Accelerating Protein Sequence Alignment with Different Parallel Hardware Platforms

Master thesis

TU-Delft University of Technology

EEMCS

Computer Science

N.R.N. Timal

February 27, 2015

Abstract

Detecting similarities between (RNA, DNA, and protein) sequences is an important part of bioinformatics. Among the algorithms used to accomplish this, the Smith-Waterman algorithm is very popular. A sequential implementation of Smith-Waterman requires quadratic running time with respect to the length of the sequences. As the amount of data in this field is continuously increasing, quick analysis through a sequential implementation is no longer feasible. One way to reduce the running time is by using parallelism and parallel platforms. There is a great diversity of hardware platforms that enable parallelism in different ways, each favoring different types of computations. The subject of this thesis is to understand the performance of the state-of-the-art parallel implementation of the Smith-Waterman algorithm, PaSWAS, on different parallel hardware platforms. PaSWAS has been designed and implemented using CUDA, the proprietary framework from NVIDIA. This choice limits the PaSWAS functionality to NVIDIA GPUs. By using OpenCL, a platform independent, standard programming model for many-cores, we enable PaSWAS to run in parallel on other hardware platforms. We show that, for NVIDIA GPUs, the portability enabled by OpenCL comes at the expense of performance. We further define a set of platform-specific parameters that have a high performance impact for the OpenCL implementation, and demonstrate empirically that their values are different for different hardware platforms. We also demonstrate that proper partitioning of the sequences can increase parallelism, which leads to better performance. Lastly, we create a performance estimator which is able to predict the execution time of the PaSWAS algorithm on different hardware platforms for given dataset. This enables us to determine a-priori which hardware platform to use for a given dataset. We conclude that PaSWAS is a very effective parallel implementation of the Smith-Waterman algorithm, which delivers excellent results for GPUs (in both CUDA and OpenCL), and can be quite effective on CPUs, too. The performance vs. portability tradeoff of OpenCL is relevant for PaSWAS, and it is ultimately the choice of the end user which of the two is more relevant.

Keywords: OpenCL, CUDA, Sequence alignment, PaSWAS, Bioinformatics, Parallelism, Smith-Waterman, Performance estimator

Contents

CHAPTER 1

Introduction

Detecting similarities between (RNA, DNA and protein) sequences is an important part of bioinformatics. By comparing/aligning these sequences with each other, one can determine, for example, whether given sequences are constructed from the same biological building blocks and/or gain a deeper understanding of the evolutionary aspects within the field of molecular biology.

1.1 Motivation and Problem Statement

When aligning two sequences, we are looking for the similarity between these sequences. Sequence alignment can be divided into two major categories: *local* and *global* sequence alignment. With local sequence alignment, we are trying to find the most similar region within two sequences, whereas with global sequence alignment we are looking for the overall similarity of two sequences. In this thesis we focus on the Smith-Waterman algorithm [Smi+81], which is used to solve the local sequence alignment problem.

The run-time complexity of the Smith-Waterman algorithm is given by $\mathcal{O}(|s| \times |t|)$, when aligning two sequences s and t (here, $|x|$ signifies the length of a certain sequence x). This problem can be regarded as a polynomial optimization problem, which resides in the complexity class PO [Aus99]. Optimization problems that reside in this class tend to be characterized as efficient and tractable. However, when the length of the sequences increases (i.e., $|s|$ and $|t|$ are very large), a quadratic growth in the execution time is expected. As the databases of sequences are continuously growing[CS 14], the time needed to align the sequences in these databases also increases considerably. Thus, it is important for biologists to accelerate the execution time of the Smith-Waterman algorithm.

One way to achieve this acceleration is by parallelizing the algorithm, and using multiple computing units (that collaborate to solve the task at hand) instead of one single processor. Ideally, this leads to a reduction of execution time proportional to the number of processors used, but the exact gain is entirely dependent on the application. For example, utilizing two processors will not necessarily reduce the execution time by 50%. Such behaviour is typically due to processors not being able to perform the computations completely independent of each other.

To enable parallelism at the platform level, one can use multi-core CPUs or clusters of multi-core CPUs. These solutions focus on coarse-grained parallelism: each node and/or core is responsible for processing a large chunk of data. Another approach is to harness the power of a *graphics processing unit* (GPU) as an accelerator. A GPU is a hardware device which enables fine grained parallelism: each core only evaluates a very small piece of the data. However, the number of available cores is much larger for a GPU than for a CPU (because GPUs are designed for graphical processing, which is characterized by massively parallel but relatively simple computations), and it is often the case that the theoretical peak performance of a GPU is much higher than that of a CPU.

In short, both these families of hardware platforms enable parallelism, but which one of them is the most suitable for the Smith-Waterman algorithm is not yet known. This thesis focuses on giving an empirical solution to this challenge.

In order to exploit the diversity of available parallel hardware platforms, thus proposing the most efficient solutions for the Smith-Waterman algorithm, we must understand how these platforms behave in the context of this application. Therefore, the main objective that drives our research is **to understand the performance of the Smith-Waterman algorithm on different types of hardware platforms.**

1.2 Approach

This thesis focuses on the performance of the Smith-Waterman algorithm. Specifically, we use a state-of-the-art implementation called PaSWAS [War+15]. PaSWAS is implemented to use CUDA, a proprietary framework provided by NVIDIA [Nic+08] for its own graphical processing units (GPUs). Thus, in its original state, PaSWAS can only use NVIDIA GPUs to achieve performance improvement over the sequential version of the algorithm.

To also enable a parallel execution on multi-core CPUs, we must choose a programming model that allows portability among multiple devices - like OpenCL or OpenACC. For example, OpenCL [Sto+10] offers *functional portability* across several platforms (including NVIDIA GPUs), with minimal code changes. Its generality does not necessarily decrease the performance on a specific platform [Du+12]. However, it is not known whether OpenCL brings additional performance penalties compared to CUDA. Therefore, our first research question is:

- (R1): Can we achieve similar performance on a NVIDIA GPU when porting the CUDA code of PaSWAS to OpenCL?

By answering this question, we can determine whether OpenCL is a viable alternative to the proprietary framework of CUDA for our particular application. Our empirical evaluation will determine whether the use of OpenCL for the Smith-Waterman algorithm is feasible and efficient, without significant performance losses compared with the version that uses CUDA.

We further extend our horizon, aiming to determine what are the parameters which determine a significant performance increase or decrease for our OpenCL implementation. We focus on the parameters which are both application-dependent and hardware-specific. Thus, we aim to answer the following research question:

- (R2): Which are the platform-specific parameters that have a high performance impact?

To answer this question, we create several PaSWAS implementations, each enabling a specific set of parameters. We empirically assess these versions and determine which set of parameters leads to the best performance, depending on the platform.

Within PaSWAS, the size of the sequences and targets can be very different. To allow for uniform processing, sequences in a database are padded to the length of the largest sequence. In the same time, there is strong correlation between the length of a sequence and the amount of parallelism that processing that sequence exhibits. In general, longer sequences tend to have less parallelism compared to shorter ones. Thus, treating all sequences as the largest sequence might incur significant performance penalties. To circumvent this issue, we must investigate smarted partitionings of the datasets into manageable lengths. Our third reseacrh question is:

- (R3): How can we partition a dataset to enable the most parallelism?

To answer R3, we designed and implemented three partitioning algorithms. We empirically determine which one leads to the highest peformance improvement (compared to the current scheme used in PaSWAS) and recommend it for further usage in PaSWAS - for CUDA and OpenCL, CPUs and GPUs alike.

The calculations done within PaSWAS can take a significant amount of time. Furthermore, different devices and their capabilities can result in significantly large performance differences in achieved performance. For both these reasons, we need a performance estimator able to predict the execution time of PaSWAS with a new dataset on different devices. Therefore, we investigate the possibility of building such a predictor:

- (R4): Can we provide a performance predictor that can estimate the performance of PaSWAS on a given platform, with a given dataset?

To answer this question, we build a model-based predictor by exploiting the inherent execution structure of the PaSWAS algorithm together with data characteristics (which are known before hand) and the hardware platform parameters. We further verify the accuracy of the prediction empirically.

1.3 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 provides a brief introduction into bioinformatics, and provides a high-level comparison between OpenCL and CUDA. To demonstrate the usage of OpenCL in practice, we also discuss an implementation of matrix multiplication. In Chapter 3, we summarize prior research on the topics related to our study. We briefly assess representative work and discuss how it influences and how it differs from our work. In chapter 4, we present a sequential counterpart of PaSWAS. We discuss PaSWAS in greater detail, in chapter 5. One of the important performance challenges in PaSWAS is related to the data transfers between devices. We analyze the benchmarking results for these transfers in chapter 6, providing a clear overview on which choices should be made in the real application. 7. We focus on the performance of PaSWAS on different platforms, using different features. In chapter 8, we describe the three partitioning algorithms we have developed in order to achieve more parallelism. Our experimental results are presented in detail in chapter In chapter 9 we describe our performance prediction model, its validation and its usability. We will conclude this thesis in chapter 10, which presents our conclusion - together with the main contributions and limitations we found in this study - and sketches promising directions for future research.

CHAPTER 2

Background

2.1 What is bioinformatics?

In this section, we briefly discuss the field of bioinformatics and how it intersects with computer science and biology. The field of bioinformatics originated from the necessity to analyse the ever growing biological data. Manual (that is, by human labour) analysis of the aforementioned data became impractical, therefore tools had to be created to automate the process of dissecting the biological data. These tools are (partially) grounded in the field of computer science.

Biological data includes DNA (DeoxyriboNucleic Acid) , RNA (RiboNucleic Acid) and protein sequences. DNA sequences are located within every cell of a living organism, these DNA sequences store biological information. This information can range from the colour of your eyes, the scent of a rose and the way in which a bacteria infects a lung cell. More specifically, DNA encodes the instructions for building protein molecules (proteins play a vital role in the nourishment of cells) [Cla08].

RNA molecules interpret the instructions encoded in the DNA and act accordingly by creating proteins within this cell [JE +90]. The aforementioned process of transitioning from DNA to proteins was introduced in [Cri70]. This article has adopted the name of the central dogma of molecular biology. Figure 2.1(a) provides a schematic overview of this process, commonly known as general information transfer.

The most important elements of the aformentioned dogma are given by replication, transcription and translation. Before examining these elements more closely, we will provide a quick overview of the structure of DNA and RNA sequences.

Both RNA and DNA sequences belong to the classes of nucleic acids and are composed of repeated units of nucleotides. A single nucleotide consists of nitrogenous base which can either be Cytosine (C), Guanine (G) or Adenine (A) (A and G are called purines). Nucleotides in DNA sequences introduce an additional base, which does not occur in RNA sequences, namely Thymine (T). However, RNA sequences also have an additional base: Uracil (U) which is a replacement for Thymine (U, T and C are regarded as pyrimidines). Besides of having a nitrogenous base, a nucleotide is composed of a (carbon

$$\begin{tikzcd}
DNA \arrow[loop above, "Replication"] \arrow[r, "Transcription"] & RNA \arrow[r, "Translation"] & Proteins
\end{tikzcd}$$

(a) General information transfer

$$\begin{tikzcd}
RNA \arrow[loop above, "Replication"] \arrow[r, "Reverse\ Transcription"] & DNA \arrow[r, "Translation"] & Proteins
\end{tikzcd}$$

(b) Special information transfer

$$\begin{tikzcd}
RNA & Proteins \arrow[l] \arrow[r] \arrow[loop above] & DNA
\end{tikzcd}$$

(c) Unknown information transfer

Figure 2.1: Information transfer between DNA, RNA, and proteins

based) sugar molecule and a phosphorus containing region (also known as phosphate group). A nucleotide from a RNA sequence contains the sugar molecule ribose, whereas in DNA sequences the sugar molecule is deoxyribose.

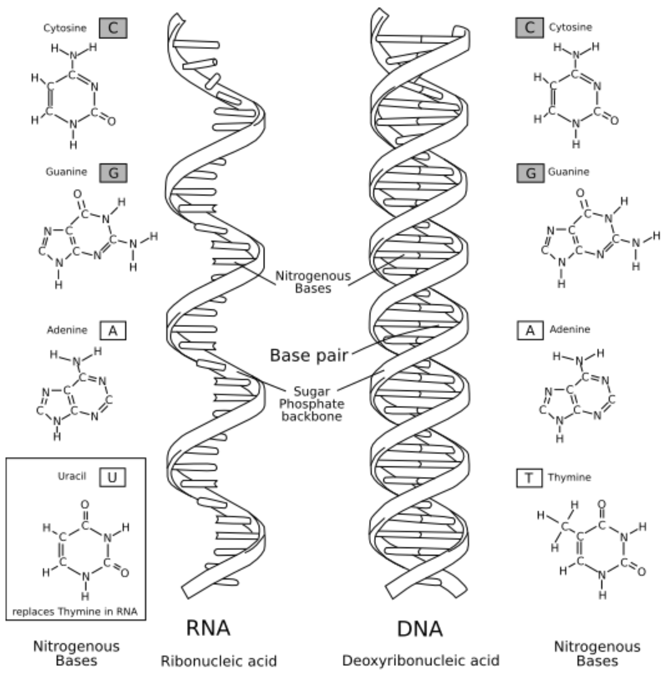


Figure 2.2: Structure of both DNA and RNA. In addition the molecular structure of the nitrogenous bases are displayed.

Figure 2.2 represents the structures of DNA and RNA sequences, in addition the molecular layout of the 5 nitrogenous bases are displayed. We can observe from figure

2.2 that a RNA sequence contains one strand of nucleotides. Whereas DNA sequences have two strands of nucleotides, which are connected by their bases. Only a subset of nitrogenous bases can be paired, these are dictated by the Watson and Crick base pairs. The bases that can be paired are: Adenine with Thymine (or Uracil) and Guanine with Cytosine. Both RNA and DNA strands have a sense of directionality, the different ends of these strands are called 5' and 3'. With DNA sequences the two strands are connected by their different ends, that is the 5' end of one strand is connected with 3' end of the other strand.

Cells reproduce continuously (for example to replace worn-out cells), DNA replication provides a cornerstone for this reproduction. As the name of this process already implies, DNA replication tries to create an two exact copies of a cell's DNA. The group of enzymes that perform these replications are DNA polymerases. Both of the (existing) DNA strands act as a template for the newly replicated strands. The enzymes cannot create a new strand, however they can extend existing ones. To this end a small RNA sequence, a primer, is used to initiate the replication. Figure 2.3 illustrates this process for a single strand of DNA. In the example the sequence TGGAC acts as a primer. Additionally the figure illustrates that the nucleoside (the same as a nucleotide but without the phosphate group) are extended from 3' end side.

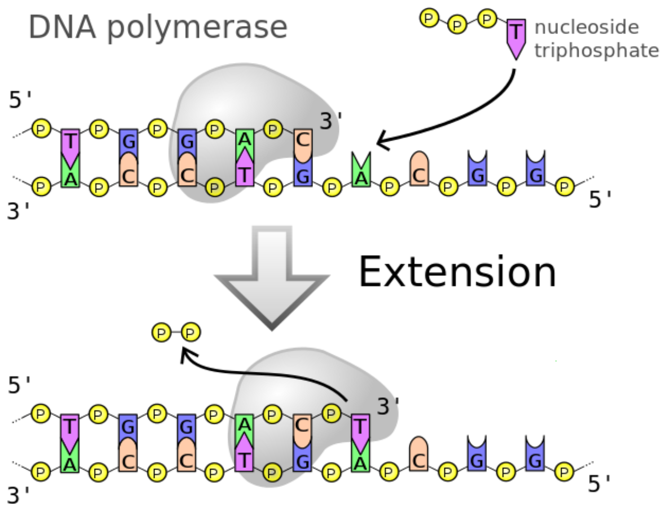


Figure 2.3: The process of a single strand DNA replication.

Transcription provides the conversion between a DNA sequence to a RNA sequence, the resulting RNA sequence is called mRNA (messenger RNA). mRNA should contain the information creating proteins. The process for constructing the mRNA sequence is similar to that of DNA replication however instead of using both strands only a single strand is used (the one in which a specific protein is encoded). The process of transcription is initiated by an enzyme called RNA polymerase. We will distinguish between two strands located in the DNA, template and coding strand. The template strand is used to construct the mRNA, employing the rules of the Watson-Crick base pairs. Note that the mRNA sequence should be similar to the coding strand except for the replacement of Thymine with Uracil.

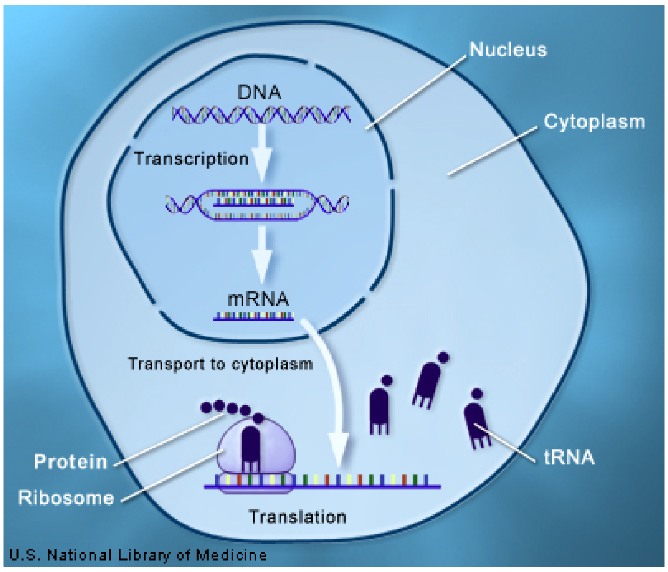


Figure 2.4: Depiction of a cell and the corresponding steps to create a protein.

From figure 2.4 we can see that transcription occurs inside the cell's nucleus. The last phase, translation, transpires outside of the nucleus more specifically in the cytoplasm. With translation the actual protein is constructed. The transcribed mRNA (outside of the nucleus) interacts with a specialized complex called ribosome. The ribosome reads the mRNA. Each sequence of three bases, called a codon, usually codes for one particular amino acid, (amino acids are the building blocks of proteins). A type of RNA called transfer RNA (tRNA) assembles the protein, one amino acid at a time. Protein assembly continues until the ribosome encounters a stop codon (a sequence of three bases that does not code for an amino acid).

Two other types of information transfers are also described in the aforementioned dogma. These are special and unknown transfers which only occur by human intervention or don't occur at all.

2.2 Sequence alignment

A commonly used algorithm within the field of bioinformatics is sequence alignment [Kle+06]. With (global) sequence alignment, one wants to measure how similar two different sequences are to each other. The aforementioned algorithm can be classified as a dynamic programming algorithm. Characteristic of a dynamic programming algorithm, constructed for a certain problem, is that the problem at hand can be solved by solving its subproblems. If we have two sequences $s = s_1 \dots s_n$ and $t = t_1 \dots t_m$ we can establish the following recurrence relationship for the sequence alignment problem:

$$A(i,j) = \max \begin{cases} A(i-1,j-1) + W(s_i,t_j) \text{ (1)}. \\ A(i,j-1) + \textit{gap penalty in } s \text{ (2)}. \\ A(i-1,j) + \textit{gap penalty in } t \text{ (3)}. \\ 0 \end{cases} \tag{2.1}$$

In 2.1 $A(i,j)$ indicates the alignment score between s_i and t_j. Thus, within the sequence alignment problem we can (1) either leave the sequences as are, by aligning s_i and t_j, using the scorings matrix W, or (2) and (3) introduce a gap in either sequences (case (2) and case (3)).

2.3 Sequence alignment within the context of bioinformatics

Within the context of bioinformatics, local sequence alignments are often preferred (e.g. Smith-Waterman algorithm) to compare strands of DNA (or strands of RNA for that matter) instead of global sequence alignments (e.g. Needleman-Wunsch Algorithm). With local sequence alignments, we are interested in subsets $x \subseteq s$ and $y \subseteq t$ such that the alignment between x and y is maximal (or at least it has a sufficient alignment score). The input to the alignment algorithm is defined as: s_i and $t_j \in \{A,C,G,T\}$ or $\{A,C,G,U\}$. Thus the sequences s and t are comprised out of repeated units of nitrogenous bases. Local/global sequence alignments for this kind of data is useful since we can infer things like common ancestry (homology) when comparing the DNA of two different organisms.

2.4 Sequence alignment outside the context of bioinformatics

A rather obvious application, outside the context of bioinformatics, which employs the sequence alignment algorithm can be a proofing tool integrated into a word processor. In the aforementioned case the sequences s and t are words from the natural language. The sequence s will be a misspelled word typed by a user whereas t is the suggested word given by the proofing tool. In addition, the sequence alignment algorithm can be used to detect plagiarism [Irv04] i.e. the algorithm can determine if two submissions are very much alike.

A not so apparent application for sequence alignment is to detect code clones (as is done in the following implementation: `https://github.com/jruoff/CloneGrid`). In this case s and t represent the source code from a certain programming language (i.e. the contents of hello.c or hello.cpp) and the elements s_i and t_j represent a single line of code. If there is a high alignment score this can indicate that the coder is adopting a copy/paste behaviour of writing code.

2.5 Processing Units: GPUs and CPUs

As data tends to increase[CS 14] one needs to find ways to efficiently (i.e. within an acceptable time frame) process the data.

