Parallelization of DNA alignment algorithms using GPUs

Gustavo Páramos Merino Faria Encarnação

Dissertação para obtenção do Grau de Mestre em Engenharia Informática e de Computadores

Júri
Presidente: Doutor Joaquim Armando Pires Jorge
Orientador: Doutor Nuno Filipe Valentim Roma
Vogais: Doutor Luis Manuel Silveira Russo

Maio de 2012
Acknowledgments

I would like to thank all my friends and family who, everyday in these past two years, through peer-pressure and constantly questioning me when I would finally deliver my thesis and finish my degree, gave me motivation to continue and to not give up.

I would also like to thank Professor Nuno Roma and Nuno Sebastião for their patience, support, advice and ideas without which this thesis would not have happened.

This thesis was performed in the scope of project “HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis”, funded by the Portuguese Foundation for Science and Technology (FCT) with reference PTDC/EEA-ELC/113999/2009, and project “TAGS: The power of the short - Tools and Algorithms for next Generation Sequencing applications”, funded by FCT with reference PTDC/EIA-EIA/112283/2009.
Abstract

Since its discovery, the Deoxyribonucleic Acid (DNA) has been the object of thorough study. The significant advances in sequencing technologies [1] have allowed researchers to obtain DNA sequences at ever increasing rates, with the consequent growth of the corresponding databases. Such huge amounts of data require efficient processing tools to extract useful information. However, these tools have advanced at a slower pace and have become one of the limiting factors of new discoveries in this field of research.

Recently, from the 3D game market, a new generation of hardware has emerged. This hardware, known as Graphics Processing Units (GPUs), now offers the capability to perform general purpose computations. With these new hardware devices, high performance computing has become possible using cheap and readily available hardware which creates new opportunities to study and improve current tools and algorithms used for the study of DNA.

The goal of this dissertation was to discover a way of using these new processing platforms to improve the current tools, techniques and algorithms used for DNA analysis. To attain this goal, several data structures and algorithms were considered and thoroughly analysed in terms of performance and flexibility when executed on a GPU. Afterwards, these structures were used in the implementation of both exact and approximate string matching algorithms especially adapted for DNA sequences.

The outcome of the performed work was the development of a new tool for heuristic DNA alignment that is capable of harnessing the raw computational performance provided by the highly parallel GPU device.

Conducted tests showed that the performed work led to the creation of a tool that is capable of competing with currently available and commonly used software in both performance and quality of results.

Keywords

Exact String Matching, Approximate String Matching, DNA Alignment, High-Performance Computing, Graphics Processing Unit (GPU)
Resumo

Desde a sua descoberta, o DNA tem sido objecto de estudo intensivo. Os avanços significativos em tecnologias de sequenciamento [1] têm permitido obter sequências de DNA a ritmos cada vez maiores, com o consequente crescimento das bases de dados correspondentes. Tamanha quantidade de dados requer ferramentas de processamento eficientes para extrair informação útil. No entanto, estas ferramentas têm avançado a um ritmo mais lento e tornaram-se num dos factores limitativos de novas descobertas neste campo de pesquisa.

Recentemente, impulsionada pelo mercado do entretenimento e dos jogos 3D, surgiu uma nova geração de processadores. Estes processadores, conhecidos como processadores gráficos (GPU), são capazes de executar não só as transformações geométricas necessárias em jogos mas também programas genéricos de alguma complexidade. Com estes novos processadores, computação de alto desempenho tornou-se possível de realizar em plataformas baratas e de acesso fácil o que cria novas oportunidades para o estudo e melhoria das ferramentas e técnicas para o estudo do ADN actuais.

O objectivo desta dissertação foi o de descobrir uma forma de aproveitar estas novas plataformas para melhorar as ferramentas para estudo do ADN actuais. Para atingir este objectivo diversas estruturas de representação de dados e algoritmos foram estudadas e cuidadosamente analisadas em termos desempenho e flexibilidade quando executados no GPU. Posteriormente, estas estruturas foram utilizadas, na implementação de algoritmos para procura exacta e aproximada de cadeias de caracteres, especialmente adaptados para sequências de DNA.

O resultado do trabalho realizado foi o desenvolvimento de uma nova ferramenta heurística para alinhamento de DNA, capaz de aproveitar o poder computacional fornecido pelos GPUs.

Os testes realizados demonstram que o trabalho realizado levou criação de uma ferramenta capaz de competir com as ferramentas actuais, tanto em termos da qualidade dos resultados como na velocidade a que estes resultados são obtidos.

Palavras Chave

Procura Exacta, Procura Aproximada, Alinhamento de DNA, Computação de Alto Desempenho, Unidade de Processamento Gráfico
# Contents

1 Introduction 1
   1.1 Motivation .................................................. 2
   1.2 Objectives .................................................. 2
   1.3 Main contributions ......................................... 3
   1.4 Dissertation outline ....................................... 3

2 General Purpose Computation on Graphics Processing Units 5
   2.1 GPGPU hardware .............................................. 6
      2.1.1 NVidia G80 and GF100 architectures .................. 6
      2.1.2 AMD FireStream architecture .......................... 7
   2.2 GPGPU performance issues ................................. 8
      2.2.1 Warp divergence and SIMT paradigm .................. 8
      2.2.2 Memory access ......................................... 9
   2.3 GPGPU programming languages ............................. 10
      2.3.1 NVidia CUDA ............................................ 11
      2.3.2 ATI FireStream SDK .................................. 13
      2.3.3 OpenCL ................................................ 14
      2.3.4 Microsoft Direct Compute ............................. 14

3 DNA Alignment Methods 15
   3.1 Local and global alignment ................................. 16
   3.2 Dynamic programming DNA alignment ....................... 16
      3.2.1 DNA analysis using strings .......................... 16
      3.2.2 Needleman-Wunsch .................................... 17
      3.2.3 Smith-Waterman ....................................... 17
   3.3 Indexed search ................................................ 18
      3.3.1 Suffix tries and suffix trees ......................... 18
      3.3.2 Suffix arrays .......................................... 23
      3.3.3 Hash tables and q-mers ............................... 24
      3.3.4 Burrows-Wheeler transform ........................... 24
   3.4 Summary of DNA Analysis tools ............................ 26

4 Exact Matching 27
   4.1 Data representation .......................................... 28
      4.1.1 Reference sequence .................................... 28
      4.1.2 Query sequences ....................................... 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>GPU Initialization</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Memory allocation</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Host to device data copy</td>
<td>34</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Kernel launch</td>
<td>34</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Memory de-allocation</td>
<td>36</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Time measurement and GPU events</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Exact matching on the GPU</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Suffix Tree Kernel</td>
<td>37</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Suffix Array Kernel</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparison and discussion of different approaches</td>
<td>41</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Overlap of communication and computation</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>Approximate Matching</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>Possible approaches for approximate matching</td>
<td>48</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Suffix arrays</td>
<td>49</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Enhanced suffix array</td>
<td>50</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Suffix trees</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Discussion and Comparison of the considered approaches</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>DNA alignment using GPUs</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>Seeding</td>
<td>64</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Parameters</td>
<td>65</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Seeding By Counting</td>
<td>65</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Seeding By Sorting</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Smith-Waterman</td>
<td>69</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Kernel modifications</td>
<td>69</td>
</tr>
<tr>
<td>6.2.2</td>
<td>GPGPU Kernel</td>
<td>70</td>
</tr>
<tr>
<td>6.3</td>
<td>Results and comparison</td>
<td>71</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Parameter influence</td>
<td>71</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Seeding by counting versus Seeding by sorting</td>
<td>72</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Comparison with BLAST</td>
<td>72</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Comparison with MUMmer</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>75</td>
</tr>
<tr>
<td>7.1</td>
<td>Future work</td>
<td>76</td>
</tr>
</tbody>
</table>
List of Figures

2.1 NVidia Architectures. ................................................................. 7
2.2 Warp divergence. ................................................................. 8
2.3 Memory access of several threads in a warp simultaneously. ................. 9
2.4 Threads accessing words stored in different memory blocks. ................. 10
2.5 Typical GPGPU execution. ......................................................... 11
2.6 Memory available by computational environment on the GPU. .................. 13
2.7 Compilation of CUDA applications ............................................. 13

3.1 DNA Alignment showing portions between query and reference that match. ...... 17
3.2 Dynamic programming matrix and formulas for Needleman-Wunsch. .............. 18
3.3 Suffix Trie and Suffix Tree representation for the string 'BANANAS'. .................. 19
3.4 Suffix tree for the string 'MISSISSIPPI'. ........................................ 19
3.5 Suffix tree construction. .............................................................. 22
3.6 Suffix array for the string "mississippi". ......................................... 23
3.7 Burrows-Wheeler transform for the string "mississippi". ............................ 25

4.1 Flattened Suffix Tree schemes. .................................................. 30
4.2 Flattened Suffix Tree with static edges. ........................................ 30
4.3 Coalesced query matrix. Each column holds a single query sequence. Each cell holds 16 DNA bases. .......................................................... 33
4.4 Example of how using two streams for concurrent memory copy and execution can improve performance by reducing time spent copying data from the host to the device. 35
4.5 Unaligned suffixes. ................................................................. 40
4.6 Performance evaluation of suffix tree and suffix array index based search algorithms in multi-core CPUs. .................................................. 42
4.7 Performance evaluation of the considered index based search algorithms in the GPU. 42
4.8 Communication and kernel execution times for the implementations in the GeForce GTX 580 GPU. .......................................................... 43
4.9 Performance evaluation of suffix tree and suffix array index based search algorithms in GPUs using overlap of communication and computation with a partitioned reference sequence. .................................................. 45
4.10 Performance evaluation of suffix tree and suffix array index based search algorithms in GPUs using overlap of communication and computation with a partitioned query set. .......................................................... 45

5.1 DNA Alignment showing portions between query and reference that match. ...... 48
List of Figures

5.2 Removing the prefix to find a better match for 'MISSIPPI'. 49
5.3 Enhanced suffix array using an extra column to hold the longest common prefix values. 50
5.4 Enhanced suffix array using an extra column to hold the longest common prefix values and another column for the index of the next suffix. 53
5.5 Example of a suffix array with various LCP intervals. 54
5.6 Example of a suffix array indexed through a hash table with a key length of 2. 55
5.7 Example of a suffix tree. 58
5.8 Suffix tree with one field holding the suffix link. 59
5.9 Suffix tree with one field holding the suffix link and each edge leading to a different edge. 59
5.10 Performance evaluation of suffix arrays based algorithms using a hash table as index on the GPU. 61
5.11 Performance evaluation of suffix trees and suffix arrays index based search algorithms on the GPU. 62
6.1 Data flow between the several processing phases. 64
6.2 Example of merging two seeds. 66
6.3 Example of merging two seeds. 67
6.4 Example of the used data structures after inserting 10 seeds. 68
6.5 Example of the sorted list of seeds. 68
6.6 Calculation of the score in blocks by the Kernel. 70
List of Tables

3.1 List of relevant algorithms. ............................................................... 26

6.1 Execution time, in seconds (s), and sum of errors (time/error) for changing seed size and minimum number of merges (fixed merge-window size= 20) when aligning the query sequences with the $250 \times 10^3$ nucleotide long reference sequence. .... 71

6.2 Execution time, in seconds (s), and sum of errors (time/error) for changing seed size and merge-window size (fixed minimum number of merges= 2) when aligning the query sequences with the $250 \times 10^3$ nucleotide long reference sequence. .... 72

6.3 Comparison of the execution time, in seconds (s), and sum of errors (time/error) for changing seed size when aligning the query sequences with the $1 \times 10^6$ nucleotide long reference sequence. .......... 72

6.4 Comparison of the execution time, in seconds (s), and sum of errors (time/error) for changing seed size when comparing the higher similarity query sequences. .... 72

6.5 Results of the alignment task using BLAST and the GPU algorithm. ......... 73

6.6 Results of the alignment task using MUMmer and the GPU algorithm. ........ 74
Introduction

Contents

1.1 Motivation .................................................. 2
1.2 Objectives .................................................. 2
1.3 Main contributions ......................................... 3
1.4 Dissertation outline ........................................ 3
1. Introduction

Since the model for DNA was initially proposed in the early 20th century great advances have been made in the area, including the unveiling of new areas of scientific research such as genetics. Nowadays, the ability to completely sequence the DNA of any organism is real and the methods for such task are constantly improving. As a result of these new and faster methods there are large amounts of data being generated. The study and analysis of this data is mainly done by comparing very large numbers of DNA sequences, which is only possible through the use of computers and specialized algorithms. These algorithms however have not improved at the same rate as the sequencing platforms have.

The developed work is part of the growing focus on parallel processing as a means to solve time consuming problems in the shortest amount of time possible. It is also part of the growing interest in GPUs as platforms for efficient and affordable parallel processing.

Although documentation for General Purpose Graphics Processing Unit (GPGPU) computing, in particular Compute Unified Device Architecture (CUDA), is plentiful the platform itself is still in its early stages with some issues that must be addressed before it becomes the ideal platform to solve complex, parallelizable problems. One of such issues is memory access which, when thousands of threads are being run concurrently, can lead to performance bottlenecks.

Some researchers who view the GPU simply as another multi-core/many-core processor are disappointed when they realize they are not so. GPUs possess many-cores but their design is only efficient in parallel processing if all cores execute the same task but on different data. Although this does limit the usability of GPUs in parallelizable problems when the individual tasks are very dissimilar it does make them extremely efficient in solving problems where the Same-Instruction Multiple-Threads (SIMT) computing model can be applied.

1.1 Motivation

A result of the difference in the performance between the sequencing platforms and the data processing tools is that there is a much greater increase on the data being created (whole DNA sequences for many different organisms) than in the ability to process such huge amounts of data in a reasonable amount of time. Furthermore, as more data is available the number of comparative analysis that needs to be performed also increases.

Studying and understanding the genetic make-up of every living being is a means to better understand ourselves and to protect us. Thus, it is important that the tools to be further developed and improved upon.

Meanwhile, the main processor manufacturers, while trying to improve their products and having reached the practical limit for the processor clock frequency, are focusing their research efforts ever more on processors with multiple physical cores, capable of running various programs simultaneously. This efforts apply not only to Central Processing Units (CPUs) but also to GPUs.

1.2 Objectives

The objective of this thesis is the development of improved tools for bioinformatics applications that make use of the large parallel processing capabilities provided by current GPUs.

To attain this goal, a thorough analysis of current available tools is done, followed by the development of two algorithms for exact and approximate sequence matching and their adaptation
1.3 Main contributions

This work provides an accurate analysis of the advantages that arise from using GPUs as the platform for parallel processing of bioinformatics applications. This analysis takes into consideration all of the overheads necessary to make use of the GPU device. More specifically, the results consider the time necessary to obtain meaningful results from the device, which include the initialization, processing and data transfer times.

From this analysis two approaches were developed to solve the exact matching problem using GPUs. Each of these approaches uses a distinct data indexing method in order to evaluate which is better suited for the task.

Lastly, the two implemented algorithms for exact matching were used as the basis for the development of two algorithms for heuristic DNA alignment. The obtained results were compared with the commonly used BLAST tool in both terms of performance and quality of results.

1.4 Dissertation outline

This dissertation is composed of seven chapters structured in the following way:

- In Chapter 2 the hardware upon which the algorithms are run is introduced. The general architecture of a GPU is described based on the architectures of the two largest chipset creators (NVidia and ATI). Several programming APIs for the GPU are also introduced and described, with a special emphasis on NVidia’s CUDA which will be adopted throughout the work.

- Chapter 3 focuses on the DNA alignment methods. The different types of alignment are described along with the algorithms most commonly used to obtain those alignments. Several alternatives to DNA sequence indexing are presented, in greater detail the suffix tree, and a method for online creation of suffix trees.

- The problem of exact sequence matching and its implementation on the GPU is approached in Chapter 4, which includes the developed solution to the problem and the underlying data structures. This chapter also describes the considerations that were taken into account when developing such solution for the GPU. The results for the exact matching problem are also included in this chapter.

- Chapter 5 describes several options to solve the approximate DNA matching problem, their benefits and drawbacks. In the end of the chapter these options are compared in order to choose which is best suited to be incorporated into an two-step algorithm for DNA alignment.

- The DNA alignment algorithm developed as a main contribution of this dissertation is presented in Chapter 6. The algorithm is explained giving more detail to the second part which involves processing the seed locations discovered through approximate matching and the GPU implementation of the Smith-Waterman algorithm. The results obtained from this algorithm are compared against other common DNA alignment tools at the end of the chapter.
1. Introduction

- In chapter 7 the conclusions are drawn, and some considerations about further possible improvements of the presented tools are also presented.
General Purpose Computation on Graphics Processing Units

Contents

2.1 GPGPU hardware ......................................................... 6
2.2 GPGPU performance issues ........................................... 8
2.3 GPGPU programming languages ................................. 10
2. General Purpose Computation on Graphics Processing Units

One factor of extreme importance to CPU performance is the clock speed, since the higher it is the more instructions the CPU can perform in the same amount of time. One side effect of high clock speeds is the significant rise in temperature that comes along with them. One way to increase the processor performance without increasing the clock speed and therefore the operating temperature is parallel processing. With this in mind the first multicore CPUs began appearing and parallel computing became popular in general purpose computers which in turn led to GPUs being adapted to general purpose computation.

2.1 GPGPU hardware

Graphics cards, with their shader engines, were initially designed to apply transformations to a large set of elements at a time (pixels or vertexes). The ability to program these shaders to perform more than geometric or color transformations opened graphics cards to multipurpose computing, although the first versions were very limited and complex to use as all computations had to be thought of as typical color or geometry transformations.

2.1.1 NVidia G80 and GF100 architectures

NVidia’s approach to the GPU in the G80 architecture [2], also named Tesla, departed from the former traditional model which used a mix of vertex and pixel shader engines. Instead, the G80 is composed by a set of multiprocessors, each of which contains a number of cores called stream processors. These stream processors can be used either as vertex or pixel engine and it is this flexibility that allows GPGPU applications to use all of them for general computation. [3]

This new hardware architecture and the software (libraries and programming languages) that allow GPGPU computation on NVidia’s products form NVidia’s parallel computing architecture named CUDA.

The basic building block in the G80 architecture is the Streaming Multiprocessor (SM). Although the number of SMs on the GPU varies from one graphics card to another, depending on their model, the SMs themselves are always the same.

Each SM contains 8 streaming processors (CUDA cores) and each of these contains 2048 registers (see Figure 2.1(a)). Each SM also contains 16KBytes of shared memory, 8KB of read-only cache memory for constants and 8KBytes of cache memory for textures. The most important aspect about these SMs is that they work according to the Same Instruction Multiple Thread (SIMT) paradigm. This means that unlike a multicore CPU, the cores of each SM on the GPU do not work independently but instead each core on the GPU executes the same instruction simultaneously on a different thread handling possibly different sets of data. There is a minimum of 4 threads per core, leading to groups of 32 threads per SM (a warp). In the event that there are not enough threads to allocate 32 threads to each SM, empty threads are created that do nothing. Threads (or warps) run in the SMs in groups called blocks. Blocks are assigned to a specific SM and are also organized in groups to form a grid.

