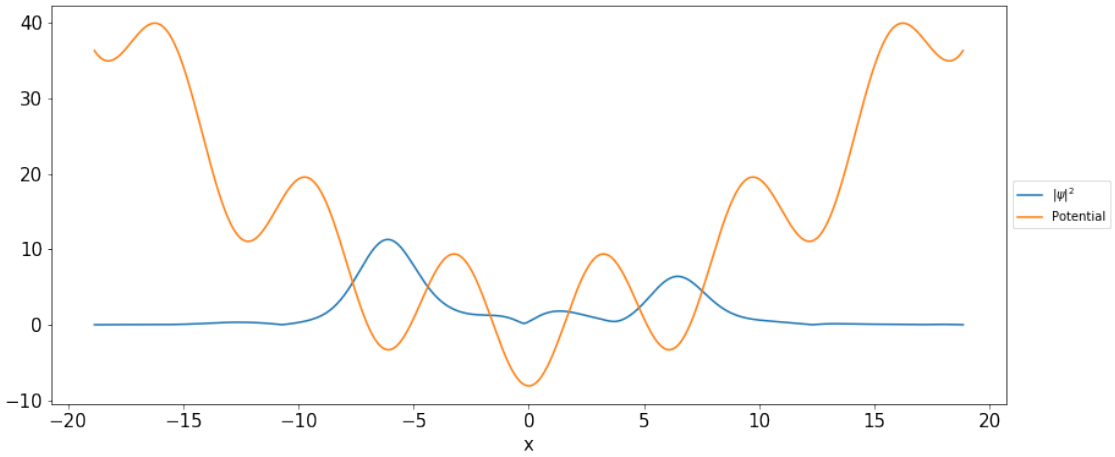


Figure 11: Modulus of a particle's wavefunction in the fluxonium potential at $t = 200$ nanoseconds; initial state is in the right well, with $E_c = 2.0$

The figure below shows the speed-up that a GPU RK4 solver gives, when compared to a CPU RK4 solver:

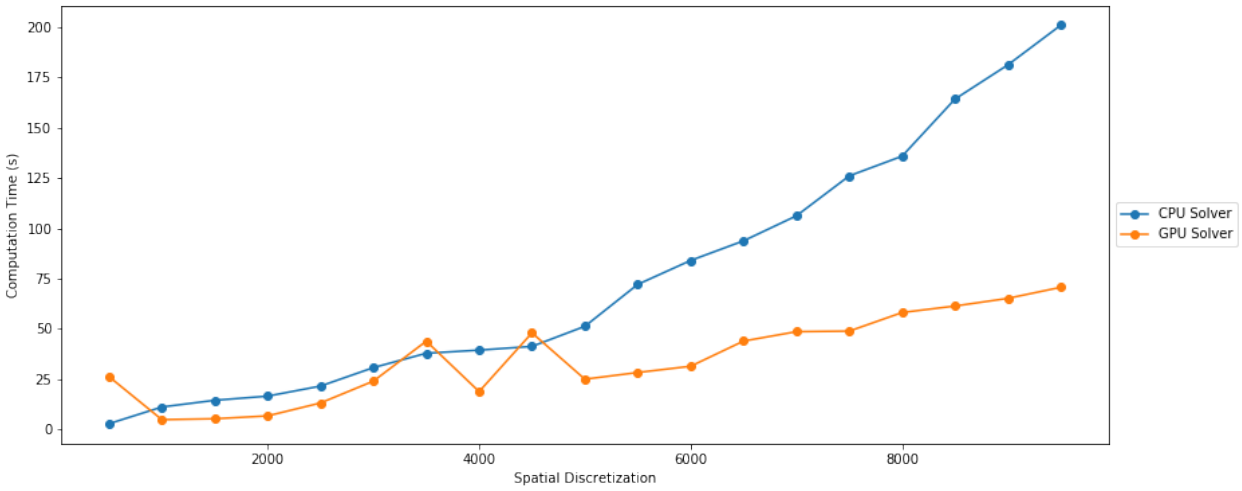


Figure 12: GPU vs. CPU Computation Time for Solving the Schrödinger equation to 0.1 units of time, timestep of 0.0001; spatial discretization ranges from 501 points to 10001 points

Indeed, we see that the GPU only becomes advantageous over the CPU when we discretize space to about 700 points or more, with the exception of a few hiccups, at $\approx 3500, 4500$.

4 Solving the Lindblad Master Equation with RK4

In this part, we finally consider numerical solutions to the Lindblad master equation via the RK4 scheme. In particular, we will solve the example of a qubit coupled to a thermal bath numerically, as well as new problems that of coupled qubits. In the case of large systems of coupled qubits, we will compare the computation time on the GPU to an identical solver on the CPU.

4.1 Explanation of the Final Code

In this section, we will describe each function in the final code, which is included in Appendix B. This section serves as a guide to the code to assist any future users.

4.1.1 Brief Description of Each Function

Note that the order of appearance in this section is *not* the order in which each function is called. The order in which they are called is given in the next section, which describes the complete pseudo-code. For the following, `current time` refers to the time that the simulation has reached.

class GPU_mesolve_internals

Inputs:

1. The initial timestep
2. The Hamiltonian
3. The initial density matrix
4. The observables
5. The Lindblad operators
6. The adaptive timestep tolerance
7. The coupling constants

Purpose:

This class transfers all inputs to the GPU. The advantage of using this class is that we do not need to pass each piece of data between the individual functions, rather we need only pass the class handle.