One way to gain a performance boost is to process data (if applicable) in parallel instead of sequentially. Using CPU based libraries like OpenMP[Ope14] the aforementioned can be achieved relatively easily (i.e. by placing a pragma above a for-loop, to indicate that the loop has to be processed in parallel).

The number of processing units within (commercially available) CPUs are quite limited, thus when (big) data is partitioned among the processing units, each processing unit still has to process a significant amount of data.

Another paradigm has arisen to enable parallelism, namely the use of GPGPUs (General-purpose computing on graphics processing units). That is, using GPUs for other tasks then graphical processing. GPGPUs are used within numerous scientific fields. These fields include, but are not limited to, astronomy, biology and computational finance. GPUs are inherently designed to compute in a massively parallel environment, since these types of computations are needed for graphics rendering. Due to the large number of processing units that are available within a GPU, the data that has to be processed by one processing unit is reduced dramatically.

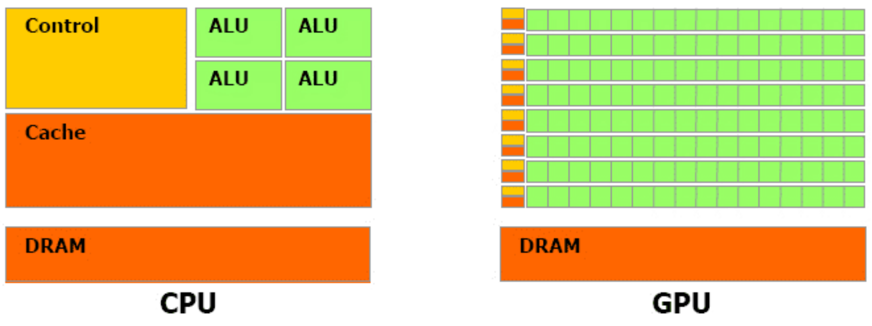


Figure 2.5: Architectural difference between a CPU and a GPU (image courtesy of NVIDIA).

Figure 2.5 provides a schematic overview of both a multi-core CPU and a GPU (more specifically a GPU with one streaming multiprocessor). As can be seen from figure 2.5, a GPU consists of many processing units (a processing unit is characterized by a single control and cache unit combined with one or more ALU circuits, to allow for threaded computation). This is in contrast of what is depicted for the CPU, that is a CPU consists of a single (bigger) cache and (a more complex) control unit. In addition a CPU has less ALU circuits. GPGPU solutions will flourish when an algorithm has a high arithmetic intensity (computations are dominant factor to influence the performance of the algorithm instead of memory accesses).

2.6 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing architecture for C/C++ and is created by NVIDIA. With CUDA a GPU (Graphical Processing Unit), manufactured by NVIDIA, can be used as a GPGPU.

A CUDA application consists of two separate code bases namely the host and the kernel. Commonly the host code will run on a CPU whereas the kernel is executed on a device. These devices are restricted to GPUs manufactured by NVIDIA. The purpose of the host is to facilitate memory management. That is, allocating memory blocks on the device and freeing these blocks when they are not needed anymore. In addition the host can read from and write to the allocated memory blocks.

Figure 2.6 gives a more elaborate view on how data is processed on a GPGPU platform. First data is copied from the main memory (which is accessible from the host) to the memory of the GPU. Typically, main memory is larger than GPU memory. When the data has been transferred from the host memory (main memory) to the device memory, the device is able to process this input data and generate output data (on the device memory). Eventually the output data is transferred back to the main memory by the host.

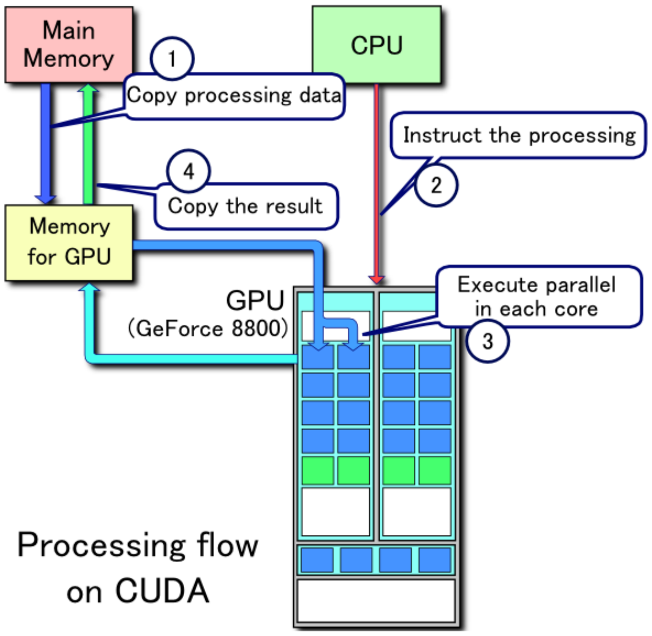


Figure 2.6: Data processing on a GPGPU platform (image courtesy of NVIDIA).

The actual algorithm (or a part of the algorithm that can be efficiently parallelized) is written in the kernel. The kernel gets executed on the device through a call within the host code (indicated by step 2 and 3 in figure 2.6).

CUDA adheres to the so called SIMT (Single Instruction Multiple Threads) execution model, meaning that each thread (spawned from a processing unit within a streaming multiprocessor) will execute the same instructions. These instructions are dictated by the kernel.

As is displayed by figure 2.7, a kernel gets executed within a grid. A grid is a collection of blocks. These blocks are then further subdivided into threads. Both the grid and block sizes are configurable by the programmer. The streaming multiprocessor can execute multiple blocks in parallel [1]. Thus when more blocks are scheduled on the multiprocessor the level of parallelism increases and subsequently the performance of the algorithm should increase. The maximum number of blocks that can be executed concurrently is limited by the GPU's hardware.

Figure 2.8 provides an overview of the memory hierarchy within CUDA. There are 4 memory types each ranging in size and access time, these are (sorted from largest in size and highest access time to smallest in size and lowest access time): global memory, constant memory, shared memory and registers.

Transferring data between the host and device can only occur in both the global and constant memory. Constant memory has the property that it does not allow to be written to by the threads participating in the kernel execution (data remains constant, is not variable). Whereas global memory allows for both reading and writing by the threads. In addition constant memory tends to be cached and thus providing faster access than global memory. Both global and constant memory are shared among every block in a kernel execution grid.

Proper use of the memory types could have a performance enhancing effect. For example one wants to minimize the accesses to global memory, in the kernel, since retrieving (and writing) data from (to) global memory is expensive. To circumvent the problem of continuously accessing global memory, shared memory is used. Shared memory is shared among the threads in the same thread block, access to this type of memory is significantly faster than global memory. Thus if a certain global memory access pattern has been established within a thread block, e.g. certain data elements are accessed more frequently, the overall performance will be positively influenced if these data elements are put in the shared memory and accessed from the shared memory. Registers are thread specific storage units.

1 internally blocks are processed in warps. These warps are collection of 32 threads that are launched and (usually) executed together, this is actually what the SIMT principle encompasses. The warp size is not configurable by the programmer

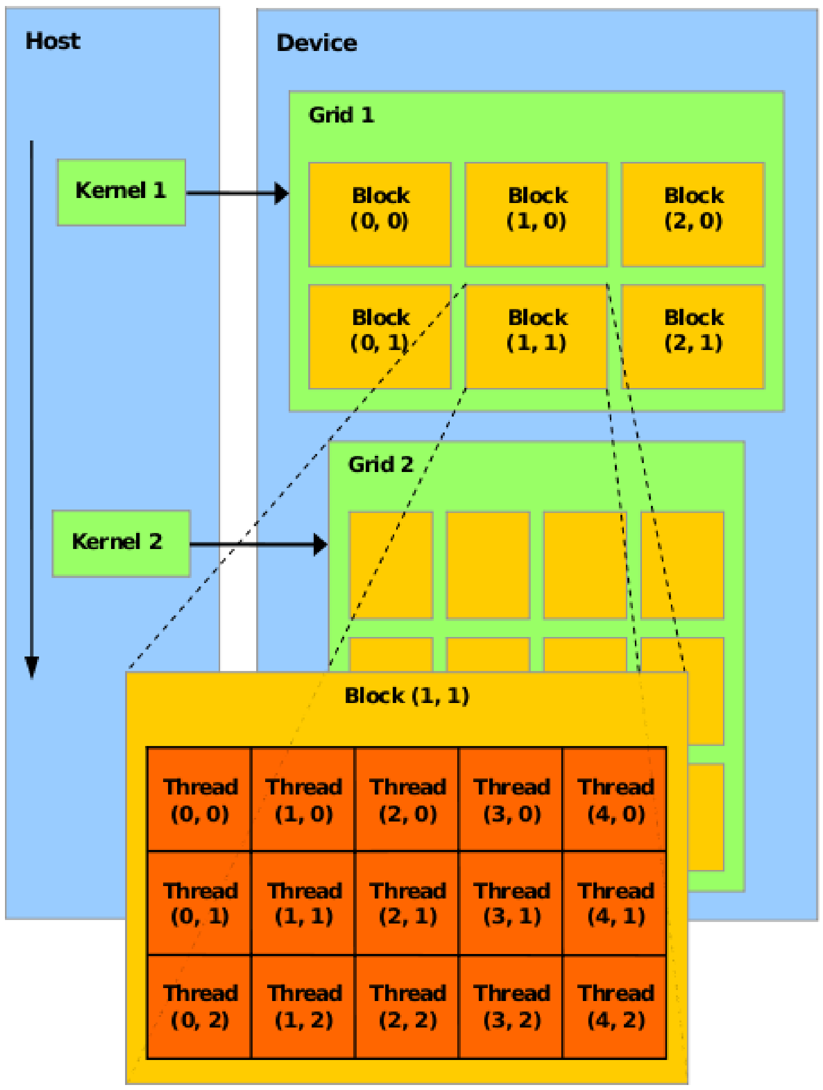


Figure 2.7: CUDA's kernel execution model (image courtesy of NVIDIA).

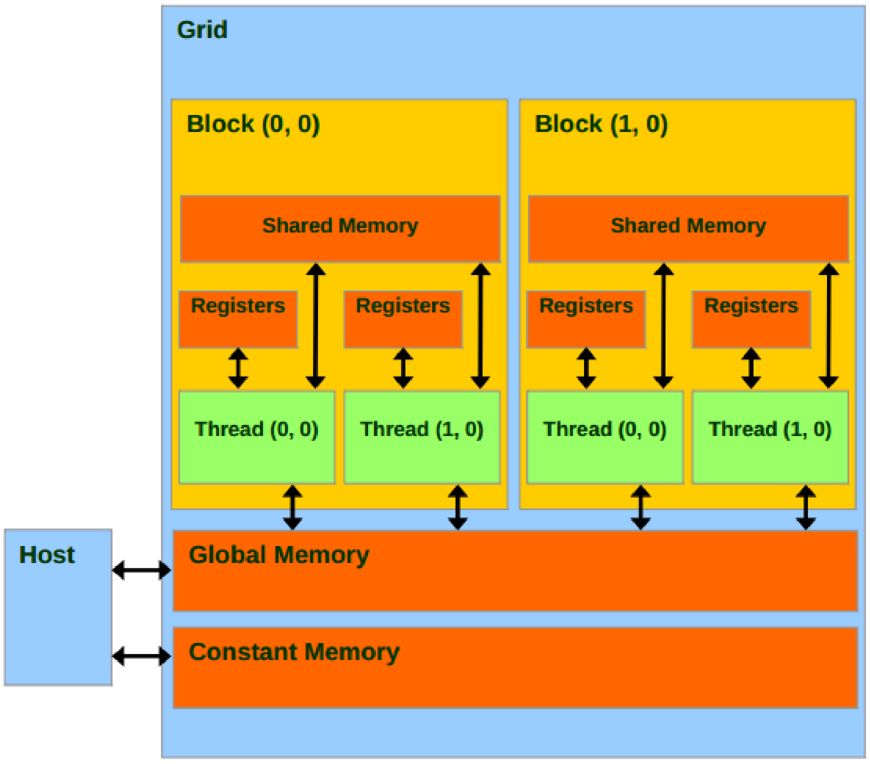


Figure 2.8: CUDA's memory model (image courtesy of NVIDIA).

2.7 OpenCL

As already stated, CUDA works solely for GPUs that are manufactured by NVIDIA. OpenCL (Open Computing Language), on the other hand, subjects its framework to an array of hardware devices, and it is not necessarily restricted to one vendor. These devices include FPGAs, DSPs, GPUs and CPUs. In this thesis, the latter two are considered.

OpenCL is managed by the Khronos group, a group devoted to creating open standards for accelerating parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices. The OpenCL framework uses a C/C++ like interface.

Architecturally, CUDA and OpenCL are strikingly similar. Just like with a CUDA implementation (see Section 2.6), OpenCL requires two code bases: the host and the kernel code. However a significant difference is that a kernel can be executed on both GPU and CPU devices. In OpenCL, these devices are partitioned into one or more compute units. A compute unit is further divided into a number of processing elements. The aforementioned is displayed in figure 2.9, which shows the platform model of OpenCL.

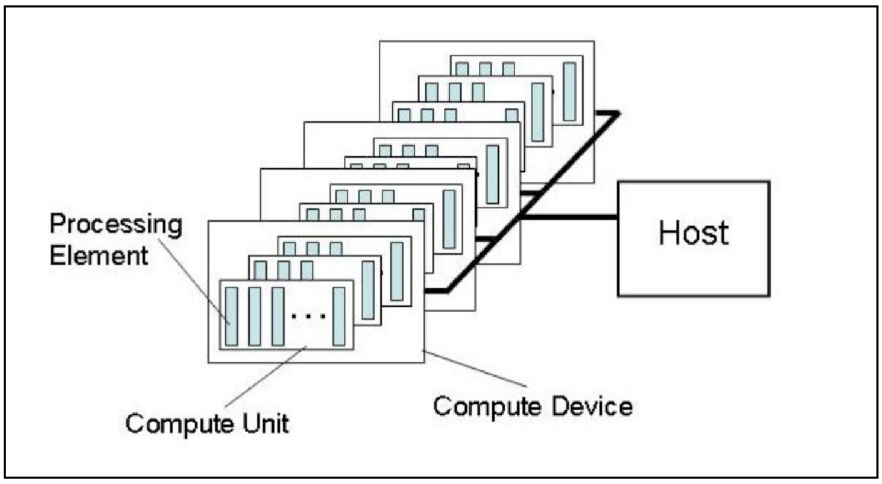


Figure 2.9: OpenCL's platform model (image courtesy of the Khronos group).

The platform model of OpenCL resembles the architecture of a GPU (see Section 2.5). A compute unit can be seen as a streaming multiprocessor. In addition a processing element can be considered as a core within the streaming multiprocessor. Although the platform model of OpenCL is not optimized for CPU architectures (since the platform model resembles the architecture of a GPU) recent research indicates[J S+13][J S+12] that OpenCL still is a viable alternative to efficiently enable parallelism on CPUs.

The kernel gets processed over an index space called N-Dimensional Range (NDRange). The NDRange is the equivalent of a grid in CUDA. An NDRange consists of a set of work-items (equivalent to threads in CUDA). These work-items are grouped in a work-groups (equivalent to blocks in CUDA). Compute units are able to process multiple work-groups in parallel. Figure 2.10 presents a 2-dimensional NDRange (OpenCL also allows the creation of 1 and 3-dimensional NDRanges). The similarities between the elements that are involved in CUDA's and OpenCL's kernel executions are depicted in table 2.1

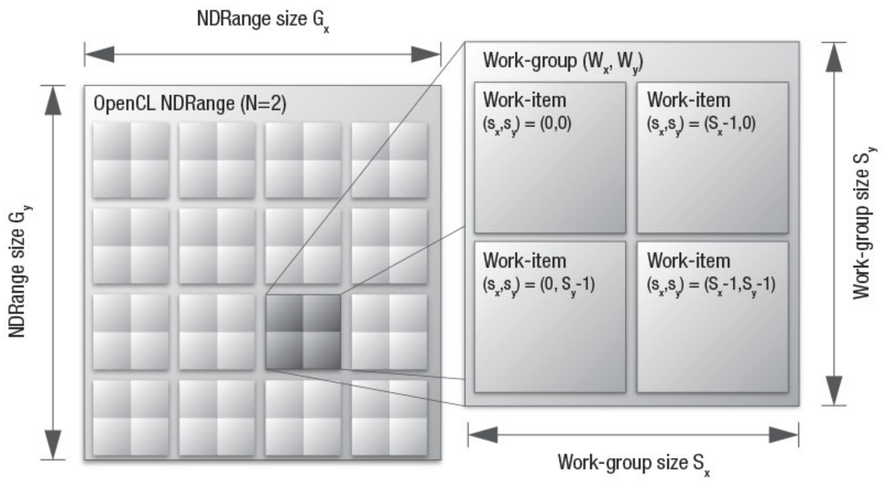


Figure 2.10: 2-dimensional NDRange (image courtesy of Khronos group).

Table 2.1: Similarities of the elements that are involved in the kernel execution between CUDA and OpenCL.

CUDA	OpenCL
Grid	NDRange
Block	Work-Group
Thread	Work-Item

Also the memory hierarchy that OpenCL introduces is highly similar with that of CUDA (see figure 2.8). The four memory types that OpenCL introduces are global, constant, local, and private memory.

The global and constant memory types can be accessed by every work-item. However constant memory is (cached) read only memory, whereas global memory facilitates both reading and writing.

Local memory is accessible by work items within the same work group (the CUDA equivalent of local memory is shared memory). The type of memory which can only be accessed by a single work-item is called private memory (the CUDA equivalent for this type of memory are registers). Table 2.2 outlines the similarities between the memory models of CUDA and OpenCL.

Table 2.2: Similarities of memory types between CUDA and OpenCL

CUDA	OpenCL
Global	Global
Constant	Constant
Shared	Local
Register	Private

CHAPTER 3

Related Work

In this chapter, we survey related work. Specifically, we briefly discuss different algorithms that solve the sequence alignment problem, we present alternative parallel solutions to accelerate the Smith-Waterman algorithm, and we address the issue of (performance) portability in OpenCL, which plays an important role in our multi-device analysis.

3.1 Sequence alignment

In the early days of protein sequence comparison, most known related proteins were compared over their whole lengths. In this case, global sequence alignment would be sufficient. However, soon proteins that shared only isolated regions of similarity were found [Mor99]. Extracting these regions is difficult through the use of the "global" Needleman-Wunsch algorithm [Nee+70]. To respond to these kinds of alignments, the Smith-Waterman algorithm [Smi+81] was constructed. Computationally wise, both these algorithms have the same sequential running time $\mathcal{O}(n \times m)$ when aligning a sequence of length m to a another sequence of length n. That is, in both cases an alignment matrix is constructed of size $n \times m$. However, the way in which a maximum scoring alignment is retrieved does differ between these algorithms.

With the Needleman-Wunsch algorithm, we can only choose entries from the last row or column as a starting point for an alignment. For the Smith-Waterman algorithm, we are not limited by this constraint. Both of these algorithms are exact, meaning that they always produce the most similar alignment between two sequences (or, in the case of local sequence alignment, the most similar region).

The exact Smith-Waterman algorithm was becoming too slow on the rapidly increasing datasets. Therefore, various heuristics have been developed to improve the runtime on large datasets. For example, BLAST [Alt+90] and FASTA[Lip+85] are both heuristics that solve the sequence alignment problem much faster, but without guaranteeing to extract the most optimal alignment.

3.2 Parallel implementations

To allow both fast execution and an exact solution, a lot of research has been conducted to determine how effectively can the the Smith-Waterman algorithm parallelized - i.e., how much performance can be gained from parallelism.

We can identify two different directions for parallelization: intra-sequence [Woz97] and inter-sequence [Rog11]. Inter-sequence parallelization is useful when many pairs of sequences need to be aligned simultaneously. Intra-sequence parallelization performs parallelization for each single pairwise alignment. PaSWAS [War+15] enables both types of parallelization.

In [Man+08], one of the first implementations of the Smith-Waterman algorithm in CUDA is explored. The authors convey the legitimacy of using a GPU for performing sequence alignments, based on the argument that the speed-up achieved over the heuristic BLAST is 2.4x (thus, the first CUDA implementation is faster and exact). Furthermore, the CUDA implementation is compared to a CPU-based solution described in [Far07]. It is noted that the CPU implementation outperforms the CUDA implementation for longer sequences. This observation further justifies the need for an in-depth study on how a specific hardware platform influences the performance of the parallel Smith-Waterman algorithm.

All these implementations utilize an affine gap penalty score, which means that we have a different score for opening and extending a gap. In addition, each of the implementations steps do not take the trace back step into account. Compared with these versions, the starting algorithm for our study is the most advanced: it produces exact solutions, it is fast, and it enables the alignment of large sequences.

3.3 OpenCL, CUDA, and portability

The differences between CUDA and OpenCL have been thoroughly analyzed. For example, in [Fan+11; Kom+10] thorough analysis is conducted to determine the explanation of the performance gap between these two parallel programming frameworks. To this end, the authors, critically evaluate the generated PTX code (a pseudo-assembly language for CUDA and OpenCL when used in conjunction with NVIDIA GPUs) of both CUDA and OpenCL. This code can be used to understand how different optimization decisions are taken by the CUDA and/or OpenCL compilers. The authors identify several optimization techniques which are automatically enabled by CUDA and should be done manually if one wants have a performance-comparable OpenCL implementation. One of these optimization techniques was also found during our research, namely CUDA automatically enables loop unrolling[Aho86], while in OpenCL this is not done by default.

