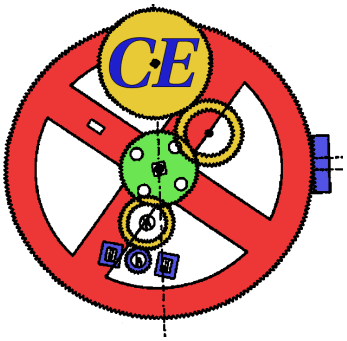


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Biological Sequence Alignment Using Graphics Processing Units

M.A. Kentie

CE-MS-2010-35

Abstract

Alignment algorithms are used to find similarity between biological sequences, such as DNA and proteins. By aligning a sequence with a database, similar sequences can be found. These can be used to identify the source of a query sequence, to find commonalities between organisms, or to infer an ancestral relation. Various methods of performing biological sequence alignment exist, including dynamic programming and heuristic methods. Dynamic programming methods are guaranteed to find all optimal alignments, but are relatively slow; heuristic methods are faster but less precise.

This thesis investigates the acceleration of one such optimal algorithm, the Smith-Waterman local sequence alignment algorithm, by using graphics processing units (GPUs). A fully functioning GPU-based protein database search tool was designed, implemented and optimized. The optimizations mostly concern the elimination of memory bottlenecks and the conversion of the database to a format well suited for GPU use. The final implementation offers the same features its CPU-based counterparts do, such as user configurable scoring and substitution matrix settings, and includes a web interface for convenient and remote usage.

The performance of the GPU accelerated implementation was evaluated and compared to other solutions. It was found to attain a performance of more than 21 GCUPS (giga cell-updates of the Smith-Waterman score matrix per second) when searching the October 2010 release of Swiss-Prot on an NVIDIA Geforce GTX 275 GPU. With this performance, it is the fastest known GPU implementation on comparable hardware. It is also faster than the BLAST heuristic. However, the cost of purchasing a GPU, its power consumption, and the relative difficulty of maintaining a GPU software product are disadvantages of GPU acceleration.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Biological Sequence Alignment Using Graphics Processing Units

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

M.A. Kentie
born in Guildford, England

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Biological Sequence Alignment Using Graphics Processing Units

by M.A. Kentie

Abstract

Alignment algorithms are used to find similarity between biological sequences, such as DNA and proteins. By aligning a sequence with a database, similar sequences can be found. These can be used to identify the source of a query sequence, to find commonalities between organisms, or to infer an ancestral relation. Various methods of performing biological sequence alignment exist, including dynamic programming and heuristic methods. Dynamic programming methods are guaranteed to find all optimal alignments, but are relatively slow; heuristic methods are faster but less precise.

This thesis investigates the acceleration of one such optimal algorithm, the Smith-Waterman local sequence alignment algorithm, by using graphics processing units (GPUs). A fully functioning GPU-based protein database search tool was designed, implemented and optimized. The optimizations mostly concern the elimination of memory bottlenecks and the conversion of the database to a format well suited for GPU use. The final implementation offers the same features its CPU-based counterparts do, such as user configurable scoring and substitution matrix settings, and includes a web interface for convenient and remote usage.

The performance of the GPU accelerated implementation was evaluated and compared to other solutions. It was found to attain a performance of more than 21 GCUPS (giga cell-updates of the Smith-Waterman score matrix per second) when searching the October 2010 release of Swiss-Prot on an NVIDIA Geforce GTX 275 GPU. With this performance, it is the fastest known GPU implementation on comparable hardware. It is also faster than the BLAST heuristic. However, the cost of purchasing a GPU, its power consumption, and the relative difficulty of maintaining a GPU software product are disadvantages of GPU acceleration.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-35

Committee Members :

Advisor:	Zaid Al-Ars, CE, TU Delft
Advisor:	Laiq Hasan, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Cor Meenderinck, CE, TU Delft
Member:	Jeroen de Ridder, Bioinformatics, TU Delft

Contents

List of Figures

Acknowledgements

I would like to thank Dr. Zaid Al-Ars for his supervision and flexibility in allowing me to do work I found interesting. I would like to thank Laiq Hasan for his ideas and practical feedback at all stages of my work. I would like to thank Diane Butterman for her proofreading work and for improving my written English in general. Finally, I would like to thank Cor Meenderinck and Jeroen de Ridder for being on my graduation committee.

M.A. Kentie
Delft, The Netherlands
November 29, 2010

1 Bioinformatics

1.1 A recap of molecular biology

What follows is a short recapitulation of the basics of molecular biology. Although no in-depth knowledge is required of the chemical processes involved, subjects such as DNA and protein construction are critical to understanding the relevance of sequence alignment, the procedure upon which much of this thesis is based. The information in this section largely comes from [37] and [57].

1.1.1 Cells, amino acids and proteins

All living organisms consist of one, or many more, examples of a basic functional unit: the *cell*. Classified as being 'alive' (the smallest organisms consist of a single cell), cells can process and excrete molecules (metabolism), alter their electrical potential and procreate by cell division. Many of the processes inside cells are governed by *proteins*. Proteins are complex chains of molecules known as *amino acids*. Some amino acids, the 'non-essential' ones, can be synthesized by the cell. The other, essential, amino acids must be procured through the ingestion and breakdown of proteins in foods such as meat. Again, this breaking down of food products is performed by proteins, this time existing outside any cell.

Proteins have a wide array of functions: for instance *actin* aids muscle contraction while the proteins of the *cytoskeleton* form a cell's 'skeleton', giving it its shape and protecting it. Another important role of proteins is to act as catalysts; these proteins are called *enzymes*. Enzymes act as catalysts by binding to the reagents of a reaction and lowering the activation energy required for it to take place. Designed to only be compatible with those specific reagents due to their structure, enzymes are not consumed in the reaction and can be reused. To return to the example of breaking down food into nutrients, there are enzymes which split proteins into their component amino acids, enzymes which break down fat molecules, and enzymes that allow ingested nucleic acid to be reused for the construction of DNA.

1.1.2 Chromosomes and DNA

It is obvious that cells require the presence of proteins, both internally and externally, to survive. In fact, the reproduction of cells relies heavily on proteins too, such as those of the aforementioned cytoskeleton, which facilitate the division of the cell membrane. Proteins are created, from scratch and to specification, within the cell itself. This is where *deoxyribonucleic acid* (DNA) comes in. DNA is stored in structures called *chromosomes*. Made up of the DNA molecules and a supporting protein packaging, chromosomes are 'wadded up' in the cell similar to a ball of string. Attached to these chromosomes, the

DNA is protected and more compact; in this way it is able to fit in the cell (nucleus). The structure and number of chromosomes varies on a per species basis, and the shape of the chromosomes is also determined by which life cycle stage the cell currently resides in. Figure 1.1 shows a bacterium cell with its single chromosome at the center.

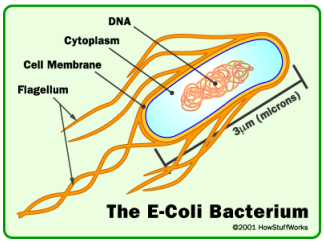


Figure 1.1: The single-cellular E. coli bacterium.

Source: howstuffworks.com [38].

The DNA itself contains the genetic instructions that describe how the various proteins should be constructed. The structure of DNA is shown in Figure 1.2. DNA consists of two long, coiled nucleotide polymer strands that take the familiar double-helix form. These polymers are strengthened by a skeleton of sugars and phosphate groups; connected to these sugars are the *bases*, pairs of molecules that specify the genetic code. One end of each strand of DNA is called the *5' end* (pronounced *five-prime*) while the other end is called the *3' end*. These ends are named after the carbon of the deoxyribose structure which they link up to. Due to the anti-parallel nature of the strands, their ends are mirrored: one strand's 3' end is matched by the other strand's 5' end. When talking about DNA, these ends can be used to indicate in which direction a strand is being built/interpreted/etc.