The successor of the G80 is the GF100 or Fermi chipset [4]. This new architecture resembles the G80 in that it still relies on SMs and SPs. The major differences are that each SM will contain four times as many cores as a G80 SM (totaling 32 cores) and a Fermi SM possesses two instruction dispatch units and two warp schedulers, allowing two warps to run concurrently on each SM (see Figure 2.1(b)). Furthermore the on-chip memory is increased. The Fermi architecture allows
for the existence of L1 and L2 cache, providing the ability to cache global memory accesses for later use, instead of caching only texture reads. The L1 cache can be configured to be either 16KB (with 48KB shared memory) or 48KB (with 16KB shared memory).

2.1.2 AMD FireStream architecture

NVidia’s main competitor, AMD, also began steering towards general purpose computation in their graphics cards. The main difference between both manufacturer’s offers is at the low level architecture. In AMD GPUs the equivalent to the stream multiprocessor is the Compute Unit (CU) which contains several stream cores (SC). These stream cores are Very Long Instruction Word (VLIW) processors of 4 or 5 segments, depending on the graphics card model. [5]

Each segment in the stream core is called a processing element and is the major difference from NVidia’s approach. Whereas in a CUDA core only one instruction is executed at any time, in a stream core one instruction holds 5 different operations and there are several pipelined instructions.

These VLIW processors allow each core to execute 4 to 5 instructions concurrently, although the total number of parallel instructions executing might be lower depending on whether they are single or double precision instructions or if they are special functions. While a single CUDA core is capable of executing any single or double precision operation in a single clock cycle, delegating any complex functions to a dedicated unit inside the SM, a stream core needs to use more than a single processing element to execute operations of double precision or other advanced functions, such as sine and cosine.

As in NVidia GPUs, there is also a minimum number of threads running on each compute unit. On the AMD GPU this number is defined by the hardware and although in every NVidia product so far it has always been 32, in the case of AMD products it varies. Each group of these threads is called a wave front and, just like in a warp, each instruction in a wave front thread is executed in lockstep.
2. General Purpose Computation on Graphics Processing Units

2.2 GPGPU performance issues

GPUs offer high performance due to their multiple processors being able to run several threads at once. This means that any task running on the GPU should be highly parallelizable or will incur in penalties to its performance. Their processing model and hardware impose some limitations to their performance that must be taken into account when programming for these devices under the penalty of severe performance degradation.

Whether the target is a NVidia or an AMD product, the underlying issues are common and so are the solutions. The main differences are mainly in the nomenclature and some hardware specific limits such as warp/wave front size. With this in mind and for ease of understanding the nomenclature and hardware limits used herein will be those corresponding to NVidia’s platforms.

2.2.1 Warp divergence and SIMT paradigm

A thread block may run as many as 512 threads and a minimum of 32, as previously described. These limits exist because the number of registers and shared memory on each SM are limited and must be divided amongst all executing threads. These threads are grouped into sets of 32 named warps. At any moment in time, all the threads in a warp are executing the same instruction. This happens because the SM only has one instruction decoder for all its cores (two in the case of GF100). Threads in different warps however are free to execute differently.

![Figure 2.2: Warp divergence.](image-url)

The SM also schedules which warp is executing and if the current active warp is waiting for an access to memory, the SM is able to switch execution to another warp ready to execute. This content switching has no overhead, since each thread and block has its own registers and shared memory. [6]

The fact that all threads in a warp must simultaneously execute the same instruction presents a major challenge to optimizing GPU code since branches in the code might lead to different execution paths in threads belonging to the same warp. When this happens we say that the threads diverge. [7]

In simple cases, the compiler is capable of automatically unrolling conditional instructions (ifs, switch’s) so that all threads calculate the predicate and then run multiple predicated instructions. In more complex cases, the compiler inserts code to verify that all threads take the same path (warp voting).

Divergence might lead to significant efficiency losses in the SIMT model, since when threads diverge they must be run sequentially (see Fig. 2.2), thus losing all benefits from multiple processing cores.
2.2 GPGPU performance issues

2.2.2 Memory access

Another big issue affecting performance is memory access in threads. Although the GPU provides high memory bandwidth (the G80 has six 64-bit memory interfaces), the downside is that the global memory latency is in the order of hundreds of clock cycles.

Device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e. whose first address is a multiple of their size) can be read or written by memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8 since only 4 out of the 32 bytes will be used.

To take advantage of the GPU's large bandwidth and due to how memory accesses work in the GPU, threads in the same warp should access memory positions neighboring to each other so that as much data as possible can be read in each memory access and reduce the number of unused words that are transferred. Memory accesses can be:

- Contiguous: Each element of the warp accesses a neighboring array element (Fig. 2.3(a));
- Coalesced: Contiguous access in which the first element is naturally aligned (Fig. 2.3(b));
- Scattered or unaligned: Non-contiguous access (Fig. 2.3(c)).

![Contiguous access](image)

**Figure 2.3:** Memory access of several threads in a warp simultaneously.
Global memory instructions support reading or writing words of size equal to 1, 2, 4, 8, or 16 bytes. Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size). If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. It is therefore recommended to use data types that meet this requirement for data that resides in global memory.

To achieve high memory bandwidth for concurrent accesses to the shared memory area, this memory is divided into equally sized modules/banks. A memory access that spans $n$ modules yields an effective bandwidth that is $n$ times the bandwidth of a single bank. On the other hand, multiple accesses to the same bank must be serialized.

Shared memory banks are organized into successive 32-bits words, assigned to consecutive banks. There are 16 banks and memory accesses are coalesced by half-warps (when a warp requests data from memory, two reads will be requested, one for the first 16 threads in the warp and a second one for the last 16 threads in the warp) so that if each thread accesses a word in a separate bank, there are no conflicts.

Figure 2.4 illustrates the previous concept. If all the threads request a word from a different memory block then all those words are returned simultaneously. However, if more than one thread tries to access the same memory block then memory read instructions must be serialized.

Texture memory is cached on the SM after it is read from global memory and it is optimized for 2D locality. Hence, threads of the same warp that read addresses close together will achieve best performance, by taking advantage of this caching mechanism.

The GPU also possesses 64KB of constant memory. Reading from the constant memory is as fast as reading from a register, provided that all threads read the same address. Otherwise the reads must be serialized, generating one memory access for each distinct address. The registers in each core have a read-after-write latency of 24 clock cycles meaning that a recently written value is only available for reading 24 clock cycles after it has been written. This latency is hidden by the SM scheduler when there are at least 192 threads (6 warps), since warps can be rotated out while they wait for the register values to become available.

### 2.3 GPGPU programming languages

A GPGPU application has two components, the CPU code and the GPU code. The CPU code, as the name implies, runs on the CPU and is the part of the application responsible for initializing
the device, allocating memory, copying data to and from the GPU memory and also launching the GPU code on the device. The programmer specifies through the CPU code how many threads should be launched on the GPU and how they should be organized.

The GPU code contains functions, named Kernels, that will run on the GPU. Only one Kernel can be active on the GPU at a time (the GF100 allows for two concurrent Kernels due to its double warp scheduler and instruction dispatcher) and all the threads running at that time will be executing the same code. A set of threads constitutes a thread block and a set of blocks constitutes the thread grid. The number of threads within a block and their layout is configured by the programmer in the CPU code before the Kernel is launched. This layout is purely logical and has no impact on the execution of the Kernel other than the identifier it is given. A block can be tri-dimensional, meaning the threads forming the block will have x, y and z coordinates. Similarly a block can be two-dimensional (the z coordinate of a block is always zero).

Figure 2.5 illustrates the main cycle in a GPGPU application. The whole processing, in a GPGPU application, revolves around a main cycle that, after initializing the device: i) copies data to be processed into the GPU, ii) executes the Kernel to process the data and iii) copies the results from the GPU memory back into the host's memory. Once there is no more data to process the GPU memory is deallocated.

There are several programming languages suitable for GPU programming, the most common being CUDA, FireStream, OpenCL and Microsoft’s Direct Compute.

### 2.3.1 NVidia CUDA

CUDA [8] is the hardware and software architecture for issuing and managing computations on the GPU. These computations are described in functions (Kernels) which are executed in parallel. There are two ways to use CUDA. One is C for CUDA, where the Kernels are defined as standard C functions and are then transformed into GPU byte code by NVCC (NVidia’s C Compiler). As the name implies, C for CUDA is based on C with a few extensions. It contains routines for memory allocation, data transfer to and from the device, error checking and timing. It also possesses a special syntax for launching Kernels, where configuration parameters (grid dimension, block dimension, shared memory per block) as well as passing arguments are defined.

The second method to use CUDA is through the CUDA Driver API. This has the benefit of not being dependent on runtime libraries and provides more control over the devices, at the cost of creating verbose code of difficult debugging.
2. General Purpose Computation on Graphics Processing Units

2.3.1.A Memory and variables

On a GPGPU application, each thread block runs on a single SM. If the number of blocks exceeds the number of SMs then each SM will handle multiple blocks as long as there are enough registers and shared memory. Otherwise, the extra blocks will wait in a queue, until there are enough resources for them to run.

Besides the registers, the shared memory and the constant and texture caches, there is an additional memory space available to GPU applications: the onboard memory (device memory).

Unlike the other memory devices the onboard memory is not located in the SM chip but in individual DRAM chips. As a consequence it is typically large (in the order of megabytes or gigabytes), but it also has a much higher latency than its on-chip counterparts. The different types of memory available and their scope are represented in Fig. 2.6.

Hence, during its execution a GPGPU application handles 4 kinds of variables which vary according to the type of memory they are stored in:

- **Global variables**: allocated and copied to the device memory by the host
- **Local variables**: private to each thread; scalars are usually assigned to registers while vectors reside in the device memory
- **Shared variables**: declared by the ‘shared’ prefix in the Kernel. Located in the shared memory, therefore limited to 16KB (divided amongst all blocks) but very fast. Used for communication between threads.
- **Constants**: defined by the ‘constant’ prefix. Their values are set by the host and are read-only by the Kernel. Useful to avoid wasting registers on essential constants.

Kernels run on the GPU and as such can only access the GPU memory (device memory), as opposed to CPU memory (host memory). Consequently, in a given GPU application, before the Kernel can be actually executed data must first be placed in the device memory, by copying it from the host memory. This memory can be allocated in one of two ways:

- **Linear memory** - similar to standard host memory
- **CUDA arrays** - opaque memory layout, optimized for texture fetching

The memory allocation method must not be confused with texture mapping. A texture can be mapped (or bound) to either linear memory or CUDA arrays. It should be noted that the GPU can only access the device memory through texture fetching (reading Texels) after it has been bound to a texture by the host. The main difference is that once read the texels are stored in the texture cache and are available for later processing unlike data in linear memory which is discarded after it has been used (in the GF100 there are already present caching mechanisms that work on global memory accesses).

2.3.1.B Compiling

As was previously mentioned, a GPGPU application has both CPU and GPU components. Therefore, it needs both a CPU code compiler and a GPU code compiler. NVCC, NVidia’s C compiler for CUDA, will call a C compiler for the CPU code, compile the GPU code into GPU byte code and later link both together to create the executable (see Fig. 2.7).
Regardless of which method is used there are some essential variables always available to all the threads:

- **gridDim**: the x and y dimensions of the grid. Grids cannot be tridimensional therefore z is always 1.
- **blockDim**: the x, y and z dimensions of the block. (3-dimensional vector)
- **blockIdx**: the index of the block within the gridDim (3-dimensional vector)
- **threadIdx**: the index of the thread within the block (3-dimensional vector)

By using these 4 special variables each thread is capable of calculating a unique threadId, which can be used to select which data it should operate on.

### 2.3.2 ATI FireStream SDK

AMD first released its Stream Computing SDK, in December 2007. This SDK includes “Brook+”, an AMD hardware optimized version of the Brook language developed by Stanford University, itself a variant of the ANSI C (C language), open-sourced and optimized for stream computing. An important part of the SDK, the Compute Abstraction Layer (CAL), is a software development layer aimed for low-level access, through the Close-To-Metal (CTM) hardware interface, to the GPU architecture for performance tuning software written in various high-level programming languages. In August 2011, AMD released version 2.5 of the ATI APP Software Development Kit, which includes support for OpenCL 1.1, a parallel computing language developed by the Khronos Group.
2. General Purpose Computation on Graphics Processing Units

2.3.3 OpenCL

OpenCL [9] is an open source project, developed by the Khronos group, as part of an effort to create a standard adopted by the major hardware manufacturers.

Like in CUDA, in OpenCL a GPGPU application is divided into two blocks: one running on the host and a second one running on the device. In OpenCL this device does not necessarily have to be a GPU, it can be any processing unit that has an OpenCL implementation.

In OpenCL, a compute device is composed of one or more compute units, each containing one or more processing elements. For ease of comparison with CUDA, the compute device can be considered the GPU, which is composed of several multiprocessors (compute units) with various cores (processing elements).

Considering that OpenCL can be executed in several different types of devices, of different manufacturers, each with varying capabilities, there are three important factors to keep in mind in an OpenCL system:

- The platform version
- The version of the device
- The supported version of the language

These factors are relevant because, just like with CUDA, the hardware and software are constantly evolving and the OpenCL application must ensure that the current device version is capable of running the requested operations.

2.3.4 Microsoft Direct Compute

With DirectX version 11, Microsoft introduced Direct Compute. Although previous versions of DirectX already allowed access to the GPU through the Direct3D API, which was specialized in 3D, the introduction of DirectCompute allows the development of generic applications for the GPU, as it is more directed to general purpose computing.

While most of the previously described concepts still apply, there are a few differences in the way this API was implemented. Nevertheless the main aim of DirectCompute was to extend a previously existing API, Direct3D. As with OpenCL and CUDA, most differences are simply nomenclature changes.
3 DNA Alignment Methods

Contents

3.1 Local and global alignment ........................................... 16
3.2 Dynamic programming DNA alignment .......................... 16
3.3 Indexed search .......................................................... 18
3.4 Summary of DNA Analysis tools .................................... 26
DNA alignment is concerned with the problem of understanding DNA and discovering the information stored within it. This problem is mainly solved by comparative analysis: by comparing the DNA from two or more similar organisms, the differences between these organisms can be related to the corresponding differences in the underlying DNA. This analysis, however, can be done in several different ways, each yielding different results relevant to different research efforts.

Approximate string matching is a variant of the string matching problem, where small differences in the sequences are considered acceptable, thus returning zones of the string which are approximately equal instead of exactly equal. The closeness between an exact match and the approximate match can be measured by the number of differences between them, namely the number of mismatches and insertions or deletions. The total number of these differences is commonly referred to as ‘edit distance’ or ‘Levenshtein distance’.

Closely related to the previously mentioned distances is the alignment score. Given a specific alignment its score is calculated by adding the value of the matching elements and subtracting the value of the mismatching ones. It is generally considered that higher scores represent better alignments. Therefore, the alignment score is considered an indicator of the degree of similarity between two sequences.

### 3.1 Local and global alignment

Before tackling the problem of DNA alignment, it is important to differentiate between local alignments and global alignments. One common way to align DNA strands is local alignment. Local alignments are more useful for dissimilar sequences that are suspected to contain regions of similarity or similar sequence motifs within their larger sequence context. Starting with two or more sequences, the alignment algorithm will search for the zone that maximizes the alignment score. Since mismatches and gaps have a negative score, local alignment results never begin with either but they might contain some of these occurrences if the gain to connect two highly similar subsequences outweighs the penalty for including the mismatching section in the alignment.

The alternative to local alignment is global alignment. Mostly useful for sequences that are very similar and roughly equal in size, these kinds of alignments will attempt to match every residue in every sequence and, as a result, the end alignment might include gaps or mismatches at the beginning or end of the sequences. When using global alignment the algorithm still looks for the highest possible score, but unlike in local alignment, it will not try to maximize it by dropping sections of the DNA strand from the alignment effort.

### 3.2 Dynamic programming DNA alignment

Dynamic programming approaches to solve DNA alignment problems typically resort to calculating all possible solutions and picking the one with the best score. As a consequence, they always conduct to the best possible solution, at the cost of a significant processing time.

#### 3.2.1 DNA analysis using strings

DNA can be viewed from a computational point of view as a string composed of only four different characters, each representing one of the four unique bases present in DNA. These
3.2 Dynamic programming DNA alignment

![DNA Alignment showing portions between query and reference that match.](image)

Four bases are: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). The standard way of representing strings, i.e. a consecutive chain of characters, is not very efficient for string matching problems. Solving the string matching problem using strings involves checking one of the strings (the reference string) for the locations where the first character of the second string (the query string) appears and in those locations apply the same method for the second character and so on until all characters of the second string have been used. This method can become quite time consuming when large strings are used, such as the ones typically present in DNA alignment problems. Due to this disadvantage, alternative ways of representing strings were created to solve string matching problems (either exact matching or approximate) when very large strings are considered. The most common of these representations are suffix trees and suffix arrays.

One of the most popular approaches to solve the string matching problem is the application of dynamic programming methods. Two particular algorithms are worth referring: the Needleman-Wunsch and the Smith-Waterman, as described below.

3.2.2 Needleman-Wunsch

The Needleman-Wunsch algorithm [10], proposed in 1970 by Saul B. Needleman and Christian D. Wunsch, was one of the first applications of dynamic programming to biological sequence comparison. This algorithm uses the string representation for DNA and builds a matrix of $n$ by $m$ dimensions (Fig. 3.2(a)), where $n$ is the size of the reference string and $m$ the size of the query string. The algorithm builds this matrix, recursively, by applying the rules in Figure 3.2(b).

In these formulas $F(i, j)$ represents the maximum score of aligning $X_i$ against $Y_j$. The function $s(X, Y)$ is the score function that determines the match value of each possible pair (in the example $s(X, Y) = 1$) and $d$ is a constant representing the cost of introducing or extending a gap ($d = -1$ in the example).

As the used sequences increase in size, this algorithm becomes too slow and impractical to use. However, it yields the best result for global alignments and can take into account mismatches and deletions or insertions.

3.2.3 Smith-Waterman

One of the most well known dynamic programming algorithms for local DNA alignment is the Smith-Waterman algorithm [11]. It was first proposed by Temple F. Smith and Michael S.
3. DNA Alignment Methods

![Alignment matrix for Needleman-Wunsch](image)

(a) Alignment matrix for Needleman-Wunsch

(b) Needleman-Wunsch formulas

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(X_i, Y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

Waterman, in 1981. This algorithm is a variation of the Needleman-Wunsch. The main difference between the two is that negative score cells in the matrix are not allowed, thus having their value reset to zero, which renders the local alignments visible.

