COMP390

2020/21

Accelerating Biological Sequence Alignment with GPU

Student Name:	Robert Szafarczyk
Student ID:	201307211
Primary Supervisor Name:	Dr. Thomas Carroll
Secondary Supervisor Name:	Professor Prudence Wong

Department of
Computer Science

University of Liverpool
Liverpool L69 3BX

Acknowledgements

I would like to thank my supervisors Dr. Thomas Carroll and Professor Prudence Wong for their support and constructive criticism during the project. Their comments have improved the quality of this work. Special thanks to Thomas for introducing me to the world of GPGPU programming and for running GPU tutorial sessions at the beginning of the academic year.

I would also like to thank all the other lecturers, teachers, and co-workers, past and present, from whom I was lucky enough to learn.

Finally, I thank my family, friends and partner for their vital support in my studies.

COMP390

2020/21

Accelerating Biological Sequence Alignment with GPU

Department of
Computer Science

University of Liverpool
Liverpool L69 3BX

Abstract

In bioinformatics, pairwise sequence alignment is the use of computational methods to find regions of similarity between two biological sequences. As sequencing technology improves and scientists are able to recover ever longer DNA or Protein sequences, the computational load of sequence alignment increased. This project uses the Graphics Processing Unit (GPU) to accelerate the Needleman-Wunsch (NW) and Smith-Waterman (SW) dynamic programming algorithms to solve global and local sequence alignment.

GPU memory sizes are an order of magnitude smaller than CPU memory, which poses a problem when aligning longer sequences on the GPU. A modular concurrent GPU kernel design is shown, which enables the alignment of very long sequences, limited only by the host memory size.

Inter-kernel synchronisation on GPUs can be expensive, which often prohibits employing all Streaming Multiprocessors (SMs) of a GPU to work on the same problem, especially if that problem requires data communication between parts of the algorithm. This project used a lightweight barrier in GPU global memory to synchronise concurrently executing kernels and enable data communication.

Throughput, latency and performance scalability results for the GPU implementation are presented and evaluated. The improvement of using more concurrently executing kernels is studied and discussed.

Contents

List of Figures

List of Code Listings

Chapter 1

Introduction

In this chapter, the problem of biological sequence alignment is introduced. The motivation behind using the GPU for this problem is explained, and some common applications are presented. The concept of General Purpose GPU programming is introduced and put into a historical context. Finally, the aims and objectives of this project are diffused into a list, and the ethical considerations of achieving them are mentioned.

1.1 Sequence alignment

Bioinformatics is an interdisciplinary field that aims to better understand the vast amounts of biological data through computational methods. As computers have become more powerful, bioinformatics tools have helped to bridge the gap between large, irregular biological data sets and human understanding. Especially in genetics, where Deoxyribonucleic acid (DNA) and Protein sequences of organisms are studied, the help of computers is invaluable - a virus is a relatively simple organism, but its genome can have up to 40 thousand base pairs; human genomes can have up to 3 billion base pairs. Sequence alignment aims to arrange sequences of DNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [12]. It is an essential part of the geneticist's toolbox. By assigning a similarity score to two sequences, researchers are better able to reason about evolution, determine ancestry and more effectively cure diseases. For example, using sequence alignment software, researchers have identified a gene believed to encode the unique ability to hop in animals [3]. Sequence alignment is a big part of "urgent computing", not least in the Covid-19 pandemic where through the use of sequence alignment software, researchers could better study the virus and quickly identify mutations [15].

As more and longer sequences are analysed, the computational complexity of alignment increases. Faster tools are desired. This project demonstrates that the Graphics Processing Unit (GPU) offers a way to accelerate sequence alignment through exploiting data parallelism ingrained in the problem.

$$\begin{array}{cccc} \mathrm{A} & \mathrm{C} & \mathrm{C} & \mathrm{T} \\ | & — & | & | \\ \mathrm{A} & \mathrm{T} & \mathrm{C} & \mathrm{T} \end{array}$$

Figure 1.1: An example alignment of *ACCT* with *ATCT*. *A's* are aligned, then a gap is inserted instead of aligning *C* with *T*, followed by aligning the *C's* and *T's*.

1.2 General Purpose GPU programming

In order to meet the computational requirements of advances in computer graphics in the 1980s and 1990s, chip manufactures started producing special-purpose processors. The ever-increasing screen resolutions meant that the CPU couldn't process all the pixels on time - it lacked parallelism needed to apply the same operation to first thousands and then millions of pixels at the same time. At the end of the millennia, the term "Graphics Processing Unit (GPU)" emerged to mean a special processor attached to the CPU, intended to accelerate computer graphics programs. A GPU trades expensive control structures (branch predictors, out-of-order execution, etc.) found on the CPU for many lightweight compute cores (figure 1.2).

In the early 2000s, the parallelism of the GPU started to be used for general purpose programming. Initially, computation was expressed through graphics APIs, like OpenGL and DirectX. However, this was cumbersome, and in 2006 Nvidia introduced the Compute Unified Device Architecture (CUDA) as a programming model for General Purpose GPU (GPGPU) Programming. In 2009, OpenCL was released as an open standard alternative to the proprietary CUDA framework.

Nvidia and AMD are currently two of the main GPU manufactures. Fundamentally, their architecture is similar, with the main differences lying in how concepts are named. This project used Nvidia GPUs with CUDA, and consequently follows terminology used by Nvidia.

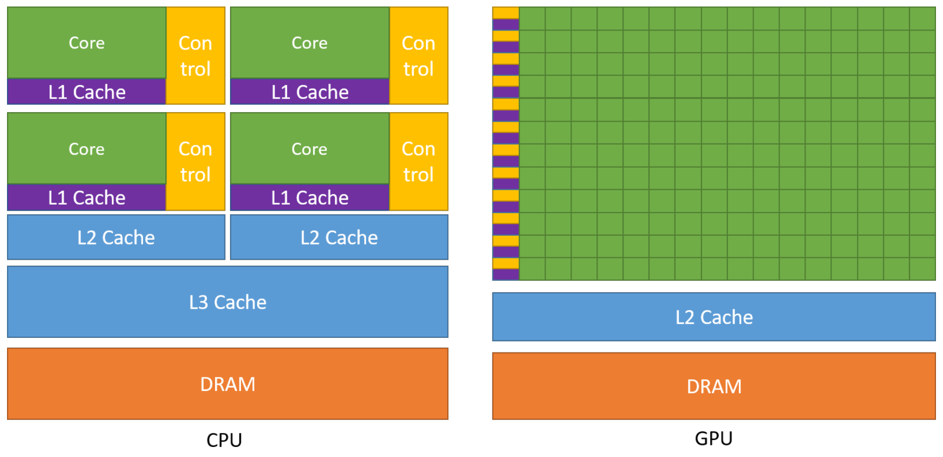


Figure 1.2: Control (yellow) transistor budget in the CPU is spent on compute units (green) in the GPU (CUDA Programming Guide [7]).

1.3 Aims & Objectives

The goal of this project was to accelerate the Needleman-Wunsch (NW) and Smith-Waterman (SW) sequence alignment algorithms on the GPU. During the project, the aim of enabling the alignment of very long sequences was added. This high level goal was decomposed into a list of aims and objectives in the Project Proposal document. The list is repeated here, with additional items added which emerged during the execution of the project.

Aims:

- Understanding the problem of sequence alignment in the context of bioinformatics.
- Understanding the GPU architecture, the underlying programming model and gaining practical experience in accelerating a workload using the GPU.
- Implementing the global sequence alignment algorithm on the GPU.
- Evaluation and understanding of the performance improvement of using a GPU.
- Implementing the local sequence alignment algorithm on the GPU *(optional)*.
- Allowing to align as long sequences as possible, i.e. the limit should be the host memory size, not the memory size of the GPU *(added)*.

Objectives:

- Implement a global sequence alignment algorithm on the CPU.
- Set up a test framework, including test data generation and performance measurement.
- Design a global sequence alignment algorithm implementation for the GPU, exploiting the unique features of the architecture.
- Implement the designed algorithm on the GPU.
- Repeat the above for local alignment *(optional)*.
- Run experiments comparing the GPU and CPU.
- Use concurrently executing kernels to employ all GPU resources in aligning longer sequences *(added)*.
- Evaluation of the performance improvement and scalability of using concurrent GPU kernels *(added)*.

1.4 Ethical considerations

The ethical conduct guidelines available on the COMP390 module page[1] and the University of Liverpool Academic Integrity Policy[2] were followed throughout the project. Some project specific considerations include:

- Test data (DNA or Protein sequences) were either produced randomly or obtained from a public domain [9] with the right permission. The data were only used in this project and were not shared further.
- Data generated in the functional and performance testing is reproducible, without any barriers or considerable effort needed.
- The software produced herein can be compiled and tested on any compatible machine, as well as on the University's Barkla compute cluster.
- The results described in this dissertation were generated through individual work. Where others' work or ideas have been used, citations were provided.
- The project was carried out in the United Kingdom and didn't involve any other participants.

There are several sources of protein and DNA sequence data available in the public domain. This dissertation made use of GenBank, which is a biological sequence database from the National Institutes of Health (NIH), a part of the U.S. Department of Health and Human Services [9]. The GenBank database is designed to provide and encourage access within the scientific community to the most up-to-date and comprehensive biological sequence information and places no restrictions on the use or distribution of the data.

[1]`https://student.csc.liv.ac.uk/internal/modules/comp390/2020-21/ethics.php`

[2]`https://www.liverpool.ac.uk/media/livacuk/tqsd/code-of-practice-on-assessment/appendix_L_cop_assess.pdf`

Chapter 2

Background

In this chapter, the problem of pairwise global and local sequence alignment is described in more detail. Next, an overview of the Nvidia GeForce GT 750M GPU is given as a representative example of the GPU architecture; together with the accompanying CUDA programming model. Finally, a non-comprehensive literature survey of related work is conducted.

2.1 Pairwise sequence alignment

First, some common terms used in the problem of sequence alignment are defined.

Alphabet:
A finite set of symbols, denoted by Σ. This project uses two alphabets:

$$\Sigma_{DNA} = \{A, C, G, T\}.$$
$$\Sigma_{Protein} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}.$$

Sequence:
A finite succession of symbols in Σ, denoted by s. Formally, $s \in \Sigma^*$.

Substring:
A substring of s is a consecutive succession of symbols in s. Given a sequence s and its substring s', such that $|s| = m$ and $|s'| = n$, it is said that s' starts at $s[i]$ and ends at $s[j]$, and both $1 \leq i \leq j \leq m$ and $j - i + 1 = n$ are satisfied. For example, `ACT` is a substring of `GACTG`, but `GTG` is not.

Subsequence:
A subsequence of s is a sequence obtained by removing zero or more symbols from s. For example, both `ACT` and `GTG` are subsequences of `GACTG`.

Alignment sequence:
A sequence p and r are pairwise alignments of the sequences s and t, if p is a subsequence of s, and if r is a subsequence of t. If $s, t \in \Sigma^*$ then $p, r \in (\Sigma \cup \{-\})^*$, i.e. the gap character is introduced to represent deletion of a symbol. Figure 1.1 shows an example of pairwise alignment.

Scoring matrix (function):
A function ρ assigning a score to aligning two characters of an alphabet, $\rho : \Sigma \times \Sigma \rightarrow \mathbb{Z}$.

Gap penalty:
A score $g \in \mathbb{Z}$ for aligning a character in Σ with "$-$" (character deletion).

There exist several types of sequence alignment techniques, each suited for a different task. This project implemented global and local alignment, which are described next.

2.1.1 Global pairwise sequence alignment

Global alignment aims to find the best alignment along the entire length of two sequences. The "best" means the one which obtains the highest score when the scoring function is applied to every character pair of the aligned sequences. The number of possible alignments is equal to 2^{n+m}; clearly, a brute-force approach quickly becomes intractable. In 1970 Needleman and Wunsch introduced the idea of using Dynamic Programming (DP) to solve the global sequence alignment problem [13]. Their algorithm has a running time of $O(nm)$ and requires $O(n\text{m})$ space. It has become the de facto standard for sequence alignment, with most subsequent algorithms being based on their work. The algorithm consists of two main parts - using DP to solve the recurrence relation, which gives the alignment score, and a traceback using information from the solved recurrence relation, which gives the alignment (see figure 2.1 for an example). The algorithm can be summarised as follows.

Input:

Sequence s (text), where $|s| = n$

Sequence t (pattern), where $|t| = m$ and $n \geq m$

Initialisation:

$$M[0,0] = 0$$
$$M[i,0] = M[i-1,0] - g, \text{ if } i > 0$$
$$M[0,j] = M[0,j-1] - g, \text{ if } j > 0$$

Recurrence relation:

$$M[i,j] = max \begin{cases} M[i,j-1] - g \\ M[i-1,j] - g \\ M[i-1,j-1] + \rho(t[i], s[j]) \end{cases}$$

At each $M[i,j]$, store the selected direction: `LEFT`, `TOP` or `DIAGONAL`.

Traceback:

```
$i = m,\ j = n$
**while** $i > 0$ or $j > 0$ **do**
    **if** $M[i,j] =$ DIAGONAL **then**
        append $s[j]$ to $p$
```

```
            append $t[i]$ to $r$
            $i = i - 1,\ j = j - 1$.
        else if $M[i,j] =$ TOP then
            append $s[j]$ to $p$
            append '–' to $r$
            $j = j - 1$.
        else if $M[i,j] =$ LEFT then
            append "–" to $p$
            append $t[i]$ to $r$
            $i = i - 1$.
        end if
    end while
    reverse $p,\ r$.
```

Output:

Alignment score: $M[m,n]$.

