Alternative streaming methods for data parallel algorithms using Cuda

Rashid Hafez

November 22, 2018

Declaration

I, Rashid Hafez confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Rashid Hafez

Date: November 22, 2018

Abstract

This paper looks at different approaches of data partitioning large datasets which exceed the maximum resources available on the GPU, and alternative methods of streaming these datasets to the GPU to execute basic matrix operations on the datasets in parallel. The intention of this paper is to compare the findings of implementing different techniques of transferring data to the GPU; and actually implementing different methods to record the most efficient one. The research conducted should provide insight to implementing an optimal solution for automated kernel rewrites to support datasets which are larger than that of the memory available.

Contents

1 Introduction

Recently, many programs and developers attempt to exploit the parallel processing speed of the GPU for general purpose computing, but when these programs contain datasets which exceed the resources available on the GPU, one would have to manually rewrite their kernels and program to support operations on larger arrays. An example of one such application which is limited by the memory available on the GPU can be seen in the computer graphics imagery sector, Blender (an open source 3D animation and design program) can render a scene on the GPU, however, when the scene exceeds the memory capacity of the GPU, the program can not function (Blender (2017)). In their source code they use Cuda to do computations of arrays on the GPU. However, their program doesn't use streams nor do they implement techniques to partition the data for this specific function (Blender (2018)).

The fundamental issue remains that one could rewrite their code to support computations of larger size to be streamed to the GPU, and it may be efficient on their current underlying GPU architecture. However if this program were to be run on a different architecture it may falter or be inefficient to the point of redundancy. This dissertation aims to provide an efficient out of core computation model to tackle issues such as these. I will be using NVIDIA's CUDA API language, to develop and test matrix computations on large data sets. I will use this model to test matrix computations such as stenciling, matrix multiplication and incrementation, this will be useful for programs, like Blender which use the GPU for matrix computations. Several other applications in the industry attempt to make use of the GPU for general purpose computing, I believe this research can be beneficial to applications such as video encoding, data mining, machine learning and big data analysis.

One boundary which may be difficult to overcome is keeping the speed and efficiency of processing the data relative to the problem set. In essence, as the data set increases more transfers must take place between the CPU and the GPU, this costs time as the PCI-E bus has a restriction on how much data can be transferred at a time, thus the program's main limitation is being memory bound. In addition to this, the approach must take into account the overhead of creating streams and partitioning kernels, and data transfers between them. To tackle memory bound limitations, this project also aims to look into implementing kernel fusion for overlapping kernels, if they are present. Exploiting locality and shared memory within the

GPU can also be looked at to improve memory bound limitations.

1.1 Aims

The aim of this paper is to offer an approach towards automating kernel rewrites and streaming pipelines for out of core computations on the GPU, without crippling the efficiency and speed of the software. The method developed will be designed to specifically use CUDA. I also aim to provide results which should be taken into consideration for creating a program which can automate kernel partitioning and streaming which would be scaleable for an arbitrary amount of datasets and operations. Ideally this would be best implemented as a function where it can automatically split your data and stream it to the GPU instead of the user having to split all their data and stream it manually.

1.2 Objectives

1. Preliminary Objectives

    1.1. Become fluent with the CUDA API, this can be demonstrated by writing and developing basic kernel functions for stenciling, matrix multiplication and incrementation.

    1.2. Research, compare and contrast different methods to compute very large datasets on the GPU where it does not fit.

2. Main Objectives

    2.1. Develop a 'streamless' method to constantly transfer data larger than the GPU to the GPU for matrix multiplication and stenciling, determine the performance on these operations.

    2.2. Develop a streaming method for executing asynchronous calls to the GPU for matrix multiplication and stenciling.

    2.3. Experiment with different data partitions to determine the most efficient partitioning technique, compare and evaluate the results of different partition methods in combination with streaming and using multiple forms of memory storage on the device for different situations, and provide a detailed analysis on the efficiency of the code.

    2.4. Present evaluation of findings from data gathered by tests in graphical and tabular format.

2.5. Propose the most efficient thread-block combinations and partition method(s) to dynamically stream data to the GPU for a variety of operations such as multiplication and stenciling.

2.6. Determine whether this process can be automated for different architectures and programs.

3. Advanced/Optional Objectives

3.1. The results obtained can then be implemented in an extension or library for automating kernel rewrites for operations such as multiplication and stenciling.

3.2. Develop library for automated dynamic partitioning and rewrites of multiplication and stencil kernels.

2 Literature Review and Background

This chapter will relay foundational information on hardware architecture of modern day computers and the CUDA API, firstly explaining from an abstract view how memory is used and accessed in the CPU (host) and the GPU (device). Along with this will be a section on previous studies conducted in this field.

2.1 Hardware Architecture and CUDA Analysis

This subsection covers an overview of the common modern architecture of computers, and how my work relates to it. Including information on how the CUDA API works along with an in-depth analysis of memory management in CUDA and CUDA streams.

2.1.1 Hardware

Traditionally the CPU contains a three level cache system to manage and transfer data to the CPU cores, the memory is managed where the most frequent and highest priority data is stored in the lower level caches. The lower level the cache is, the closer it is physically to the CPU core, meaning its faster to access, but the less memory space it has. When these caches are full the data is stored in physical memory (RAM). When a core needs to access this data stored in RAM, the operating system must search for it in the RAM and the data must physically travel from the RAM to the Northbridge and through the front side bus (FSB) to the CPU. RAM is used to store temporary data and it is cleaned out and reset whenever the computer is turned off. RAM is also limited in memory capacity, therefore when it is full the operating system virtualises memory on the disk to store and fetch data. However, fetching data from disk is much slower as the disk is physically further away from the CPU and the operating system has to search through the disk for the data needed. This is a long and cumbersome process which can delay the speed of the computer. Lets say we have a single core CPU and we needed to process some operation on two very large arrays. All of their contents wouldn't fit into the cache, so we store it on disk. To process this, the CPU needs to fetch addresses to operate on our arrays, it goes through the long process of searching RAM and then having to search the disk and then transfer all the way back through the Southbridge and Northbridge to the CPU core. A CPU core can generally execute 4 32 bit instructions per clock cycle, we could process the computation needed quite fast, but the CPU is also in charge of managing many other things, such as I/O, other programs and the operating system. It would be useful if we could outsource our array computations to an external device, the GPU.

A study from Harvard (Lu et al. (2010)) compared the speed performance between the GPU and CPU an an image processing program, they showed up to thirty times increase in performance on the GPU. The graphics processing unit is traditionally known to mainly handle and process video and display tasks to the screen, nowadays they're being used for general purpose computing. A graphics card is fundamentally created by multiple streaming multiprocessors (SMs) which contain multiple CUDA cores. Common architectures contain up to 32 units per SM. Each SM can execute one instruction, all cores process one thread in parallel, this is a lot faster than the CPU, in addition to this, all the 32 execution units inside the SM have shared memory with each other.