Keeping up with the evolution of computing power and programming paradigms, some dynamic programming algorithms have been adapted for parallel execution, in order to take advantage of the most recent technologies. This is the case for the Smith-Waterman algorithm which has several parallel implementations either in SMP processors or even in GPU platforms. [12] [13]

3.3 Indexed search

Contrasting to the approximate string matching techniques previously described, exact matching is another technique sometimes used during DNA sequence alignment. In fact, since DNA is often represented as a string, the problem of exact string matching can be adapted to the DNA alignment problem. By taking the two strings and finding the portions which are common between them, it is possible to find the high similarity zones in the DNA strands that are included in the best alignments. To achieve the final alignment, it is necessary to arrange these zones in the way that generates the highest score.

Algorithms which implement such search procedures take advantage of specialized data representation schemes based on indexes, as well as searching methods particularly adapted to these representations, in order to discover a solution in a smaller computation time.

3.3.1 Suffix tries and suffix trees

A suffix trie is, in essence, a directed acyclic labeled graph where each edge/transition between states represents a character in the string. Paths in the graph that do not have branches represent a unique substring of the text. These paths can be converted into a single node, where the edge leading into it is labeled with the substring it represented before the merge. By doing this, several nodes can be represented by a single one, thus reducing both the edge and node count in the final graph. This kind of specialized suffix-trie is called a Patricia Trie/Tree or suffix tree.

Figure 3.3 illustrates the differences between a suffix trie and a suffix tree. While in a suffix tree (Fig. 3.3(a)) every character corresponds to a state transition in a suffix tree (Fig. 3.3(b)) a state transition might represent a string of characters. This results in gains in space since there are less nodes and edges in a suffix tree.

In either a suffix trie or in a suffix tree, exact string matching is trivial. To discover if a particular
pattern exists within a larger text, all that is necessary is to travel through the graph created by the trie by using the pattern as the input sequence. If the entire pattern can be used, then there exists such a substring in the original text.

Exact matching using suffix trees follows the following procedure:

1. Start at the root node;
2. Pick the edge that starts with the same character as the next character in the query string;
3. Match as many query characters as the length of the edge;
4. Jump to the destination node and pick a new edge following the same rule as #2.

If the algorithm is able to match all the characters in the query string then it exists in the reference string. To find out where exactly it is located, it is only necessary to find all the leaves reachable from the node where the matching ended. If the match ended on an edge, the considered node must be the destination of the edge. Even when not all the characters in the query string are matched, it is still possible to obtain a partial match corresponding to the number of characters it was possible to match.
3. DNA Alignment Methods

If an exact matching algorithm tries to match the sequence 'ISSI' against the suffix tree generated from the sequence 'MISSISSIPPI$' (Fig. 3.4) it will start from the root node (Step 1) and pick the edge that starts with the first character of the query (Step 2): 'I'. That edge represents a single character and so the algorithm matches it (Step 3) and jumps to the edge's destination node (Step 4). The process repeats for the string 'SSI' and the algorithm finishes at node '5'. At this point to find out all the matching locations all that is required is to apply a Depth First Search (DFS) from node 5 of the suffix tree onwards, which yields leafs 'E' and 'F'.

To finalize the matching algorithm it is just necessary to calculate the actual match position from the leafs. This is done by doing a simple calculation for each leaf encountered:

\[
\text{match start} = \text{reference length} - (\text{matched bases} + \text{remaining bases until end})
\]

The formula, essentially, calculates the length of the path taken from the root node until the end of the leaf. Since this path represents a suffix of the reference sequence, by knowing its size it is possible to calculate where it begins.

Each node in a suffix tree has an edge-path associated with it. The edge-path is the concatenation of the labels of every edge that leads from the root node of the suffix tree to the node.

Suffix tree nodes are connected with a network of suffix links. The destination node of a suffix link has the same edge-path as the origin node except for the first character which has been removed. This allows an algorithm to find the largest common subsequence by performing the exact match as described before, but instead of terminating when a mismatch is found, it follows the suffix link and continues the matching from the same character.

One example of a DNA alignment tool that makes use of suffix trees is MUMmer [14]. MUMmer is a multiple step heuristic alignment tool which starts by creating a suffix tree of the reference sequence. Once the suffix tree is completed, MUMmer adds more states to the suffix tree, representing the query sequence. After this is done any node which has leafs from both sequences represents a matching subsequence. This matching subsequence is passed on to a second phase where the alignment is computed.

There is a parallel version of MUMmer using GPUs, the MUMmerGPU [15], which uses the traditional exact matching method using suffix trees to simultaneously match multiple queries to a single reference string. Furthermore, MUMmerGPU makes use of the suffix links to obtain the longest common subsequence.

MUMmer is composed of multiple tools and does not perform only string matching. When the matching part of the algorithm is completed, MUMmer takes the results and sorts them in such a way as to obtain as much coverage of the reference string as possible (through the ‘gaps’ or ‘mgaps’ tools) and finishes by filling in the gaps between the sequences to generate the alignment (through the ‘NUCmer’ or ‘PROmer’ tools).

3.3.1.A Online suffix tree construction

There has been a significant effort to develop efficient methods to create a suffix tree in a reduced amount of time. Some of these efforts led to the development of suffix tree construction algorithms by Edward McCreight and Esko Ukkonen. Ukkonen’s algorithm [16] is of particular interest, since it runs in linear time and processes the string from left to right, therefore allowing to add new characters to the string during the construction of the suffix tree and consequently allows the suffix tree to be expanded after it has been created.

Ukkonen’s algorithm works by creating a suffix-tree for the first character in a string and then
expanding the tree by adding the next character. In general terms, the suffix tree for a string of
length \( i \) \((T_i)\) is created by adding the \( i \)-th symbol of the original text to the suffix tree \( T_{i-1} \) which
must have been created in a previous phase of the algorithm (up to \( T_1 \)).

It is clear that for a text of size \( n \) Ukkonen’s algorithm will have \( n \) phases. These phases are
further divided into \( j \) extensions, where \( 1 \leq j \leq n \). In each of these extensions a different suffix is
added into the tree. In simpler terms, the algorithm has \( n \) phases and in each of them it processes
a prefix of the whole text. This processing is done in \( j \) extensions, each adding a suffix into the
tree.

The actual work is done inside these \( j \) extensions by following a couple of simple rules:

1. If a suffix ends at a leaf in the tree, then to extend this suffix in the future it is only needed to
   add the new character to the end of the label for the leaf.
2. If a suffix ends at a node but there is no edge leading out of that node labeled with the
   character required to extend the suffix, then a new branch (leaf) must be added to the node
   and the edge leading to it must be labeled with the new character.
3. If a suffix ends at a node and there is already an edge leading out of the node labeled with
   the required character, no further work needs to be done.

The advantage of Ukkonen’s algorithm is the fact that it runs in \( O(n) \) time. To achieve this,
some special tricks are necessary, while processing the text. One of the easier tricks to under-
stand is that edge labels should not have the actual substrings in them but rather the start and
finish indexes of the strings. This makes creating and changing labels much simpler and faster,
taking constant time, since only two numbers must be changed in any circumstance and occup-
ying potentially much less space.

Next, it should be noted that the algorithm has no mechanism to add child nodes to a leaf. It
can only extend the labels leading to them. When a leaf is created it will always be a leaf (since it
cannot have any child nodes added to it). One important point to note is that a leaf always ends
at the last character of the sequence. By simply labeling it (setting the indexes for the edge label)
from \( m \) (the current phase) to \( i \) (the text size) it is not needed to care about leaf extensions in
future phases of the algorithm. When the algorithm finishes, the label leading to it will have a
suffix of the text as label.

Figure 3.5 illustrates the main point behind Ukkonen’s Suffix Tree construction algorithm: Split-
ting edges and adding nodes. The suffix tree in fig. 3.5(a) represents the state of the algorithm
after it has processed the prefix ‘MISS’. At this point the algorithm adds ‘MISSI’ and all its suffixes
to the tree. When ‘SI’ is added the edge starting with ‘S’ must be split in order to allow for the new
suffix (fig. 3.5(b)). Afterwards a new edge starting with ‘I’ is added to the recently created node
(fig. 3.5(c)).

Another important detail in the algorithm is that in the \( j \) extensions of each phase the suffixes
are added in order of diminishing size. This allows to detect that sometimes, when adding small
enough suffixes, they are already present in the tree (and so will any smaller suffixes in the same
phase). This fact, together with rule #3, enables to immediately skip to the next phase of the
algorithm.

An important concept appears at this point: the endpoint. Whenever rule #3 is activated, an
endpoint has been found and the current phase can simply end. The endpoint has an interesting
property in that it represents the first character of a suffix already present in the suffix tree. By
analyzing this, it can be seen that this means that all the characters from the endpoint forwards belong to a repeating pattern in the text. In future phases of the algorithm, the following characters will either keep following this previously processed path or will deviate from it. By keeping a record of where the endpoint was in the previous phase, it is possible to know where to insert the next character (from the next phase) should it deviate from the repeating pattern. This 'insertion point' is called the active point.

The effects of the endpoint and the active point are also visible in figure 3.5. After adding 'MIS' to the suffix tree the algorithm obtains the suffix tree in figure 3.5(a). The next phase of the algorithm will add 'MISS' into the tree. However, since there is already an edge starting by 'S' in the tree, on the 4th extension of the phase (adding the 4th character, 'S') the algorithm does not need to perform any work and marks that point as the endpoint. This endpoint becomes the active point for the next phase (the active point being the sequence 'S'). When the algorithm attempts to insert 'MISSI' into the suffix tree in the next phase it will examine the active point and discover that the edge starting by 'S' needs to be split since the following character is not 'I' (the algorithm is in the 5th phase and the 5th character to insert is 'I').

As was mentioned previously, edges are labeled through two indexes referencing the string and not a copy of the subsequence. This makes finding and updating the active point somewhat complex, since it might refer to an implicit node (a node that is not actually represented as a physical node in the tree; it is, instead, a point somewhere along an edge). To work with the active point (which is comprised of a node and an index), the algorithm requires it to be in canonical form, which is the same as saying that the node to which it points must be the closest physical/explicit parent node and the index must indicate how many characters into the edge leading out of the node the active point is. Once the index for the endpoint has been found, it is changed to canonic form to obtain the active point. Canonizing is nothing more than walking down the tree starting at some node (the current active point's node) and following the path dictated by the suffix that starts at the endpoint. In order to accelerate this process and knowing that each node cannot have two edges leading out of it whose label starts with the same character the algorithm makes use of a trick named "skip/count". This trick merely compares the first character of the label of the edges against the current starting character of the suffix to be followed. If it is a match, the next \( n \) comparisons are skipped, where \( n \) is the length of the edge, since it is already known that this edge must be the right one (by rule #3 the suffix MUST already be in the tree so there is no need to actually compare all characters).

Every phase in Ukkonen's algorithm starts by testing whether the active point meets the re-
3.3 Indexed search

3.3.2 Suffix arrays

A simple and straightforward way of representing a reference string, while at the same time retaining enough structural information to allow a fast search for a specific query string, is a list of all the suffixes present in the reference string. By creating an empty list and adding suffixes of increasing length to it (starting with adding simply the last character of the reference, then the last two characters and so forth until the whole string is added to the list) and afterwards sorting it alphabetically, exact string matching problems become trivial.

By only checking suffixes that start with the same character as the search pattern and limiting the list to suffixes of equal or greater length than the pattern, it is possible to limit the search to a small subset. Such a list of suffixes can be considered a suffix array instead of a suffix list, since the total number of elements is known beforehand (one element for each character in the string) and never changes. Also, considering that every element of the list references to a suffix, they can be implemented as pointers to the start of the suffix or indexes, instead of the substring itself. When combined, it is these two factors that allow for a very space efficient representation of a string, while at the same time providing sufficient additional information to allow for a fast search [17].

Suffix arrays can be used for DNA alignment in a similar way as suffix trees. The main difference being how the matching between reference and query strings is achieved. A suffix array is a sorted list of all the suffixes present in the reference therefore discovering whether a query string is present in the reference string is simply a matter of applying a search algorithm to the array. Without auxiliary structures to increase the search speed, one of the fastest methods is binary search. This method repeatedly divides the array in half until it contains only elements that match the query string.
Figure 3.6 illustrates the two steps in the construction of a suffix array. First all the suffixes of the reference sequence are added into the array (fig. 3.6(a)) and then the suffixes are sorted lexicographically (fig. 3.6(b)) resulting in a structure suitable for searching.

Although suffix arrays require less space than suffix trees this comes at the cost that some advanced searching methods are not easy to implement, such as finding the largest common substrings or partial matching of the first characters of the query. There are modifications of the basic suffix array (often denoted as 'enhanced suffix arrays') which were developed to hold additional information about the string structure and are better suitable to solve some problems that cannot be easily addressed using the plain version. [18] [19]

### 3.3.3 Hash tables and q-mers

A very fast alternative to suffix trees and suffix arrays are hash tables. Creating a hash table with a key of length $q$ (a q-mer) allows any subsequence of length $q$ of the query string to be immediately found, provided it exists in the reference string. There is, however, a big penalty for this increased speed paid in the amount of space required to store the hash table. If such an approach is used to represent a DNA string, each DNA base can be represented by a minimum of 2 bits. This means that a key of length $q$ has $2 \times q$ bits and there are $2^{2q}$ possible entries. Adding one DNA base to the hash key requires adding 2 bits and multiplying the hash table size by a factor of 4. As such this method is best suited for very short key lengths.

Hash tables are the fastest exact matching method, by simply requiring one operation to find all occurrences of a query string of size $q$ in the reference string. By creating a hash table with a key of length $q$ (q-mers, where $q$ is the size of the query) and assigning each set of $q$ characters of the reference string to the correct hash table entry, exact matching becomes trivial. Using the mentioned hash table, finding if a query string exists in the reference string requires only that the hash key for the query sequence be calculated and the corresponding element from the hash table be retrieved. This approach, however, takes a large amount of space and is not suitable for approximate string matching.

BLAST [20] is an example of a tool which makes use of hash tables to perform DNA alignment. BLAST starts by creating a list of words, contiguous q-mers, present in the reference string. These words have length $q$. Therefore, there are $l - q + 1$ words (where $l$ is the length of the reference). BLAST uses the q-mers, in conjunction with an hash table, to quickly find the occurrences of small portions of any query string in the reference. Afterwards, it uses all the found occurrences as seeds to isolate possible high similarity locations, upon which it performs a more exhaustive DNA alignment algorithm, such as Smith-Waterman [11].

### 3.3.4 Burrows-Wheeler transform

The last string indexing method that is analyzed in this chapter is the Burrows-Wheeler Transform (BWT) [21]. This method, created in 1994 by Michael Burrows and David Wheeler, was primarily intended to be a fast data compression algorithm. However, it is suitable and has been adapted to DNA representation and multiple short read alignment problems. Earlier algorithms addressed the problem of aligning multiple small DNA strands against a whole genome through hashing. However, the Burrows-Wheeler transform allows the solution to be achieved with a much smaller memory footprint.
This representation is created by first generating a matrix BWT of every possible rotation of the original text (fig. 3.7(a)) and afterwards sorting these rotations lexicographically (fig. 3.7(b)). In the BWT matrix each row represents one rotation of the original text. The last column of the matrix, composed of the last character of each row/rotation in the sorted list composes the representation of the string T, BWT(T). Clearly, there are as many possible rotations as there are characters in the original text, and by taking one character from each row it can be observed that the final representation takes up as much space as the string it represents.

It is interesting to note that since the BWT matrix is sorted lexicographically and each row starts with a different suffix of T the BWT matrix can be seen as closely related to the suffix array representing T. However, this matrix has a property called 'last first (LF) mapping'. The \( i \)th occurrence of character \( X \) in the last column corresponds to the same text character as the \( i \)th occurrence of \( X \) in the first column.

Hence, with the Burrows-Wheeler matrix (the set of all rotations) it is possible to find exact matches of a query string within the reference string by calculating matrix row ranges for successively longer query suffixes. If the range becomes empty, then the suffix does not appear in the text.

Using the Burrows-Wheeler transform for exact matching problems relies on the fact that the transformation is reversible. The "last first mapping" property of the transformation underlies on any algorithm that uses the BWT index to navigate or search the text.

Hence, by using the 'last first mapping' property of the Burrows-Wheeler matrix it is possible to revert the process and from the Burrows-Wheeler transform generate the original matrix. Since the BWT matrix can be generated from the characters given by BWT(T) and the matrix is all that is necessary for exact string matching, then the problem can also be solved with only BWT(T) and generating the BWT matrix as necessary.

Using the BWT to perform exact string matching is simply a matter of searching for the first and last rows in the matrix that begin with the given query sequence. Since the matrix rows are sorted lexicographically, any rows in the range between the first and last will also verify that condition.

If the BWT matrix is carefully studied it can be seen that the suffix array is present within it. By using the last-first property of the Burrows-Wheeler transform to obtain the Burrows-Wheeler matrix it is also possible to obtain the suffix array and perform exact string matching through it.

By combining the Burrows-Wheeler compression algorithm with suffix arrays it is possible to obtain a compressed suffix array which acts much as an index to the original sequence [22] [23].
3. DNA Alignment Methods

### Table 3.1: List of relevant algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Platform</th>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blast</td>
<td>CPU</td>
<td>Indexed</td>
<td>Alignment</td>
</tr>
<tr>
<td>Bowtie</td>
<td>CPU</td>
<td>Indexed</td>
<td>Matching</td>
</tr>
<tr>
<td>SSearch</td>
<td>CPU</td>
<td>Dynamic</td>
<td>Alignment</td>
</tr>
<tr>
<td>CUDASW++</td>
<td>GPU</td>
<td>Dynamic</td>
<td>Alignment</td>
</tr>
<tr>
<td>MUMmer</td>
<td>CPU</td>
<td>Indexed</td>
<td>Alignment</td>
</tr>
<tr>
<td>MUMmerGPU</td>
<td>GPU</td>
<td>Indexed</td>
<td>Matching</td>
</tr>
</tbody>
</table>

This index proposed by Paolo Ferragina and Giovanni Manzini is called the FM-Index.

Bowtie [24] framework indexes the reference genome using a scheme based on the BWT and the FM index [25]. By using these two tools, Bowtie aligns short DNA sequence reads to large genomes. After calculating the BWT for the reference sequence, Bowtie, in a series of steps, applies the search algorithm to calculate the range of matrix rows beginning with successively longer suffixes of the query. At each step, the size of the range either shrinks or remains the same. When the algorithm completes, rows beginning with the entire query correspond to exact occurrences of the query in the text. If the range is empty, the text does not contain the query.

#### 3.4 Summary of DNA Analysis tools

There are several other tools that are used for DNA analysis, either using sequence matching or sequence alignment (see Table 3.1). From these tools SSearch provides the best results due to the fact that it executes the Smith-Waterman algorithm. However, its run time prevents it from being used on very large sequences.

