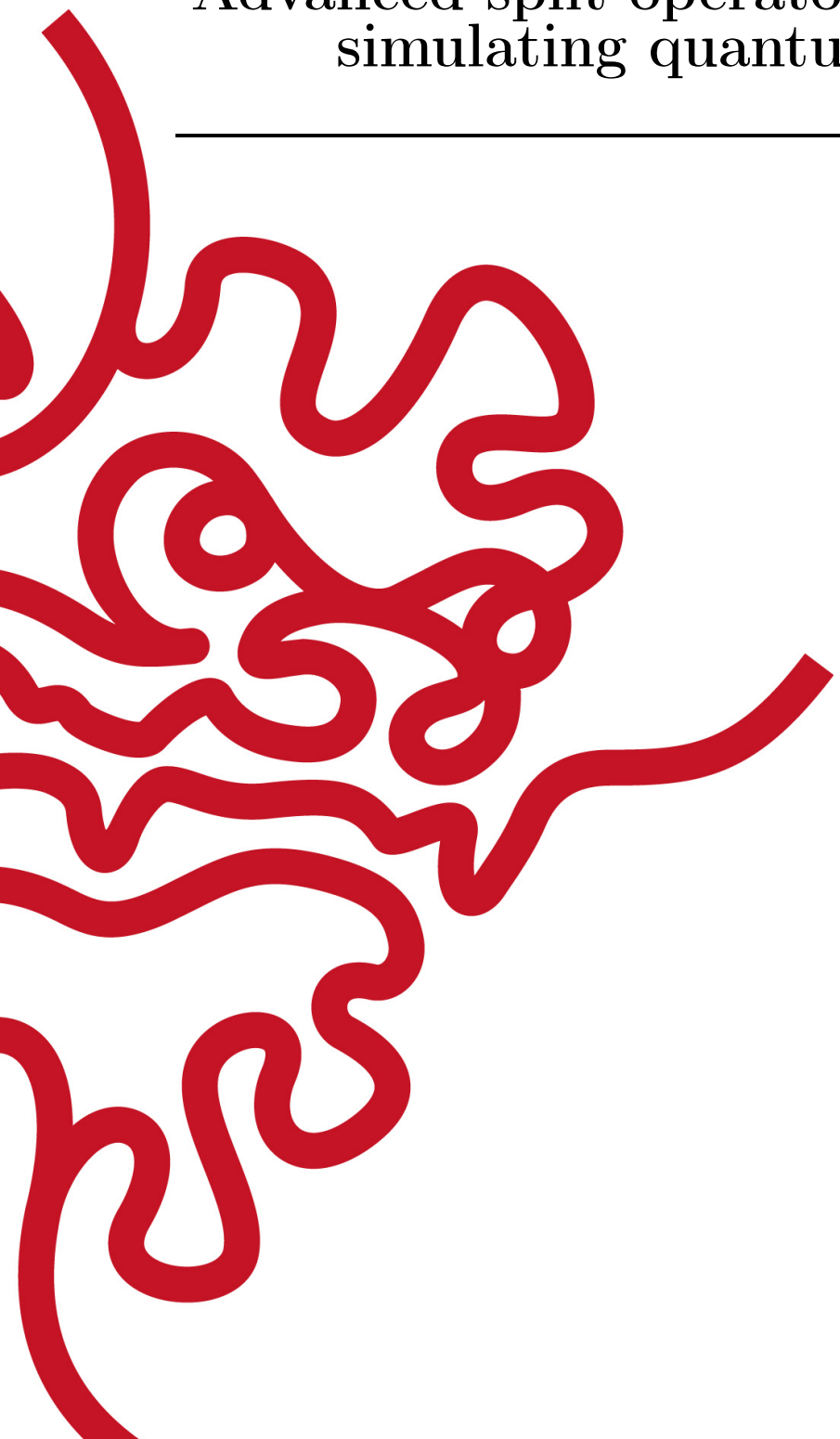
Okinawa Institute of Science and Technology
Graduate University

Thesis submitted for the degree

Doctor of Philosophy

Advanced split-operator techniques for simulating quantum systems



by

James Schloss

Supervisor: **Thomas Busch**

July, 2019

Contents

Chapter 1

Introduction General Purpose computing with Graphics Processing Units and the GPUE codebase

The Graphics Processing Unit (GPU) is a computing card that typically connects to the motherboard through a Peripheral Component Interconnect (PCI) slot. As the name implies, the GPU is designed to rapidly manipulate memory to create images or graphics that are sent to a display device, such as a monitor. Because individual pixels in images are independent of each other and modern computers require updating all pixels on the display device quickly, the GPU has been developed as a massively parallel computing device, capable of efficiently performing simple tasks (such as pixel generation or manipulation) rapidly by distributing the computation among many computing cores. This design methodology starkly contrasts the few, powerful cores on the Central Processing Unit (CPU), which is the default computing device on modern desktop systems. Due to this difference in hardware design, there are also several optimizations to consider when programming for massively parallel GPU devices, and several of these techniques will be covered in this chapter.

As GPU technology grew, other areas of computational science became increasingly

hungry for computing power, specifically in the area of scientific computing on High-Performance Computing (HPC) systems. Historically, HPC systems were often developed as large, distributed networks of computing nodes intended for CPU-based computation. As such, these systems facilitated the development of highly parallel and distributed numerical methods to perform scientific computation.

With new, parallel algorithms being developed for HPC systems and GPU technology advancing rapidly to perform more computation in parallel to satiate the consumer demands for high-quality videos and graphics for video games and other media, it became possible to use the GPU as a scientific computing device with a new technique called General Purpose computing on Graphics Processing Units (GPGPU). Modern HPC design incorporates the GPU into each computing node, thereby increasing the throughput of the system, overall, and the fastest known supercomputer today (Summit, ORNL [1]), is almost entirely composed of GPU nodes with NVIDIA Tesla V100 cards (32 GB of available RAM), connected with NVlink and IBM's power architecture. In addition to the utility of GPGPU for scientific computing, GPU technology has also been rapidly developed for AI and related fields.

In this chapter, I will discuss the design methodology for the hardware and software related to GPGPU before proceeding to the development of GPUE, the GPU-based Gross-Pitaevskii Equation solver, which will be used for the remainder of this work. To start, we must first look into different types of parallelism and how these affect different hardware and software practices.

1.1 Types of parallelism

Older CPU architecture with a single core was designed as SISD (Single Instruction, Single Data) according to Flynn's taxonomy [2]. This simply means that no parallelism exists in the instructions or data. Even now, most code is naïvely written as if it is to be executed on SISD architecture, even though it is rare to find such a system in a

modern environment. For capable devices, there are two separate methods to parallelize computation: *Task parallelism* and *data parallelism*.

Task parallelism allows programmers to split their computation along as separate, non-interacting *tasks* or instructions, where each core performs its designated computation before moving on. On the other hand, data parallelism allows programmers to perform the same, repetitive task along a large data set by distributing threads across the data. Task parallelism is often better for dealing with a large number of specific actors, while data parallelism is often better for dealing with a large number of similar tasks on the same data, such as a large matrix. If a computing architecture allows for multiple instructions, but only a single data stream, it is considered to be MISD (Multiple Instruction, Single Data) by Flynn's taxonomy; meanwhile, if the architecture allows for multiple data streams with only a single instruction, it is considered to be SIMD (Single Instruction, Multiple Data). Most modern HPC systems are designed to be MIMD (Multiple Instruction, Multiple Data), and both task and data parallelism is exploited by developers; however, for GPU computation, data parallelism is often used more frequently.

In the realm of data parallelism, there is an extreme case where the data is *embarrassingly parallel*. Here, there could be a large matrix of data to manipulate, but no single element depends on any other. This means that when distributing computation along this matrix, we can simply assign tasks to each core without considering interactions with the rest of the data set. In this way, it is embarrassingly easy to parallelize, and hence the term *embarrassingly parallel*. As a note, matrix multiplications are embarrassingly parallel operations; however, FFTs are not [3]. As such, the SSFM is not overall embarrassingly parallel; however, because the FFTs are handled by the CuFFT library, programmers do not often need to consider task parallelism at all when developing SSFM code. Even so, understanding all the features of GPUE and its future directions requires a strong understanding of GPGPU, and this will be discussed in the following section.

