

CM3202 - One Semester Project (40 Credits)
6th May 2016

Parallelisation of Matrix Exponentials in C++/CUDA for Quantum Control

-- Final Report --

Author: Peter Davison (C1310136)
Supervisor: Dr. Frank C Langbein
Moderator: Dr. Irena Spasic

Tables of Contents

1 - Introduction

- 1.1 - What is a Matrix Exponential?
- 1.2 - What is it used for?
- 1.3 - The problem
- 1.4 - The solution

2 - Background

- 2.1 - Taylor Series
- 2.2 - Padé Approximation
- 2.3 - Existing solutions
- 2.4 - How to optimise a program
- 2.5 - What is a GPU?
- 2.6 - Aims

3 - Design

- 3.1 - Environment
- 3.2 - Programming language
- 3.3 - Handling Memory
- 3.4 - Pseudocode
- 3.5 - Class Diagram

4 - Implementation

- 4.1 - Sequential
 - 4.1.1 - Header files (.h | .cuh)
 - 4.1.2 - Code files (.cpp | .cu)
 - 4.1.3 - Constructors & Destructors
 - 4.1.4 - Getters, Setters & Booleans
 - 4.1.5 - Matrix functions
 - 4.1.6 - Padé Approximation
 - 4.1.7 - Operator overrides and output
- 4.2 - Parallel (CUDA)
 - 4.2.1 - Installing CUDA
 - 4.2.2 - Significant Changes

5 - Numerical Experiments

- 5.1 - Method
- 5.2 - Testing scripts
 - 5.2.1 - Matlab
 - 5.2.2 - C++/CUDA
- 5.3 - Results

6 - Future Work

7 - Conclusion

8 - Appendix

8.1 - Screenshots

8.2 - Code

8.2.1 - CUDAMatrix.cu

8.2.2 - CUDAMatrix.cuh

8.2.3 - CUDATimer.cu

8.2.4 - CUDATimer.cuh

8.2.5 - main.cpp

8.3 - Table of Abbreviations

8.4 - Glossary

1 - Introduction

1.1 - What is a Matrix Exponential?

The exponential of a matrix is written e^A where A is a real or complex matrix. It is also commonly referred to by the function $\expm(A)$. In a 1x1 matrix (A), the output of $\expm(A)$ will be a matrix whose single element equals $e^{A_{1,1}}$. In any other diagonal matrix, the matrix exponential can be calculated such that the diagonal elements equal $e^{A_{x,x}}$ Where x, x represents each diagonal value.^{1 2}

1.2 - What is it used for?

Matrix exponentials are commonly used in number theory and lie algebra. In the context of quantum control, they are used to break down wave functions and measure the spin of a particle (eg. a photon). We can observe how the energy state of a system changes over time by using a constant hamiltonian (H) and Schrödinger's Equation (see below). This equation gives the current energy state at a given time (t)³

$$\left| \psi(t) \right\rangle = e^{-i H t / \hbar} \left| \psi(0) \right\rangle$$

Where i is any complex number, \hbar equals 1, H is a hamiltonian matrix and t is the current time step.

1.3 - The problem

Despite the fact that modern computers are exponentially faster than they were 50 years ago⁴, matrix exponentials are still very intensive on a processor. When used in a quantum control problem, many of these calculations are required and the time this takes escalates rapidly as both the size of the matrices and the quantity of calculations increases.

¹ Weisstein, EW. "Matrix Exponential -- from Wolfram MathWorld." 2004.

<<http://mathworld.wolfram.com/MatrixExponential.html>>

² "Matrix exponential - Wikipedia, the free encyclopedia." 2011. 4 May. 2016

<https://en.wikipedia.org/wiki/Matrix_exponential>

³ "Hamiltonian (quantum mechanics) - Wikipedia, the free encyclopedia." 2011. 6 May. 2016

<[https://en.wikipedia.org/wiki/Hamiltonian_\(quantum_mechanics\)](https://en.wikipedia.org/wiki/Hamiltonian_(quantum_mechanics))>

⁴ "File:Supercomputing-rmax-graph2.svg - Wikimedia Commons." 2015. 6 May. 2016

<<https://commons.wikimedia.org/wiki/File:Supercomputing-rmax-graph2.svg>>

This is not helped by the ending of Moore's Law.⁵ With processors unlikely to maintain their exponential growth in speed, we won't see many increases in performance with our current methods of calculation. Instead we must look to other methods such as optimisation.

1.4 - The solution

In order to increase the performance of the algorithm, we need to increase the amount of parallel processing we can do. This requires either a network, a server or a dedicated device with many cores - Also known as a GPU or graphics card. We can use a compatible graphics card with Nvidia's CUDA program which is designed for parallelisation to split the problem across many cores and therefore retrieve the solution much quicker.

⁵ "Moore's law - Wikipedia, the free encyclopedia." 2011. 6 May. 2016
<https://en.wikipedia.org/wiki/Moore's_law>

2 - Background

2.1 - Taylor Series

There are a few ways to calculate a matrix exponential. Except for some special cases (eg. diagonal matrices), these calculations are all approximations. The simplest way is to use a variation of the Taylor Series (See below). As with any series, the further you pursue it, the more accurate your result becomes.

$$e^A = I_n + \frac{1}{1!}A + \frac{1}{2!}A^2 + \frac{1}{3!}A^3 + \dots + \frac{1}{n!}A^n + \dots$$

While this method is simple in theory, a computer must follow the series until at least the 40th term before the result is accurate enough to use. This can take a long time to calculate and there are better methods available.

2.2 - Padé Approximation

This is where the Padé Approximation ⁶ comes in. This variation of the equation above was developed by french mathematician Henri Padé ⁷ in the 1890s. His method involves selectively picking terms from the Taylor Series depending on the properties of the input matrix. This results in a series that may only consist of ~7 terms and is therefore much quicker to calculate.

The terms that the algorithm chooses are selected from a Padé Table ^{8 9}. Each cell of the table gives an equation for the best 'version' of the Taylor Power Series for the given inputs. It calculates the inputs m and n by comparing the 1 norms of the input matrix and its powers to a fixed array of coefficients.

The first column of the Padé Table (where $n = 0$) contains the exact successive terms of the Taylor Series and the first row (where $m = 0$) contains the reciprocals of these.

This is why the Padé Algorithm is erratic when it comes to performance time. It chooses a different calculation each time it runs depending on the input matrix and the complexity of these calculations differ enormously.

⁶ "Padé approximant - Wikipedia, the free encyclopedia." 2011. 4 May. 2016

<https://en.wikipedia.org/wiki/Pad%C3%A9_approximant>

⁷ <http://en.wikipedia.org/wiki/Henri_Pad%C3%A9>

⁸ "Padé table - Wikipedia, the free encyclopedia." 2011. 6 May. 2016

<https://en.wikipedia.org/wiki/Pad%C3%A9_table>

⁹ McCabe, JH. "1 The Padé Table." 2007.

<http://www.mcs.st-andrews.ac.uk/pg/pure/Analysis/Papers/John_McCabe/Paper.pdf>

2.3 - Existing solutions

Both the methods above have been researched and implemented a lot in recent years. MatLab ¹⁰, GNU Octave ¹¹ and SciPy ¹² all currently offer versions of an optimised Padé Approximation, but these still don't make any leaps or bounds. This is because despite being parallelised, they still run on the CPU. This is likely a for compatibility reasons. The vast majority of people do not have access to dedicated computation devices and only a segment of those who do will have a CUDA compatible device.

This program will not run unless such a device is present. While you could consider this a drawback, those people without a device can still use the existing libraries out there. This is aimed at people who want to run complex algorithms over many simulation loops, so performance is key. On smaller problems, an improvement may not be noticeable and may even be slower than the CPU equivalent as a result of longer I/O times between the host and the device.

2.4 - How to optimise a program

Optimisation is often overlooked in everyday programming in favour of faster and more expensive hardware, but it is becoming more and more important in a world where our current processing techniques are limited by physics. While many people are researching alternative methods of computation (such as quantum computers themselves), we must find ways to optimise our current problems with existing technologies first. This is especially true if these optimisation problems help us to control quantum computers sooner.

There are several ways to optimise code. One such way is to include 'special cases' in your algorithm. These special cases will look for 'shortcuts' in your calculations. For example, the exponential of an identity matrix (I) is simply a matrix where the values of the diagonal cells equal e^1 . This can be calculated very easily and running the full algorithm on this matrix would be unnecessary and time consuming, not to mention, less accurate. Creating these special cases allows the program to decide which method of calculation it should use at runtime.

The method this program will focus on for optimising code is parallelisation. This takes on the concept of cores and threads running your code synchronously. This means that a processor can perform many calculations at once and feed the results

¹⁰ "MATLAB - MathWorks - MathWorks United Kingdom." 2014. 4 May. 2016
<<http://uk.mathworks.com/products/matlab/>>

¹¹ "GNU Octave." 2011. 4 May. 2016 <<https://www.gnu.org/software/octave/>>

¹² "SciPy.org — SciPy.org." 2011. 4 May. 2016 <<https://www.scipy.org/>>

back to a 'main thread' which will put the results together in the desired fashion. While this doing this on a CPU will speed up your results significantly, even the most extreme processors ¹³ still only have a number of cores in double digits. This is why we need a GPU for this project.

2.5 - What is a GPU?

GPUs are specialised processors with up to and beyond a thousand cores. These devices are designed to run specific calculations in parallel. Consumer GPUs are usually used for rendering videos and playing demanding video games, but they can also be used to solve complex mathematical equations.

In this project, I plan to use a graphics card to show a significant increase in performance compared to a standard processor while calculating matrix exponentials.

2.6 - Aims

The aim of this project is to decrease the computation time of a matrix exponential by increasing the performance and efficiency of the existing Padé Algorithm.

This will be achieved with the use of parallelisation and optimisation in C++ and CUDA on a dedicated graphics card. Timing experiments will need to be performed on an implementation of this idea and also on an existing CPU implementation such as MatLab. The comparison in speed between these two programs will determine the success of the project.

¹³ "Intel unveils 72-core x86 Knights Landing CPU for exascale ..." 2013. 4 May. 2016
<<http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing>>

3 - Design

3.1 - Environment

For this project I will be using two computers. Both need to have CUDA compatible graphics cards and be capable of running intensive algorithms. Below are the listed specifications for the machines I will be using

	Desktop PC	Laptop
Processor	Intel i7 4770K	Intel i7 4720HQ
Memory	16GB (2x8)	16GB (2x8)
Graphics Card(s)	2x Nvidia GTX 760 (2+2GB)	Nvidia GTX 980M (3GB)
OS	Windows 10 Pro 64bit	Windows 10 Pro 64bit
IDE	Visual Studio 2013 Pro	Visual Studio 2013 Pro
Graphics Library	CUDA 7.5	CUDA 7.5

Both of these machines are similar in specification, so I don't expect too much of a difference in runtime, however to keep everything consistent, all my timing experiments will be performed on the desktop PC.

In terms of software, CUDA 7.5 ¹⁴ is the most up-to-date library for communicating with Nvidia's graphics cards. Unfortunately, Visual Studio 2015 is not compatible with CUDA < 8.0 ¹⁵ and as this is unreleased at the time of writing, I am having to use a slightly older version (2013). This is not overly problematic as most of the 2015 features aren't significant to this project.

Visual Studio includes a free git repository which I will be using to sync my code between my two machines. This also gives me a reliable backup and version control which is very useful for referencing the changes made throughout the project. It also allows me to easily view my changes and revert anything if it goes wrong.