With regard to the interplay of OpenCL with CPUs, the authors of [J S+13] have defined a set of performance traps that should be avoided when programming in OpenCL for a CPU device. The architectural model of OpenCL somewhat resembles the hardware architecture of a GPU, therefore we have an inherent architectural mismatch between a

CPU and a GPU. Careful code specialization should be performed to allow OpenCL code to obtain good performance on the CPU. According to the authors of [J S+13], the most important mismatches are appear due to the parallelism granularity and memory model. OpenCL assumes fine-grained parallelism, i.e., each core performs a small amount of work, but CPU cores are equipped for larger workloads (i.e., multi-core CPUs prefer coarse-grained parallelism). In addition, the memory model that OpenCL uses does not directly map to a CPU. This is especially hurtful, performance-wise, for local memory, which typically leads to performance improvements when used on the GPU, and performance penalties when used on CPUs. This mismatch happens because the OpenCL local memory does not map to a specific cache, but rahter to the global memory itself. When creating our OpenCL PaSWAS implementation, we took into account these differences, and quantified the impact of these perfrmannce parameters for each platform, CPU or GPU.

CHAPTER 4

SeqSWAS

We will explore the sequential solution to the Smith-Waterman algorithm, in this chapter. This algorithm is used to calculate the speedup (i.e., achieved performance benefit) of our parallel implementations. The sequential algorithm has adopted the name SeqSWAS which stands for Sequential Smith-Waterman Alignment Software.

We can distinguish four steps within the execution of SeqSWAS, these are:

1. Initialize the datastructure
2. Calculate the alignment matrix
3. Trace back from the alignment matrix
4. Output the alignment(s)

4.1 Initialization

In the initialization phase, two files are read from the filesystem: the sequences and the targets. Both of these files are composed out of one or more sequences with variable length (for a more elaborate view on what type of data these sequences contain we refer you to 2.2). While reading these sequences into memory they are padded to a constant length. This constant length is determined by the size of the largest sequence.

Thus, if we have two sequences s_0 and s_1 and $|s_0| > |s_1|$ in a single file, sequence s_2 is padded with the character x such that $|s_0| = |s_1|$. The padded sequences are then concatenated and stored in a variable $S = s_0 s_1$. The padding scheme, facilitates a somewhat trivial manner to retrieve a single sequence s_i from S.

Because the padded characters are added artificially (i.e., these characters were not present in the original data), aligning characters to x will result in a very low alignment score. In this way we do not compromise the overall result of the alignment, since the Smith-Waterman algorithm is a maximization problem, i.e., we are looking for high alignment scores. From now on, we will consider s_0 and t_0 as the largest sequence, and target, respectively.

(i.e., $s_0 > s_i$ voor alle i>0 and $t_0 > t_i$ voor alle i>0)

4.2 Calculate the alignment matrix

After the initialization phase, the actual alignment matrix A can be calculated. Each element a_{ij}, where $1 \leq i \leq |s_0|$ and $1 \leq j \leq |t_0|$, is computed using the recurrence relationship defined in 2.1. The variables $|s_0|$ and $|t_0|$ represent the length of the longest sequence and target. An example of an alignment matrix is given below:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & a_{11} & a_{12} & a_{13} & \dots & a_{1|t_0|} \\ 0 & a_{21} & a_{22} & a_{23} & \dots & a_{2|t_0|} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{|s_0|1} & a_{|s_0|2} & a_{|s_0|3} & \dots & a_{|s_0||t|} \end{bmatrix}$$

Figure 4.1: The initialization of the alignment matrix

The first row and column of the matrix A are padded with 0, to simplify the process of evaluating the recurrence relationship. The pseudocode for generating the remaining values a_{ij} is presented in Algorithm 1.

From Algorithm 1 we can see that an element a_{ij} can either originate from its upper (a_{i-1j}), left (a_{ij-1}) or upper left (a_{i-1j-1}) neighbor. In the former two cases we introduce a gap score and in the latter case we align s_i with t_j and use the matrix W to determine the (mis)alignment score of these two characters. Figure 4.2 provides a visual representation of the aforementioned. If all of these values are negative, a_{ij} remains 0 and the direction from which a_{ij} originates is undetermined (this refers to the direction ND in algorithm 1, which stands for No Direction).Thus every element in the alignment matrix is positive (i.e., ≥ 0)

$$\begin{tikzcd}
a_{i-1j-1} & a_{i-1j} \\
a_{ij-1} & a_{ij} \arrow[l, "Gap"'] \arrow[u, "Gap"] \arrow[ul, "W(s_i{,}t_j)"']
\end{tikzcd}$$

Figure 4.2: The dependencies between elements of the alignment matrix

Within the calculation of the alignment matrix we also keep track of the overall maximum (which is signified by the variable *global_max* in Algorithm 1) and the direction from where a score originated, stored in the matrix D (using the abbreviated form of the direction: U(p), L(eft) and U(pper)L(eft)). These elements are added to simplify the process of tracing back.

```
Input: Sequence s, Target t, $s_0$, $t_0$
Output: Matrix A, Matrix D, global_max
A[$s_0$+1][$t_0$+1] = {0};
D[$s_0$][$t_0$] = {0};
global_max = 0;
for $i = 1 \ldots s_0$ do
    for $j = 1 \ldots t_0$ do
        score=0;
        direction=ND;
        // Compute surrounding values
        up=A[i-1][j]+gap;
        left=A[i][j-1]+gap;
        upper_left=A[i-1][j-1]+W($s_i$,$t_j$);
        if up > score then
            score = up;
            direction = U;
        end
        if left > score then
            score = left;
            direction = L;
        end
        if upper_left > score then
            score = upper_left;
            direction = UL;
        end
        A[i][j] = score;
        D[i-1][j-1] = direction;
        global_max = max(score,global_max);
    end
end
```

Algorithm 1: Sequential alignment matrix calculation

4.3 Trace back from the alignment matrix

The trace back routine is responsible for generating the (higher scoring) alignments. To this end, we introduce a new data structure, namely a list of StartingPoint objects. In addition we utilize the Direction matrix D constructed in Algorithm 1. A StartingPoint indicates from where in the alignment matrix A to start the trace back.

Element a_{ij} is chosen as a StartingPoint when it has a sufficiently high alignment score (i.e., $a_{ij} \geq x \times global\_max$ where x is configurable by the user, most commonly such that $0 < x \leq 1$). When a StartingPoint has been found, the actual trace back can be conducted. That is, we want to determine the path that was taken to this StartingPoint.

This path is recorded within the direction matrix D. The end of an alignment is indicated by an a_{xy} for which $a_{xy} = 0$. Note that these elements have an undetermined direction (ND), therefore the path of the alignment can not be extended any further. Thus the end of an alignment is indicated by a neighbour who has a direction equal to ND. To indicate this end, a new direction type is introduced: stop direction (SD). The pseudocode for the trace back routine is presented in Algorithm 2

When a_{ij} is marked, it has a negative value. Because each element is positive ($a_{ij} \geq 0$), when $a_{ij} < 0$ we can consider it to be marked. An important thing to note about algorithm 2 is that the Matrix A is evaluated from element $a_{XY} \ldots a_{11}$ whereas with Algorithm 1 the elements are processed in the order of $a_{11} \ldots a_{XY}$.

```
Input: $s_0$, $t_0$, Matrix A, Matrix D
Output: StartingPointList
for $i = s_0 \dots 1$ do
    for $j = t_0 \dots 1$ do
        score = A[i][j];
        if score is sufficiently high $\wedge$ unmarked then
            set score as StartingPoint;
            Add StartingPoint to StartingPointList;
            mark score;
        end
        // Perform actual trace back
        while score is marked do
            Determine neighbor of score by looking at matrix D[$score_i$][$score_j$];
            set score to neighbors value (e.g A[i-1][j],A[i][j-1] or A[i-1][j-1]);
            if score equals 0 then
                D[i][j] = SD;
            else
                mark score;
            end
        end
    end
end
```

Algorithm 2: Sequential trace back calculation

4.4 Output the alignment(s)

Using the data generated by Algorithm 2, the actual alignments can be presented to the user. By iterating through the StartingPoint list and using the Direction matrix, we can determine how a sequence $s = s_1 \dots s_{s_0}$ is aligned to a target $t = t_1 \dots t_{t_0}$. That is, we can present to the user where a gap is placed (in either the sequence or the target), or which s_i and t_j are matched. The full algorithm of SeqSWAS is presented in Algorithm 3.

```
Input: Sequences S, Targets T, $s_0$, $t_0$
S:= read content of sequence file;
T:= read content of target file;
for $i = 1 \ldots |S|$ do
    for $j = 1 \ldots |T|$ do
        call Algorithm 1 with sequence $s_i$ and target $t_j$;
        call Algorithm 2 with alignment matrix A and direction matrix D;
        Output results using StartingPointList and Direction matrix;
    end
end
```

Algorithm 3: SeqSWAS

CHAPTER 5

PaSWAS

PaSWAS (Parallel Smith-Waterman Alignment Software) is the parallel counterpart of the sequential SeqSWAS. We discuss both CUDA and OpenCL In this chapter we will take a closer look at PaSWAS. More specifically, the way in which parallelism is enabled, is examined. PaSWAS enables both inter and intra-kernel parallelization. Inter-kernel parallelization is accomplished by performing multiple sequence alignments in one iteration, instead of processing the sequence aligments one by one (see algorithm 3). Since each sequence alignment can be executed independently, the act of inter-kernel parallelization is relatively easily accomplished. From SeqSWAS we identified four steps, these were:

1. Initialization (One time only).
2. Calculation of the alignment matrix (Done for each sequence alignment).
3. Trace back from the alignment matrix (Done for each sequence alignment).
4. Output the alignment(s) (Done for each sequence alignment).

Steps two and three are subjected to intra-kernel parallelization, i.e., the computation of the alignment Matrix A is parallelized. From the previous chapter, we have seen an inherent dependency between the elements a_{ij} (see Figure 4.2)

With this data dependency, it is not trivial to introduce parallelism. By computing the anti-diagonal at the same time, parallelism is enforced. This scheme is commonly known as the wavefront pattern [Anv+01], which works as follows: when a_{11} is known, the entries a_{12} and a_{21} can be computed in parallel by two work-items. The entries a_{13}, a_{22} and a_{31} can only be evaluated in parallel (by three work-items) when a_{12} and a_{21} are known. The maximum number of work-items that can be active within a sequence alignment is given by $min(|s_0|,|t_0|)$, where $|s_0|$ is the length of the largest sequence and $|t_0|$ the length of the largest target. This process is illustrated in Figure 5.1 for $|s_0| = |t_0| = 3$.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & {\color{red}a_{11}} & {\color{blue}a_{12}} & {\color{green}a_{13}} \\ 0 & {\color{blue}a_{21}} & {\color{green}a_{22}} & {\color{orange}a_{23}} \\ 0 & {\color{green}a_{31}} & {\color{orange}a_{32}} & {\color{yellow}a_{33}} \end{bmatrix}$$

Figure 5.1: Parallel calculation of alignment matrix A.

Elements that have the same color can be computed in parallel. With regard to the trace back step, the same parallelization technique is used. However, the computations start from element $a_{|s_0||t_0|}$. A single work-item in this step is responsible for determining whether it is a StartingPoint and to mark his neighbor when it is a part of an alignment.

In OpenCL (or CUDA) work-item synchronization can only occur within a work-group. These synchronizations come in the form of barriers. A barrier ensures that the kernel execution of a work-item resumes after each work-item (within a work-group) has accessed this barrier. This construct is particularly useful in case of the wavefront pattern, since we can ensure the correct order of execution.

The number of work-items that can execute within a work-group is limited by the hardware. For example the NVIDIA GTX480 allows for 1024 work-items in a single work-group. Since we are evaluating a matrix, a possible work-group configurations could be 1x1024, 2x512, 256x4 and so on. What happens when $|s_0| \times |t_0|>1024$? In this case we would need multiple blocks to evaluate this alignment. However, synchronization between work-groups cannot be enforced [Seo+11] within a single kernel execution.

A possible solution to this problem is to execute kernels sequentially [Reh+09], within the host. Each kernel is then reponsible for computing one or more work-groups, which can be processed independently. The illustration in figure 5.2 gives an impression of this computation. In this case we need nine work-groups to compute a single alignment. Work-groups (denoted by wg_{ij}) with the same color can be executed in a single kernel run.

Thus the wavefront pattern is applied on both the work-item and work-group scale. When we calculate the alignment matrix we start from wg_{00} and work our way through wg_{22}. Whereas with the trace back phase we start from wg_{22} and end in wg_{00}.

$$\begin{bmatrix} {\color{red}wg_{00}} & {\color{blue}wg_{01}} & {\color{green}wg_{02}} \\ {\color{blue}wg_{10}} & {\color{green}wg_{11}} & {\color{orange}wg_{12}} \\ {\color{green}wg_{20}} & {\color{orange}wg_{21}} & {\color{yellow}wg_{22}} \end{bmatrix}$$

Figure 5.2: The order in which the work-groups are processed.

5.1 Local-Memory in PaSWAS

As already stated in Sections 2.6 and 2.7, the usage of local-memory (or shared-memory in CUDA) can have a performance enhancing effect on the GPU. To this end a PaSWAS algorithm is created that employs the faster access times of the local memory compared to continuously accessing global memory. We present this algorithm in this section. We assume that a single work-group has a dimension of S_s by S_t (note that for the GTX480, $S_s \times S_t \leq 1024$). In addition, we assume that the sequences and targets are padded such that they divide perfectly by S_s and S_t, respectively.

The total amount of work-groups needed to compute a single alignment is then given by $\frac{|s_0|}{S_s} \times \frac{|t_0|}{S_t}$.

The data structures that are populated for the calculation of the alignments are: the alignment matrix A, the direction matrix D, and maximum matrix M. Each of these data structures are stored within the local-memory domain. If we look at wg_{00} from figure 5.2, its local alignment matrix would appear as follows:

$$\begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & a_{11} & a_{12} & \dots & a_{1S_t} \\ 0 & a_{21} & a_{22} & \dots & a_{2S_t} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 0 & a_{S_s1} & a_{S_s2} & \dots & a_{S_sS_t} \end{bmatrix}$$

Figure 5.3: Local alignment matrix A of first work-group.

The actual computation of each element a_{ij} does not differ between work-groups. That is, the same type of computations are performed for wg_{00} or any arbitrary wg_{ij}. However, the way in which the local-memory is initialized does vary. wg_{00} does not depend in any way on the computations of other work-groups. Therefore, during initialization the first row and column of this local alignment matrix are padded with 0's.

On the other hand, wg_{01} does depend on the scores that wg_{00} has generated. In this case the first row of the alignment matrix of wg_{01} is padded with 0's whereas the first column (except for a_{00}) is filled with halo values (in this case $a_{1S_t} \dots a_{S_sS_t}$) . These halo values are retrieved from global memory.

After each work-group has computed its local alignment matrix, the contents are stored back to the global memory with the omission of the halo and padded values. The possible configurations of the local alignment matrix A (other than the first work-group) are displayed in Figure 5.4. In this Figure, x represents the to-be-computed elements by a single work-group. In addition N,E, and NE represent a single global memory access, of respectively, the upper, left, and upper-left work-groups.

The pseudocode for generating these elements x is displayed in algorithm 4

As with SeqSWAS Algorithm 1 we also want to calculate the overall maximum score in PaSWAS. The same pattern depicted in figure 5.2 is applied for calculating the

$$\begin{bmatrix} 0 & N & N & N \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix}$$

(a) Upper wg is already computed

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ E & x & x & x \\ E & x & x & x \\ E & x & x & x \end{bmatrix}$$

(b) Left wg is already computed

$$\begin{bmatrix} NE & N & N & N \\ E & x & x & x \\ E & x & x & x \\ E & x & x & x \end{bmatrix}$$

(c) Upper-left wg is already computed

Figure 5.4: An overview of the possible local alignment matrix initializations

```
**Input**: Sequence s, Target t
**Output**: Matrix A, Matrix D, Matrix M
Determine work-item id ($w_{S_s}$,$w_{S_t}$);
Initialize local alignment matrix based on work-group id $L_a[S_s + 1][S_t + 1]$;
Initialize local direction matrix $L_d[S_s][S_t]$ to ND;
Initialize local max matrix $L_m[S_s][S_t]$ by retrieving max value of neighboring $wg$;
Determine the to be aligned characters ($s_i$,$t_j$) based on work-item and
work-group id;
**for** $i = 0 \ldots S_s + S_t - 1$ **do**
    **if** $i$=$w_{S_s}$+$w_{S_t}$ **then**
        up=$L_a[w_{S_s}][w_{S_t}$+1]+gap;
        left=$L_a[w_{S_s}$+1][$w_{S_t}$]+gap;
        upper_left=$L_a[w_{S_s}][w_{S_t}]$+W($s_i$,$t_j$);
        Determine the best *score* and its *direction*;
        $L_a[w_{S_s}+1][w_{S_t}+1]$ = *score*;
        $L_d[w_{S_s}][w_{S_t}]$ = *direction*;
        Determine $L_m[w_{S_s} - 1][w_{S_t} - 1]$ according to 5.5
    **end**
    Barrier;
**end**
Store $L_d$ into global direction memory;
Store $L_a$ without padded and halo values into global alignment memory;
Store work-group maximum $L_m[S_s - 1][S_t - 1]$ into global maximum memory;
```

Algorithm 4: Matrix A, Matrix D, Matrix M calculation

maximum score. Each work-group calculates its maximum and propagates this value to its neighbor, this propagation occurs via global memory. For example, wg_{11} utilizes the (already stored in global memory) maximum value of the work-groups wg_{10}, wg_{01}, and wg_{00} to determine its own maximum. Eventually, the last work-group (i.e. wg_{22} in 5.2) will store the overall maximum of an alignment. Figure 5.5 gives an overview of how the maximum is calculated within a work-group. The value m_{00} is determined by max(L_a[1][1], maximum value from surrounding wg). m_{10} in turn is computed using max(L_a[2][1],m_{00}) As you may notice these computations also adhere to the wavefront pattern. That is, m_{11} can only be computed when m_{01} and m_{10} are known.

$$\begin{tikzcd}
m_{00} & m_{01} \arrow[l, "max"] & \cdots \arrow[l, "max"] & m_{1S_t-1} \arrow[l, "max"] \\
m_{10} \arrow[u, "max"] & m_{11} \arrow[l, "max"] \arrow[u, "max"] & \cdots \arrow[l, "max"] \arrow[u, "max"] & m_{1S_t-1} \arrow[l, "max"] \arrow[u, "max"] \\
\vdots \arrow[u, "max"] & \vdots \arrow[l, "max"] \arrow[u, "max"] & \cdots \arrow[l, "max"] \arrow[u, "max"] & \vdots \arrow[l, "max"] \arrow[u, "max"] \\
m_{(S_s-1)0} \arrow[u, "max"] & m_{(S_s-1)1} \arrow[l, "max"] \arrow[u, "max"] & \cdots \arrow[l, "max"] \arrow[u, "max"] & m_{S_s-1S_t-1} \arrow[l, "max"] \arrow[u, "max"]
\end{tikzcd}$$

Figure 5.5: The dependencies between elements of the alignment matrix

As already stated, the parallel trace back step also adheres to the wavefront pattern. Only, in this step, we start our computations from the last work-group and work our way to wg_{00}. A single work-group is processed according to algorithm 5

In algorithm 5, the StartingPoints are added to the StartingPointList using a global index value. This global index value is incremented atomically[1] [Stu+92] and signifies where a StartingPoint may be stored in the StartingPointList.

In addition, the barriers between marking neighboring scores are needed to circumvent race conditions. Race conditions occur when two or more threads adjust the contents of the same memory location [Net+92]. For example, in figure 5.3, a_{21}'s upper neighbour is the same as a_{12}'s left neighbor (namely a_{11}). Thus, without the barriers, it might be possible that both a_{21} and a_{12} will simultaneously adjust a_{11}, leading to inconsistent results.

Intra-kernel parallelization can be enabled (e.g., computing multiple alignments concurrently) by launching multiple work-groups at the same time. Referring to figure 5.2, instead of executing one work-group wg_{00}, a multitiple of wg_{00} can be launched, each solving a different alignment. In the next iteration a multitude of work-groups wg_{10} and wg_{01} are executed, further solving their alignments. This process is repeated until we reach wg_{22}. The amount of intra-kernel parallelization is dependent on $|s_0|$,

1 Atomic operations enforce that a set of instruction are executed without interruption.

```
Input: $s_0$, $t_0$, Matrix A, Matrix D, Matrix M
Output: StartingPointList
Determine work-item id ($w_{S_s}$,$w_{S_t}$);
Retrieve alignment $S_s \times S_t$ scores based on work-group id from global alignment
memory and store in local memory $L_a$ ;
Retrieve maximum from global maximum matrix;
for $i = 0 \ldots S_s + S_t - 1$ do
    if $i$=$w_{S_s}$+$w_{S_t}$ then
        score = $L_a[w_{S_s}][w_{S_t}]$;
        if score is sufficiently high & unmarked then
            set score as StartingPoint;
            Add StartingPoint to StartingPointList;
            mark score;
        end
        Mark upper-left neighbor when marked score came from the upper-left
        direction;
        barrier;
        Mark upper neighbor when marked score came from the upper direction;
        barrier;
        Mark left neighbor when marked score came from the left direction;
        barrier;
    end
    barrier;
end
```

Algorithm 5: Parallel trace back calculation

$|t_0|$, and the memory available on the device.