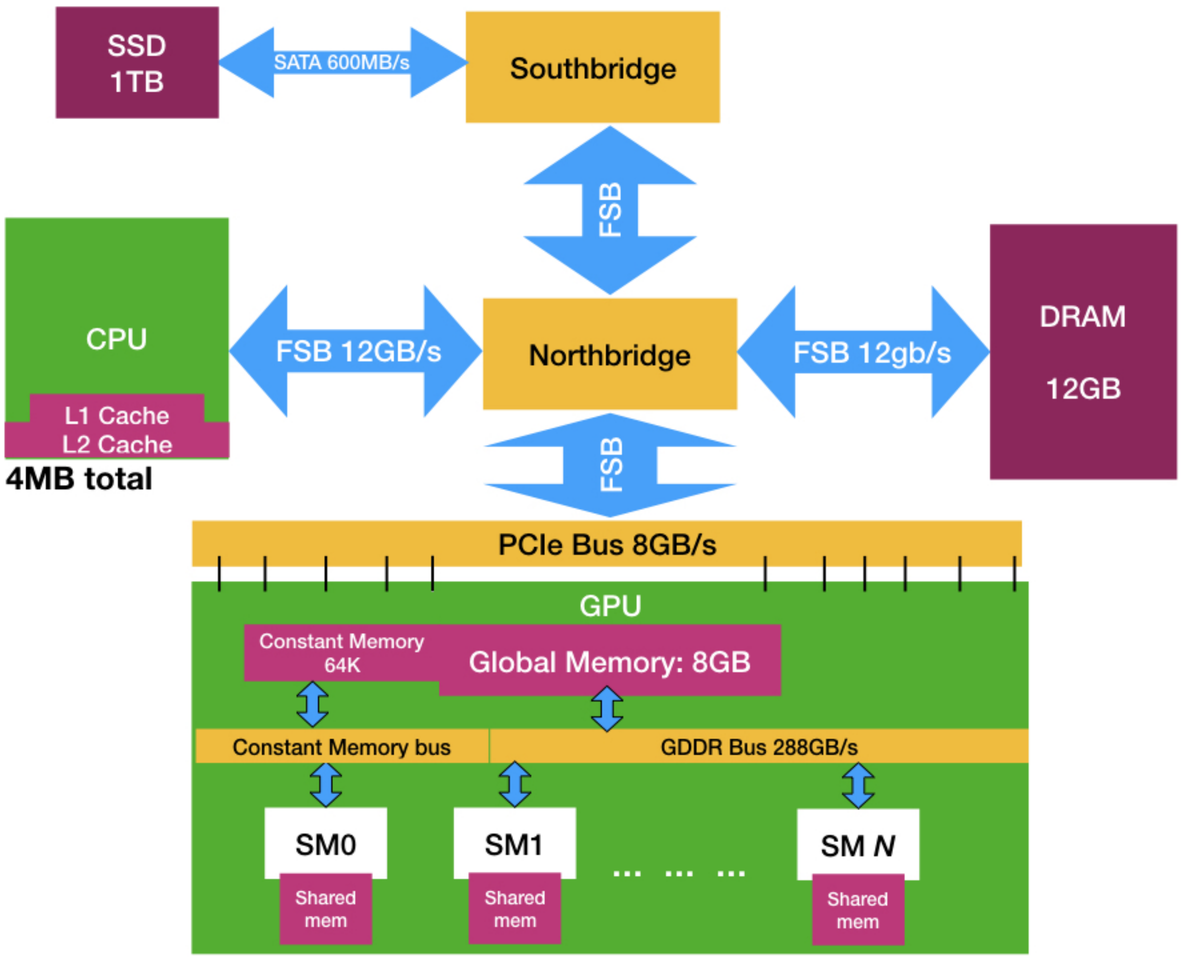


Figure 1: Interpretation of Architecture

Inside the GPU it contains memory space for texture memory, constant memory and shared memory. (Cook (2013)). Global memory is supplied via GDDR (Graphic Double Data Rate) on the graphics card. This is a high-performance version of DDR (Double Data Rate) memory. The memory bus on the device can usually be around 512 bits wide, potentially five times more bandwidth than the host. This allows for a higher rate of transfer of data between SMs, which is more beneficial than using the CPU.

The GPU is connected to the motherboard via the PCIe port, as you can see from 1 the GPU only needs to transfer data through the PCIe to the CPU, whereas if we were using virtual memory inside the SSD it would take a lot longer to transfer, this is one of the reasons why outsourcing our array computation to the GPU is a lot more efficient than storing it on the disk. On common modern architectures the PCI-E bus is limited to data transfers of up to around 4GB/s possibly more, whereas the bandwidth for the global memory bus for devices such as Tesla K40 have bandwidths of up to 288GB/s. The bottleneck of the system is the PCIe bus between the CPU and the graphics card, as the bandwidth is capped around 4GB/s but the GPU could actually handle more than that.

In summary the CPU is best suited for complex but small computations, whereas the GPU's potential can be seen when working with very large but simple computations. The benefits can be seen when there is a large amount of data which needs processing, it can be transferred via the PCIe to the GPU and stored in the GPU memory. The device global memory can be accessed by each SM much quicker than the CPU would be able to access the onboard RAM, as the GDDR is integrated inside the device with a much wider bus. The host has a larger memory capacity in total, whereas the device contains a finite amount of memory but is much faster at processing computations on large data. Some limitations can be seen in the architecture, the PCIe has a limit on how much data can be transferred at a time, and the global memory on the device is two to three orders of magnitude smaller than the capacity of memory available on the host.

2.1.2 CUDA

"CUDA is a parallel computing platform and programming model developed by NVIDIA for general purpose computing on graphical processing units (GPUs)" (Nvidia (2017)).

The language presents itself as a simple way to access the GPU and utilize it's parallel processing power. NVIDIA's architecture uses a variant of SIMD, called Single Program Multiple

Data (SPMD) (see Cook, 2013, chap5). Their architecture runs on hardware that at the core is SIMD. CUDA splits problems into grids of blocks, each containing multiple threads. A grid is essentially an entire array associated with a single kernel launch. Each Kernel is executed on only one device, but multiple kernels can be ran concurrently on the GPU. A kernel is essentially a function which can be executed on the GPU in parallel. The blocks contain groups of threads, blocks may run in any order. Only a subset of the blocks will ever execute at any one point in time. A block must execute from start to completion and may be ran on any one of N SMs (symmetrical/streaming multiprocessors). Each block contains multiple warps, a single warp is a group of 32 threads. In most architectures a SM is made up of 32 Cuda cores (see figure 2), although the name is similar to CPU core it doesn't exactly recreate the same functionality of a core. Instead, each CUDA core executes one thread at a time and all the CUDA cores can be executed at the same time (in parallel) (Messmer (2013)). Blocks are allocated, from the grid of blocks, to any SM that has free slots. Initially this is done on a round-robin basis so each SM gets an equal distribution of blocks. For most kernels, the number of blocks needs to be in the order of eight or more times the number of physical SMs on the GPU (see Cook, 2013, chap2).