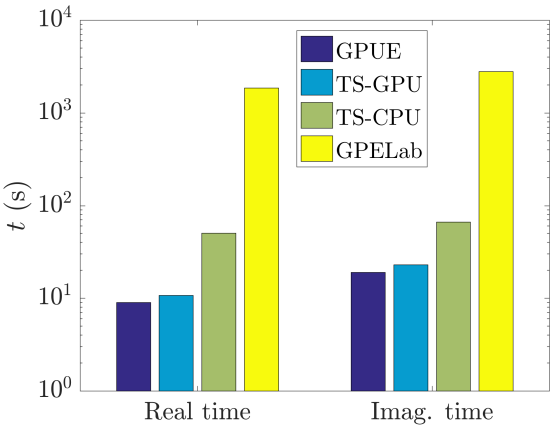


Figure 1.1: Comparison between GPUE (CUDA), Trotter-Suzuki on both GPU (CUDA) and CPU (C++), and GPElab (Matlab). Here, we see that GPUE is marginally faster than Trotter-Suzuki, but both GPU implementations are faster than the CPU-based variants. Both software packages are much faster than GPElab.

1.2 General purpose computing with graphics processing units

GPGPU programming is a relatively new development to the computing world and is generally much faster than CPU-based computation for tasks that can be easily parallelized in a SIMD fashion. Though benchmarks vary greatly depending programming languages, code quality, and intent of the software being benchmarked, our GPUE codebase is often 5 to 10 times faster than well-optimized C/C++ code and 100-200 times faster than Matlab code that is simulating the same system [4]. This is shown in Figure 1.1, where a comparison between GPUE, GPElab [5], and Trotter-Suzuki [6] are shown. These benchmarks are consistent with other GPGPU programs [7–9].

As it is possible to massively increase the performance of certain programs by using the GPU hardware, it is important to discuss the differences between GPGPU and CPU-based computation, along with important optimizations for GPU computing that will be used throughout this work. For the remainder of this work, I will use the term *host* interchangeably with CPU and *device* with GPU.

1.2.1 Limitations of GPU computing

GPGPU and massively parallel computation are best suited for embarrassingly parallel systems and there are several problems that are poorly suited to parallelization. For example, any task that is inherently iterative (such as summation) or recursive (such as tree traversal) is not suited for parallel computation. Even so, there are methods to reframe these problems such that they are better optimized for massively parallel devices, and these will be covered when relevant to the development of GPUE.

In addition to these algorithmic limitations, GPU cards have several notable drawbacks in terms of memory available on individual cards, which is often much less than the amount available on the host. As such, when simulating a large system on the GPU, we often limit the resolution to what can fit onto the GPU memory. In addition the data transfer between GPUs and between the GPU and CPU through the PCI bandwidth is a slow process. Until recently, these limited the size of our simulated wavefunction with GPUE to roughly 512^3 on a single Tesla K80 card. Higher resolution simulations could be performed with more recent cards (such as the Tesla V100) or by using multiple cards; however, because it takes time to transfer data between GPUs, we preferred to use a single card where possible. At this point, it is worthwhile to fully discuss GPU hardware and software ideologies, with particular focus on areas relevant to the development of GPUE. We will discuss important methods used in the development of GPUE to overcome these shortcomings afterward in Section 1.3.

1.2.2 GPU hardware architecture

Even though several programming frameworks exist with the capability of running code on the GPU, most of these hide necessary optimizations from the user. As such, we have chosen to focus exclusively on programming frameworks that expose the hardware for software developers, such as CUDA, OpenCL, and Julia. Though the following discussion will primarily focus on CUDA, a brief discussion of OpenCL and Julia can be found in

Listing 1.1: An example of vector addition performed in C or C++ for a, b, and c, all of size n

```
for (int i = 0; i < n; ++i){
    c[i] = a[i] + b[i];
}
```

Section 1.2.3, and example code for both languages can be found in the Appendix A. For the purposes of this discussion, we will cover only the GPU memory architecture of NVIDIA GPU devices as these are the most common computing devices for HPC systems.

This topic is easiest to describe by dividing it into two parts: an introduction to the software interface as defined by the CUDA API, followed by a discussion of the memory and thread hierarchy of GPU devices. Throughout these sections, we will discuss performance tips to ensure maximum GPU utilization, memory throughput, and instruction throughput.

Introduction to CUDA software interface

The CUDA parallel computing platform bares the hardware of the GPU to software developers. This means that important elements of this programming interface will appear in subsequent sections regarding hardware limitations and performance guidelines. Much of this discussion can be found in the *CUDA C Programming Guide* [10], while other sources will be cited as necessary. Full code for this discussion can be found in the Appendix A.

To start, let us assume a simple example where we would like to add two vectors such that $\mathbf{a} + \mathbf{b} = \mathbf{c}$. This can be done with a simple `for` loop in C, shown in Listing 1.1. In this case, we take each element with a specified ID in $\mathbf{a}$ and $\mathbf{b}$ and add them to the appropriate ID $\mathbf{c}$. In some parallel programming models (OpenACC [11], OpenMP [12], GPUifyLoops.jl, and many others), parallelization of this method is possible by adding a macro to the start of the loop to specify that this operation is to be performed in parallel; however, this obscures GPU hardware for the user and does not always have the

Listing 1.2: An example of a vector addition kernel in CUDA

```
1 __global__ void vecAdd(double *a, double *b, double *c){
2
3     // Global Thread ID
4     int id = threadIdx.x;
5
6     c[id] = a[id] + b[id];
7 }
```

same performance guarantees [13]. As such, CUDA takes a slightly different approach by encouraging software developers to write *kernels*, specific to the computation at hand. An example CUDA kernel for vector addition is shown in Listing 1.2, which has a number of notable differences to the `for` loop in C in Listing 1.2. This kernel is already remarkably different than an expected function on the CPU, and it is worth comparing Listings 1.1 and **??** in detail for a better understanding of GPU hardware.

The first peculiarity appears in line 1 with the `__global__` function specifier. This is a necessary element of all CUDA kernels that specifies where and when this kernel is capable of being called. A `__global__` kernel can be called by either host (with a standard CPU function) or the device (with a GPU kernel). A `__host__` kernel is exactly the same as a CPU function and can only be called by other CPU functions. Finally, a `__device__` kernel can only be called by other `__device__` or `__global__` kernels. As a note `__global__` kernels are incapable of returning vectors or other variables, and must instead mutate the variables, themselves. This is why the `__global__ vecAdd(...)` kernel does not return c, but instead assumes it is a pre-allocated variable.

Another peculiarity appears on line 6, where the addition, itself, occurs. Though there was a necessity for a `for` loop in Listing 1.1, there does not seem to be one at all in Listing 1.2. This is because the GPU is handling the parallelism behind the scenes on line 4 with the `int id = threadIdx.x` command. In this line, we are identifying which element of the array we are operating on with the CUDA-specific `threadIdx.x` variable.

Here, each *thread* is an individual instructional element acted on in parallel with other

threads in the same *block*, which is further subdivided into *grids*. All threads in the same block have a *shared memory* resource, while all three have access to *global memory*. In general, it is important to use shared memory when possible, as it has a lower latency than global memory, and this will be discussed further in subsequent sections. As a note, threads often work without feedback from other threads; moreover, it may be necessary to stop thread execution until all other threads have caught up. This can be done with the `__syncthreads()` function in CUDA.

Threads, blocks, and grids are all `dim3` variables with x, y, and z attributes. The way in which these threads and blocks are allocated are defined by the user before kernel execution. Often times, a thread number of 1024 is chosen, and the number of blocks is decided based on the size of the input array. If there are more elements to compute than threads in a block, we then need to use the `blockDim.x` and `blockIdx.x` variables to access the appropriate threads for computation. All threads are indexed as one-dimensional vectors even in a multidimensional space, as shown in Figure 1.2. Even though threads are rarely acted on sequentially, the thread ID has important ramifications that will be discussed further with other performance tips.