5.2 Global-Memory in PaSWAS

In addition to the local memory implementation, we have created another implementation of PaSWAS which only utilizes the global memory. One of the main reasons for creating this algorithm is to determine whether employing a solely global memory based implementation can have a performance enhancing effect on the CPU. It can also be used as a validation for the local memory implementation, as to whether it is beneficial to have global-local memory transfers for the GPU.

In general, with OpenCL CPU implementations, all memory spaces (e.g. constant, local, global memory spaces) are mapped onto the same hardware memory, so a kernel that makes explicit use of constant and local memory spaces may actually incur more overhead than a kernel that only uses global memory references[Sto+10]. Thus, we expect that for the CPU the global memory implementation should yield superior performance compared to the local implementation.[1]

The global implementation was derived from the local one, by omitting the transfers between the global and local memory spaces. Moreover, any reference made to the local memory was replaced by their global counterpart. However, this is not entirely efficient. For example, the types of initializations for the work-group alignment matrix illustrated in figure 5.4 are not needed in the global case, since we can directly access these halo and padded values from the global memory. The only initialization that needs to be performed on the alignment matrix is that we start with a matrix such as figure 4.1 i.e., the first row and column of the matrix should be padded with zeroes. Using the function `ClEnqueueFillBuffer`, introduced in OpenCL 1.2, one can achieve this.

5.2.1 Increasing the Granularity

In [She+13] it is stated that the fine grained parallelism that OpenCL naturally offers can be both advantageous as well as disadvantageous for the performance of an application that will be executed on an CPU. The fine grained parallelism is enabled in PaSWAS by letting each work-item be responsible for a single element of the alignment,direction and maximum matrix. With coarse-grained parallelism we want a work-item to be responsible for the computation of more than one element. The coarse-grained implementation will be discussed in this section.

We introduce two new parameters namely WL_s and WL_t. Their values determine how many entries of the matrices have to be computed by a single work-item. More specifically, a work-item processes $WL_s \times WL_t$ entries (WL is an abbreviation of workload). Within a work-group we still compute $S_s \times S_t$ elements, thus the total

1 We will further refer to this global memory implementation as "global implementation", and to the local memory version as "local implementation".

amount of work-items that can be active within a work-group is:

$$\frac{S_s \times S_t}{WL_s \times WL_t} \tag{5.1}$$

The algorithm for calculating the scores can then be rewritten to algorithm 6

```
**Input**: Sequence s, Target t
**Output**: Matrix A, Matrix D, Matrix M
Determine work-item id ($w_{\frac{S_s}{WL_s}}$,$w_{\frac{S_t}{WL_t}}$);
for $i = 0 \dots \frac{S_s}{WL_s} + \frac{S_t}{WL_t}$ do
    if $i$=$w_{\frac{S_s}{WL_s}}$+$w_{\frac{S_t}{WL_t}}$ then
        for $j = 0 \dots WL_s - 1$ do
            for $k = 0 \dots WL_t - 1$ do
                Retrieve character $s_x$;
                Retrieve character $t_y$;
                Determine best score and its direction;
                Calculate maximum score;
            end
        end
    end
    Barrier;
end
```

Algorithm 6: Coarse grained alignment, direction and max matrix calculation

Essentially, this algorithm is the same as algorithm 4, but it introduces two extra for-loops. The index x in algorithm 6 is a function of WL_s and S_s, while y is determined by WL_t and S_t.

The process of tracing back does differ from algorithm 5 and is presented in algorithm 7. The main difference between algorithm 7 and 5 (besides the nested for-loop) is that the barriers are replaced with semaphores, when marking neighboring values. Barriers are not suitable in this case, since we cannot ensure that all work-items will access these barriers. Semaphores [Dij68] are particularly useful because they enforce that only one work-item can access a specific resource (in this case an element of the alignment matrix). The functions `getSemaphore` and `releaseSemaphore` are displayed in algorithm 8 and algorithm 9.

We have a semaphore for each element a_{ij} in the alignment matrix, thus we have $|s_0| \times |t_0|$ semaphores. These semaphores are initialized to 0 with `ClEnqueueFillBuffer`, The function `atom_xchg` is provided by OpenCL and stores a value in $semaphore_{xy}$ while returning the previous content of this semaphore.

```
Input: $s_0$, $t_0$, Matrix A, Matrix D, Matrix M
Output: StartingPointList
Determine work-item id $(w_{\frac{S_s}{WL_s}}, w_{\frac{S_t}{WL_t}})$;
Retrieve maximum from global maximum matrix;
for $i = 0 \dots \frac{S_s}{WL_s} + \frac{S_t}{WL_t}$ do
    if $i = w_{\frac{S_s}{WL_s}} + w_{\frac{S_t}{WL_t}}$ then
        Retrieve $score_{xy}$ from alignment matrix;
        for $j = 0 \dots WL_s - 1$ do
            for $k = 0 \dots WL_t - 1$ do
                if score is sufficiently high & unmarked then
                    set score as StartingPoint;
                    Add StartingPoint to StartingPointList;
                    mark score;
                end
                $getSemaphore_{x-1y-1}$ when marking upper-left neighbor;
                $releaseSemaphore_{x-1y-1}$ when upper-left neighbor is marked;
                $getSemaphore_{xy-1}$ when marking upper-left neighbor;
                $releaseSemaphore_{xy-1}$ when upper-left neighbor is marked;
                $getSemaphore_{x-1y}$ when marking upper-left neighbor;
                $releasSemaphore_{x-1y}$ when upper-left neighbor is marked;
            end
        end
    end
    barrier;
end
```
Algorithm 7: Coarse grained parallel trace back calculation

```
Input: $semaphore_{xy}$
occupied:=atom_xchg($semaphore_{xy}$,1);
while occupied > 0 do
    occupied:=atom_xchg($semaphore_{xy}$,1);
end
```
Algorithm 8: The getSemaphore primitive

```
Input: $semaphore_{xy}$
atom_xchg($semaphore_{xy}$,0);
```
Algorithm 9: The releasSemaphore primitive

CHAPTER 6

Data Transfers

In this chapter we investigate the effect of data transfers on the overall performance. In turn, we describe techniques to alleviate the costs incurred by data transfers.

An important issue for the performance of an OpenCL application (or CUDA for that matter) is the data transfer between a host and a device [Kar+10]. Commonly these transfers are bidirectional. That is the host transfers data to a device, in turn the device generates the output from the input data and sends it back to the host. In PaSWAS, only a single datastructure is send from the host to device (H2D), namely `h_maxPossibleZeroCopy`. This data structure aids in fine tuning the alignments accumulated from the trace back phase. Both the StartingPointList and Direction matrix are send back from the device to the host (D2H) after the trace back phase is completed.

When using a discrete GPU device, these transfers occur via the PCI-Express (PCIe) bus. The NVIDIA GTX480 GPU has the capabilty of PCI-E 2.0x16, which enables a memory bandwidth of 8 GB/s. To put things into perspective, the device memory of the GTX480 can be accessed with a bandwidth of 177.4 GB/s. Thus H2D and D2H transfers can be a serious bottleneck within an application [M D+11]. Figure 6.1 provides a visual overview of the transferring scheme between a host and a device.

A way in which data transfer can be minimized is by utilizing zero-copy memory [Pha+10]. When data has been allocated into zero-copy memory, a device can access it from the host without performing an explicit copy, hence the name zero-copy.

Although OpenCL provides portability between different platforms and vendors, each of these vendors have their own tricks to enable zero-copy. In this section we will elaborate on how to create zero-copy memory with OpenCL and CUDA.

For Intel CPUs data can be page-locked by invoking the function `clCreateBuffer` with one of the flags `CL_MEM_USE_HOST_PTR` or `CL_MEM_ALLOC_HOST_PTR`.

According to [Int14] `CL_MEM_ALLOC_HOST_PTR` is commonly used when you want the runtime to handle the creation of the memory buffer object. Whereas `CL_MEM_USE_HOST_PTR` should be utilized when we already have some data on the host and want to load this data in a memory buffer. Using the function `clEnqueueMapBuffer` we can give the host

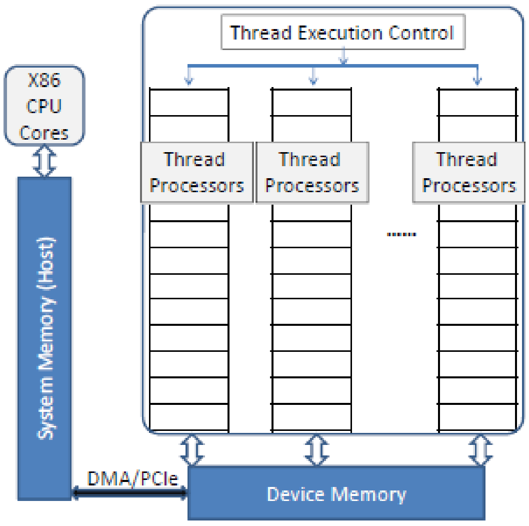


Figure 6.1: data transfer between a host and a device (image courtesy of [M D+11]).

read and write access to the memory object. These permissions are given to the device by invoking the function `clEnqueueUnmapMemObject`

Creating pinned memory for OpenCL NVIDIA is slightly different[Nvi11]. A buffer, that will likely be zero-copy , is created with the flag `CL_MEM_ALLOC_HOST_PTR`. More specifically, two calls to the function `clCreateBuffer` are made. One buffer is created using one of the flags `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` or `CL_MEM_READ_WRITE`. In addition, another call is made to `clCreateBuffer` only this time the flag

`CL_MEM_ALLOC_HOST_PTR` is used, in combination with one of the three flags mentioned above. With `clEnqueueMapBuffer` the host data is mapped to the device buffer. H2D and D2H communication occurs (e.g. with `clEnqueueReadBuffer` or `clEnqueueWriteBuffer`) between the host pointer and the buffer created without the flag `CL_MEM_ALLOC_HOST_PTR`. When applying zero-copy in this manner the page-locked buffers are allocated twice on a GPU device.

Lastly CUDA, enables the creation of zero-copy memory by invoking the function `CudaHostAlloc` with the flag `CudaHostAllocMapped`. With `CudaHostGetDevicePointer` one retrieves a device pointer from the host memory. This device pointer can then be used in a kernel.

6.1 Empirical Evaluation

By running a microbenchmarking suite we want to determine the best configuration for the creation of memory objects. The kernel for this microbenchmarking suite computes the square of each element in a vector $size^2$ and stores the result in a output vector. The experiments included three hardware platforms namely, a Intel multi-core CPU and two NVIDIA GPUs. These hardware platforms will be used throughout this thesis and the specifications of these devices can be found in table 7.1.

With regard to the CPU hardware platform, we define three ways to allow for memory transfers these are: normal memory transfers (indicated by *none* in figure 6.2, 6.3 and 6.4), Zero-copy handled by the runtime (through `CL_MEM_ALLOC_HOST_PTR`) and finally zero-copy initialized by the user, through the flag of `CL_MEM_USE_HOST_PTR`). The results for the CPU are shown in figure 6.2. We can immediately see that no zero-copy incurs important performance penalties. Utilizing either `CL_MEM_ALLOC_HOST_PTR` or `CL_MEM_USE_HOST_PTR`, the desired effect of zero-copy is accomplished, i.e., H2D an D2H have been reduced to zero (which, in turn, leads to better end-to-end performance of the kernel).

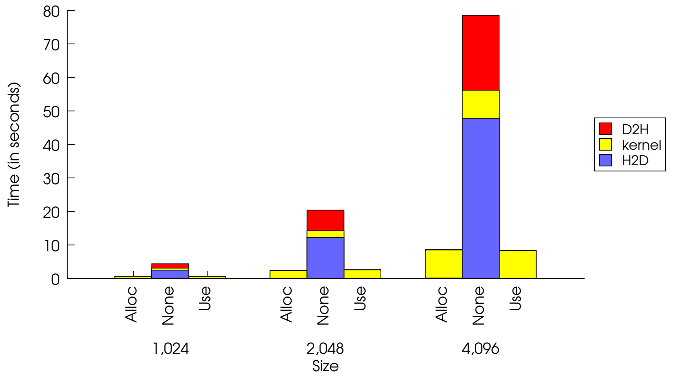


Figure 6.2: Performance of memory transfers CPU.

The performance difference between `CL_MEM_ALLOC_HOST_PTR` and `CL_MEM_USE_HOST_PTR` is too small to be easily visible; however, when looking at the raw performance data, we observe that `CL_MEM_USE_HOST_PTR`-based solution has an slight performance advantage over the `CL_MEM_ALLOC_HOST_PTR`-solution. Therefore, we will utilize `CL_MEM_USE_HOST_PTR` when needing a zero-copy implementation in our subsequent analysis.

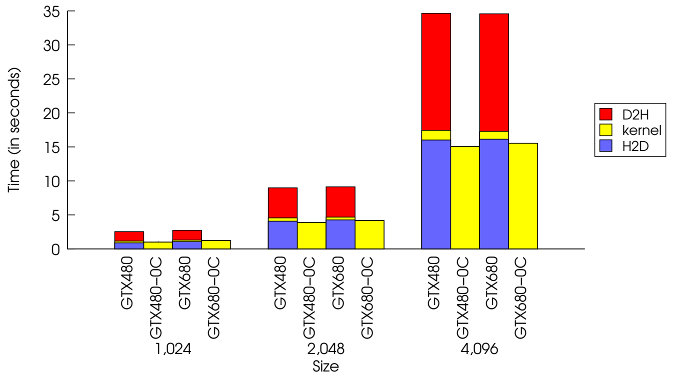


Figure 6.3: Performance of memory transfers CUDA.

With regard to CUDA, we have a single scheme to enforce zero-copy - depicted by either GTX480-0C and GTX680-0C in figure 6.3. In this figure, we note that the zero-copy scheme is tightly coupled with an increase in kernel execution time. Indeed, the memory transfers seem to be reduced to zero, but the execution time of the kernel has now increased significantly. This behavior is due to the fact that we are still working with discrete graphics cards, and therefore data still needs to be transferred through the PCIe bus, only this is now included (from our measurements), in the kernel time. We do, however, notice a significant improvement in the end-to-end kernel execution time between the non-zero copy and zero-copy.

For the OpenCL GPU implementations, we combine the methods suggested by [Int14] and [Nvi11] to determine which scheme will yield the best performance. These are given by Use, $Alloc$ and, the one suggested by NVIDIA, $Alloc - NVIDIA$. We present in figure 6.4, a performance comparison of all these memory transferring schemes. From this figure, we see that the performance utilizing no zero-copy is the worst performing version. The second best solution is given by the `CL_MEM_USE_HOST_PTR` construct (note that the H2D and D2H transfers are still present). The construct that was discussed in [Nvi11] provides the best performance, although - as expected - these transfers are not completely removed (we are, again, using a discrete GPU).

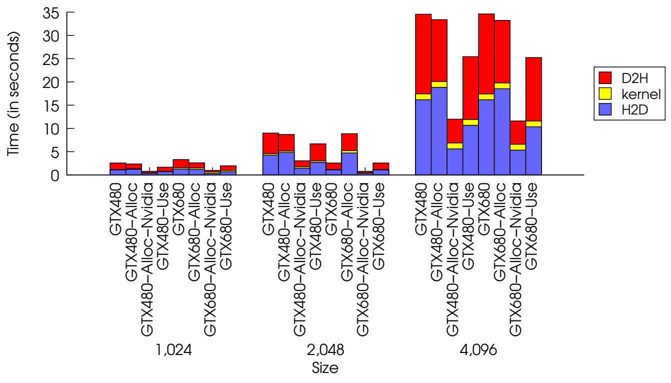


Figure 6.4: Performance of memory transfers GPU.

To summarize, we have empirically evaluated the ways in which the cost of data transfer can be minimized. A common technique for minimizing data transfer is to enable the so-called zero-copy. Each platform (hardware *and* software) has its own specific mechanisms for enabling zero-copy, which can lead to performance improvement. The best such mechanism for each platform will be used in our OpenCL PaSWAS implementation.

CHAPTER 7

Experiments and Results

In this chapter we describe the results of our empirical evaluation of the OpenCL PaSWAS as described in chapter 5. Our goal is to report and rank the performance of the different types of algorithms we have implemented, and analyze the causes of the observed behavior.

7.1 Experimental Setup

The experiments were executed on the wide-area distributed system, called DAS-4 [Bal+00], a multi-cluster build by a consortium of research institutions in the Netherlands. The goal of DAS-4 is to provide a common computational infrastructure for researchers. The system has support for various programming models, including OpenCL and CUDA, and it allows us to test different platforms (GPUs and multi-CPU systems). In addition, DAS-4 allows for isolation of tasks, avoiding situations where machines would be running different tasks concurrently, thus corrupting the performance measurements.

7.1.1 Hardware Platforms

Throughout the experiments, we utilize three hardware platforms, namely two NVIDIA GPUs and one multi-core CPU. The characteristics of these devices are listed in 7.1. The GPUs belong to different NVIDIA architecture families: GTX480 has a Fermi architecture, which is the predecesor of the Kepler architecture from GTX680. There are several important differences between these architectures - e.g., the amount of processing unit within a streaming multiprocessor has increased from 32 on Fermi to 192 on Kepler; similarly, the memory sizes (for both the main memory and the caches) have been increased, and memory bandwidth is also higher. Although, the GTX480 has more streaming multiprocessors, its total number of processing units - i.e., threads or CUDA cores is lower ($32 \times 15 = 480$) than those in GTX680 ($192 \times 8 = 1536$). Note that both the GTX680 and GTX480 use an Intel Xeon E5620 as a host processor.

Table 7.1: The hardware platforms used to evaluate PaSWAS.

Type	Name	Clock Frequency	Memory size	Memory bandwidth	Multi processors
CPU	Intel Xeon E5620	2458 MHz	24GB	25.6 GB/s	16
GPU	NVIDIA GTX480	700MHz	1.5GB	144 GB/s	15
GPU	NVIDIA GTX680	1006MHz	2GB	192.2 GB/s	8

7.1.2 Datasets

We consulted existing literature to retrieve credible datasets which are commonly used with sequence alignment [Hai+11],[Man+08] [Liu+13]. Each dataset is characterized by two sets of sequences, namely *the sequence* and *the target* databases. Table 7.2 lists the details of each dataset. The columns $|S|$, $|s_{avg}|$ and $|s_0|$ represent the characteristics of the sequence database. These are given by number of sequences ($|S|$), average sequence length ($|s_{avg}|$) and the largest sequence length (s_0) . With the target database these parameters are denoted by $|T|$, $|t_{avg}|$ and $|t_0|$. DS3 to DS6 utilize protein databases (retrieved from `http://www.ensembl.org/info/data/ftp/index.html`) from rats, dogs, mice and humans, respectively. These are aligned to predefined protein sequences. The remaining two datasets were gathered from `https://github.com/swarris/PaSWAS/tree/master/PaSWAS/data` and were used in the past for the evaluation of the PaSWAS CUDA implementation.

Table 7.2: Datasets used for the evaluation of PaSWAS.

Dataset	$\|S\|$	$\|s_{avg}\|$	$\|s_0\|$	$\|T\|$	$\|t_{avg}\|$	$\|t_0\|$
DS1	401824	292	446	4	20	21
DS2	55295	410	706	7	69	76
DS3	26153	508	7358	34	1359	5478
DS4	25160	577	35180	34	1359	5478
DS5	52998	448	35213	34	1359	5478
DS6	99436	374	35991	24	1801	5478

Table 7.3: Abbreviations used in figures 7.1 to 7.12.

Abbreviation	Meaning
GLOBAL_0C	Only utilizing global memory with zero-copy memory transfers
GLOBAL_N0C	Only utilizing global memory without zero-copy memory transfers
SHARED_0C	Global+local memory with zero-copy memory transfers
SHARED_N0C	Global+local memory without zero-copy memory transfers
GLOBAL_CG	Only utilizing global memory with increased granularity and with zero-copy memory transfers

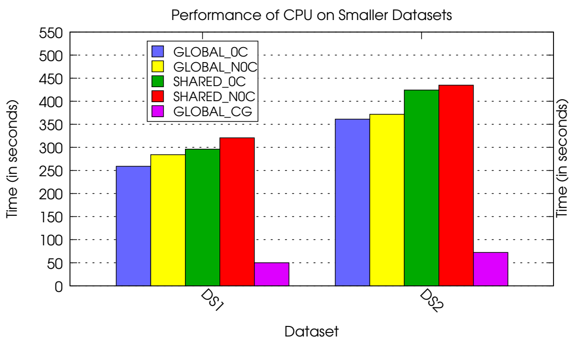
7.2 Experiments

Our evauation focuses on measuring the performance of the implementations described in sections 5.1, 5.2 and 5.2.1. Thus, experiments are executed on the local memory, only global memory and coarse grained granularity implementations of PaSWAS. Additionally we discuss the effects of zero-copy memory transfers on each hardware platform and, for each platform, we use the best-performing scheme, deduced from the analysis performed in the section 6. Table 7.3 presents the abbreviations used along this section and throughout figures 7.1 to 7.12.