¹⁴ "CUDA Toolkit | NVIDIA Developer." 2011. 4 May. 2016 <<https://developer.nvidia.com/cuda-toolkit>>

¹⁵ "When will Visual Studio 2015 be supported? - NVIDIA Developer ..." 2015. 4 May. 2016
<<https://devtalk.nvidia.com/default/topic/883704/when-will-visual-studio-2015-be-supported-/>>

3.2 - Programming language

This program's primary objective is to be fast. For this reason C and C++ are great choices that offer a lot of depth and compatibility. CUDA is the obvious choice when it comes to GPU programming and is written for C and C++. Choosing between C and C++ is a simple matter of coding style and library availability.

While C supposedly allows you to do pretty much anything you want, it lacks the support that C++ has when it comes to libraries and only allows for basic functional programming. C++ introduced object orientation and classes which gives users of the program a much easier time integrating their own code.

Creating a Matrix Class allows others to extend the functionality or to use its methods for another purpose, but doesn't impact performance in any significant manner. Some would argue that its code is more readable too and doesn't add much to the complexity of the project.

3.3 - Handling Memory

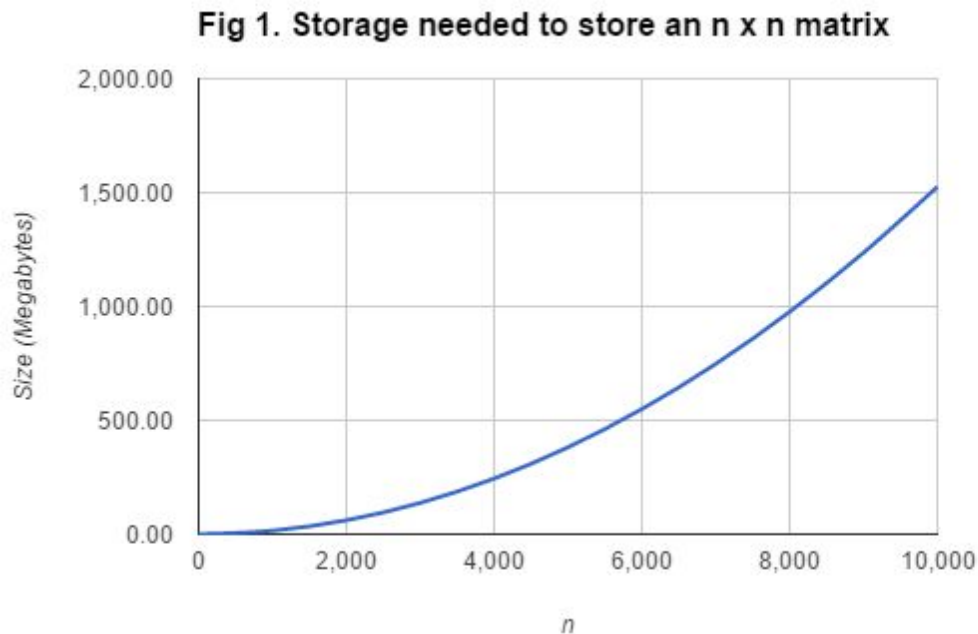
Deciding on a memory structure is quite fundamental to the way a program works. This project's main aim is to be efficient. This means reducing I/O times and memory utilisation. An easy way to do this is to make sure you're not passing the matrices around the program by value. In C++, passing by value makes a copy of the object in a new block of memory which is very inefficient. Instead we can pass matrices by reference. This means they are created once and then referred to by their address in other methods.

However, In order to perform calculations on the device (GPU), we must pass the matrix by copying its value. This is because the device cannot read from the host's memory (RAM) and instead has its own dedicated memory (VRAM). This should not be allocated at the same time as the host matrix as you should only keep the data you are currently processing on the GPU. Instead, you allocate the memory block when a method is called on that matrix. When the GPU completes its calculation, it returns the data to the host memory and deallocates the memory block to free it up for other data.

When designing a class efficiently, you also have to consider the structures you are using to store your data. C++ 11 provides some nice wrappers for vector arrays and strings, but CUDA only supports basic variable types such as *int* or *char*. For this reason, when creating a matrix class, it is better to use a pointer to an array of values than a C++ 11 wrapper that must be translated when allocating the GPU memory. Doubles are a good data type for this program as they have excellent

precision and are supported by the C++ `std::complex<>` type and CUDA's `thrust::complex<>` type.

Matrices will not always contain complex numbers, but they are commonly used together and all the methods in the program need to support them. Complex numbers are stored as two values - a real value and a complex value. If both values are stored as doubles (8 bytes) then each complex number uses 16 bytes. This means that while a 1000 x 1000 matrix only uses ~15.25 Megabytes of memory, a 10,000 x 10,000 matrix will use nearly 1.5 Gigabytes (see Figure 1). This is large enough that it could fill the memory on a modern graphics cards with just a few matrices. While this is a large problem, its only real solution is to split the matrices into smaller 'sub-matrices' and perform the calculations on those. This becomes complicated and messy very quickly, so the user of this program will need hardware that accommodates the size of their problem.



3.4 - Pseudocode

Below is the pseudocode that describes how some of the core aspects of my program work.

```
MatrixClass {

    // A series of constructors allow for users of the class to create various
    // matrices upon their creation. Each matrix is initialised using the input
    // variables and then set. If the users creates an empty matrix, initialised
    // is set to false and no memory is allocated.

    Constructor(input variables) {
        init()
        setMatrix()
    }

    // This initialiser sets the matrix properties and then allocates the
    // corresponding sized block of memory on the host.

    init(rows, cols) {
        Set rows, cols, els and size
        alloc()
        initialised = true
    }

    Destructor() {
        dealloc()
    }

    exampleMatrixMethod(input A, input B, output R) {
        if (input/output variables are initialised) {
            cudaAlloc(input A, input B)
            syncDevice(input A, input B)
            Get CUDA parameters
            Call CUDA kernel
            syncHost(output R)
            cudaDealloc(output R)
        }
    }

    // Other significant methods

    alloc()          // Allocates memory on the host
    dealloc()        // Deallocates memory on the host
    cudaAlloc()      // Allocates memory on the device
    cudaDealloc()    // Deallocates memory on the device
    syncDevice()     // Sync the input matrices memory from the host to the device
    syncHost()       // Sync the output matrix memory from the device to the host
}
```

3.5 - Class Diagram

CUDAMatrix

```
-----
- h_matrix: std::complex<double>*
- d_matrix: thrust::complex<double>*
- numRows: int
- numCols: int
- numEls: int
- size: size_t
- initialised: bool
-----
- alloc(): void
- dealloc(): void
- syncHost(): void
- syncDevice(): void
- getCUDAParams(): cudaParams
- getPadeParams(): padeParams
- ell(): int
- getPadeCoefficients(): std::vector<double>
-----
+ CUDAMatrix()
+ init(): void
+ ~CUDAMatrix()
-----
+ add(): CUDATimer // Addition
+ sub(): CUDATimer // Subtraction
+ mul(): CUDATimer // Multiplication
+ pow(): CUDATimer // Powers
+ tra(): CUDATimer // Transpose
+ inv(): CUDATimer // Inverse
+ exp(): CUDATimer // Exponential
+ abs(): CUDATimer // Absolute
-----
+ isInitialised(): bool
+ isSquare(): bool
+ isDiagonal(): bool
+ isIdentity(): bool
+ isZero(): bool
+ isSmall(): bool
+ isComplex(): bool
-----
+ setCell(): void
+ setMatrix(): void
+ setIdentity(): void
+ setRandom(): void
-----
+ getNorm(): double
+ getNormAm(): double
+ getCell(): std::complex<double>
+ getMatrix(): std::complex<double>*
+ getNumRows(): int
+ getNumCols(): int
+ getNumEls(): int
+ getSize(): size_t
```

CUDATimer

```
-----
- time: float
- t1: cudaEvent_t
- t2: cudaEvent_t
-----
+ start(): void
+ stop(): void
+ clear(): void
+ getTime(): float
```

4 - Implementation

4.1 - Sequential

Creating a sequential version of the program first is a good idea as it will help debug fundamental errors in the code. It also helps to lay out the concept of the program first without the complexities of CUDA.

4.1.1 - Header files (.h | .cuh)

The first thing that should be created when writing a well planned program is a header file. C++ (and CUDA) use this header file when someone calls or includes the class. It lists all the methods and variables available in its corresponding C++ file. C++ files are named with the extensions .cpp and .h for code and header files respectively. CUDA files use the extensions .cu and .cuh.

Listing all the program's methods in a header file at the beginning of the project is a good way to keep track of everything you need to implement. Visual Studio will also be able to read this and give you typing hints and suggestions.

4.1.2 - Code files (.cpp | .cu)

Code files are where the functional code should be. Once its header file is included, you can implement all the class methods and any other methods included in the header. There is no need for any class wrapper and it's already been declared. You simply have to refer to it when writing your method eg. `Classname::method() {}`

4.1.3 - Constructors & Destructors

Constructors are methods that create instances of classes or objects. They are designed so that you can feed them parameters which initialise the object. In this case the user of the class may want to set the size of the matrix upon its creation. They could also set the values manually or give it parse it an object to copy. The latter is called a copy constructor and is an efficient way of duplicating an instance of an object.

In this program, the constructor will firstly, initialise the properties of the matrix (rows, cols, els and size) and allocate the correct amount of memory to store the values. It will then copy the values from the input parameters to the allocated space or fill it with zeros if none are provided. If the constructor sends no parameters then no

memory is allocated and a flag (initialised) is set to false. This means that no methods can perform a calculation on something that has not been set yet.

A destructor does the opposite of a constructor. It is called when an object goes out of scope. This is usually when the block of code (denoted by curly braces '{}') the object was created in ends. This program will use the destructor to deallocate memory from the host. If this is not done, it can cause memory leaks.

4.1.4 - Getters, Setters & Booleans

These methods communicate data between the user and the class's private variables in a controlled manner. Using these methods instead of allowing direct access to variables means that errors are less likely to occur as validation can be performed. It also allows the class to hide some of its complexity. For example, this program uses a 1D array to store a matrix - something which is considered 2D. Instead of the user having to work out the index of a cell, they can call a setter which takes an x and a y value and does the maths for you. Not only does this make things easier, but it reduces code repetition and is more efficient.

There should be a getter for every piece of private data that the end user should be able to see. For the CUDAMatrix class, those variables are the numRows, numCols, numEls and size. A pointer to the host matrix should also be returnable, but there is no point in returning the device matrix pointer as it cannot be used in host code and will only ever be used from within the class.

There are no setters for the matrix properties in this class. This is because users should not be allowed to change it's size after values have been set. Doing this would mean recalculating each of the other properties and filling the extra spaces with zeros or trimming values. This is not a valid function and can be performed more cleanly by copying values within a range.

Setters are included to change the values of each cell or the entire matrix at once with a few different parameters available. The user of this class can also set the matrix to an identity matrix or assign it random values within a range.

Boolean methods return true or false statements about an object. Writing these methods helps to reduce code reproduction. In the CUDAMatrix class, it is also used to return the private initialised variable to the user.