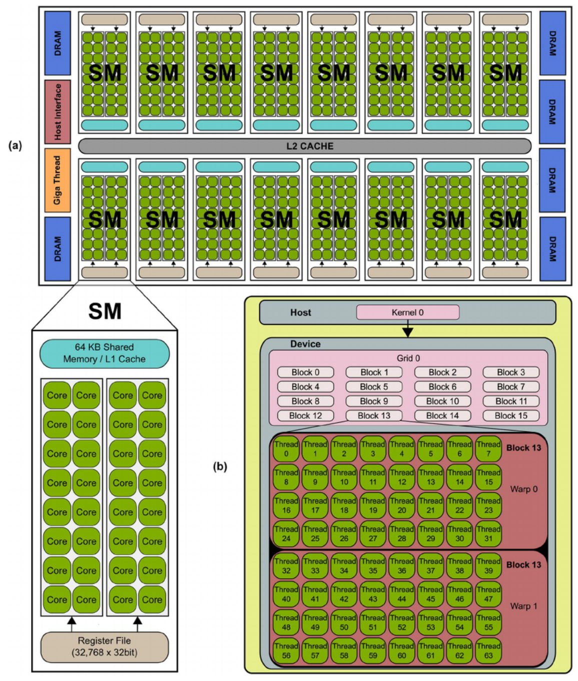


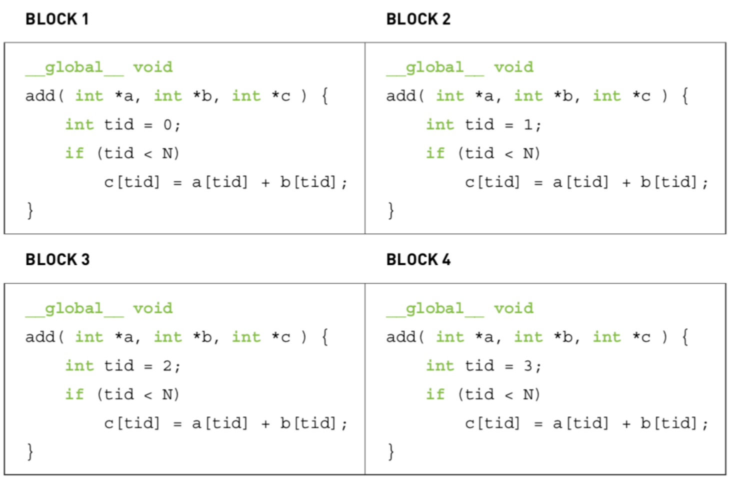
Figure 2: Hernandez Fernandez et al. (2013)Diagram of GPU Architecture

Using the GPU for general purpose must be done in a specific way, the host must first allocate the capacity it needs for the data and then copy it to the GPU. Pseudo code is provided below to briefly explain how invocations of GPU functions are used, the code is to simply add two arrays together, A and B, on the device, store them in C and return the result to the host. There are more complex and efficient procedures to do this, the example shown is simplistic:

```
int N = 100;
//host arrays
int * hostA;
int * hostB;
int * hostC;

malloc(&hostA, N* sizeof(int)); //allocate the space for the array on the host (do
    this for all host arrays)
FillArrays(hostA,hostB); //add data to A and B

//device arrays
int * devA;
int* devB;
int* devC;

cudaMalloc(&devA, N*sizeof(int)); //Do this for all device arrays... What if this
    fails?

cudaMemcpy(devA,hostA,N*sizeof(int),HostToDevice); //Send data in host array to
    the device

add<<<N, 1>>>(devA,devB,devC); // N blocks, 1 thread
```

First we allocate space on the host for the arrays, fill the arrays on the host, the n allocate a section on the GPU to store the data using $cudaMalloc$. Then we transfer these arrays to that space on the device using $cudaMemcpy$.

The invocation of the add function on the GPU is specified by the <<<>>> brackets. Inside these angled brackets we specify the initialisation of blocks and threads. When this function is called the kernel is assigned to a grid containing the amount of blocks we specify (in this example $blocks = N$). Each block contains 1 thread, and will execute the addition in parallel (as shown in figure3). Each thread will be executed in parallel on one of the 32 execution units contained in the SM.



BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Figure 3: (Sanders and Kandrot (2010)) Parallel Execution of Addition on the GPU

Different thread-block execution configurations can result in different speeds of outputs depending on what implementation the configuration is being applied to. This project will explore different thread block combinations on different implementations on the device, such as stenciling and matrix multiplication, assuming that the same thread block configuration would produce different results for each implementation.

One of the limitations of the architecture is that the hardware limits the amount of blocks which can be executed at launch and similarly with threads, they are limited to 1024 threads per block, assuming the device is of compute type 2.x or more. Also the global and shared memory is much more limited in the device than that of the hard disk (see Sanders and Kandrot, 2010, chap 5.2.1).

Commented next to the cudaMalloc function“what if this fails?”. If our dataset is too large to allocate on the GPU $cudaMalloc$ will return an $OutOfMemory$ error, we need another way around this. To efficiently automate a method around allocating large memory to the GPU it is essential to understand types of memory allocation and how memory is managed on the host/device.

2.1.3 Memory Management

Physically the GPU is incapacitated from directly accessing memory stored on the host, when we call malloc on the host, physical memory is used, or virtual memory from disk, the GPU can’t access this memory, therefore, the data we want to calculate must be transferred from the host to device. The CUDA API allows such transfer of data through specific functions such as *cudaMemcpy*, doing this will then store the data from the CPU to the device on global memory. There are different abstractions of memory on the device and locations of where they're held are different to one another. The main forms of memory on the device are, texture, constant and global, these are all intercommunicated between each SM within the device. In addition to this each SM has its own integrated storage of shared memory which is only shared between processors within the SM, this is much more limited in memory compared to the other shared forms.

There are different ways to transfer data to the GPU. Traditionally one would use *malloc*() to allocate data on the CPU, then use *cudaMemcpy* to transfer it to the GPU. However, there are other ways to allocate memory, such as, pinned memory and unified memory. Pinned memory is a traditional method used to prevent the data we need on the physical memory being swapped out to the disk, this can improve the speed of the program as there would be less page misses in the memory as it is constantly in physical memory. It's generally used for when data is needed to be transferred to the device and its needed to be done concurrently. More recently developed is unified memory, as previously stated, the GPU can not access memory partitions on the host, well this function is very useful as it allows the host and device to both access memory stored in the same location.

The CUDA architecture consists of different implementations of shared memory on the device, each with different speeds and size, generally the speed is relative to the size of the memory space. Global memory in the device is typically the slowest form of memory storage and access, but has the most space in contrast to shared memory on the registers, where they have the fastest access internally but typically can only contain up to 64K of data per SM. Speedup of using registers to read and write to variables compared to global memory on the device have been seen to increase by up to 9 times faster (see Cook, 2013, chap6). Essentially shared memory would be used where, inside a block containing a group of threads, certain data can be reused inside that SM, instead of repeatedly pulling from the global memory. Constant memory is seen to be significantly faster than global memory, however this memory capacity is also more limited than global. Studies such as (Ren Wu (2009)) show up to about 2.5x improvement on performance with constant memory usage.

In order to efficiently develop an automated kernel rewrite, it is essential to understand and fully utilize different forms of memory on the device to obtain as much efficiency as possible from the device without ever over using limited forms of memory.

2.1.4 Streams and Partitions

CUDA streams are used to integrate task parallelism into the application, a stream represents a queue of GPU operations that get executed in a specific order (Sanders and Kandrot (2010)). In addition to this, CUDA streams provide other features, significantly it allows the kernel to execute partitions of data from an array at a time and transferring a portion of the data at a time through the PCIe. Therefore streams can be applied to situations where the data is larger than the memory available on the GPU.