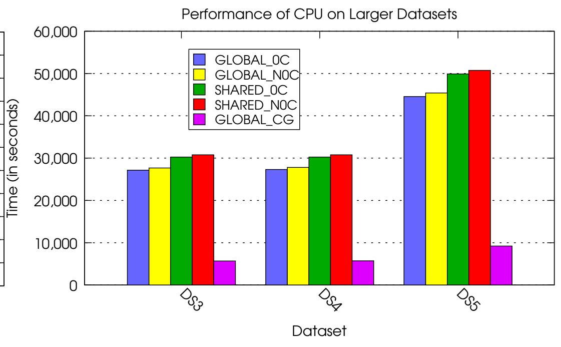
We experiment on work-group sizes ranging from 8×8 to 16×16. We only consider square work-groups, since it can be formally proven that this configuration yields the most parallelism (see Appendix A for the proof). The datasets are statically partitioned according to the algorithm described in section 8.2, using a threshold value of 1024 (empirically chosen to deliver fairly balanced sub-datasets). The overall performance measure for a single dataset is retrieved by adding the execution time of each resulting sub-dataset.

Table 7.4: Data characteristics of first 5 runs.

Run	$\|s\|$	$\|t\|$	$Concurrency$	$Speed-up$
0	35992	5480	685	4.48
1	35992	4064	508	3.04
2	35992	3008	376	2.89
3	35992	1504	188	2.78
4	35992	464	58	1.74

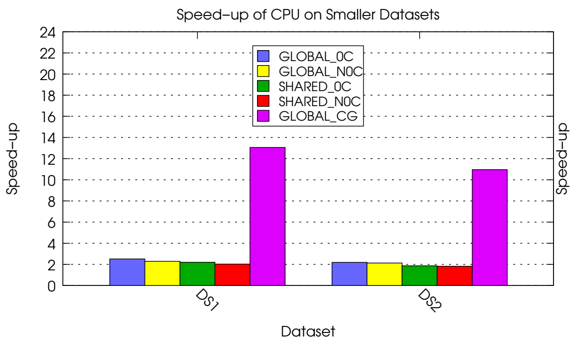


(a) DS1 & DS2

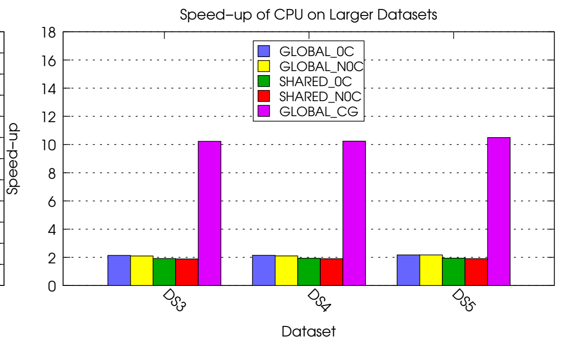


(b) DS3, DS4 & DS5

Figure 7.1: CPU performance of DS1,DS2 DS3, DS4 & DS5 with 8×8 work-group configuration (note the difference in scale).

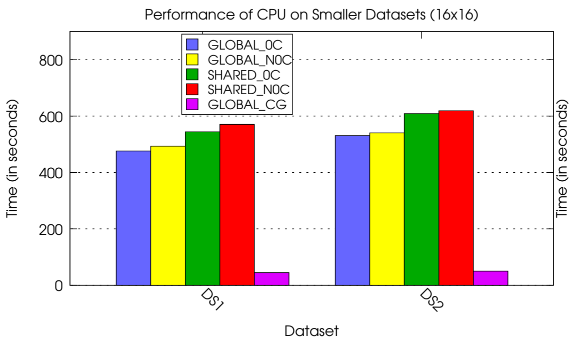


(a) DS1 & DS2

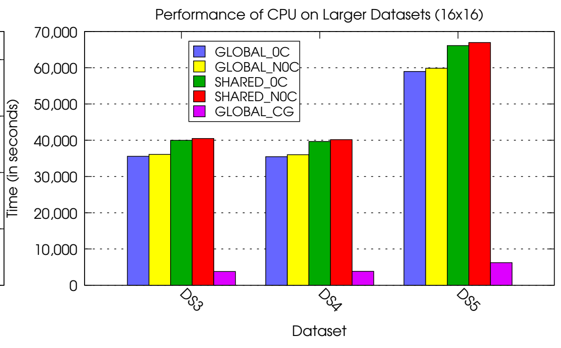


(b) DS3, DS4 & DS5

Figure 7.2: CPU speed-up of DS1,DS2 DS3, DS4 & DS5 with 8×8 work-group configuration (note the difference in scale)..

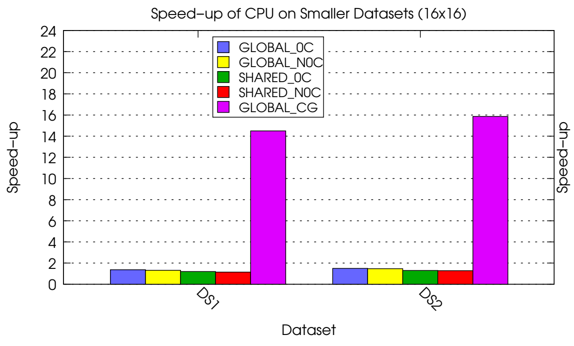


(a) DS1 & DS2

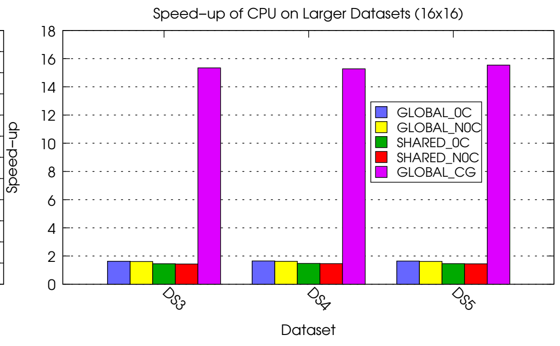


(b) DS3, DS4 & DS5

Figure 7.3: CPU performance of DS1,DS2 DS3, DS4 & DS5 with 16×16 work-group configuration (note the difference in scale)..



(a) DS1 & DS2



(b) DS3, DS4 & DS5

Figure 7.4: CPU speed-up of DS1,DS2 DS3, DS4 & DS5 with 16×16 work-group configuration (note the difference in scale).

7.2.1 CPU

For the CPU, we evaluate all configurations described in table 7.3. The results are presented in figures 7.1 to 7.4. Figures 7.1 and 7.3 represent the execution time of the PaSWAS algorithm on 5 datasets (DS1 to DS5), with work-group sizes of 8×8 and 16×16. We run each configuration three times to remove any noise that can be present in our performance data[1]

From figures 7.1 and 7.3 we can construct a performance ranking, as follows:

$$\text{GLOBAL\_CG} > \text{GLOBAL\_0C} > \text{GLOBAL\_N0C} > \text{SHARED\_0C} > \text{SHARED\_N0C} \quad (7.1)$$

The relation of $X > Y$ in equation 7.1 is defined as X outperforming Y, in terms of execution time. We make three observations.

First, the implementations that use global memory only are outperforming the ones using local memory for the datasets $DS1$ to $DS5$. This behavior is somewhat expected, because memory objects created through OpenCL are cached by the hardware. Explicit caching through local memory will perform worst than hardware caching, and will also introduce an unnecessary overhead[Int11].

Second, we see that zero-copy memory transfers is preferred with respect to "normal" memory transfers, i.e., GLOBAL_0C > GLOBAL_N0C. This is also expected behavior, since both the host and the device are the same hardware platform, namely the CPU. No additional copy of data is needed from host to device, as the memory is directly accessible. This is exactly what zero-copy enables.

Third, the best performing implementation is GLOBAL_CG, where the workload of a single work-item is increased. From figure 7.2, we can see that increasing the workload

1 We omit the error bars because we observe that the results are fairly stable.

has a positive effect on the CPU. This effect is further enhanced when moving to a work-group of size 16×16, see figure 7.4. In both these figures the speed-up is plotted relative to the sequential implementation SeqSWAS described in section 4. The increase in performance can be attributed to the way in which a work-item processes the elements given to it. These elements are computed in a row-major order, which is a CPU-friendly way of data accessing. As a result we preserve cache locality within each work-item[J S+13], which in turn leads to performance improvement. When we have a work-group size of 16×16 each work-item computes 8×8 elements. We believe that the cache was still not fully saturated in this case, allowing for faster access of more elements compared to the case of when we have a work-group size of 8×8. The opposite is true for the other four configurations, where we experience an increase of approximately 10000 seconds for each configuration other than GLOBAL_CG.

We will now take a closer look at the largest dataset (DS6) and explain how the speed-up is achieved. For this dataset, the sequence file is divided into 13 files. The target file is partitioned into 5 files. Thus we have in total $65 = 13 * 5$ runs. Figure 7.5 displays the speed-up achieved for each run [1].

The number of work-groups that can be executed concurrently is dependent on the size of the smallest sequence that we want to align. When sequences of similar length are aligned, the amount of work-groups that can be executed concurrently is at its maximum and, consequently, we have an increase in speed-up. However, if a sequence is significantly smaller compared to the others, this will yield a lower speed-up since less work-groups will be able to run concurrently. This is exemplified in table 7.4, which

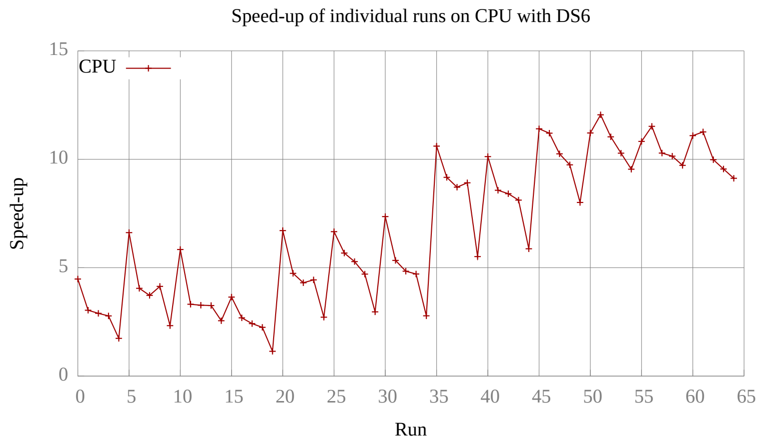


Figure 7.5: Speed-up of individual runs on GLOBAL_CG with DS6.

1 A run is defined as an independent execution of the PaSWAS algorithm with a given parameter set. A parameter set contains among others, the length of the largest sequence and the length of the largest target for a specific dataset.

shows the achieved speed-up, for different pairs of sequence lengths ($|s|$ and $|t|$) for the first 5 runs. In addition, the amount of work-groups that can be processed concurrently is given by *Concurrency*. We can see that run 0 provides a much better speed-up than run 4. This is because $|s|$ is fixed, and $|t|$ decreases between subsequent runs. It follows that *Concurrency* will also then decrease between subsequent runs. In general, when there is a big difference between $|s|$ and $|t|$, the speed-up is negatively impacted.

We can see from figure 7.5, in general, that when the runs are increasing in size, the speed-up is also increasing, because the sequences tend to get smaller in length but more in number. Due to this, the sequential implementation SeqSWAS needs to perform more iterations, while PaSWAS can do more in parallel. However, when the opposite is true, that is, we either need to compute a low number of sequences or we have little parallelism, then the overhead caused by using OpenCL reflects in a lower speed-up. These two observations, are valid in general and are not limited by a specific hardware platform (as will be seen in the subsequent sections).

7.2.2 GPU: GTX480

As with the CPU, we can construct a performance hierarchy for the GTX480 using the data presented in figures 7.6 to 7.9. This ranking is:

$$\text{SHARED\_0C} > \text{SHARED\_N0C} > \text{GLOBAL\_0C} > \text{GLOBAL\_N0C} \tag{7.2}$$

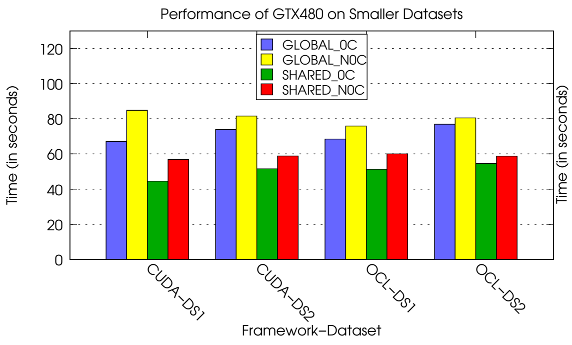
We make the following observations.

First, compared to the CPU, the positions of the implementations using global and local memory are interchanged. This happens as local memory is mapped to a physical, fast on-chip memory on GPUs. This allows for faster access times and thus better performance [Kom+10]. To achieve this, the programmer needs to explicitly move elements from global memory to local memory. There might be cases where the overhead of transferring data from global memory to local memory outweigh the benefits of faster access times - for exampe, when data is not accessed very often. However, by looking at the figures 7.6 to 7.9, this does not happen.

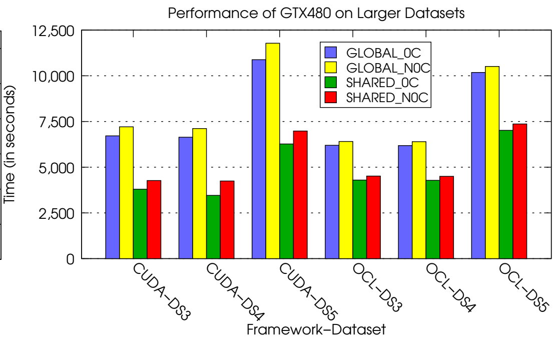
Second, from the same figures 7.6 to 7.9, we conclude that the best implementation of CUDA (SHARED_0C) outperforms OpenCL's best implementation (SHARED_-0C). This behavior appears because (1) the CUDA compiler makes more aggresive optimizations (as observed by our direct PTX code analysis), (2) CUDA uses predefined, optimized operators (e.g., max(a,b)), and (3) OpenCL kernel launches are more expensive. This was established by a simple benchmarking experiments, where we ran empty kernels (i.e., with no code in the body of the kernel) and measured the kernel execution time. The results are presented in tables 7.5 and 7.6, which show the empty kernel execution times for both CUDA and OpenCL, as well as a the ration $\frac{OpenCL}{CUDA}$, to quintify how much worse OpenCL's kernel launch overhead is compared to CUDA. We can clearly see that OpenCL imposes a 50-60% additional overhead for launching a single kernel.

Since PaSWAS launches its kernel iteratively, i.e., it utilizes multiple kernel launches to solve the alignment problem, this extra overhead will have an immediate impact on the performance of our OpenCL implementation, lowering it visibly.

Third, figure 7.10 displays the speed-up achieved for every run on DS6. Note that there are more runs in figure 7.5, because not every alignment could be processed by the GPU, due to memory constraints. The figure illustrates a relatively constant peformance gap between the CUDA and OpenCL performance, in the favor of the former. Also, we point out that figures 7.10 and 7.5 show the same trends, whie means that our observation that a larger number of small sequences is more beneficial to parallelism (and, therefore, performance) than a few long sequences. The impact of this observation on the overall application will be further explored in Chapter 8.

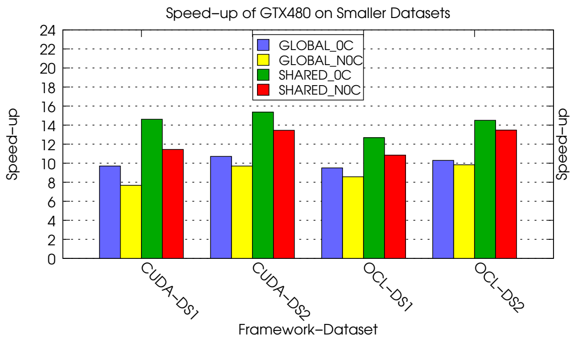


(a) DS1 & DS2

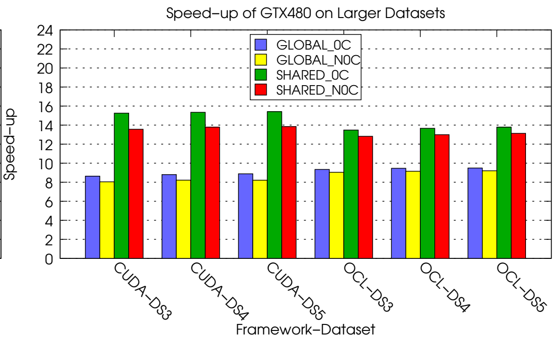


(b) DS3, DS4 & DS5

Figure 7.6: GTX480 performance of DS1,DS2 DS3, DS4 & DS5 with 8 × 8 work-group configuration (note the difference in scale).



(a) DS1 & DS2

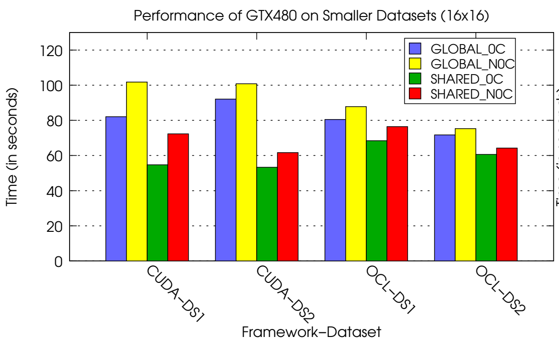


(b) DS3, DS4 & DS5

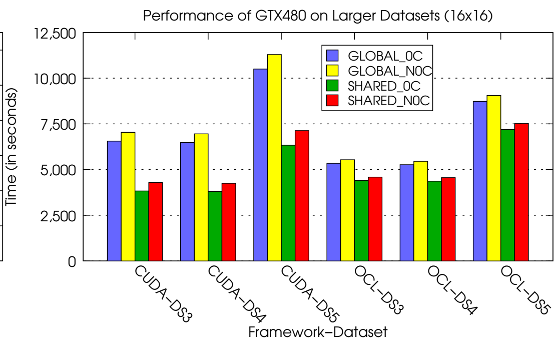
Figure 7.7: GTX480 speed-up of DS1,DS2 DS3, DS4 & DS5 with 8 × 8 work-group configuration (note the difference in scale).

7.2.3 GPU: GTX680

Finally, we report the performance results we have observed when using he newer GPU architecture from NVIDIA, namely the Kepler-based card GTX680. The experiments

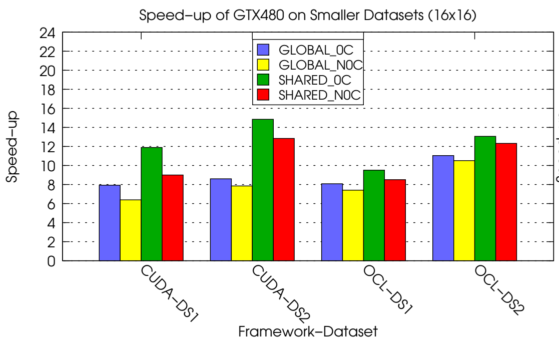


(a) DS1 & DS2

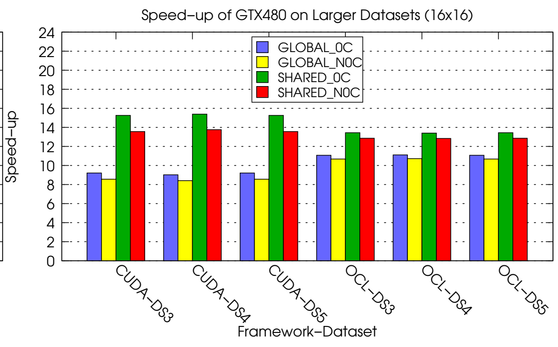


(b) DS3, DS4 & DS5

Figure 7.8: GTX480 performance of DS1,DS2 DS3, DS4 & DS5 with 16 × 16 work-group configuration (note the difference in scale).



(a) DS1 & DS2



(b) DS3, DS4 & DS5

Figure 7.9: GTX480 of DS1,DS2 DS3, DS4 & DS5 with 16 × 16 work-group configuration (note the difference in scale).

Table 7.5: Empty kernel time of first 5 runs DS4 on Fermi GPU.

$CUDA$	$OpenCL$	$\frac{OpenCL}{CUDA}$
0.251212	0.39641	1.577989905
0.143879	0.227866	1.583733554
0.477382	0.75581	1.583239418
0.190479	0.299574	1.572740302
0.162582	0.252844	1.55517831

we ran are similar to those performed on the GTX680 GPU. The results are shown in figures 7.11 to 7.14.

For the first time, we notice a different ranking for the CUDA and OpenCL versions.

Table 7.6: Empty kernel time of first 5 runs DS4 on Kepler GPU.

$CUDA$	$OpenCL$	$\frac{OpenCL}{CUDA}$
0.290487	0.455344	1.567519373
0.167555	0.271146	1.618250724
0.562043	0.897534	1.596913403
0.224472	0.354497	1.579248191
0.19206	0.299833	1.561142351

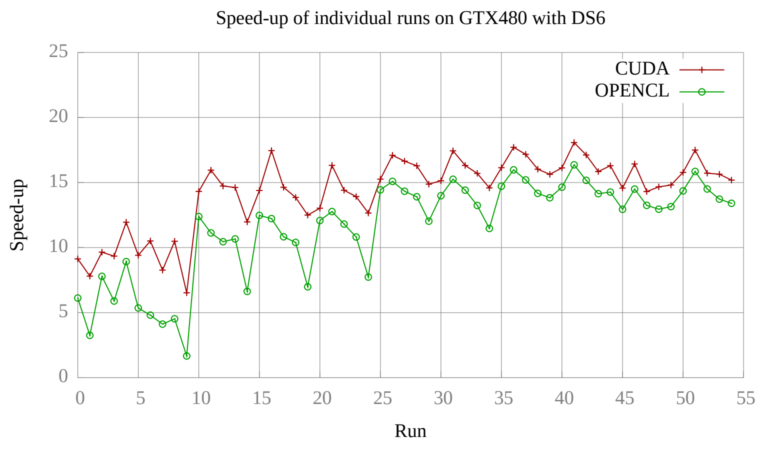


Figure 7.10: Speed-up of individual runs on SHARED_0C with DS6 for GTX480 on both CUDA and OpenCL.