Alignment sequences: $p,\ r$.

(1) Input

$s = TGGCA,\ t = AGCA$

$g = 5,\ \rho =$

	A	C	T	G
A	4	−5	−5	−5
C	−5	4	−5	−5
T	−5	−5	4	−5
G	−5	−5	−5	4

(2) Initialisation

	j	0	1	2	3	4	5
i		–	T	G	G	C	A
0	–	0	-5	-10	-15	-20	-25
1	A	-5					
2	G	-10					
3	C	-15					
4	A	-20					

(3) Filling out M using the recurrence relation.

	j	0	1	2	3	4	5
i		–	T	G	G	C	A
0	–	0	-5	-10	-15	-20	-25
1	A	-5	↖ -4	-9	-14	-19	-16
2	G	-10	-9	↖ 0	← -5	-10	-15
3	C	-15	-14	-5	-4	↖ -1	-6
4	A	-20	-19	-10	-9	-6	↖ 3

(4) Traceback & Output

$p = TGGCA$

$r = AG-CA$

$score = 3$

Figure 2.1: Needleman-Wunsch Dynamic Programming algorithm simulation.

2.1.2 Local pairwise sequence alignment

Local alignment, in contrast to global alignment, is used in cases where the sequences share local regions of similarity but are not necessarily related from the beginning to the end. The alignment of any substring of *s* and any substring of *t* is a local alignment of *s* and *t*. It might seem that this additional search space dimension increases the computational complexity of a suitable local alignment algorithm. In 1981, Smith and Waterman (SW) proposed a slight modification to the NW dynamic programming algorithm that allows finding a local alignment, while keeping the same time and space complexity bounds [17]. The idea relies on the presence of negative scores in the scoring function, which can cause the running score to be negative. Adding a fourth term to the recurrence relation resets the score of *M[i, j]* back to zero whenever adding a substitution or a gap to the alignment would result in a negative score. This is what allows the local alignment algorithm to consider all possible starting positions in *s* and *t*. The rest of the algorithm is mostly unchanged, with the only other difference that the traceback starts at the cell in *M* with the maximum score and stops once a score of 0 is encountered.

Input:

Sequence s (text), where $|s| = n$

Sequence t (pattern), where $|t| = m$ and $n \geq m$

Initialisation:

$M[i,0] = 0$, $M[i,0] = 0$, for all i, j.

Recurrence relation:

$$M[i,j] = max \begin{cases} M[i,j-1] - g \\ M[i-1,j] - g \\ M[i-1,j-1] + \rho(t[i],s[j]) \\ 0 \end{cases}$$

At each $M[i,j]$, store the selected direction: `LEFT`, `TOP`, `DIAGONAL` or `STOP` (for zero score).

Traceback:

```
$i, j = arg\ max_{i,j}\ M[i,j]$
while $M[i,j] \neq$ STOP do
    if $M[i,j] =$ DIAGONAL then
        append $s[j]$ to $p$
        append $t[i]$ to $r$
        $i = i-1,\ j = j-1.$
    else if $M[i,j] =$ TOP then
        append $s[j]$ to $p$
```

```
            append '–' to $r$
            $j = j - 1$.
        else if $M[i, j]$ = LEFT then
            append "–" to $p$
            append $t[i]$ to $r$
            $i = i - 1$.
        end if
    end while
    reverse $p$, $r$.
```

Output:

Alignment score: $max_{i,j}\, M[i, j]$.

Alignment sequences: p, r.

This project implemented the recurrence relation on the GPU, while the traceback has only a CPU implementation. This is because the traceback is purely sequential, and also because the limited amount of GPU memory would pose a problem to executing the traceback for longer sequences on the GPU. It can also be argued that such a setup would make more sense in a real system - the GPU can have a work-pool for solving the recurrence relation, while the CPU could perform the traceback, thus increasing performance of the overall system. Although matters of the increased data transfer between the host and GPU would have to be better studied.

2.2 GPU architecture

The Nvidia GeForce GT 750M[1] chip (from here forth 750M) is a parallel, shared memory processor composed of 384 lightweight cores, dubbed CUDA cores. The cores are aggregated into 2 Streaming Multiprocessors (SMs), with each SM having 192 CUDA cores. Within a single SM, computation is orchestrated by a warp scheduler. A warp is a collection of 32 threads, which share an instruction pointer and execute in lockstep. It has a clear mapping to the hardware - a warp is executed on a 1024-bit SIMD unit, with a CUDA core corresponding to a single SIMD lane. A word on a GPU is thus 32-bits - a CUDA core is just a single 32-bit FMA functional unit, rather than a core which one might find in a CPU. Masking is used to handle divergent threads within a warp, with usually an additional time unit spent per each branch; a warp can take up to 32 time units to execute in the worst case. The challenge of GPU programming is to ensure that the CUDA cores are always performing useful computations, i.e. there should always be a warp to schedule, and all threads within a warp should follow the same execution path. This task is left to the programmer of the GPU, with no transistors on the GPU being spent on branch prediction or deep pipeline control circuits. This is probably the core difference between a GPU and a CPU - all the control circuits found in super-scalar CPUs are traded for compute units, with the job of extracting parallelism being left to the programmer. See Figure 1.2 for a conceptual comparison between the CPU and GPU.

[1]`https://www.Nvidia.com/en-gb/geforce/gaming-laptops/geforce-gt-750m/specifications/`

Memory access is often the main bottleneck in programs running on the CPU [8], and it is no different on the GPU. To combat this, GPU manufactures offer a hierarchy of memory spaces. Figure 2.3b shows a simplified view. The GPU is sometimes called a "throughput machine" - a common technique is GPGPU programming is to hide the memory access times by enough computation. The task of the programmer is to use access patterns which exploit the memory hierarchy. Local memory is a per-thread substrate of global memory and is designated to hold data that doesn't fit in registers ("spilled registers"). The compiler manages this space, and the programmer doesn't need to worry about it. Next in the hierarchy, each SM has its own shared memory which is accessible by all threads in a thread block. This space is addressable by the programmer, with all cores being able to access data in 1-3 cycles. To offer such a low latency, the size of the shared memory is limited to 48 KiB per SM, even on the latest GPUs. Shared memory is banked, with the number of banks equal to the warp size. If all threads within the warp access a separate bank, all requests execute in unit time. If there is more than one access to the same bank, it will be synchronised, thus stalling the whole warp (unless all threads access exactly the same address). There are many documented access techniques for shared memory, which are covered in the CUDA C++ programming guide [7]. Last in the hierarchy is global device memory, accessible by all SMs. It's composed of DRAM chips with an access latency of several hundred cycles, and has a size of 2 GB on the 750M. Global memory is cached transparently to the programmer, and has fewer access time guarantees compared to shared memory. Furthermore, GPU manufacturers only provide a weak memory model, meaning read and writes are not ordered, making some synchronisation and concurrency patterns hard to implement. Fortunately, CUDA provides access to an atomic Compare-And-Swap (CAS) operation, from which all other concurrency patterns can be implemented, albeit not always efficiently. Figure 2.2 shows a latency comparison between global memory, shared memory and registers in a GPU.

Memory	**Scope**	**Access**	**Latency in clock cycles**	**Size on GT 750M**
Global	global	read/write	400-600	2 GiB
Shared	block	read/write	4	48 KiB
Registers	thread	read/write	1	65536 per block

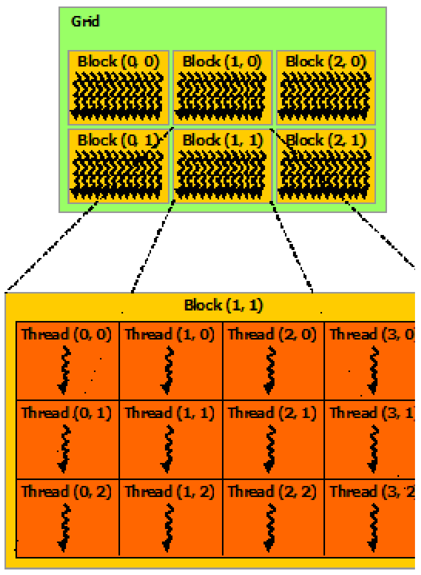
Figure 2.2: GPU memory hierarchy access times (CUDA Programming Guide [7]). Note that constant, texture and local memory are a substrate of global memory; they have the same latency, but varying access privilege.

2.3 CUDA programming model

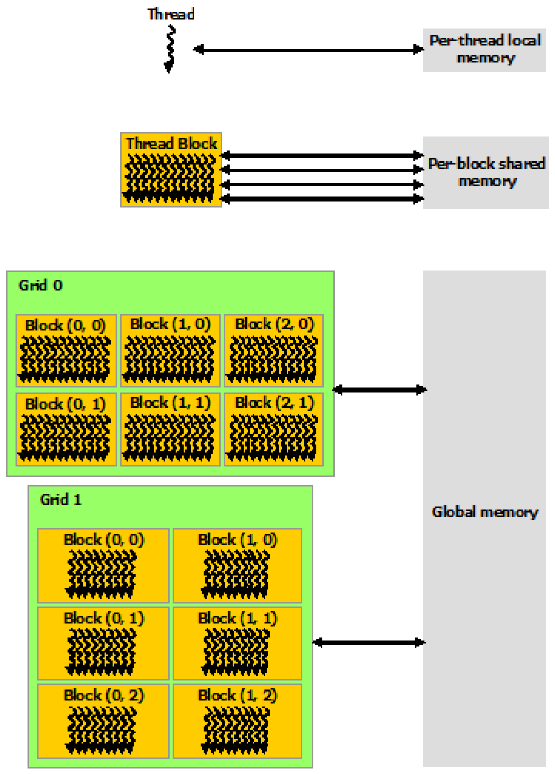
CUDA follow a Single Program Multiple Data (SPMD) paradigm and can be thought of as a special case of Multiple Instructions Multiple Data (MIMD) in Flynn's taxonomy of parallelism [16]. The programmer writes a scalar program for a single thread, called a "kernel". The number of threads is specified in a launch configuration. Threads are organised into thread blocks, and the thread blocks are further organised into a grid (as shown in figure

2.3a). The maximum number of threads in a thread block is 1024. The programmer specifies the number and dimensions of thread blocks for a particular kernel in a launch configuration. This interface is scalable by allowing a single kernel to be executed on various GPU configurations, with the actual hardware resources scheduling being left to the warp scheduler. Within the kernel, the programmer can refer to individual threads by their id, with the possibility of divergent execution based on this value. CUDA offers built-in synchronisation calls to enforce barriers at the thread block level. Synchronisation of threads across thread blocks is expensive and generally discouraged.

A GPU kernel executes concurrently with the host code. Most API calls in CUDA have an asynchronous version, allowing memory transfers between the host and the GPU to execute concurrently as well. This is possible thanks to the Direct Memory Access (DMA) engines on the GPU. A DMA engine allows the GPU to access system main memory asynchronously. To use this feature, the memory written to by the DMA engine must be "pinned"; it cannot be virtual memory managed by the Operating System. It is also possible to execute multiple CUDA kernels concurrently by using multiple CUDA streams. A CUDA stream is a command queue that handles host to device data communication, and device code execution. These features were important in the design of the GPU algorithms, which will be described shortly. First, a brief review of related work is conducted.



(a) Blocks are organised into a one-, two-, or three-dimensional grid of thread blocks.



(b) CUDA threads may access data from multiple memory spaces during execution.

Figure 2.3: Thread and memory hierarchy of the GPU (CUDA Programming Guide [7]).

2.4 Related work

Sequence alignment is a well researched topic in bioinformatics, with the two mainly used algorithms being Needleman-Wunsch and Smith-Waterman, described before. Established software tools, which implement these algorithms, have been used successfully for many years. FASTA is a DNA and Protein alignment tool initially developed in 1985 [14]. Before performing a full NW or SW algorithm, it uses a heuristic to decrease the search space for long sequences. Also in the 1980s, BLAST (Basic Local Alignment Search Tool) was introduced and has become one of the most widely used programs for sequence alignment to this day [2]. Similarly to FASTA, BLAST uses heuristics to decrease the search space, before applying a NW or SW algorithm. Because of this, both FASTA and BLAST cannot guarantee that the optimal alignment is found. An option to not use the heuristic exists in both tools. In recent years, "Bioinformatics tools as a service" have emerged, most notably from the European Bioinformatics Institute (EMBL-EBI) [11]. EMBL-EBI provides access to many bioinformatics tools (including NW, SW, BLAST) and a database of biological data through an unified browser interface and a REST API. This tool was used to verify correctness of the NW and SW implementation in this project.

Since the advent of GPGPU there has been substantial effort to use GPU resources to accelerate the NW and SW algorithms. Carroll's thesis offers insights into targeting the GPU for solving bioinformatics problems [4], with an efficient GPU implementation of semi-global sequence alignment in the `GPUGapsMis` tool [5]. This work investigates global and local sequence alignment on the GPU. GPU-BLAST is a GPU implementation of the BLAST tool suite, achieving a speedup of 3-4x compared to the CPU version [18]. CUDASW++ 3.0 is a very efficient SW implementation, which specifically targets the Nvidia Kepler architecture [10]. It reports an up to 10x speed up over a CPU version of BLAST. However, it uses SIMD intrinsic instructions available only in the Kepler architecture; newer Nvidia GPUs emulate them in software and consequently run slower. Neither of these tools address the issue of the working data set not fitting in GPU memory, with CUDASW++ 3.0 reporting results on sequences of maximal length of 35k.