To explain how streams work in detail, I will provide a simple pseudo code snippet. For this example lets say our GPU is extraordinarily small and holds only 40 elements at a time, and our full data size is 100 units, this is for simplicity.

```
DATASIZE = 100; //full datasize
N = 20; // The "chunk" of data to process
cudaStream stream; // the stream we will add our instructions to

HostMalloc(host_a, sizeof(int*DATASIZE)); //allocate pinned memory on host
HostMalloc(host_b, sizeof(int*DATASIZE));

fillArrays(host_a,host_b);

cudaMalloc(dev_a, sizeof(int * N )); //Allocate space for 20 elements in
    the device
cudaMalloc(dev_b, sizeof(int * N )); //Allocate space for another 20
    elements in the GPU for our second array

for (int i=0; i<DATASIZE; i+= N) {
cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int), HostToDevice, stream )
    ); //adding instruction to stream
cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int), HostToDevice, stream )
    ); //adding instruction to stream
```

```
cudaKernel<<<block, thread, stream>>>(dev_a, dev_b) //specify which stream
    to queue the kernel launch in

cudaMemcpyAsync( host_c+i, dev_c, N * sizeof(int), DeviceToHost, stream )
    ); //send the contents back to the CPU
}
```

Firstly pinned memory must be specified on the host using HostMalloc, this can be done on the full data set (for now), this prevents the pages of data which need to be transferred to the device from being swapped out of memory onto the disk. In order to copy the contents needed to the stream asynchronously, the contents are required to be allocated on pinned memory, the benefit of this is to allow other tasks to be executed concurrently as the copy takes place. However pinned memory also has its drawbacks, as physical memory (RAM) on the host is also limited.

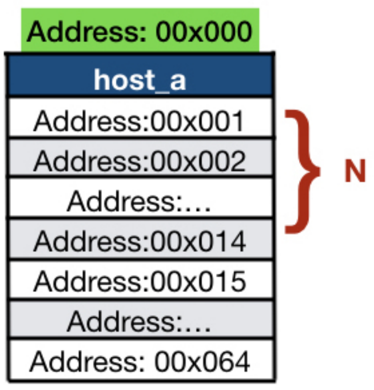


Figure 4: The array of data in memory with corresponding addresses

The next step is to specify an allocated slot in the device. Since the device has limited memory its possible to allocate a fixed amount of memory on the device using cudaMalloc on only a part of the data size (for instance: N), what proceeds this is interesting. As mentioned previously, data must be physically transferred to the GPU from the host, this is done using $cudaMemcpy$. To execute it in an asynchronous manner $cudaMemcpyAsync$ is used, this will be elaborated in more detail shortly. This function takes a couple parameters: the array we want to fill up, the pointer to the array that contains the data we want to transfer, the size of

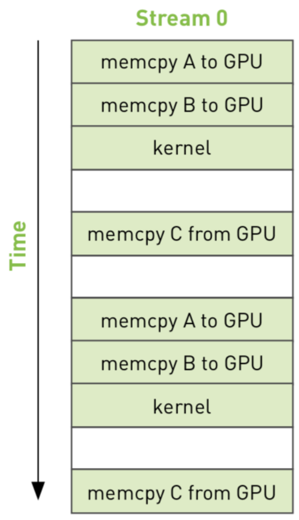


Figure 5: (Sanders and Kandrot (2010)) A typical stream containing a queue of operations

the chunk we want to take, what type of transfer (i.e. host to device or device to host), and finally the stream we want to add the instruction to. In the code, the obscurity arises at the memcopy function call inside the for loop, as the second parameter shows $host_a + i$. Well, on the first iteration, i = 0, what does this mean. Well i acts an an offset to the header of the address of the storage. Looking at figure 4 if $i = 0$ we are telling the program that from this point we want to transfer the next twenty elements to the GPU, this is done using the second parameter of the memcpy function. To specify that we want to transfer a size of 20, N is of 20 in the third parameter, and this will be the chunk transferred to the GPU. Then the next iteration the offset i increments to the 20th element address, thus making the pointer start at address 00x014.

Figure 5 shows the queue of instructions that will be executed in order. These are all the instructions from the code added to the queue of the stream, since $cudaMemcpyAsync$ was used, this allows an extra level of parallelism into the program, task parallelism. The main benefit of this is that while the data is transferring to the GPU the program can concurrently execute other instructions. This is more of a latency hiding technique (concurrency) than actual parallelism, which is still beneficial to the speed of the program. For example, while the stream is enqueued for transferring data to the GPU, the CPU can execute separate code without being blocked by the memory copy call.

For my implementation I must consider using multiple streams and asynchronous execution to allow further concurrency in the program. Pinned memory is also a limiting factor to take into account, possibly allocating pinned memory on portions of the data at a time could be considered.

2.2 Critical Analysis of Related Work

A couple have intentionally attempted to tackle this issue of out of core computations for very large datasets directly. One such study (Ren Wu (2009)) looked into streaming the K-means algorithm to the GPU, where the cluster was larger than the memory in the GPU. They partitioned the cluster into large blocks and streamed them from the host to the GPU for execution. They used asynchronous transfers to hide latencies, thus being able to use the CPU to concurrently execute partitions of their K-means algorithm while transferring their partitions to the GPU. Ren Wu also suggested taking maximum advantage of texture and constant memory residing in the device to store data which remains constant during a single kernel invocation. Due to the fact that the GPU is optimized for multiple threads working together on continuous memory addresses, instead of having each thread work on its own they suggest that,

"It is important to design GPU data structure to be column-based instead of row-based as commonly found in algorithms designed for CPU" Their findings suggest that there is an unnecessary need for more than two streams, the results show that two streams with overlapping functions are faster than one stream or more than two. This research proves useful as to types of memory management on the GPU, they claim that constant memory used was substantially faster (2.2x) than texture memory, and even global memory proved faster than texture memory, however these findings were situational to the fact where the data is to large to fit into texture memory and there is a cache miss then the fetch time increases substantially. Therefore it's advised to use texture memory very carefully.

Their study proved an increase in speed of up to 80x to process their K-means algorithm on the GPU compared to the CPU. However if the dataset does fit into the GPU its still typically around 2x faster than if it didn't.

A study conducted by (Wu et al. (2012)) implemented an automated engine to pick kernels and fuse them together to improve locality and reduce redundancy of memory and code, they implemented for data-warehousing functions such as SELECT and JOIN operations on database tables. Their study aims to tackle a bottleneck in the system, mainly the fact that the PCIe bus is fairly limited in bandwidth, therefore this study exploits locality and asynchronous execution. They mentioned briefly that their implementation can be useful for situations where table entries in the database wouldn't fit on the GPU. But this study actually focuses more on fusing kernels, and splitting kernels into segments and asynchronously computing segments in the GPU whilst transferring some of the data from the GPU back to the host. This is known as kernel fission. Fusion is the method of conjuring kernel codes together to "Fuse the code bodies of two GPU kernels to i) eliminate redundant operations across dependent kernels, ii) reduce data movement between GPU registers and GPU memory, iii) reduce data movement between GPU memory and CPU memory, and iv) improve spatial and temporal locality of memory references". Memory access patterns were monitored and their specific model attempted to reduce banking conflicts between threads.

Their automated system consisted of a directed graph to store similar kernels together and then their program could iterate through this graph locating similar kernels for potential fuses. Essentially this paper concludes that fusing kernels together can be important when their is multiple fetches from global memory for the same variable but in different kernels. Their fuse provided a speedup of about 30% and improved throughput by about 2x, as less data needed transferring through the PCIe.

Tabik et al. (2015) produced a paper on improving performance of stencils on the GPU. They adapted Gaussian code for stenciling on the first four stages. They then proceeded to derive the most optimal thread block configurations for their stencil program. Their model essentially should automate the thread block configuration according to their benchmarks. For example it appears that using the maximum thread limit on both block dimensions for their architecture actually reduces the optimal time to process the stencil for stage 1 stencil pipeline. (I.E."The total number of threads per thread-block must be smaller than or equal to a given limit $Threads_Max$, $Blk.x * Blk.y < Threads_Max$"). Another benchmark suggested is that all SMs must be proactive at the time of execution, meaning all SMs must be allocated to at least one block.