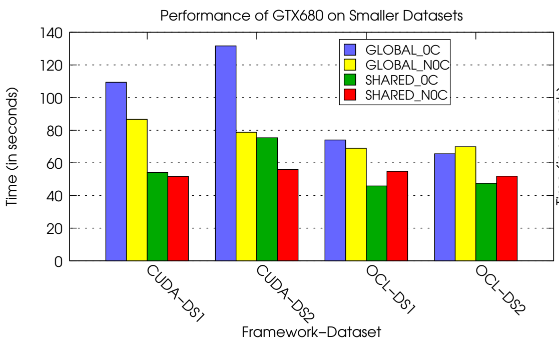
For CUDA, in general:

$$\text{SHARED\_N0C} > \text{SHARED\_0C} > \text{GLOBAL\_N0C} > \text{GLOBAL\_0C} \tag{7.3}$$

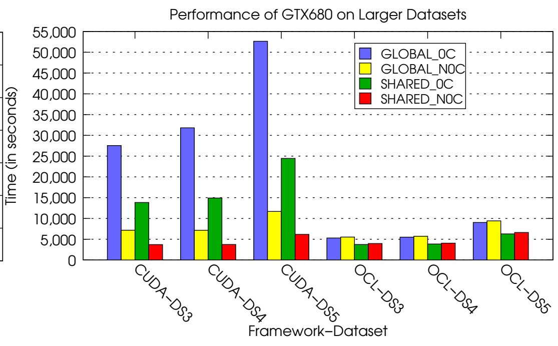
For OpenCL, in general:

$$\text{SHARED\_0C} > \text{SHARED\_N0C} > \text{GLOBAL\_0C} > \text{GLOBAL\_N0C} \tag{7.4}$$

Note however that, for OpenCL, the differences between zero-copy and non zero-copy tend to be quite small. Also note that the significant outliers in these pictures are the numbers measured for CUDA with zero-copy enabled. In other words, very surprisingly, the CUDA implementation does not react well to zero-copy memory transfers for this card. We have no clear explanation of this strange behaviour. We have ruled out defect of the card and/or the usage of the wrong compiler or CUDA version. We have also ruled out the possibility that the unified address space is the cause for this. This new CUDA functionality simplifies the process of enabling zero-copy, but we have noticed the same behavior for older CUDA versions. We are further investigating the issue by building a parameterized memory microbenchmark that can analyze this behavior systematically.

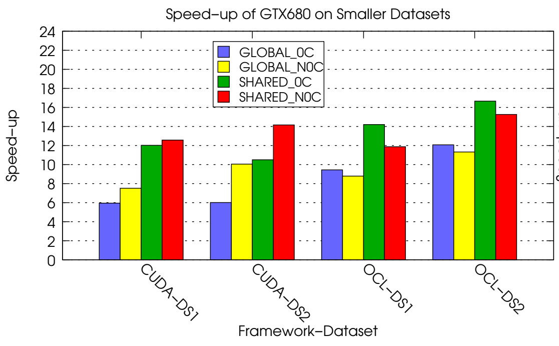


(a) DS1 & DS2

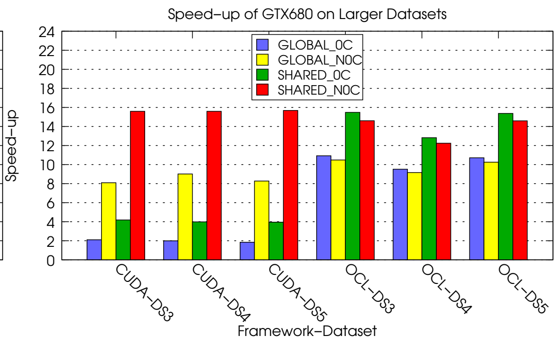


(b) DS3, DS4 & DS5

Figure 7.11: GTX680 performance of DS1,DS2 DS3, DS4 & DS5 with 8×8 work-group configuration (note the difference in scale).

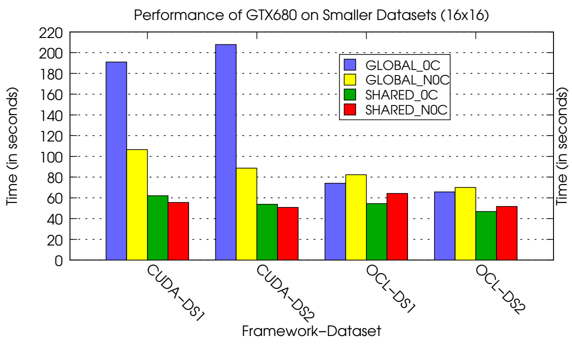


(a) DS1 & DS2

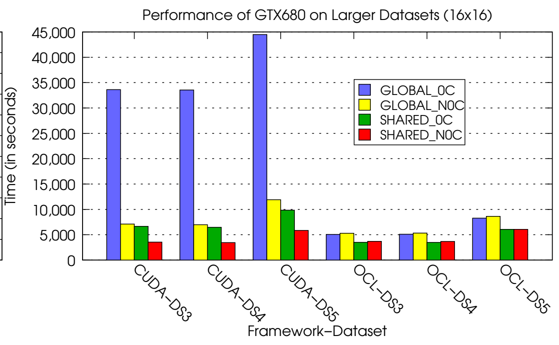


(b) DS3, DS4 & DS5

Figure 7.12: GTX680 speed-up of DS1,DS2 DS3, DS4 & DS5 with 8×8 work-group configuration (note the difference in scale).

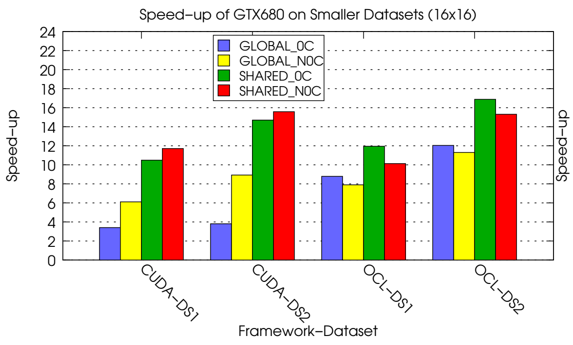


(a) DS1 & DS2

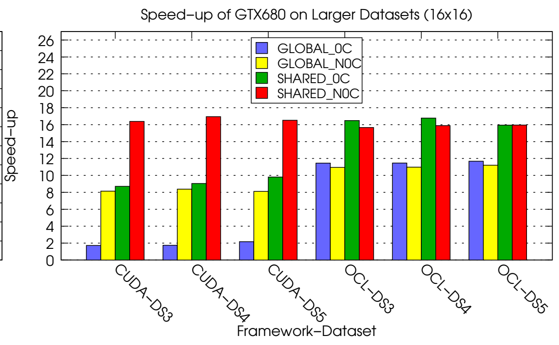


(b) DS3, DS4 & DS5

Figure 7.13: GTX680 performance of DS1,DS2 DS3, DS4 & DS5 with 16×16 work-group configuration (note the difference in scale).



(a) DS1 & DS2



(b) DS3, DS4 & DS5

Figure 7.14: GTX680 of DS1,DS2 DS3, DS4 & DS5 with 16×16 work-group configuration (note the difference in scale).

CHAPTER 8

Partitioning

In this chapter we will describe three partitioning algorithms, which we have created to investigate whether the performance can be increased when we partition a dataset. We will first start by giving a proper definition of the problem and the need for a partitioning algorithm. After the problem has been made clear, the three algorithms will be discussed more indepthly. We conclude this section, by empirically evaluating the three partitioning algorithms.

8.1 Problem Description

In PaSWAS, intra-kernel parallelization is enabled by computing multiple alignments in parallel. Commonly, we have a set of sequences S=$\{s_0 \ldots s_{n-1}\}$ that we want to align to a set of targets T=$\{t_0 \ldots t_{n-1}\}$, that is:

$$\forall_{0 \leq i < n, 0 \leq j < m} \, a_{ij}$$

.

Here, a_{ij} signifies the alignment between s_i and t_j. Furthermore, these sets are sorted such that $|s_0| \geq \ldots \geq |s_{n-1}|$ and $|t_0| \geq \ldots \geq |t_{m-1}|$ where $|x|$ indicates the length of sequence x.

Currently, the number of alignments that can be computed in parallel is determined by s_0 and t_0 (e.g. the largest sequence and the largest target). Using $|s_0|$, the remaining sequences are padded to this maximum length. This is also performed for the targets, which are padded to the length of t_0. Moreover, the number of alignments that can be computed in parallel is strictly limited by the memory capacity of a device (besides $|s_0|$ and $|t_0|$). That is, if we have 2MB of memory available, and we need 1MB to compute the alignment a_{00}, we could evaluate at most 2 alignments in parallel.

When we have sequences of similar length (e.g. $|s_0| \approx \ldots \approx |s_{n-1}|$ and $|t_0| \approx \ldots \approx |t_{m-1}|$), this scheme provides a good approximation of the degree of intra-kernel parallelism. However, when we have $|s_0| >> |s_1| \approx \ldots \approx |s_{n-1}|$ and/or $|t_0| >> |t_1| \approx \ldots \approx |t_{m-1}|$, basing the amount of parallelism on the largest sequence and the largest target will yield sub-optimal results, because we excessively pad much shorter sequences and/or targets and perform a lot of needless computations. This solution limits the amount of parallelism, which limits the potential performance of the overall application.

Let's illustrate this trade-off with an example: let us assume we have S=$\{s_0,s_1\}$ and T=$\{t_0,t_1\}$ such that the set of alignments A is given by A=$\{a_{00},a_{01},a_{10},a_{11}\}$. We have 2 sequences and 2 targets, and we need to conduct 4 alignments. We further assume that to compute the alignment a_{00} we need 1 MB, and for all the remaining alignments we need 0.3 MB. On a device with 1 MB of memory, basing all the alignments on a_{00} would yield no intra-kernel parallelism, since each alignment needs to be evaluated sequentially, thus we need 4 iterations to perform all the alignments on the device. However, when we partition the data into two sets - $A_1 = \{a_{00}\}$ and $A_2 = A - A_1 = \{a_{01},a_{10},a_{11}\}$, set A_2 can be evaluated in one iteration, by evaluating all its three alignments in parallel.

To enable such a scheme, which yields higher parallelism, we need (1) a partitioning algorithm to determine the best number of sets and their sizes, and (2) separate executions of the algorithm, one for each set (i.e., one for A_1 and one for A_2 in the previous example). To this end, we discuss three algorithms designed and implemented for this smart partitioning.

8.2 Static Partitioning

The most simple solution to partition a dataset is by defining a threshold and partitioning the sets S and T based on this threshold. That is, if $|s_0| - |s_1| > threshold$ then two separate sets are created: one for sequence s_0, and another one (S_1) for all the remaining sequences. The set S_1 is then further partitioned using the rule described above, only the condition is changed to $|s_1| - |s_2| > threshold$. The same algorithm is applied for T. The implementation of this algorithm is described in Algorithm 10.

```
**Input**: Set X, threshold
**Output**: Partitioning = Set of sets = $X_{sets}$
$x_{prev}$:= $x_0$;
sets:= 1;
**for** $i = 0 \dots |X| - 1$ **do**
    **if** $x_{prev} - |x_i| > threshold$ **then**
        sets++;
        Create new set $X_{sets}$;
    **end**
    add $x_i$ to $X_{sets}$;
**end**
```

Algorithm 10: Algorithm for static partitioning of a set X

After both sets S and T are partitioned by Algorithm 10, the sequences in each subset of S are aligned to the targets of each subset of T.

The theoretical upper bound of this algorithm is given by $\mathcal{O}(|S| + |T|)$. The algorithm does not utilize any knowledge about the distribution of the data and can thus be considered simplistic. The next two algorithms attempt more intelligent decisions by using more analysis.

8.3 Dynamic Partitioning

In this section, we describe two algorithms which attempt to maximize the number of alignments that can be computed in parallel. This metric determines how the sets S and T should be partitioned. An approximation of the number of alignments that can be conducted on a device is given by:

$$P_{ij} = \frac{Memory}{mem(a_{ij})} \tag{8.1}$$

In equation 8.1, mem(a_{ij}) is the amount of memory needed for a single alignment of s_i with t_j. Thus, P_{ij} expresses how many alignments we can process in parallel with s_i and t_j, as the largest sequence and target, given a certain memory capacity, ($Memory$). For example, for $P_{00} = 4$, we can evaluate 4 alignments using s_0 and t_0 as a reference. This leads us to the following three possible configurations:

1. Align 4 sequences to 1 target
2. Align 2 sequences to 2 targets
3. Align 1 sequence to 4 targets

All these configurations could be executed in parallel, but they do not lead to the same performance. If the set T only contains 1 target then using options 2 and 3 would be a waste of resources. When $|T| \geq 4$ and $|S| \geq 4$ all three options are viable and choosing a configuration is not that trivial. Algorithm 11 provides the means to determine the best configuration given a certain sequence s_k and a target t_l

The variables $iter_s$ and $iter_t$ represent the number of iterations needed to process $|S| - k$ sequences and $|T| - l$ targets. From Algorithm 11, we can observe that the best configuration is the one which minimizes the total number of iterations ($iter_s \times iter_t$) given $|s_k|$ and $|t_l|$. In other words, we are maximizing the number of alignments that can be evaluated in parallel ($par_s \times par_t$). Note that the memory usage depends on five factors namely:

- The length of the sequence s_i.
- The length of the target t_j.
- The number of sequences that can be processed in parallel par_s.
- The number of targets that can be processed in parallel par_t.
- The number of iterations needed to process all sequences $iter_s = \lceil \frac{|S|}{par_s} \rceil$.

```
**Input**: Set S, Set T, sequence $s_k$, target $t_l$, Memory
**Output**: configuration($par_s$,$par_t$,$s_s$,$s_t$)
$iter_s$:= $|S| - k$;
$iter_t$:= $|T| - l$;
$iter_{current}$ := $iter_s \times iter_t$+1;
for $i = 1 \dots |S| - k$ do
    $iter_s$:=$\lceil \frac{|S|-k}{i} \rceil$;
    for $j = 1 \dots |T| - l$ do
        $iter_t$:=$\lceil \frac{|T|-l}{j} \rceil$;
        Determine memory usage ($|s_k|$,$|t_l|$,i,j,$s_s$);
        if $memory\ usage \leq Memory\ \&\ iter_s \times iter_t < iter_{current}$ then
            $par_s$ := i;
            $par_t$ := j;
            Found a possible configuration ($par_s$,$par_t$,$iter_s$,$iter_t$);
            $s_{current}$ := $iter_s \times iter_t$ ;
        end
    end
end
Return configuration($par_s$,$par_t$,$s_s$,$s_t$);
```

Algorithm 11: Algorithm for determining best configuration

Thus, the three possible configurations listed above (for $P_{00} = 4$) can have different memory needs and P_{ij} provides an approximation for each of these configurations. Determining a configuration based on 11 is performed with a time complexity $\mathcal{O}(|S||T|)$

8.3.1 Partitioning Algorithm 1

Our goal is to partition both sets S and T such that we enable the most parallelism, based on equation 8.1. A first algorithm for this task is presented in Algorithm 12.

The basic idea of this algorithm is to compute as much sequences in parallel for a certain alignment a_{i0} $(0 \leq i < |S|)$, then determining whether we can improve the parallelism when moving to a smaller target t_j $(0 < j < |T|)$. The size of the list *Runs* determines how many separate runs of PaSWAS we need to evaluate a whole dataset. A *Run* contains all the parameters needed to invoke a single run of PaSWAS.

We add a *Run* to the *Runs* list if and only if it increases the parallelism. When this is not the case, the number of iterations of the previous *Run* is increased by 1, so either $iter_s$ or $iter_t$ is increased by 1. A threshold based version of this algorithm has also been implemented in which we enforce that the amount of parallelism should at least increase by $(1 + \alpha)$, before it is registered as a separate run.

The time complexity of this algorithm is $\mathcal{O}(|S|^2|T|^2)$. Although it has a significantly larger complexity (compared with the complexity of the static partitioning, $\mathcal{O}(|S|+|T|)$), this algorithm produces more balanced partitions.

```
Input: Set S, Set T, Memory
Output: Partitioning = Set of sets = $Runs$
$i$:= 0;
$j$:= 0;
$P$:= 0;
$Runs$ := $\emptyset$;
while $i < |S|$ do
    Calculate $P_{i0}$ if $P_{i0} \leq 0$ then
        Alignment cannot be done;
        $i := i + 1$
    else
        if $P_{i0} > P$ then
            $P := P_{i0}$;
            Determine configuration according to Algorithm 11;
            Add (i,0,$par_s$,$par_t$,1,1) to $Runs$;
            Iterate over T starting from j:=$par_t$ and add them to $Runs$ IFF we
            gain enough parallelism;
            i := i + $par_s$;
        else
            We did not gain enough parallelism when moving to a smaller
            sequence;
            Increase $s_{iter}$ by 1 in previous run;
            Use $par_s$ of previous run;
            i := i + $par_s$;
        end
    end
end
Return $Runs$;
```

Algorithm 12: First (dynamic) partitioning algorithm.

8.3.2 Partitioning Algorithm 2

The second partitioning algorithm is inspired by the Set Cover Problem (SCP) [Sla96].

This is an optimization problem known to be NP-hard [Gar+79]. Within the computational complexity theory, problems of this complexity class are regarded as unfeasible. This is due to their exponential theoretical running time to produce an optimal solution. Heuristics [Kor+00] are often used to enable a polynomial running time in the expense of an exact optimal solution (e.g. approximated).

The SCP problem definition follows. Let's assume an universe $U = \{u_1, u_2, \ldots, u_n\}$ and a set of sets $H = \{H_1, H_2, \ldots, H_m\}$. For each set H_i we have $H_i \subseteq U$. For this problem, we want to find *a minimal cover $C \subseteq H$ such that the union of each set $H_i \in C$ is U.*

The mapping between SCP and the problem of partitioning the sets S and T is achieved by perceiving the universe U as a collection of all alignments: $U = \{a_{00}, \dots, a_{|S|-1|T|-1}\}$ and $|U| = |S||T|$. We construct a set H_{ij} for each alignment, thus $|H| = |S||T|$. The set H_{ij} contains the following elements of the universe:

$$H_{ij} = \{a_{kl} \mid k \geq i \textit{ and } l \geq j\} \tag{8.2}$$

The resulting optimization problem then becomes choosing a minimal cover such that we maximize $\sum P_{ij}$.

Note that H_{00} contains all alignments. However, P_{00} might be very low for this set. Figure 8.1 presents the structure of the matrix H. From this figure we can also deduce the relationship between different sets. For an arbitrary H_{ij}, all elements below H_{ij} and to the right of H_{ij} are contained in this set. The sets which are contained in H_{11} are highlighted in red in figure 8.1.

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} & \cdots & H_{0|T|-1} \\ H_{10} & H_{11} & H_{12} & H_{13} & \cdots & H_{1|T|-1} \\ H_{20} & H_{21} & H_{22} & H_{23} & \cdots & H_{2|T|-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ H_{(|S|-1)0} & H_{(|S|-1)1} & H_{(|S|-1)2} & H_{(|S|-1)3} & \cdots & H_{|S|-1|T|-1} \end{bmatrix}$$

Figure 8.1: Structure of the set matrix H and the depiction of the sets which are contained in H_{11} are highlighted in red

In our implementation of this partitioning algorithm, each of $H_{ij} = |\{a_{kl} \mid k \geq i \textit{ and } l \geq j\}|$ and $U = |U|$

That is, H_{ij} denotes the number of alignments that can be processed using s_i and t_j as the largest sequence and target. Also, the universe is set to the total amount of alignments that need to be processed. In addition, we introduce a matrix P (figure 8.2). Each P_{ij} is calculated according to equation 8.1 and provides an approximation of the amount of alignments we can evaluate in parallel when again using s_i and t_j as the largest sequence and target. It is possible that $P_{ij} > H_{ij}$, in this case P_{ij} is adjusted such that $P_{ij} = H_{ij}$.

Algorithm 13 provides an overview of the the second partitioning algorithm.