Four different bases exist: *adenine* (A), *cytosine* (C), *guanine* (G) and *thymine* (T). A base on one strand will be matched by and connected to its twin base on the other by means of hydrogen bonds. Adenine is always paired with thymine; cytosine and guanine form the second combination. This duplication of the bases is central to the replication of DNA, and as such that of the cell and the survival of the host organism.

During the replication of DNA mutations can occur, thus altering the genes of a host organism. This ties into the theory of evolution and the field of phylogenetics, the study of relatedness among organisms by comparing their genetic makeup.

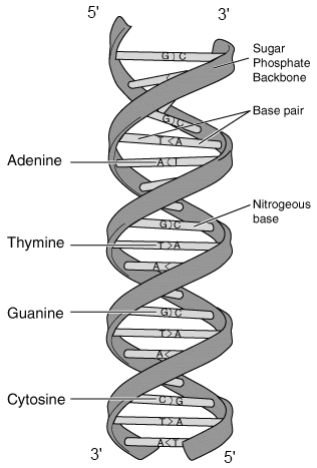


Figure 1.2: The structure of DNA.

Source: National Institute of Health [67].

1.1.3 RNA and transcription

The bases of DNA can be seen as letters (A, C, G and T). These letters are com-

bined together in groups of three to rep-
resent 'words' called *codons*. Each codon describes a single amino acid (as discussed, amino acids are the building blocks of proteins). A portion of DNA that codes for a protein is known as a *gene*. As there are four bases and as they appear in words of three, there are 4^3 possible codons. However, only 20 different amino acids are encoded. This means that some words code for the same amino acid. Additionally, some words have special functions: the start (ATG) and stop (TAG, TAA, TGA) codons function as markers to aid in the correct interpretation of the code at the *ribonucleic acid* (RNA) stage. A sequence of codons that starts with a start codon and ends with a stop codon is called an *open reading frame*.

The process of interpreting the genetic code and using it to synthesize proteins is called *genetic expression*. The first step is the generation of RNA, which will mirror the gene in question and be transported to the cell's 'protein factory'. Genes are *transcribed* to RNA by an enzyme called *RNA polymerase*; this is bound to the correct place on the DNA by means of a *promoter*, which is a sequence of codons that influence the binding of RNA polymerase directly or indirectly by means of proteins. The DNA strand that the RNA will be based on is called the *coding strand*; this can be either of the strands, depending on the gene in question. When generating RNA, the strands are separated and the complementary strand, called the *template strand*, is traversed in the 3' $\rightarrow$ 5' direction. The strand's bases are then paired with a new strand of again complementary bases (with thymine replaced by *uracyl* (U)). This is the RNA. This strand is separated from the DNA once transcription is complete, after which the DNA's structure is restored. In effect, the created RNA is a copy of the coding strand with T replaced by U.

Example: consider a strand of DNA coding a gene:
5' A T G G C C T G G A C T T C A ... 3' coding strand
3' T A C C G G A C C T G A A G T ... 5' template strand

The resultant RNA will then be:
5' A U G G C C U G G A C U U C A ... 3'

Note the start codon ATG (AUG for the RNA).

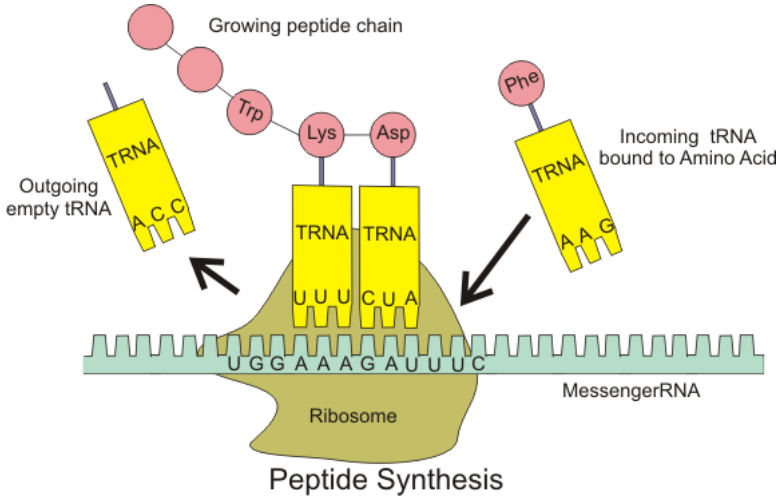


Figure 1.3: A ribosome building a peptide (amino acid chain).

Source: Wikipedia [34].

This RNA is transported to the *ribosomes*, the cell components which assemble proteins by chaining together amino acids. Figure 1.3 shows a ribosome in action. Here the RNA is traversed and interpreted from the start to the stop codon. The codons are interpreted by means of *transfer-RNA* (tRNA). These tRNA molecules carry amino acids to link up to the protein peptide chain. By having a structure complementary to that of the codons, they are matched to the sequences that code for the amino acids they are carrying. The ribosomes themselves (again) consist of proteins and ribosomal RNA.

This review of genetic expression glosses over many things. Although the process is more involved than described here, especially in humans, more information is not required to appreciate the workings of sequence alignment.

1.2 Bioinformatics and sequence alignment

The field of bioinformatics involves the development of algorithms and software that can analyze huge amounts of biological data and automate previously labor-intensive tasks. It is also the development of tools, for example with which to view 3D models of biological structures.

Examples of bioinformatics goals and research are the determination of a protein's shape and function from a sequence of amino acids (protein folding), the sequencing (i.e. mapping) of genomes, the construction of evolutionary trees and the determination of protein functions [37].

1.2.1 Biological sequences

An important aspect of bioinformatics is the analysis of DNA and protein sequences.

DNA

As discussed in Section 1.1.2, DNA consists of the four bases A, C, G and T. One might say that DNA is a sequence, or string, of the alphabet {A,C, G,T}. Not surprisingly, RNA can be looked at similarly, with the alphabet {A,C,G,U}.

Proteins

Proteins, too, can be viewed as strings of an alphabet. In this case, the alphabet of the 20 amino acids {A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y}. The amino acids corresponding to these letters are shown in Table 1.1.

Table 1.1: The 20 amino acids

Letter	Amino acid	Letter	Amino acid
A	Alanine	L	Leucine
R	Arginine	K	Lysine
N	Asparagine	M	Methionine
D	Aspartic acid	F	Phenylalanine
C	Cysteine	P	Proline
Q	Glutamine	S	Serine
E	Glutamic acid	T	Threonine
G	Glycine	W	Tryptophan
H	Histidine	Y	Tyrosine
I	Isoleucine	V	Valine

1.2.2 Sequencing

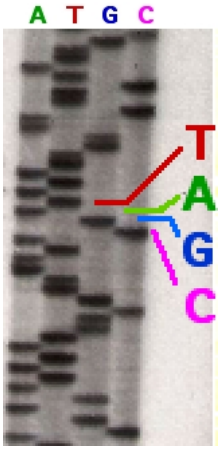


Figure 1.4: Chain-sequencing result strip; the dark bands show dideoxy bases.

Source: John Schmidt [54].

The nucleotide makeup of DNA/RNA and the amino acid makeup of proteins can be determined by methods known as *sequencing*. Once a sample has been sequenced, its letter sequence is known and it can, from a bioinformatics perspective, then be treated like the aforementioned sequences or strings. There are multiple approaches to both types of sequencing.

A popular approach for DNA is the chain termination method [57][54], in which the DNA in question is mixed with: a 'primer' strand of DNA; DNA polymerase; the four bases; and a dideoxy version of one base. The mixture is heated until the DNA separates; the primer DNA then attaches to the now separate coding strand and DNA polymerase initiates the copying process, effectively recreating the template strand. However, if one of the dideoxy bases is used (quite how often this happens depends on the ratio between the amount of normal and dideoxy variant present), the new strand cannot be extended and is terminated. The process is repeated until replication has terminated at all possible points. The lengths of the new strands then reveal where the base, of which the dideoxy version was, used is present. The process is then, in turn, repeated for each of the four bases. The locations of the dideoxy bases can be shown by means of gel electrophoresis; Figure 1.4 gives an example, the type of which might well be familiar from popular media.