A parallel version of the Smith-Waterman algorithm is implemented in CUDASW++. This implementation makes use of GPUs to accelerate the alignment of multiple queries against a single reference sequence. This implementation encounters some limitations in the maximum size of the reference sequence allowed due to the way it uses the GPU memory.

Among the mentioned tools two of them perform only sequence matching: MUMmerGPU and Bowtie. Although Bowtie is mentioned as a short-read aligner, it performs approximate matching on the sequences it analyses by combining suffix arrays with an indexing strategy based on the Burrows-Wheeler Transform. In its algorithm Bowtie allows for a very small number of errors in the sequence. This makes Bowtie very fast but has the disadvantage of not working well for sequences that have gaps or more errors than the implementation allows.

MUMmerGPU is the adaptation to CUDA and GPUs of the first phase of the MUMmer heuristic algorithm. By using suffix trees MUMmerGPU manages to perform approximate matching on a large number of query sequences against a reference sequence. This approach is limited to finding subsequences of the query that exist in the reference, which is useful when used in combination with filtering and seeding phases of an heuristic. MUMmerGPU cannot, however, align DNA sequences by itself.
4

Exact Matching

Contents

4.1 Data representation ........................................... 28
4.2 GPU Initialization ............................................. 33
4.3 Exact matching on the GPU ................................. 37
4.4 Comparison and discussion of different approaches .... 41
4. Exact Matching

The proposed solution for DNA alignment is based on exact and approximate string matching. This chapter describes the study that was done into several possibilities for exact matching, their advantages and limitations, especially when adapted to GPGPU computing.

4.1 Data representation

Considering the previously described limitations and performance issues when using general purpose graphics processing units, the data representation becomes a very relevant factor in the overall performance of the algorithm and the whole DNA alignment procedure.

One other factor to take into account when choosing the data representation is the alignment method to be used, since different methods have different needs and memory access patterns that can be optimized through efficient memory allocation for the data structures.

The considered methods in this work are based on suffix trees and suffix arrays. The former for its speed and ease of creation and matching; and the later for its reduced memory footprint, which allows for larger reference sequences to be used and lower overhead for memory copy.

4.1.1 Reference sequence

Typically, the reference sequence will be a large sequence of DNA bases to be index using a suffix tree or a suffix array. The access pattern in the reference sequence is unpredictable, as it depends on the query being matched. Since every thread will be testing a different query, each thread will have its own memory access pattern. This causes memory access coalescing to be difficult to predict. In such cases, it is beneficial to store the data in such a way as to benefit from caching. At this respect, it is worth to recall that while early NVidia cards only possessed caching mechanisms on textures, the latest versions already have caching mechanisms for global memory accesses.

Regardless of which data representation is used to hold the reference sequence, it must be noted that both suffix trees and suffix arrays only hold indexes to the original sequence. As such, the GPU must hold in memory not only the index structure but also the original sequence. Nevertheless, the DNA sequence being a simple string that has 4 distinct symbols can be represented using a 2-bit encoding reducing the space by a factor of 4.

4.1.1.A Suffix Trees

While constructing a suffix tree, the final size is not known ahead of time (at maximum a suffix tree has \(2^n\) nodes and on average \(1.8n\) nodes), since the total amount of internal nodes and edges is only known when the process ends. Therefore, it is hard to allocate all the necessary memory at the beginning. Although real-time reallocation of memory is possible, it encounters serious problems when a very large block of memory is necessary, which may not be available contiguously and thus fail.

Due to these issues, suffix tree construction is typically done with resort to dynamically allocated memory and memory pointers. Nevertheless, this poses a problem when transferring the suffix tree structure to the GPU memory, since the pointers are only valid with the CPU context and will reference invalid memory positions within the GPU memory banks.
To solve this problem, the suffix tree will be represented as a vector and the pointers to its nodes will be replaced by indexes of that vector. The vector has a fixed size where the number of elements is calculated after the suffix tree is created.

Each element of the vector will represent one edge leading out of a particular node and as such holds the three edge values as unsigned integers:

- The index position for the starting character of the edge in the original reference sequence
- The index position for the ending character of the edge in the original reference sequence
- The destination node given as the index of its first edge in the vector representation of the suffix tree

Additionally, since three 32-bit integers take up 12 bytes of memory, the structure of each vector element is kept aligned in multiples of 4 bytes, by adding one extra integer to each edge as a pad field, thus ensuring that each edge takes up exactly 16 bytes of memory.

During the matching algorithm it will be necessary to pick an edge leading out from a node starting by a specific character. However, in the vector version of the suffix tree there are no edge lists for the nodes, as there were in representation using memory pointers. As such, the process of selecting the appropriate edge or even discovering if such an edge exists becomes more complex as it involves a sequential test of the edges, starting from the node’s first edge, with the added problem that a node can have from two to four edges and there is no boundary defining the end of a node’s edges.

There are several options to solve this problem. The simplest one is to create an additional data structure describing the structure of the tree, namely, how many and which edges each node has. This solution would require an additional integer (or short integer) per node and would increase the complexity of the matching function by requiring the number of edges to be calculated every time the algorithm entered a new node. This would also increase the memory copy overhead, as well as the processing time, considering it would require an additional memory read which, because of its unpredictability cannot, be properly coalesced. However, since in a suffix tree each node must have, at least, two nodes it is possible to reduce the used memory if the node description is stored within the last integer of the first edge (used for padding and memory alignment). The node description is a bit mask representing each of the possible edges for that specific node as either existent (1) or non-existent (0) (fig. 4.1(a)).

Another alternative is to mark each edge as being the last of its node or not (fig. 4.1(b)). This would simplify the process of sequentially searching the edges, since such process could stop as soon as the last edge of the node was encountered. This could also be implemented using the padded space at the end of each edge, requiring no additional memory. The downside of this alternative would be that it would still require all the edges to be read from memory and tested. Additionally, one extra condition would have to be checked for each of them: whether it is the last edge or not.

The last considered alternative proved to be less computationally expensive, although it makes more use of memory. By representing all nodes as always having the maximum number of edges (four) and by lexicographically sorting these edges in the vector by their starting character, the process of fetching the correct edge is as simple as reading the edge at the correct offset and becomes instantaneous (fig. 4.2). However, this introduces the problem that nodes will now have edges that they did not have before. If such an edge is picked, it automatically means that
4. Exact Matching

(a) Flattened Suffix Tree with dynamic edges and node descriptor.

(b) Flattened Suffix Tree with dynamic edges and node delimiter.

Figure 4.1: Flattened Suffix Tree schemes.

A mismatch is present and to guarantee the same result is achieved, such edges are created in such a way that they will always mismatch against any query, by setting its starting and finishing indexes to the sentinel character (last character) in the reference sequence, i.e. a character that cannot appear in the queries.

Figure 4.2: Flattened Suffix Tree with static edges.

The process of converting the traditional suffix tree into a vector is called flattening. This process begins by calculating the number of nodes present in the suffix tree. Since during the construction of the tree an unique identifier is assigned to each node, calculating the number of created nodes is trivial.

After knowing the number of nodes in the tree, space for the vector is allocated and initialized. At this respect, it is important to realize that the vector will hold edges and the nodes are groups of 4 edges starting in indexes multiple of 4. Hence, for a suffix tree holding $n$ nodes, the vector will require $16 \times n$ elements, since each node will have 4 edges and each edge has 4 elements.

A non-existant edge is represented by having both its starting and ending positions set to the last character (the sentinel) in the reference sequence (equal to the length of the sequence) and its destination node set to zero (the root node). This will ensure that any path that leads
4.1 Data representation

to it will inevitably fail its next comparison, since the sentinel character cannot exist in the query sequences.

Since while travelling the tree nodes might be found that have fewer than 4 edges, it is crucial that the edges these nodes lack be identified as non-existent. This is done by initializing every edge on each node as if it were non-existent and afterwards update the existent edges with the correct information.

After knowing the unique identifier for the origin node of an edge, it is trivial to calculate where that specific edge should be stored in the vector. Each node will be stored in the 16 elements starting at index $nodeId \times 16$ and each edge of that node will be stored at index $nodeId \times 16 + firstCharacterOfEdge$ where the characters have values between 0 and 3, based on their 2-bit encoding.

Updating the vector with the correct information is achieved by performing a DFS on the suffix tree, to ensure that every node is reached and every edge is travelled. For each travelled edge, its information on the vector is updated.

A special case are the leaf edges, since they do not have a destination node. In fact, the destination node of leaves is unnecessary since any match that follows the whole leaf will find the sentinel character at the end and fail. However, for control purposes it is useful to represent it anyway. Whenever a leaf edge is identified, its destination node is set to be its origin node. This makes leaf edges and non-existent edges easy to identify since no real edge can jump to the root of the suffix tree.

Program 4.1.1.A is an example of the code used to transform (or flatten) a suffix tree from a structure with pointers into a vector with a fixed number of elements.

---

Program 4.1 Pseudo-code for flattening a suffix tree into a vector.

```plaintext
Flatten(node)
{
    node starting position = node id * 16

    for each possible edge in the node (edge value = 0 to 3)
        edge starting position = node starting position + edge value * 4
        initialize edge as non-existent

    for each real edge in node
    {
        edge value = first character of edge (A = 0, C = 1, G = 2, T = 3)
        edge starting position = node starting position + edge value * 4

        update edge values (start, end, destination node)

        if edge is a leaf edge
            destination node = node starting position
        if edge is not a leaf edge
            destination node = destination node * 16
        Flatten(destination node)
    }
}
```
4. Exact Matching

4.1.1.B Suffix Arrays

As mentioned before, the size and number of elements in a suffix array is known beforehand, as it is the same as the size of the reference sequence. This fact means that the suffix array can be created as a vector without any need for memory pointers and as such it can be copied bit-for-bit from the host memory to the device memory.

When using suffix arrays for exact matching, the memory access pattern depends not only on the query sequences but also on the matching algorithm being used. In any case, the pattern is still unpredictable and accesses to the suffix array cannot be coalesced meaning that the suffix array must be stored in such a way that it can be efficiently cached.

4.1.2 Query sequences

The query sequences can be stored as a single string by concatenating all the queries together and keeping the information of the length and starting position of each query within the string. Nevertheless, this kind of representation is not optimal, since it does not allow for coalescing of memory accesses. Contrasting with the reference sequence, query sequences have a predictable access pattern, regardless of what matching algorithm is used. Matching is done one character at a time, sequentially, and all threads are synchronized meaning that it should be easy to coalesce the query sequence accesses.

To achieve coalesced memory accesses, it is simply necessary to ensure that threads read adjacent positions in memory and that these positions are properly aligned: when thread #0 reads the first element of its query sequence, it should access memory position 0 at the same time that thread #x would read the first element of its query sequence at position x. Also, the first element's address should be a multiple of 32 as it is a 32 bit element.

With the previous details in consideration, the query sequences are stored in a matrix where each individual query sequence is stored as a column (fig. 4.3). To ensure that every row of the matrix is aligned in memory, the matrix must have a width multiple of the warp size, meaning that when the number of queries is not a multiple of the warp size, additional padding columns must be added. The height of the matrix is defined by the longest query sequence, which is not a problem when all queries are of similar length. In the event that queries of very different lengths are present, there is wasted space, as shorter queries will have empty cells at the bottom of their column.

In order to reduce the amount of space that is required to store the query matrix, the DNA queries are stored using a 2-bit encoding, instead of the common 8-bit encoding. This means that each cell of the matrix, being a 32-bit integer, will be able to store 16 bases instead of 4. A beneficial side effect of this encoding is that wasted space in the form of padding (both in columns and rows) is reduced by a fourth.

As it was referred before, in order to accommodate the DNA data in matrix form, some additional information will be required. Whenever queries of different lengths are allowed, an additional vector will be necessary indicating the length of each column. This is necessary because otherwise it would be impossible to detect the end of the query, considering that there is no way to represent a fifth (sentinel) symbol when using the 2-bit encoding scheme. The other parameter that is also necessary to keep is the width of the matrix. Such value is necessary in order to know the stride that should be considered every time the process advances from one row to the next.
and to keep each thread accessing the correct element. However this can be inferred from the number of queries to process and does not need to be explicitly saved.

Figure 4.3: Coalesced query matrix. Each column holds a single query sequence. Each cell holds 16 DNA bases.

4.2 GPU Initialization

Before the GPU function that performs the alignment, (i.e. the Kernel), can be invoked, some preparation is required. Considering that the GPU kernel cannot access the host memory, the device memory must be allocated and the data must be copied explicitly from the host to the device prior to running the Kernel.

4.2.1 Memory allocation

GPU memory allocation is fairly straightforward and works in a very similar way as regular memory allocation. An important aspect to have into account is that the Kernel will need memory space to save the results. Considering that each query is ran by a single thread and will yield a single result, then each thread will require a single memory position to save its result into. These positions must be allocated prior to running the Kernel. Otherwise, an error will occur when attempting to write into a memory position that is not allocated.

Besides the space used to store the results, memory positions must also be allocated for the data structure (suffix tree or suffix array) corresponding to the reference sequence. Considering that the suffix tree is transformed into a vector and the suffix array is innately a vector, allocating memory for these structures is achieved by asking the GPU to allocate a linear block with the appropriate dimensions. However, it is important to recall that the suffix tree or suffix array alone holds little meaning. In fact the reference sequence must also be available to the kernel, since both the previously mentioned structures hold indexes of such sequence.

Lastly, the query sequences matrix must also be copied onto the GPU and as such it also needs its own space. Although it is conceptually considered a matrix, the set of query sequences is in reality stored in a linear memory block. At this point, it is necessary to calculate the total dimension of the matrix, including padding columns, and ask for a memory block suitable to hold such array.
4. Exact Matching

4.2.2 Host to device data copy

The actual memory copy operation is very simple and requires a single command to copy one block of data from the host to the device memory. There is also the option to perform the copy procedure in synchronous or asynchronous mode, with the use of cuda streams. In asynchronous mode the memory copy instruction is not blocking, allowing the CPU to continue processing information while data is, simultaneously, being copied into the device.

Usually, the kernel cannot begin until all the data has been copied to the device. However, since this copy may take a long time to complete, it is sometimes useful to use the asynchronous memory copy, so the CPU can continue the processing while the copy takes place. Additionally, by using the asynchronous copy mode and by partitioning the data into subsets, it is also possible to implement a double buffer scheme, thus significantly reducing the time spent just on memory copy operations.

To implement a double buffer scheme it is necessary to use CUDA streams. These are a feature of CUDA that allows asynchronous instructions to be issued and executed concurrently. If issued on the same stream even asynchronous instructions are executed sequentially. However, asynchronous instructions issued on different streams run as soon as possible. The way double buffering is implemented is by having two CUDA streams and while one is copying data from the host to the device the second is processing previously copied data. There are some important factors and restrictions to be careful of:

- The first Kernel cannot begin processing before the first block of data has finished copying
- The memory used by the first Kernel cannot be deallocated before it has finished processing

Since instructions issued on the same stream are executed sequentially regardless of them being asynchronous or synchronous, ensuring the first restriction is handled natively by the GPU as long as both the memory copy instruction and the kernel invocation are issued on the same stream.

The second restriction must be handled by CPU code. By monitoring the state of the stream currently executing a Kernel the CPU is able to know when the Kernel has finished executing. At this point the memory used by the first Kernel is deallocated.

4.2.3 Kernel launch

After the memory has been allocated and all the necessary data has been copied to the device, it is necessary to invoke the exact matching kernel. For the kernel to work correctly, it needs to have access to certain information which must be passed as parameters:

- The reference sequence and its index (suffix tree or suffix array), query matrix and space reserved for the result (through their addresses, unless they have been allocated as textures, in which case the texture identifier is a global variable of the kernel);
- The number of elements in the flattened suffix tree (if a suffix tree is used);
- The number of query sequences;
- The dimensions of the work grid (number of threads per block and number of blocks).
4.2 GPU Initialization

4.2.3.A Block size considerations

In CUDA, the work grid is defined by the number of blocks and by the number of threads in each block. These blocks and threads can be two or three dimensional, to make data addressing more intuitive. However, for the considered case one dimensional blocks and grid are sufficient.

With this in mind, the remaining question is how many threads should a block have and how many blocks should be created. Considering that a warp is composed of 32 threads and since each block should have several warps to mitigate the effects of global memory access latency, by optimizing towards achieving the best CUDA core occupancy rate it was considered that each block should hold at least 4 warps (128 threads). This value was calculated by taking into account the number of registers present in each SM and the number of registers required by the Kernel (determined at compile time). NVidia offers a tool that calculates the various possible occupancy rates for the GPU according to various parameters (block size, registers per thread, shared memory per block).

The number of blocks should be enough to hold as many active threads as required. Unless some kind of double buffering is being used, the number of active threads equals the number of query sequences to match. Hence, the number of blocks can be calculated as:

$$blocks = \frac{activethreads}{threadsperblock}$$

4.2.3.B Double buffering schemes

In case double buffering is used, some extra information is also required for proper data processing:

- Number of queries in each iteration;
- Number of queries already processed (so the current threads can save their results in the correct place)

![Figure 4.4](image.png)

Figure 4.4: Example of how using two streams for concurrent memory copy and execution can improve performance by reducing time spent copying data from the host to the device.

By partitioning the total query set in smaller parts, it is possible to copy a small portion to the GPU and begin execution on that subset as soon as that partial copy ends, while simultaneously copying the next query subset (fig. 4.4). In the cases where the Kernel has many memory read/write operations, double buffering does not provide much of a benefit. However, in cases where the Kernel has a large percentage of logical and arithmetic operations, this leaves the
4. Exact Matching

memory controller free to copy the remaining data, thus eliminating much of the overhead for copying data.

An alternative to partitioning the query set is to partition the reference sequence and generate multiple indexing structures, one for each partition. During the matching phase these smaller structures can be asynchronously copied from the host to the device while the Kernel is executing. This will allow the first Kernel to begin execution earlier as the initial setup time (the time to copy the data into the GPU) is smaller. It will also allow the subsequent Kernels to begin executing earlier (or possibly immediately) as part (or the whole) reference structure will already have been copied.

Precautions must be taken to ensure that any match that would begin in one partition and end in the next are found, when partitioning the reference sequence is used. This is achieved by creating an overlapping section, of the same length as the query sequences, between any two consecutive partitions.

4.2.4 Memory de-allocation

Once the results have been copied from the GPU's memory to the host's the GPU's memory should be de-allocated so other applications or later stages of the matching algorithm can still make use of it. It would be possible to simply free all the used memory. However, it might be advantageous to allow some of the already copied data to remain in the GPU if it will be needed for posterior matches, thus increasing the speed and performance of the whole process.

One such example is the reference string, which will most likely need to be present in the GPU's memory if a second stage Kernel is required for more precision matching.

4.2.5 Time measurement and GPU events

Many lengthy GPU operations happen asynchronously or have the possibility of being executed asynchronously. This means that after the invocation of such operations, control immediately returns to the CPU and as such traditional CPU timers are unreliable. Considering, for example the kernel launch instruction (which can be regarded as an asynchronous call) the time interval between the instants before and after the call would be very small, since the kernel launch instruction is non blocking and returns immediately.