Finding a tuple (k,l) in the matrix P is achieved by finding the maximum element in this matrix. However, there is a constraint in how this tuple is chosen. If the previously chosen tuple is (i,j) then all tuples which are characterized by $(k < i, l < j)$ are excluded from the search. This restriction originates from the PaSWAS implementation. Take the situation where tuple (1,1) is processed by the PaSWAS algorithm, which means that all the alignments in H_{11} have already been performed. If we then choose tuple (0,0), with set H_{00} which contains all the sequences and targets then PaSWAS will again

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} & \dots & P_{0|T|-1} \\ P_{10} & P_{11} & P_{12} & P_{13} & \dots & P_{1|T|-1} \\ P_{20} & P_{21} & P_{22} & P_{23} & \dots & P_{2|T|-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ P_{(|S|-1)0} & P_{(|S|-1)1} & P_{(|S|-1)2} & P_{(|S|-1)3} & \dots & P_{|S|-1|T|-1} \end{bmatrix}$$

Figure 8.2: Structure of the set matrix P

```
**Input**: Set S, Set T, Memory
**Output**: Partitioning = Set of sets = $Runs$
$U$:= $|S| \times |T|$;
$Runs$ := $\emptyset$;
Initialize matrix $H$ and $P$;
while $U > 0$ do
    Find ($i$,$j$) in $P$ with maximum parallelism;
    $U$ := $U$ - $H_{ij}$;
    Update matrix $H$ and $P$;
    Determine configuration according to Algorithm 11;
    Add (i,j,$par_s$,$par_t$,$s_s$,$s_t$) to $Runs$;
end
Return Runs;
```

Algorithm 13: Second (dynamic) partitioning algorithm

perform the alignments which were already executed with H_{11}. Since in its current form, PaSWAS will align every sequence to every target. However when we first choose tuple (1,0) then the resulting H matrix is given by figure 8.4. In this case we can choose H_{00}, since we need to compare s_0 to every target. These are precisely the alignments that have not yet been conducted.

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} & \dots & H_{0|T|-1} \\ H_{10} & & & & & \\ H_{20} & & & & & \\ \vdots & & & & & \\ H_{(|S|-1)0} & & & & & \end{bmatrix}$$

Figure 8.3: Structure of the set matrix H after the tuple (1,1) is chosen

Given a tuple (i,j) the update routine updates the matrices H and P. That is, for all

$$H = \begin{bmatrix} H_{00} & H_{01} & H_{02} & H_{03} & \dots & H_{0|T|-1} \end{bmatrix}$$

Figure 8.4: Structure of the set matrix H after the tuple (1,1) is chosen

k and l:

$$H_{kl} := H_{kl} - (H_{kl} \cap H_{ij}) := H_{kl} - H_{max(k,i)max(j,l)} \tag{8.3}$$

For example if $(i,j) = (1,0)$, the set H_{01} will become:

$$H_{01} := H_{01} - (H_{0,1} \cap H_{1,0}) := H_{01} - H_{11} \tag{8.4}$$

Again, if there are H_{kl} for which $P_{kl} > H_{kl}$ then P_{kl} is updated such that $P_{kl} = H_{kl}$. Algorithm 13 runs in $\mathcal{O}(|S|^2|T|^2)$, since, in the worst case, the **while** loop is executed $\mathcal{O}(|S||T|)$ times. The routines "update", "find maximum", and "determine configuration" run in $\mathcal{O}(|S||T|)$ time. Thus the overall worst case running time is given by $\mathcal{O}(|S|^2|T|^2)$.

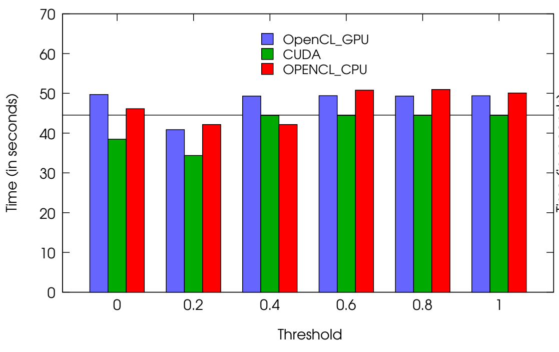
8.4 Empirical evaluation

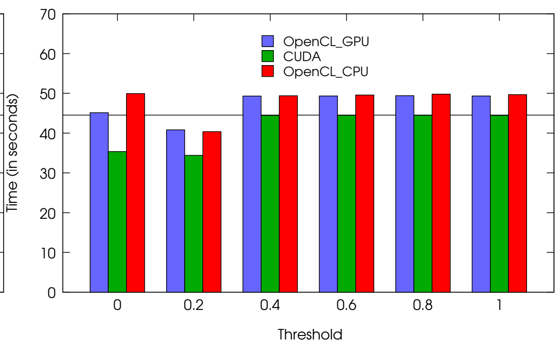
Both of dynamic partitioning algorithms have the same theoretical complexity of $\mathcal{O}(|S|^2|T|^2)$, but they take a different approach in how these partitions are constructed. To determine which of these algorithms is best suitable for this task, we empirically evaluate both partitioning algorithms and recommend the winning one as the partitioning algorithm to be used.

Figures 8.5 to 8.10 present the overall performance of the algorithm when using the two dynamic partitioning algorithms, and indicate their performance for each programming model and platform we use. The hardware platforms were limited to Intel Xeon CPU and the GTX480 GPU. In every graph, we also list the best performance obtained with static partitioning (Chapter 7) for that dataset. Thus, all the bars below this line indicate solutions, based on dynamic partitioning, that lead to better performance than the best static partitioning.

We observe that dynamic partitioning can lead to better performance than the static partitioning, especially for larger datasets. For example, for DS1 and DS2, the difference between static partitioning and dynamic partitioning is still relatively small, so static partitioning is a feasible solution. However, for DS3 to DS6, all datasets show clear perforance improvement when using dynamic partitioning, with differences as large as 60%. Note that on the x-axis of these figures we list the threshold value. This value dictates how strict we are with respect to starting new runs. When the threshold value is high less, runs will be executed. Consequently when the threshold value is low, more runs are executed.

We see from figures 8.5 to 8.10 that in most of the cases a dynamic partitioning

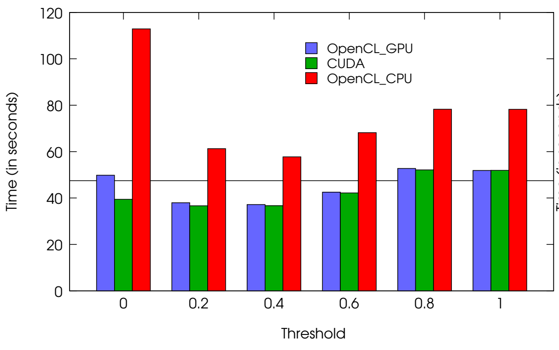


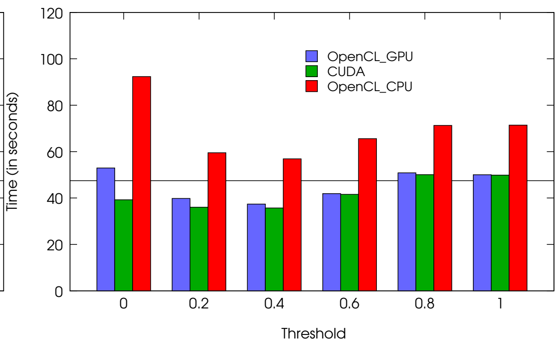


(a) Dynamic partitioning - algorithm 1.

(b) Dynamic partitioning - algorithm 2.

Figure 8.5: Using different partitioning algorithms on DS1. The horizontal line represents the best performance achieved with static partitioning.

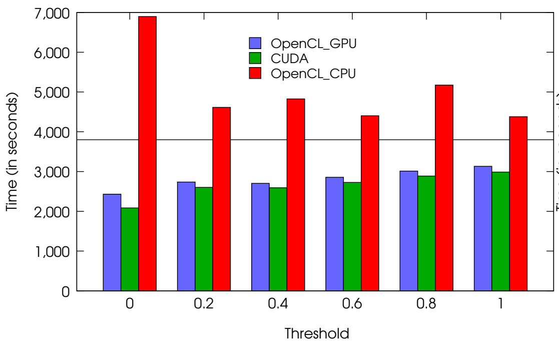


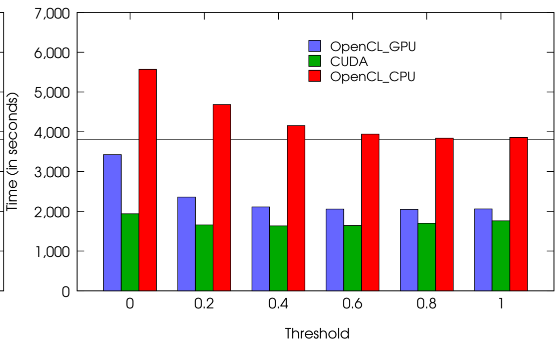


(a) Dynamic partitioning - algorithm 1.

(b) Dynamic partitioning - algorithm 2.

Figure 8.6: Using different partitioning algorithms on DS2. The horizontal line represents the best performance achieved with static partitioning.

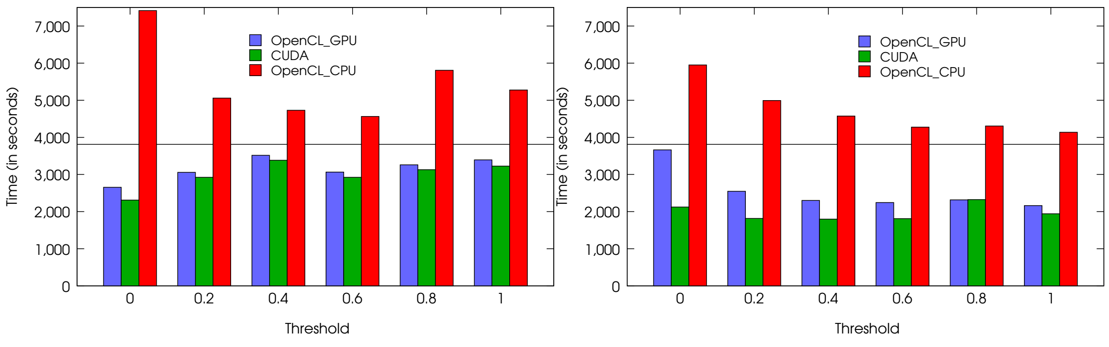




(a) Dynamic partitioning - algorithm 1.

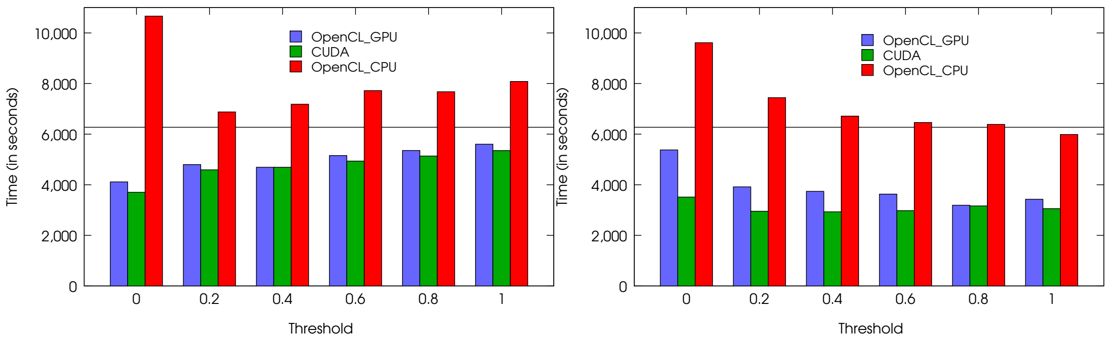
(b) Dynamic partitioning - algorithm 2.

Figure 8.7: Using different partitioning algorithms on DS3. The horizontal line represents the best performance achieved with static partitioning.



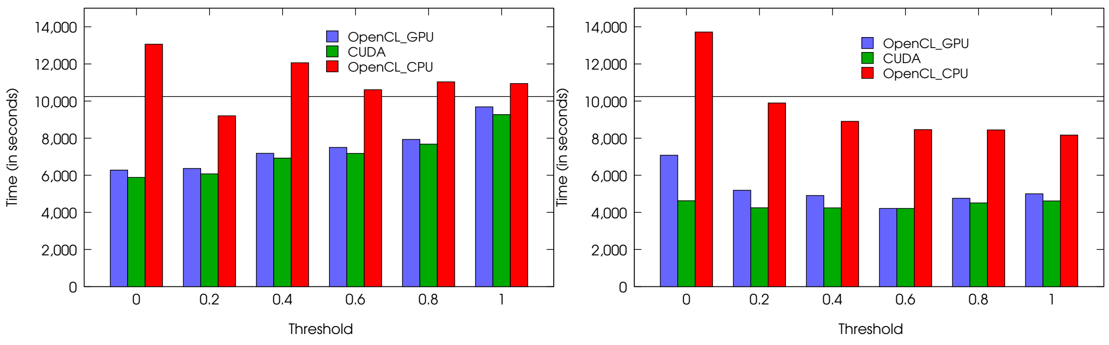
(a) Dynamic partitioning - algorithm 1. **(b)** Dynamic partitioning - algorithm 2.

Figure 8.8: Using different partitioning algorithms on DS4. The horizontal line represents the best performance achieved with static partitioning.



(a) Dynamic partitioning - algorithm 1. **(b)** Dynamic partitioning - algorithm 2.

Figure 8.9: Using different partitioning algorithms on DS5. The horizontal line represents the best performance achieved with static partitioning.



(a) Dynamic partitioning - algorithm 1. **(b)** Dynamic partitioning - algorithm 2.

Figure 8.10: Using different partitioning algorithms on DS6. The horizontal line represents the best performance achieved with static partitioning.

algorithm is a better alternative for increasing performance. In addition we observe that dynamic partitioning algorithm 2 provides faster executions of PaSWAS. Using our findings, we recommend dynamic partitioning algorithm 2 in conjunction with PaSWAS.

CHAPTER 9

Performance Prediction

In this chapter we describe a model for predicting the performance of PaSWAS. Specifically, our goal is to provide an approximation of the execution time of the analysis for a random dataset on a given platform, *before* executing the application.

9.1 Modeling

In order to create the model for PaSWAS, we used one of the larger datasets (more specifically, DS4) as a reference. This dataset is statically partitioned with a threshold of 1024 into 38 sub-datasets (thus, the full analysis requires 38 runs).

The parameters of our model are the length of the sequence, $|s_0|$, the length of the target, $|t_0|$, and the size of a work-group $S_s \times S_t$. The number of alignments that can be executed in parallel on the device is expressed as $n_s \times n_t$. The time needed for an iterative kernel is given by:

$$T = \sum_{i=1}^{n+m-1} T_i \tag{9.1}$$

In equation 9.1, $n = \frac{|s_0|}{S_s}$, $m = \frac{|t_0|}{S_t}$, and T_i denote the time for a single kernel run. The bound $n + m - 1$ stems from the wavefront pattern - e.g., when $n = 3$ and $m = 3$, we need 5 iterations (as can be seen from figure 5.2). If we would have unlimited resources (e.g. the number of processors and the amount of memory are unlimited), each T_i can be approximated by the time it takes to evaluate one work-group T_{wg}, and equation 9.1 can be rewritten to:

$$T = (n + m - 1) \times T_{wg} \tag{9.2}$$

When we have limited amounts of processors and memory (i.e., the real case), each T_i can have a different value, since *not all work-groups in one iteration can be processed* *in parallel.* Therefore, T_i is dependent on the hardware device - more specifically, by the number of work-groups it can process concurrently.

Due to the wavefront pattern of processing, we observe $T_i, 0 \leq i \leq n+m-1$ following three phases:

- Increasing: the number of work-groups to be executed in parallel is increasing until we reach the maximum, which is given by $p = \min(n,m)$. In this phase, we observe an increasing T_i, i.e., $T_i \leq T_{i+1}, 0 \leq i < p$.
- Constant: T_i remains constant since we are continuously processing p work-groups.
- Decreasing: this last phase sees T_i decreasing (i.e., the perfect opposite of the increasing phase). This happens because the number of work-groups to be scheduled for execution decreases in subsequent iterations. Thus, we observe a decreasing T_i, i.e., $T_i \geq T_{i+1}, n+m-1-p-1 \leq i < n+m$.

Each of these phases occur in a specific region of the iterative kernel execution. That is, the number of work-groups increases in the region $i = \{1,\ldots,p\}$, remains constant in $i = \{p+1,\ldots,m+n-1-p-1\}$ and decreases between $i = \{m+n-1-p,\ldots,m+n\}$. To prove our intuition is correct, figures 9.1(a) and 9.1(b) present the execution time of each iteration (i.e., the T_i) for a single sub-dataset in DS4. We observe the three distinct phases emerging in the graphs.

Utilizing the structure of the wavefront pattern, each T_i can be defined as:

$$T_i = \begin{cases} T_1 + \frac{T_p - T_1}{p-1} \times i & 1 \leq i \leq p \\ T_p & p+1 \leq i \leq m+n-1-p-1 \\ T_p - \frac{T_p - T_1}{p-1} \times i & m+n-1-p \leq i \leq m+n \end{cases}$$

Substituting these values in 9.1 yields an approximation of the total execution time T.

Thus, T can be modeled by three "lines" - i.e., one with a positive slope, a horizontal line, and a line with a negative slope. These lines and the slopes are defined by T_1 and T_p (which can be easily measured). Therefore, T_x for $x = \{1,p\}$ is determined by:

$$T_x = \begin{cases} h_c & n_s \times n_t \times b_x \leq h_{wg} \\ \frac{n_s \times n_t \times b_x}{h_{wg}} \times h_c & Otherwise \end{cases}$$

The constant h_{wg} is hardware-related, and depends on the maximum amount of work-groups that can be scheduled in parallel on a device. For example, GTX480 has 15 Streaming Multiprocessors on which 8 work-groups can be scheduled; in this case, $h_{wg} = 120$.

The constant h_c is determined empirically, and it represents the time needed to compute the elements in a work-group of size $S_s \times S_t$. This value is hardware and implementation dependent. For example, the CUDA implementation has a lower h_c than the OpenCL one, according to our experimental findings (see chapter 7). h_c was extracted by processing the complete reference dataset and averaging the timing details for the iterations which ran fully concurrent (e.g. $n_s \times n_t \times b_x \leq h_{wg}$, where b_x is the number work-groups requested in iteration x).

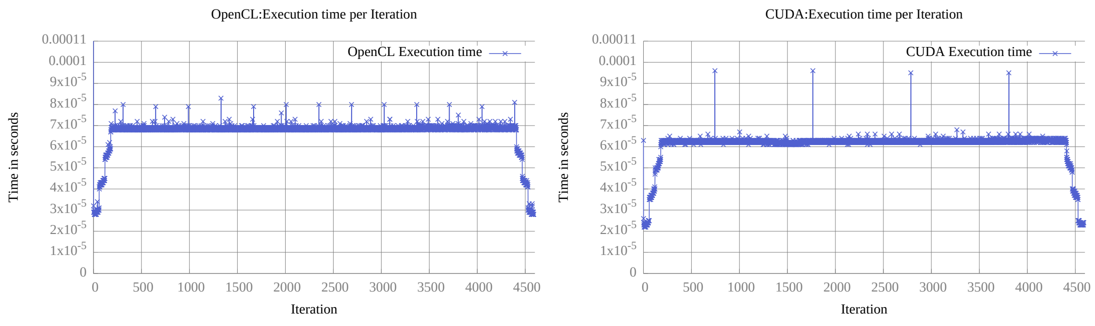
However, as already stated, not every iteration can accomplish a fully concurrent execution. In this case, we approximate T_x by penalizing the extra scheduling steps by the factor of $\frac{n_s \times n_t \times b_x}{h_{wg}}$. This approximation gives us an impression on the additional computation time when more work-groups have to be scheduled then the hardware can process concurrently.

Table 9.1 displays the constants that we have used to model the calculate score kernel. These values were retrieved from the training datasets using a work-group configuration of 8×8. We can model the 16×16 case by multiplying h_c by 2. Since $2 \times (8 + 8 - 1) \approx 16 + 16 - 1$, that is we have to do 2 as much iterations in a work-group of size 16×16, compared to an 8×8 case.

Table 9.1: The constants h_{wg} and h_c for each device and platform.

	GTX480(CUDA)	GTX480(OCL)	GTX680(CUDA)	GTX680(OCL)
h_c	0.000024	0.00003	0.000025	0.000033
h_{wg}	120	120	128	128

We note that our model is less applicable to CPUs because it does not explicitly take into account the scheduling cost of work-groups and context switching between work-items. With CPUs, these costs can be significant.

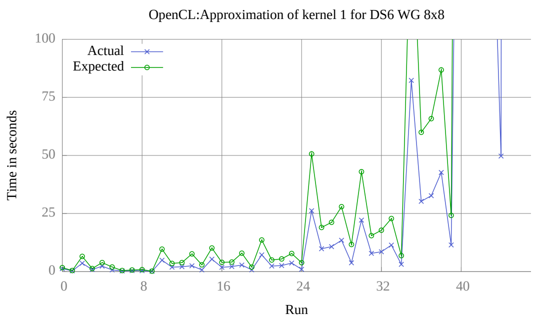


(a) OpenCL GTX480 **(b)** CUDA GTX480

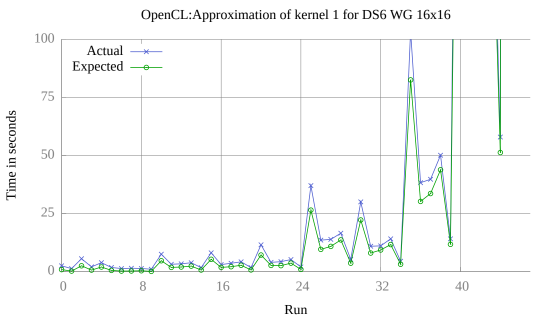
Figure 9.1: Execution time per iteration for the GTX480 platform, on a single sub dataset.