There is one item in this class whose purpose is not immediately obvious: `self.const_lindblad_term`. Note that when we compute the Lindblad terms in the master equation,

$$\sum_j \gamma_j (\hat{L}_j \rho \hat{L}_j^\dagger - \frac{1}{2}\{\hat{L}_j^\dagger \hat{L}_j, \rho\}),$$

we must compute 6 matrix multiplications for each Lindblad operator. In particular, the density matrix entry is changing for each k_1, k_2, k_3, and k_4 in the RK4 scheme, so we cannot compute any products including the density operator ahead of time. However, the $\hat{L}_j^\dagger \hat{L}_j$ term

does not change at *any* point of the computation. Therefore, `self.const_lindblad_term` stores the product $\hat{L}_j^\dagger \hat{L}_j$. This results in having to compute 4 matrix multiplications instead of 6 each time we need to compute these Lindblad terms.

runrk4_mesolve

Inputs:

1. The initial density matrix
2. List of times for which the user wants output for the expectation values
3. The Hamiltonian
4. The Lindblad operators
5. The observables
6. The coupling constants
7. The option to do adaptive timestepping (0 if adaptive timestep *is to not be done*, in which case the default timestep is 0.001, and 1 if adaptive timestep is to be done). This option is 0 by default.
8. The error tolerance for adaptive timestep, which is 0.1 by default.

Purpose:

This is the function that begins the numerical simulation. This function's job is to initialize the `GPU_mesolve_internals` class and pass it to the function `runrk4_mesolve_core`, which will be described next. After calling `runrk4_mesolve_core`, the function `exp_vals_calc` is called.

It should be noted that the list of times for which the user requires expectation values of the observables is referred to as the `tlist`. Within `runrk4_mesolve`, there is a `for` loop that goes through each adjacent pair of elements in `tlist`. In particular, suppose that

$$\texttt{tlist} = \{t_0, t_1, \ldots, t_n\}$$

then there is a `for` loop that gives the function `runrk4_mesolvE_core` the `GPU_mesolve_internals` class, as well as the pair t_i, t_{i+1}.

runrk4_mesolve_core

Inputs:

1. The handle of the class `GPU_mesolve_internals`
2. The pair t_i, t_{i+1} from `tlist`

Purpose:

This function calls the RK4 stepper function and evolves the density matrix from t_i to t_{i+1}. If the adaptive timestep switch is 1, meaning an adaptive timestep is used, then an adaptive stepper is called rather than a standard stepper.

There are many technical details to this function, which ensure that the simulation is executed properly; in particular, there are logical statements that ensure that the timestep is never too large such that the density matrix is evolved past t_{i+1}.

rk4_gpu_stepper

Inputs:

1. A `GPU_mesolve_internals` object
2. A binary variable `half_or_full_step`

Purpose:

Simply put, this function executes a single RK4 step. The variable `half_or_full_step` serves the following purpose: if 0 is given, then a half step is executed, and if 1 is given, then a full step is executed. This is used *only if* an adaptive timestep is being used (the adaptive timestep requires two half steps and one full step in order to estimate error). Within the `GPU_mesolve_internals` class, there are appropriate variables that allow for this half versus full step to be executed properly without mixing up the full step density matrix with the half step density matrix.

In order for this function to be as simple and concise as possible, the individual terms in the Lindblad master equation are evaluated in the two functions that follow: `von_Neumann` and `lindblad`.

von_Neumann

Input:

1. Two matrices, A and B

Purpose:

This function returns the value $-i[A, B]$. Note, this is the term given in the von_Neumann equation:

$$\dot{\rho} = -i[\hat{H}, \rho].$$

lindblad

Input:

1. The handle for the class `GPU_mesolve_internals`
2. An arbitrary matrix; this is usually the density matrix, but it is $\rho + \frac{1}{2}k_1$, $\rho + \frac{1}{2}k_2$, or $\rho + k_3$ if we are computing k_2, k_3, or k_4.

Purpose:

This function computes the term in the Lindblad master equation that is *not* the von Neumann term; namely:

$$\sum_j \gamma_j (\hat{L}_j \rho \hat{L}_j^\dagger - \frac{1}{2}\{\hat{L}_j^\dagger \hat{L}_j, \rho\}).$$

exp_vals_calc

Input:

1. A `GPU_mesolve_internals` object

Purpose:

This function computes the trace of the matrix product $\rho \hat{E}_i$ where $\hat{E}_i$ are the observables stored in `GPU_mesolve_internals`.

adaptive_stepper

Input:

1. A `GPU_mesolve_internals` object
2. The current time that the solver has reached
3. The time t_{i+1}

Purpose:

This function first saves the current density matrix in a storage variable, `rho_storage`. Then, the function executes one full step and two half steps, call these `rho_full` and `rho_half`. The error is then estimated to be the maximum element in the array `rho_full - rho_half`. Of course, this "maximum" is not well defined since these contain complex numbers and $\mathbb{C}$ is not well-ordered. Therefore, by "maximum element", we mean the element with maximum complex modulus.

We then set the new timestep to be the minimum value between $t_{i+1}-$ the current time and $dt * \frac{tolerance}{error}$.

Finally, `rho_full` is reset as the initial density matrix, `rho_storage` and a full timestep is executed with the new timestep.

4.1.2 Total Pseudo-Code

We now give the entire pseudo-code:

1. Call `runrk4_mesolve`, within which we:
   (a) Initialize `scikit.cuda`, which oversees matrix multiplication
   (b) If adaptive timestep is turned on, set dt to be $\frac{1}{2}(t_1 - t_0)$ (where t_i are elements of the `tlist`), otherwise set dt to be 0.001
   (c) Initialize the `GPU_mesolve_internals` class
   (d) Run a for loop, which goes through pairs $(t_0, t_1), (t_1, t_2), \ldots (t_i, t_{i+1}), (t_{i+1}, t_{i+2}), \ldots, (t_{n-1}, t_n)$. In this for loop, call `runrk4_mesolve_core` by giving it the `GPU_mesolve_internals` class and each pair (t_i, t_{i+1}).
2. In `runrk4_mesolve_core`, call the RK4 stepper to solve from t_i to t_{i+1}. Before calling the stepper, it is necessary to check that $dt < current\ time - t_{i+1}$, so that the stepper does not solve for a time greater than t_{i+1}. In particular, call the adaptive stepper instead of the standard RK4 stepper if the adaptive timestep is activated
3. Before going to the next pair (i.e. (t_{i+1}, t_{i+2})), call the `exp_vals_calc` function to harvest the expectation values requested by the user. These expectation values can be saved instead of the entire ρ, which is highly advantageous in terms of memory, since ρ is an N-by-N matrix (where N is the number of qubits being modeled).