They also claim "Reducing global memory transactions in stencils is possible by increasing data reuse in the on-chip shared memory and registers". Their model also proposes a solution to reduce memory bank conflicts with internal threads when shared memory is used.

Unfortunately, this model seems to be best suited for when the dataset fits nicely inside the GPU, in other words, it is not suited for out of core computations, however many positive factors can be derived from this study. However, A couple years later (Tarek S. Abdelrahman (2018)) produced a paper on automation of rewrites for stencil computations in the GPU, this study will also be taken into account.

All these findings suggest that extensive use of task parallelism and shared memory is critical in developing an effective automation of kernel rewrites and dynamic partitions. Ren Wu's (2009) study also provided some insight to the benefits of constant and texture memory, however showing that texture memory can be delicate. It is also important to realise, studies such as (Tabik et al. (2015)) attempt to optimise thread block configurations by attempting to map all possible blocks to be executed on the hardware. In other words making sure every SM has a job to do in parallel means there is a certain amount of blocks which must be present upon execution. The only downfall to this paper is their model works best on situations where the data fits into the GPU. This paper also realises extensive research into kernel fusion (such as Wang et al. (2010) and Wu et al. (2012)), but only briefly touched upon this, as it is not the main goal of this paper. However these studies show it's possible to eliminate latencies in the program, by using minimum kernels as possible and fully utilising locality and shared memory. A common interest to take into account is monitoring memory access patterns to ensure there are as few bank conflicts as possible, or potentially none. A best suited algorithm would be a derivative of the previous findings, an algorithm which can process some of the matrix computations on the CPU as well as the GPU, to reduce overall workload and which exploits locality and shared memory.

3 Requirements Analysis

This paper is intended for research purposes, my aim is to create an automated dynamic data partitioning system which can partition and stream data to the GPU in partitions for very simple matrix operations. As there is much research going into this field to accomplish my goal I must rigorously test different techniques of memory storage, transfers, streams and locality on the system performance and evaluate these results and provide my findings to establish efficient techniques which can be applied to future research. Therefore a traditional requirements analysis such as MOSCOW shall not be provided. I have provided a critical analysis of qualitative data collected from previous research studies conducted in this field.

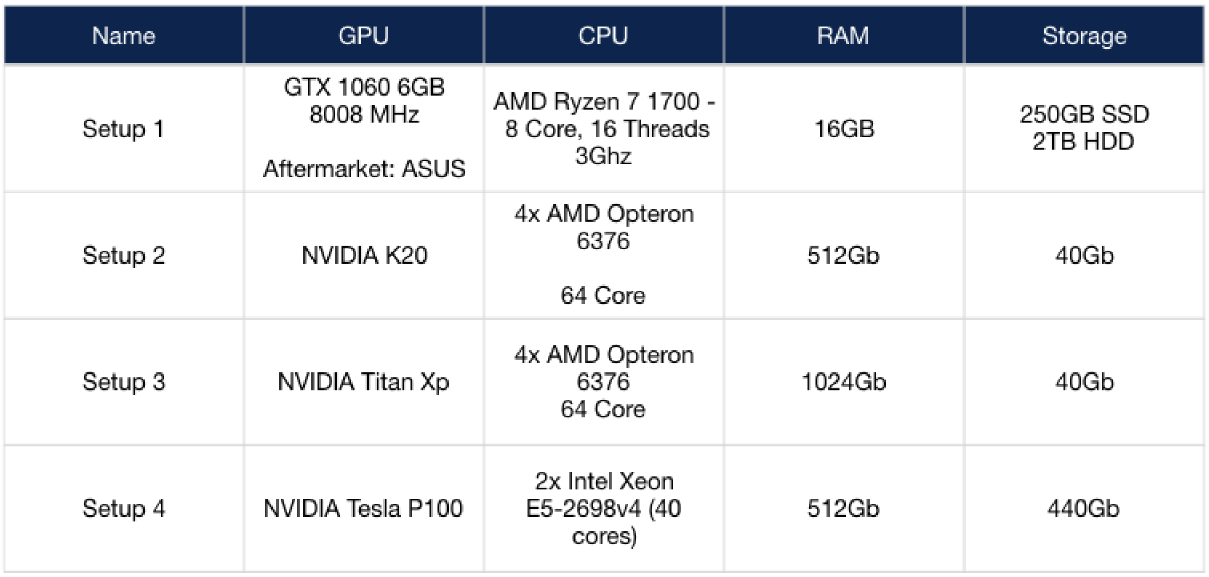
4 Research Methodology

This chapter will describe the experimental design of the project, discussing first the main goal, the benchmarks and environment which it will be conducted in, the stages on how I will implement the project in relation to the objectives the paper sets out to achieve and the evaluation strategy in place.

This project conducts exploratory research in the field of alternative dynamic parallel methods for calculations on the GPU. This project offers the most efficient technique for partitioning data to fit into the GPU for matrix computations without sacrificing efficiency. This will be done by collecting quantitative data on rigorous tests of the implementation I will develop, these results will be represented in graphical and tabular format in order to compare and evaluate the data.

4.1 Testing Environment

Thanks to Heriot Watt University and Edinburgh Centre for Robotics (2014), I am able to run tests of a significant level on multiple architectures and devices on their cluster.



Name	GPU	CPU	RAM	Storage
Setup 1	GTX 1060 6GB 8008 MHz Aftermarket: ASUS	AMD Ryzen 7 1700 - 8 Core, 16 Threads 3Ghz	16GB	250GB SSD 2TB HDD
Setup 2	NVIDIA K20	4x AMD Opteron 6376 64 Core	512Gb	40Gb
Setup 3	NVIDIA Titan Xp	4x AMD Opteron 6376 64 Core	1024Gb	40Gb
Setup 4	NVIDIA Tesla P100	2x Intel Xeon E5-2698v4 (40 cores)	512Gb	440Gb

Figure 6: Table 1: List of workable environments

4.2 Benchmarks

I will develop two to three benchmark applications to test this system. These application will consist of matrix computations such as matrix multiplications and stencils. These benchmarks have been chosen as they are generic staple computations in this industry and are simple to implement, yet sturdy enough to thoroughly test the program. Previous studies mentioned in section 2.2 have implemented similar applications therefore these benchmarks seem most appropriate.

A preliminary experiment was carried out of vector addition of multiple sizes, see setup 1 in table 1. The GPU was unable to allocate data once the total element size in both arrays reached over $8 * 10^8$. Therefore a suitable testing scheme would be to work from there and have a benchmark of about 3 or more orders of magnitude larger.

Thread block configurations will be determined throughout development of the program. Ideally it would make sense to stress test the system at first starting with the maximum configuration of thread blocks and working around this to discover which other combinations would be optimal. Generally the limit is 1024 threads per block

From research collected it can be derived that the best suited algorithms consist of utilising the CPU as well as the GPU asynchronously.

In the literature background we discussed code which can partition data and send it for computation on the GPU. This paper wants to determine whether or not there is different chunk combinations better suited for kernel computations. Essentially determining what is the most efficient partitioning technique to stream data to the GPU, as per objective 2.3.

To satisfy objective 2.6 the program can be run on any one of the multiple setups mentioned in table 1.