4.1.5 - Matrix functions

Basic methods: `add()`, `sub()` & `mul()`

The design for this program has 3 basic matrix functions; `add()`, `sub()` and `mul()` for addition, subtraction and multiplication respectively. Each of these has 2 variants. A matrix-matrix calculation and a matrix-scalar calculation. The matrix-matrix methods take two input matrices and the matrix-scalar methods take an input matrix and a scalar value of the type `std::complex<double>`. All six methods store the result of the calculation in an output variable which is sent by reference to the function.

This gives us a performance advantage as the method does not have to return by value. As discussed earlier, parsing values around a program is much less efficient than creating an object in one place and then referring to it from elsewhere as the values don't have to be copied around memory.

The remaining functions are listed below:

`pow()` - The power function is a simple variation on the matrix-matrix multiplication method. It takes two input variables (A matrix and an integer) and one output variable (The resulting matrix). The original version of this function simply called the `mul()` method as many times as required, each time feeding in the response from the previous iteration as an input. This turned out to be largely inefficient as the program would check the initialisation (among other things) several times in one method. For this reason, the code was changed to a looped variation of the `mul()` method rather than calling it directly.

`tra()` - The transpose function is a relatively lightweight method which simply loops through each row of an input matrix and adds its values to the corresponding column in an output matrix. This is done with a nested loop (a for-loop in a for-loop). Using a nested loop instead of a single loop over each element makes it easier to index transposed cells. This is because in a 1D loop, the index must be calculated on each iteration whereas you can simply switch the x, y values around when using a 2D loop. While this method has a tiny bit more overhead, it saves in calculation time and makes the code easier to read.

`inv()` - This is by far the most computationally intensive, fundamental matrix function in this program. When calculating matrix exponentials, the inverse operation is needed to perform a matrix 'division' at the end. The matrix being input to this division will always be symmetric and therefore Cholesky's Decomposition method ¹⁶ is perfect. However, an LU Decomposition was used as it can handle non-symmetric

¹⁶ Krishnamoorthy, A. "Matrix Inversion Using Cholesky Decomposition." 2011.
<<https://arxiv.org/pdf/1111.4144>>

matrices and can therefore be used in other calculations. It is also only very marginally slower than the Cholesky method. Other algorithms such as Gaussian Elimination and Gauss-Jordan were considered, but both are more complex and do not perform as well as a decomposition method.¹⁷

This is the section of the program that the most time was spent on. The final program only includes a CPU implementation of the algorithm because CUDA was behaving in an unexpected manner. The complexity of the 3 loops were difficult to translate to a CUDA kernel and this part of the program may have been better served by including an existing library such as cuBLAS¹⁸ or Eigen.¹⁹

abs() - The absolute function is another very simple method that simply runs through each element of the input matrix and sets the corresponding element of the output matrix to its absolute value.

getNorm() - This method is actually a getter, but as it performs a calculation on the matrix, it could also be considered a matrix function. This is a relatively simple method that finds either the 1 norm or the euclidian norm depending on an input argument. To find the 1 norm, the algorithm sums up the absolute values of each column and find the maximum. The euclidian or 2 norm is a bit more complicated and isn't implemented in this version as it is not required to perform a matrix exponential. I used this webpage to learn how to do this.²⁰

In addition to all of the above, every the method must check to see whether the input and output variables are initialised before any calculation is performed. If one of them is not initialised, the program will throw an error and terminate. This is so that calculations aren't performed on memory that does not exist yet, or worse, memory that belongs to something else. All these calculations were checked with the Reshish²¹ and Wolframalpha²² calculators.

4.1.6 - Padé Approximation

The only matrix function not mentioned above is `exp()` - the exponential function. As previously discussed, there are a few ways to calculate this. The first method implemented was a taylor series expansion. This simply allowed an input matrix and

¹⁷ "LU Decomposition LU decomposition is a better way to implement ..." 2014. 5 May. 2016

<<http://www.cl.cam.ac.uk/teaching/1314/NumMethods/supporting/mcmaster-kiruba-ludecomp.pdf>>

¹⁸ "cuBLAS | NVIDIA Developer." 2011. 6 May. 2016 <<https://developer.nvidia.com/cublas>>

¹⁹ "Eigen." 2006. 6 May. 2016 <<http://eigen.tuxfamily.org/>>

²⁰ "Matrix norms." 2007. 6 May. 2016

<http://www.personal.soton.ac.uk/jav/soton/HELM/workbooks/workbook_30/30_4_matrix_norms.pdf>

²¹ "Reshish Matrix Calculator." 2012. 6 May. 2016 <<http://matrix.resish.com/>>

²² "Wolfram|Alpha: Computational Knowledge Engine." 2012. 6 May. 2016

<<https://www.wolframalpha.com/>>

an integer denoting how many terms the expansion should use (ie. how accurate it should be). It would then loop through each term and sum them all up. This method is simple, but computationally expensive and the point of this project is to speed this method up.

A padé approximation is the fastest, accurate method of calculating a matrix exponential. The MatLab implementation of the `expm()` method also uses this method, so this was a good place to start.

The first thing that the Padé algorithm does is to determine what the fastest method of calculation will be. If the input matrix is diagonal or zero then the answer is quickly retrieved and the rest of the algorithm is skipped. This saves a lot of time when working with simple matrices.

Next, it creates some temporary variables to store working data. These are all set to the same size as the input matrix and initialised. Once that is done, `getCUDAParams` and `getPadeParams` are both called. `getCUDAParams` is a simple private method that determines the grid and block size for the kernel depending on the size of the input matrix. `getPadeParams` works out the 'm value', the scaling needed and also stores some of the powers of the input matrix which are needed later and shouldn't be calculated twice.

The m-value determines which 'sum' the algorithm should perform. It works this out by finding the 1 norms of the input matrix and its powers and comparing them to an array of predetermined coefficients. It then looks up the calculation for the given m-value in the Padé Table mentioned earlier. When $m = 13$, the exponential is at its highest computational difficulty and will therefore take longer to calculate.

Just before the selected sum is performed, the algorithm checks the scaling values and applies the appropriate scaling to the input matrix and its powers. This makes the calculations slightly simpler to perform and reduces the possibility of rounding errors.

Each operation must be performed on the graphics card in the correct order. This is because, unlike ordinary numbers, matrices are immutable ($A * B \neq B * A$). This makes trying to run multiple calculations at once fairly difficult. Streams can help with this by running kernels alongside one another, but this is not implemented in this program. This is certainly an improvement that could be made to further improve performance time.

When the sum is completed, the result must be unscaled and synced back to the host. At this point the temporary variables are destroyed.

4.1.7 - Operator overrides and output

Operator overrides improve the usability of a class by its user. The CUDAMatrix class has a single override for the '<<' operator. This allows the user to place their matrix variable as if it were a string. For example they could type `std::cout << A;` and the program would output the contents of the matrix named 'A'. These overrides can also allow the use of arithmetic operators such as '+' or '*' though these were not implemented in the final version as they are slower than method calls.

4.2 - Parallel (CUDA)

With a sequential version of the program running correctly, most of the work is done. Adding CUDA to a C/C++ program doesn't change the layout or structure of the program, and infact, besides the change in filename extensions (.h and .cpp to .cu and .cu), the code can stay identical and the CUDA compiler will run it without a complaint. Having said this, getting CUDA to work in the first place is certainly not as easy as it could be.

4.2.1 - Installing CUDA

Installing software isn't hugely relevant to this report, but simply because of the amount of work that went into performing this seemingly simple task, I have included it anyway.

CUDA is written and maintained by Nvidia for use of their consumer and professional grade cards and is designed to be compatible with Microsoft's Visual Studio. Infact it is quite difficult to avoid using this combination of software as there is a distinct lack of support for any other programming IDEs. Theoretically, all you have to do is install Visual Studio and then install the CUDA toolkit that corresponds to your Visual Studio version - in that order. In my case, this was Visual Studio 2013 Pro and CUDA 7.5.

For whatever reason, this decided not to work on the first try and despite reinstalling and playing around with environment variables and project properties for hours, It would not run.

Eventually I removed everything from the computer and started from scratch while following the guide referenced here.²³ Finally, this worked, although only on my desktop. Not on my laptop.

²³ "Machinomics: Setting up CUDA in Windows and Visual C++ 2008 ..." 2012. 5 May. 2016
<<http://machinomics.blogspot.com/2012/04/setting-up-cuda-in-windows-and-visual-c.html>>

4.2.2 - Significant Changes

As stated above, most of the code stayed identical in this version of the program, however there are a couple of fundamental changes which should be noted here. Firstly, we now have to handle the memory space of the device (graphics card) as well as our host memory. In order to keep memory utilisation as low as possible, matrices should only be allocated when they are needed (eg. at the beginning of each matrix function). This also means deallocating them when functions finish and the result has been returned to the host's memory.

Secondly we must create an instance of the CUDATimer class with each method. This allows the execution time of the method to be returned by using CUDA's built in libraries.

Kernels are used run a calculation on the graphics card. These kernels are global methods that are not included in the class structure. hold the code that will run on the device. They can only access variables stored on the device and they can only use types and methods that are supported in CUDA.

For the most part, Thrust (A standard library included with CUDA), ²⁴ has everything you need as far as data types and structures are concerned. It provides a version of C++'s complex class which is fully compatible and requires no work to get them talking to one another.

²⁴ "Thrust :: CUDA Toolkit Documentation - NVIDIA Documentation." 2013. 5 May. 2016
<<http://docs.nvidia.com/cuda/thrust/>>

5 - Numerical Experiments

5.1 - Method

To collect timing data for MATLAB's `expm()` function, I created a small script (see below) that would perform a matrix exponential on a random matrix in a loop and then average the result. I ran each of these a few times to make sure that the random matrix wasn't computationally simple in any way. For matrices where $n < 1000$, I used 1000 loops. For matrices where $n > 1000$, I could only manage about 10 loops before computation time became excessive. This still gives a good idea of how long the method takes to run. I then created an identical program in C++/CUDA.

5.2 - Testing scripts

5.2.1 - Matlab

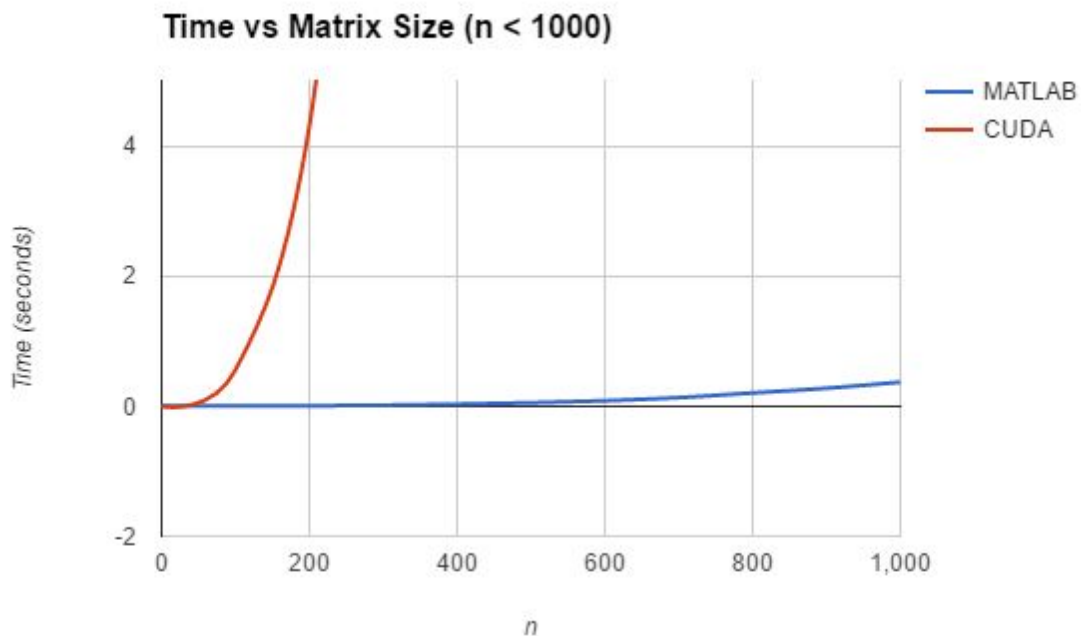
```
n = 100;
A = rand(n);
tic
for i = 1:1000
    expm(A);
end
t = toc/1000
```

5.2.2 - C++/CUDA

```
n = 100;
CUDAMatrix A(n);
CUDAMatrix eA(n);
A.setRandomDoulbe();
for (int c1 = 0; c1 < 1000; c1++) {
    t = CUDAMatrix exp(A, eA);
    total += t.getTime();
}
total /= 100;
```

5.3 - Results

My first set of findings were quite alarming and it became apparent after the first couple of readings that something was wrong. The graph below clearly shows that MatLab is significantly faster at all sizes of n .