Chain termination sequencing can work with sequences of up to around 900 bases. For longer sequences, approaches such as shotgun sequencing are used: random pieces are sequenced after which the results are stitched together in a process known as *sequence assembly*, which is another area of bioinformatics. For more information on protein sequencing, see [53].

1.2.3 Sequence alignment

If two DNA, RNA or amino acid sequences are similar, there is a chance that they are *homologous*. Homologous sequences share a common ancestral sequence and their relative differences are the result of mutations. These mutations might manifest themselves in various ways: as *substitutions* where one symbol is replaced by another, *insertions* where a new symbol is inserted into the sequence, and *deletions*, the removal of a symbol. To establish the degree of homology, the sequences are *aligned*: lined up in such a way that the degree of similarity is maximized. Some applications of sequence alignment will now be given:

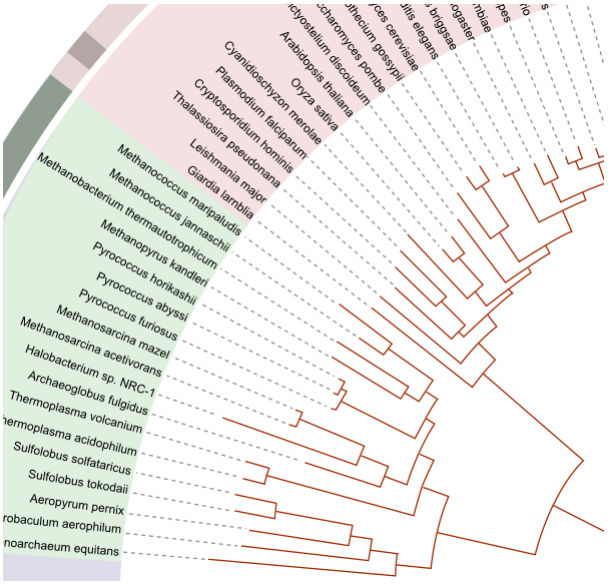


Figure 1.5: Part of a phylogenetic tree based on sequenced genes.

Finding homology

One of the main purposes of sequence alignment is to find homology. Homology means that two sequences share a common ancestor; evolution dictates that all cells must eventually come from the same ancestor. Finding homology between organisms might enable knowledge of one to be applied to the other, or to infer the function of one organism's gene from that of a related species.

Determining the origin of a sequence

If a DNA or protein sample is recovered but its originating species is unknown, sequence alignment can be used to find likely sources, that is to say, the known sequences that most closely match the sample.

Finding specific sequences

Suppose that we have discovered the function of part of species X's genetic code. It might then be attractive to search species Y's code for the sequence. If something similar is found, it might give clues as to the location of a similar gene in Y [37]. Similarly, let us suppose that we have perhaps found the piece of code that expresses a trait, such as a physical characteristic or the presence of a genetic disease, in one piece of genetic code. Searching other pieces known to either feature or lack this trait might help validate or disprove the theory.

Constructing Hidden Markov Models

Hidden Markov models (HMMs) are probabilistic tools which can be used for sequence alignment [39], to find sequences in genetic code [37], to infer protein structure [52] or to build profiles of DNA and proteins [52]. Such profile HMMs can be used to determine whether a sequence is part of a family of DNAs or proteins.

HMMs are based on the probability of a sequence adhering to certain characteristics. To determine these probabilities, machine learning principles are used; in the case of a profile HMM, the learning set is a multiple alignment. For more information on HMMs as sequence alignment tools see Section 2.3. For a more in-depth description of the workings of (profile) HMMs, see Chapter 3 of [52].

Constructing evolutionary trees

From homology data, evolutionary (*phylogenetic*) trees can be constructed [52]. These trees are built using the 'genetic distance' between species and give insight into inter-species relationships and the course of evolution. Using the concept of an *evolutionary rate*, the species' sequence homology can be translated into the time taken to develop from ancestral species. The actual construction of the tree can be done in many ways; examples include maximum parsimony methods (building the tree so that the lowest amount of evolutionary change is required) and distance methods such as the UPGMA algorithm which builds the tree from the result matrix of a multiple alignment. Figure 1.5 provides an example of a phylogenetic tree.

1.2.4 Types of sequence alignment

1.2.4.1 Structural alignment

Structural alignment [50] is the process of attempting to infer similarity between proteins by comparing their three dimensional shapes, or *tertiary structures*. As a protein's shape is determined by its amino acid makeup which, in turn, determines its function, it is obvious that structural alignment is an attractive tool for homology research. In fact, different protein letter sequences might result in similar 3D structures [50]: from an evolutionarily point of view, protein structure is better conserved than protein sequence [35].

Unfortunately, the determination of the tertiary structure of proteins requires costly, time consuming procedures such as X-ray crystallography and nuclear magnetic resonance imaging (bioinformatics databases contain far fewer protein structures than letter sequences) [37]. One field of bioinformatics concentrates on unraveling the mysteries of *protein folding*, the process in which an unfolded *random coil* amino acid gains its characteristic tertiary structure. Using computational protein folding, any of the myriad available protein sequences could be converted to a 3D representation. Then, in turn, structural alignment could be used to infer homology. Currently, however, protein folding is still an open-ended problem, and current approaches have such high computational requirements that researchers have turned to super or distributed computing [58].

Although mainly used for proteins, structural alignment is also promising for strands of RNA [49]. It is not suitable for DNA as that always has the double-helix structure.

1.2.4.2 Global alignment

The following alignment methods operate directly on the sequence letters. As stated, the idea is to line up two (or more) sequences so that their degree of similarity is maximized.

For DNA and RNA this means matching identical bases; in the case of proteins, amino acids are matched if they are identical or can be derived from one another through substitutions that are likely to occur [37]. Although matching two sequences directly will take into account substitution mutations, to handle insertions and deletions the notion of *gaps* is introduced. Marked by the symbol '-', a gap can be chosen to be inserted into any of the sequences to obtain a closer match.

Example (from [52]). With base sequences TACCAGT and CCCGTAA:

No gaps	Gaps
T A C C A G T	T A C C A G T – –
C C C G T A A	C – C C – G T A A

The alignment with gaps is more relevant and better exposes the similarities between the sequences. Note that other alignments are possible: an option would be

T A C C A G T – –
– – C C C G T A A

As multiple alignments are possible even in this simple case, it makes sense to devise a way to rate and then select the best alignment(s). A simple way to accomplish this is to assign scores to the alignment letters. A simple scheme is 1 for a match, -1 for a mismatch and -2 for a gap. Such a scheme is said to have a *linear gap penalty*. A more advanced method is to introduce an *affine gap penalty*; this assigns different scores to the starting of a new gap and the extension of a current one. Generally, starting a new gap is given the largest penalty as this is biologically the hardest [52]. Using the aforementioned linear scoring system, the first gapped alignment scores (-1-2+1+1-2+1+1-2-2)=-5 and the second option does so as well with (-2-2+1+1-1+1+1-2-2)=-5. So in this case, both gapped alignments are 'as good as' one another. However, this does not automatically mean that they both have the same biological relevance. To judge how relevant an alignment's score is, probabilistic methods can be used; the idea is to check whether the probability of an alignment attaining the score in question is adequately small (see Chapter 7 of [52]). Note that if an affine gap penalty system had been used, the second alignment would have had the best score as it contains two gaps instead of three.