4.1.3 Explanation of the (Dis)advantage of an Adaptive Timestep

Note that in the function `adaptive_stepper`, we set dt to be the minimum value between ($t_{i+1}-$ the current time) and ($dt * \frac{tolerance}{error}$). Thus, if $t_{i+1} - t_i$ is small, then the maximum

value of dt is small. Therefore, an adaptive stepper is advantageous when the cardinality of tlist is small.

4.2 Numerical Solutions to the Lindblad Master Equation

We can now simulate open quantum systems by numerically solving the Lindblad master equation with RK4! First, we test the RK4 code given in Appendix B with the problem of a qubit coupled to a bath. We then look at examples of systems of coupled qubits.

Note, this solver uses CUBLAS (via the `skcuda.linalg` library); therefore, the solver is hereafter referred to as the "CUBLAS Solver."

4.2.1 Qubit Coupled to a Bath

Recall the problem of a qubit coupled to a thermal bath from the end of Section 2.2.2. In particular, the numerical solver is given by the adaptive CUBLAS solver. We see that, for both sets of constants, the numerical and exact solution are visually identical:

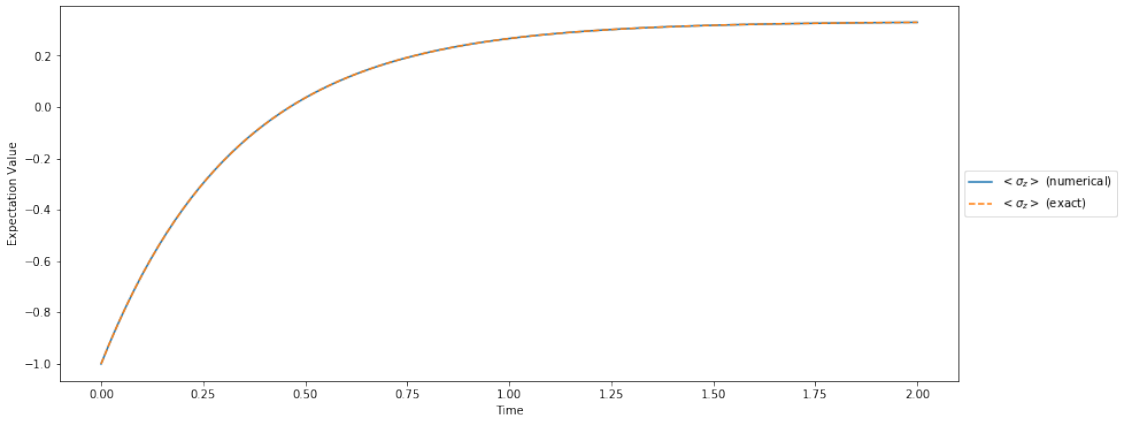


Figure 13: Exact solution to a qubit coupled to a bath; with initial condition $\langle \sigma_3(0) \rangle = -1$ and constant values $\gamma_e/\gamma_r = 2$

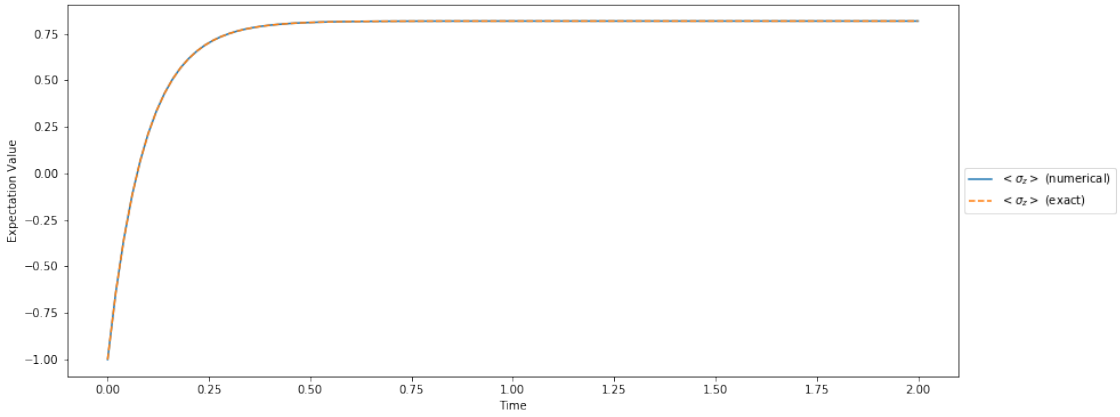


Figure 14: Exact solution to a qubit coupled to a bath; with initial condition $\langle \sigma_3(0) \rangle = -1$ and constant values $\gamma_e/\gamma_r = 10$

4.2.2 Systems of Coupled Qubits

For this example, we consider a system of coupled qubits. Suppose we have n qubits, each with a Hamiltonian of $\frac{1}{2}\omega\sigma_z$; moreover, suppose that qubit i is coupled to qubit $i+1$ with $\hat{\sigma}_x$ and coupling strength $g_{i,i+1}$. Therefore, the system of the coupled qubits has the Hamiltonian (where the superscript denotes the qubit number):

$$\hat{H} = \sum_n \frac{1}{2}\omega\sigma_z^n + \sum_n g_{n,n+1}\sigma_x^n\sigma_x^{n+1}$$

$$= \frac{1}{2}\omega\sigma_z^1 \otimes \mathbb{1}^2 \otimes \cdots \otimes \mathbb{1}^n + \mathbb{1}^1 \otimes \frac{1}{2}\omega\sigma_z^2 \otimes \mathbb{1}^3 \otimes \cdots \otimes \mathbb{1}^n + \cdots + \mathbb{1}^1 \otimes \cdots \otimes \frac{1}{2}\omega\sigma_z^n$$