As another note, CUDA will execute code on unallocated memory if you do not tell it to otherwise. As such, if we had failed to set the number of threads in our block to the number of elements in our array in Listing 1.2, the kernel would exhibit undefined behavior. For this reason, we need to take into account potential out-of-bounds computation. If we take the above example of vector addition, assuming that the thread count is higher than can fit in one block and take into consideration potential out-of-bounds behavior, the kernel would instead look like Listing 1.3.

Finally, we need to discuss how software developers call these CUDA kernels in host code. This requires the developer to allocate space on the device for use in the CUDA kernel via a `cudaMalloc(...)` command. Often times, arrays on the host must also be established and transferred to the device with `cudaMemcpy(...)` functions as well. In addition, the kernel must be configured before running with `<<<grid, threads,...>>>`.

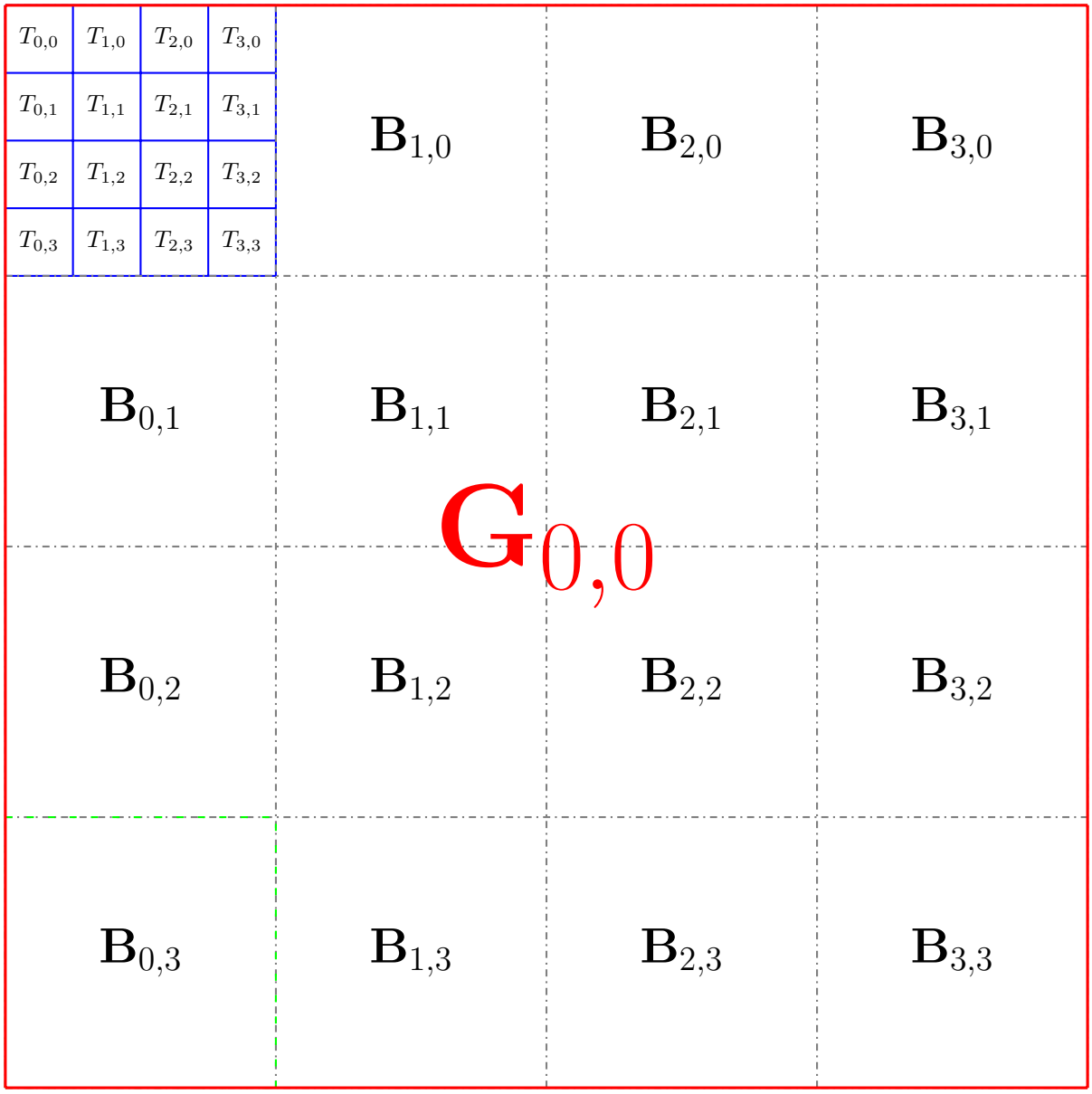


Figure 1.2: Each grid is subdivided into multiple blocks, which is further subdivided into threads for computation. Each thread has a specified ID, which acts as a one-dimensional array, even in a two or three-dimensional system. Here, all areas outlined in red have access to global memory, and any area outlined in blue has access to shared memory. [slight modification necessary to show 1D indexing!]

Listing 1.3: An example of a vector addition kernel in CUDA using blocks and threads, and ensuring no computation happens beyond the size of the array, n.

```
__global__ void vecAdd(double *a, double *b, double *c, int n){

    // Global Thread ID
    int id = blockDim.x * blockIdx.x + threadIdx.x;

    if (id < n){
        c[id] = a[id] + b[id];
    }
}
```

When everything is considered, the host code might look like what is shown in Listing 1.4

All said, vector addition is often known as the "Hello World!" of GPU programming as it is the first application that shows the parallelism of GPU devices; however, even here, we begin to see a distinction between users and developers. As more complex software is developed, it becomes more important to write software in such a way that users do not directly interface with CUDA code, and the full ramifications of this will be discussed in later in this chapter. In addition, this discussion highlights the important distinction between host and device code, including the concept of threads and blocks, shared memory, and the ability to transfer data from the host to the device. For now, we will continue this discussion by moving to GPU hardware, focusing on thread and memory hierarchy.

Discussion of GPU thread and memory hierarchy

Every time host code invokes a CUDA kernel call (as shown in Listing **??**), the data is mapped to a scalable array of multiprocessors on GPU hardware. Multiple blocks might be distributed to the same multiprocessor, but blocks are always distributed contiguously. Each multiprocessor is designed to execute hundreds of threads in parallel by using a unique SIMT (Single Instruction, Multiple Threads) architecture, which is similar to SIMD and can be used as such for most cases; however, there are performance benefits to optimizing instruction-level parallelism at the thread level. Each multiprocessor dis-