A big part of this project was concurrent kernel execution, referred to in the literature as GPU inter-block communication or GPU concurrency. Alglave et al. study concurrent behaviour of GPUs, providing guidance on how to take the weak memory model of GPUs into account [1]. Xiao and Feng provide examples of several algorithms which benefit from concurrent kernel execution on GPUs [19]. These pieces of work show that it is possible and often desirable to have some kind of synchronisation between GPU SMs, contrary to the usual belief. For example, since the introduction of L2 cache in Kepler atomic operations have become a viable option for performant concurrent patterns.

Chapter 3

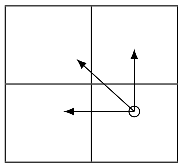
Design for parallelism

In this chapter, the design of the GPU algorithms for pairwise global and local sequence alignment is described. First, ideas about how to parallelise the recurrence relation are explored and compared. Next, data structures needed to support the parallelism are described, together with their mapping to the GPU memory hierarchy. Last, a modular design for aligning longer sequences based on concurrent GPU kernel execution is described.

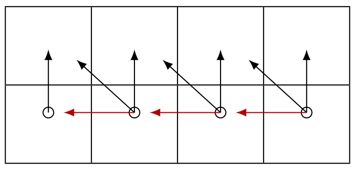
3.1 Dependencies in the recurrence relation

The recurrence relations, described in section 2.1, are identical with the difference that for local alignment the minimum score is 0. The ideas described herein apply equally well to global and local alignment.

One of the first steps in parallelising an algorithm is to look for dependencies between operations. If a number of operations have no dependencies between each other and don't write to the same memory location, then they can be safely executed in parallel. In NW and SW, a basic operation is the calculation of the score for a given cell in the M matrix. From the recurrence relation, it can be seen that a cell $M[i,j]$ depends on the cell to the left, top and left-diagonal, i.e. $M[i,j-1]$, $M[i-1,j]$ and $M[i-1,j-1]$:

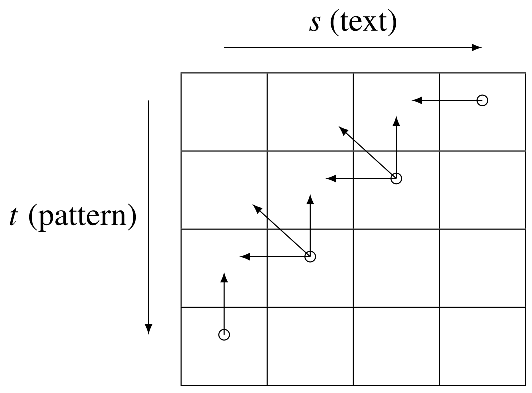


The first idea of parallelisation explored in this project was to calculate the scores within the rows in parallel, i.e. the parallelism was expressed horizontally:



It quickly became clear that this approach is not correct - the calculation of the scores to the left within a row has to be serialised (red arrows in the above diagram). A more sophisticated design was needed to better exploit the GPU. The effort wasn't all wasted though; some research into serialising computation on the GPU informed later design decisions.

Back to the drawing board, it was noticed that to get rid of the left dependency, the parallelism could be expressed across the diagonal:



For example, the filling out of the above 4×4 matrix M can be done as follows.

1. $M[0,0]$ is calculated.
2. $M[0,1], M[1,0]$ are calculated in parallel.
3. $M[0,2], M[1,1], M[2,0]$ are calculated in parallel.
4. $M[0,3], M[1,2], M[2,1], M[3,0]$ are calculated in parallel.
5. $M[1,3], M[2,2], M[3,1]$ are calculated in parallel.
6. $M[2,3], M[3,2]$ are calculated in parallel.
7. $M[3,3]$ is calculated.

All calculations within a single step (1, 2, ..., 7) can be executed in parallel, without any dependencies within the diagonal. This design was a bit more difficult to implement because the length of the diagonal isn't constant, first growing and then shrinking. The fact that some diagonals have only a few cells might seem like a potential performance bottleneck for the GPU. In reality, there will only be a performance hit when the length of the diagonal is not enough to saturate all active warps. This overhead becomes inconsequential as the length of the sequences grows to more than a couple hundred characters. Figure 3.1 shows that the throughput of the diagonal approach scales up more or less linearly with the length of the sequences, while the horizontal approach never performs better than 10 MCUPS (Millions of Cell Updates per Second). The diagonal approach also has some nice properties which make the data structures easier to design. This part is described next.

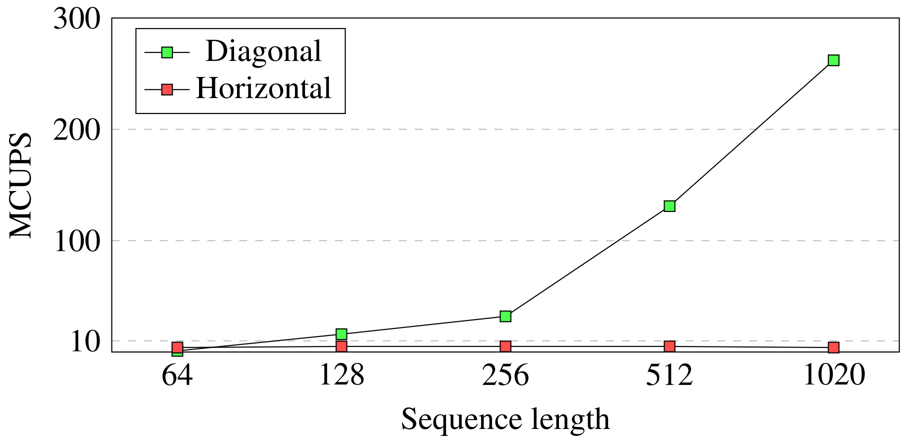


Figure 3.1: Throughput results for horizontal and diagonal parallelism. Measured in Millions of Cell Updates per Second (MCUPS).

3.2 Data structures

The previous section established the idea of expressing parallelism across the diagonal of the M matrix. The performance of the ultimate implementation also depends on how the data used in the recurrence relation are accessed. There are two parts that are considered here: the dynamic programming matrix M, and the scoring function ρ.

3.2.1 Dynamic programming matrix *M*

Consider the 4×4 matrix M used as an example in the previous section. Let the threads in a diagonal be $t_0, t_1, t_2, ...$, and successive diagonals in M be $d_0, d_1, d_2, ...$. The diagonal d_0 will contain t_0, the diagonal d_1 will contain t_0, t_1, and so on. First, note that there is data reuse across threads in a diagonal. In general, in the same diagonal the cell to the left of t_i is the cell to the top of t_{i+1}. The cell to the top of t_i in the diagonal d_j, becomes the cell to the left-diagonal of t_i in the diagonal d_{j+i}.

The properties of the data access into M suggest to keep the scores of M in shared memory. However, as seen in table 2.2, shared memory has a limited size. If all scores in the matrix are to be stored, this limit will quickly be reached. Luckily, only the scores from the last two diagonals are needed. The decision was made to use three buffers to store the scores of M in shared memory for the diagonals: d_i, d_{i-1}, d_{i-2}. The scores from the diagonals before that can be discarded since they are not used anywhere else. The selected directions at each cell have to be stored - they are later used in the traceback. They can be written back to global memory, to be later transferred to the host. From this point, the matrix M is assumed to only hold the selected directions at each cell, while the scores are only kept in the three shared memory buffers:

thisScores: stores the scores of d_i.

prevScores: stores the scores of d_{i-1}.

prevPrevScores: stores the scores of d_{i-2}.

Using these buffers, the recurrence relation for diagonal d_j becomes:

$$thisScores[i] = max \begin{cases} prevScores[i] - g \\ prevScores[i-1] - g \\ prevPrevScores[i-1] + \rho(t[i], s[j]) \end{cases}$$

To set up the buffers for diagonal d_{j+1}, the ping-pong buffer technique can be used (memory pointers are swapped, instead of moving data):

$$\begin{aligned} prevScores &= thisScores \\ prevPrevScores &= prevScores \\ thisScores &= prevPrevScores \end{aligned}$$

This access pattern guarantees that a single word in the buffers is only accessed by a single thread, avoiding any memory access serialisation within a warp - each thread within a warp will access a different memory bank, so that 32 look-ups can be served in unit time. Furthermore, with the row-major ordering of the M matrix, it can be guaranteed that there is a constant mapping between indexes into the diagonal and indexes into the pattern sequence throughout the execution of the algorithm, i.e. every thread can read its pattern character from memory just once. The access of the text characters is also predictable, with each thread accessing a unique byte avoiding any global memory access serialisation. This is visualised in figure 3.2.

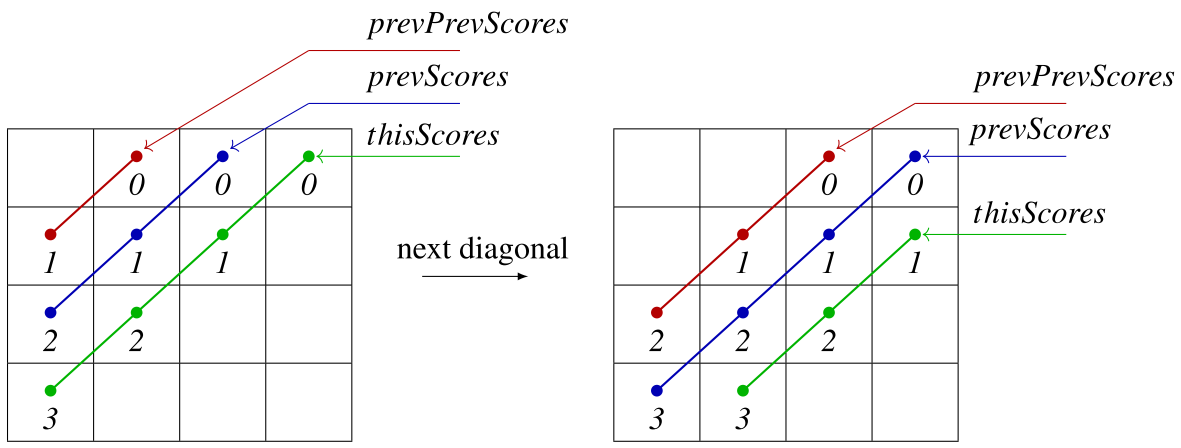


Figure 3.2: Shared memory buffers (green, blue, red) used in the recurrence relation. They are indexed from top to bottom, with every diagonal index maintaining the same row throughout execution.

In the previous chapter, the convention was adopted that the pattern sequence t cannot be longer than the text sequence (if this is not satisfied by the input, the two sequences can just be swapped without any difference to the result). From this follows, that the maximum length of the diagonal will always be equal to the number of rows in the matrix M. As mentioned in the overview of CUDA (section 2.3), the maximum number of threads within a thread block is 1024. With the above approach the maximum size of M that can be worked on by a thread block is $1024 \times m$, where m the length of the text sequence. This is not a hard limit, but increasing the number of rows beyond the maximum number of threads in a single thread block would require code causing significant thread divergence. Another possible limit is the global memory size, so if m is large and the $1024 \times m$ M matrix storing the selected directions doesn't fit into global memory, then the size needs to be scaled down. With the upper limit of 1024 threads, each buffer has to hold a maximum of 1024 values. Assuming 32-bit values, the 3 buffers need 24 KiB, fitting comfortably in the 48 KiB size shared memory, even leaving room for the scoring Look Up Table described next.

3.2.2 Scoring function ρ

The signature of the scoring function is $\rho : \Sigma \times \Sigma \to \mathbb{Z}$. The longest supported alphabet is the alphabet of amino acids used in Protein sequence alignment, with a length of 23 characters. The domain of ρ has $23 * 23 = 529$ elements, making it a good candidate to be implemented as a Look Up Table (LUT). Assuming 32-bit score values, the LUT needs less than 1 KiB of space. The decision was made to place the ρ LUT is shared memory:

scoreMatrix: stores the function ρ as a Look Up Table.

In order for this to work, the Σ characters have to be mapped into the range $[0,23]$ for protein sequences, and $[0,4]$ for DNA sequences. With this internal representation of the sequences, calculating the score of an alignment translates to one Fused-Multiply-Add (FMA) operation and one shared memory access, as shown in figure 3.3.

$$\Sigma_{DNA} = \{A,C,T,G\} \quad \xRightarrow{\text{Transform.}} \quad \Sigma'_{DNA} = \{0,1,2,3\}$$

$$s = A,T,T,C \quad \xRightarrow{\text{Transform.}} \quad s' = 0,2,2,1$$

$$t = A,G,C,A \quad \xRightarrow{\text{Transform.}} \quad t' = 0,3,1,0$$

$$\begin{aligned} \rho_{DNA} = \{&((A,A),5),\\ &((A,C),-2),\\ &((A,T),-4),\\ &...\} \end{aligned} \quad \xRightarrow{\text{Transform.}} \quad \begin{aligned} \rho'_{DNA} = [&5,-2,-4,-4,\\ &-4,5,-4,-1,\\ &-4,-2,5,-3,\\ &-4,-4,-4,5] \end{aligned}$$

$$\rho_{DNA}(T,G) = \rho'_{DNA}[2 * |\Sigma_{DNA'}| + 3] = \rho'[11] = -3$$

Figure 3.3: Sequence characters are transformed into an internal representation to make the scoring function implementable as a Look Up Table.