4.3 Stages of Implementation

To determine if automation is possible, I must first set out to discover whether or not if arbitrary sizes of matrices can actually be processed and streamed to the GPU. Then the project must define the most efficient method(s). To accomplish all of this, I must structure my objectives and implementation in an orderly fashion. Logically each stage should be implemented in order as per objectives, and tasks can be attempted to be tackled in parallel where possible. Objectives 1.1 and 1.2 are already partially complete.

- Stage 1: Developing kernels for industry staple matrix computations (stencils and multiplication). Preferably I will be concurrently researching deeper into streams and out of core computations on the GPU. As per preliminary objectives 1.1 and 1.2.
- Stage 2: To determine whether its possible to compute data larger than the memory available on the GPU. Develop a 'streamless' out of core model to compute matrices on the device. And develop a method which uses streams.
- Stage 3: Develop a generic CPU exclusive program of the matrix operations, with mostly optimised code which could include latency hiding and parallelism.
- Stage 4: Extensively test the system, I will test the efficiency of the CPU only method against the streaming and non streaming method. This will include rewriting multiple versions of the streaming program with reworked code for various partition and stream combinations and various dataset sizes. In addition to this I will also have to experiment with shared memory and different forms of integrated memory storages, such as global memory and texture memory. Ultimately this stage requires the most time and preliminary research to have been conducted. This will satisfy objectives 2.3 - 2.5.
- Stage 5: Run the different GPU programs on various architectures provided by Edinburgh Centre for Robotics (2014).

4.4 Evaluation Strategy

Rigorous and thorough tests based on benchmarks of the program will be conducted. To record these results I will use inbuilt functions such as the system clock supported in linux, and NVIDIAs profiler (aka: nvprof) which is beneficial to monitoring streams and memory access patterns. Quantitative findings of the timings will be recorded from when the kernel for matrix operations are executed, and will be timed from start to finish, i.e. when all the data is finished being computed on the kernel. The evaluations will take place in order every time the thread-block and partitions are edited, meaning every time I change the methods I will compare the findings with one another.

1. Time execution from start to finish for CPU exclusive program
2. Time execution from start to finish for GPU stream program

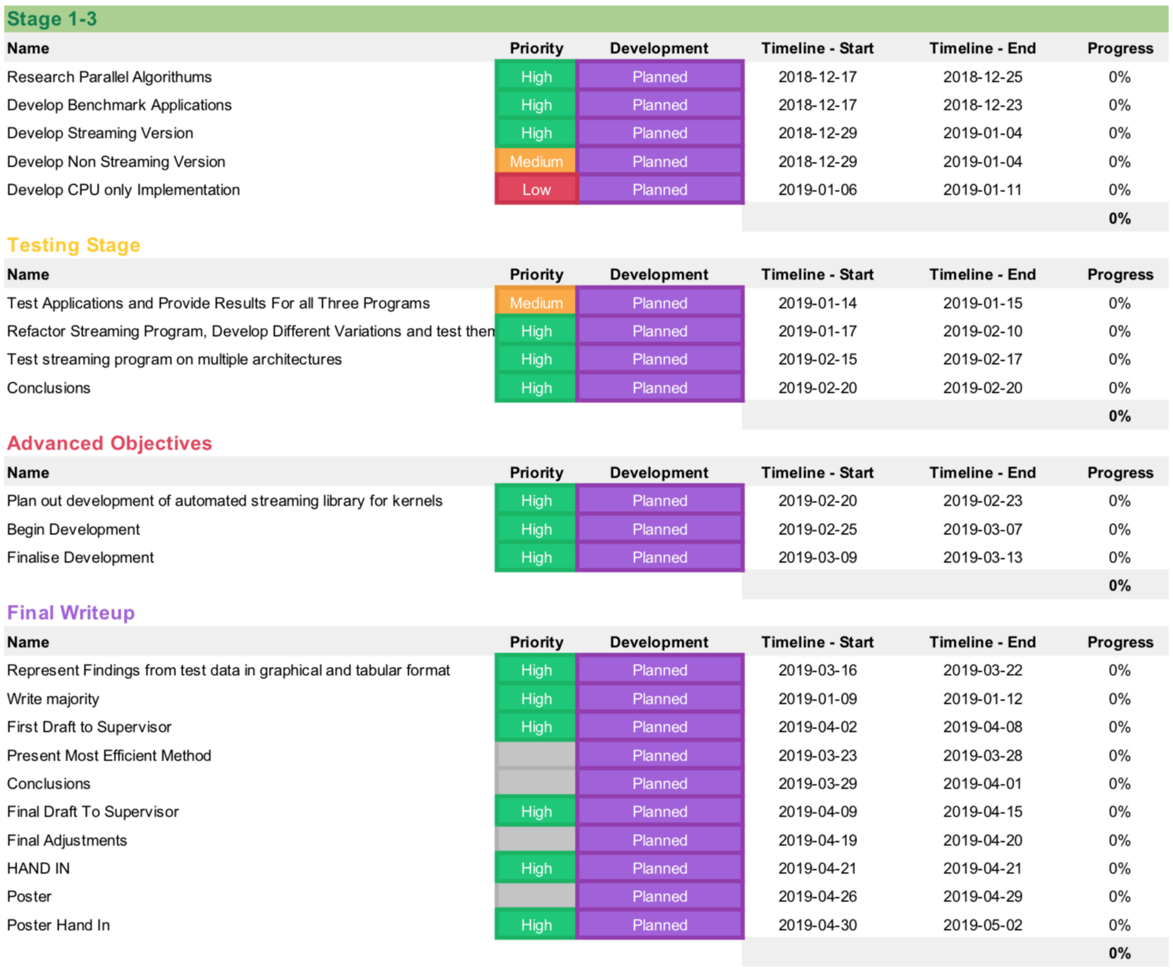
3. Time execution from start to finish for GPU streamless program
4. compare findings.
5. Make memory and thread optimisations on streaming program (could be memory management or partition data change.)
6. Time execution on new streaming program
7. Compare the new changes on streaming program with the original
8. Repeat steps 5-7 a number of times until I am satisfied with results and can coherently provide an efficient method of memory management and partition techniques.
9. Time execution of preferred method on multiple architectures
10. Time execution of alternative methods on same architectures
11. Compare methods to see which method is the most reliable and efficient for all architectures.

Once the results are obtained it shall be clear which method is suitable to implement into a library for automation of matrix computations.

5 Project Management

Section to describe project management and to provide project plan in a table and gantt chart.

None of the core concepts of project management for software engineering suit this dissertation, or will benefit me in any way. Since I still need to effectively manage my dissertation project, I will solely adopt an obscured variation of extreme programming, derivative of the agile development model, to incrementally implement the GPU streaming and non streaming programs. I would like to effectively and timely produce these programs and test them in increments. Proceeding this will consist of numerous hours of research and trial and error for development of different techniques for streaming data.