As noted, the same approach can be used for amino acids as opposed to DNA bases. Instead of working with fixed scores, amino acid substitutions have been rated by their evolutionary likeliness and are available as standard 20x20 triangular *substitution matrices*. The two most well-known matrices are the PAM and BLOSUM families [37]. Figure 1.6 shows the BLOSUM62 matrix.

The *Needleman-Wunsch algorithm* (Section 2.1.1) performs global alignment.

1.2.4.3 Local alignment

Local alignments are similar to global ones, only instead of attempting to align the complete sequences to one another, portions of similarity are aligned.

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	-1	5																		
Asn	-2	0	6																	
Asp	-2	-2	1	6																
Cys	0	-3	-3	-3	9															
Gln	-1	1	0	0	-3	5														
Glu	-1	0	0	2	-4	2	5													
Gly	0	-2	0	-1	-3	-2	-2	6												
His	-2	0	1	-1	-3	0	0	-2	8											
Ile	-1	-3	-3	-3	-1	-3	-3	-4	-3	4										
Leu	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4									
Lys	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5								
Met	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5							
Phe	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6						
Pro	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7					
Ser	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4				
Thr	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5			
Trp	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11		
Tyr	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	
Val	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Figure 1.6: The BLOSUM62 amino acid substitution matrix.

Example: with sequences GTGTACTCCAGAG and GTACCCAAG:

Global alignment	Local alignment
G T G T A C T C C A G A G	G T G T A C T C C – A G A G
G – – T A C – C C A – A G	– – G T A C – C C A A G – –

Looking for a local alignment will better expose 'patches' of homology in two relatively dissimilar sequences; this might lead to more biologically relevant results [52].

The *Smith-Waterman algorithm* (Section 2.1.2) performs local alignment.

1.2.4.4 Multiple alignment

The previous examples focused on aligning only a pair of sequences, but in some cases it might be more interesting to consider the similarities between a group of sequences. For example, if the structure of a protein is unknown, a similarity to a group of other proteins might give clues. Global and local alignment algorithms can be adapted to deal with multiple alignments, though this quickly becomes extremely computationally expensive. An alternative is to use specifically designed heuristic algorithms; an example is *ClustalW* (Section 2.2.3).

1.3 Bioinformatics databases

Various databases and search engines for biological data, such as protein and DNA sequences, exist. This section takes a look at the major offerings. Bioinformatics databases are incredibly useful for research purposes as they give researchers access to huge amounts of data, which can be searched, inspected or used for (multiple) sequence alignment. This is critical as, for example, searching for homology is quite irrelevant without a large amount of species to compare to. The databases also allow researchers to submit data and share it with the rest of the community. When a biological sequence is submitted to a database it is assigned an *accession number*, a unique global identifier that allows the sequence to be updated when needed and referred to in publications.

1.3.1 International Nucleotide Sequence Database Collaboration

The International Nucleotide Sequence Database Collaboration (INSDC) [16] is a collaboration between the three major global bioinformatics database providers: NCBI, EBI and DDBJ. Data gets shared between the three so that each is more complete, has less visitors to serve and can apply its search engine(s) to as much data as possible.

1.3.1.1 NCBI

The American National Center for Biotechnology Information (NCBI) [17] offers the GenBank database, which is "a collection of publicly available annotated nucleotide sequences, including mRNA sequences with coding regions, segments of genomic DNA with a single gene or multiple genes, and ribosomal RNA gene clusters." [13]. Data is submitted by third parties and, though some quality control checking is done, it is mostly up to the author to check and revise any submitted data. Database entries contain a description, protein codon ranges and amino acid translations for DNA/RNA and finally sequence data in a simple text format e.g. 'atgtagc'. GenBank nucleotide sequences that code a protein will be annotated with the protein's name and a link to its amino acid sequence data. GenBank data is exchanged daily with the other INSDC participants.

The NCBI also offers the Reference Sequence (RefSeq) database, a curated collection of DNA, RNA, and protein sequences. Curated means that its contents is hand-picked and carefully annotated with details such as gene functions by the NCBI. As opposed to GenBank, which can contain multiple variants of the same record, RefSeq only offers a single, 'best' one, which is updated and annotated as new data becomes available.

1.3.1.2 EBI

The EBI [10] (European Bioinformatics Institute)'s EMBL-Bank database is similar to GenBank. Its data is shared with INSDC. The EBI hosts the ClustalW2 multiple alignment tool, see Section 2.2.3.

1.3.1.3 DDBJ

DDBJ (DNA Data Bank of Japan) [9], the sole nucleotide data bank in Asia, is the third of the INSDC participants and, again, similar to the others.

1.3.2 UniProt

UniProt (Universal Protein Resource) [29] is a curated protein sequence database; no DNA or RNA data is kept. UniProt gets its protein sequences from the INSDC databases, but further processes the data for one of two sub-databases: Swiss-Prot and TrEMBL. Swiss-Prot contains manually curated and annotated proteins, and is considered to be the gold standard for protein information [66]. Of course, the high quality of Swiss-Prot entries means that only a fraction of available proteins are documented there. However, UniProt's second database, TrEMBL, does offer all other proteins available, although with lower quality, automatically generated annotations. Apart from getting data from the INSDC databases, UniProt annotations also link to their entries.

1.3.3 Search engines

As there are massive amounts of data available in aforementioned bioinformatics databases, various ways of accessing them are offered. Users can retrieve entries by accession number, browse by taxonomy or use a variety of search engines to find relevant sequence data. Appendix A illustrates the use of these tools with a practical example.

1.3.3.1 Database content search engines

All databases offer plain-text search engines where users can search for terms such as 'tissue inhibitor' or 'mouse'. NCBI offers the Entrez search engine, EBI has EB-eye. DDBJ has ARSA. These search engines allow one to sort results by category such as 'nucleotide sequences', 'protein sequences' or 'enzymes', and in the case of ARSA also search UniProt. UniProt's own search engine obviously only searches proteins and allows filtering by Swiss-Prot or TrEMBL results. Figure 1.7 shows an example Entrez search for 'mouse'. Note how the results include nucleotide data, three dimensional structures and species information.

1.3.3.2 Database alignment search engines

In many cases one will want to search the databases not for plain-text but for (similarity to) a sequence. Being able to perform alignments with all known sequences makes for a very powerful tool. To do this, all of the discussed databases offer web interfaces to the FASTA and/or BLAST local sequence alignment tools. The user simply enters a nucleotide or amino sequence and sets some parameters if desired. A heuristic-based sequence alignment tool will then search the entire database and present a ranked list of results: the sequences that managed to attain the highest scores. Figure 1.8 shows the settings available to FASTA users; some might be familiar from Section 1.2. Note that in this instance the EBI's search engine also allows searches of the UniProt database, once again an example of how closely tools and data are shared. The FASTA algorithm is discussed in Section 2.2.1; BLAST in 2.2.2.

The databases also offer the Clustal multiple alignment tool. With this multiple sequences (if desired, from the database using accession numbers) can be aligned. The Clustal algorithm is discussed in Section 2.2.3. Appendix A shows an example of how the databases are integrated and the usage of their search tools.

Figure 1.7: An Entrez (NCBI) search for 'mouse'.

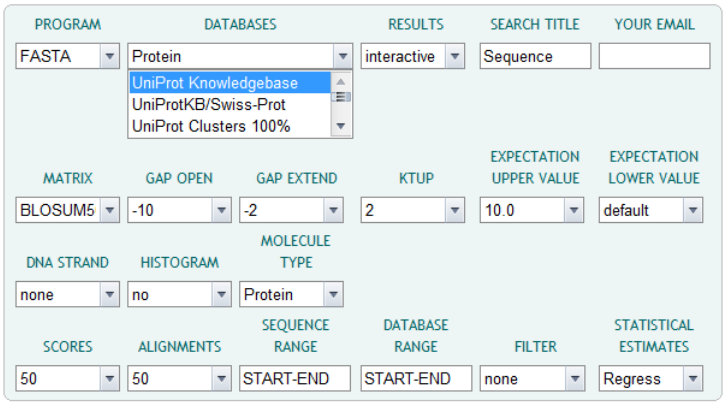


Figure 1.8: The settings of the EBI's FASTA alignment tool.