After some digging through code, it became apparent that the inverse method was the cause of the slow speed. However running the code without the inverse method gives roughly the same time no matter what size of matrix you use which is obviously incorrect. These timing experiments do not look accurate at all and it's therefore inappropriate to include them here.

Smaller matrices run too fast to do a visual inspection of runtime, but by using larger matrices, we can visibly see the time it takes the program to execute. This program is at least as fast as MatLab on medium sized matrices ($500 < n < 1000$) and is visibly faster on large matrices (up to $n = 8000$) by up to a second. While these findings aren't particularly scientific, the results are encouraging.

When doing a visual inspection on smaller matrices, I used a loop to exaggerate the runtime (Performing the same calculation many times). It appeared that for these matrices, MatLab was faster, by up to a couple of seconds. The number of loops used should have scaled to the same timeframe as the larger matrix tests, but the overhead involved is obviously too much to handle on small matrices and the speed up is not of enough significance to be of use.

6 - Future Work

There are many ways this program could and should be improved in the future. The inverse method's performance is severely hampered because it still runs on the CPU. Creating a CUDA method that does this instead would greatly decrease processing time and make syncing the device and host unnecessary in the middle of a method. This syncing is what makes this part of the program so slow compared with MatLab.

The transpose function is another method that runs on the CPU. While finding the transpose isn't very expensive on a small matrix, these calculations can add up fast when used consecutively or when their size grows larger. Rewriting this in CUDA would be a relatively easy addition to the program as it's a simple loop that accesses each array element independently.

There is also performance to be found by using more special cases. The exponential function makes good use of special cases for zero and diagonal matrices, but there are many other methods that could have 'shortcuts' in their computation. For example, the inverse method is much simpler when performed on a scalar matrix. Each diagonal element simply inverts itself independently. While this is a relatively rare case, you don't want the program performing unnecessary calculations.

The inverse function could also check if the matrix is symmetric prior to calculation and run a Cholesky Decomposition instead of an LU Decomposition if it is. The overhead on this may be larger than the gain in performance. This is something that would have to be tested.

The getNorm method could be improved using a getNormAm method in certain cases where the result needs to be achieved fast instead of accurately. MatLab's CPU architecture uses a version of this which is well documented here.²⁵

The final major improvement that would benefit this program is the use of CUDA's streams. This lets you run several kernels at the same time. When you have a high quantity of small matrices, this is very useful and allows for a higher utilisation of the graphics card. Besides this, CUDA is a very technical language and there are always tiny improvements to be made in how the kernel is called and how much processing power it is assigned. Most of these improvements would be made through trial and error and would again, need to be tested.

²⁵ "normAm(A,m) - File Exchange - MATLAB Central." 2011. 6 May. 2016

<<http://www.mathworks.com/matlabcentral/fileexchange/29576-matrix-exponential-times-a-vector/content/normAm.m>>

7 - Conclusion

In conclusion, for the most part, this project has been successful. With a bit more time and research, this program does have the ability to be fast. Very fast. My consumer grade graphics card is not the best example of a GPU and professional equipment would almost certainly improve these results. The program shows that you can indeed increase the performance of a matrix exponential by running the calculation on a dedicated hardware device.

The program shows an improvement in performance across most of the available matrix functions and there is plenty of space for further optimisations.

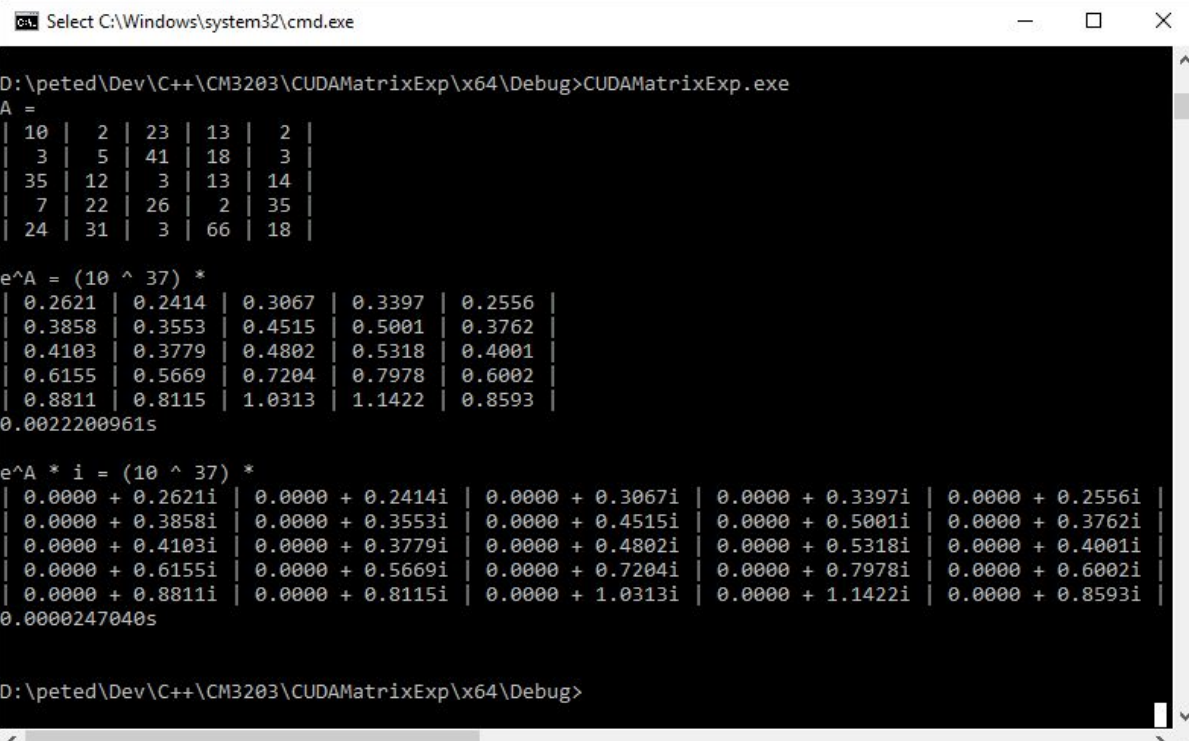
However, some parts of the project are still slow, including the fairly fundamental inverse algorithm. This method is performed at the end of every matrix exponential and slows down performance greatly. This is, in part, due to the parallel implementation of the code not working correctly. As a result, the sequential CPU version is implemented instead. This combined with the I/O times of transferring the data back to the CPU and then back to the GPU again increases computation time greatly.

With some small tweaks and fine tuning this program could prove very helpful in the hunt for better, faster and more accurate simulations of quantum effects. It could help to reduce the intensity of algorithms relating to quantum control on the CPU, but a bit more work would be required to get it there and a few simple improvements could go a long way towards this.

8 - Appendix

8.1 - Screenshots

Example program output. See main.cpp in 8.2.5



```
Select C:\Windows\system32\cmd.exe

D:\peted\Dev\C++\CM3203\CUDAMatrixExp\x64\Debug>CUDAMatrixExp.exe
A =
| 10 | 2 | 23 | 13 | 2 |
| 3 | 5 | 41 | 18 | 3 |
| 35 | 12 | 3 | 13 | 14 |
| 7 | 22 | 26 | 2 | 35 |
| 24 | 31 | 3 | 66 | 18 |

e^A = (10 ^ 37) *
| 0.2621 | 0.2414 | 0.3067 | 0.3397 | 0.2556 |
| 0.3858 | 0.3553 | 0.4515 | 0.5001 | 0.3762 |
| 0.4103 | 0.3779 | 0.4802 | 0.5318 | 0.4001 |
| 0.6155 | 0.5669 | 0.7204 | 0.7978 | 0.6002 |
| 0.8811 | 0.8115 | 1.0313 | 1.1422 | 0.8593 |
0.0022200961s

e^A * i = (10 ^ 37) *
| 0.0000 + 0.2621i | 0.0000 + 0.2414i | 0.0000 + 0.3067i | 0.0000 + 0.3397i | 0.0000 + 0.2556i |
| 0.0000 + 0.3858i | 0.0000 + 0.3553i | 0.0000 + 0.4515i | 0.0000 + 0.5001i | 0.0000 + 0.3762i |
| 0.0000 + 0.4103i | 0.0000 + 0.3779i | 0.0000 + 0.4802i | 0.0000 + 0.5318i | 0.0000 + 0.4001i |
| 0.0000 + 0.6155i | 0.0000 + 0.5669i | 0.0000 + 0.7204i | 0.0000 + 0.7978i | 0.0000 + 0.6002i |
| 0.0000 + 0.8811i | 0.0000 + 0.8115i | 0.0000 + 1.0313i | 0.0000 + 1.1422i | 0.0000 + 0.8593i |
0.0000247040s

D:\peted\Dev\C++\CM3203\CUDAMatrixExp\x64\Debug>
```

The same program as above in MatLab.

```
Command Window
New to MATLAB? See resources for Getting Started.

>> A

A =

    10     2    23    13     2
     3     5    41    18     3
    35    12     3    13    14
     7    22    26     2    35
    24    31     3    66    18

>> eA = expm(A)

eA =

    1.0e+37 *

    0.2621    0.2414    0.3067    0.3397    0.2556
    0.3858    0.3553    0.4515    0.5001    0.3762
    0.4103    0.3779    0.4802    0.5318    0.4001
    0.6155    0.5669    0.7204    0.7978    0.6002
    0.8811    0.8115    1.0313    1.1422    0.8593

>> eAi = eA * complex(0, 1)

eAi =

    1.0e+37 *

    0.0000 + 0.2621i    0.0000 + 0.2414i    0.0000 + 0.3067i    0.0000 + 0.3397i    0.0000 + 0.2556i
    0.0000 + 0.3858i    0.0000 + 0.3553i    0.0000 + 0.4515i    0.0000 + 0.5001i    0.0000 + 0.3762i
    0.0000 + 0.4103i    0.0000 + 0.3779i    0.0000 + 0.4802i    0.0000 + 0.5318i    0.0000 + 0.4001i
    0.0000 + 0.6155i    0.0000 + 0.5669i    0.0000 + 0.7204i    0.0000 + 0.7978i    0.0000 + 0.6002i
    0.0000 + 0.8811i    0.0000 + 0.8115i    0.0000 + 1.0313i    0.0000 + 1.1422i    0.0000 + 0.8593i

fx >> |
```