3.3 Concurrent kernel execution

The design of the dynamic programming matrix data structure is henceforth referred to as a 'kernel'. It was mentioned that the maximum number of rows in a kernel is 1024. In this section, it is shown how this kernel can be used as a building block to support the alignment of longer sequences. It is assumed that the $1024 \times m$ matrix M fits in global memory; if it doesn't, the number of rows can be scaled down until it does.

The first thought that came to mind when trying to align longer sequences is to use the same kernel to fill in the rows 0..1023, then move on to the rows 1023..2047, and so on, progressing sequentially. Surprisingly though, there is a degree of parallelism between subsequent kernel invocations, coming from the fact that the matrix is filled using the diagonal approach. This was noted while implementing the data communication between sequential kernel invocations. There is a special kind of parallelism called "pipeline parallelism" in the subsequent kernels - once the first column in the first kernel is finished, the next kernel can already start, executing concurrently with the first. In other words, once the *thisScores* diagonal of $kernel_i$ reaches its maximum length, $kernel_{i+1}$ can start executing while $kernel_i$ is still computing its later diagonals. Figure 3.4 shows a conceptual view.

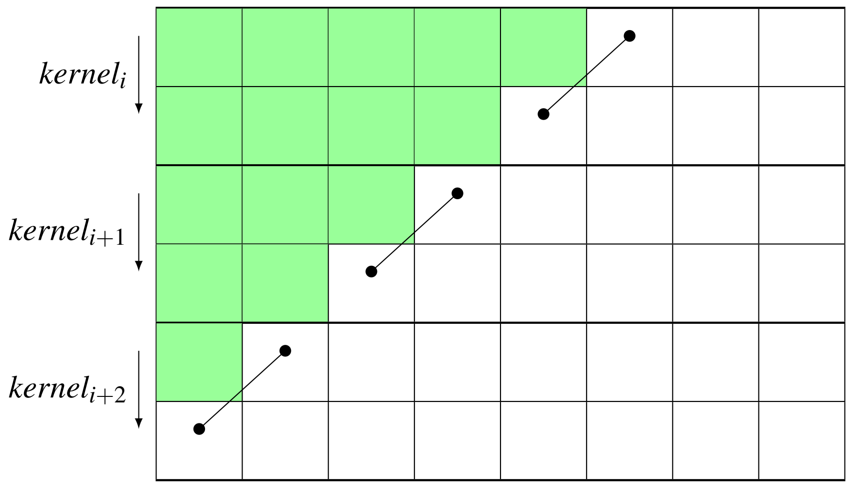


Figure 3.4: Three 2×8 kernels are able to execute concurrently, once the pipeline is sufficiently filled. Green cells represent work already done, and the black lines represent the currently executing diagonals.

With each kernel being executed on a dedicated Streaming Processor (SM), multiple SMs of a GPU can be employed to work on the algorithm, increasing the overall efficiency. Kernels can be assigned to available SMs in a round robing fashion, for example, in the case of 2 SMs, once one SM finishes with $kernel_i$, it can be assigned to work on $kernel_{i+2}$. This scheduling is done by the CUDA runtime, transparently to the programmer.

In pipeline parallelism there is usually a constant cost associated with filling of the pipeline. In this case, this cost can be easily approximated. Let k be the number of rows in a kernel (2 in the example from figure 3.4). Assuming unit time for calculating one diagonal, $kernel_{i+1}$ has to wait k time units before starting its work, $kernel_{i+2}$ has to wait $2k$ time units, and so on. If m is much larger than n (which is sometimes the case in local alignment), then this technique can increase the efficiency of the GPU algorithm significantly. Since there is a one-to-one correspondence between the number of SMs and number of kernels, GPUs with more SMs should perform better. The 750M only has only 2 SMs, but newer Nvidia GPUs have tens of SMs.

Inter-kernel data communication is the core challenge of this approach. $kernel_{i+1}$ has to wait for $kernel_i$ to send its results before it can start executing. This problem is simpler than the general problem of data communication between concurrent threads. *One-directional* communication is all that is needed. In concurrency theory, this pattern is called a fence or a barrier. A design decision was made to have a barrier data structure reside in GPU global memory, where all kernels can query and update it. Because of the weak memory ordering of the GPU architecture, this is dangerous territory. Care was taken in designing operations on the data structure to eliminate any determinacy races. Only one kernel writes to a location in the data structure at a time, while the one-to-one mapping between kernels and SMs ensures that there is no starvation.

Let *columnState* be an array with a cell holding a $kernel_{id}$ and *score* for each column in the M matrix. This data structure has two operations. For $kernel_{i+1}$, it must support access to the result of $kernel_i$, when it is available; and for $kernel_i$ it must support storing its result and removing a barrier:

```
function GET_PREV_SCORE($kernel_{id}$, $column_{id}$)
    while $columnState[column_{id}].kernel_{id} \neq kernel_{id}$ do
        wait
    end while
     return $columnState[column_{id}].score$
end function

function SET_DONE($kernel_{id}$, $column_{id}$, $newScore$)
    atomically do
        $columnState[column_{id}].score = newScore$
        $columnState[column_{id}].kernel_{id} = kernel_{id} + 1$
    end atomically
end function
```

The semantics of the two operations guarantee that only one kernel will have access to the score of a column state at any given point in time - for any given $kernel_{id}$ (1) GET_PREV_SCORE() and SET_DONE() are called exactly once; (2) GET_PREV_SCORE() is always called before SET_DONE().

3.3.1 Discussion

A natural question to ask is "why not use the kernel size launch configuration available in CUDA to employ multiple SMs, instead of going to all the trouble of using concurrent kernels?". The main motivation behind the decision to use concurrent kernels was that each kernel needs to transfer its part of M matrix to the host after it finishes. The limited size of GPU global memory was the reason behind decomposing the problem into multiple kernels in the first place. Also, using the launch configuration would not make the problem of communicating data across SMs go away. It may well be that there is a more elegant or efficient way to handle this problem. One way could be to let CUDA implicitly manage the intermediate data transfers by using unified CPU and GPU memory. This approach was briefly explored in the implementation, however, the performance was much poorer compared to using explicit data transfers, so this avenue was abandoned. It could be argued that the more fine-grained control over kernel execution that comes with concurrent kernels is worth paying the price of a little more complexity in the implementation.

Chapter 4

Scalable implementation

This chapter describes the implementation of the ideas developed in the design stage in the form of the `alignSequence` program. First, the dataflow graph of the program is depicted. Next, the implementation of the GPU algorithms for global and local alignment is described. Finally, the implementation of the concurrent kernels idea is explained.

4.1 Program data flow

With the idea of parallelism and the supporting data structures established, a high level overview of how the data flow through the program is presented. The program starts by an invocation through the command line:

```
$ ./alignSequence
Usage: alignSequence [-p -d -c -g] [--score-matrix <int>]
                     [--gap-penalty <int>]  <file> <file>
       -d, --dna             - align dna sequences (default)
       -p, --protein         - align protein sequence
       -c, --cpu             - use cpu device (default)
       -g, --gpu             - use gpu device
       --global              - use global alignment (default)
       --local               - use local alignment
       -s, --score-matrix    - next argument is a score matrix file
       --gap-penalty         - next argument is a gap penalty (default 5)
```

At a minimum, the user is expected to supply two files with the sequences to be aligned. If neither of the flags are specified, the default configuration is used. The supplied arguments are passed to `parseArguments()` where they are parsed and checked for correctness, together with any supplied files. If the arguments are wrong, for example, the supplied sequences use characters not present in the default alphabet, then a suitable error message is printed. When the program finds itself in other incorrect states, like running out of memory for the sequences, then other appropriate error messages are printed. Once parsed successfully, a `Request` object is built which contains all the necessary data for an alignment, and a `Response` object which will hold the resulting alignment (see figure 4.1 and the `SequenceAlignment.hpp` header file in the code submission).

```
struct Request
{
    programArgs deviceType;
    programArgs sequenceType;
    programArgs alignmentType;
    char *textBytes;
    uint64_t textNumBytes;
    char *patternBytes;
    uint64_t patternNumBytes;
    const char *alphabet;
    int alphabetSize;
    int scoreMatrix[...];
    int gapPenalty;
};
```

```
struct Response
{
    char *alignedTextBytes;
    char *alignedPatternBytes;
    uint64_t numAlignmentBytes;
    uint64_t startInAlignedText;
    uint64_t startInAlignedPattern;
    int score;
};
```

Figure 4.1: Request and Response objects used internally to pass data to and from the alignment algorithms (memory allocation and deallocation omitted here).

Both objects are responsible for their allocated memory and will be destroyed once going out of scope - following the Resource Acquisition Is Initialisation (RAII) technique in C++. Next, there is a runtime dispatch to fill in the dynamic programming matrix based on the alignment type (global or local) and on the selected device (CPU or GPU). Once the *M* matrix is filled, the traceback algorithm is executed on the CPU. Finally, the aligned sequence is printed in a structured way, along with some useful information. Figure 4.2 shows an example output. The high level data flow of the program is summarised in figure 4.3.

```
$ ./alignSequence --gpu data/dna/01.txt data/dna/02.txt

 1 -ATGAAG-T-T-GTTCGC-CTTACTTTTAATTCTACTCT-CTC-CTCGAG   50
    ||.||. | | |.|||. ||||.|...|..   || |. |.| | | ||
 1 CATAAAACTCTCGGTCGGGCTTAGTACCAGG---AC-CGGCGCAC-C-AG   50

51 ATTCGTC   57
   |.| |
51 AGT-G--   57

# Length:       57
# Identity:     28/57 (49.1%)
# Gaps:         16/57 (28.1%)
# Score:        8
```

Figure 4.2: Example output from the `alignSequence` program.

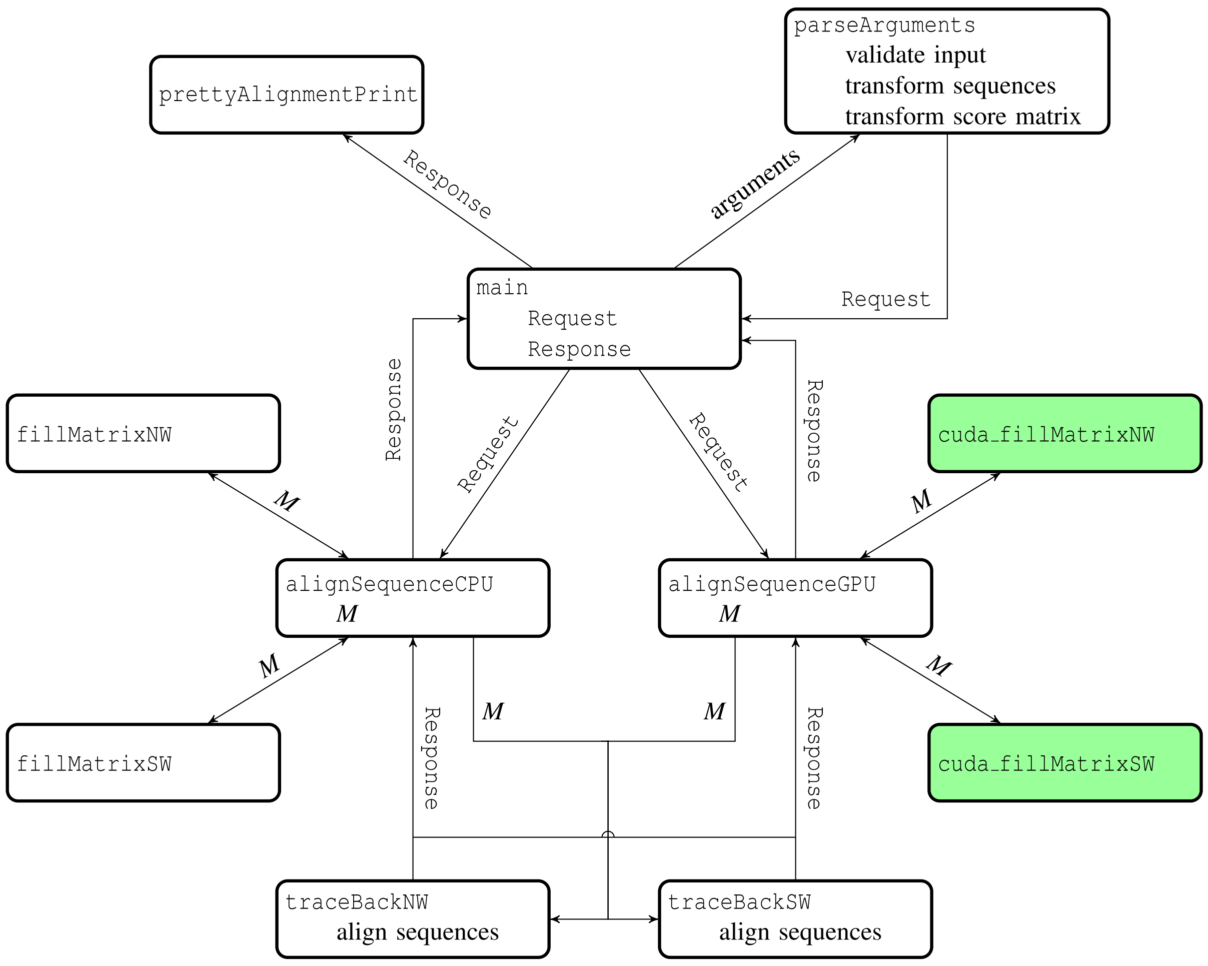


Figure 4.3: High-level data flow of `alignSequence`. Green boxes represent GPU kernels.