2 Sequence alignment algorithms

This chapter covers algorithms that attempt to determine the optimal local or global alignment(s) of two sequences (for more on sequence alignment, see Section 1.2.4). Figure 2.1 shows an overview of the various sequence alignment algorithms. There are two classes of algorithms; the first class, the dynamic programming algorithms, will always recover all optimal alignments. However, these algorithms are computationally expensive and searching a database with them would require every single database entry to be run through the algorithm, which is prohibitively expensive. The second class, the *heuristic* methods, are much faster (and are used in database searches) but are not guaranteed to find the optimal alignment(s); interesting alignments might be overlooked. Heuristic methods can be based either on dynamic programming or probabilistic methods. For both optimal and heuristic algorithms speedups are, of course, desired: the best case would be the ability to use the optimal dynamic programming algorithms for database searches.

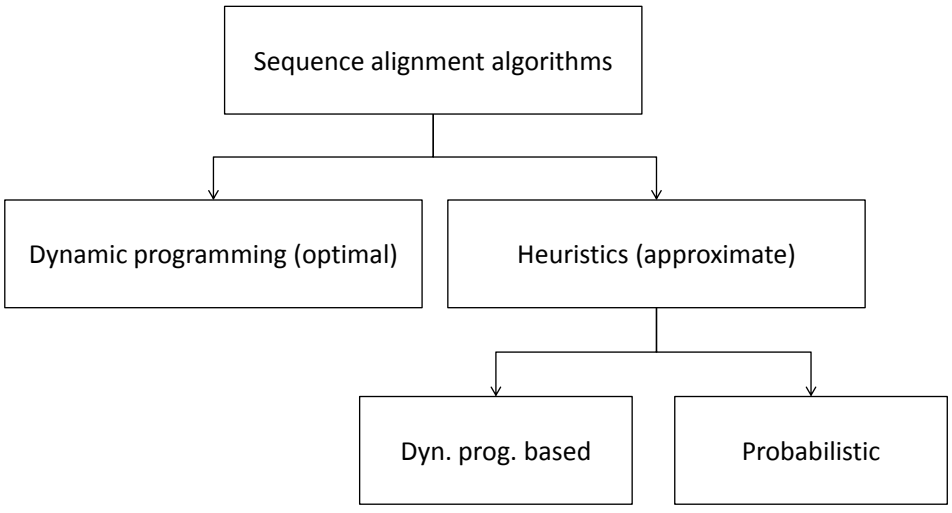


Figure 2.1: Classification of sequence alignment algorithms.

2.1 Dynamic Programming algorithms

Dynamic Programming (DP) is the approach of breaking down a problem into smaller subproblems. This section shows how the Needleman-Wunsch (global alignment) and Smith-Waterman (local aligment) algorithms use DP for sequence alignment by producing an optimal alignment from a matrix of subsequences.

Note that these algorithms perform pairwise alignment; as stated in Section 1.2.4.4, they can be used to 'brute-force' multiple alignments but the sheer amount of work involved due to this method's exponential complexity means that specifically designed heuristic methods (Section 2.2.3) are preferred.

The content in this section is largely based on Chapter 2 of [52], a recommended reference on these algorithms.

2.1.1 The Needleman-Wunsch algorithm

Two versions of the Needleman-Wunch algorithm exist: one with linear and one with affine gap penalties (see Section 1.2.4 for information on gap penalties). Both will now be described.

2.1.1.1 Linear gap penalties

The Needleman-Wunsch algorithm [68] has three phases: *initialization*, *matrix fill* and *traceback*. Suppose we have a sequence x of length n and another sequence, y, of length m. The algorithm operates on an $(n+1) \times (m+1)$ matrix F. For each of the matrix elements for $i, j > 0$ the (i, j)th element is the score of an optimal alignment between $x_{1..i}$ and $y_{1..j}$. For $i = 0$ the (i, j)th element is the score resulting from aligning $y_{1..j}$ to a gap of length j and analogous to that for $j = 0$ the (i, j)th element is the score resulting from aligning $x_{1..i}$ to a gap of length i. These scores are calculated as in Section 1.2.4, using scoring rules for nucleotide sequences and, additionally, scoring matrices for proteins.

Initialization

The $(0, 0)$th matrix element is set to 0. The top row and and leftmost column are initialized with the costs of gaps of lengths i and j. Figure 2.2 shows the initialization phase for the nucleotide sequences GAATCT and CATT with gap penalty 2.

Matrix fill

Next the matrix is processed from the top left to bottom right corner. The exact approach does not matter (row by row, column by column, alternating rows and columns) as long as the other elements on which the current cell depends are available. To assign a score to each matrix element (i, j), three possible options are considered: aligning i with j, aligning i with a gap and aligning j with a gap. Respectively, this translates to continuing on from the $(i, j-1)$th, $(i-1, j)$th and $(i-1, j-1)$th elements and adding the proper score. The highest-scoring option is taken to ensure an optimal alignment. In formula form:

	0	1	2	3	4
F	–	C	A	T	T
0 –	0	-2	-4	-6	-8
1 G	-2				
2 A	-4				
3 A	-6				
4 T	-8				
5 C	-10				
6 T	-12				

Figure 2.2: Needleman-Wunsch initialization phase.

This example has been adapted from Example 2.1 in [52].

$$F(i,j) = max \begin{cases} F(i-1,j-1) + s(x_i, x_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases} \tag{2.1}$$

With $s(x, y)$ the score function for the alignment of the elements, which will be match/mismatch for nucleotides and a matrix lookup for proteins. d is the gap penalty. Figure 2.3 shows the example alignment after the matrix fill phase. The gap penalty is still -2, the score for an alignment is 1 and a misalignment is -1. Once the matrix fill phase is complete, the bottom right element, (n, m), holds the score of the alignment: in this case, -2.

	0	1	2	3	4
F	–	C	A	T	T
0 –	0	-2	-4	-6	-8
1 G	-2	-1	-3	-5	-7
2 A	-4	-3	0	-2	-4
3 A	-6	-5	-2	-1	-3
4 T	-8	-7	-4	-1	0
5 C	-10	-7	-6	-3	-2
6 T	-12	-9	-8	-5	-2

Figure 2.3: Needleman-Wunsch matrix fill phase.

Traceback

During the matrix fill phase it is recorded which of the three options was chosen for each cell. Note that if multiple options result in the same score, all are recorded. This can be done by storing 'predecessor' pointers for each cell, or

	0	1	2	3	4
F	–	C	A	T	T
0 –		$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$
1 G	$(0,0)$	$(0,0)$	$(0,1)$ $(1,1)$	$(0,2)$ $(1,2)$	$(0,3)$ $(1,3)$
2 A	$(1,0)$	$(1,0)$ $(1,1)$	$(1,1)$	$(2,2)$	$(2,3)$
3 A	$(2,0)$	$(2,0)$ $(2,1)$	$(2,1)$ $(2,2)$	$(2,2)$	$(2,3)$ $(3,3)$
4 T	$(3,0)$	$(3,0)$ $(3,1)$	$(3,2)$	$(3,2)$	$(3,3)$
5 C	$(4,0)$	$(4,0)$	$(4,2)$	$(4,3)$	$(4,3)$ $(4,4)$
6 T	$(5,0)$	$(5,1)$	$(5,1)$ $(5,2)$	$(5,2)$ $(5,3)$	$(5,3)$

Figure 2.4: Predecessor pointers for Needleman-Wunsch traceback.

by saving the results of each of the three equations in separate matrices and looking up which had the highest result for a cell [43]. Figure 2.4 shows the predecessor pointers for the example matrix. Then, from the bottom-right cell, a route is traced back to the top left cell; this gives an alignment. If multiple routes are possible, there are multiple optimal alignments. Looking at Figure 2.4, starting at element $(6,4)$ we trace back to $(5,3)$, from which it was derived; this is a diagonal movement (the top option in Formula 2.1 was used) which means the letters are aligned (TT). On the other hand, $(5,3)$ was derived from $(4,3)$, a vertical movement (second option in Formula 2.1), which means a gap (C−). $(4,3)$ was derived using a diagonal movement (TT). $(3,2)$ can be aligned as AA by tracing to $(2,1)$ or A− by means of $(2,2)$. This is because its score of -2 could have been attained from -3+1 or 0-2. Tracing back all the way, and taking into account that multiple routes exist where two options from the formula resulted in an equal score, the three optimal alignments, with score -2, are:

G A A T C T G A A T C T G A A T C T
C – A T – T C A – T – T – C A T – T

2.1.1.2 Affine gap penalties

Note that the preceding explanation used a linear gap penalty. To extend the algorithm with support for affine gap penalties, an additional two $(n \times m)$ matrices are required, say I_x and I_y.