8.2 - Code

8.2.1 - CUDAMatrix.cu

```
//
// Cardiff University | Computer Science
// Module:      CM3203 One Semester Project (40 Credits)
// Title:       Parallelisation of Matrix Exponentials in C++/CUDA for Quantum Control
// Date:        2016
//
// Author:      Peter Davison
// Supervisor:  Dr. Frank C Langbein
// Moderator:   Dr. Irena Spasic
//

// Include header file
#include "CUDAMatrix.cuh"

// KERNELS

__global__ void cudaAdd(thrust::complex<double>* A, thrust::complex<double>* B,
thrust::complex<double>* R, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n && col < n) {
        R[row * n + col] = A[row * n + col] + B[row * n + col];
    }
    __syncthreads();
}

__global__ void cudaAddScalar(thrust::complex<double>* A, thrust::complex<double> scalar,
thrust::complex<double>* R, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n && col < n) {
        R[row * n + col] = A[row * n + col] + scalar;
    }
    __syncthreads();
}

__global__ void cudaSub(thrust::complex<double>* A, thrust::complex<double>* B,
thrust::complex<double>* R, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n && col < n) {
        R[row * n + col] = A[row * n + col] - B[row * n + col];
    }
    __syncthreads();
}

__global__ void cudaSubScalar(thrust::complex<double>* A, thrust::complex<double> scalar,
thrust::complex<double>* R, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
```

```

        int col = blockIdx.x * blockDim.x + threadIdx.x;
        if (row < n && col < n) {
            R[row * n + col] = A[row * n + col] - scalar;
        }
        __syncthreads();
    }

__global__ void cudaMul(thrust::complex<double>* A, thrust::complex<double>* B,
thrust::complex<double>* R, int n) {
    thrust::complex<double> sum = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n && col < n) {
        for (int i = 0; i < n; i++) {
            sum += A[row * n + i] * B[i * n + col];
        }
    }
    R[row * n + col] = sum;
    __syncthreads();
}

__global__ void cudaMulScalar(thrust::complex<double>* A, thrust::complex<double> scalar,
thrust::complex<double>* R, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n && col < n) {
        R[row * n + col] = A[row * n + col] * scalar;
    }
    __syncthreads();
}

__global__ void cudaAbs(thrust::complex<double>* A, thrust::complex<double>* R, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < n && col < n) {
        R[row * n + col] = abs(A[row * n + col]);
    }
    __syncthreads();
}

// MEMORY HANDLERS

void CUDAMatrix::alloc() {
    h_matrix = (std::complex<double>*) malloc(size);
    cudaError_t result = cudaMalloc((void**) &d_matrix, size);
    if (result != cudaSuccess) {
        throw std::runtime_error("Failed to allocate device memory");
    }
}

void CUDAMatrix::dealloc() {
    free(h_matrix);
    cudaError_t result = cudaFree(d_matrix);
    if (result != cudaSuccess) {

```

```

        throw std::runtime_error("Failed to free device memory");
    }
}

// CUDA STUFF

void CUDAMatrix::syncHost() {
    if (isInitialised()) {
        cudaError_t result = cudaMemcpy(h_matrix, d_matrix, size,
cudaMemcpyDeviceToHost);
        if (result != cudaSuccess) {
            throw std::runtime_error("Failed to allocate device memory");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::syncDevice() {
    if (isInitialised()) {
        cudaError_t result = cudaMemcpy(d_matrix, h_matrix, size,
cudaMemcpyHostToDevice);
        if (result != cudaSuccess) {
            throw std::runtime_error("Failed to allocate device memory");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

CUDAMatrix::cudaParams CUDAMatrix::getCUDAParams(int rows, int cols) {
    cudaParams cp;
    cp.tpb = dim3(rows, cols);
    cp.bpg = dim3(1, 1);
    if (rows*cols > 512) {
        cp.tpb.x = 512;
        cp.tpb.y = 512;
        cp.bpg.x = (int) (ceil(double(rows) / double(cp.tpb.x)));
        cp.bpg.y = (int) (ceil(double(cols) / double(cp.tpb.y)));
    }
    return cp;
}

// INTERNAL PADE APPROXIMATION CODE

int CUDAMatrix::ell(CUDAMatrix& A, double coef, int m) {
    CUDAMatrix sA(A.getNumRows());
    CUDAMatrix::abs(A, sA);
    double scale = std::pow(coef, (1 / (double) (2 * m + 1)));
    CUDAMatrix::mul(sA, scale, sA);
    //double alpha = sA.getNormAm(2 * m + 1) / A.getNorm(1);    2 LINES BELOW ARE
    TEMPORARY REPLACEMENT

```

```

        CUDAMatrix::pow(sA, (2 * m + 1), sA);
        double alpha = sA.getNorm(1) / (double) (A.getNorm(1));
        /////
        return utils::max((int) (ceil(log2(2 * alpha /
std::numeric_limits<double>::epsilon()) / (2 * m))), 0);
    }

CUDAMatrix::padeParams CUDAMatrix::getPadeParams(CUDAMatrix& A) {
    // Init
    double d4, d6, d8, d10, eta1, eta3, eta4, eta5;
    int ar = A.getNumRows();
    int ac = A.getNumCols();
    std::vector<double> theta;
    std::vector<double> coef;
    // Init P;
    padeParams p;
    p.pow.resize(11);
    p.scale = 0;
    // Get coefficients and theta values
    coef = {
        (1 / 100800.0),
        (1 / 10059033600.0),
        (1 / 4487938430976000.0),
        (1 / 5914384781877411840000.0),
        (1 / 113250775606021113483283660800000000.0)
    };
    theta = {
        1.495585217958292e-002,
        2.539398330063230e-001,
        9.504178996162932e-001,
        2.097847961257068e+000,
        5.371920351148152e+000
    };
    // Get powers of A
    p.pow[2] = new CUDAMatrix(ar, ac);
    p.pow[4] = new CUDAMatrix(ar, ac);
    p.pow[6] = new CUDAMatrix(ar, ac);
    p.pow[8] = new CUDAMatrix(ar, ac);
    p.pow[10] = new CUDAMatrix(ar, ac);
    cudaParams cp = getCUDAParams(A.getNumRows(), A.getNumCols());
    cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, A.d_matrix, p.pow[2]->d_matrix,
ar);
        cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (p.pow[2]->d_matrix, p.pow[2]->d_matrix,
p.pow[4]->d_matrix, ar);
        cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (p.pow[2]->d_matrix, p.pow[4]->d_matrix,
p.pow[6]->d_matrix, ar);
        cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (p.pow[4]->d_matrix, p.pow[4]->d_matrix,
p.pow[8]->d_matrix, ar);
        cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (p.pow[4]->d_matrix, p.pow[6]->d_matrix,
p.pow[10]->d_matrix, ar);

    // NOT IDEAL .. PERFORM GETNORM ON DEVICE IF POSSIBLE. THIS MEANS SYNCING BETWEEN
    HOST AND DEVICE IS UNNECESSARY
    p.pow[2]->syncHost();

```

```

p.pow[4]->syncHost();
p.pow[6]->syncHost();
p.pow[8]->syncHost();
p.pow[10]->syncHost();
////

// Find mVal
d4 = std::pow(p.pow[4]->getNorm(1), (1.0 / 4));
d6 = std::pow(p.pow[6]->getNorm(1), (1.0 / 6));
eta1 = utils::max(d4, d6);
if ((eta1 <= theta[0]) && (ell(A, coef[0], 3) == 0)) {
    p.mVal = 3;
    return p;
}
if ((eta1 <= theta[1]) && (ell(A, coef[1], 5) == 0)) {
    p.mVal = 5;
    return p;
}
if (true) { //(A.isSmall()) {
    d8 = std::pow(p.pow[8]->getNorm(1), (1.0 / 8));
} else {
    //d8 = pow(p.pow[4]->getNormAm(2), (1.0 / 8));
}
eta3 = utils::max(d6, d8);
if ((eta3 <= theta[2]) && (ell(A, coef[2], 7) == 0)) {
    p.mVal = 7;
    return p;
}
if ((eta3 <= theta[3]) && (ell(A, coef[3], 9) == 0)) {
    p.mVal = 9;
    return p;
}
if (true) { //(A.isSmall()) {
    d10 = std::pow(p.pow[10]->getNorm(1), (1.0 / 10));
} else {
    //d10 = std::pow(p.pow[2]->getNormAm(5), (1.0 / 10));
}
// Find scaling factor
eta4 = utils::max(d8, d10);
eta5 = utils::min(eta3, eta4);
p.scale = utils::max((int) (ceil(log2(eta5 / theta[4]))), 0);
CUDAMatrix sA(ar, ac);
double multiplier = 1.0 / std::pow(2, p.scale);
CUDAMatrix::mul(A, multiplier, sA);
p.scale += ell(sA, coef[4], 13);
if (std::isinf((double) p.scale)) {
    std::cout << "S = INF" << std::endl;
    int exp;
    double t = std::frexp(A.getNorm(1) / theta[4], &exp);
    p.scale = exp - (t == 0.5);
}
p.mVal = 13;
return p;
}

```

```

std::vector<double> CUDAMatrix::getPadeCoefficients(int m) {
    switch (m) {
        case 3:
            return { 120, 60, 12, 1 };
        case 5:
            return { 30240, 15120, 3360, 420, 30, 1 };
        case 7:
            return { 17297280, 8648640, 1995840, 277200, 25200, 1512, 56, 1 };
        case 9:
            return { 17643225600, 8821612800, 2075673600, 302702400, 30270240,
2162160, 110880, 3960, 90, 1 };
        case 13:
            return { 64764752532480000, 32382376266240000, 7771770303897600,
1187353796428800, 129060195264000, 10559470521600, 670442572800, 33522128640, 1323241920,
40840800, 960960, 16380, 182, 1 };
        default:
            throw std::runtime_error("Invalid m value");
    }
}

// CONSTRUCTORS

CUDAMatrix::CUDAMatrix() {
    initialised = false;
}

CUDAMatrix::CUDAMatrix(int inNumRowsCols) {
    init(inNumRowsCols, inNumRowsCols);
    setMatrix(0.0);
}

CUDAMatrix::CUDAMatrix(int inNumRows, int inNumCols) {
    init(inNumRows, inNumCols);
    setMatrix(0.0);
}

CUDAMatrix::CUDAMatrix(int inNumRowsCols, std::initializer_list<std::complex<double>>
inMatrix) {
    if (inMatrix.size() == inNumRowsCols*inNumRowsCols) {
        init(inNumRowsCols, inNumRowsCols);
        setMatrix(inMatrix);
    } else {
        throw std::runtime_error("Initialiser-list size does not match matrix
size");
    }
}

CUDAMatrix::CUDAMatrix(int inNumRows, int inNumCols,
std::initializer_list<std::complex<double>> inMatrix) {
    if (inMatrix.size() == inNumRows*inNumCols) {
        init(inNumRows, inNumCols);
        setMatrix(inMatrix);
    } else {