To solve this problem, NVidia GPGPUs have a feature called CUDA Events [6]. It is possible to queue an event after an asynchronous operation, so that it is executed immediately after the operation ends and records the time it happened. Hence, CUDA Events allow for precise time measurements on GPU operations.

In the considered system, six events were created to monitor the time spent copying the data from the host to the device, the time spent on the kernel and the time spent copying the results back from the device to the host. Each of these measure requires two events, one to mark the beginning and a second one to mark the end. The time spent is calculated by subtracting the first from the second.
4.3 Exact matching on the GPU

4.3.1 Suffix Tree Kernel

During the matching procedure, each thread of the implemented algorithm (shown in pseudo-code in program 4.2) will try to walk down the suffix tree using the query sequence. For this, it is necessary that the algorithm has a pointer to the current query character, as well as the information regarding the current edge being used: starting index, ending index and the index of its destination node.

Due to the way the query sequences are stored in memory every read of the query sequence yields one to sixteen DNA bases (since a 32-bit integer holds sixteen 2-bit encoded elements but, if the query length is not a multiple of 16 then the last 32-bit integer will have less than 16 DNA bases) which are then stored in a temporary buffer. The algorithm must take into consideration how many DNA bases are in the buffer at all times for proper execution. As a consequence, it is necessary to know the length of the query and how many groups of 16 bases have already been read to calculate if the next group is the last one and, in such case, how many bases it will hold.

After a packed group of bases has been read, using them for suffix tree transversal is simply a matter of picking the first element; once this element has been used it can be discarded and is removed from the buffer. Picking a 2-bit encoded element from a 32-bit integer, considering such an integer to be composed by 16 pairs of bits, equates to picking one of those pairs. There are multiple schemes involving binary masks and shifts that can be used but the simplest and the most efficient one is simply applying a bit-mask to the buffer by performing a binary AND between the buffer and the value 3 ('11' in binary). This has the effect of zeroing all the bits in the buffer with the exception of the two less significant bits, which will represent the next element. After this element is no longer necessary, it is easily removed by simply right shifting all the elements in the buffer by 2-bits. An additional counter is also used to keep track of how many elements have been used from the buffer. This counter is decremented after each shift and, every time it reaches zero, a buffer refill operation is necessary.

The exact matching process starts at the root of the tree. Due to the used scheme to flatten the suffix tree, this node is present at the beginning of the vector and the first 16 elements represent its four possible edges. The initial node is always the same and does not require any calculations. The same, however, is not true for the initial edge.

The initial edge to transverse depends on the first character of the query sequence. Therefore, it is necessary to read this character from the buffer before any edge can be selected. Once it is known the character with which the edge should start, the correct edge is selected from the node by using the character value as offset. Knowing that the ‘A’ edge is located at index ‘0’ and each edge has a length of four integers, edges ‘C’, ‘G’ and ‘T’ will be located at indexes $0 + 4 \times 1$, $0 + 4 \times 2$ and $0 + 4 \times 3$. Hence, the edge selection expression can be generalized to any node in the suffix tree and any character of the query sequence, by just replacing the appropriate values: $nodeindex + 4 \times character$.

After having a query buffer and an active edge to traverse, the next step of the algorithm is to consume characters from the buffer and compare them with the edge’s label. While the next character of the buffer equals the next character of the edge, the comparison continues in a loop.

Whenever the buffer is empty, it must be refilled. When the edge’s label is finished, the active node is updated to be the destination node of the active edge and a new active edge is selected.
4. Exact Matching

by the method above.

**Program 4.2** Kernel for exact string matching using suffix trees.

TreeKernel(activeThreads)
{
    if thread id > activeThreads
        end this thread

    initialize control variables and buffers

    while next character of edge equals next character of query
    {
        if query has ended
            return match found
        fetch next character of query

        if edge has ended
            get new edge based on next query character
        fetch next character of edge
    }

    return match not found
}

There are three possible ways the algorithm exits the comparison loop:

- The algorithm runs out of query characters: meaning that a match has been found;
- The algorithm reaches the end of a leaf: meaning that a complete match could not be found (a partial match can be found in the final characters of the reference sequence);
- The algorithm encounters a mismatch in the comparison cycle.

In either case, the Kernel saves the result into the reserved space for each thread. The saved value is the index of the destination node of the current active edge. This allows a DFS of the suffix tree starting at this node to identify all the leafs, each of which represents a match. In cases where a mismatch is detected, the node identifier that is saved is zero, i.e. the root node.

4.3.1.1 **Results from the suffix tree Kernel**

For each thread the Kernel returns a single integer, representing the unique identifier of a node. To transform the results given by the GPU processing into a location in the reference sequence a final processing stage is required. This processing involves finding all the leafs that are reachable from the node represented at the index returned by the Kernel.

The simplest way is to, before running the Kernel and during the flattening of the suffix tree, create an auxiliary data structure that relates indexes in the vector to nodes in the suffix tree. By using this structure later, finding the desired leafs requires only that a standard DFS algorithm is applied on the resulting node. However, if memory usage must be kept low, the DFS phase can also be applied over the vector representation as long as checks are made to ensure that edges leading back to the root of the tree (leafs and non-existant edges) are not followed.
4.3 Exact matching on the GPU

Due to GPU limitations (recursion is not allowed and kernels cannot allocate memory dynamically), it is impractical to execute a DFS on the GPU. Thus, the DFS must be executed by the CPU after the results have been copied from the device to the host.

4.3.2 Suffix Array Kernel

Exact matching using a suffix array works very differently from using a suffix tree. Since the representation of the inner structure of the reference sequence is not available it is not possible to navigate it. Instead, a sorted list of all possible suffixes is available and the challenge is to find one element in that list.

Finding an element in a sorted list is a very common problem with several solutions already developed and tested. The fastest way is by using a binary search, which compares the element to be found (the query) within the list and recursively reduces the amount of possible matches until only the correct element is left.

Binary search has three main phases: comparison of elements, reducing the test set, picking a new comparison element from the list. In the particular case of DNA exact matching, the last two phases remain unchanged but the way two elements are compared must be adapted to the situation. Program 4.3 shows the main loop of the suffix array matching kernel where the binary search is done.

If the reference sequence and the query sequence were both represented as strings a simple string comparison function would suffice. However, it would be highly inefficient in a GPU, considering that it would have to read each 8-bit character individually and sequentially, and compare them. Instead, in the conceived system these sequences are represented by integers, where each DNA base is encoded with 2-bit, leading to sixteen bases per integer. Not only does this allow for much less memory reads, but it also allows the comparison of all sixteen bases held in an integer to be compared against another sixteen in a single operation, as long as the following rules are respected:

1. An earlier character in the sequence must be represented in more significative bits in the integer than a later character. This allows the comparison to return the correct result when a mismatch is found. Since the portion of the list to be eliminated depends on whether the current item is larger or smaller than the query, it is crucial that the result be accurate.

2. Both integers being compared must hold the exact same number of DNA bases. However, as an element in the suffix array might point to a position that is not a multiple of 16 (i.e. a suffix starting at position 3), that suffix will not be aligned in an integer (the first integer will have DNA bases 0 through 15 and, DNA bases 0, 1 and 2 are not part of the suffix). This fact means that some characters must be discarded from the buffer. However, the comparison result must be accurate and as such some characters might need to be discarded from the query buffer as well to keep the number of DNA bases represented by each integer equal.

The need to discard DNA bases from the reference sequence is illustrated in Figure 4.3.2. As can be seen, the reference is represented by integers holding 16 DNA bases. However, if the wanted suffix begins at an index that is not a multiple of 16 (11 in the figure) the retrieved integer will represent more than what is desired. In such cases the extra bases must be discarded so the comparisons are correctly performed.
4. Exact Matching

An additional consideration is that exact matching might yield, more than one correct result. Therefore, the binary search cannot simply return the first match that is found. In fact, when a match is found there is no way to know if the previous item in the list would not match as well. Hence, since all possible matches are wanted, the algorithm must continue running and shortening the list until only one element, (the smallest) remains in the list. The exception is when the suffix in the array has the exact same size as the query, in which case it is known that the previous item cannot possibly match.

After the smallest element is found, it is known that all the matching elements will be in sequence after it in the list. Hence, retrieving the matching elements simply requires finding out, of the elements in the suffix array, how many begin by the one found; a test that can be simplified by storing this number during the suffix array construction.

The matching process using a suffix array is done within a cycle as in the suffix tree version. However, the algorithm does not exit this cycle as the first mismatch is encountered, as this could simply mean that the algorithm had not yet sufficiently shortened the test set. What this means is that for each query there will be most of the time a fixed number of iterations, the number depending on the size of the suffix array.

Considering that each cycle reduces the test set by an half and the routine ends when the test set is of size one (unless the previous mentioned exception, where the suffix is exactly equal to the query) then the number of cycles is: \( \log_2(\text{reference size}) \)

Program 4.3 Kernel for exact string matching using suffix arrays.

```c
while test set size > 1
{
   while ( query buffer has data
   test buffer has data
   query buffer equals test buffer )
   {
      refill query buffer
      refill test buffer
   }
   if test element > query
      pick first half of test set
   else (if test element <= query)
      pick second half of test set
}
```
4.4 Comparison and discussion of different approaches

4.3.2.A Results from the suffix array Kernel

The suffix array kernel returns, for each thread, one integer representing the element of the suffix array that matched the query. To get, from each entry on the suffix array, the position in the reference sequence it is simply necessary to subtract the length of the element from the total number of bases in the reference sequence.

Like in the suffix tree approach each query might match with more than one region. While in the suffix tree approach, to get all the appropriate locations, it was necessary to perform a DFS on the suffix tree in the suffix array approach all that is needed is to walk the suffix array.

All the elements following the element returned by the kernel that have a prefix equal to the returned element will be a match. To avoid making unnecessary comparisons at this point of the execution it is possible to calculate beforehand how many results should be returned. If suffix array element #0 is a prefix os elements #1 and #2 then, if the suffix kernel returns element #0 as a result, the algorithm only needs to retrieve for elements #0 through #2 the corresponding starting position in the reference sequence.

4.4 Comparison and discussion of different approaches

To evaluate the performance of the developed algorithms [26] a series of sets of query sequences were created. These sets were composed of 1024 to 4194304 DNA sequences of 200 base pairs each. The sequences were extracted from the Homo Sapiens Chromosome 1 (NT_167186.1) and from the Mus Musculus Chromosome 1 (NT_039170.7). Each set had equal numbers of sequences from each organism.

The reference sequence which was used was created from the first $10 \times 10^6$ nucleotides of the Homo Sapiens Chromosome 1 (NT_167186.1)

The tests were conducted on a system composed of an Intel Core i7 950 quad-core processor, running at 3GHz, with 6GB of RAM. This platform also includes a NVIDIA GeForce GTX 580 GPU, with 512 processing cores running at 1.54GHz and 1.5GB of RAM.

The first test conducted compared the performance of implementations of sequence matching algorithms based on either suffix trees or suffix arrays and the popular tools Bowtie and SSEARCH35. The algorithms were compiled to make use of the POSIX threads API to support parallel execution of multiple concurrent threads. It is important to note that the results obtained when using 8 threads were achieved by making use of the Hyper-Threading technology.

From the obtained results (see Fig. 4.6) it can be observed that as expected suffix arrays have a lower performance than suffix trees. This is due to binary search having a greater asymptotic complexity than that of the suffix trees. In practice the performance of both implementations is quite similar. This result was already observed in [18] [27], and is mainly due to a more efficient usage of the cache memory by the suffix array, which is achieved due to its smaller and more regular structure.

Furthermore, by comparing the execution time results with the Bowtie and SSEARCH35 programs, it is possible to observe that the implemented suffix tree and suffix array algorithms are significantly faster, justifying their adoption if high performance DNA alignment is required.

The performance of the developed algorithms was also assessed in a GPU platform, namely the NVIDIA GeForce GTX 580. The algorithms were also compared with MUMmerGPU MUMmerGPU [15], another tool for sequence matching using suffix trees. The obtained results (Fig. 4.7
4. Exact Matching

Figure 4.6: Performance evaluation of suffix tree and suffix array index based search algorithms in multi-core CPUs.

Figure 4.7: Performance evaluation of the considered index based search algorithms in the GPU.

correspond to the total execution time of the algorithms, while searching for the corresponding number of query sequences in the reference sequence. The total execution time considers all
4.4 Comparison and discussion of different approaches

Figure 4.8: Communication and kernel execution times for the implementations in the GeForce GTX 580 GPU.

the required data transfers (host to GPU and GPU to host), as well as the kernel execution time. From Fig. 4.8 it is possible to see that the data input time is very significant especially when the number of query sequences to be searched is very small. This data input time is the main responsible for the lower performance values provided by the GPU implementations, when compared to the corresponding CPU implementations. However, as more query sequences are considered by the algorithm and more of the processing cores in the GPU are used, the GPU implementations offer a significantly better performance, with speedup values as high as 85 for the suffix array implementation and 25 for the suffix tree implementation.

These observations reveal that contrary to what was obtained in the CPU, the GPU implementation clearly favors the suffix array index structure. This is in part due to the execution flow of the algorithm being more regular and it makes better use of the cache memory available and, in part, because the space occupied by the suffix array index is much smaller than that of the suffix tree index. Since the suffix array is much smaller than the suffix tree the suffix array implementation always presents a much lower transfer time from the host to the GPU device.

The results obtained with MUMmerGPU were consistently higher than the implemented suffix tree and suffix array runtimes. The better performance obtained by the developed suffix tree implementation is explained by simplifications and optimizations introduced into the algorithm which are not present in MUMmerGPU.

Comparison with CUDASW++ software was not possible since it has a limitation on the maximum reference size of about $64 \times 10^3$ base pairs. Overall, the proposed suffix array implementation achieves the best results when there is no overlap of computation and communication.
4. Exact Matching

4.4.1 Overlap of communication and computation

Taking into consideration the overhead imposed by the memory transfer operations during the setup stage and its impact on the overall performance of the algorithm, a double buffering scheme was implemented and evaluated. As previously described, in such scheme the data to process is divided into several parts. During the processing phase of the algorithm the data for the next processing phase is concurrently copied to the while the data for the current processing phase is being used.

4.4.1.A Partition over reference sequence

One of the methods to implement double buffering on the exact matching algorithm is to partition the reference sequence into multiple smaller sequences. Afterwards, for each of these smaller sequences the respective index is constructed.

This method leads to smaller suffix trees or suffix arrays to index the smaller sequence, which reduces the time necessary to copy this information from the host memory into the device memory. However, since there will be multiple reference sequences it is necessary to store a result for each of the sequences in the query set for each of the partitions of the original reference sequence. This extra results, although simple to implement, require additional information to be copied from the device to the host decreasing the overall performance of the algorithm.

Using the double buffering method together with the suffix arrays it can be seen (Fig. 4.9) that there is no performance improvement. This is due to the fact that suffix arrays are already very space efficient and reducing them in size does not outweigh the extra time required to copy the results from the device to the host. Additionally, since the index for the reference sequence is very small, the concurrent copy takes only a small fraction of the time the kernel takes to execute.

Using the same method but indexing the reference using a suffix tree the performance of the algorithm improves (see Fig. 4.9). As in the previous approach, since the reference is partitioned into two subsequences, the time to copy the first index is halved. However, since suffix trees are typically much larger than their suffix array counterparts this leads to a greater gain in performance when the double-buffering method is used.

Furthermore, the smaller suffix trees allow for a faster matching of the query set, due to the specific algorithm used to match a sequence. This happens since a smaller reference sequence generates a smaller tree, with less branches, leading to less memory accesses. Another factor is that a smaller reference sequence has a larger chance of not matching a query sequence. Such larger chance means that the matching algorithm, which traverses the tree, is likely to terminate sooner (when a mismatch is found). This is contrary to what happens with suffix arrays in which the number of steps is constant (using binary search).

These two factors combined compensate for the extra time required to copy the results back to the host from the device and provide a performance improvement when the number of query sequences is low. As the number of query sequences to match increases the performance begins to decrease, as more time is spent copying the query sequences into the device and the results to the host.
4.4 Comparison and discussion of different approaches

Figure 4.9: Performance evaluation of suffix tree and suffix array index based search algorithms in GPUs using overlap of communication and computation with a partitioned reference sequence.

Figure 4.10: Performance evaluation of suffix tree and suffix array index based search algorithms in GPUs using overlap of communication and computation with a partitioned query set.
4. Exact Matching

4.4.1.B Partition over query set

An alternative to partitioning the reference sequence is to partition the query sequence sets. This method has the benefit of not needing the additional memory to store multiple sets of results. However, in cases where the reference sequence is large and the query set is small it may not provide as much of an improvement as the previous method.

Partitioning the query sequences also has the benefit of not requiring any overlap of the references, unlike the case with partitioning the reference sequence. This means the total amount of data to copy does not change with the number of created partitions. Since the index of the reference sequence only needs to be copied once, at the start of the algorithm, its size (and consequently the time it takes to copy the index to the device) becomes less relevant as the number of queries to match increases.

Using the query partition approach combined with suffix arrays provides large performance improvements by reducing both the time required to start the execution of the first kernel and the time required for the kernel itself to run as there are less sequences to test. Since suffix arrays are already very space efficient, most of the time is spent copying the query sequences into the device, even when the number of sequences to match is small. As can be seen in the results (Fig. 4.10) the performance gradually increases as more query sequences are analyzed.

The same trend that is observed when using suffix arrays can also be observed, from the results, when using suffix trees (Fig. 4.10). However, as suffix trees are larger than suffix arrays for the same sequence, the impact of partitioning the query sequences is smaller when the number of query sequences is reduced. As the number of query sequences to match increases, the overall performance gains of the algorithm also increase, as the time required to copy the representation of the reference sequence becomes a smaller fraction of the time used to copy the data into the device.
5. Approximate Matching

As it was shown in the previous chapter, finding whether two sequences are exactly equal can be easily achieved through suffix trees and suffix arrays. However, real life biological sequences seldom are completely identical. In most cases, there are some small differences between sequences in the form of insertions, deletions or mutations, which make pure exact matching unsuitable for the purpose of DNA alignment.

To get information with biological meaning, by comparing two similar but not equal sequences through string matching, traditional exact matching algorithms cannot be relied upon. By taking into consideration that two sequences that have some biological connection are expected to be largely similar, it is possible to reduce the problem of approximate string matching to finding the sections in which the two sequences are equal (i.e. through the previously studied exact matching mechanisms) and afterwards bridging the gaps between these sections using a more exhaustive search algorithm (i.e. using the Smith-Waterman algorithm).

![DNA Alignment showing portions between query and reference that match.](image)

Figure 5.1: DNA Alignment showing portions between query and reference that match.