4.2 Needleman-Wunsch (NW) GPU kernel

In this section, the implementation of the NW GPU kernel is discussed. The kernel was implemented using CUDA C++ and can be found in the `alignSequenceGPU.cu` file in the code submission. Pseudocode highlighting the approach taken during the implementation is shown in Listing 1. The kernel takes data structures and variables described in the Design chapter as kernel parameters, with their memory handling described shortly. The *startRow* and $kernel_{id}$ parameters, together with the *columnState* data structure make it possible for the kernel to work on any part of the M matrix, supporting the modularity aimed at in the design stage. The very first row and column of the recurrence relation follow from the initialisation and are not computed. There are two for-loops to fill the matrix M with a `DIRECTION::LEFT`, `DIRECTION::TOP` or `DIRECTION::DIAG` value for each cell, needed for the traceback. The first loop deals with the case when the diagonal is growing, and the second with the case when it is shrinking. Having two loops made it easier to implement the indexing into the three shared memory score buffers, minimising the number of thread divergences inside of a warp.

The kernel makes use of helper functions: `ping_pong_buffers()`, `get_prev_score()`, set_done(), and choose_direction_NW(), all of which have been described in the previous chapter, and are not repeated here. It is worth noting that the functions used for data communication between kernels are executed just on one thread - a common pattern when implementing any kind of concurrency on the GPU.

Listing 1 fillMatrixNW

```
// Initialisation. Set up shared memory buffers, copy score matrix...
// Each thread gets one row (one pattern letter).
const char patternByte = patternBytes[tid + startRow - 1];

/* First half of matrix filling */
int fromLeft = 0, fromDiag = 0, fromTop = 0, fromPrevKernel = 0, diagonalSize = 0;
for (int i_text = 1; i_text < numCols; ++i_text)
{
    ping_pong_buffers(thisScores, prevScores, prevPrevScores);

    diagonalSize = min(diagonalSize + 1, numRows);
    const int idxInRow = i_text - tid;

    // Thread 0 (= row 0) gets score from the previous kernel.
    if (tid == 0 && kernelId > 0)
        fromPrevKernel = get_prev_score(colState, i_text, kernelId);
    __syncthreads();

    if (tid < diagonalSize)
    {
       fromLeft = prevScores[tid];
       fromDiag = prevPrecScores[tid - 1];
       fromTop = (tid == 0) ? fromPrevKernel : prevScores[tid - 1];

       const char textByte = textBytes[idxInRow - 1];
       const int scoreMatrixIdx = textByte * alphabetSize + patternByte;

       // Recurrence relation returning score and selected direction.
       auto scoreDirPair = choose_direction_NW(fromLeft, fromTop, fromDiag, gapPenalty,
                                               scoreMatrix[scoreMatrixIdx]);
       thisScores[tid] = scoreDirPair.first;
       M[tid*numCols + idxInRow] = scoreDirPair.second;

       // Last row in kernel updates the columnState data structure.
       if ((tid + startRow) == endRow)
           set_done(columnState, idxInRow, scoreDirPair.first, kernelId);
    }
}

/* Second half of matrix filling */
for (int i_pattern = 1; i_pattern < numRows; ++i_pattern)
    // Fill the rest of M, with adjusted indexing and no get_prev_score().

// Matrix M is filled. The score is at columnState[numCols - 1].score
```

4.3 Smith-Waterman (SW) GPU kernel

The implementation of the SW GPU kernel (also found in `alignSequenceGPU.cu`) is similar to the NW kernel, with two main differences. Firstly, `choose_direction_NW()` has changed to `choose_direction_SW()`, to handle the different recurrence relation. Secondly, an additional `DIRECTION::STOP` value was introduced which will be stored in M for every cell with a 0 score. Thirdly, the kernel has to keep track of the maximum score and the corresponding index in M. The score of a local alignment is not guaranteed to be in the *columnState* data structure, like in the NW kernel. To keep track of the best score, each thread within the kernel has a *thisBestScore* and *thisBestScoreIdx* local variable. Each time a thread calculates the score of a cell, it updates both of these variables if the newly calculated score is higher than its *thisBestScore*. Thus, at the end of the kernel, every thread will hold the maximum score it has seen, and the index within M of that score needed for the traceback. The last step is to choose the maximum score out of every active thread - a set of values needs to be reduced to one.

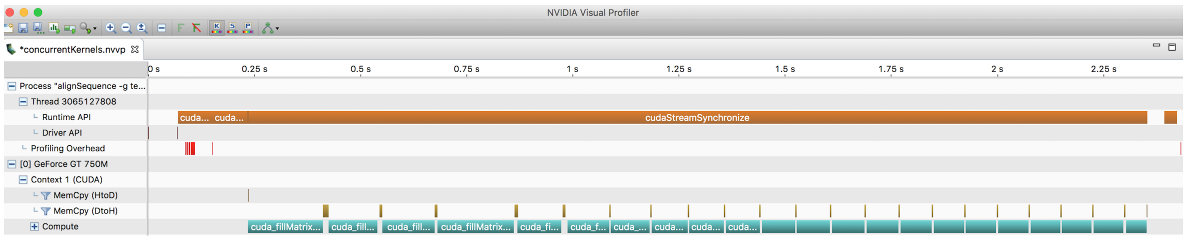
Reductions are a common pattern in parallel programming. The max-reduction at the end of the SW kernel is guaranteed to be within a single SM and involve no more than 1024 threads. Any of the three score buffers can be re-used to store the *thisBestScore* values for each thread in shared memory. Once in addressable memory, a simple tree reduction can be performed (shown in Listing 2). At the end, each thread can check if the maximum score is equal to its score, and the first thread to succeed this test will set the index within M corresponding to that score. There can be several local alignments and the GPU kernel will choose one non-deterministically.

Listing 2 `max_reduce`

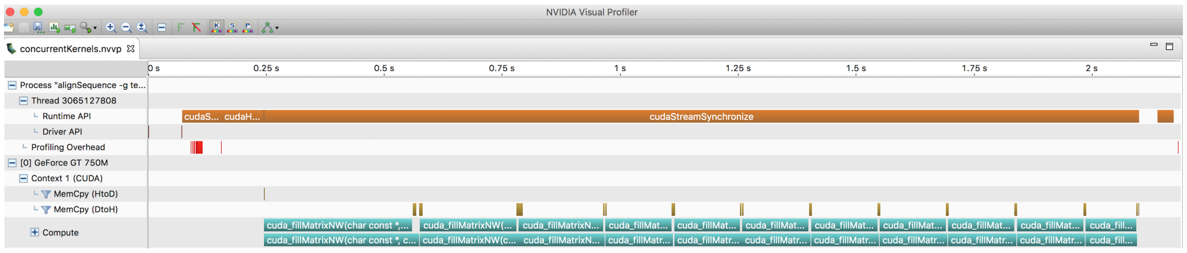
```
/// Given an array "values", compute the maximum of of the N first items,
/// and store it in values[0]. Example:
///    [2, 0, 3, 1, 0]
///    [2, -, 3, -, 0]
///    [3, -, -, -, 0]
///    [3, -, -, -, -]
__device__ __forceinline__ void max_reduce(int *values, const int N)
{
    const int tid = threadIdx.x;

    // Tree reduction with log2 levels.
    for (int pow2 = 1; pow2 < N; pow2 *= 2)
    {
        // Only threads at pow2 indexes do work.
        if ((tid & pow2) == 0 && (tid + pow2) < N)
            values[tid] = max(values[tid], values[tid + pow2]);

        __syncthreads();
    }
}
```

4.4 Concurrent kernel orchestration

CUDA kernels are invoked by the host. By default, subsequent kernel invocations are blocking; executing sequentially. CUDA streams, which are nothing else than command queues, can be used to launch concurrent kernels and to overlap GPU data transfer with computation. Figure 4.4 visualises the difference in the execution timeline between sequential and concurrent kernels.



(a) A single SM works on a sequence of kernels.



(b) Two SMs work on a sequence of kernels concurrently.

Figure 4.4: Output from the Nvidia Visual Profiler tool, showing the timeline of a sequential and concurrent kernel execution.

A CUDA stream has to be created and managed as any other resource on the GPU. In the implementation, a `initMemory()` function was created whose responsibility it is to prepare the GPU resources for the kernel launch, including CUDA streams. This involves:

- Querying the GPU for the number of SMs and the size of the global memory. These parameters determine how many kernels can be launched concurrently, and how big they can be.
- Setting the number and dimensions of kernels, and the number of CUDA streams.
- Allocating GPU resources for the required data structures and CUDA streams.
- Transferring the sequences and scoring matrix to the GPU.

Dually, there exist a `cleanUp()` function which destroys the created resources at the end of the kernel, or in case of any error.

Once the number of kernels, CUDA streams and other parameters are known, the kernels can be launches as in Listing 3. The number of CUDA streams is equal to the number of SMs on the GPU. It is quite likely that the number of kernels needed is larger than the number of

CUDA streams. In this case, a single CUDA stream would get multiple kernels to work on, in a round-robin scheduling fashion.

Within each stream, after a kernel has finished, the calculated direction in the device M matrix are transferred back to the host by the same stream. This ensures that the stream doesn't start the next kernel (potentially overriding data) until the transfer has finished. Kernels scheduled on other streams are free to execute concurrently with the data transfer, thanks to the Direct Access Memory (DMA) engines on the GPU. As discussed in the Background chapter, the memory address written to by the DMA engine must be physical, not virtual. CUDA provides an API to allocate pinned physical memory on the host, which is taken care of in the `initMemory()` function.

An important thing to note from Listing 3 is that each kernel gets a unique $kernel_{id}$ value, needed for the `get_prev_score()` and `set_done()` functions, described in the Design chapter.

Listing 3 Concurrent kernel orchestration

```
for (int i_kernel=0; i_kernel < numKernels; ++i_kernel)
{
   // Round-robin scheduling for CUDA streams.
   auto i_stream = i_kernel % numCuStreams;
   currCuStream = cuStreams[i_stream];
   // Kernels are enumerated from 0 to numKernels-1.
   kernel<<<1, numThreads, sharedMemSize, currCuStream>>> (.. i_kernel, d_M[i_stream]);
   // Each kernel transfers its results back to the host,
   // while other kernels are still executing.
   cudaMemcpyAsync(curr_h_M, d_M[i_stream], numThreads*numCols, ..);
}
```

Chapter 5

Testing & Evaluation

In this chapter, the correctness and performance tests used in the project are described. The first section goes over the correctness verification of the `alignSequence` program. Next, the methodology behind the performance tests is explained and the results are presented. At the end, the concurrent kernel approach is evaluated and discussed.

5.1 Correctness verification

The project, especially in the early stages of the implementation, followed a Test Driven Development (TDD) methodology. Before implementing a given function, a test case was written specifying the expected behaviour. This methodology helped to stay on track during the implementation by breaking the program down into individual pieces, which could be tested in separation.

The Catch2 testing framework was used to implement the unit and functional tests [6]. It is a single header lightweight library, providing several macros for the usual unit test types. Listing 4 shows an example unit test, written using Catch2, which verifies that a score matrix file was parsed correctly.

Listing 4 Example unit test

```
TEST_CASE("parseScoreMatrixFile")
{
    SequenceAlignment::Request request;
    request.alphabet = SequenceAlignment::DNA_ALPHABET;
    request.alphabetSize = SequenceAlignment::NUM_DNA_CHARS;
    parseScoreMatrixFile("scoreMatrices/dna/blast.txt",
                         request.alphabetSize, request.scoreMatrix);

    CHECK(getScore('A', 'A', request.alphabet,
                   request.alphabetSize, request.scoreMatrix) == 5);
    CHECK(getScore('G', 'T', request.alphabet,
                   request.alphabetSize, request.scoreMatrix) == -4);
    // ...
}
```

The same framework was used for the functional testing of the complete `alignSequence` program. The first stage of the project was to develop a CPU implementation of the NW and SW algorithms. The CPU version was tested against the EMBL-EBI online toolkit suite[1]. Once the CPU version was working correctly, the GPU version was tested against that.

In the submission, there are 20 protein and 20 DNA sequences of varying sizes. These data comes from GenBank[2], which is a publicly accessible database of DNA and protein sequences. The test suite aligns every possible pair of sequences on the CPU and GPU and compares the alignment score and aligned sequences. This is done both for local and global alignment, with the caveat that in local alignment only the score is checked since there can be multiple optimal local alignments and the GPU algorithm chooses one non-deterministically. Overall, there are over a 1000 checks performed. The test suite was ran on three Nvida GPUs, each from a different generation, and none reported any errors. Access to high-end Nvidia GPU cards was possible through the University of Liverpool Barkla compute nodes[3]. The functional and performance tests described in this chapter can be easily repeated if one has access to Barkla; there are only two commands needed to run the test or benchmark suite. The steps to do that are described in the README, which is available in the code submission, and is also appended to this dissertation for convenience (appendix B).