Initialization

The $(0,0)$th matrix element of F is set to 0. The top row and and leftmost column are initialized with the costs of gaps of lengths i and j, keeping in mind the gap opening and extension penalties (elements $(0,1)$ and $(1,0)$ are set using the opening penalty, the rest extend from there). The I matrices are left blank. Figure 2.5 shows the initialized F matrix for gap opening penalty 3 and extension penalty 2.

Matrix fill

For each element of I_x, the (i,j)th element is the score of an optimal alignment

	0	1	2	3	4
F	–	C	A	T	T
0 –	0	-3	-5	-7	-9
1 G	-3				
2 A	-5				
3 A	-7				
4 T	-9				
5 C	-11				
6 T	-13				

Figure 2.5: Needleman-Wunsch initialization phase for affine gap penalties.

between $x_{1..i}$ and $y_{1..j}$ given that the alignment ends with x_i aligned to a gap. I_y is analogous, but for y_i. In effect, these new matrices store either the score of starting a new gap or extending an existing one. Additionally, the original matrix F now does not have the option of aligning with gaps anymore; instead, the options are to continue an existing alignment or to start a new one (closing a gap). In formula form:

$$F(i,j) = max \begin{cases} F(i-1,j-1) + s(x_i, x_j) \\ I_x(i-1,j-1) + s(x_i, x_j) \\ I_y(i-1,j-1) + s(x_i, x_j) \end{cases} \tag{2.2}$$

$$I_x(i,j) = max \begin{cases} F(i-1,j) - d \\ I_x(i-1,j) - e \end{cases} \tag{2.3}$$

$$I_y(i,j) = max \begin{cases} F(i,j-1) - d \\ I_y(i,j-1) - e \end{cases} \tag{2.4}$$

With d the gap opening penalty and e the gap extension penalty.

Traceback

Traceback now starts at one of the three matrices, depending on which has the maximum value for the final (bottom right) entry. Otherwise it proceeds similar to that for linear gap penalties, but with the possibility of jumping from matrix to matrix.

2.1.2 The Smith-Waterman algorithm

The Smith-Waterman algorithm [77] is similar to Needleman-Wunsch, but produces local alignments, which might be more biologically interesting.

Initialization

The top row and leftmost column are initialized to 0.

Matrix fill

A new option is introduced: that of starting a new alignment, which might result in a better score than continuing an existing one. This translates into resetting the current score to 0. In equation form:

$$F(i,j) = max \begin{cases} 0 \\ F(i-1,j-1) + s(x_i, x_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases} \tag{2.5}$$

Traceback

As the alignment can end before the sequences are over, instead of starting at the bottom right matrix element, traceback starts at the maximum value in the matrix and then traces back until a 0 is encountered.

Smith-Waterman can be extended for affine gap penalties in a fashion similar to Needleman-Wunsch [81]; in this case the F matrix top row and leftmost column are initialized to zeros:

$$F(i,j) = max \begin{cases} 0 \\ F(i-1,j-1) + s(x_i, x_j) \\ I_x(i,j) \\ I_y(i,j) \end{cases} \tag{2.6}$$

$$I_x(i,j) = max \begin{cases} F(i-1,j) - d \\ I_x(i-1,j) - e \end{cases} \tag{2.7}$$

$$I_y(i,j) = max \begin{cases} F(i,j-1) - d \\ I_y(i,j-1) - e \end{cases} \tag{2.8}$$

2.2 Heuristics: FASTA, BLAST and ClustalW

As stated, heuristic approaches are imperfect and might overlook optimal alignments. However, they are much faster than their dynamic programming counterparts and are widely used for database searches, see Section 1.3. FASTA and BLAST both perform pairwise alignment; ClustalW is a multiple alignment tool. There is no clear guideline on when to use FASTA and when to use BLAST; supposedly, FASTA is more sensitive but slower than BLAST [1]. The bioinformatics databases can be searched with either. In any case, as high scoring alignments can be found by chance, statistical analysis of the significance of the resulting scores is advised [52].

2.2.1 FASTA

The FASTA heuristic is based on the Smith-Waterman algorithm. Looking at the similar Needleman-Wunsch traceback example, it is apparent that only part of the matrix is used for the traceback; the almost diagonal 'route' back to $(0, 0)$. In larger matrices, this difference between the interesting area used during traceback and the size of the complete matrix only grows. FASTA is meant to exploit this difference by concentrating on these areas of interest: the areas are determined and Smith-Waterman is run only on these, which is where the speed boost comes from.

These areas of interest are constructed using a 'dot matrix' method. The algorithm operates as shown in Figure 2.6 [32] [73]:

Find runs of matches
Shown in Figure 2.6a. A matrix of both sequences is constructed, and runs of matches between the sequences are marked (for protein sequences whether two letters 'match' is determined using a scoring matrix). Next only matching sequences of the given length k are kept. This *k-tuple* length can be set by the user, see KTUP in Figure 1.8. A larger k-tuple value will speed up the search, but make it less sensitive. For protein sequences, a value of 2 is frequently used; for nucleotide sequences a value between 4 and 6 is recommended. On-line search engines might enforce certain values; for example EBI requires a minimum value of 6 for nucleotide sequences [10]. The top ten matches advance to the next step.

Trim match regions
Shown in Figure 2.6b. The regions are trimmed using a second pass with the scoring matrix or scores for nucleotide insertions/deletions. Regions are allowed to become smaller than the k-tuple size.

Join regions
Shown in Figure 2.6c. Regions from the previous step are joined if they do not overlap and the gaps between them are smaller than a set cutoff value. Only the new highest-scoring region advances to the final step.

Apply DP to area
Shown in Figure 2.6d. Smith-Waterman is run on an area surrounding the region from the previous step; the width of this region determines how exhaustive (but slow) this step is. A width of 32 is used for the EBI search engine [10]. Cells outside this area are assigned a value of $-\infty$, making them completely unattractive choices for the Smith-Waterman matrix fill step to build on.