$$+ g_{1,2}\sigma_x^1 \otimes \sigma_x^2 \otimes \mathbb{1}^3 \otimes \cdots \otimes \mathbb{1}^n + \cdots + g_{n-1,n}\mathbb{1}^1 \otimes \cdots \otimes \mathbb{1}^{n-2} \otimes (\sigma_x)^{n-1} \otimes (\sigma_x)^n$$

where $\omega = 10\pi$ and $g_{i,i+1} = \pi$. We take our Lindblad operators to be σ_- *for each* qubit; in other words, our Lindblad operators are of the form:

$$L_i \in \{\sigma_-^1 \otimes \mathbb{1}^2 \otimes \cdots \otimes \mathbb{1}^n, \mathbb{1}^1 \otimes \sigma_-^2 \otimes \mathbb{1}^3 \otimes \cdots \otimes \mathbb{1}^n, \ldots, \mathbb{1}^1 \otimes \mathbb{1}^2 \otimes \cdots \otimes \mathbb{1}^{n-1} \otimes \sigma_-^n\},$$

Our observables are σ_z, in an analogous tensor product.

Moreover, we take our initial state to be the *first* qubit in the excited state and all other qubits in the ground state. For systems of various numbers of qubits, we observe the following dynamics:

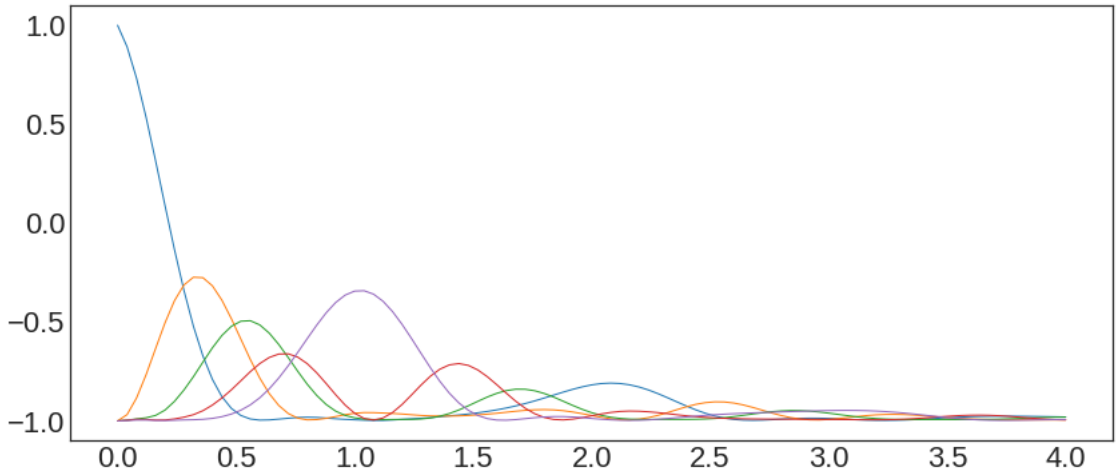


Figure 15: Numerical solutions to the Lindblad master equation for 5 coupled qubits; the first qubit starts in the excited state and all other qubits start in the ground state; up to 10 nanoseconds; Lindblad and observables are as above

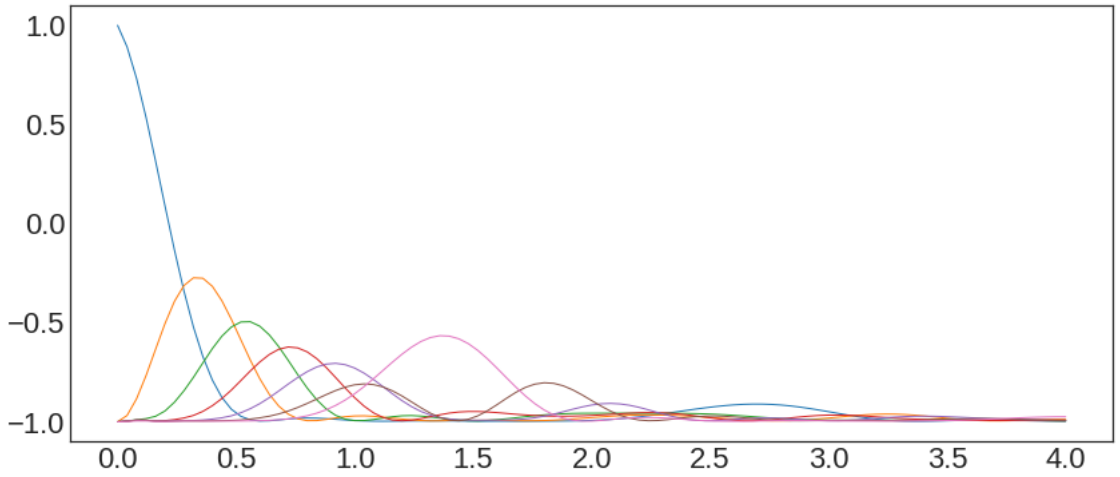


Figure 16: Numerical solutions to the Lindblad master equation for 7 coupled qubits; the first qubit starts in the excited state and all other qubits start in the ground state; up to 10 nanoseconds; Lindblad and observables are as above

Note, if we have no Lindblad operators (or, equivalently, set $L_i = 0$), then there is no dissipation:

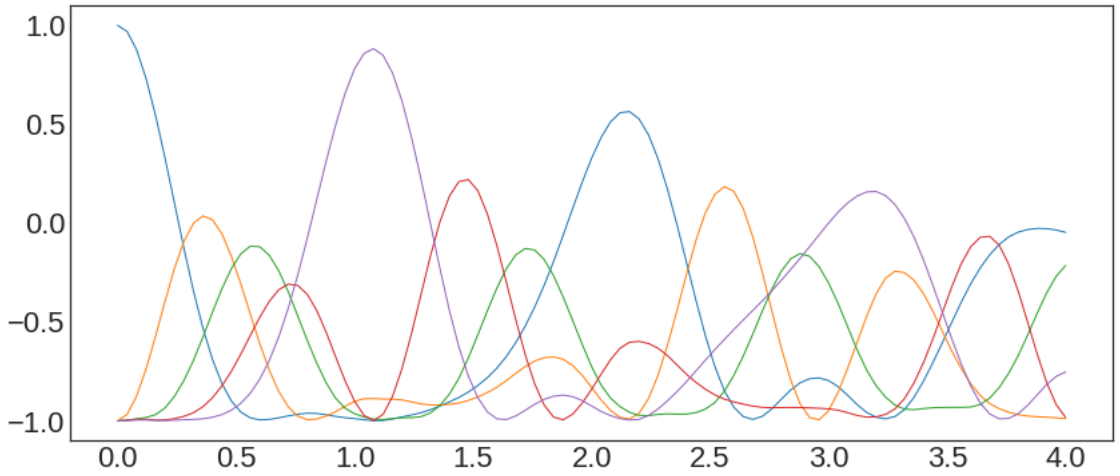


Figure 17: Numerical solutions to the Lindblad master equation for 5 coupled qubits; the first qubit starts in the excited state and all other qubits start in the ground state; up to 10 nanoseconds; observables are as above; no Lindblad operators

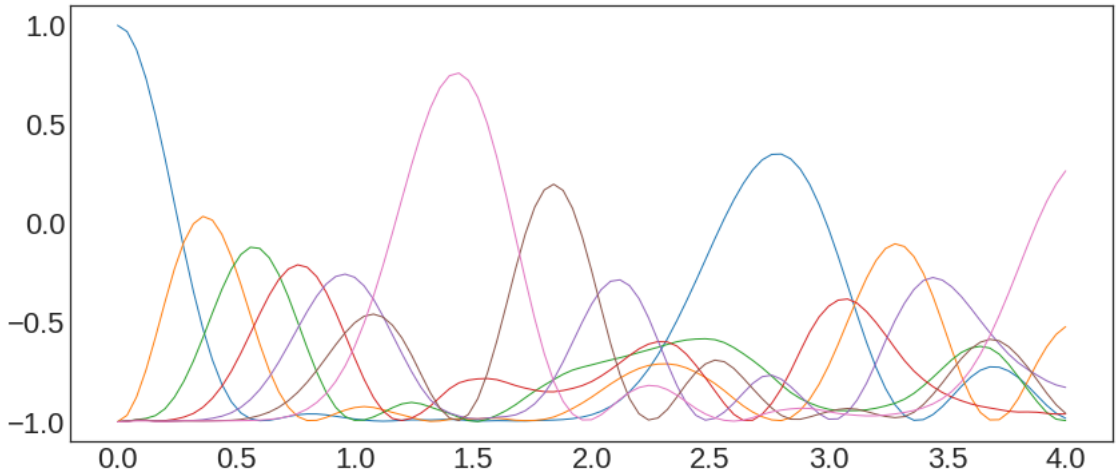


Figure 18: Numerical solutions to the Lindblad master equation for 7 coupled qubits; the first qubit starts in the excited state and all other qubits start in the ground state; up to 10 nanoseconds; observables are as above; no Lindblad operators

4.2.2.1 GPU versus CPU Computation Time of RK4 for the Lindblad Master Equation

As always, we expect that the GPU becomes advantageous when we are considering a *large* enough system of qubits. The size of our Hilbert space is given by 2^N, where N is the number of qubits in our system.

First, we compare a CPU master equation solver and a GPU master equation solver:

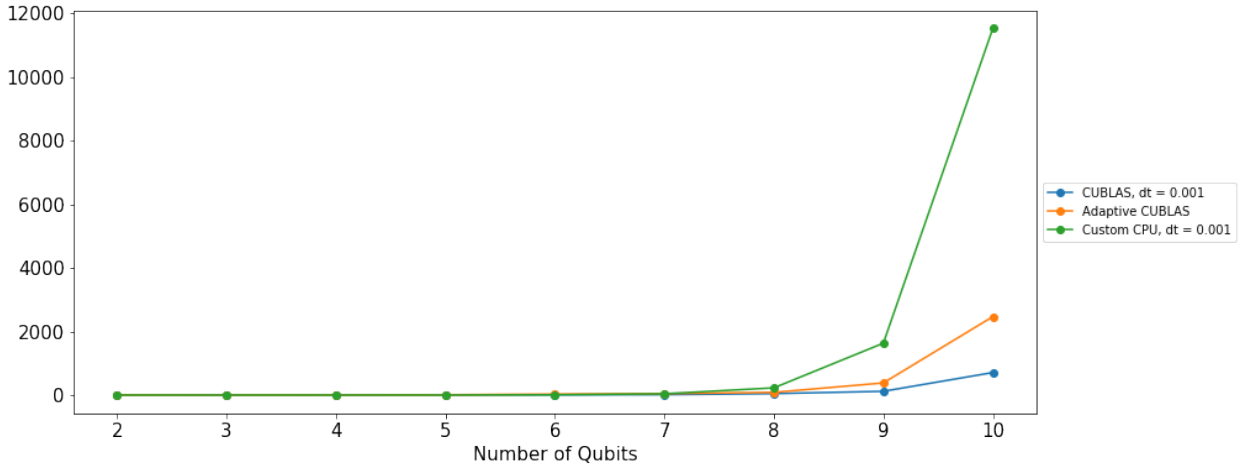


Figure 19: RK4 Lindblad master equation computation time on a CPU versus on a GPU; solving systems from 2 qubits to 10 qubits up until 0.5 units of time; tlist size of 2