Listing 1.4: An example of host code to run Listing 1.3.

```
int main(){

    int n = 1024;

    // Initializing host vectors
    double *a, *b, *c;
    a = (double*)malloc(sizeof(double)*n);
    b = (double*)malloc(sizeof(double)*n);
    c = (double*)malloc(sizeof(double)*n);

    // Initializing all device vectors
    double *d_a, *d_b, *d_c;

    cudaMalloc(&d_a, sizeof(double)*n);
    cudaMalloc(&d_b, sizeof(double)*n);
    cudaMalloc(&d_c, sizeof(double)*n);

    // Initializing a and b
    for (size_t i = 0; i < n; ++i){
        a[i] = i;
        b[i] = i;
        c[i] = 0;
    }

    cudaMemcpy(d_a, a, sizeof(double)*n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, sizeof(double)*n, cudaMemcpyHostToDevice);

    dim3 threads, grid;

    // threads are arbitrarily chosen
    threads = {100, 1, 1};
    grid = {(unsigned int)ceil((float)n/threads.x), 1, 1};
    vecAdd<<<grid, threads>>>(d_a, d_b, d_c, n);

    // Copying back to host
    cudaMemcpy(c, d_c, sizeof(double)*n, cudaMemcpyDeviceToHost);

    // Check to make sure everything works
    for (size_t i = 0; i < n; ++i){
        if (c[i] != a[i] + b[i]){
            std::cout << "Yo. You failed. What a loser! Ha\n";
            exit(1);
        }
    }

    std::cout << "You passed the test, congratulations!\n";

    free(a);
    free(b);
    free(c);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

tributes its parallel processes into *warps*, which are units of 32 threads that execute a single common operation at a time. Notably, the way a block is distributed into warps is always the same, so it is important to ensure that the input data is in powers of 32 to avoid wasting unnecessary computation. Outside of this, developers can often ignore SIMT behavior as long as they do not allow threads in a warp to have separate operations.

Now we will turn our focus to memory within GPU devices. There are three forms of GPU memory that are useful for most applications of GPGPU for scientific computation: Global memory, Shared memory, and texture memory. As described above, global memory is a shared between all grids, blocks, and threads and is considered to be the slowest memory bank. As such, whenever a warp accesses global memory, it tries to perform as few accessing operations as possible, which is made easier if the warp needs to access contiguous memory blocks. If the warp is required to access non-contiguous blocks, more accesses will be necessary and thus performance will take a relatively large hit. For this reason, it is important to make sure all data accesses are *coalesced*, which ensures that the warp will access consecutive elements as depicted in Figure 1.2.

For optimal memory throughput, shared memory is an essential tool to understand and use appropriately. As described, shared memory is on-chip memory that is shared between all threads in a block. The amount of shared memory available is hardware-dependent and configurable on kernel execution. In general, it is worthwhile to transfer data with a large number of accesses to shared memory for performance. Shared memory is split into several memory banks which can be accessed simultaneously. If two memory accesses are required of the same bank, there will be a conflict and the operation can no longer be performed in parallel. It is sometimes necessary to pad variables to prevent bank conflicts from occurring [14].

Of the three types of memory mentioned, texture memory is the least-often used and is primarily on the GPU for graphics computation and focuses on performance for two-dimensional structures. Texture memory has a relatively long write time, but is quick to read. It is also faster than global memory for non-coalesced access patterns and therefore

can be useful for certain stencil-based calculations. Unfortunately, it uses single-precision values and thus will not be used for the remainder of this work.

In addition to appropriate usage of memory on GPU architecture, it is also essential to minimize data transfers between the device and host and even between devices in multi-GPU setups. The data transfer between the host and device or between devices must send data through the PCI slot on the motherboard, which is a slow operation. For data transfer between devices, this transfer time can be slightly alleviated on Power architecture where NVlink technology can directly transfer data from device to device [15], but the data transfer between devices will still likely be the slowest part of the computation. In addition, CUDA-aware MPI for multi-GPU setups may add an huge burden on software development time [16, 17]. As such, developers often try to keep all of their computation on a single card, if possible, and several optimization strategies are used when multiple GPU cards are needed. These strategies will be covered on a case-by-case basis as they arise in this work.

As a final note, one optimization strategy for CUDA code that will not be discussed in-depth in this work is the maximization of instruction throughput. This simply means that programmers can increase the number of instructions performed over a specified period by trading precision for speed and minimizing thread synchronization. Because we are performing high-precision superfluid simulations, we cannot perform this trade-off. An important caveat here comes from conditionals, like `if` and `switch` statements. Here, programmers need to be careful not to accidentally cause the operation executed on threads in a warp to diverge.

1.2.3 Comparison between various languages for GPGPU computation

As one might expect, specialized programming languages are necessary to write code that compiles and runs on GPU architecture. There are several known libraries to extend

modern programming languages such as Matlab, python, and C++ to GPU devices; however, we will limit this discussion to common programming methods that allow fine-grained control of GPU memory and could be used for the development of GPUE. We will briefly discuss the advantages and disadvantages of three competing languages here: CUDA, OpenCL, and Julia, and as a simple example, vector addition in these languages is shown in Appendix A.

CUDA

CUDA is a computing API provided by NVIDIA for interfacing with NVIDIA GPUs and is the industry standard for GPGPU programming. CUDA is primarily limited by the NVIDIA-specific hardware it runs on, and although NVIDIA currently produces the most common GPUs for GPGPU programming, AMD GPU devices are also available and often cheaper for a similar level of computational power. In addition, CUDA support has recently ceased for Mac OS systems as NVIDIA cards are no longer bundled with current generation Mac computers, so CUDA code can only be used on Windows and Linux devices.

GPUE was written entirely in CUDA; however, due to the aforementioned limitations, there has been some consideration to re-writing the software in OpenCL or Julia.

OpenCL

Though CUDA is the industry-standard for GPGPU programming, OpenCL (Open Compute Language) is competitive in terms of performance and has the benefit of being compatible with NVIDIA and AMD GPU devices. OpenCL is also completely open-source and works as additional libraries to C or C++, which allows developers to compile OpenCL code with traditional compilers like `gcc` or `clang`. OpenCL has nearly identical structure to CUDA with slightly more verbose syntax, and thus provides all necessary functionality to develop and maintain scientific software. In addition, compute kernels are compiled at run-time, meaning that users can potentially modify kernels without recompiling the

code. This could be a huge boon for developers writing software for users who may need to quickly simulate a slightly modified system. Unfortunately, OpenCL has a rather cumbersome interface and has less peripheral support than CUDA. As such, it is rarely used for scientific computing software.

In the end, although OpenCL does provide the ability to more easily construct dynamic kernels, the increased engineering time necessary to write software in OpenCL is often not worth the cost; however, further advances in compiler design for heterogeneous architecture has been made in the past few years [18], which has provided the unique opportunity for computer scientists to write maintainable and fast code in new languages, like Julia.

Julia

Julia is a new language to scientific computing, but boasts promising results and claims to be as usable as python, but as performant as C [19]. This is a huge boon for maintainability. In addition, Julia's runtime is comparable to CUDA C for GPGPU computation and allows for similar hardware optimizations [20, 21], while also allowing users to edit the compiler implementations at will. This is an important point that will be discussed in more detail in Section 1.3.2.

In addition, because Julia is much easier to write than C for new programmers, GPU-based Julia code could allow developers to provide fast, efficient code with a usable interface for scientists and engineers. The trade off between performance and readability in programming has been described as the "two-language" problem, as most scientific computing solutions to this point have required using two languages: a fast language for the back-end and a readable language for the user interface. Julia succeeds in bridging the gap between the languages, effectively solving the two-language problem and allowing scientists and engineers to write efficient code that is even compilable on the GPU. For these reasons, we have begun porting our CUDA code to Julia, as it will lead to simpler and more maintainable code in the future. This will be further discussed in the conclusion

of this work, Chapter **??**.

1.3 Introduction to the GPUE codebase for n-dimensional simulations of quantum systems on the GPU

At this point, all the motivation and background necessary has been provided to discuss GPUE, the GPU-based Gross-Pitaevskii Equation solver. This codebase will be used for all remaining simulations performed in this work and its development has also inspired the development of other computational libraries such as the `DistributedTranspose.jl` package, which will also be discussed in Section 1.4. Some additional information on prior development of GPUE can be found in other sources [22]. For this section, I will first describe the FFT optimizations used in GPUE, followed by additional features necessary for dynamic simulations on GPU architecture.

1.3.1 FFT optimization

As mentioned in Chapter **??** and previously in this chapter, the SSFM is primarily limited by the complexity of the FFT operations. For a three dimensional simulation with gauge fields in the $\hat{x}$, $\hat{y}$, and $\hat{z}$ directions, one set of global FFTs and three sets of one-dimensional FFTs must be performed. This is equivalent to two three-dimensional FFT operations, which become much more complex when scaling to multiple GPU devices [3]. The CuFFT API provides an option for computing FFT's on separate batches of an array in GPU memory with the `cufftPlanMany(...)` command; however, if this command is repurposed for one-dimensional FFT operations, it does not provide the necessary functionality for FFTs across the $\hat{y}$ or $\hat{z}$ dimensions. With the CuFFT library, all FFT operations performed with this plan must follow an indexing pattern, such that