Stage 1-3

Name	Priority	Development	Timeline - Start	Timeline - End	Progress
Research Parallel Algorithums	High	Planned	2018-12-17	2018-12-25	0%
Develop Benchmark Applications	High	Planned	2018-12-17	2018-12-23	0%
Develop Streaming Version	High	Planned	2018-12-29	2019-01-04	0%
Develop Non Streaming Version	Medium	Planned	2018-12-29	2019-01-04	0%
Develop CPU only Implementation	Low	Planned	2019-01-06	2019-01-11	0%
					0%

Testing Stage

Name	Priority	Development	Timeline - Start	Timeline - End	Progress
Test Applications and Provide Results For all Three Programs	Medium	Planned	2019-01-14	2019-01-15	0%
Refactor Streaming Program, Develop Different Variations and test then	High	Planned	2019-01-17	2019-02-10	0%
Test streaming program on multiple architectures	High	Planned	2019-02-15	2019-02-17	0%
Conclusions	High	Planned	2019-02-20	2019-02-20	0%
					0%

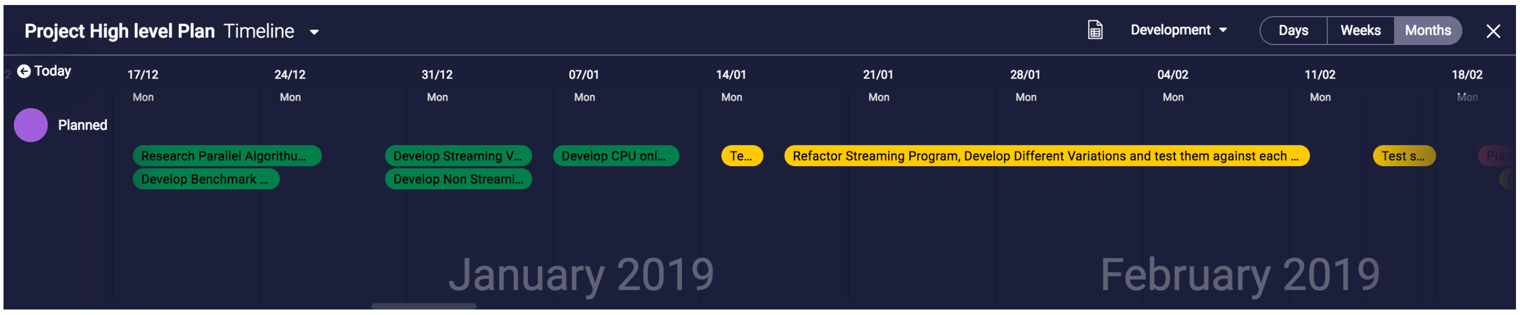
Advanced Objectives

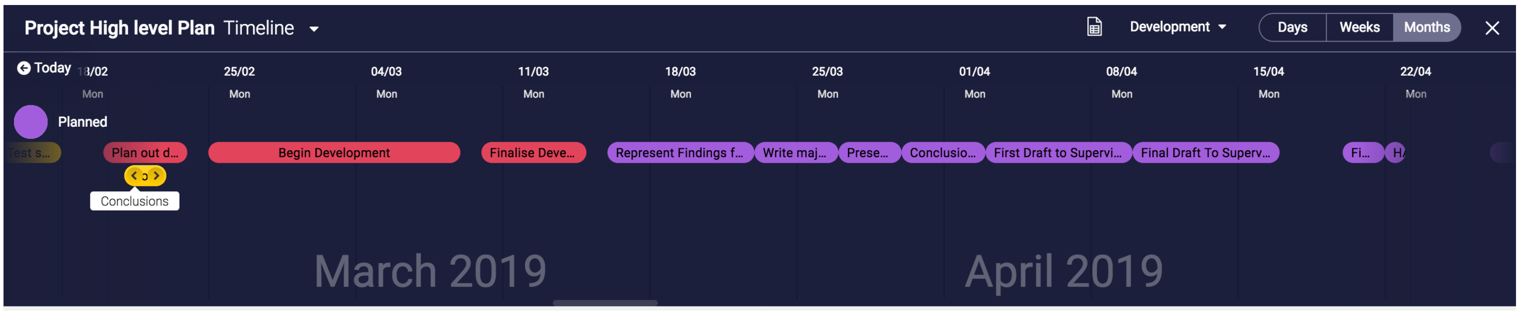
Name	Priority	Development	Timeline - Start	Timeline - End	Progress
Plan out development of automated streaming library for kernels	High	Planned	2019-02-20	2019-02-23	0%
Begin Development	High	Planned	2019-02-25	2019-03-07	0%
Finalise Development	High	Planned	2019-03-09	2019-03-13	0%
					0%

Final Writeup

Name	Priority	Development	Timeline - Start	Timeline - End	Progress
Represent Findings from test data in graphical and tabular format	High	Planned	2019-03-16	2019-03-22	0%
Write majority	High	Planned	2019-01-09	2019-01-12	0%
First Draft to Supervisor	High	Planned	2019-04-02	2019-04-08	0%
Present Most Efficient Method		Planned	2019-03-23	2019-03-28	0%
Conclusions		Planned	2019-03-29	2019-04-01	0%
Final Draft To Supervisor	High	Planned	2019-04-09	2019-04-15	0%
Final Adjustments		Planned	2019-04-19	2019-04-20	0%
HAND IN	High	Planned	2019-04-21	2019-04-21	0%
Poster		Planned	2019-04-26	2019-04-29	0%
Poster Hand In	High	Planned	2019-04-30	2019-05-02	0%
					0%

Figure 7: Plan





I have used a professional and easy to use and visualise online tool, called Monday (monday.com LTD (2018)), I can access and edit the project plan through the phone app and on their website. Gantt charts are outdated and in overall just frustrating to work with in slow programs like Excel, they may be useful for projects which are related to software development and such, however my situation is different.

The first three stages is indicated in green, the last two stages (testing) is in yellow, advanced objectives in red and the final writeup is purple. Each individual pulse (sprint) is either done concurrently with another or linearly but I have added in buffer periods of about one or two days between each individual sprint, incase one takes longer than expected.

6 Professional, Legal, Ethical and Social Issues

This chapter discusses legal and ethical issues related to my work.

6.1 Professional

As a computer scientist professional and student of Heriot Watt University, it is my responsibility to ensure that my dissertation is applied in a professional procedure. As the author I accept full responsibility for this project and the consequences resulting of. It is my job as a representative of Heriot Watt University, to make sure that no harm comes to its reputation as a result of unacceptable stands of professionalism. In regard to this, attention must be brought to the Heriot Watt University Code of Good Practice in Research, this is taken into full account for my research project. As the program (BSc Computer Systems) which this project is being done under is accredited by the British Computer Society (BCS), full acknowledgement will be taken towards their Code Of Good Practice.