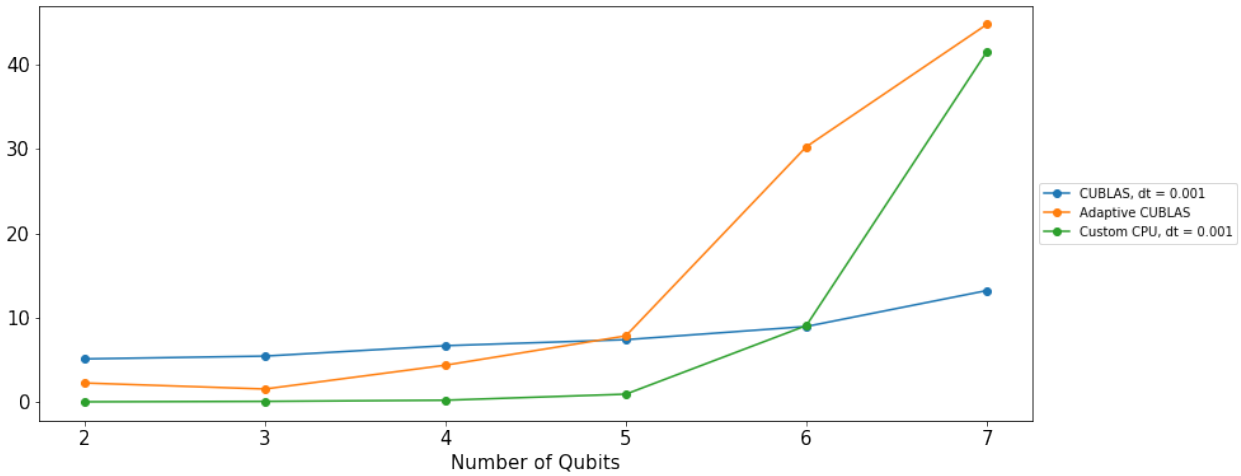


Figure 20: RK4 Lindblad master equation computation time on a CPU versus on a GPU; solving systems from 2 qubits to 7 qubits up until 0.5 units of time; tlist size of 2

Thus, we observe the crossover, wherein the GPU becomes faster, is at 6 qubits. If we were to further increase the number of qubits, we would see that using a GPU becomes even more advantageous. The issue, though, is that our Tesla K40 runs out of memory at 12 qubits.

5 Concluding Remarks

The goal of this project has been to simulate open quantum systems by numerically solving the Lindblad master equation on a GPU. As we have seen, at the end of Part 3, this goal was successfully achieved. On the way, we have seen crucial data on the speed with which a GPU can execute certain algorithms; in particular, we looked at list addition and matrix multiplication, which are the two most common operations when solving differential equations numerically.

In order to perform complicated computations in quantum physics, many researchers use Qutip [18]. However, there is no function in the Qutip library that solves the Lindblad master equation with a GPU. Once my code was complete, we began comparing its speed to Qutip's master equation solver:

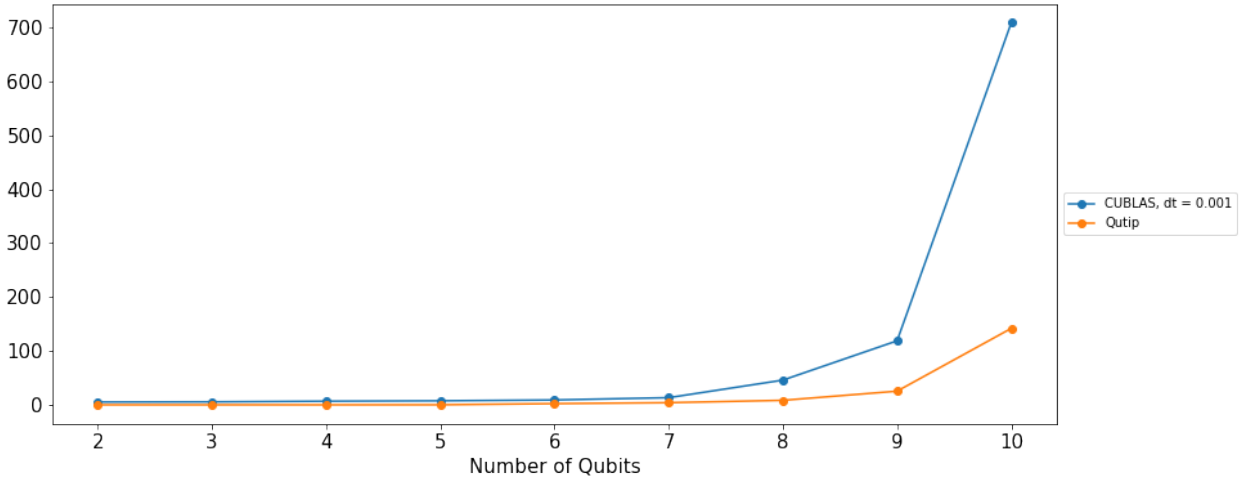


Figure 21: Custom GPU code compared to Qutip's solver; solving systems from 2 qubits to 10 qubits up until 0.5 unit of time; Hamiltonian and Lindblad operators are the same as in Section 4.2.2

We then began looking into why it takes so long for my GPU implementation to overtake Qutip's CPU code. We came across two main reasons:

1. Qutip uses sparse matrices
2. Qutip uses Scipy's numerical integration function and provides it with precompiled C code.

Therefore, the main next step in this project is to implement methods of sparse matrices on a GPU (including creation of the data structure and computations such as sparse matrix multiplication). As was mentioned at the end of Section 4.2.2.1, the Tesla K40 runs out of memory at 12 qubits, thus the implementation of sparse matrices would not only speed-up matrix computations, but also allow us to simulate even more qubits. An additional step would be to implement some analog of precompiling code.

References

[1] Feynman, Richard. "Simulating Physics with Computers." International Journal of Theoretical Physics 21.6/7 (1982): 467-88. Web. 29 May 2017. <https://people.eecs.berkeley.edu/ christos/classics/Feynman.pdf>.

[2] Nielsen, Michael A., and Isaac L. Chuang. Quantum Computation and Quantum Information. 7th ed. Cambridge: Cambridge UP, 2010. Print.