```
input[b*idist + x*istride]
output[b*odist + x*ostride]
```

where `b` is the batch ID, `x` is the element ID, `idist` and `odist` are the distances between batches for the input and output array, respectively, and `istride` and `ostride` are the strides between consecutive elements for computation with the input and output array, respectively. If data is transferred to the GPU, it must be re-indexed as a one-dimensional array, such that

```
array[i,j,k] = array[i + j*xDim + k*xDim*yDim]
```

where `i`, `j`, and `k` are iterable variables in the $\hat{x}$, $\hat{y}$, and $\hat{z}$ directions, and `xDim`, `yDim`, and `zDim` are the dimensions in $\hat{x}$, $\hat{y}$, and $\hat{z}$, respectively. As such, it is not possible to use the `cufftPlanMany` functionality to perform one-dimensional FFTs in $\hat{y}$ and $\hat{z}$; however, if we increase the number of batches to `xDim*yDim*zDim`, set the distance between each stride to 1, and assume the stride between each element is `xDim*yDim`, we can recreate the functionality of the FFT in the $\hat{z}$ direction. For the $\hat{y}$ FFT operations, we need an external loop that iterates over each xy slab, performing `xDim` operations on each slab, and this greatly hampers performance.

[Not sure if images will help here]

With this considered, only one-third of the necessary FFT operations are appropriately coalesced in memory for three dimensional simulations. Because FFTs are global operations that are best performed on contiguous chunks of memory, multi-GPU simulations with the SSFM are even less optimal. This has motivated the development of other packages to allow for memory coalescence with FFT operations, such as the distributed transpose, which will be described in Section 1.4. Even though the three-dimensional FFT operations are the biggest bottleneck in the GPUE codebase, it is not easy to avoid usage of the `cufftPlanMany(...)` operation while still using CUDA. Next, we will focus on another feature that was inhibited by the CUDA framework, but is nevertheless possible: methods to enable dynamic quantum engineering with expression trees.

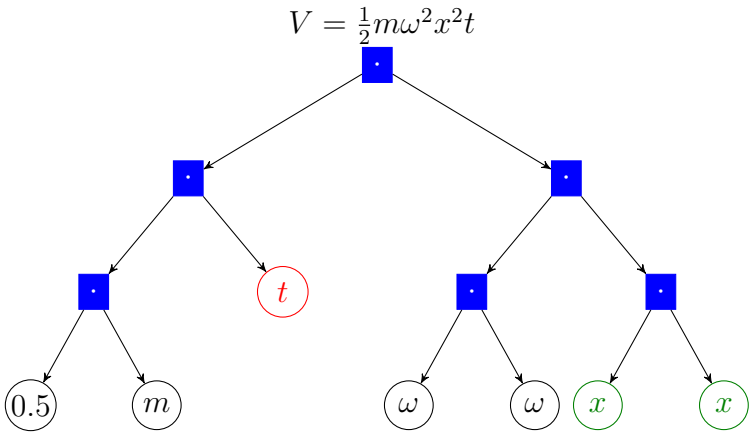


Figure 1.3: Example of expression tree for $V = \frac{1}{2}m\omega^2x^2t$. Blue, filled nodes are operations, leaf nodes are variables, time has been highlighted in red, and spatially-dependent variables are in green. This visualization was modified from a form provided by Xadisten during a Twitch livestream.

1.3.2 Dynamic field input and output in GPUE with expression trees

As mentioned in Chapter **??**, quantum engineering typically requires some form of time-dependent variables, along with evolution in real time. This means that the user must be able to input a time-dependent equation to GPUE. Because we chose to write GPUE in CUDA, there is no straightforward method for the user to input time-dependent fields without recompiling the source code and modifying CUDA kernels at will, which is unnecessarily cumbersome for the user. As such, we have provided a method for users to input the fields of their choosing as strings, which will be transpiled into an array of operations to perform on the GPU through expression trees, which are similar to Abstract Syntax Trees (ASTs) in compiler design [23, 24].

An example of an expression tree can be seen in Figure 1.3. These are evaluated depth-first to follow the traditional order of operations. With this method, a user can type in a string, like `"V = m*omega*omega*x*x*t"`, and this will be parsed into a set of operations to be performed on-the-fly by the GPU. After parsing user-provided equations, we designate certain leaf nodes that are either spatially or temporally dynamic. In the

case of spatially dynamic variables (`x`, `y`, and `z`), we pull values from constituent vectors based on their `threadIdx.xyz` values, and for any equation that is dependent on `t`, we pull upon a stored `time` variable. This operation necessitates the usage of a dictionary data structure to hold all variables in some fashion, which inhibits host performance; however, because the bulk of the computation is performed on the GPU, this does not significantly impact GPUE performance, overall. On the GPU, each necessary variable can be stored in a shared memory buffer, and because we are replacing the embarrassingly parallel element-wise matrix multiplications with these expression trees, the performance is not severely impacted; however, because parsing expression trees is an inherently iterative process, the longer the expression, the less optimal using this method is. Ultimately, more work could be done in the future to maximize instruction throughput with our implementation of GPU-accelerated expression trees.

Not only do expression trees allow for STA and quantum optimal control methods to be used with GPUE, but they also eliminate the need to store any operators in GPU memory, effectively increasing the available memory by a factor of 5 for each three-dimensional simulation, as V, K, A_x, A_y, and A_z no longer need to be stored. This allowed us to perform higher-resolution simulations and could allow for dynamical turbulence studies in the future; however, in order to scale beyond this limit, either multiple GPUs must be used or we must find some way to compress the wavefuntion. This will be discussed further in Section 1.3.5. As a final note, this feature can be implemented easier in other GPU frameworks, such as OpenCL and Julia.

The implementation of expression trees in GPUE effectively decreases the memory footprint of our simulations and allows for dynamical studies on the GPU; however, dynamical studies also require a large amount of fileIO. Though we had initially considered using the Compressed Split-Step Fourier Method (CSSFM) **??** to compress the size of our wavefunction, we will ultimately found this method to be undesirable for n-dimensional vortex simulations and instead reworked GPUE fileIO with HDF5. That said, for two-dimensional vortex simulations, we often do not need to output the entire condensate

wavefunction, but can instead output only the vortex locations. Methods of vortex tracking and highlighting will be discussed in the following section.

1.3.3 Vortex tracking and highlighting

In order to analyze the motion of vortices in a superfluid system, some form of vortex tracking must be implemented, and the current vortex tracking methods used in GPUE for two dimensions can be found in prior work [22] or the GPUE documentation [25]. It is important to describe two dimensional vortex tracking first before continuing to three dimensional vortex analysis, which is a much more complicated process.

At a first glance, one might assume that vortices are located at areas of low density in a superfluid system; however, this is not always the case. Because our condensate does not necessarily extend to the edges of the simulated domain, there will be large areas of zero density outside our condensate. In addition, sound waves and other perturbations with minimal density can occur. As such, locations of low density should only be used as educated guesses as to where actual vortices are located, but should not be used as the final predictor.

Instead, the phase can be used to uniquely identify vortex locations, as shown in Figure 1.4. In the highlighted region, all elements sum to a value of 2π. In this way, vortex tracking essentially becomes a straightforward task of locating all the 2π phase windings in the simulated domain via minimization routines where we attempt to find any four grid elements whose sum is 2π. This process also necessitates a mask for regions outside of the BEC domain, and further discussion on how to refine this position can be found in previous work [22, 25].

[I don't think vortex position refinement is relevant to what I did, which was the vortex highlighting in 3D, but I can add it in...].

In three dimensions, vortices are no longer confined to a plane and can extend in any direction, so long as the vortex lines either end at the end of the superfluid or reconnect in the form of vortex rings or more complicated vortex structures. This is a much more

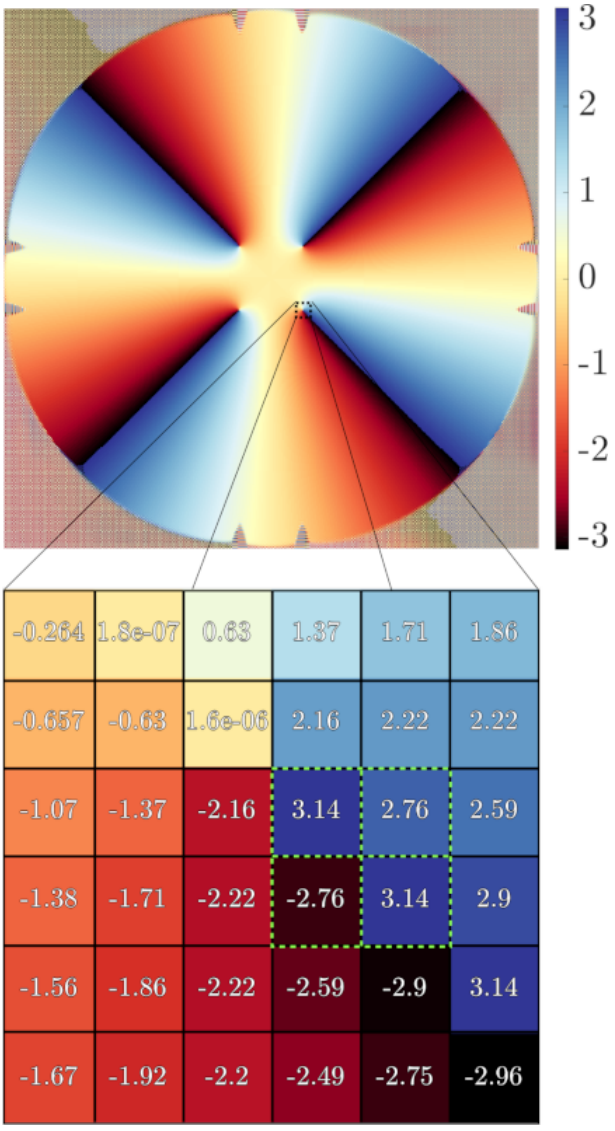


Figure 1.4: An example phase plot of a condensate with four vortices. The inset shows the values of the grid around each vortex location and highlights where the sum is 2π for vortex tracking.

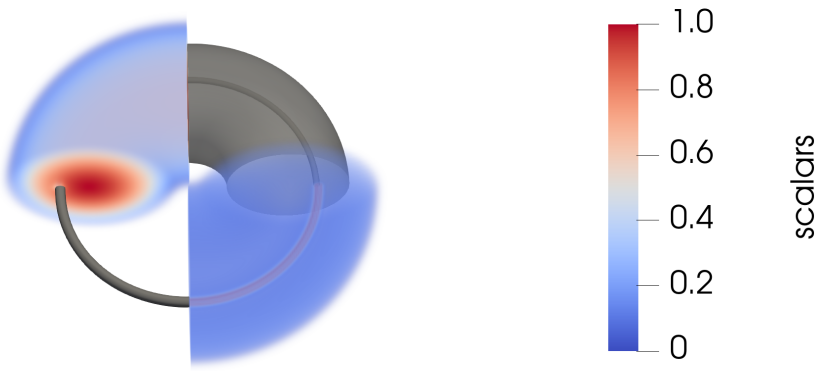


Figure 1.5: An example of vortex highlighting with a Sobel filter. The upper left quadrant is the superfluid density with no modifications and the upper right quadrant is an isosurface of the density with an opacity of 0.6. Note here that if there is no opacity set, it is not possible to see the vortex because it is obscured by the outside boundary of the BEC. The lower right quadrant is the superfluid density after Sobel filtering and the bottom left quadrant is an isosuface on the Sobel filtered density. Here, we can easily create isosurfaces of vortices that would be occluded when using the density, alone. The scale varies depending on whether it is coloring the normalized wavefunction density or the filtered density.

difficult problem which does not have many solutions in superfluid simulations where the superfluid does not fill the simulation domain. The current state-of-the-art solution has been proposed by Villois *et. al* [26], and requires finding density dips in the superfluid as initial guesses as to where a vortices might exist. From there, a vorticity plane is determined and the entire vortex is discovered by moving perpendicularly to the vorticity plane at each gridpoint. This is a tedious and time-consuming process that does not lend itself well to GPGPU computation without communication between the host and device. As such, we are currently seeking a more computationally efficient method for tracking vortices in three dimensions.