Figure 5.1 shows how two different sequences with high similarity are composed by sub-sequences that match exactly, separated by some mismatching characters. Although a complete exact match between them is impossible, it is possible to find two partial matches. Then, an attempt might be made to join them into a single alignment, provided that the distance between them is not too great and would not lower the alignment score under acceptable values.

5.1 Possible approaches for approximate matching

When adapting the algorithms from exact to approximate matching, the issue becomes how to implement this change so that the algorithms identify multiple portions of the sequences that are equal rather than simply testing two whole sequences for complete equality. In order to achieve this, some requisites become apparent:

- Whenever a mismatch is found the algorithm cannot simply return, as it did before;
- For each mismatch, a new matching attempt must be started;
- Each matching attempt must yield a result.

Comparing exact matching to approximate matching, the underlying alignment algorithm should remain the same. The only change should be observed when a mismatch is encountered. Finding a difference in the sequences pair marks the end of a matching phase and its results can then be saved. Beginning the next phase requires that the next starting position for the matching cycle to be changed, implying that the algorithm should discard some beginning portion of the query sequence.

Hence, by moving the starting position of the algorithm one or several positions forward, a prefix of the query sequence is being discarded, allowing the matching algorithm to follow a completely different path and possibly find a better local exact match.
5.1 Possible approaches for approximate matching

Figure 5.2 exemplifies how removing the prefix of the query sequence can lead to better matches. The first row in the figure represents the reference sequence being considered and the following rows represent various suffixes of the query sequence 'MISSIPPI'. The first column indicates the length of the prefix that is being discarded. If the original query is considered, a maximum matching sequence of 5 bases is found. However, if the first character is discarded, two possible matching sequences of 4 bases each become apparent. Similarly, if a prefix of length 4 is removed the largest possible sequence for this case is identified as 'ISSIPPI'.

The metric by which these small sequences are evaluated may vary since size alone does not determine whether that subsequence belongs to the final alignment or not: in fact several small sequences in close proximity might yield a better alignment score than a single long sequence.

5.1.1 Suffix arrays

When attempting to match two sequences using suffix arrays, a search for a specific element is done on the whole array. The binary search method previously used allows the search to be done in $O(n \times \log(n))$ time, by repeatedly dividing the array in half. This approach is fast and when combined with the less time needed to copy the data from the host to the GPU device makes it faster than the other studied alternatives.

Adapting the binary search approach on a suffix array for approximate matching is simply a matter of enveloping the algorithm in an extra loop that performs the binary search and removes the initial character of the query once a mismatch is determined. However, this approach is not very efficient, since the algorithm cannot make a decision about whether the sequence matches or not until all search options have been exhausted, reducing the array to a single element, the best match. In an array with n elements this means $\log_2(n)$ sequence comparisons need to be performed to find the best match.

Program 5.1 Pseudo-code for approximate matching done using regular suffix arrays.

```
ArrayKernel(activeThreads)
{
    while (query length - query start index > minimum limit)
    {
        do regular binary search { ... }
        if perfect match
            end
        else
            remove the prefix from the query
    }
}
```
5. Approximate Matching

Program 5.1 illustrates how a modified suffix array kernel would attempt to do approximate matching by removing an increasingly larger prefix from the query. The main loop performs the matching through binary search. Every time the binary search loop ends the results are evaluated. If the binary search did not find a perfect match for the query, the first character of the sequence is removed and the whole process starts again.

The inability to detect a mismatch without testing all $\log_2(n)$ entries stems from the root of the algorithm that recursively attempts to find a better match between the reference and the query sequences. The best possible match (if it exists) is not known until the very last step of the algorithm, except in rare cases where there is an exact match.

Hence, using simple suffix arrays to perform approximate matching between two sequences means restarting the whole binary search algorithm multiple times for a single query sequence, once for each character discarded from the prefix.

When combined with the fact that each matching cycle requires several individual comparisons, restarting the matching algorithm each time a mismatch is detected will lead to a very large number of comparisons per query sequence. Additionally, during the binary search when a new suffix array entry is picked for analysis any previous progress is discarded even if the current entry is very similar to the last one forcing some unnecessary character comparisons to be repeated.

It should be mentioned that, like in exact matching, the access pattern in the suffix array is unpredictable. Due to this, the best option to optimize memory accesses to the suffix array is to use textures, or rely in the global memory cache if available.

### 5.1.2 Enhanced suffix array

In an attempt to keep the small memory usage of suffix arrays and, at the same time, to reduce the number of comparisons required per query sequence additional information may be added to the suffix array about the considered reference sequence internal structure. When this kind of change is introduced to a suffix array, it is common to rename it to enhanced suffix array.

There are many different fields of information that can be added to the suffix array. The most common being the 'longest common prefix' or LCP for short. This allows the suffixes to be grouped, based on their prefixes, and effectively creates groups or intervals of entries in the array.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Index</th>
<th>LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>ippi</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>issippi</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>ississippi</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>mississippi</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pi</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>ppi</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>sippi</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>sissippi</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ssi</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>ssissippi</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 5.3: Enhanced suffix array using an extra column to hold the longest common prefix values.

Figure 5.3 illustrates an enhanced suffix array with the longest common prefix. In this case the
LCP was calculated by comparing one element with the previous. It becomes clear that the first entry in the array cannot have any prefix in common since there are no previous entries and thus its LCP is zero. The second entry has only the first character in common with the previous entry ("ippi" has 'i' in common with 'i') thus its LCP is 1. The fourth entry has 'issi' in common with the previous element which is indicated by its LCP of 4. These first 4 entries can be considered to be the 'i' interval.

By using the LCP and the resulting suffix intervals, the binary search function can be changed so that it no longer reduces the test set by half in each step. Instead, it may change the left and right borders of the interval to match that of the current pivot's interval.

While this scheme to reduce the test set in order to include only elements that are known to share the same prefix might seem beneficial, in reality it might not generate any improvement at all and may even lead to a performance loss. The justification for this relies on the fact that it requires many memory accesses to find the boundaries of the current element's interval and even after finding the new boundaries for the test set the number of entries removed in each step might be less than the number that would be removed by using the traditional rule.

**Program 5.2** Calculating new boundaries for binary search after a is mismatch detected.

```plaintext
if current element is larger than query
    walking direction is up
else
    walking direction is down

walk the array reading the LCP on each entry
{
    stop when LCP < number of matching characters
}

set second boundary to the last element read
```

Program 5.2 exemplifies the problem, by using the LCP to calculate the boundaries for the next step of the binary search. Depending on whether the current element has a larger or smaller value than the query, the search must go in different directions in the array. Regardless the direction taken, the array must be walked, entry by entry, checking the LCP. Any entry with a LCP larger than the number of matched characters will also hold a mismatching character. Ideally, the end result would be a set of entries which have an LCP exactly equal to the number of matched characters. In reality, the LCP may vary by large amounts and there might not be a LCP with that exact value.

There are a couple of factors that make this approach undesirable:

- The need to store and copy the extra information;
- The need to access extra memory positions each time the LCP needs to be read;
- The need to introduce control logic to decide whether to use the standard binary search reduction or to use the LCP;
- Even with this change every new binary search cycle will have to start from the beginning regardless of how long the matched section was.
5. Approximate Matching

Program 5.3 Kernel for approximate DNA matching using enhanced suffix arrays.

\[
\text{ESA-ArrayKernel}(\text{activeThreads})
\]
\[
\{
\quad \text{while (query length - query start index > minimum limit)}
\]
\[
\quad \{
\quad \quad \text{while test set > 1}
\]
\[
\quad \quad \{
\quad \quad \quad \text{compare query against suffix until a mismatch is found}
\]
\[
\quad \quad \quad \text{if test element > query}
\]
\[
\quad \quad \quad \quad \text{while current element lcp > number of matched characters}
\]
\[
\quad \quad \quad \quad \quad \text{current element = previous element}
\]
\[
\quad \quad \quad \quad \quad \text{right pointer = current element}
\]
\[
\quad \quad \quad \quad \quad \text{else (if test element <= query)}
\]
\[
\quad \quad \quad \quad \quad \text{while next element lcp > number of matched characters}
\]
\[
\quad \quad \quad \quad \quad \quad \text{current element = next element}
\]
\[
\quad \quad \quad \quad \quad \quad \text{left pointer = current element}
\]
\[
\quad \quad \}
\]
\[
\quad \}
\]
\[
\text{if perfect match}
\]
\[
\quad \text{end}
\]
\[
\text{else}
\]
\[
\quad \text{query start index = query start index + 1}
\]
\[
\}
\]

As it can be seen from program 5.3, the whole comparison routine must be adapted so that it not only returns which of the elements is greater and which is smaller, but it must also keep track of (and return) how many equal characters existed in the two sequences. Considering that the previously implemented method works by comparing integers holding from one to sixteen DNA bases, a whole new method that compares characters one by one must be implemented. This method will inevitably be slower, as it requires more comparisons to be done than before.

An alternative to the previous method to speed up the matching procedure would be to find a way to avoid having to restart the whole process from the beginning after every step, when the query starting index is changed. One way to achieve this would be to add to each array entry the location in the array of the next suffix. By adding to each suffix array entry the index of the string corresponding to the removal of the first character of that entry it is possible to immediately jump, in the next iteration of the cycle, to a similar suffix, therefore eliminating several unnecessary comparisons.

By using the mentioned method to jump to a similar suffix at the beginning of a new matching phase it is possible to ignore the first comparison, since as the suffixes will be equal in everything except the first character. If a mismatch was previously found at the \(i^{th}\) character, then it is certain that there will be a mismatch at the \((i - 1)^{th}\) character of the new suffix and the current element can be set as one of the boundaries of the test set.

Figure 5.4 shows the same sequence, this time represented through an enhanced suffix array with both LCP and suffix column. The suffix column holds the index of the array entry that corresponds to the sequence generated by discarding the first character from the current entry. If the
5.1 Possible approaches for approximate matching

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Index</th>
<th>LCP</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ippi</td>
<td>7</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>issippi</td>
<td>4</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>ississippi</td>
<td>1</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>mississippi</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>pi</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ppi</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>sippi</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>sissippi</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ssippi</td>
<td>5</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>ssissippi</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 5.4: Enhanced suffix array using an extra column to hold the longest common prefix values and another column for the index of the next suffix.

The first character ‘i’ is removed from sequence ‘ississippi’, stored in the array element number 3 (zero indexed), it generates the sequence ‘ssissippi’ which is stored at the array element 10. Hence, in row 3 the suffix column holds the value 10. This rule holds true for every entry. The first element is an exception since it only has one character. Removing it would yield the empty sequence which is not represented in the array. In this case, its suffix index is itself, although any algorithm that makes use of the suffix column should check for the empty sequence.

The binary search algorithm requires two boundaries for the test set so it can calculate the next element to compare. Assuming the current element to be one of these margins the algorithm lacks the second boundary. The simplest way would be to set the second boundary of the interval to either zero or to the length of the reference sequence, depending on which half of the array it is needed to search next. However with the longest common prefix information it is possible to set the test interval to include only the entries that have a certain amount of characters in common with the current element.

Again, the issue of walking through the array to read the LCP for each entry, while trying to set the boundaries for the next iteration comes up. By carefully studying the suffix array, it is noticeable that since there are only a small number of DNA bases, the possible combinations are also small. At LCP of length 1 there are four possible intervals, one for each DNA base. At LCP of length 2, each of the previous 4 intervals subdivides into 4, yielding 16 total intervals and every subsequent LCP level will yield 4 times as many suffix intervals as the previous level.

Considering that the LCP level is the indexing value and does not need to be explicitly stored and that each interval is defined by its starting element and length, it is possible to take this information and construct a structure that holds all the LCP intervals for any required level. This would allow the binary search algorithm to immediately jump from one element to its first suffix, while also setting the algorithm interval’s left and right border to the most restrict interval possible, based on the number of matched characters in the last comparison.

Figure 5.5 shows a possible implementation of an enhanced suffix array using a column for the next suffix index. Instead of the single LCP column of the previous examples, this suffix array has multiple LCP columns each for a different LCP length. The LCP(1) column represents the intervals of entries which share a prefix of length 1 by their first and last element. All entries that begin with the same character will belong in the same LCP(1) interval as can be seen by elements 0 through 3 where all begin with ‘i’ and have 0:3 assigned in the respective column. By following
this rule, it becomes clear why entries 9 and 10, which share the prefix 'ssi', have the common interval 9;10 for LCP(3).

By using multiple LCP intervals and suffix index columns the algorithm can easily jump from one entry to the next and, according to the number of matching bases, set the testing interval boundaries. For example, if the algorithm matches the sequence ‘ip’ and then encounters a mismatch it can remove the ‘i’ from the sequence, jump to its suffix (which would be ‘ppi’) and set the testing interval to 5;6 (LCP(1) is used since two characters were matched but one was discarded when the suffix index was followed).

Although very useful, the LCP intervals approach is impractical if very long matching subsequences are expected, since it requires storing many different LCP levels. Since each interval requires $2 \times 4$ bytes of storage and each entry has its own intervals, adding many levels substantially increases the array size. The first interval doubles the size of the array, while adding 2 levels of intervals triples the size of the array from its original size. However, considering that the matches being looked for are usually relatively small common subsequences, the depth of the intervals can be limited to the maximum size of the subsequence that is expected to be found, in order to save memory.

It is also possible to add a level of indirection between the suffix array and the LCP tree, to avoid storing two integers per entry in the suffix array. If each entry in the suffix array indexes one entry in a separate structure, which then holds the indexes for the first and last elements of that family, it is possible to save some space. In large intervals, where all the elements index the same interval, having one degree of indirection would save 4 bytes per interval member after the first. However, for single element intervals this would actually use 4 more bytes than the original method. Furthermore, the indirection would require one additional memory read, which can be very costly in GPUs.

The cornerstone to efficiently solve the problem of using suffix arrays for approximate string matching is the fact that in most cases the algorithm only needs to find the exact match of a small subsequence of the query: the prefix. What this means is that there is never any need to compare more than those initial characters and neither are LCP-intervals of length larger than the prefix necessary.

To solve this problem, a structure was created that indexes the entries in the suffix array based on their prefix. To create such a structure, it must be understood that every time the binary search
5.1 Possible approaches for approximate matching

reinstates the algorithm, it is faced with a new query sequence. Previously, it was tried to use the
past work to accelerate the new phase, by skipping a few matching cycles. The new approach
discards the past work and, instead of jumping to an element in the suffix array related to the last
query sequence, it picks a LCP interval of fixed length and immediately begins the search in that
interval, without complex calculations.

This new structure is, in fact, a hash table holding the indexes of the suffix array elements
and the hash key to these entries is the element itself or at least its prefix. Since as the
query sequence is already represented by integers and the DNA bases are encoded using 2-bits,
retrieving a suffix array index from the hash table is achieved by reading the value stored in the
hash[query] position.

<table>
<thead>
<tr>
<th>Hash Key</th>
<th>Hash Value</th>
<th>Array Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>im</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ip</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>is</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>mi</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>mm</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>mp</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>ms</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>pi</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>pm</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>pp</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>ps</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>si</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>sm</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>sp</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>ss</td>
<td>15</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 5.6: Example of a suffix array indexed through an hash table with a key length of 2.

The hash table in figure 5.6 was created by assigning to each entry the suffix array index of
the first entry that begins with the hash key. The characters were assigned to values such that
when the key is sorted by its numerical value, the key is also lexicographically sorted. When the
hash key is 'mi' the first suffix entry that begins with this sequence is 'mississippi', in index 4, thus
in the hash table 'mi' is the key for the value '4'. If one key does not have a matching entry in the
suffix array it is given the previous value. This ensures that the left border of the test interval will
be correctly set. The right border of the test interval is set to the value of the next key in the hash
table. This would set an interval of [4;4] for the sequence 'mi', since its value and the next are
both 4. In this particular case, there is only one entry beginning with 'mi' and this approach allows
us to find the match almost instantaneously.

With its simplicity, this approach offers four important benefits:

- It requires very little computational effort to calculate the hash key from the query sequence;
- Reading the left and right boundaries for the binary search algorithm is as simple as reading
  hash[query] and hash[query + 1], which are very likely to be cached for small hash tables;
- Eliminating unnecessary depth of the LCP-tree leads to memory usage improvements;
- The size of the hash key can be fine tuned to any size, as long as it is a multiple of two.
5. Approximate Matching

As previously mentioned, the binary search method reduces the test set in half for each comparison it does. Considering a reference sequence of size $n$, which also generates a suffix array of size $n$, it takes $\log_2(n)$ to reduce the test set to a single element (the requirement to decide between match and mismatch).

Assuming the reference sequence as a somewhat balanced distribution of DNA bases, selecting just the suffix array elements that begin with a certain DNA base eliminates $3/4$ of the array, the equivalent to two binary search steps. Since each base is represented with 2-bits, it can be said that for each bit used in the hash key one step of the binary search is removed.

Hence, for a reference sequence of $n$ characters it takes $\log_2(n)$ steps to reach the final state using a regular binary search. If a hash table is implemented with a key length of $\log_2(n)$ bits (the key represents $\log_2(n)$ DNA bases from the sequence) it holds exactly $n$ elements (one element per suffix in the reference). Using such a hash table, there might be a direct and immediate translation from the query to its position in the suffix array, as long as every entry in the suffix tree has a different hash key value. If there are suffixes present in the array that have a LCP of length $\frac{\log_2(n)}{2}$ or larger, then all of them would address the same value in the hash table. In such cases, there is no collision and only the first element in the suffix array is stored.

**Program 5.4** Approximate DNA matching using suffix arrays indexed by hash tables.

```plaintext
Hash-ArrayKernel(activeThreads)
{
    if thread id > activeThreads
        end this thread

    while query length > minimum length
    {
        calculate hash-key with initial characters of the query
        left = hash[key]

        if key points to last element of hash table
            right = last element of suffix array
        else
            right = hash[key+1]

        Binary search for query between left and right

        if perfect match
            end
        else
            remove prefix from query
    }
}
```
5.1 Possible approaches for approximate matching

Program 5.4 outlines the steps taken for approximate DNA matching using the mentioned suffix array and hash table approach. By using the first characters of the query to calculate the hash key it is possible to retrieve from the hash table the first occurrence in the suffix array which matches the query and set the left border for the test interval. The second border is set by reading the next element of the hash table. A simple precaution must be taken in the cases where the hash key indexes the last element of the hash table, to avoid reading out of bounds. In such cases, the second border is set to the last element of the suffix array.

If it is considered that, in a DNA alignment, the individual portions that match exactly between the reference sequence and the query sequence have a short length, then there is no need to continue matching a specific suffix of the query sequence past the maximum considered length for the exact matches. Even in cases where the DNA alignment is composed of long exact matching subsequences, these subsequences can be considered as the result of merging several small sequences. Taking this into consideration, it is possible that a variant of the previous approach, using suffix arrays and hash tables, performs well if only the first few characters of each suffix of the query sequence are matched.