2.2.2 BLAST

BLAST (Basic Local Alignment Search Tool) is another heuristic sequence alignment algorithm, although unlike FASTA it is not based on a dynamic programming method. The steps are as follows [32]:

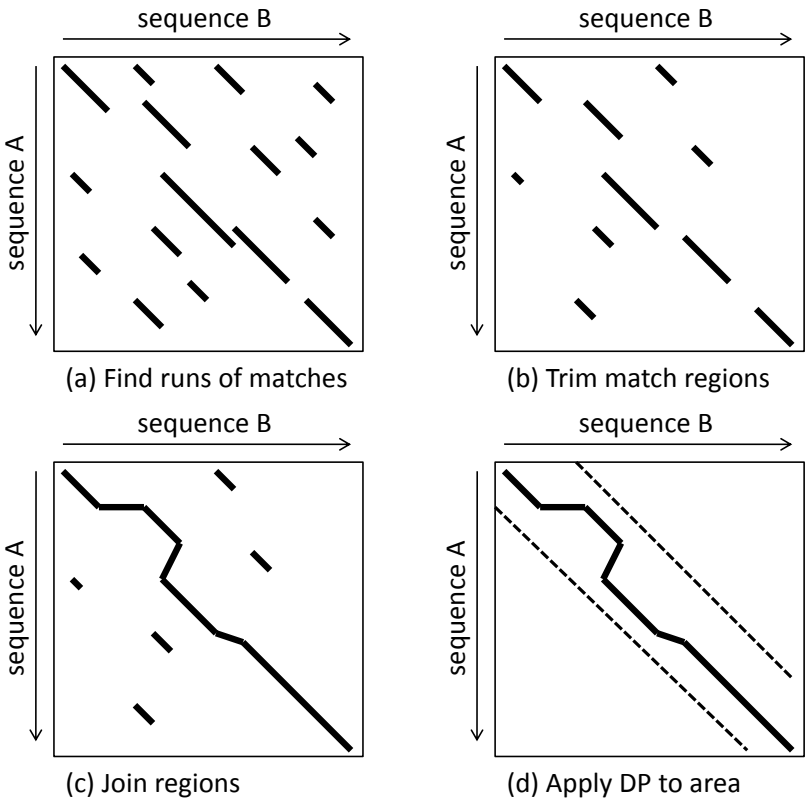


Figure 2.6: The FASTA algorithm.

Score query words

The query sequence is split up into a set of k-letter words, with k usually 3 for proteins and 11 for DNA/RNA. For example, the amino acid sequence ARNDCQ gives the three-letter words {ARN, RND, NDC, DCQ}. All possible k-letter words in the alphabet (20^3 for proteins and 4^{11} for DNA/RNA) are then scored with these query words. The scoring is again done using scoring matrices for amino acids and fixed scores for (mis)matches of nucleotides. All words that score higher than a certain threshold continue on to the next step. Setting k higher will increase the search sensitivity but decrease speed; setting the scoring threshold higher will do the opposite as more words will make it to the next step [30].

Search database

The database is searched for the selected words, requiring an exact match. To accommodate this, the database has been preprocessed and converted to BLAST format [23].

Extend matches

The matches are extended (to some maximum extent dictated by a parameter)

in both directions in an attempt to decrease gaps and increase the score of the alignment to be higher than some threshold. These extended query words are scored as before. The score threshold and extension degree are again parameters that are either part of the algorithm or can be set by the user.

BLAST has been extended with filters to automatically remove biologically uninteresting parts from query sequences. Furthermore, variations on BLAST exist such as PSI-BLAST [21] which is geared towards finding distant evolutionary relationships and BLAT [55] which is faster than BLAST (as it keeps the entire target genome in RAM) but less sensitive.

2.2.3 ClustalW

ClustalW [4] is a heuristic multiple alignment tool that uses a progressive alignment method; though the dynamic programming algorithms can be used to determine multiple alignments, this requires expanding them to operate on n-dimensional matrices (to align n sequences), which quickly becomes prohibitively expensive. ClustalW is based on the Clustal program; its second version, ClustalW2 is currently in use. The Clustal algorithm works as follows [79], see Figure 2.7:

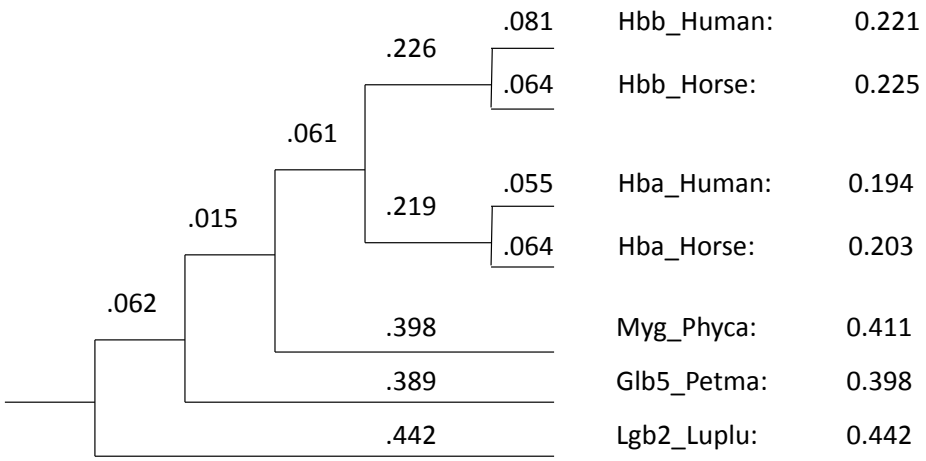
Pairwise alignment
Shown in Figure 2.7a. All pairs of target sequences are aligned using an approximate method that scores the alignments in a seemingly BLAST like fashion which is detailed in [80]. ClustalW introduced the ability of using the more accurate, but slower, dynamic programming algorithms. The scores of each pairwise alignment are stored in a triangular matrix as distances from 0 to 1.

Similarity tree
Shown in Figure 2.7b. A similarity tree is constructed in two steps: first an unrooted tree is constructed from the distance matrix using the Neighbor-Joining method of creating phylogenetic trees. Then the tree is transformed to a rooted version; for the Clustal algorithm all sequences get the same total weight, for the newer ClustalW version a sequence's weight depends on its distance from the root and what branches it has in common with other sequences. For example the 'Hbb_Human' sequence in the example figure is scored as its own unique branch plus half of the branch it shares with 'Hbb_Horse' plus a quarter of the branch to the 'Hbb'/'Hba' groups plus one fifth of the branch those share with 'Myg_Phyca' plus one-sixth of the branch also encompassing 'Glb5_Petma'. This gives it the shown score of $(0.081 + 0.113 + 0.01525 + 0.003 + 0.0103) = 0.221$.

Progressive alignment
Shown in Figure 2.7c. The progressive alignment is performed from the ends of the tree branches back to the root. Looking at the example figure, the order of alignments would be (1) 'Hbb_Human' vs. 'Hbb_Horse'; (2) 'Hba_Human' vs. 'Hba_Horse'; (3) the result of (1) vs. the result of (2); (4) the result of (3) vs 'Myg_Phyca'; (5) the result of (4) vs 'Glb5_Petma' and finally the result of (5) vs. 'Lgb2_Luplu'. Once all sequences have been compared a final alignment is found;

Hbb_Human	1	-				
Hbb_Horse	2	.17	-			
Hba_Human	3	.59	.60	-		
Hba_Horse	4	.59	.59	.13	-	
Myg_Phyca	5	.77	.77	.75	.75	-
Glb5_Petma	6	.81	.82	.73	.74	.80
Lgb2_Luplu	7	.87	.86	.86	.88	.93
		1	2	3	4	5

(a) Pairwise alignment



(b) Similarity tree

```
Hbb_Human       atgcatgcatcgatgc
Hbb_Horse       aggcgtgcaccgatgc
```

(c) Progressive alignment

Figure 2.7: The Clustal algorithm.

Source: Adapted from [79].

of course, gleaning actual biologically interesting information will require careful interpretation.

These progressive alignments are performed using dynamic programming algorithms, with some side notes. When a gap is introduced, it cannot be removed at a later stage. Furthermore, if a gap is introduced within an existing gap, the full gap creation penalty is deducted. ClustalW expands on this by varying the

scoring matrices used depending on the distances between the sequences being compared and, in turn, varying the gap creation and expansion penalties depending on the current scoring matrix, sequence similarity, sequence lengths and the current position of the alignment within the sequences.