As our system do not necessarily fill the contents of our simulation domain, the proposed method will not work without some modification. We could still use the method if we have some understanding of the trapping geometry; however, as we discussed in Chapter **??**, this is not always the case with gauge fields.

As such, instead of focusing on vortex *tracking*, we have instead implemented a simple vortex *highlighting* scheme for three dimensions. This can be done with a Sobel filter on the condensate density, and can easily create crisp visualizations like those found in the computer graphics literature [27]. An example of a vortex highlighted wavefunction density, along with an isosurface of both the density and the highlighted density can be seen in Figure 1.5. In this figure, we show that the density after being Sobel filtered can be more easily used to isolate vortex structures without the background BEC. Though it would be possible to use further edge detection methods, such as the Canny edge detector [28], this would add a significant computational overhead and thus was not implemented in the current work. The problem of efficiently tracking vortex skeletons in three-dimensions is a difficult problem that requires further study; however, vortex highlighting is enough for most three dimensional vortex simulations. In Chapter **??**, we show and example simulation where vortex highlighting has been used to determine the vortex isosurfaces.

1.3.4 Energy calculation for superfluid simulations

As discussed in Chapter **??**, energy calculations can play an essential role in SSFM simulations and can be used to help understand vortex dynamics in certain simulations. More importantly, energy calculations lie at the heart of convergence criteria for imaginary time propagation. Essentially, in order to avoid unnecessary computation, many SSFM implementations will cease simulating the system in imaginary time when the change in energy every timestep drops below a certain threshold value. Though we have this option available in GPUE, it is not a native feature for at least three important reasons:

1. Certain systems, such as large vortex lattices with high rotation, have a seemingly degenerate ground state with different vortex configurations [22, 29, 30].

2. GPUE is often run on a computing cluster where the maximum simulation time is set before-hand. For this, the user must be able to estimate the duration of their

simulation, and this is not straightforward if imaginary-time propagation finishes at an unknown time.

3. The energy calculation is memory and operation-intensive and requires at least one additional object of the size of the wavefunction to be created and stored on GPU memory.

The first and second of these are somewhat self-explanatory, but the third requires further elucidation.

Energy calculations in GPUE are essentially composed of the following operation,

$$E = \langle \Psi | \hat{\mathcal{H}} | \Psi \rangle \tag{1.1}$$

The first problem with this operation is that it requires a summation for the final energy value, and as discussed, this is a poorly-suited problem for GPU hardware. Even though we have a robust implementation of parallel reduction, this is still a slow process. The next problem comes from the nature of the Hamiltonian, itself. As described in Chapter **??**, the Hamiltonian is essentially composed of three separate components for vortex simulations:

$$\hat{\mathcal{H}}_v = V_0 + g|\Psi|^2 + \frac{m\mathbf{A}^2}{2} \tag{1.2}$$

$$\hat{\mathcal{H}}_p = \frac{p^2}{2m} \tag{1.3}$$

$$\hat{\mathcal{H}}_{pv} = p\mathbf{A} \tag{1.4}$$

where $\hat{\mathcal{H}}_v$, $\hat{\mathcal{H}}_p$, and $\hat{\mathcal{H}}_{pv}$ are the Hamiltonians in position-space, momentum-space, and mixed-space, respectively. These operations can be considered with expression trees; however, for three-dimensional simulations they still require either a set of forward and inverse FFT's or a derivative function with fixed stride along with the parallel reduction operation. This ultimately amounts to the same number of operations required for a single step of imaginary-time evolution; however, because we do not want to influence the

simulated wavefunction, it requires at least one additional allocation of a wavefunction-sized array. Due to the computational time required for each energy calculation, we request users to input the set of timesteps they would like to compute the energy for before-hand. In addition, at certain points, it is impossible to run GPUE with the energy calculation, simply because there is not enough memory available on the device.

Though finding the energy of the wavefunction is a useful feature for certain simulations, it should not be used regularly for memory-limited tasks or tasks that should be performed quickly. Even so, for most applications of GPUE on HPC environments, there should be no problem running the energy-calculation alongside the simulation, itself.

1.3.5 Future direction and multi-GPU development

At this point, I would consider GPUE to be close to feature-complete. It is capable of simulating a wide-variety of quantum systems and can even perform dynamic quantum engineering studies with minimal fileIO. Though more work can be done to maximize instruction throughput, this will not significantly improve the performance of the code because we rely heavily on double-precision.

The next logical step for GPUE development is scaling to larger simulations. This means that we either need to increase the number of GPUs used for the simulation or decrease the size of the wavefunction, itself. Though the CSSFM method **??** should allow for the latter, we ultimately found it unsuitable for our purposes. As such, we are now attempting to scale GPUE to multiple GPU devices; however, as I hope to have impressed by now, this is not a trivial task. Even though the CuFFT library can support multiple GPU devices, this comes with a huge performance penalty, especially for the `cufftPlanMany(...)` functionality.

For this reason, we have begun the development of GPUE.jl, which has similar performance to GPUE, but is currently lacking the expression tree functionality. Once GPUE.jl is at feature parity with GPUE, we will then focus on it as the primary future direction of GPUE. Ultimately, the Julia language allows us to develop GPUE in a much more main-

tainable fashion, and also allows for us to access GPU hardware in a more convenient way. Because of this, we have already begun development on the tool that should allow for multiple GPU simulations with GPUE to be possible: the DistributedTranspose.jl package.

1.4 DistributedTranspose.jl

At its heart, the two-dimensional transpose is a straightforward operation consisting of a simple swapping of all row and column elements. Unsurprisingly, this is a rather difficult task to ensure memory coalescence, and the current recommended method for transposes on massively parallel devices requires heavy use of shared memory tiles to speed up the process [14]. In this case, it is possible to perform a two-dimensional transpose at the same performance as a simple copy, so long as the operation is out-of-place in memory. The transpose becomes even more difficult to create when we wish to transpose large three-dimensional matrices, potentially spanning across multiple GPU devices, while also ensuring the operation is in-place in memory.

In principle, there are three types of three-dimensional transposes:

Simple Copy A benchmark for other transpositions,

$$A_{xyz} \to A_{xyz} \tag{1.5}$$

Involution A transpose where a two-dimensional transpose is operated on a three-dimensional data structure,

$$A_{\mathbf{xyz}} \to A_{x\mathbf{zy}} \tag{1.6}$$

$$A_{\mathbf{xy}z} \to A_{\mathbf{xy}z} \tag{1.7}$$

$$A_{\mathbf{xyz}} \to A_{\mathbf{zyx}} \tag{1.8}$$

Rotation A fully three-dimensional transpose,

$$A_{\mathbf{xyz}} \to A_{\mathbf{yzx}} \tag{1.9}$$

$$A_{\mathbf{xyz}} \to A_{\mathbf{zxy}} \tag{1.10}$$

It has been shown that for out-of-place transpositions, it is possible to perform all of these operations as efficiently as a a simple copy; however, in-place rotational transposes can only attain 60% of the performance based on currently known methods [31]. In addition, distributed transposes of this nature have not been discussed except for an out-of-place array [32].

At its current state, the DistributedTranspose.jl package is able to do out-of-place, distributed transposes; however, when feature-complete, it should allow for the implementation of new distributed methods for such computation.

[I will probably move this section and GPUE future dev to conclusions / future work... But the research for this area should be completed by the time of the thesis defense. I am not sure whether to leave it in or not.]

Appendix A

Simple vector additions in CUDA, OpenCL, and JuliaGPU

This is a compilation of an introductory GPGPU example in three languages (CUDA, OpenCL, and Julia) to show the differences in different approaches to GPGPU computation and the necessary abstractions required by users in order to perform basic tasks using GPGPU

A.1 Vector addition with C++

Firstly, a simple vector addition without GPGPU as a baseline:

```
#include <iostream>

int main(){

    int n = 1024;

    // Initializing host vectors
    double *a, *b, *c;
    a = (double*)malloc(sizeof(double)*n);
    b = (double*)malloc(sizeof(double)*n);
```

```
    c = (double*)malloc(sizeof(double)*n);

    // Initializing a and b
    for (size_t i = 0; i < n; ++i){
        a[i] = i;
        b[i] = i;
        c[i] = 0;
    }

    // Vector Addition
    for (size_t i = 0; i < n; ++i){
        c[i] = a[i] + b[i];
    }

    // Check to make sure everything works
    for (size_t i = 0; i < n; ++i){
        if (c[i] != a[i] + b[i]){
            std::cout << "Yo. You failed. What a loser! Ha\n";
            exit(1);
        }
    }

    std::cout << "You passed the test, congratulations!\n";

    free(a);
    free(b);
    free(c);
}
```

A.2 Vector addition with CUDA

Now for vector addition with CUDA. Here, it is important to note that it is not practical to ask users to write CUDA kernels, as they are not skilled in GPGPU. In addition, direct kernel manipulation would require a recompilation every run, which is cumbersome for most users along with dedicated directories for differently built binaries if running multiple GPUE simulations simultaneously. As such additional techniques are used in GPUE to allow users to write their own functions without recompiling the GPUE codebase every run, and these methods are discussed in Chapter **??**

```
#include <iostream>
#include <math.h>
#include <chrono>

__global__ void vecAdd(double *a, double *b, double *c, int n){

    // Global Thread ID
    int id = blockIdx.x*blockDim.x + threadIdx.x;

    // Check to make sure we are in range
    if (id < n){
        c[id] = a[id] + b[id];
    }
}

int main(){

    int n = 1024;

    // Initializing host vectors
    double *a, *b, *c;
    a = (double*)malloc(sizeof(double)*n);
    b = (double*)malloc(sizeof(double)*n);
```

```
c = (double*)malloc(sizeof(double)*n);

// Initializing all device vectors
double *d_a, *d_b, *d_c;

cudaMalloc(&d_a, sizeof(double)*n);
cudaMalloc(&d_b, sizeof(double)*n);
cudaMalloc(&d_c, sizeof(double)*n);

// Initializing a and b
for (size_t i = 0; i < n; ++i){
    a[i] = i;
    b[i] = i;
    c[i] = 0;
}

cudaMemcpy(d_a, a, sizeof(double)*n, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(double)*n, cudaMemcpyHostToDevice);

dim3 threads, grid;

// threads are arbitrarily chosen
threads = {100, 1, 1};
grid = {(unsigned int)ceil((float)n/threads.x), 1, 1};
vecAdd<<<grid, threads>>>(d_a, d_b, d_c, n);

// Copying back to host
cudaMemcpy(c, d_c, sizeof(double)*n, cudaMemcpyDeviceToHost);

// Check to make sure everything works
for (size_t i = 0; i < n; ++i){
    if (c[i] != a[i] + b[i]){
        std::cout << "Yo. You failed. What a loser! Ha\n";
```

```
            exit(1);
        }
    }

    std::cout << "You passed the test, congratulations!\n";

    free(a);
    free(b);
    free(c);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