Regardless of which method is used, as in regular suffix arrays, in enhanced suffix arrays the access pattern is also unpredictable. Due to this, the best method to improve performance in memory accesses is to make use of the caching mechanisms (texture or global memory caching when available). The hash table, if it is small, might be able to fit in the GPU's shared memory. However, since shared memory relies on different threads accessing addresses on different memory blocks and the access pattern to the hash table is unpredictable, there might not be a performance improvement.

5.1.3 Suffix trees

Given the issues surrounding the suffix arrays and the arisen difficulties, an alternate solution based on suffix trees was considered, since they already hold much of the structural information in the reference sequence and might therefore be a better candidate for approximate matching.

The followed approach for approximate matching is to repeatedly remove the first character of the query sequence and attempt a better match against the reference sequence. One particular trait of suffix trees becomes extremely important in this task: the suffix link.

In a suffix tree, such as the one represented in figure 5.7, every internal node represents a prefix of a suffix of the reference sequence. This sequence is held in the edges leading from the root of the tree to the specified node. Each node also has a suffix link, defined as a special edge that leads from one internal node directly to another node representing the same prefix with the first character removed. The first node is going to be referred to as the source node and the second the destination node.

Every node has a suffix link. In figure 5.7 these are represented by dashed lines connecting the nodes 5 to 4, 4 to 3 and 3 to 1. Nodes 1, 2 and 6 also have suffix links but since they lead back to the root of the tree they were omitted for simplicity.

MUMmerGPU [15] is one application that makes use of the suffix link to match the various suffixes held in a query sequence against the reference sequence. This is done is by looking for the first match of the complete query sequence, as usual. However, once a mismatch is found, instead of starting a new matching cycle from the root of the tree the algorithm follows the suffix link from the current node. By following the suffix link the algorithm is lead to an internal node with
the same prefix but without the first character. Hence, since only the first character was removed the match is still in a valid state for the new query sequence. However, some precautions must be taken. In fact, since there is no guarantee that the mismatch was found at the beginning of an edge, when the algorithm jumps to the destination node it is necessary to find the correct edge to follow. Despite, the query sequence being temporarily held in a buffer, it is discarded once characters are matched. In this undesired scenario, it is certain that the character that lead to the selection of the edge to walk through next will already have been discarded.

The simplest solution to the previous problem would be to keep track of the number of characters matched, excluding any characters matched on the edge where the mismatch was found. After following the suffix link, the algorithm should continue the match ignoring the characters previously mentioned. However, this method would need to discard some of the work already done (the characters that were excluded from the count) as well as would need to reload information into the buffer, with the performance penalty it involves (the need to calculate which element to read, how many bases of that element to discard and discarding those same bases).

One fact to consider is that, since a second stage of the matching algorithm is essentially matching the same sequence, any previous progress should remain completely valid. If the algorithm found a mismatch in characters into an edge and then followed a suffix link, the next \( n \) characters are guaranteed to match. The algorithm does not need to match these characters again.

The naive way of solving the problem of the selection of the edge that should be considered after following a suffix link is to calculate which query character would be compared next, reload the buffer for that specific query section, remove extra leading characters and then use the character to perform the selection process as usual. This method is time consuming and requires read memory operations.

A more optimized and very simple approach is to store the last character used to make an edge selection every time the algorithm enters a new node. This character is called the backtrack character and can be stored in a register of the stream processor so a memory read is not required to access it.

Due to the way the suffix tree is flattened into a vector, the suffix links are not attached to a node, since there is not an explicit representation of nodes. Instead, the suffix links are attached to each individual edge using the \( 4^{th} \) integer that was previously used for padding to store the destination of the suffix link, as can be seen in Figure 5.8. In this example, the first node has a
5.1 Possible approaches for approximate matching

suffix link to the node at index position 32 (the third node since each node takes 16 positions and
the first node starts at index zero).

<table>
<thead>
<tr>
<th>Index</th>
<th>Start</th>
<th>End</th>
<th>Destination</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>80</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>128</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>13</td>
<td>14</td>
<td>48</td>
<td>128</td>
</tr>
<tr>
<td>99</td>
<td>99</td>
<td>0</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>99</td>
<td>0</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>288</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.8: Suffix tree with one field holding the suffix link.

Typically, the suffix link destination would be a node that, in the vector representation of the
suffix tree, would correspond to the index of the first edge of that node. However, since every
time a suffix link is taken an edge from the new node will have to be selected, the value stored
as the suffix link should not be the addressed node but the address of the edge which would be
selected as it is shown in figure 5.9. What this means is that if the algorithm’s active edge is the
$i^{th}$ edge of a node when a mismatch is found then, after following the suffix link, it must be in the
$i^{th}$ edge of the new node. By pre-calculating the edge in this way the need to keep the backtrack
character in a register is eliminated and the edge selection process following a suffix link jump
can be completely removed.

<table>
<thead>
<tr>
<th>Index</th>
<th>Start</th>
<th>End</th>
<th>Destination</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>32</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>80</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>128</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>13</td>
<td>14</td>
<td>48</td>
<td>128</td>
</tr>
<tr>
<td>99</td>
<td>99</td>
<td>0</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>99</td>
<td>0</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>288</td>
<td>131</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.9: Suffix tree with one field holding the suffix link and each edge leading to a different
eedge.

Even knowing beforehand which edge to choose from the destination node, one concern that
still remains is observed when the mismatching character is found in the middle of an edge. After
following the suffix link, that edge may have been split and new nodes introduced. If this happens,
then either a complex edge skipping routine has to be implemented or some, possibly a large
amount, of the progress will have to be discarded.

An example of the previous mentioned situation can happen if a query sequence \texttt{ISSP} is
matched against the suffix tree represented in fig. 5.7. The algorithm will be half-way through the
dge starting by ‘S’ and leading out of node \#1 when a mismatch is found. The suffix link leads
back to the root node, however, the ‘SS’ sequence which was previously matched in a single edge
is now split in two edges: ‘S’ and ‘SI’.

The edge skipping routine will begin by assuming that all the progress so far continues valid,
5. Approximate Matching

so no comparisons need to be made along the edges. Every time an edge is selected a number of characters equal to the pending progress are skipped along it. If there are not enough characters in the edge, then the algorithm must jump directly to the destination node of that edge and pick the appropriate edge. Again, this raises the issue of reading the query data into a buffer and discarding unnecessary characters. However this would have to happen, regardless of the adopted approach, with the benefit that by skipping the edges a very smaller number of comparisons is needed.

This approach of using suffix trees and suffix links has the important benefit of comparing only as many characters as the length of the query, although sometimes it is necessary to re-read portions of the query, which is a big advantage over the suffix array methods. Regardless of how many times the process is restarted, by following a suffix link the previous progress is always kept, meaning that the comparisons can continue where the last stage left off.

**Program 5.5** Kernel for approximate DNA matching using enhanced suffix trees with suffix links.

```c
TreeKernel(activeThreads)
{
    while query length > limit
    {
        while pending comparisons >= edge length
            skip this edge

        while next character of edge equals next character of query
            { 
                if query has ended
                    return match found

                if edge has ended
                    get new edge based on next query character
                    reset pending comparisons

                    fetch next characters to compare
                    increment pending comparisons
            }

        follow the suffix link
    }
}
```

As in the suffix array approximate matching kernels, the suffix array kernel for approximate matching in program 5.5 also relies on a loop to check if the current query sequence is large enough for the matching to proceed. Inside the main loop, the first step is to skip any edges if there are matched characters pending from the last phase. During the tree walking algorithm that does the actual matching, any matched characters are considered as pending. When the destination node from an edge is reached, the pending characters counter is reset to zero. Should a mismatch be found, the algorithm simply moves from the current node to the suffix node and jumps to the beginning of the main loop.

Again, the only optimizations and GPU specific adaptations that can be applied are regarding
the matrix of query sequences. As in the suffix array and enhanced suffix array approaches, the access pattern to the reference sequence is too unpredictable and prevents any optimization of the data structures that hold it. Access to the suffix tree can only be improved by way of the caching mechanisms offered by the GPU.

5.2 Discussion and Comparison of the considered approaches

The adaptation of the index based search algorithms used for exact string matching to a version suitable for approximate matching was evaluated by using the same sequence data as in the previous chapter.

The algorithms were also evaluated in the same computational system that was used to evaluate the algorithms in the previous chapter, which includes a NVIDIA GeForce GTX 580 GPU, with 512 processing cores running at 1.54GHz and 1.5GB of RAM.

In the tests involving suffix arrays, to accurately simulate the scenario of removing prefixes from the query sequences in order to perform the approximate matching, the algorithms were set to finish after the first 100 base pairs had been removed from the query sequence.

In a preliminary evaluation it was studied the impact of the hash key length on the performance, by running the implementation of the enhanced suffix array algorithm with a hash table index of various hash key lengths. It is important to note that larger hash key lengths will lead to larger hash tables, which will take longer to transfer from the host memory to the device memory. Simultaneously, large hash key lengths index the suffix array with a finer precision, which leads to faster binary search results. This procedure allowed to compare the performance of the algorithm with various hash key lengths and find which would lead to the best compromise to compare against other implementations.

![Performance evaluation of suffix arrays based algorithms using a hash table as index on the GPU.](image)

Figure 5.10: Performance evaluation of suffix arrays based algorithms using a hash table as index on the GPU.

The obtained results (see Fig. 5.10) indicate that larger hash keys provide better performance by finishing the problem faster however they have a large overhead cost for transferring the hash table. The cost for transferring the hash table is offset by faster running binary searches. At the maximum limit for the hash key length (14 base pairs) the algorithm consumes a large portion of the time transferring data from memory however with the hash table created the interval used by
5. Approximate Matching

the binary search part of the algorithm is very small leading to run times competitive with the other hash key options.

The following step was to compare the other alternatives for approximate DNA matching against the enhanced suffix array based algorithm, making use of the best obtained hash key length. A naive suffix array implementation, a suffix tree implementation, a hash table indexed enhanced suffix array and a variant of the former algorithm that compares only the first 5 base pairs of the query sequence were tested. The results are presented in Fig. 5.11.

![Figure 5.11: Performance evaluation of suffix trees and suffix arrays index based search algorithms on the GPU.](image)

The first conclusion that can be drawn is that for very small test sets all algorithms have a very similar performance. Only when the test sets grow in size do significant differences appear. Most notable is the exponential growth of the naive suffix array implementation. Since it does not have any optimizations, it quickly becomes too slow to be considered competitive.

With larger test sets, the algorithms based on indexed suffix arrays index start to differ as well. The variant which only compares small portions of the query sequence showed slower results, as the algorithm must now keep track of the number of matched base pairs as well and the search algorithm begins to encounter memory access problems, which cannot be solved by the warp scheduler as threads are requesting memory accesses very often.

Lastly, the suffix tree approach presented the best results, mainly because it does not need to rely on large auxiliary indexing structures and does not restart the matching algorithm for every suffix of the original query sequence. Unlike the suffix array based algorithms, the suffix tree algorithm already takes into account how many characters have been matched, making it the optimal candidate for searching for small clusters of matching bases without loss in performance.

Overall, suffix array implementations, which can be very optimized for sequential comparisons in exact matching, present a very poor performance when approximate matching is required. On the other hand, suffix trees that were slower in exact matching, have features that make them very efficient for approximate matching. The suffix arrays alternative, even when supported on a hash table and combined with only searching for small clusters present a poor performance in the GPU.
DNA alignment using GPUs

Contents

6.1 Seeding ................................................................. 64
6.2 Smith-Waterman .................................................... 69
6.3 Results and comparison ......................................... 71
6. DNA alignment using GPUs

The optimal local alignment algorithm presented by Smith and Waterman is a dynamic programming algorithm with a runtime complexity of $O(nm)$, where $n$ and $m$ represent the sizes of the sequences being aligned. However, with the huge amount of sequencing data that is currently available, the runtime of this algorithm quickly becomes a bottleneck. A way of using the Smith-Waterman algorithm for DNA sequence alignment is to reduce the size of the sequences being aligned. Therefore, other heuristic algorithms, such as BLAST [20], have been proposed to significantly reduce the alignment time.

![Data flow between the several processing phases.](image)

The heuristic algorithms typically operate in three phases: i) an initial exact match phase; ii) a filtering phase of the potential alignment locations; and iii) a refinement of the obtained score by considering gaps in the alignment.

### 6.1 Seeding

In the previous chapter an algorithm for approximate matching between two DNA sequences was presented. This algorithm extracts from two sequences zones of high similarity which can then be processed by the Smith-Waterman algorithm since they are much smaller in size than the whole original sequence. The number of these sequences can however be very high and as such a filtering step needs to be introduced before they can be passed onto the Smith-Waterman algorithm (gapped alignment phase). This process of filtering the matching zones is named seeding.
6.1 Seeding

6.1.1 Parameters

The process of restricting the DNA alignment to certain zones of the query sequence is considered a heuristic and as with any heuristic the quality of its results varies with the working parameters that guide the choices made by the algorithm.

The first of these parameters is the minimum length of the approximate match to be found during the first phase (seed size). The minimum length for a match to be considered has a significant impact on the total execution time to obtain the alignments. With a smaller minimum length for the match the algorithm is more sensitive, thus presenting better results, but the runtime will also increase significantly. At the lower limit, when the minimum length is a single DNA base, the algorithm returns every position as a possible match. Larger minimum lengths will yield less possible zones to match and if the considered length is too long it might not return any zones at all.

In most cases the optimal DNA alignment encases not only a single zone but a series of consecutive zones, located close together. An algorithm that filters the zones to analyze based on how many zones are in its proximity is capable of filtering out isolated zones that matched in the initial phase while retaining important sequence zones. To decide if two zones are close together the distance between them must be determined and compared against a parameter that defines the maximum distance allowed between two ‘close together zones’. This parameter is denoted as the merge window.

6.1.2 Seeding By Counting

One way of analyzing all the locations returned from the matching phase of the algorithm is to simply store them in a linked list and walk through the list marking in the reference sequence the locations where a match was found. This approach is based on the premise that locations in the reference sequence where there is a high concentration of matches are more likely to be part of a good alignment.

Implementing this solution requires the allocation of an array of integers with as many elements as the number of DNA bases on the reference sequence, the map array. During the transversal of the match list, for each entry on the list the corresponding elements in the map array are marked as a match. This is done by taking from each element its starting location in the reference sequence that directly corresponds to an element in the map array. After finding the symbol (nucleotide) corresponding to the start position, that symbol and all of the posterior ones that belong to the matching section are marked by incrementing the value held in the map. If the parameter set for the minimum length of each match is \( n \) then \( n \) elements are marked. In Figure 6.2(a) the locations of the reference sequence where some part of the query matched are marked in the map.

Once all the locations have been marked, the final phase to filter the locations is to extract the areas which have a high occurrence of matches. By walking through the map and counting the size of the gaps between two match locations the algorithm can decide whether to merge two consecutive locations or not. If the gap size is smaller than the parameter set for the merge window then the two locations are considered as a single one.

In practice, the merge of locations does not require any additional work. During the traversal of the map, the starting position of the current location being considered is kept until a gap larger
6. DNA alignment using GPUs

than the merge window is found. When a gap larger than the merge window parameter defined is
found the algorithm is in a position to decide whether to discard or not the previous location. If the
score for the previous location exceeds a predefined threshold then the location is passed on to
the next phase otherwise it is simply ignored. Figure 6.2(b) shows an example of how merging is
done, seeds A and B are separated by a gap smaller than the merge window and will be extracted
as a single location. Seeds C and D are not within the merge window of any other seed and will
be extracted independently or possibly discarded.

This method is fast and allows for several metrics to be used when deciding the score of a
location: the number of DNA bases that fall within the location, the length of the location or the
average number of DNA bases matching per element. The greatest downside of this method is
that it does not consider the relative order between matching sections.

6.1.3 Seeding By Sorting

The seeding by counting implementation of the filtering phase is very efficient in terms of
computation time, however it allows a very large number of locations to pass on to the final phase
of the DNA alignment algorithm, the gapped alignment phase, which is typically the slowest. Due
to this fact the overall performance of the algorithm decreases.

One way of reducing the number of locations passed on to the final phase of the algorithm
is to only consider, as high-scoring, locations which contain a certain number of merged seeds.
This number, denoted as minimum number of merges, is the key factor in significantly reducing
the number of locations analyzed in the gapped alignment phase.

Merging two seeds is only possible when two conditions are met:

1. the two seeds are in the same order in the reference and in the query sequences;

2. the distance between the end of one seed and the beginning of the merge candidate seed is
   less than a runtime-specified value (calculated both on the reference and query sequences).

In Fig. 6.3(a) it is possible to observe the case in which two seeds will not be merged. In this
example although the first condition is verified, the second condition is violated. In Fig. 6.3 it is
presented an example in which all the conditions are met and thus the two seeds can be merged.
6.1 Seeding

Figure 6.3: Example of merging two seeds.

The seed size parameter (minimum size a match must have during phase 1) used during the approximate matching phase, the merge-window size and the minimum number of merges control the performance and sensitivity of the alignment algorithm.

To implement the described seed filtering procedure [28], in particular to allow for an efficient merge of the seeds, the set of seeds returned by the approximate matching phase must first be sorted as these results are unordered.

6.1.3.A Sorting the seeds

To avoid processing bottlenecks common when sorting a large number of items using lists, the sorting of the seeds is done using auxiliary structures which are afterwards converted into a linked list.

When dealing with the large datasets that are involved in DNA sequence alignment, it is beneficial to use a binary search tree to efficiently sort the elements in the dataset. Furthermore, binary trees can easily be converted into linked lists.

On average, the insertion-sort time of \( n \) elements in a binary search tree is \( O(n \log_2 n) \) and \( O(n^2) \) on the worst case. The worst case situation happens if the elements are added to a binary search tree in order. The results that come from the approximate matching phase are considered to be randomized. Nevertheless, to try to achieve a tree as balanced as possible, a dummy root element is initially added to the structure. This dummy root element has a value that is equal to the average between the maximum and the minimum values that such tree will possibly hold. For example, if the tree will hold the seeds that begin in positions 0 through 999 of the reference sequence, a dummy node of value 500 (\( \frac{999 - 0 + 1}{2} \)). Any additional values the tree node might have is set to zero.

Since a binary search tree with less elements takes, on average, less time to perform the insertion-sort, it is advantageous to have several smaller trees instead of just a larger one. In order to improve performance, several trees are created, each responsible for holding the seeds that occur at a specified interval in the reference sequence. For example, all the seeds that occur between the location index 200 and 399 of the reference sequence will be added to the same tree, while those at location indexes 400 thru 599 will be added to another tree.

A reference to each of the binary search trees created is stored in an index array for speed of access during execution. Since the intervals of the values stored in the trees are fixed the algorithm can easily calculate in which tree it should add a new element and using the index array can efficiently access the tree.
6. DNA alignment using GPUs

6.1.3.B Merging the seeds

After inserting all the seeds in the binary search trees, these are traversed to obtain the final ordered linked list of seeds to be used by the merging algorithm. The several trees are traversed in the order they are found through the index array. During this traversal, the dummy root elements are also removed. The transformation between the binary search trees and the linked list only involves changing memory pointers between the elements of the trees to build a single list, thus avoiding memory allocation and copy operations. An example of the resulting list of ordered seeds is presented in Fig. 6.5.