```



```

        throw std::runtime_error("Initialiser-list size does not match matrix
size");
    }
}

CUDAMatrix::CUDAMatrix(const CUDAMatrix &obj) {
    if (obj.initialised) {
        h_matrix = obj.h_matrix;
        d_matrix = obj.d_matrix;
        numRows = obj.numRows;
        numCols = obj.numCols;
        numEls = obj.numEls;
        size = obj.size;
        initialised = obj.initialised;
    } else {
        throw std::runtime_error("Cannot copy uninitialised matrix");
    }
}

void CUDAMatrix::init(int inNumRows, int inNumCols) {
    numRows = inNumRows;
    numCols = inNumCols;
    numEls = inNumRows*inNumCols;
    size = sizeof(std::complex<double>) * numEls;
    alloc();
    initialised = true;
}

CUDAMatrix::~CUDAMatrix() {
    dealloc();
}

// MATRIX OPERATIONS

CUDATimer CUDAMatrix::add(CUDAMatrix& A, CUDAMatrix& B, CUDAMatrix& R) {
    if (A.isInitialised() && B.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int br = B.getNumRows();
        int bc = B.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == br && br == bc && bc == rr && rr == rc) {
            A.syncDevice();
            B.syncDevice();

            cudaParams cp = getCUDAParams(ar, ac);
            CUDATimer t;

            t.start();
            cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, B.d_matrix,
R.d_matrix, A.getNumRows());
            t.stop();

```

```

        R.syncHost();
        return t;
    } else {
        throw std::runtime_error("Matrix sizes do not match");
    }
} else {
    throw std::runtime_error("Cannot perform matrix operations before
initialisation");
}
}

CUDATimer CUDAMatrix::add(CUDAMatrix& A, std::complex<double> scalar, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == rr && rr == rc) {
            A.syncDevice();

            cudaParams cp = getCUDAParams(ar, ac);
            CUDATimer t;

            t.start();
            cudaAddScalar KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, scalar,
R.d_matrix, A.getNumRows());
            t.stop();

            R.syncHost();
            return t;
        } else {
            throw std::runtime_error("Matrix sizes do not match");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

CUDATimer CUDAMatrix::sub(CUDAMatrix& A, CUDAMatrix& B, CUDAMatrix& R) {
    if (A.isInitialised() && B.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int br = B.getNumRows();
        int bc = B.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == br && br == bc && bc == rr && rr == rc) {
            A.syncDevice();
            B.syncDevice();

            cudaParams cp = getCUDAParams(ar, ac);
            CUDATimer t;

```

```

        t.start();
        cudaSub KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, B.d_matrix,
R.d_matrix, A.getNumRows());
        t.stop();

        R.syncHost();
        return t;
    } else {
        throw std::runtime_error("Matrix sizes do not match");
    }
} else {
    throw std::runtime_error("Cannot perform matrix operations before
initialisation");
}
}

```

```

CUDATimer CUDAMatrix::sub(CUDAMatrix& A, std::complex<double> scalar, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == rr && rr == rc) {
            A.syncDevice();

            cudaParams cp = getCUDAParams(ar, ac);
            CUDATimer t;

            t.start();
            cudaSubScalar KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, scalar,
R.d_matrix, A.getNumRows());
            t.stop();

            R.syncHost();
            return t;
        } else {
            throw std::runtime_error("Matrix sizes do not match");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

```

```

CUDATimer CUDAMatrix::mul(CUDAMatrix& A, CUDAMatrix& B, CUDAMatrix& R) {
    if (A.isInitialised() && B.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int br = B.getNumRows();
        int bc = B.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == br && br == bc && bc == rr && rr == rc) {
            A.syncDevice();

```

```

        B.syncDevice();

        cudaParams cp = getCUDAParams(ar, ac);
        CUDATimer t;

        t.start();
        cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, B.d_matrix,
R.d_matrix, A.getNumRows());
        t.stop();

        R.syncHost();
        return t;
    } else {
        throw std::runtime_error("Matrix sizes do not match");
    }
} else {
    throw std::runtime_error("Cannot perform matrix operations before
initialisation");
}
}

CUDATimer CUDAMatrix::mul(CUDAMatrix& A, std::complex<double> scalar, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == rr && rr == rc) {
            A.syncDevice();

            cudaParams cp = getCUDAParams(ar, ac);
            CUDATimer t;

            t.start();
            cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, scalar,
R.d_matrix, A.getNumRows());
            t.stop();

            R.syncHost();
            return t;
        } else {
            throw std::runtime_error("Matrix sizes do not match");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

CUDATimer CUDAMatrix::pow(CUDAMatrix& A, int pow, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();

```

```

        int rc = R.getNumCols();
        if (ar == ac && ac == rr && rr == rc) {
            A.syncDevice();
            CUDAMatrix T(ar);
            T.setIdentity();
            T.syncDevice();

            cudaParams cp = getCUDAParams(ar, ac);
            CUDATimer t;

            t.start();
            for (int c1 = 0; c1 < pow; c1++) {
                cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, T.d_matrix,
T.d_matrix, ar);
            }
            t.stop();

            T.syncHost();
            R.setMatrix(T.getMatrix());
            return t;
        } else {
            throw std::runtime_error("Matrix sizes do not match");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

```

```

CUDATimer CUDAMatrix::inv(CUDAMatrix& A, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == rr && rr == rc) {

            CUDATimer t;
            CUDAMatrix L = CUDAMatrix(ar, ac);
            CUDAMatrix U = CUDAMatrix(ar, ac);
            CUDAMatrix Z = CUDAMatrix(ar, ac);
            CUDAMatrix I = CUDAMatrix(ar, ac);
            I.setIdentity();

            t.start();

            int n = ar;
            int i, j, k;
            // LU Decomposition
            for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                    if (j < i) {
                        U.setCell(i, j, 0);
                    } else {

```

```

        U.setCell(i, j, A.getCell(i, j));
        for (k = 0; k < i; k++) {
            U.setCell(i, j, (U.getCell(i, j) -
U.getCell(k, j) * L.getCell(i, k)));
        }
    }
    for (j = 0; j < n; j++) {
        if (j < i) {
            L.setCell(j, i, 0);
        } else if (j == i) {
            L.setCell(j, i, 1);
        } else {
            L.setCell(j, i, (A.getCell(j, i) / U.getCell(i,
i)));
            for (k = 0; k < i; k++) {
                L.setCell(j, i, (L.getCell(j, i) -
((U.getCell(k, i) * L.getCell(j, k)) / U.getCell(i, i))));
            }
        }
    }
    for (i = 0; i < n; i++) {
        // Find Z ( $L^{-1}$ ) with Forward Substitution
        for (j = 0; j < n; j++) {
            Z.setCell(j, i, I.getCell(j, i));
            for (k = 0; k < n; k++) {
                if (k != j) {
                    Z.setCell(j, i, (Z.getCell(j, i) -
(L.getCell(j, k) * Z.getCell(k, i))));
                }
            }
        }
        // Find X ( $A^{-1}$ ) with Backward Substitution
        for (j = n - 1; j >= 0; j--) {
            R.setCell(j, i, Z.getCell(j, i));
            for (k = 0; k < n; k++) {
                if (k != j) {
                    R.setCell(j, i, (R.getCell(j, i) -
(U.getCell(j, k) * R.getCell(k, i))));
                }
            }
            R.setCell(j, i, R.getCell(j, i) / U.getCell(j, j));
        }
    }

    t.stop();
    return t;
} else {
    throw std::runtime_error("Matrix sizes do not match");
}
} else {
    throw std::runtime_error("Cannot perform matrix operations before
initialisation");
}

```

```

    }
}

CUDATimer CUDAMatrix::tra(CUDAMatrix& A, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ac == rr) {
            A.syncDevice();

            int c1, c2;
            CUDATimer t;

            t.start();
            for (c1 = 0; c1 < A.getNumRows(); c1++) {
                for (c2 = 0; c2 < A.getNumCols(); c2++) {
                    R.setCell(c1, c2, A.getCell(c2, c1));
                }
            }
            t.stop();

            R.syncDevice();
            return t;
        } else {
            throw std::runtime_error("Transpose matrix is the wrong size");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

CUDATimer CUDAMatrix::exp(CUDAMatrix& A, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == rr && rr == rc) {
            A.syncDevice();
            CUDATimer t;
            int c1, c2;
            int n = utils::max(ar, ac);
            // Special Cases
            if (A.isDiagonal()) {
                t.start();
                for (c1 = 0; c1 < n; c1++) {
                    R.setCell(c1, c1, std::exp(A.getCell(c1, c1)));
                }
                t.stop();
                R.syncDevice();
            } else if (A.isZero()) {

```

```

        t.start();
        R.setMatrix(0);
        t.stop();
        R.syncDevice();
    // Normal Case
} else {
    // Create Matrices
    CUDAMatrix U(ar, ac);
    CUDAMatrix V(ar, ac);
    CUDAMatrix I(ar, ac); // Identity
    CUDAMatrix T(ar, ac); // Tally
    CUDAMatrix TMP(ar, ac); // Temporary
    I.setIdentity();
    I.syncDevice();
    // Get CUDA params
    cudaParams cp = getCUDAParams(ar, ac);
    // Get Pade params
    padeParams p = getPadeParams(A);
    int s = p.scale;
    int m = p.mVal;
    std::vector<CUDAMatrix*> pow = p.pow;
    // Get Pade coefficients
    std::vector<double> c = getPadeCoefficients(m);
    // Start timer
    t.start();
    // Scaling
    if (s != 0) {
        double multiplier;
        multiplier = 1.0 / std::pow(2, s);
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(A.d_matrix, multiplier, A.d_matrix, n);
        for (c1 = 2; c1 <= 6; c1 += 2) {
            multiplier = 1.0 / std::pow(2, (s * c1));
            cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[c1]->d_matrix, multiplier, pow[c1]->d_matrix, n);
        }
    }
    // Approximation
    if (m == 3 || m == 5 || m == 7 || m == 9) {
        for (c1 = (int) (pow.size()) + 2; c1 < m - 1; c1 += 2)
{ //for (k = strt:2:m-1)
            cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (pow[c1 -
2]->d_matrix, pow[2]->d_matrix, pow[c1]->d_matrix, n);
        }
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(I.d_matrix, c[1], U.d_matrix, n);
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(I.d_matrix, c[0], V.d_matrix, n);
        for (c2 = m; c2 >= 3; c2 -= 2) { //for (j = m : -2 :
3)
            cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[c2 - 1]->d_matrix, c[c2], TMP.d_matrix, n);
            cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb)
(U.d_matrix, TMP.d_matrix, U.d_matrix, n);