A.3 Vector addition with OpenCL

Vector addition with OpenCL has notable advantages to CUDA, namely that users can update the kernels without recompilation. Though it is tempting to use OpenCL due to this, it is notably more cumbersome to write OpenCL code than CUDA, and it we lack the engineering resources to develop an OpenCL variant of GPUE. In addition, Julia provides similar benefits to OpenCL with much higher maintainability.

```
#define __CL_ENABLE_EXCEPTIONS

#include <CL/cl.hpp>
#include <iostream>
#include <vector>
#include <math.h>

// OpenCL kernel
const char *kernelSource =                                    "\n" \
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable       \n" \
```

```
"__kernel void vecAdd( __global double *a,          \n" \
"                      __global double *b,          \n" \
"                      __global double *c,          \n" \
"                      const unsigned int n){       \n" \
"                                                   \n" \
"    // Global Tread ID                             \n" \
"    int id = get_global_id(0);                     \n" \
"                                                   \n" \
"    // Remain in boundaries                        \n" \
"    if (id < n){                                   \n" \
"        c[id] = a[id] + b[id];                     \n" \
"    }                                              \n" \
"}                                                  \n";

int main(){
    unsigned int n = 1024;

    double *h_a, *h_b, *h_c;

    h_a = new double[n];
    h_b = new double[n];
    h_c = new double[n];

    for (size_t i = 0; i < n; ++i){
        h_a[i] = 1;
        h_b[i] = 1;
    }

    cl::Buffer d_a, d_b, d_c;

    cl_int err = CL_SUCCESS;
    try{
        std::vector<cl::Platform> platforms;
```

```
cl::Platform::get(&platforms);
if(platforms.size() == 0){
    std::cout << "Platforms size is 0\n";
    return -1;
}

cl_context_properties properties[] =
    { CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[0])(), 0 };

cl::Context context(CL_DEVICE_TYPE_GPU, properties);
std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

cl::CommandQueue queue(context, devices[0], 0, &err);

d_a = cl::Buffer(context, CL_MEM_READ_ONLY, n*sizeof(double));
d_b = cl::Buffer(context, CL_MEM_READ_ONLY, n*sizeof(double));
d_c = cl::Buffer(context, CL_MEM_WRITE_ONLY, n*sizeof(double));

queue.enqueueWriteBuffer(d_a, CL_TRUE, 0, n*sizeof(double), h_a);
queue.enqueueWriteBuffer(d_b, CL_TRUE, 0, n*sizeof(double), h_b);

cl::Program::Sources source(1,
    std::make_pair(kernelSource,strlen(kernelSource)));
cl::Program program_ = cl::Program(context, source);
program_.build(devices);

cl::Kernel kernel(program_, "vecAdd", &err);

kernel.setArg(0, d_a);
kernel.setArg(1, d_b);
kernel.setArg(2, d_c);
kernel.setArg(3, n);
```

```
        cl::NDRange localSize(64);

        cl::NDRange globalSize((int)(ceil(n/(float)64)*64));

        cl::Event event;
        queue.enqueueNDRangeKernel(
            kernel,
            cl::NullRange,
            globalSize,
            localSize,
            NULL,
            &event
        );

        event.wait();
        queue.enqueueReadBuffer(d_c, CL_TRUE, 0, n*sizeof(double), h_c);
    }
    catch(cl::Error err){
        std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")\n";
    }

    // Check to make sure everything works
    for (size_t i = 0; i < n; ++i){
        if (h_c[i] != h_a[i] + h_b[i]){
            std::cout << "Yo. You failed. What a loser! Ha\n";
            exit(1);
        }
    }

    std::cout << "You passed the test, congratulations!\n";

    delete(h_a);
    delete(h_b);
```

```
    delete(h_c);
}
```

A.4 Julia

Julia is a relatively new language, but boasts the performance of CUDA without as much bulk. In addition, Julia allows users to more easily look at the AST for compilation, thus rendering GPUE's AST process obsolete. Here is vector addition in Julia.

```
using CUDAnative, CUDAdrv, CuArrays, Test

function kernel_vadd(a, b, c)
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x
    j = (blockIdx().y-1) * blockDim().y + threadIdx().y
    @inbounds c[i,j] = a[i,j] + b[i,j]
    return nothing
end

function main()

    res =1024

    # CUDAdrv functionality: generate and upload data
    a = round.(rand(Float32, (1024, 1024)) * 100)
    b = round.(rand(Float32, (1024, 1024)) * 100)
    d_a = CuArray(a)
    d_b = CuArray(b)
    d_c = similar(d_a)  # output array

    # run the kernel and fetch results
    # syntax: @cuda [kwargs...] kernel(args...)
    @cuda threads = (128, 1, 1) blocks = (div(res,128),res,1)
       kernel_vadd(d_a, d_b, d_c)
```

```
    # CUDAdrv functionality: download data
    # this synchronizes the device
    c = Array(d_c)
    a = Array(d_a)
    b = Array(d_b)

    @test isapprox(a+b, c)
end
```

Bibliography

[1] J. A. Kahle, J. Moreno, and D. Dreps. 2.1 summit and sierra: Designing ai/hpc supercomputers. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 42–43, Feb 2019.

[2] JR Gurd. A taxonomy of parallel computer architectures. In *1988 International Specialist Seminar on the Design and Application of Parallel Digital Processors*, pages 57–61. IET, 1988.

[3] Kenneth Czechowski, Casey Battaglino, Chris McClanahan, Kartik Iyer, P-K Yeung, and Richard Vuduc. On the communication complexity of 3d ffts and its implications for exascale. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 205–214. ACM, 2012.

[4] P Wittek. Comparing three numerical solvers of the gross-pitaevskii equation, 2016.

[5] Xavier Antoine and Romain Duboscq. Gpelab, a matlab toolbox to solve gross–pitaevskii equations i: Computation of stationary solutions. *Computer Physics Communications*, 185(11):2969–2991, 2014.

[6] Peter Wittek and Fernando M Cucchietti. A second-order distributed trotter–suzuki solver with a hybrid cpu–gpu kernel. *Computer Physics Communications*, 184(4):1165–1171, 2013.

[7] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick,

Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE micro*, 28(4):13–27, 2008.

[8] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[9] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[10] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[11] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[12] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[13] Ruymán Reyes, Iván López-Rodríguez, Juan J Fumero, and Francisco De Sande. accull: an openacc implementation with cuda and opencl support. In *European Conference on Parallel Processing*, pages 871–882. Springer, 2012.

[14] Mark Harris. An efficient matrix transpose in cuda c/c++. *Retrieved July*, 26:2018, 2013.

[15] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[16] Vladimir Lončar, Luis E Young-S, Srdjan Škrbić, Paulsamy Muruganandam, Sadhan K Adhikari, and Antun Balaž. Openmp, openmp/mpi, and cuda/mpi c programs

for solving the time-dependent dipolar gross–pitaevskii equation. *Computer Physics Communications*, 209:190–196, 2016.

[17] Hao Wang, Sreeram Potluri, Devendar Bureddy, Carlos Rosales, and Dhabaleswar K Panda. Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605, 2013.

[18] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, 132:29–46, 2019.

[19] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[20] Tim Besard, Pieter Verstraete, and Bjorn De Sutter. High-level gpu programming in julia. *arXiv preprint arXiv:1604.03410*, 2016.

[21] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: unleashing julia on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, 2018.

[22] Lee James O'Riordan. *Non-equilibrium vortex dynamics in rapidly rotating Bose-Einstein condensates*. PhD thesis, Okinawa Institute of Science and Technology Graduate University, 2017.

[23] Robert F Cohen and Roberto Tamassia. Dynamic expression trees and their applications. In *SODA*, pages 52–61, 1991.

[24] Ruyman Reyes and Francisco de Sande. Automatic code generation for gpus in llc. *The Journal of Supercomputing*, 58(3):349–356, 2011.

[25] James Schloss and Lee O'riordan.

[26] Alberto Villois, Giorgio Krstulovic, Davide Proment, and Hayder Salman. A vortex filament tracking method for the gross–pitaevskii model of a superfluid. *Journal of Physics A: Mathematical and Theoretical*, 49(41):415502, 2016.

[27] Yulong Guo, Xiaopei Liu, Chi Xiong, Xuemiao Xu, and Chi-Wing Fu. Towards high-quality visualization of superfluid vortices. *IEEE transactions on visualization and computer graphics*, 24(8):2440–2455, 2018.

[28] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[29] LJ O'Riordan, AC White, and Th Busch. Moiré superlattice structures in kicked bose-einstein condensates. *Physical Review A*, 93(2):023609, 2016.

[30] Lee James O'Riordan and Thomas Busch. Topological defect dynamics of vortex lattices in bose-einstein condensates. *Physical Review A*, 94(5):053603, 2016.

[31] Jose L Jodra, Ibai Gurrutxaga, and Javier Muguerza. Efficient 3d transpositions in graphics processing units. *International Journal of Parallel Programming*, 43(5):876–891, 2015.

[32] Gregory Ruetsch and Massimiliano Fatica. *CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming*. Elsevier, 2013.