[3] "NVIDIA's Next Generation CUDATM Compute Architecture: Fermi.": n. pag. Web. <http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute _architecture_whitepaper.pdf>.

[4] Shankar, Ramamurti. Principles of Quantum Mechanics. 2nd ed. New York, NY: Springer, 1994. Print.

[5] Griffiths, David J. Introduction to Electrodynamics. 3rd ed. Harlow: Prentice-Hall, 1999. Print.

[6] Feynman, Richard P., Robert B. Leighton, and Matthew L. Sands. The Feynman Lectures: Volume III: Quantum Mechanics. 3rd ed. Pasadena, CA: California Institute of Technology, 2010. Print.

[7] Bardeen, John, Leon Cooper, and John Schrieffer. "Theory of Superconductivity." Physical Review 108.5 (1957): 1175-204. Web. 29 May 2017. <https://journals.aps.org/pr/pdf/10.1103/PhysRev.108.1175>.

[8] <https://en.wikipedia.org/wiki/File:Single_josephson_junction.svg>.

[9] Manucharyan, V. E., J. Koch, L. I. Glazman, and M. H. Devoret. "Fluxonium: Single Cooper-Pair Circuit Free of Charge Offsets." Science 326.5949 (2009): 113-16. Web.

[10] Vool, Uri, and Michel H. Devoret. "Introduction to Quantum Electromagnetic Circuits." [1610.03438] Introduction to Quantum Electromagnetic Circuits. N.p., 11 Oct. 2016. Web. 30 May 2017. <https://arxiv.org/abs/1610.03438>.

[11] Kou, A., W. C. Smith, U. Vool, R. T. Brierley, H. Meier, L. Frunzio, S. M. Girvin, L. I. Glazman, and M. H. Devoret. "A Fluxonium-based Artificial Molecule with a Tunable Magnetic Moment." [1610.01094] A Fluxonium-based Artificial Molecule with a Tunable Magnetic Moment. N.p., 05 Dec. 2016. Web. 30 May 2017. <https://arxiv.org/abs/1610.01094>.

[12] Brasil, Carlos Alexandre, Felipe Fernandes Fanchini, and Reginaldo De Jesus Napolitano. "A Simple Derivation of the Lindblad Equation." Revista Brasileira De Ensino De Fsica 35.1 (2013). Web.

[13] Breuer, Heinz-Peter, and Francesco Petruccione. The Theory of Open Quantum Systems. 2nd ed. Oxford: Clarendon, 2003. Print.

[14] Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA TESLA : A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE (n.d.): n. pag. IEEE, 2008. Web. <http://people.cs.umass.edu/ emery/classes/cmpsci691st/readings/Arch/gpu.pdf>.

[15] Tian, Xiaonan, Rengan Xu, Yonghong Yan, Sunita Chandrasekaran, Deepak Eachempati, and Barbara Chapman. "Compiler Transformation of Nested Loops for General Purpose GPUs." Concurrency and Computation: Practice and Experience 28.2 (2015): 537-56. Web.

[16] "CUDA." CUDA — GeForce. N.p., n.d. Web. 03 June 2017. <http://www.geforce.com/hardware/technology/cuda>.

[17] Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical Recipes in C. 2nd ed. Cambridge: Cambridge U Pr., 1997. Print.

[18] Nation, P. D., and J. R. Johansson. QUTIP - Quantum Toolbox in Python. Web. <http://qutip.org/>.

Appendix A Measuring Computation Time on a GPU

How one measures the computation time of a function on a GPU is not as simple as measuring a function on a CPU. On a CPU, we can harvest the time at which a function is called and the time right after the function ends using simple python functions. However, using a python timer is very much suboptimal for timing GPU functions. This is because when we call a GPU function in python, it is possible that the next function in the python script, which might be a python timer, will be called *before* the GPU function has finished. This is because python does not wait for the GPU to finish a computation before it moves to the next function in the python script.

The solution to this problem is the use of *events*. An event is, heuristically speaking, an object that is placed in the queue of a stream. Let's say we want to time a GPU function, which we will simply call `function`. Before we call `function`, we will create an event (this is done via the function `Event()` in the `pycuda.driver` library). By calling the function `Event()`, an event is added to the queue of a stream. After we call `function`, we would then create another event. In the queue of the stream, we essentially have a sandwich in which `function` is between two events.

One of the attributes of an event object is the time at which the stream passes it in the queue. Therefore, we can harvest the time at which the first event is called and then harvest the time at which the second event is called. The difference between the two is, clearly, the time that `function` takes to execute.