6.2 Legal

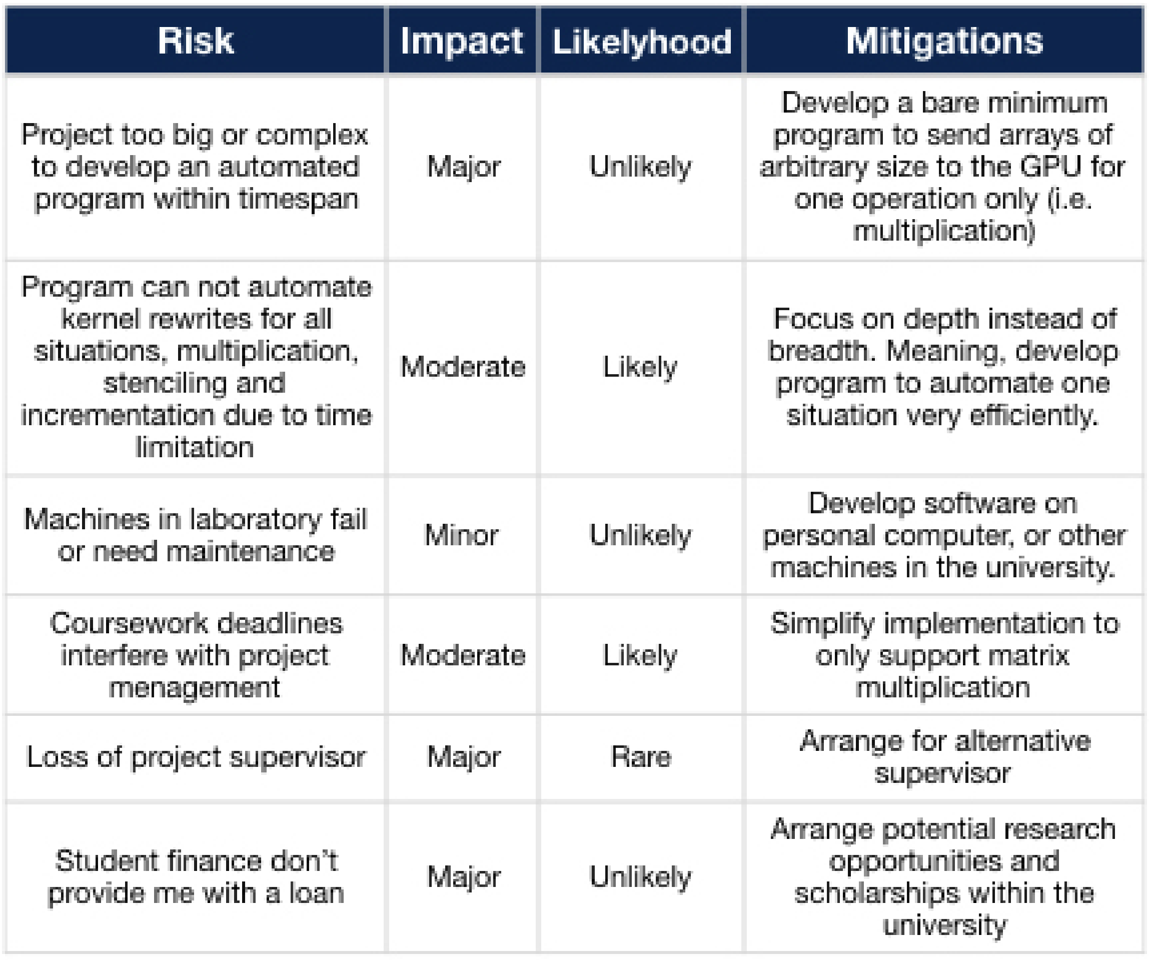
NVIDIA's CUDA API is under free to use licensing. I must abide by their End User License Agreement ([See]NVIDIA (2018)).

6.3 Ethical and Social Issues

There are no apparent human or social interactions which take place in this study, data or consent of humans are not collected, thus no ethical or social matters are present.

7 Risk Management

This section will cover potential risks, the possibility of occurrence and potential management of said risks in tabular format.



Risk	Impact	Likelyhood	Mitigations
Project too big or complex to develop an automated program within timespan	Major	Unlikely	Develop a bare minimum program to send arrays of arbitrary size to the GPU for one operation only (i.e. multiplication)
Program can not automate kernel rewrites for all situations, multiplication, stenciling and incrementation due to time limitation	Moderate	Likely	Focus on depth instead of breadth. Meaning, develop program to automate one situation very efficiently.
Machines in laboratory fail or need maintenance	Minor	Unlikely	Develop software on personal computer, or other machines in the university.
Coursework deadlines interfere with project menagement	Moderate	Likely	Simplify implementation to only support matrix multiplication
Loss of project supervisor	Major	Rare	Arrange for alternative supervisor
Student finance don't provide me with a loan	Major	Unlikely	Arrange potential research opportunities and scholarships within the university

Figure 8: Risk Analysis

References

Blender (2018), 'Blender source code', `https://developer.blender.org/diffusion/B/repository/master/`. Accessed: 22-10-2018.

Blender, F. (2017), 'Blender manual', `https://docs.blender.org/manual/en/dev/render/cycles/gpu_rendering.html#why-does-a-scene-that-renders-on-the-cpu-not-render-on-the-gpu`. Accessed: 22-10-2018.

Cook, S. (2013), *CUDA Programming, A Developer's Guide to Parallel Computing With GPUs*, Morgan Kaufmann Waltham, MA 02451, USA, ISBN: 978-0-12-415933-4.

Edinburgh Centre for Robotics (2014), 'Robotarium cluster'. The cluster is part of the EPSRC Centre for Doctoral Training in Robotics and Autonomous Systems (RAS) in Edinburgh grant (EP/L016834/1) funded by The Engineering and Physical Sciences Research Council (EPSRC) (UK).
URL: *https://doi.org/10.5281/zenodo.1455754*

Hernandez Fernandez, M., Guerrero, G., Cecilia, J., Garcia, J., Inuggi, A., Jbabdi, S., E J Behrens, T. and Sotiropoulos, S. (2013), 'Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus', *PloS one* **8**, 61892, DOI: 10.1371/journal.pone.0061892.

Lu, Peter J. Oki, H. F. C. A., Chamitoff, G. E., Chiao, L., Fincke, E. M., Foale, C. M., Magnus, S. H., McArthur, W. S., Tani, D. M. and Whitson, P. A. (2010), *Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station*, Vol. 5, ISSN:1861-8219, DOI: 10.1007/s11554-009-0133-1, Published: Harvard.

Messmer, P. (2013), 'Cuda part a: Gpu architecture overview and cuda basics; peter messmer (nvidia)', `https://www.youtube.com/watch?v=nRSxp5ZKwhQ`. Accessed: 22-10-2018.

monday.com LTD (2018), 'Monday', `https://monday.com`. Accessed: 10-11-2018.

Nvidia (2017), 'Nvidia developer', `https://developer.nvidia.com/cuda-zone`. Accessed: 22-10-2018.

NVIDIA (2018), 'End user license agreement', `https://docs.nvidia.com/cuda/eula/index.html`.

Ren Wu, Bin Zhang, M. H. (2009), *GPU Accelerated Large Scale Analytics*, HewlettPackard Development Company LP, HPL- 2009-38.

Sanders, J. and Kandrot, E. (2010), *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st edn, Addison-Wesley Professional, ISBN: 0131387685, 9780131387683.

Tabik, S., Peemen, M., Guil, N. and Corporaal, H. (2015), 'Demystifying the 16 x 16 thread-block for stencils on the gpu', *Concurr. Comput. : Pract. Exper.* **27**(18), 5557–5573, Published: John Wiley and Sons Ltd. Chichester, UK, ISSN:1532–0626.
URL: *http://dx.doi.org/10.1002/cpe.3591*

Tarek S. Abdelrahman, J. G. (2018), A strategy for automatic performance tuning of stencil computations on gpus, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada M5S 3G4, p. 24.

Wang, G., Lin, Y. and Yi, W. (2010), Kernel fusion: An effective method for better power efficiency on multithreaded gpu, *in* 'Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing', GREENCOM-CPSCOM '10, IEEE Computer Society, Washington, DC, USA, pp. 344–350.
URL: *http://dx.doi.org/10.1109/GreenCom-CPSCom.2010.102*

Wu, H., Diamos, G., Wang, J., Cadambi, S., Yalamanchili, S. and Chakradhar, S. (2012), Optimizing data warehousing applications for gpus using kernel fusion/fission, *in* '2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum', IEEE, ISBN: 978-1-4673-0974-5, DOI: 10.1109/IPDPSW.2012.300.