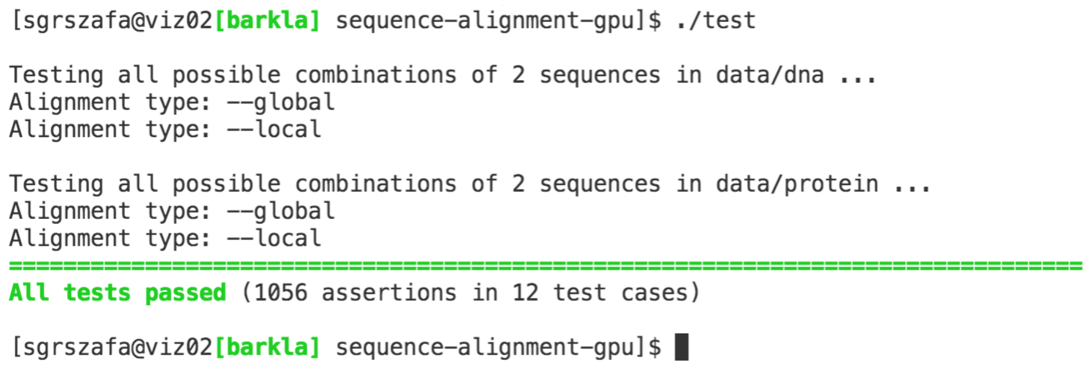

```
[sgrszafa@viz02[barkla] sequence-alignment-gpu]$ ./test

Testing all possible combinations of 2 sequences in data/dna ...
Alignment type: --global
Alignment type: --local

Testing all possible combinations of 2 sequences in data/protein ...
Alignment type: --global
Alignment type: --local
===============================================================================
All tests passed (1056 assertions in 12 test cases)

[sgrszafa@viz02[barkla] sequence-alignment-gpu]$ █
```

Figure 5.1: Test suite output on a Barkla compute node.

5.2 Performance results

For evaluating the performance of the NW and SW implementations, three types of performance tests were performed. The timer resolution used in the tests is in microseconds, but timing results are reported in seconds. Data used in these benchmarks were randomly sampled from the protein alphabet. This is justified by the fact that there are no data-dependent steps in the NW and SW recurrence relations. The traceback step of the algorithms is data-dependent, however, it is not the main point of the evaluation.

[1]`https://www.ebi.ac.uk/Tools/psa/`

[2]`https://www.ncbi.nlm.nih.gov/genbank/`

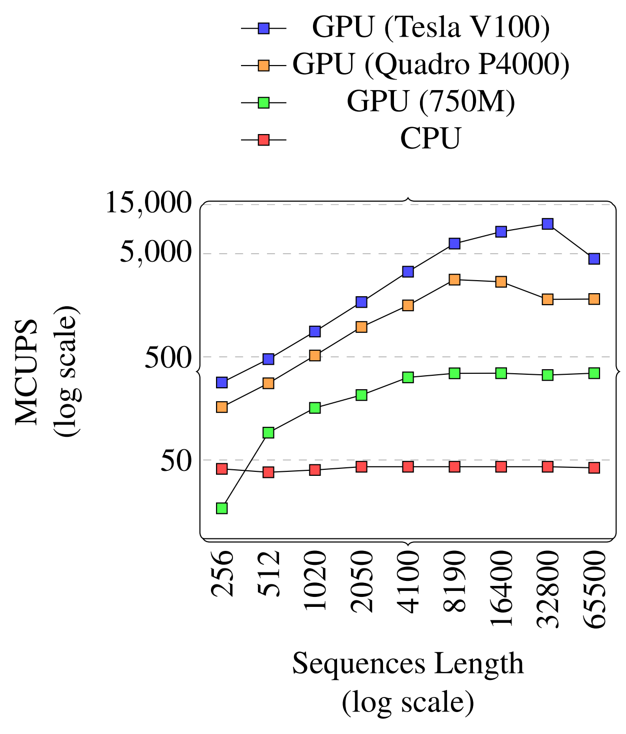
[3]`https://www.liverpool.ac.uk/csd/advanced-research-computing/facilities/high-performance-computing/`

In addition to performance results from various classes of GPUs, the results of a reference CPU implementation running on a single thread of an Intel i7-4850HQ mobile CPU are presented. They are compared to results from an equivalent class mobile GPU (both chips would typically be found in consumer laptops from around 2013). It is *not* an objective comparison between the two technologies - the reference CPU implementation from this project was not as highly tuned as the GPU implementation, it is not using the full potential of a single core with all the SIMD units, and it is only using one of the four available cores. The sole purpose of the comparison is to give an indication of the level of improvement one might expect when running a data parallel kernel on a CPU and GPU. Results from two other GPUs are also reported: the Nvidia Quadro P4000, which is a mid-range card with 14 SMs, and the Nvidia Tesla V100, which is a high end card with 80 SMs. More detail about all these chips and the experimental setup can be found appendix A. The three benchmark types are now described in more detail.

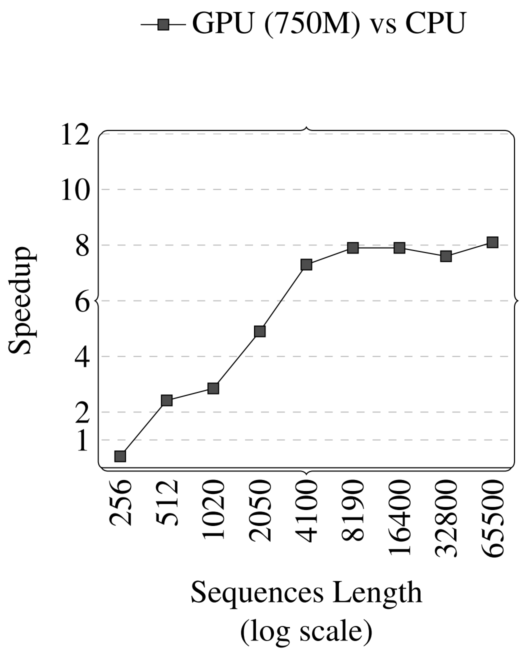
Throughput of the filling out of the matrix M is measured in Millions of Cell Updates Per Second (MCUPS). Traceback time is not included. The result is obtained by dividing the number of cells in the M matrix, divided by the measured time taken to fill the matrix. Thus, the throughput results are equivalent to measuring the latency of filing out the matrix, without traceback. For the GPU implementation, the timer is started once all data is transferred to the device, and is stopped once all results are transferred back to the host. The same test configuration is ran five times and the best result is selected.

Figure 5.2 shows the results. In global alignment, the two input sequences have equal length, whereas in local alignment the second sequence is fixed at 32k characters. The NW results show that the throughout of the GPU cards grows as more data is available. The CPU implementation, on the other hand, is not able to scale its throughput with more data - its peak is 49 MCUPS for all sequence lengths. The 750M card reaches peak throughput of **347 MCUPS** once sequences have more than 4000 characters. The speedup graph shows that the GPU version is up to **8x** faster on sufficiently long sequence inputs, compared to the CPU version. For sequences shorter than 512 characters, the GPU cannot achieve a higher throughput than the CPU. The other GPUs continue to scale up their throughput beyond that of the 750M, with the best NW throughput result of **9724 MCUPS** reported on the Tesla card for a 32768×32768 sequence input. Of interest is the drop in performance for the Quadro and Tesla cards for longer sequences. This point will come back into discussion in the next section.

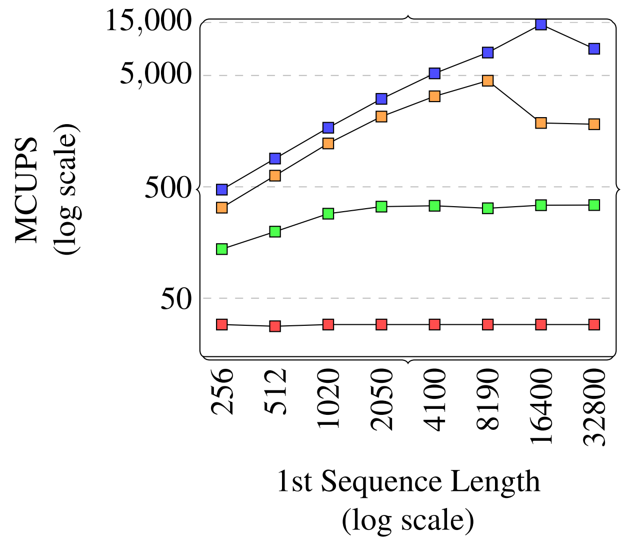
The throughput results improve for the SW algorithm. This is because the GPU implementation performs better on input instances where one sequence is longer than the other. Why exactly that is will also be explained in the next section. When comparing the mobile chips, the GPU implementation shows up to a **12x** improvement over the CPU. The added speedup comes from the fact that the CPU implementation achieves less throughput in the SW algorithm. The higher end GPUs report higher throughput for SW, with the best result of **14,354 MCUPS** achieved when aligning a 16384×32768 sequence pair on a Tesla card.



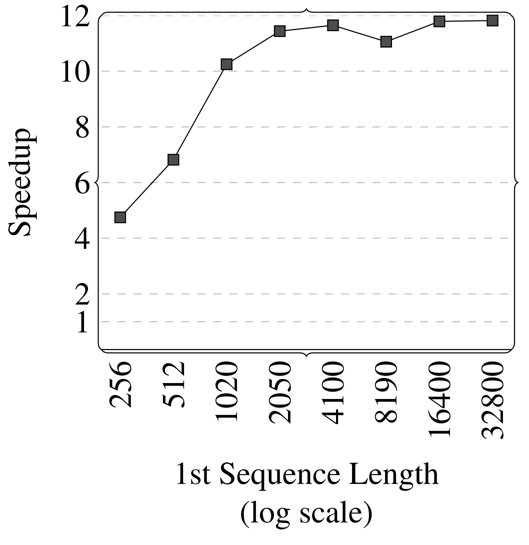
(a) Throughput (NW)



(b) Speedup (NW)



(c) Throughput (SW)



(d) Speedup (SW)

Figure 5.2: Throughput results for filling out the matrix *M* in NW (top) and SW (bottom). In NW, sequences are of equal length. In SW, the second sequence length is fixed at 32k.

End-to-End latency of the `alignSequence` program is the combined time to solve the recurrence relation and to perform the traceback. The timer is started before passing the `Request` object to the `alignSequenceGPU` or `alignSequenceCPU` function, and it is stopped once the `Response` object has been fully filled with the alignment score and resulting alignment sequences. The GPU speedup over the CPU is expected to drop in this benchmark, which is confirmed by the results in figure 5.3.

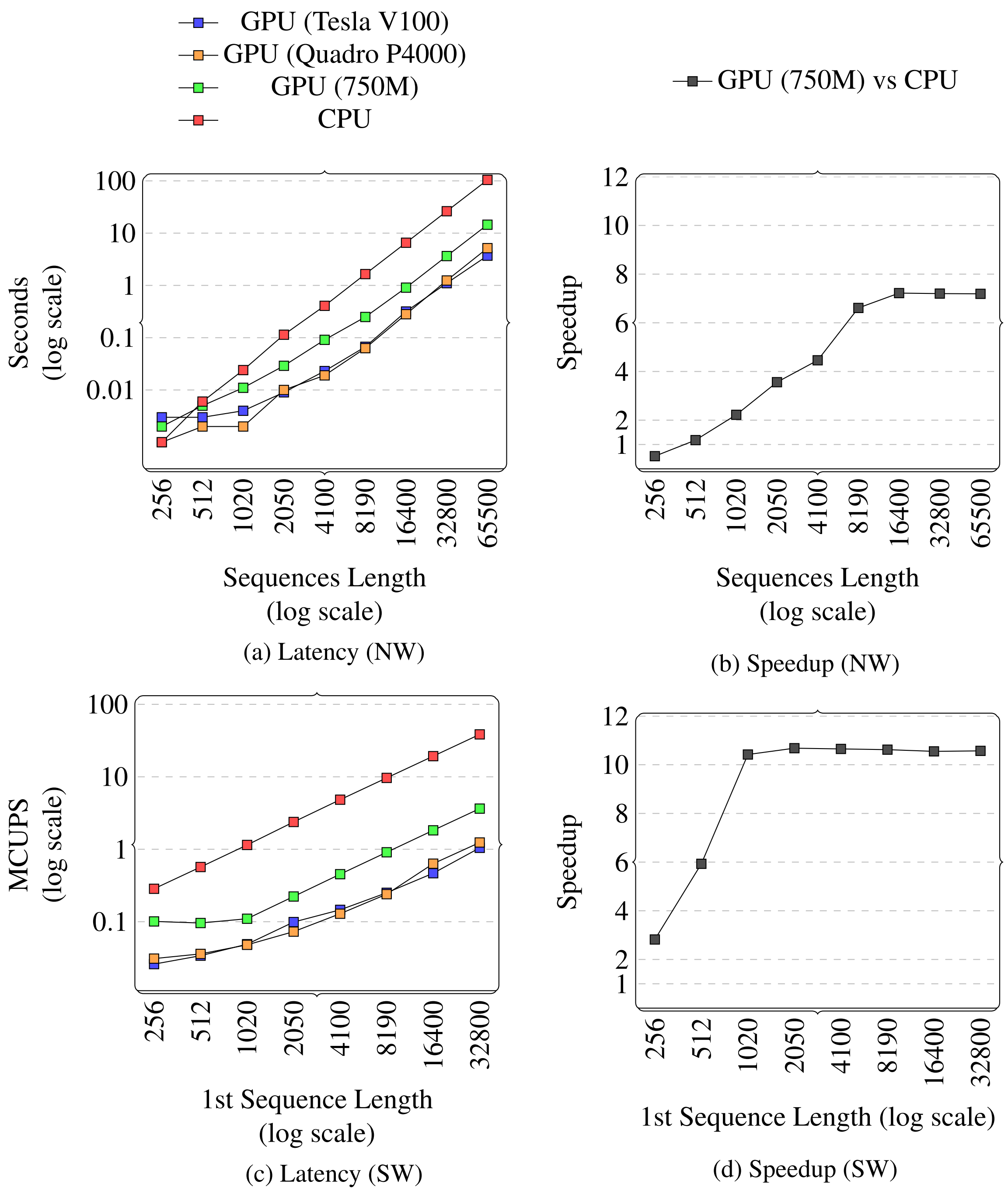
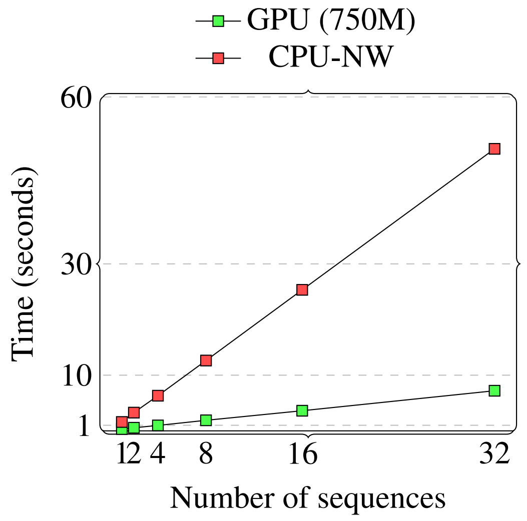