The merging process is a straightforward process. It uses two iterators: one used for traversing the entire list of seeds; and a second for traversing the same list of seeds, searching for possible merge candidates. The second iterator starts at the element following the element currently selected by the first iterator and stops when it reaches a seed that is outside of the merge-window. When a possible merge is found, the seed pointed to by the first iterator is updated with the new values resulting from the merge (namely the end position in the reference and query sequences) and the seed pointed to by the second iterator is removed. If the second iterator reaches a seed that is outside the merge-window, the first iterator is moved on to the next element. This process occurs until the first iterator reaches the last element of the list.

To achieve the intended goal of removing locations that do not contain the minimum amount of merged seeds, these locations must be removed from the linked list of seeds. This removal is done during the merger of seeds described previously. When the first iterator advances in the list, if the current element does not meet the required criteria to qualify as a high-scoring location it is removed from the list.
Upon the conclusion of this sort and merge procedure, the resulting list contains the high-scoring locations that will be passed to the gapped alignment phase.

### 6.2 Smith-Waterman

The final phase of the heuristic algorithm is to run the selected locations through the Smith-Waterman algorithm in order to find the best alignment possible and to choose the best way to bridge the gaps between the seeds. As there are multiple query sequences each with its own set of high probability locations this problem is very well suited for parallel processing on the GPU, assigning one GPU thread for each of the locations to be considered.

The Smith-Waterman algorithm calculates a matrix of $n \times m$ dimensions and for each cell it has to access the corresponding DNA residue for the two sequences along side with the value stored for the previously calculated top, left and diagonal cells. These five memory accesses per matrix cell can lead to a memory bottleneck and low performance therefore some special consideration must be taken into account when implementing the algorithm.

#### 6.2.1 Kernel modifications

The implementation of the Smith-Waterman algorithm used was based on the already existing CUDASW++ [13].

CUDASW++ is able to align both DNA and protein sequences. Due to this fact, the algorithm uses a $32 \times 32$ matrix to hold the score matrix. To calculate how to score a match or a mismatch the algorithm will pick the cell in the matrix corresponding to current query character (column) and reference character (row).

Before running the alignment algorithm in the GPU, a profile for each query sequence is created. In this profile, each character is replaced by the index of the column of the score matrix corresponding to that character. This allows for the algorithm, when executing on the GPU, to avoid having to compare the character against all columns to discover which one to use. By using such a profile, the algorithm easily knows which column of the score matrix to retrieve.

The presented work is mainly focused on DNA alignment, this final phase (Smith-Waterman alignment) can also be optimized for such scenario. Due to this fact, instead of a score matrix, fixed scores are given to the 4 possible situations: i) match, ii) mismatch, iii) gap open and iv) gap extend. Considering that the range of choices is much smaller and that a match or mismatch have the same score value independently of the considered nucleotide pair, when aligning DNA the score matrix can be quite simplified. Since the score matrix is very simple, there was no need to construct a query profile and instead the real subsequence of the query is copied to the GPU.

Another relevant change when the algorithm is optimized for DNA alignment is that the sequences can be stored using 2-bit encoding. CUDASW++ stores the reference sequence and the query profile using 8-bits per character (since it is necessary to use 5 bits to represent all possible proteins). After optimizing the algorithm for DNA alignment, the sequences are stored using the previously mentioned 2-bit encoding which allows a single 32-bit register to hold 16 nucleotides versus 4 proteins.

To perform the alignment, CUDASW++ uses one loop that reads the elements of the query profile (4 columns). A second, inner loop, reads the reference sequence characters (4 rows in each element read). The actual alignment is done in a third loop, which iterates the values
6. DNA alignment using GPUs

Figure 6.6: Calculation of the score in blocks by the Kernel.

read from the reference sequence, by calculating the alignment score for this 4 block. Since these values have data dependency between themselves they are computed sequentially. In each iteration of the innermost cycle 4 cell values (in the same row) are calculated.

The last difference between the original and the modified version of the algorithm is the loop unrolling. CUDASW++ calculates the alignment in blocks of 4 columns (Fig. 6.6). The algorithm starts by calculating the first 4 cells (A0, B0, C0, D0) for the first row in the innermost cycle. Once these values are calculated the algorithm advances to the next row and repeats the calculation for the new row (A1, B1, C1, D1). If the reference sequence has n characters, this procedure is repeated until the last row (An, Bn, Cn, Dn) is calculated. Once the last row is filled, the algorithm repeats the whole procedure for the following 4 columns.

In the modified implementation, a single integer read yields 16 DNA bases, which allows 16 cells to be calculated in the same cycle. However only 8 cells are calculated in order to reduce the number of registers used by each thread. This lower register usage improves the overall efficiency of the algorithm by allowing more threads to be concurrently executed.

6.2.2 GPGPU Kernel

Initialization of the kernel begins by writing all of the constant values necessary for the algorithm to run in the constant memory area. These values are placed in this area for faster access and include the scores for match/mismatch/gap open/gap extend as well as the number of DNA residues in the reference sequence and in the query locations.

It should be noted that all of subsequences to be passed onto the Smith-Waterman algorithm have the same length (which is a multiple of 16) and the first query residue is aligned to 32 bits (in the original query sequence). This is achieved by adding as many extra bases as necessary to ensure this constraints at the beginning and at the of the subsequence. These constraints allow unnecessary operations and verifications to be avoided (no need to discard unwanted residues and no need to prevent using more residues than the ones available).

After the constants are set, a matrix holding the query subsequences, optimized for coalesced memory access, is transferred to the device memory. Note that there is no need to copy the reference sequence at this stage, due to the fact that it was already copied to the device in an
### 6.3 Results and comparison

Table 6.1: Execution time, in seconds (s), and sum of errors (time/error) for changing seed size and minimum number of merges (fixed merge-window size= 20) when aligning the query sequences with the $250 \times 10^3$ nucleotide long reference sequence.

<table>
<thead>
<tr>
<th>min merges</th>
<th>seed size</th>
<th>5</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8.18/26776</td>
<td>1.32/68499</td>
<td>1.19/78796</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6.65/43120</td>
<td>1.27/79440</td>
<td>1.18/98471</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6.49/71433</td>
<td>1.29/92365</td>
<td>1.18/121008</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6.51/91159</td>
<td>1.25/110537</td>
<td>1.17/135161</td>
<td></td>
</tr>
</tbody>
</table>

earlier phase and it was not deleted.

The kernel execution cycle is composed of four nested cycles: i) The reference sequence is read in blocks of 16 DNA residues; ii) Each block of 16 DNA residues is processed in two sub-blocks of 8; iii) The query sequence is also read in blocks of 16 DNA residues; iv) The cells corresponding to the 8 residues from the reference sequence from (ii) are calculated over 16 rows referring to the 16 DNA residues of the query (iii).

During its execution the algorithm keeps track of the highest score found for each query sub-sequence and returns those values once all the calculations have been completed.

### 6.3 Results and comparison

Both filtering methods described previously were compared, taking into account their performance and quality of results. Furthermore, the results are also compared to the widely used heuristic alignment program BLAST and, to determine their quality, to a straight-forward implementation of the optimal Smith-Waterman (S-W) algorithm.

The quality of the results was assessed using two different values:

- **i)** the first compares the number of obtained scores, using a given heuristic alignment algorithm, that are identical to the scores obtained using just the optimal S-W algorithm;

- **ii)** the second, denoted as sum of errors, represents the sum of the score differences, for each sequence pair, between the result of the heuristic algorithm and the result of the optimal S-W algorithm.

The dataset used to obtain the results in this section consists of two different sized reference sequences, with $250 \times 10^3$ and $1 \times 10^6$ nucleotides, respectively. The dataset used for the query sequences includes 1000 sequences with 200 nucleotides each. The results presented in this section were obtained by aligning the entire set of query sequences to each of the reference sequences. The used system is the same described in section 4.4.

#### 6.3.1 Parameter influence

To determine the effect of the size of the merge window and the seed size parameters in the filtering phase of the algorithm using the seed sorting method, several alignments were performed.

Table 6.1 shows the variation of the time and associated error when varying the seed size and the minimum number of merges (min merges) parameters. The increase of either parameter reduces the algorithm run time, as it has the effect of filtering many more locations. However, the sensitivity of the algorithm (analyzed using the sum of errors) is also reduced.
Table 6.2: Execution time, in seconds (s), and sum of errors (time/error) for changing seed size and merge-window size (fixed minimum number of merges= 2) when aligning the query sequences with the $250 \times 10^3$ nucleotide long reference sequence.

<table>
<thead>
<tr>
<th>merge window</th>
<th>seed size</th>
<th>5</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7.18/32262</td>
<td>1.31/74273</td>
<td>1.21/85273</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>7.65/29206</td>
<td>1.30/71598</td>
<td>1.22/81757</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>8.18/26776</td>
<td>1.32/68499</td>
<td>1.19/78796</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>8.69/23314</td>
<td>1.35/66597</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

The variation of the time and quality of results when changing the seed size and the merge-window size are presented in Table 6.2. It is possible to observe that the sum of errors decreases with the increase in the merge-window size, while the execution time is not significantly changed when considering the larger seed sizes.

In terms of quality of results, the larger merge-window size together with the smaller minimum number of merges and seed size, leads to the best results. However, this has a significant cost in terms of execution time. Specifically, the seed size is the parameter with the most significant effect on the execution time.

### 6.3.2 Seeding by counting versus Seeding by sorting

The results obtained when comparing the previously described seeding methods (observed in Table 6.3) show that execution time is significantly reduced when sorting the seeds. However, the quality of the results is reduced.

If only query sequences with a high degree of similarity are considered, the quality of results of the seeding by sorting method is better than that of the seeding by counting (Table 6.4). This shows that, for high similarity sequences, not only the seeding by sorting implementation attains a smaller execution time but it also presents a higher quality of results.

### 6.3.3 Comparison with BLAST

The DNA alignment algorithm developed was compared to the commonly used BLAST program in terms of error and execution time. For comparison purposes, the program was executed both with the default parameters, which attained the lowest runtime, and with predefined seed sizes equal to those used in the described GPU alignment algorithm.

Table 6.3: Comparison of the execution time, in seconds (s), and sum of errors (time/error) for changing seed size when aligning the query sequences with the $1 \times 10^6$ nucleotide long reference sequence.

<table>
<thead>
<tr>
<th></th>
<th>seed size</th>
<th>5</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seeding by counting</td>
<td>-</td>
<td>4.34/43961</td>
<td>3.09/64847</td>
<td>2.58/84068</td>
</tr>
<tr>
<td>Seeding by sorting</td>
<td>36.27/62370</td>
<td>3.09/64847</td>
<td>2.58/84068</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Comparison of the execution time, in seconds (s), and sum of errors (time/error) for changing seed size when comparing the higher similarity query sequences.

<table>
<thead>
<tr>
<th></th>
<th>seed size</th>
<th>5</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seeding by counting</td>
<td>-</td>
<td>4.34/19751</td>
<td>3.09/7241</td>
<td>2.58/16604</td>
</tr>
<tr>
<td>Seeding by sorting</td>
<td>36.27/7557</td>
<td>3.09/7241</td>
<td>2.58/16604</td>
<td></td>
</tr>
</tbody>
</table>

72
Due to the fact that the BLAST program, running with the default parameters, does not report an alignment score for every query sequence, the comparisons regarding the error of the alignment score only considers the subset of queries that have a score returned by BLAST.

In Table 6.5 it is presented the execution time and the error of the BLAST program and of the GPU alignment algorithm running with different seed sizes for aligning the query sequences to both of the used reference sequences. This table includes the sum of errors for the sequences reported by BLAST with the default seed size (BLAST subset) and the sum of errors for all of the reported alignments (reported). In this table it is also shown the number of query sequences for which an alignment is reported (# alignments).

It is possible to observe that the default seed size of BLAST attains a good result in terms of execution time. Nevertheless, the number of query sequences for which there is a reported alignment is significantly small when compared to all of the other analyzed options. Moreover, the associated error is significantly larger than the error resulting from the other executed programs, with the exception of the GPU algorithm with a seed size of 10.

The results also show that the GPU algorithm executed with a seed of size 8 has a smaller error than BLAST with the default parameters, a smaller execution time (considering the larger reference) and a much higher number of query sequences for which an alignment is reported.

The presented results allow to conclude that the new algorithm is capable of outperforming BLAST both in terms of speed and quality of results. Furthermore, the developed algorithm also allows the user to select the parameters to exploit the balance between quality of results and execution time, thus being flexible enough to be used in a wide range of scenarios.

### 6.3.4 Comparison with MUMmer

MUMmer is a well known alternative to BLAST. While MUMmer also performs DNA alignment through an heuristic algorithm the results reported are in terms of matched bases (percentage of identity) and not alignment score. MUMmer’s strategy is to try and find a maximal combination of small seeds. After finding the combination of seeds that results in the longest sequence (including gaps) MUMmer then aligns the sequences by finding the best way to merge the seeds (bridging the gaps).

<table>
<thead>
<tr>
<th>Alignment tool</th>
<th>BLAST</th>
<th>GPU algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>180.79</td>
<td>83.84</td>
</tr>
<tr>
<td># alignments</td>
<td>999</td>
<td>981</td>
</tr>
<tr>
<td>Error (BLAST subset)</td>
<td>26470</td>
<td>35234</td>
</tr>
<tr>
<td>Error (reported)</td>
<td>25817</td>
<td>32062</td>
</tr>
</tbody>
</table>

Table 6.5: Results of the alignment task using BLAST and the GPU algorithm.

(a) 250000 reference sequence

<table>
<thead>
<tr>
<th>Alignment tool</th>
<th>BLAST</th>
<th>GPU algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>416.93</td>
<td>220.34</td>
</tr>
<tr>
<td># alignments</td>
<td>1000</td>
<td>986</td>
</tr>
<tr>
<td>Sum of errors (BLAST subset)</td>
<td>1920</td>
<td>1934</td>
</tr>
<tr>
<td>Sum of errors (reported)</td>
<td>25817</td>
<td>32062</td>
</tr>
</tbody>
</table>
6. DNA alignment using GPUs

The heuristic employed by MUMmer has configuration parameters that make comparing both MUMmer and the developed algorithm from a quality of results perspective unreliable. However, it is possible to compare the execution time (Table 6.6) for similar working parameters since the goal of both tools is the same: to align two sequences of DNA.

As can be observed from the results presented in Table 6.6 and unlike the behavior of the developed algorithm, the seed size does not have a large influence in the execution time. The main parameter that defines MUMmer’s sensitivity is the separation allowed between matches in a cluster. This value is used during the cluster sorting and filtering phases to calculate the gaps between the seeds that need merging. Increasing the separation from 10 to 15 greatly reduces MUMmer’s running time, although such reduction significantly reduces the number of matches it can identify.

MUMmer’s fastest case presents an execution time that is 2.37 times smaller than the developed algorithm. However, in its slowest case it is 6.42 times slower for comparable seed sizes.

Table 6.6: Results of the alignment task using MUMmer and the GPU algorithm.

(a) 1000000 reference sequence and Separation 10

<table>
<thead>
<tr>
<th>Seed size</th>
<th>MUMmer Time</th>
<th>GPU algorithm Time</th>
<th># Match bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>16.42</td>
<td>36.27</td>
<td>55372</td>
</tr>
<tr>
<td>8</td>
<td>16.25</td>
<td>3.09</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>16.58</td>
<td>2.58</td>
<td>-</td>
</tr>
</tbody>
</table>

(b) 1000000 reference sequence and Separation 15

<table>
<thead>
<tr>
<th>Seed size</th>
<th>MUMmer Time</th>
<th>GPU algorithm Time</th>
<th># Match bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.32</td>
<td>36.27</td>
<td>1859</td>
</tr>
<tr>
<td>8</td>
<td>1.21</td>
<td>3.09</td>
<td>1859</td>
</tr>
<tr>
<td>10</td>
<td>1.09</td>
<td>2.58</td>
<td>1858</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Conclusions

Contents

7.1 Future work ............................................................. 76

75
7. Conclusions

The solutions considered for the exact matching problem and the approximate matching problem as well as the results obtained from them show that although similar these problems do not share one method as best.

Exact matching is at its core a simple searching problem and can be solved by multiple search algorithms already developed. However, to solve the problem on the GPU, as opposed to solving it on a typical CPU, some additional steps must be taken. GPUs cannot efficiently access the host memory, this means that to perform any sort of computation on data present on the host memory, this data must first be copied over to the GPUs from the host. Additionally, for the CPU to access the results produced by the GPU this information must be copied to the host memory. These two limitations decrease the gains achieved from executing the search algorithm on thousands of threads concurrently. This leads to the optimal solution found to be the one which relies on the most compact data representation since it means less time is spent copying data.

The most compact data representation is not the best solution when dealing with approximate matching. Departing from a simple search and heading into more complex searches requires additional information than that which is provided by short and compact structures indexing the data. Results showed that providing the extra information, which was decreasing performance in exact matching, actually improved the performance in approximate matching by allowing faster matches following the first.

The size of the reference sequence index is extremely important. However, it is not the only factor to consider when implementing a GPGPU application. Due to the constraints in memory access, both the structure of the reference sequence and the structure holding the query sequence data must be considered. The GPU will typically execute thousands of threads concurrently, which will generate thousands of memory access instructions. If the data structures used by the algorithm (the query sequences and the reference sequence index) are not setup in a way to minimize the memory access instructions generated the performance of the algorithm will suffer a performance penalty.

Taking the information learned from the tests conducted in exact and approximate matching and combining it with current heuristic algorithms allowed to develop a DNA alignment algorithm, running on the GPU, which results showed can compete with current techniques in both quality and speed.

It is expected that, in the next years, as technology allows for more and faster cores to be included in the GPU and as better and faster memory chips are developed, the GPUs will become the standard for parallel processing in problems suitable to the SIMT programming model. The algorithm developed is in a position to benefit, effortlessly, from such advances.

7.1 Future work

The presented work has the potential to be further improved by applying the studied double-buffering schemes to both phases of the DNA alignment algorithm that run on the GPU due to the fact that both the approximate matching phase and the Smith-Waterman alignment phase can benefit from concurrent communication and computation. Furthermore, it is also possible to adapt the algorithm in such a way that the CPU continues to process data while waiting for work being done on the GPU to finish. When these two optimizations are combined, they may provide even further advantages over the traditional techniques, especially when considering the hardware
7.1 Future work

improvements over the next years, which will likely yield faster memory and better memory access rates.

Another important aspect that can be improved is the filtering phase. There are many heuristics for sorting the seeds obtained from the initial search phase which can lead to better quality of results and performance.
7. Conclusions