9.2 Validation

To validate our model, we present in figures 9.3 and 9.2 a graphical comparison between the predicted performance and the measured execution time. We see that the behavior is predicted quite accurately, though more calibration might be necessary for some of the platforms.

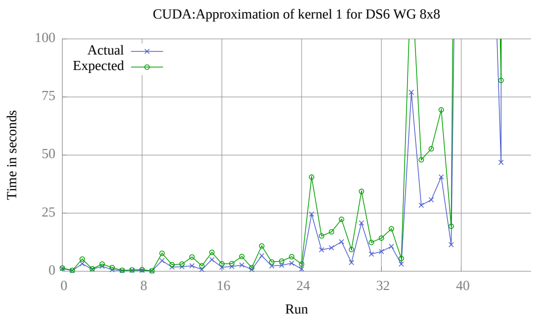


(a) Prediction for the execution time of the OpenCL version with 16x16 work-groups.

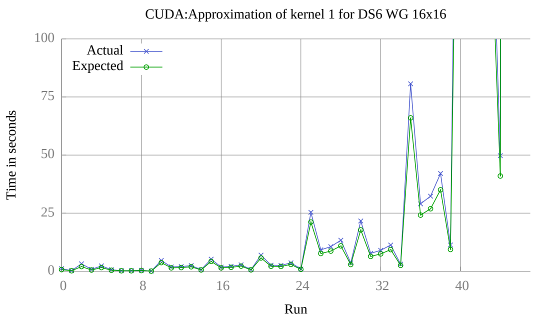


(b) Prediction for the execution time of the OpenCL version with 16x16 work-groups.

Figure 9.2: Execution time per iteration for the GTX480 platform, on a single sub dataset.



(a) Prediction for the execution time of the OpenCL version with 16x16 work-groups.



(b) Prediction for the execution time of the OpenCL version with 16x16 work-groups.

Figure 9.3: Execution time per iteration for the GTX480 platform, on a single sub dataset.

To quantify the accuracy of our prediction (i.e., the difference between our model and the real execution time), we define a similarity measure:

$$Sim(T_{rm},T_{ra}) = \frac{|T_{ra} - T_{rm}|}{T_{rm}} \tag{9.3}$$

In this equation, T_{rm} and T_{ra} represent the predicted execution and the actual running time in run r. The similarity measure for all the runs in a dataset (e.g. DSx) the relative error is then given by:

$$error(DSx) = \frac{\sum_{r=1}^{\#runs} Sim(T_{rm}, T_{ra})}{\#runs} \tag{9.4}$$

In other words, the overall similarity measure is determined by taking the average of the difference between the predicted execution time against the actual time for each run. Table 9.2 presents an overview of these errors for datasets DS3, DS5, and DS6 (datasets DS1 and DS2 are not interesting, because they are too small).

We make the following observations. First, the average errors are acceptable for the implementations using 16×16 work-groups, but are quite large for the implementations using 8×8 work-groups. We believe this is due to insufficient calibration of the parameters, and we plan to further improve them. Second, we note that the prediction for CUDA is more accurate than the prediction for OpenCL. We believe this happens because the execution time per iteration is more stable in CUDA than in OpenCL, and because we do not model correctly the kernel launch overhead. These hypotheses are to be validated empirically.

Overall, we believe our model is able to predict quite well the trends of the execution, it is fairly accurate for executions using 16×16 work-groups, and enables a correct performance-based ranking of the potential platforms for analyzing a dataset. More calibration and tuning are necessary to be able to predict the exact execution time for all platforms and all execution configurations.

Table 9.2: Performance prediction errors for each dataset.

	Dataset	Min	Max	Avg
OpenCL 8x8 GTX480	DS3	0.674	1.120	0.947
	DS5	0.295	1.109	0.906
	DS6	0.369	3.052	1.105
OpenCL 16x16 GTX480	DS3	0.009	0.098	0.043
	DS5	0.013	0.260	0.068
	DS6	0.0028	0.220	0.047
OpenCL 8x8 GTX680	DS3	1.022	1.620	1.370
	DS5	0.504	1.666	1.408
	DS6	0.577	3.929	1.542
OpenCL 16x16 GTX680	DS3	0.278	0.070	0.021
	DS5	0.001	0.226	0.049
	DS6	0.003	0.003	0.059
CUDA 8x8 GTX480	DS3	0.368	0.783	0.626
	DS5	0.130	0.747	0.604
	DS6	0.221	2.304	0.775
CUDA 16x16 GTX480	DS3	0.148	0.199	0.175
	DS5	0.154	0.376	0.198
	DS6	0.149	0.376	0.203
CUDA 8x8 GTX680	DS3	0.586	1.089	0.857
	DS5	0.104	0.707	0.568
	DS6	0.277	2.765	1.014
CUDA 16x16 GTX680	DS3	0.032	0.151	0.098
	DS5	0.001	0.146	0.052
	DS6	0.002	0.273	0.066

CHAPTER 10

Conclusion and Future Work

Comparing RNA, DNA and protein sequences is a common activity in bioinformatics. If two sequences in an alignment have a common ancestor, mismatches can be interpreted as point mutations, and gaps as insertion and deletion mutations. By analyzing these mutations, one can determine in what aspects these organisms differ.

A commonly used algorithm for performing sequence alignments is the Smith-Waterman algorithm. This algorithm performs alignments of two sequences s and t with a time complexity of $\mathcal{O}(|s| \times |t|)$. To align n sequences against m other sequences, the time complexity changes to $\mathcal{O}(n \times |s| \times m \times |t|)$. As datasets are constantly increasing in size, i.e., n,m,$|s|$ and $|t|$ are growing larger, the Smith-Waterman algorithm can be quite slow to compute a set of alignments (from tens of minutes to hours for a common dataset). To reduce this execution time, various parallel versions of the Smith-Waterman algorithm have been developed (see section 3. For this thesis, we have chosen to start our evaluation from the implementation in [War+15].

In this context, the goal of this thesis was to provide an empirical evaluation of the parallel Smith-Waterman algorithm on different hardware platforms (different GPUs and multi-core CPUs), aiming to understand which ones are more suitable for large, fast bioinformatics processing. In this chapter we summarize the insights we have gathered from this evaluation and draw a set of conclusions. We further sketch several future work directions.

10.1 Contributions and Findings

We revisit here our research questions from section 1.2, listing our findings.

- (R1): Can we achieve similar performance on a NVIDIA GPU when porting the CUDA code of PaSWAS to OpenCL?

In sections 7.2.2 and 7.2.3, we have shown that OpenCL has reasonable performance for the PaSWAS algorithm. In general, CUDA takes on average 30% less than OpenCL to process the same datasets. This happens due to the ability of the CUDA compiler to better unroll loops (as observed by our direct PTX code analysis). Additionally, OpenCL kernel launches are more expensive (about 50-60% slower than CUDA). Since PaSWAS launches its kernel iteratively, i.e., it utilizes multiple kernel launches to solve the alignment problem, this will have an immediate impact on the performance of our OpenCL implementation, lowering it visibly.

- (R2): Which are the platform-specific parameters that have a high performance impact?

Our OpenCL implementation has allowed a similar implementation for both families of devices - CPUs and GPUs. In order to specialize the code in a performance-aware manner for each type of platform. Specifically, we had tuned the local memory usage on and off, we had enabled and disabled zero-copies, and we have tuned the workload granularity per thread. All these are, in fact, parameters of a generic OpenCL implementation of PaSWAS. From sections 7.2.2, 7.2.3, and 7.2.1, we found that a PaSWAS implementation that uses solely global memory, zero-copy memory transfers, and coarser granularity (i.e., each work-item computes more than one element) provided the best performance on the CPUs. Furthermore, for GPUs, a shared memory implementation achieves the best performance. In terms of zero-copy memory, the effects were surprinsingly different: the Kepler GPUs (GTX 680) show a drop in performance when using zero-copy memory transfers, while the Fermi GPUs (GTX 480) showed a significant performance increase when memory transfers were enabled through zero-copy. Overall, we conclude that this OpenCL code specialization via these parameters is important for allowing both GPUs and CPUs to be competitive, performance-wise.

- (R3): How can we partition a dataset to enable the most parallelism?

Two challenges need to be met when using accelerators to improve the performance of PaSWAS: sequences are getting larger and more "irregular" (i.e., sequences to be aligned are of significantly different lengths), and the memory of devices remains limited. Therefore, a smart partitioning of the input datasets is required to obtain as much as possible from the parallelization of the problem. In chapter 8, we demonstrated that properly partitioning a dataset directly leads to exposing more parallelism, and thus better performance. This relates to the observation that larger sequences limit the possible parallelism to be achieved. By clustering sequences in batches of similar sizes, we could maximize the amount of parallelism for every run. We have also observed that the most specialized algorithm does not always provide the best solution to the partitioning problem - instead, a less complex algorithm, leading to a coarser-grain clustering provides the best solution in terms of performance.

- (R4): Can we provide a performance predictor that can estimate the performance of PaSWAS on a given platform, with a given dataset?

Even after parallelization, the execution time of PaSWAS depends heavily on the size of the dataset and the platform used for processing. We attempted to predict this execution time for each device by using an analytical model, calibrated with a short training phase. In chapter 9, we discuss the modeling of PaSWAS for different platforms. Using the empirical data from one of our datasets, we were able to train our performance estimator and predict the performance of other datasets on each hardware platform. Our predictor works very well for GPUs, where the execution time is fairly stable, and very limited training is needed for calibration. We have observed, however, that the performance on the CPUs is much more difficult to predict. In conclusion, we are able to predict the execution time for the GPU platforms, enabling, for example, the a-priori selection of the best platform for running the analysis of a given dataset, but an accurate prediction for CPUs requires more work to understand (or, rather, reverse engineer) the mapping of the OpenCL platform to the hardware itself.

10.2 Future Work

Based on our findings we suggest three interesting directions for further research.

First, our OpenCL implementation of PaSWAS is based on the CUDA version from [War13]. Such a reference does not exist for the CPU platforms. To this end, we believe an OpenMP version can be created to determine whether OpenCL and OpenMP provide comparable performance.

Second, our performance predictor must be tweaked and tuned to work better for CPU prediction as well. This is really important for cases where many datasets need to be analyzed, and both the CPU and the GPU can be employed to solve this task: a high-level (static) scheduler can use the prediction information to partition the datasets such that both the CPU and the GPUs finish in comparable time.

Third, and somewhat a generalization of the previous point, is the use of heterogeneous, distributed systems. Since each alignment can be computed independently, utilizing multi-node and/or multi-device platforms should be feasible. The biggest research challenge in this case would be to determine how to distribute the work on different nodes and devices such that a reasonable work-balance is achieved.

Bibliography

[Aho86] Alfred V Aho. 'Compilers: Principles, Techniques, And Tools Author: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Publisher: Addison Wesle'. In: (1986) (cit. on p. 20).

[Alt+90] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 'Basic local alignment search tool'. In: *Journal of molecular biology* 215.3 (1990), pp. 403–410 (cit. on p. 19).

[Anv+01] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. 'Generating parallel programs from the wavefront design pattern'. In: *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. IEEE. 2001, 8–pp (cit. on p. 29).

[Aus99] Giorgio Ausiello. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 1999 (cit. on p. 1).

[Bal+00] Henri Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, Jason Maassen, Rob Van Nieuwpoort, John Romein, Luc Renambot, Tim Rühl, et al. 'The distributed ASCI supercomputer project'. In: *ACM SIGOPS Operating Systems Review* 34.4 (2000), pp. 76–96 (cit. on p. 43).

[Cla08] S. Clancy. 'RNA functions. Nature Education'. In: *Nature Education* 1 (2008), pp. 1–102 (cit. on p. 5).

[Cri70] F. Crick. 'Central Dogma of Molecular Biology'. In: *Nature* 227 (1970), pp. 561–563 (cit. on p. 5).

[CS 14] J. Tan C.S. Greene. 'Big Data Bioinformatics'. In: *Journal of Cellular Physiology* 229 (2014), pp. 1896–1900 (cit. on pp. 1, 10).

[Dij68] Edsger W Dijkstra. 'A constructive approach to the problem of program correctness'. In: *BIT Numerical Mathematics* 8.3 (1968), pp. 174–186 (cit. on p. 36).

[Du+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 'From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming'. In: *Parallel Computing* 38.8 (2012), pp. 391–407 (cit. on p. 2).

[Fan+11] Jianbin Fang, Ana Lucia Varbanescu, and Henk J. Sips. 'A Comprehensive Performance Comparison of CUDA and OpenCL'. In: *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*. 2011, pp. 216–225 (cit. on p. 20).

[Far07] Michael Farrar. 'Striped Smith–Waterman speeds database searches six times over other SIMD implementations'. In: *Bioinformatics* 23.2 (2007), pp. 156–161 (cit. on p. 20).

[Gar+79] M. Garey and D. Johnson. 'Computers and Intractability: A Guide to the Theory of Np-completeness'. In: *W. H. Freeman* (1979) (cit. on p. 59).

[Hai+11] Doug Hains, Zach Cashero, Mark Ottenberg, Wim Bohm, and Sanjay Rajopadhye. 'Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices'. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 490–501 (cit. on p. 44).

[Int14] Intel. 'Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics'. In: (2014) (cit. on pp. 39, 42).

[Int11] Intel. 'Writing Optimal OpenCL Code with Intel OpenCL SDK.' In: (2011) (cit. on p. 47).

[Irv04] Robert W Irving. 'Plagiarism and collusion detection using the Smith-Waterman algorithm'. In: *University of Glasgow* (2004) (cit. on p. 9).

[J S+12] H. Sips J. Shen J. Fang and A.L. Varbanescu. 'Performance Gaps between OpenMP and OpenCL for Multi-core CPUs'. In: *Parallel Processing Workshops (ICPPW), 2012 41st International Conference* (2012), pp. 116–125 (cit. on p. 15).

[J S+13] H. Sips J. Shen J. Fang and A.L. Varbanescu. 'Performance Traps in OpenCL for CPUs'. In: *PDP 2013* (2013), pp. 38–45 (cit. on pp. 15, 20, 21, 48).

[JE +90] H.F. Lodish J.E. Darnell and D. Baltimore. 'Molecular Cell Biology'. In: 1 (1990) (cit. on p. 5).

[Kar+10] Kamran Karimi, Neil G. Dickson, and Firas Hamze. 'A Performance Comparison of CUDA and OpenCL'. In: *CoRR* abs/1005.2581 (2010) (cit. on p. 39).

[Kle+06] J. Kleinenberg and E. Tardos. 'Algorithm Design'. In: *Pearson education* (2006) (cit. on p. 8).

[Kom+10] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. ‘Evaluating performance and portability of OpenCL programs’. In: *The fifth international workshop on automatic performance tuning*. 2010, p. 7 (cit. on pp. 20, 49).

[Kor+00] Madhukar R Korupolu, C Greg Plaxton, and Rajmohan Rajaraman. ‘Analysis of a local search heuristic for facility location problems’. In: *Journal of algorithms* 37.1 (2000), pp. 146–188 (cit. on p. 59).

[Lip+85] David J Lipman and William R Pearson. ‘Rapid and sensitive protein similarity searches’. In: *Science* 227.4693 (1985), pp. 1435–1441 (cit. on p. 19).

[Liu+13] 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* 14.1 (2013), p. 117 (cit. on p. 44).

[M D+11] A. Aji M. Daga and W. Feng. ‘On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing’. In: *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)* abs/1005.2581 (2011), pp. 141–149 (cit. on pp. 39, 40).

[Man+08] Svetlin A Manavski and Giorgio Valle. ‘CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment’. In: *BMC bioinformatics* 9.Suppl 2 (2008), S10 (cit. on pp. 20, 44).

[Mor99] Burkhard Morgenstern. ‘DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment.’ In: *Bioinformatics* 15.3 (1999), pp. 211–218 (cit. on p. 19).

[Nee+70] Saul B Needleman and Christian D Wunsch. ‘A general method applicable to the search for similarities in the amino acid sequence of two proteins’. In: *Journal of molecular biology* 48.3 (1970), pp. 443–453 (cit. on p. 19).

[Net+92] Robert HB Netzer and Barton P Miller. ‘What are race conditions?: Some issues and formalizations’. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.1 (1992), pp. 74–88 (cit. on p. 33).

[Nic+08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. ‘Scalable parallel programming with CUDA’. In: *Queue* 6.2 (2008), pp. 40–53 (cit. on p. 2).

[Nvi11] Nvidia. ‘Best Practices Guide OpenCL Nvidia’. In: (2011) (cit. on pp. 40, 42).

[Ope14] OpenMP. *OpenMP*. 2014. URL: `http://openmp.org/wp` (cit. on p. 10).

[Pha+10] Vu Pham, Phong Vo, Vu Thanh Hung, et al. 'GPU implementation of extended gaussian mixture model for background subtraction'. In: *Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2010 IEEE RIVF International Conference on*. IEEE. 2010, pp. 1–4 (cit. on p. 39).

[Reh+09] M Suhail Rehman, Kishore Kothapalli, and PJ Narayanan. 'Fast and scalable list ranking on the GPU'. In: *Proceedings of the 23rd international conference on supercomputing*. ACM. 2009, pp. 235–243 (cit. on p. 30).

[Rog11] Torbjørn Rognes. 'Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation'. In: *BMC bioinformatics* 12.1 (2011), p. 221 (cit. on p. 20).

[Seo+11] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 'Performance characterization of the NAS parallel benchmarks in OpenCL'. In: *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 137–148 (cit. on p. 30).

[She+13] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. 'An application-centric evaluation of OpenCL on multi-core CPUs'. In: *Parallel Computing* 39.12 (2013), pp. 834–850 (cit. on p. 35).

[Sla96] Petr Slavík. 'A tight analysis of the greedy algorithm for set cover'. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM. 1996, pp. 435–441 (cit. on p. 59).

[Smi+81] Temple F Smith and Michael S Waterman. 'Identification of common molecular subsequences'. In: *Journal of molecular biology* 147.1 (1981), pp. 195–197 (cit. on pp. 1, 19).

[Sto+10] John E Stone, David Gohara, and Guochun Shi. 'OpenCL: A parallel programming standard for heterogeneous computing systems'. In: *Computing in science & engineering* 12.1-3 (2010), pp. 66–73 (cit. on pp. 2, 35).

[Stu+92] Bernard Stumpf, George M Stabler, Richard G Bahr, Stephen J Ciavaglia, Barry J Flahive, and Hugh Lauer. *Method and apparatus for bus lock during atomic computer operations*. US Patent 5,175,829. Dec. 1992 (cit. on p. 33).

[War+15] S. Warris, F. Yalcin, K.J.L. Jackson, and J.P. Nap. 'Flexible, fast and accurate sequence alignment profiling on GPGPU with PaSWAS'. In: *PLOS ONE* (2015). In press. (cit. on pp. 2, 20, 73).

[War13] Sven Warris. *PaSWAS: Parallel Smith-Waterman alignment Software*. `https://github.com/swarris/PaSWAS`. 2013 (cit. on p. 75).

[Woz97] Andrzej Wozniak. 'Using video-oriented instructions to speed up sequence comparison'. In: *Computer applications in the biosciences: CABIOS* 13.2 (1997), pp. 145–150 (cit. on p. 20).

List of Figures

List of Tables

APPENDIX A

Appendix A: Work-group configuration

In this appendix we will proof that a square work-group is the best choice. A work-group is evaluated in:

$$I(x,y) = x + y - 1 \tag{A.1}$$

Where $I(x,y)$ signifies the number iterations needed when we have a work-group dimension of $x \times y$

$$T = x \times y \tag{A.2}$$

Where T denotes the number of work-items.
Our goal is to minimize A.1. Using A.2, A.1 can be written as:

$$I(x) = x + \frac{T}{x} - 1 \tag{A.3}$$

In order to minimize A.1 we have to show that A.3 has an extreme and that the extreme is indeed a minimum. The value where the extreme of A.3 occurs, can be calculated by setting the first derivative of A.3 to 0. Which means:

$$1 - \frac{T}{x^2} = 0 \tag{A.4}$$

This gives us A.5, from A.2 it immediately follows that A.6 also holds. Further it can be easily shown that the second order derivative of A.3 is greater than 0. Which means that the extreme is indeed a minimum.

$$x = \sqrt{T} \tag{A.5}$$

$$y = \sqrt{T} \tag{A.6}$$

Since $x = y$, it follows that a square work-group requires a minimum number of iterations. Q.E.D.

Acknowledgments

There are a number of people without whom this thesis would not come into fruition and to whom I am extremely grateful. I would like to take this opportunity to thank them.

My supervisor Ana Lucia Varbanescu, for her valuable feedback and guidance throughout this thesis. She was always ready to help and her positive outlook on things were very motivating to me. She has been a key figure for the realization of this thesis.

Sven Warris, for filling the gap in my knowledge of bioinformatics and biology in particular. In addition, he also provided pointers for finding credible biological dataset.

My father, Radheshamsing Eddy Timal, for his continuing support and motivation. Also for providing help during programming, whenever needed. Dad, thanks for the "Vers bakkie koffie" every morning.

My mother, Chandra Debi Ratan, for nurturing me while my sweat drops were pouring on my thesis. Additionally, I would like to thank my mother for being my mother in general.