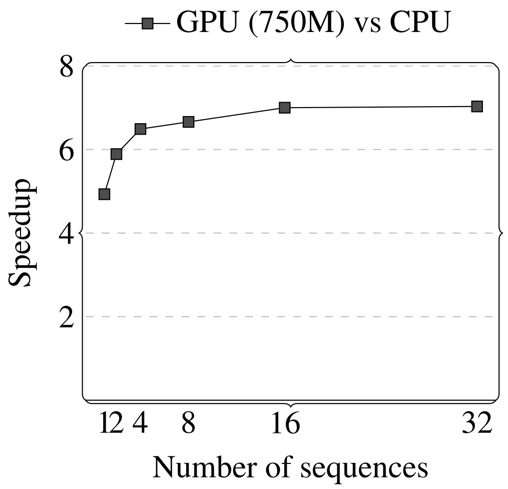
Figure 5.3: End-to-End `alignSequence` latency results for NW (top) and SW (bottom). In NW, sequences are of equal length. In SW, the second sequence length is fixed at 32k.

The NW GPU speedup drops from a peak of 8x to **7x**, while the SW speedup drops from a peak of 12x to **10.5x**. On average, the time of the matrix filling function on the GPU takes up around **68%** of the total runtime of the `alignSequence` program, while the CPU matrix filling function takes up more than 95% of the runtime. These data show that, consistent with Amdahl's law, the maximum possible speedup is limited by the sequential traceback part of the algorithm. This is the motivation for the next benchmark.

Batch processing latency measures the end-to-end latency of the `alignSequence` program when aligning multiple sequences in succession. It simulates a server-client model where requests are submitted to a work queue and worked on by the available CPU and GPU resources. The motivation of this benchmark is to show another benefit of offloading the matrix filling function to the GPU. Namely, the CPU can work on the traceback and the GPU can work on the recurrence ralation at the same time, if multiple sequences need to be aligned in succession; the GPU can immediately start processing the next input, while the CPU is performing the traceback. Thus, the overall latency of the system should decrease. Figure 5.4 demonstrates the validity of this approach. Here, the length of the input is fixed at 8192×8192. The results show that the speedup of using both a CPU and GPU over just using a CPU grows from around 5x for a single sequence up to 7x once more than four sequences are aligned in batch - a **30%** speedup. Recalling that the traceback takes up around 32% of the program runtime on average, this is almost completely offset in this batch processing scenario. This shows the benefit of a CPU and GPU co-execution model for a problem which consists of both parallel and sequential regions.



(a) Latency



(b) Speedup

Figure 5.4: End-to-End latency of aligning multiple sequences in batch using NW. The length of the sequences is fixed at 8192×8192.

5.3 Concurrent kernel evaluation

It is of interest to study the possible speedup of employing more SMs in the filling out of the M matrix. Figure 5.5 shows how the matrix is divided between concurrent kernels.

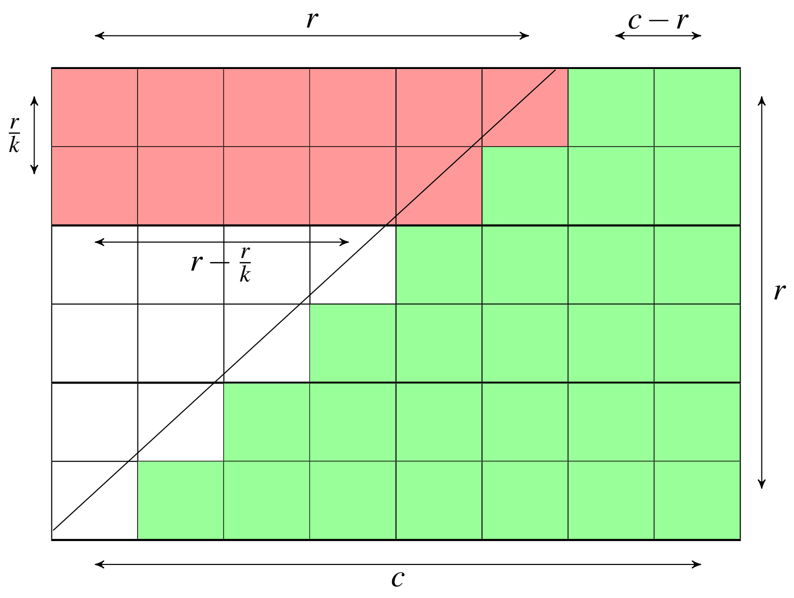


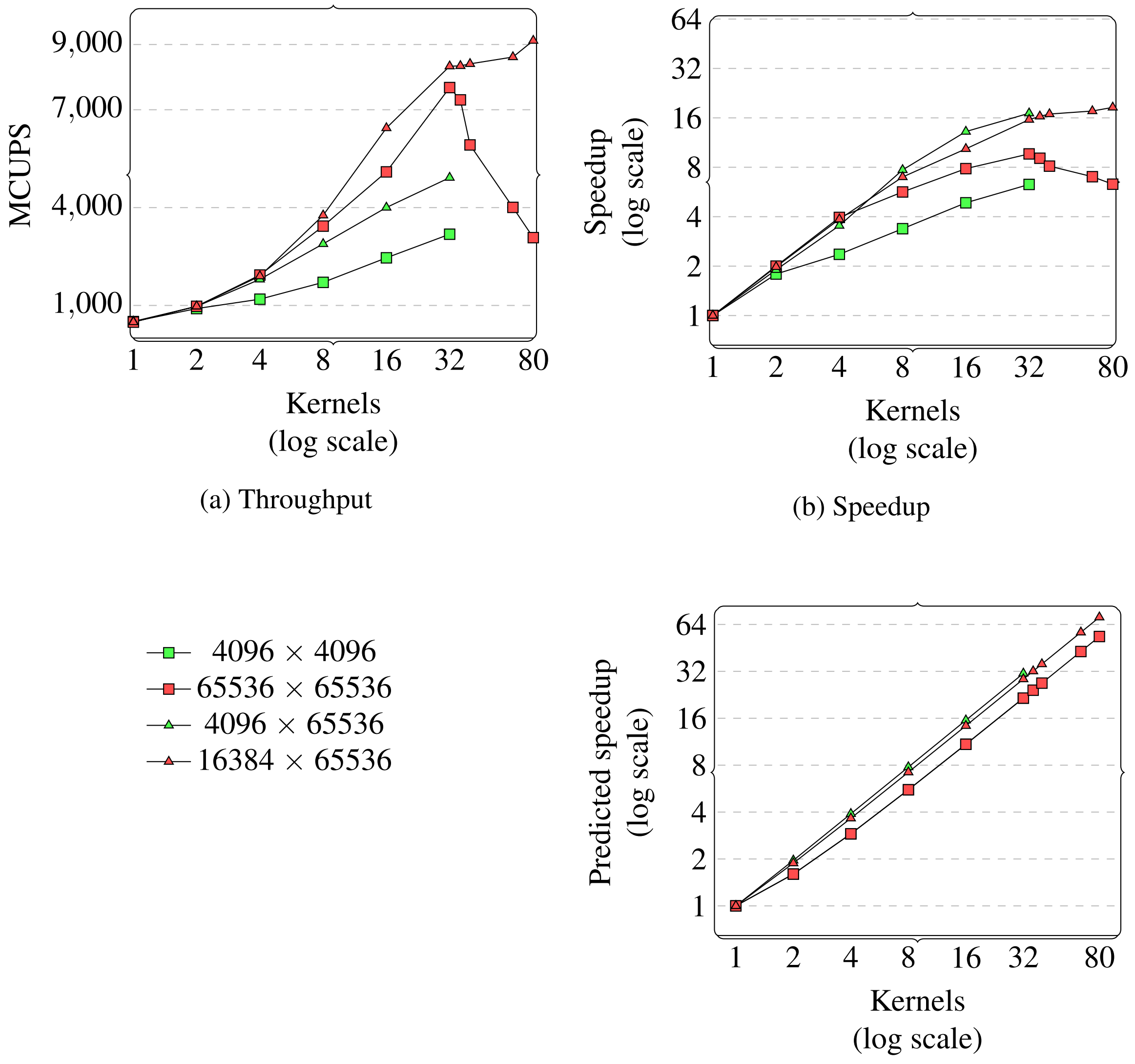
Figure 5.5: Domain decomposition of an $r \times c$ matrix into k concurrently executing kernels. The red trapezoid represents the area of the matrix needed to be filled before all kernels can execute concurrently. The green area can be worked on by all kernels.

The red area represents work needed to be done by the first kernel, before all kernels are able to run concurrently; the green area can be worked on by all kernels. With this decomposition, a simplistic speedup prediction can be made based on the geometry of the M matrix. Let $t_1 = rc$ be the time needed to fill the whole matrix using one kernel, where r, c stand for the number of rows and columns, respectively. Let t_k be the time needed to fill the matrix using k kernels. It can be approximated by adding the area of the red trapezoid to the area of the green trapezoid divided by the number of kernels:

$$t_k = \frac{r}{k} \frac{r + r - \frac{r}{k}}{2} + \frac{r\,\frac{c+c-r}{2}}{k} = \frac{r}{k} \frac{2c - r - \frac{r}{k}}{2}$$

The predicted speedup of using k concurrent kernels is then equal to t_1/t_k. Figure 5.6c shows that the expected speedup of using more kernels is almost linear in the number of kernels, at least for inputs where one sequence is much longer than the other. This prediction is rather simplistic, ignoring any features of the GPU which would affect the actual performance. What it does reveal is that as the c/r ratio increases, the speedup of using more kernels also increases.

A separate benchmark was constructed to measure the actual speedup of using more kernels. A problem size was fixed, and the number of concurrent kernel was varied (1, 2, 4, 8, 16, 32, 40, 64 or 80 kernels). This benchmark was ran on a Tesla V100 card, which has 80 SMs. Figure 5.6 shows the results of this experiment for various sequence lengths. The results show that, as the problem size grows, the benefit of using more kernels increases. Also, where one sequence is much longer than the second, the improvement is higher and more consistent, confirming the insight from the speedup prediction. The error between predicted and actual speedup is within 15% up until using 16 kernels, and increases sharply after that. The most important thing to note from the results is that the performance scaling slows when going from 16 to 32 kernels, and even drops in some cases once going beyond 32 kernels.



(a) Throughput

(b) Speedup

(c) Predicted speedup

Figure 5.6: Throughput scaling with increased number of concurrently executing kernels on an Nvidia Tesla V100 GPU (80 SMs). Throughput is measured in Millions of Cell Updates Per Second (MCUPS).

5.3.1 Discussion

This project has not investigated deeply enough why using more than 16 or 32 concurrent kernels doesn't give the predicted speedup. What follows are pure speculations, not backed up by data. The first suspected culprit is the memory sub-system. Concurrent kernels are synchronised via global memory, and the increased number of memory accesses might cause significant delay. Another bottleneck might be the host-device interconnect. As more kernels run concurrently, the number of data transfers of the filled out portion of the M matrix increases. The PCIe bus bandwidth is far away from being saturated; the problem is the large number of requests to serve.

Being able to efficiently use more than 32 SMs would be the next part of this project, had there be more time. For example, in newer Nvidia cards there is the ability to start CUDA kernels from another CUDA kernel (dubbed "dynamic parallelism" by Nvidia). This might decrease the number of memory requests used to synchronise between kernels. A more sophisticated strategy to transfer data back to the host would also probably help. These questions would be an interesting thread of future work.