```



```

                                cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[c2 - 1]->d_matrix, c[c2-1], TMP.d_matrix, n);
                                cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb)
(V.d_matrix, TMP.d_matrix, V.d_matrix, n);
                                }
                                cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (U.d_matrix,
A.d_matrix, U.d_matrix, n);
                                } else if (m == 13) {
                                    // This is the equivalent of ..
                                    // U = A * (p[6] * (c[13] * p[6] + c[11] * p[4] + c[9]
* p[2]) + c[7] * p[6] + c[5] * p[4] + c[3] * p[2] + c[1] * I);          RUN IN STREAM 1
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[6]->d_matrix, c[13], T.d_matrix, n);          // p[6] * c[13] -> T
Needs new TMP var
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[4]->d_matrix, c[11], TMP.d_matrix, n);          // p[4] * c[11] -> TMP
                                    (Cannot be used in multiple streams)
                                    cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);          // T + TMP -> T
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[2]->d_matrix, c[9], TMP.d_matrix, n);          // p[2] * c[9] -> TMP
                                    cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);          // T + TMP -> T
                                    cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[6]->d_matrix, T.d_matrix, T.d_matrix, n);          // p[6] * T -> T
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[6]->d_matrix, c[7], TMP.d_matrix, n);          // p[6] * c[7] -> TMP
                                    cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);          // T + TMP -> T
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[4]->d_matrix, c[5], TMP.d_matrix, n);          // p[4] * c[5] -> TMP
                                    cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);          // T + TMP -> T
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[2]->d_matrix, c[3], TMP.d_matrix, n);          // p[2] * c[3] -> TMP
                                    cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);          // T + TMP -> T
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(I.d_matrix, c[1], TMP.d_matrix, n);          // I * c[1] -> TMP
                                    cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);          // T + TMP -> T
                                    cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix,
T.d_matrix, U.d_matrix, n);          // A * T -> U
                                    // This is the equivalent of ..
                                    // V = p[6] * (c[12] * p[6] + c[10] * p[4] + c[8] *
p[2]) + c[6] * p[6] + c[4] * p[4] + c[2] * p[2] + c[0] * I;          RUN
IN STREAM 2
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[6]->d_matrix, c[12], T.d_matrix, n);          // p[6] * c[12] -> T
                                    cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[4]->d_matrix, c[10], TMP.d_matrix, n);          // p[4] * c[10] -> TMP
                                    cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);          // T + TMP -> T

```

```

        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[2]->d_matrix, c[8], TMP.d_matrix, n);          // p[2] * c[8]  -> TMP
        cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);                      // T + TMP      -> T
        cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[6]->d_matrix, T.d_matrix, T.d_matrix, n);      // p[6]
T
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[6]->d_matrix, c[6], TMP.d_matrix, n);          // p[6] * c[6]  -> TMP
        cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);                      // T + TMP      -> T
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[4]->d_matrix, c[4], TMP.d_matrix, n);          // p[4] * c[4]  -> TMP
        cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);                      // T + TMP      -> T
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(pow[2]->d_matrix, c[2], TMP.d_matrix, n);          // p[2] * c[2]  -> TMP
        cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);                      // T + TMP      -> T
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb)
(I.d_matrix, c[0], TMP.d_matrix, n);                // I * c[0]    -> TMP
        cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, V.d_matrix, n);                      // T + TMP      -> V
    }
    // This is the equivalent of ..
    //  $R = (V - U) / (2 * U) + I$ ;  ||??  $R = (-U + V) / (U + V)$ ;
    cudaSub KERNEL_ARGS2(cp.bpg, cp.tpb) (V.d_matrix, U.d_matrix,
T.d_matrix, n);
        cudaMulScalar KERNEL_ARGS2(cp.bpg, cp.tpb) (U.d_matrix, 2,
TMP.d_matrix, n);
        //cudaInv KERNEL_ARGS2(cp.bpg, cp.tpb) (TMP.d_matrix,
TMP.d_matrix, n); // TEMP CODE BELOW
        T.syncHost();
        CUDAMatrix::inv(T, T);
        T.syncDevice();
        //
        cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix,
TMP.d_matrix, T.d_matrix, n);
        cudaAdd KERNEL_ARGS2(cp.bpg, cp.tpb) (T.d_matrix, I.d_matrix,
R.d_matrix, n);
        // Squaring
        for (int k = 0; k < s; k++) {
            cudaMul KERNEL_ARGS2(cp.bpg, cp.tpb) (R.d_matrix,
R.d_matrix, R.d_matrix, n);
        }
        t.stop();
        R.syncHost();
    }
    return t;
} else {
    throw std::runtime_error("Matrix sizes do not match");
}
} else {

```

```

        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

CUDATimer CUDAMatrix::abs(CUDAMatrix& A, CUDAMatrix& R) {
    if (A.isInitialised() && R.isInitialised()) {
        int ar = A.getNumRows();
        int ac = A.getNumCols();
        int rr = R.getNumRows();
        int rc = R.getNumCols();
        if (ar == ac && ac == rr && rr == rc) {
            A.syncDevice();

            cudaParams cp = getCUDAParams(ar, ac);
            CUDATimer t;

            t.start();
            cudaAbs KERNEL_ARGS2(cp.bpg, cp.tpb) (A.d_matrix, R.d_matrix, ar);
            t.stop();

            R.syncHost();
            return t;
        } else {
            throw std::runtime_error("Matrix sizes do not match");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

```

// BOOLEANS

```

bool CUDAMatrix::isInitialised() {
    return initialised;
}

bool CUDAMatrix::isSquare() {
    if (initialised) {
        if (numCols == numRows) {
            return true;
        } else {
            return false;
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

bool CUDAMatrix::isDiagonal() {
    if (initialised) {
        if (!isSquare()) {

```

```

        return false;
    }
    for (int c1 = 0; c1 < numRows; c1++) {
        for (int c2 = 0; c2 < numCols; c2++) {
            if (c1 != c2 && getCell(c1, c2) != 0.0) {
                return false;
            }
        }
    }
    return true;
} else {
    throw std::runtime_error("Cannot perform matrix operations before
initialisation");
}
}

bool CUDAMatrix::isIdentity() {
    if (initialised) {
        for (int c1 = 0; c1 < numRows; c1++) {
            for (int c2 = 0; c2 < numCols; c2++) {
                if ((c1 != c2 && getCell(c1, c2) != 0.0) || (c1 == c2 &&
getCell(c1, c2) != 1.0)) {
                    return false;
                }
            }
        }
        return true;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

bool CUDAMatrix::isZero() {
    if (initialised) {
        for (int c1 = 0; c1 < numRows; c1++) {
            for (int c2 = 0; c2 < numCols; c2++) {
                if (getCell(c1, c2) != 0.0) {
                    return false;
                }
            }
        }
        return true;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

bool CUDAMatrix::isSmall() {
    return utils::max(numRows, numCols) < 150;
}

bool CUDAMatrix::isComplex() {

```

```

        std::complex<double> cell;
        for (int c1 = 0; c1 < numEls; c1++) {
            cell = getCell(c1);
            if (cell.imag() != 0.0) {
                return true;
            }
        }
        return false;
    }

// SETTERS

void CUDAMatrix::setCell(int row, int col, std::complex<double> val) {
    if (isInitialised()) {
        h_matrix[numCols * row + col] = val;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::setCell(int i, std::complex<double> val) {
    if (isInitialised()) {
        h_matrix[i] = val;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::setMatrix(std::complex<double> val) {
    if (isInitialised()) {
        for (int c1 = 0; c1 < getNumEls(); c1++) {
            h_matrix[c1] = val;
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::setMatrix(std::complex<double>* inMatrix) {
    if (isInitialised()) {
        for (int c1 = 0; c1 < numEls; c1++) {
            h_matrix[c1] = inMatrix[c1];
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::setMatrix(std::initializer_list<std::complex<double>> inMatrix) {
    if (isInitialised()) {

```

```

        if (inMatrix.size() == getNumEls()) {
            std::copy(inMatrix.begin(), inMatrix.end(), h_matrix);
        } else {
            throw std::runtime_error("Initialiser-list size does not match matrix
size");
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::setIdentity() {
    if (isInitialised()) {
        int row, col;
        for (int c1 = 0; c1 < getNumEls(); c1++) {
            row = getCurRow(c1);
            col = getCurCol(c1);
            if (row == col) {
                h_matrix[c1] = 1;
            } else {
                h_matrix[c1] = 0;
            }
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::setRandomDouble(double min, double max) {
    if (isInitialised()) {
        double r;
        std::default_random_engine rng((unsigned int) (time(0)));
        std::uniform_real_distribution<double> gen(min, max);
        for (int c1 = 0; c1 < numEls; c1++) {
            r = gen(rng);
            setCell(c1, r);
        }
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

void CUDAMatrix::setRandomInt(int min, int max) {
    if (isInitialised()) {
        int r;
        std::default_random_engine rng((unsigned int) (time(0)));
        std::uniform_int_distribution<int> gen(min, max);
        for (int c1 = 0; c1 < numEls; c1++) {
            r = gen(rng);
            setCell(c1, r);
        }
    }
}

```

```

        } else {
            throw std::runtime_error("Cannot perform matrix operations before
initialisation");
        }
    }

// GETTERS

double CUDAMatrix::getNorm(int n) {
    int c1, c2;
    double sum, max = 0;
    if (n == 1) {
        // 1 Norm
        for (c1 = 0; c1 < numCols; c1++) {
            sum = 0;
            for (c2 = 0; c2 < numRows; c2++) {
                sum += std::abs(getCell(c2, c1));
            }
            if (std::norm(sum) > std::norm(max)) {
                max = sum;
            }
        }
        return max;
    } else if (n == INFINITY) {
        // Inf Norm
        for (c1 = 0; c1 < numRows; c1++) {
            sum = 0;
            for (c2 = 0; c2 < numCols; c2++) {
                sum += std::abs(getCell(c2, c1));
            }
            if (std::norm(sum) > std::norm(max)) {
                max = sum;
            }
        }
        return max;
    } else {
        // Euclidian
        // Not called from anywhere. Requires SVD implementation to work.
        return -1;
    }
}

int CUDAMatrix::getCurRow(int i) {
    if (isInitialised()) {
        return (int) (floor(i / numCols));
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

int CUDAMatrix::getCurCol(int i) {
    if (isInitialised()) {
        return (int) (i - (numCols*getCurRow(i)));
    }
}

```

```

        } else {
            throw std::runtime_error("Cannot perform matrix operations before
initialisation");
        }
    }

std::complex<double> CUDAMatrix::getCell(int row, int col) {
    if (isInitialised()) {
        return h_matrix[row*numCols + col];
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

std::complex<double> CUDAMatrix::getCell(int i) {
    if (isInitialised()) {
        return h_matrix[i];
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

std::complex<double>* CUDAMatrix::getMatrix() {
    if (isInitialised()) {
        return h_matrix;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

int CUDAMatrix::getNumRows() {
    if (isInitialised()) {
        return numRows;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

int CUDAMatrix::getNumCols() {
    if (isInitialised()) {
        return numCols;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

int CUDAMatrix::getNumEls() {
    if (isInitialised()) {
        return numEls;
    }
}

```



```

        } else {
            throw std::runtime_error("Cannot perform matrix operations before
initialisation");
        }
    }

size_t CUDAMatrix::getSize() {
    if (isInitialised()) {
        return size;
    } else {
        throw std::runtime_error("Cannot perform matrix operations before
initialisation");
    }
}

// UTILS

int utils::getNumDigits(double x) {
    if (x > 1.0 || x < -1.0) {
        return (int) (floor(log10(abs(x))) + 1);
    }
    return 1;
}

int utils::max(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}

double utils::max(double x, double y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}

int utils::min(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return y;
    }
}

double utils::min(double x, double y) {
    if (x < y) {
        return x;
    } else {
        return y;
    }
}