Appendix B Code

The Final RK4 Solver

```
from __future__ import division
import numpy as np
import pycuda.driver as cuda
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
from pycuda.compiler import SourceModule
import skcuda.linalg as linalg
import skcuda

class GPU_mesolve_internals(object):
    def __init__(self, dt_cpu, hamiltonian_cpu, rho_initial_cpu, e_ops, diss_ops, tlist, tol, adapt, coupling_consts):

        self.dimension = hamiltonian_cpu.shape[0]
        self.hamiltonian = gpuarray.to_gpu(np.asarray(hamiltonian_cpu).astype(np.complex_))

        self.dt_original = np.asarray(dt_cpu)
        self.dt_dummy = self.dt_original
        self.dt_full = gpuarray.to_gpu(np.asarray([self.dt_dummy], dtype=np.complex_))
        self.dt_half = 0.5*self.dt_full

        result_cpu = np.zeros_like(hamiltonian_cpu, dtype=np.complex_)
        self.result = gpuarray.to_gpu(result_cpu)

        self.rho_evolved_cpu = np.asarray(rho_initial_cpu).astype(np.complex_)
        self.rho_evolved = gpuarray.to_gpu(self.rho_evolved_cpu)
        self.rho_evolved_for_adaptive = gpuarray.to_gpu(self.rho_evolved_cpu)
        self.rho_storage = gpuarray.zeros_like(self.rho_evolved)

        self.dissipation_ops = []
        self.dissipation_ops_herm = []
        self.couple_consts = []
        self.const_lindblad_term = []

        for ii in range(0,len(diss_ops)):
            self.dissipation_ops.append(gpuarray.to_gpu(np.asarray(diss_ops[ii]).astype(np.complex_)))
            self.dissipation_ops_herm.append( linalg.hermitian(self.dissipation_ops[ii]))
            self.const_lindblad_term.append(linalg.dot(self.dissipation_ops_herm[ii], self.dissipation_ops[ii]))
            self.couple_consts.append(gpuarray.to_gpu(np.asarray(coupling_consts[ii]).astype(np.complex_)))
```

```
        # everything for expected values
        self.e_ops_gpu = []
        self.elt_num_gpu = []
        self.exp_vals_gpu = []

        for ii in range(0,len(e_ops)):

            self.exp_vals_gpu.append([])
            self.e_ops_gpu.append(gpuarray.to_gpu(np.asarray(e_ops[ii]).astype(np.complex_)))

        self.dummy_array = gpuarray.to_gpu(np.zeros_like(rho_initial_cpu, dtype=np.complex_))
        self.tolerance = gpuarray.to_gpu(np.asarray(tol))

        self.do_adaptive = adapt


def runrk4_mesolve(rho_initial, time_list, hamiltonian, diss_ops, e_ops, coupling_consts, do_adaptive = 0, tol = 0.1):

    skcuda.misc.init()

    if(do_adaptive == 1):
        dt = 0.5*(time_list[1] - time_list[0])

    if(do_adaptive == 0):
        dt = 0.001

    gpu = GPU_mesolve_internals(dt, hamiltonian, rho_initial, e_ops, diss_ops, time_list, tol, do_adaptive, coupling_consts)

    exp_vals_calc(gpu)

    for ii in range(0,len(time_list)-1):

        runrk4_mesolve_core(gpu, time_list[ii], time_list[ii+1])

        exp_vals_calc(gpu)

    return gpu.exp_vals_gpu

def runrk4_mesolve_core(gpu, start_time, final_time):

    current_time = start_time

    while current_time < final_time:

        if gpu.dt_dummy > (final_time - current_time):

            gpu.dt_full = gpuarray.to_gpu(np.asarray([final_time - current_time], dtype=np.complex_))

            rk4_gpu_stepper(gpu, 1)

            gpu.dt_full = gpuarray.to_gpu(np.asarray([gpu.dt_dummy], dtype=np.complex_))
            gpu.dt_half = 0.5*gpu.dt_full

            current_time = final_time

        if current_time < final_time:

            if(gpu.do_adaptive == 1):
                adaptive_stepper(gpu, current_time, final_time)

            if(gpu.do_adaptive == 0):
                rk4_gpu_stepper(gpu, 1)

            current_time += gpu.dt_dummy

    return

def rk4_gpu_stepper(gpu, half_or_full):

    if(half_or_full == 1): # full
        rho = gpu.rho_evolved
        dt = gpu.dt_full

    if(half_or_full == 0): # half
        rho = gpu.rho_evolved_for_adaptive
        dt = gpu.dt_half

    lindblad(gpu, rho)

    k1_gpu = skcuda.misc.multiply(dt, (von_Neumann(gpu.hamiltonian, rho) + gpu.dummy_array))

    lindblad(gpu, rho + 0.5*k1_gpu)

    k2_gpu = skcuda.misc.multiply(dt, (von_Neumann(gpu.hamiltonian, rho + 0.5*k1_gpu) + gpu.dummy_array))

    lindblad(gpu, rho + 0.5*k2_gpu)

    k3_gpu = skcuda.misc.multiply(dt, (von_Neumann(gpu.hamiltonian, rho + 0.5*k2_gpu) + gpu.dummy_array))

    lindblad(gpu, rho + k3_gpu)
```

```
    k4_gpu = skcuda.misc.multiply(dt, (von_Neumann(gpu.hamiltonian, rho + k3_gpu) + gpu.dummy_array))

    gpu.rho_evolved = gpu.rho_evolved + k1_gpu/6.0 + k2_gpu/3.0 + k3_gpu/3.0 + k4_gpu/6.0

    return

def von_Neumann(first_term, second_term):
    return -1j*(linalg.dot(first_term, second_term) - linalg.dot(second_term, first_term))

def lindblad(gpu, density):

    gpu.dummy_array.fill(0.0)

    for ii in range(0, len(gpu.dissipation_ops)):

        a = linalg.dot(density, gpu.dissipation_ops_herm[ii])

        gpu.dummy_array += skcuda.misc.multiply(gpu.couple_consts[ii], linalg.dot(gpu.dissipation_ops[ii], a) -
              0.5*(linalg.dot(gpu.const_lindblad_term[ii], density) + linalg.dot(density, gpu.const_lindblad_term[ii])))

    return

def exp_vals_calc(gpu):

    for ii in range(0, len(gpu.e_ops_gpu)):

        gpu.exp_vals_gpu[ii].append(linalg.trace(linalg.dot(gpu.rho_evolved, gpu.e_ops_gpu[ii])))

    return

def adaptive_stepper(gpu, current_time, final_time):

    gpu.rho_storage = gpu.rho_evolved

    rk4_gpu_stepper(gpu, 1) # full_step

    rk4_gpu_stepper(gpu, 0) # half_step
    rk4_gpu_stepper(gpu, 0) # half_step

    difference = gpu.rho_evolved - gpu.rho_evolved_for_adaptive

    error = gpuarray.max(difference.__abs__())

    gpu.dt_dummy = min(final_time - current_time, (gpu.dt_full * (gpu.tolerance / error)).get().astype(np.float64))

    gpu.dt_full = gpuarray.to_gpu(np.asarray(gpu.dt_dummy, dtype=np.complex_))
    gpu.dt_half = 0.5 * gpu.dt_full

    gpu.rho_evolved = gpu.rho_storage
    gpu.rho_evolved_for_adaptive = gpu.rho_storage

    rk4_gpu_stepper(gpu, 1)

    return
```