Chapter 6

Conclusion

This chapter looks at how well the aims and objectives of this project have been met. The outcomes of of the project are related to the requirements for course accreditation from the Chartered Institute for IT. Finally, a critical self-reflection on the project is performed, together with possible future work.

6.1 Aims & Objectives

In the Introduction, the main goal of this project was specified as "*accelerating the Needleman-Wunsch and Smith-Waterman sequence alignment algorithms on the GPU, with the focus on enabling the alignment of long sequences*". It can be concluded that this goal has been achieved. A breakdown of the aims and objectives needed to achieve this goal and where they were achieved in this project follows.

Aims:

- "*Understanding the problem of sequence alignment in the context of bioinformatics*" was achieved during the research phase of the project. Evidence for that is the Background chapter, where global and local sequence alignment formally defined. Other alignment types and heuristic techniques were also explored during the research phase.

- "*Understanding the GPU architecture, the underlying programming model and gaining practical experience in accelerating a workload using the GPU*"
was also achieved in the initial stage of the project through learning sessions with the project supervisor, and through individual research and exercises. Evidence for that can be found in the discussion of the GPU architecture and programming model in the Background chapter. The learning and practise continued throughout the project. Admittedly, this project only scratched the surface of GPGPU programming but it was a good introduction and building block to expand knowledge in the future.

- "*Implementing the global sequence alignment algorithm on the GPU*"
was achieved as witnessed by the Design and Implementation chapters. The *optional* aim of implementing local alignment was also achieved.

- *Evaluation and understanding of the performance improvement of using a GPU*
  was achieved in the Testing & Evaluation part of this dissertation. The improvement of using a GPU over a CPU was discussed in several scenarios. The predicted and actual speedup of using multiple concurrent GPU kernels was also presented.

- "*Allowing to align as long sequences as possible*"
  was achieved through a modular design discussed in the Design chapter, and an implementation using concurrently executing kernels in the Implementation chapter. The limit on the length of the sequences is imposed by the host memory size, not the much smaller memory size of the GPU. For example, on the 750M system with 16 GB host and 2 GB device memory the maximum input size of the two sequences increases from around 40000 to 120000.

Objectives:

- "*Implementing a global sequence alignment algorithm on the CPU*"
  was one of the first activities of the project. Altough not discussed in detail in this dissertation, the CPU implementation can be found in the `alignSequenceCPU.cpp` file in the code submission. Performance results from the CPU version were given in the Testing & Evaluation chapter.

- "*Setting up a test framework, including test data generation and performance measurement*"
  was achieved in the `tests.cu` file in the code submission. Unit and functional tests expressed in an industry grade testing framework were used. Evidence and results are given in the Testing & Evaluation chapter.

- "*Designing a global sequence alignment algorithm implementation for the GPU, exploiting the unique features of the architecture*"
  was achieved in the Design chapter, with the thinking process and ideas tried discussed. The relevant data structures and operations on them were described, with a mapping to the GPU architecture.

- "*Implementing the designed algorithm on the GPU*"
  was achieved in the Implementation chapter. The *optional* goal of implementing a local alignment algorithm was also achieved.

- "*Running experiments comparing the GPU and CPU*"
  was achieved in the `benchmarks.cu` file in the code submission. Evidence and results were given in the Testing & Evaluation chapter, testing the implementation in multiple scenarios.

- "*Using concurrently executing kernels to employ all GPU resources in aligning longer sequences*"
  was achieved, with the evidence in the Design and Implementation chapters.

- "*Evaluation of the performance improvement and scalability of using concurrent GPU kernels*" was achieved in the last section of the Testing & Evaluation chapter. The conclusions behind the evaluation results from this step were interesting but incomplete. This objective was added during the course of the project, and it could be expanded on as part of future work.

6.2 BCS Criteria

The British Computer Society (BCS) Chartered Institute for IT expects several outcomes from a final year dissertation in order for the degree to be accredited by the organisation. These were met in the course of this project. How and where is listed below

- "*An ability to apply practical and analytical skills gained during the degree programme*" was demonstrated in the design, implementation and evaluation stage of this project. Practical computer programming skills were needed to finish the implementation stage; analytical skills were needed in the design of the implementation and in the evaluation.
- "*Innovation and/or creativity*"
  were demonstrated by using concurrent GPU kernels. Creativity was shown in designing this approach. Calling the use of concurrently executing kernels "innovative" would be an overstatement but it was an unconventional way of approaching the problem and the synchronising code between kernels was objectively original.
- "*Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution*"
  was demonstrated through the learning and application of GPGPU programming. This was a novel concept and had first to be learned. Information from various sources was synthesised and adapted to meet the goal of this project. The solution was of sufficient quality and was evaluated objectively.
- "*That the project meets a real need in a wider context*"
  can be argued by the prevalence of sequence alignment tools in bioinformatics. This was discussed in the Introduction chapter, which goes into more detail behind the motivation for accelerating sequence alignment on the GPU.
- "*An ability to self-manage a significant piece of work*"
  was demonstrated by adhering to the project proposal throughout the months of work on the project. Additional aims and objectives were added during the project, which had to be seen through to completion in order to not interfere with the original workload. This was a significant piece of work, which had to be well managed, especially with the added workload of other University modules.
- "*Critical self-evaluation of the process*"
  has been performed in this dissertation (see next section).

6.3 Self-Reflection & Future Work

In my opinion, this project has been a success. My main goal was to learn about the GPU architecture and gain practical experience in accelerating a non-trivial piece of code. This has mostly been achieved. The learning of GPGPU programming started with tutorial sessions with my supervisor and has continued throughout the project where I was able to apply the acquired knowledge in practice.

The aims and objectives detailed in the project proposal were followed through, with additional objectives added during the project. Managing these has improved my ability to self-manage and plan ahead. Being able to break up complex and long-term tasks into manageable pieces is a useful skill to have in life, and this project has taught me a thing or two about that.

As is expected in a project spanning several months, not everything went to plan. The predicted time spent on individual objectives from the project proposal timeline were sometimes quite different from reality. Especially the testing and evaluation stage took longer than I had expected. The performance tests spotlighted several places where the implementation could be improved, which increased the time needed to complete this objective. It has taught me to integrate at least some performance tests with the implementation stage in the future. Another unexpected lesson was the gulf between a working prototype and a complete piece of work, such as this dissertation. The work needed to gather results, analyse them, draw conclusions and write them up was sometimes daunting, but rewarding in the end.

There are several pieces of work in this project that could be improved or expanded on. Firstly, there is no need to use a full byte to store directions in the M matrix, which can only take four possible values. Directions from four cells could be packed into a single byte, decreasing the space requirement by 4x for a small price of a compression and decompression computation. The traceback part of the algorithm was also not particularly well optimised, and it consumes up to 30% of the program runtime for some input sizes. Improving this step would be a focus of future work. Other improvements would include adding more features that the users of the software would be interested in. The ability to use an affine gap penalty is common in alignment tools, penalising the opening of a gap and extending it differently. This wouldn't be hard to add to the existing implementation.

The most interesting thing for me to work on further would be improving the concurrent kernel performance scaling beyond 16 or 32 kernels. First, I would try to pinpoint what exactly is causing the performance scaling degradation at that point. This information would then drive the design of a workaround, or even a change in the core design of the implementation.

In summary, the project was a great opportunity to apply the knowledge learned during my whole degree. It has taught me several lessons which could only be learned when managing a larger piece of work. I am left satisfied with the results and encouraged for further study in the area of computer architecture and software performance engineering.

Bibliography

[1] Jade Alglave et al. “GPU Concurrency: Weak Behaviours and Programming Assumptions”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 577–591. ISBN: 9781450328357. DOI: `10.1145/2694344.2694391`. URL: `https://doi.org/10.1145/2694344.2694391`.

[2] Christiam Camacho et al. “BLAST+: architecture and applications”. In: *BMC bioinformatics* 10.1 (2009), pp. 1–9.

[3] Miguel Carneiro et al. “A loss-of-function mutation in RORB disrupts saltatorial locomotion in rabbits”. eng. In: *PLoS genetics* 17.3 (Mar. 2021). PGENETICS-D-20-01498[PII], e1009429–e1009429. ISSN: 1553-7404. DOI: `10.1371/journal.pgen.1009429`. URL: `https://doi.org/10.1371/journal.pgen.1009429`.

[4] Thomas Carroll. *Graphics Processing Units: Abstract Modelling and Applications in Bioinformatics*. University of Liverpool, 2020. URL: `https://books.google.co.uk/books?id=KqLmzQEACAAJ`.

[5] Thomas Carroll, Jude-Thaddeus Ojiaku, and Prudence W. H. Wong. “Semiglobal Sequence Alignment with Gaps using GPU”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* PP (Apr. 2019), pp. 1–1. DOI: `10.1109/TCBB.2019.2914105`.

[6] Catchorg. *catchorg/Catch2*. URL: `https://github.com/catchorg/Catch2/tree/v2.x`.

[7] *CUDA C++ Programming Guide*. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[8] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007.

[9] *GenBank Overview*. URL: `https://www.ncbi.nlm.nih.gov/genbank/`.

[10] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. “CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions”. In: *BMC bioinformatics* 14 (Apr. 2013), p. 117. DOI: `10.1186/1471-2105-14-117`.

[11] Fábio Madeira et al. “The EMBL-EBI search and sequence analysis tools APIs in 2019”. In: *Nucleic acids research* 47.W1 (July 2019), W636–W641. ISSN: 0305-1048. DOI: `10.1093/nar/gkz268`. URL: `https://europepmc.org/articles/PMC6602479`.

[12] D.W. Mount and Cold Spring Harbor Laboratory. Press. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Series. Cold Spring Harbor Laboratory Press, 2004. ISBN: 9780879697129.

[13] Saul B. Needleman and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". English (US). In: *Journal of Molecular Biology* 48.3 (Mar. 1970), pp. 443–453. ISSN: 0022-2836. DOI: 10.1016/0022-2836(70)90057-4.

[14] William R Pearson and David J Lipman. "Improved tools for biological sequence comparison". In: *Proceedings of the National Academy of Sciences* 85.8 (1988), pp. 2444–2448.

[15] Ranjit Sah et al. "Complete Genome Sequence of a 2019 Novel Coronavirus (SARS-CoV-2) Strain Isolated in Nepal". In: *Microbiology Resource Announcements* 9.11 (2020). Ed. by Simon Roux. DOI: 10.1128/MRA.00169-20. eprint: https://mra.asm.org/content/9/11/e00169-20.full.pdf. URL: https://mra.asm.org/content/9/11/e00169-20.

[16] D. B. Skillicorn. "A taxonomy for computer architectures". In: *Computer* 21.11 (1988), pp. 46–57. DOI: 10.1109/2.86786.

[17] T.F. Smith and M.S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. URL: http://www.sciencedirect.com/science/article/pii/0022283681900875.

[18] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. "GPU-BLAST: using graphics processors to accelerate protein sequence alignment". In: *Bioinformatics* 27.2 (Nov. 2010), pp. 182–188. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btq644. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/2/182/6688171/btq644.pdf. URL: https://doi.org/10.1093/bioinformatics/btq644.

[19] S. Xiao and W. Feng. "Inter-block GPU communication via fast barrier synchronization". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470477.

Appendix A

Experimental setup

This project used three types of machine for performance testing: (1) a mobile laptop, (2) a visualisation node on the University of Liverpool compute cluster "Barkla", and (3) a compute node on Barkla. The details of these machines are given below.

	Nvidia GT 750M @ 0.93 GHz	**Nvidia Quadro P4000** @ 1.79 GHz	**Nvidia Tesla V100** @ 1.31 GHz
Architecture	Kepler	Pascal	Volta
GPU memory	2 GB	8 GB	16 GB
SMs	2	14	80
CUDA cores per SM	192	128	64
CUDA cores total	384	1792	5120
CPU	Intel i7-4850HQ @ 2.30 GHz	Intel Xeon Gold 6138 @ 2.00 GHz	Intel Xeon Gold 5118 @ 2.50 GHz
Host memory	16 GB	394 GB	394 GB
OS	MacOS 10.13	CentOS 7.9	CentOS 7.9

Appendix B

sequence-alignment-gpu readme

Run on Barkla GPU visualisation nodes (no prerequisites required)

```
1. Get code base:
       tar -xf ˜sgrszafa/sequence-alignment-gpu.tgz
2. Load Cuda and a C compiler:
       module load libs/nvidia-cuda/10.1.168/bin
       module load compilers/gcc/8.3.0
3. Run:
       // Will run on Nvidia Quadro P4000
       ./test
       ./benchmark
       ./alignSequence

       // Will run on Nvidia V100 (or P100) using a queuing system.
       sbatch barkla_runBenchmark.sh
       sbatch barkla_runTest.sh
       sbatch barkla_alignSequence.sh
```

Run locally

```
1. Check prerequisites.
2. Unzip code.
3. Compile with:
       make -j
4. Run:
     // Main program
     ./alignSequence

     // test suite
     ./test

     // performance tests
     ./benchmark
```

Prerequisites

```
    - Nvidia GPU with compute capability >3.0 (Kepler, and later)
    - CUDA (tested with 10.1 and 11.1)
    - A C++14 compliant host compiler (tested with clang and gcc)
    - Make
```