```

```
}
```

```
// OPERATOR OVERRIDES
```

```
std::ostream& operator<<(std::ostream& oStream, CUDAMatrix& A) {
    if (A.isInitialised()) {
        // Init
        std::complex<double> cell;
        bool isComplex = A.isComplex();
        bool scientific = false;
        int c1, c2, r, i;
        int reallength = 0, imagLength = 0, exp = 0;
        double divider;
        int precision = 0;
        int maxFixedDigits = 4;
        // Get info
        for (c1 = 0; c1 < A.getNumEls(); c1++) {
            cell = A.getCell(c1);
            // Check if it's decimal
            if ((cell.real() - (int) (cell.real())) != 0.0 ||
                (cell.imag() - (int) (cell.imag())) != 0.0) {
                precision = 4;
            }
            // Get maximum exponent
            r = utils::getNumDigits(cell.real());
            i = utils::getNumDigits(cell.imag());
            if (r - 1 > exp) {
                exp = r - 1;
                reallength = r;
            }
            if (i - 1 > exp) {
                exp = i - 1;
                imagLength = i;
                if (abs(cell.imag()) == 1.0) {
                    imagLength++;
                }
            }
        }
        // Check if the output should be in fixed or scientific form
        if (exp >= maxFixedDigits) {
            scientific = true;
        }
        // Get divider for scientific form
        divider = std::pow(10, exp);

        // Output name and multiplier
        oStream << " = ";
        if (scientific) {
            oStream << "(10 ^ " << exp << ") *";
        }
        // Output cell
        oStream << std::endl << std::setprecision(precision) << std::fixed;
        for (c1 = 0; c1 < A.getNumEls(); c1++) {
            cell = A.getCell(c1);
```

```

oStream << "| ";
// Spacing and formatting for scientific/fixed
if (scientific) {
    cell /= divider;
} else {
    r = utils::getNumDigits(cell.real());
    for (c2 = 0; c2 < (reallength - r); c2++) {
        oStream << " ";
    }
}
// Output real
oStream << cell.real() << " ";
// Output complex
if (isComplex) {
    if (cell.imag() != 0.0) {
        if (cell.imag() > 0.0) {
            oStream << "+ ";
        } else {
            oStream << "- ";
        }
        if (abs(cell.imag()) != 1.0) {
            oStream << std::abs(cell.imag());
        } else {
            oStream << " ";
        }
        oStream << "i ";
    } else {
        i = utils::getNumDigits(cell.imag());
        for (c2 = 0; c2 < imagLength + 3; c2++) {
            oStream << " ";
        }
    }
}
// Output new line if row end reached
if (A.getCurRow(c1 + 1) > A.getCurRow(c1)) {
    oStream << "|";
    if (A.getCurRow(c1 + 1) < A.getNumRows()) {
        oStream << std::endl;
    }
}
}
oStream << std::endl;
return oStream;
} else {
    Throw std::runtime_error("Cannot perform matrix operations before
initialisation");
}
}

```

8.2.2 - CUDAMatrix.cuh

```
//
// Cardiff University | Computer Science
// Module:      CM3203 One Semester Project (40 Credits)
// Title:       Parallelisation of Matrix Exponentials in C++/CUDA for Quantum Control
// Date:       2016
//
// Author:      Peter Davison
// Supervisor:  Dr. Frank C Langbein
// Moderator:   Dr. Irena Spasic
//

// Precompiler include check
#ifndef cudamatrix_h
#define cudamatrix_h
// Include C/C++ stuff
#include <vector>
#include <iostream>
#include <math.h>
#include <iomanip>
#include <random>
#include <complex>
// Include CUDA stuff
#include "cuda.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <thrust/complex.h>
#include "cuda_intellisense.h"
#include "CUDATimer.cuh"

#include "cuda_profiler_api.h"

// KERNELS
__global__ void cudaAdd(thrust::complex<double>* A, thrust::complex<double>* B,
thrust::complex<double>* R, int n);
__global__ void cudaAddScalar(thrust::complex<double>* A, thrust::complex<double> scalar,
thrust::complex<double>* R, int n);
__global__ void cudaSub(thrust::complex<double>* A, thrust::complex<double>* B,
thrust::complex<double>* R, int n);
__global__ void cudaSubScalar(thrust::complex<double>* A, thrust::complex<double> scalar,
thrust::complex<double>* R, int n);
__global__ void cudaMul(thrust::complex<double>* A, thrust::complex<double>* B,
thrust::complex<double>* R, int n);
__global__ void cudaMulScalar(thrust::complex<double>* A, thrust::complex<double> scalar,
thrust::complex<double>* R, int n);
__global__ void cudaAbs(thrust::complex<double>* A, thrust::complex<double>* R, int n);

class CUDAMatrix {
private:
    // STRUCTURES
    struct cudaParams {
        dim3 tpb; // Threads per block
        dim3 bpg; // Blocks per grid
    };
};
```

```

};
struct padeParams {
    int scale;
    int mVal;
    std::vector<CUComplex*> pow;
};
// VARIABLES
std::complex<double>* h_matrix;
thrust::complex<double>* d_matrix;
int numRows, numCols, numEls;
size_t size;
bool initialised;
// MEMORY HANDLERS
void alloc();
void dealloc();
// CUDA STUFF
void syncHost();
void syncDevice();
static cudaParams getCUDAParams(int rows, int cols);
// INTERNAL PADE APPROXIMATION CODE
static padeParams getPadeParams(CUComplex* A);
static int ell(CUComplex* A, double coef, int m);
static std::vector<double> getPadeCoefficients(int m);
public:
    // CONSTRUCTORS & DESTRUCTOR
    CUComplex();
    CUComplex(int inNumRowsCols);
    CUComplex(int inNumRows, int inNumCols);
    CUComplex(int inNumRowsCols, std::initializer_list<std::complex<double>>
inMatrix);
    CUComplex(int inNumRows, int inNumCols,
std::initializer_list<std::complex<double>> inMatrix);
    CUComplex(const CUComplex &obj);
    void init(int inNumRows, int inNumCols);
    ~CUComplex();
    // MATRIX OPERATIONS
    static CUDATimer add(CUComplex* A, CUComplex* B, CUComplex* R);
    static CUDATimer add(CUComplex* A, std::complex<double> scalar, CUComplex* R);
    static CUDATimer sub(CUComplex* A, CUComplex* B, CUComplex* R);
    static CUDATimer sub(CUComplex* A, std::complex<double> scalar, CUComplex* R);
    static CUDATimer mul(CUComplex* A, CUComplex* B, CUComplex* R);
    static CUDATimer mul(CUComplex* A, std::complex<double> scalar, CUComplex* R);
    static CUDATimer pow(CUComplex* A, int pow, CUComplex* R);
    static CUDATimer tra(CUComplex* A, CUComplex* R);
    static CUDATimer inv(CUComplex* A, CUComplex* R);
    static CUDATimer exp(CUComplex* A, CUComplex* R);
    static CUDATimer abs(CUComplex* A, CUComplex* R);
    // BOOLEANS
    bool isInitialised();
    bool isSquare();
    bool isDiagonal();
    bool isIdentity();
    bool isZero();
    bool isSmall();

```

```

bool isComplex();
// SETTERS
void setCell(int row, int col, std::complex<double> val);
void setCell(int i, std::complex<double> val);
void setMatrix(std::complex<double> val);
void setMatrix(std::complex<double>* inMatrix);
void setMatrix(std::initializer_list<std::complex<double>> inMatrix);
void setIdentity();
void setRandomDouble(double min = 0, double max = 1);
void setRandomInt(int min = 0, int max = 1);
// GETTERS
double getNorm(int n);
//double getNormAm(int n);
// WRITE (Maybe)
int getCurRow(int i);
int getCurCol(int i);
std::complex<double> getCell(int row, int col);
std::complex<double> getCell(int i);
std::complex<double>* getMatrix();
int getNumRows();
int getNumCols();
int getNumEls();
size_t getSize();
};

// OPERATOR OVERRIDES
std::ostream& operator<<(std::ostream& oStream, CUDAMatrix& A);

// UTILS
namespace utils {
    int getNumDigits(double x);
    int max(int x, int y);
    double max(double x, double y);
    int min(int x, int y);
    double min(double x, double y);
}

#endif

```

8.2.3 - CUDATimer.cu

```
//
// Cardiff University | Computer Science
// Module:      CM3203 One Semester Project (40 Credits)
// Title:       Parallelisation of Matrix Exponentials in C++/CUDA for Quantum Control
// Date:        2016
//
// Author:      Peter Davison
// Supervisor:  Dr. Frank C Langbein
// Moderator:   Dr. Irena Spasic
//

// Include header file
#include "CUDATimer.cuh"

void CUDATimer::start() {
    clear();
    cudaEventCreate(&t1);
    cudaEventCreate(&t2);
    cudaEventRecord(t1, 0);
}

void CUDATimer::stop() {
    cudaEventRecord(t2, 0);
    cudaEventSynchronize(t2);
    cudaEventElapsedTime(&time, t1, t2);
    cudaEventDestroy(t1);
    cudaEventDestroy(t2);
}

void CUDATimer::clear() {
    time = 0;
}

float CUDATimer::getTime() {
    return time;
}

std::ostream& operator<<(std::ostream& oStream, CUDATimer& t) {
    oStream << std::setprecision(10) << std::fixed << t.getTime()/1000 << "s" <<
    std::endl;
    return oStream;
}
```

8.2.4 - CUDATimer.cuh

```
//
// Cardiff University | Computer Science
// Module:      CM3203 One Semester Project (40 Credits)
// Title:       Parallelisation of Matrix Exponentials in C++/CUDA for Quantum Control
// Date:        2016
//
// Author:      Peter Davison
// Supervisor:  Dr. Frank C Langbein
// Moderator:   Dr. Irena Spasic
//

// Precompiler include check
#ifndef timer_h
#define timer_h
// Include C/C++ stuff
#include <iostream>
#include <iomanip>
// Include CUDA stuff
#include "cuda.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "cuda_intellisense.h"

class CUDATimer {
private:
    float time;
    cudaEvent_t t1, t2;
public:
    void start();
    void stop();
    void clear();
    float getTime();
};

// OPERATOR OVERRIDES
std::ostream& operator<<(std::ostream& oStream, CUDATimer& A);

#endif
```


8.2.5 - main.cpp

```
//
// Cardiff University | Computer Science
// Module:      CM3203 One Semester Project (40 Credits)
// Title:       Parallelisation of Matrix Exponentials in C++/CUDA for Quantum Control
// Date:       2016
//
// Author:      Peter Davison
// Supervisor: Dr. Frank C Langbein
// Moderator:   Dr. Irena Spasic
//
#include "Main.h"
#include "cuda_profiler_api.h"

int main(int argc, char **argv) {

    try {

        cudaProfilerStart();

        int size = 5;

        // Create variables
        std::complex<double> i = std::complex<double>(0, 1);
        CUDAMatrix A(size, {
            10, 2, 23, 13, 2,
            3, 5, 41, 18, 3,
            35, 12, 3, 13, 14,
            7, 22, 26, 2, 35,
            24, 31, 3, 66, 18
        });
        CUDAMatrix eA(size);
        CUDAMatrix eAi = CUDAMatrix(size);

        // Calculations
        CUDATimer t1 = CUDAMatrix::exp(A, eA);
        CUDATimer t2 = CUDAMatrix::mul(eA, i, eAi);

        // Output
        std::cout << "A" << A << std::endl;
        std::cout << "e^A" << eA << t1 << std::endl;
        std::cout << "e^A * i" << eAi << t2 << std::endl;

        cudaProfilerStop();

    } catch (std::exception e) {
        std::cout << std::endl << e.what() << std::endl;
    }

    return 0;
}
```

8.3 - Table of Abbreviations

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
I/O	Input/Output
IDE	Integrated Development Environment
LU	Lower Upper (Referring to LU Decomposition)

8.4 - Glossary

1D	One dimensional
2D	Three dimensional
Device	Refers to a GPU
Hamiltonian	The current energy state of a particle (Stored as a matrix)
Host	Refers to the host computer and CPU
Matrix	A series of numbers stored in a 2D array