2.3 Heuristics: Hidden Markov Models and HMMER

Hidden Markov Models are an approach rooted in probability theory that can be applied to sequence alignment problems, for example multiple alignment [37] and database searches [42]. The information on hidden Markov models in this chapter comes from [42] and Chapter 3 of [52].

2.3.1 Hidden Markov Models

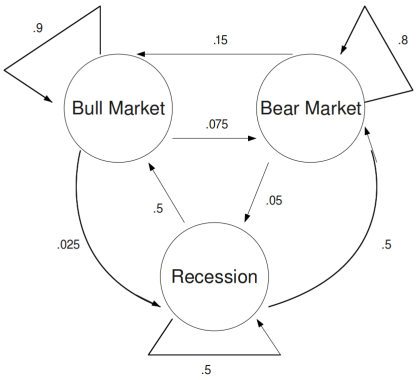


Figure 2.8: A Markov chain.

Hidden Markov models come from Markov chain theory. A Markov chain is a state machine: for each state, there are probabilities (adding up to a total of 1) of entering another (or the same) state. Furthermore, they adhere to the *Markov property* that the next state only depends on the current state, not the past state(s) that have been visited. To determine where the chain starts, *initialization* probabilities, for each state the probability of starting there, are used. The chain is said to *generate* a sequence of states: an example sequence generated by the chain in Figure 2.8 would be {'Bull Market' 'Bull Market' 'Recession'}.

Markov chains can be used to model phenomena. For instance, if one has a Markov chain that captures the probabilities of each of the four nucleotides succeeding one another, it can be evaluated whether some nucleotide string has a significant probability of being generated by that model. Then, if that probability is higher than a certain threshold, one can decide that the string adheres to the model and is, as such, a valid nucleotide sequence. If we expand our model to take into account substitutions, deletions and insertions, the same sequence can be generated in multiple ways. For example the sequence 'CATG' could arise from 'T' being inserted into 'CAG', or from 'T' being deleted from 'CATTG'. In cases where only the generated strings are observed, and not the probabilities behind them (the way the string was generated), the model is said to be a *hidden Markov model* (HMM); a HMM that models, or *profiles* a set of sequences is called a *profile hidden Markov model*.

Evaluating the probability of a sequence being generated by a certain HMM can be done using the dynamic programming *Forward algorithm*; finding the most likely path

through the HMM for a given sequence can be accomplished using the similar *Viterbi algorithm*.

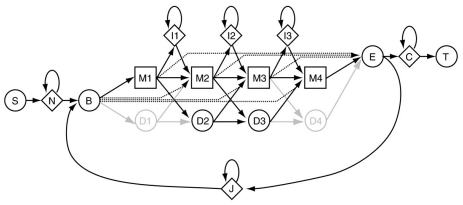


Figure 2.9: The Plan7 nucleotide/protein HMM connectivity [69].

Note the insertion, deletion and match states marked I, D and M respectively.

As with any Markov chain, a profile HMM consists of a layout, or *connectivity* and a set of transition probabilities. The connectivity is designed by hand to match the phenomena at the core of the model being investigated; Figure 2.9 shows such a connectivity. The transition probabilities can be determined in two ways. In the case of a biological sequence, the simplest approach is to infer the transition probabilities from an existing multiple alignment and *build* the HMM using these probabilities. A more useful approach, however, might be to *train* the HMM using a collection of unaligned sequences, the *training set*. Using algorithms such as *Baum-Welch training* or *Viterbi training* the HMM's transition probabilities can be estimated.

Using all this knowledge, we can use a profile HMM to perform a multiple sequence alignment:

Connectivity selection
: A fitting connectivity is chosen, with a length that matches the data being worked with; in case of a multiple alignment, it can be set to the average length of the sequences in question.

Training
: Using one of the training algorithms, the Markov chain's probabilities are estimated from the target sequences.

Path determination
: The Viterbi algorithm is used to determine the most likely path through the model for each sequence.

Path comparison
: The resulting paths are compared: if two paths share the same match states for a letter, the sequences' letters are introduced, otherwise a gap is added. Different alignments can arise from the fact that sequences of unequal lengths can be lined up in various ways during this step.

2.3.2 HMMER

HMMER can be used to align sequences to a profile HMM or to extract a profile HMM from a multiple alignment; it can also be used as a database search tool like FASTA and BLAST. Sequences can be searched against sequence and profile databases (such as Pfam [45]); profiles, in turn, can be searched against sequence databases. As given in [14], HMMER is described as follows:

"HMMER is used for searching sequence databases for homologs of protein sequences, and for making protein sequence alignments. It implements methods using probabilistic models called profile hidden Markov models (profile HMMs). Compared to BLAST, FASTA, and other sequence alignment and database search tools based on older scoring methodology, HMMER aims to be significantly more accurate and more able to detect remote homologs because of the strength of its underlying mathematical models. In the past, this strength came at significant computational expense, but in the new HMMER3 project, HMMER is now essentially as fast as BLAST."

However, HMMER cannot search for nucleotide sequences [24] and it is currently not offered in web interface format by any of the bioinformatics databases discussed in Section 1.3.

HMMER profiles use the 'Plan 7' connectivity shown in Figure 2.9, built from multiple alignments. Although HMMER cannot be trained using unaligned sequences (its author recommends using Clustal to align these first), it can create a profile from a single sequence for its `phmmer` sequence vs. sequence database search. It does this by converting protein score matrix (BLOSUM62) entries to probabilities [24].

2.4 Hardware acceleration of sequence alignment algorithms

As discussed at the start of this chapter, dynamic programming approaches to sequence alignment can be too computationally expensive to be applied to database searches. In fact, even the speed of heuristic algorithms might sometimes be insufficient. It would be desirable to speed up both types of algorithms, ideally getting enough speed out of dynamic programming to be able to search databases in a quick manner that still produces optimal alignments. This is where hardware acceleration comes in: a dedicated, (semi)specifically designed piece of hardware will (generally) be faster at a given task than a general purpose CPU such as that in a PC. Of course, trade-offs arise, such as cost, ease of development, flexibility and configurability.

2.4.1 Acceleration options

Although efforts have been made to implement sequence alignment algorithms on dedicated chips or *application specific integrated circuits* (ASICs) [36] [40] [47], especially in research more flexible platforms such as *field-programmable gate arrays* (FPGAs) and *graphics processing units* (GPUs) are preferred. GPU acceleration is dealt with in Chapter 3 and is the main focus of this thesis. However, the FPGA approaches discussed in this chapter show earlier work that compares to, and contrasts with, GPU implementations.

FGPAs are integrated circuits that can be configured via software using description languages such as VHDL. This makes it easy to quickly try and modify approaches to hardware acceleration; this ease of use means development costs are much cheaper. Trade-offs apply: FPGAs are generally not as fast as ASICs, consume more power, and their per-unit costs are higher [11] [33]. Still, FPGAs have been used in commercial sequence alignment products, for example those offered by [3] and [26].

2.4.2 The systolic array

One approach to FPGA acceleration is the systolic array. This focuses on the matrix fill step of the Smith-Waterman algorithm (Section 2.1.2) as used for database searches. As Smith-Waterman produces local alignments, it is a more popular choice than Needleman-Wunsch, although both can be hardware accelerated in similar ways. Furthermore, the matrix fill step is the most time consuming phase of the algorithm, and needs to be run for each (database) sequence to be compared. As such, it makes sense to implement this step in hardware and return the maximum matrix value: the score of the alignment. The top-scoring sequence pairs, which are a tiny group compared to the complete database, can then have traceback performed in a separate step, probably on the personal computer that controls the FPGA board. Not performing traceback on the FPGA frees up hardware resources to speed up matrix filling.

Looking at the Smith-Waterman algorithm in Section 2.1.2, each matrix element is dependent on just three others: $F(i-1, j-1)$, $F(i-1, j)$ and $F(i, j-1)$. This means that elements on the matrix diagonals are calculated sequentially. Figure 2.10 shows this;

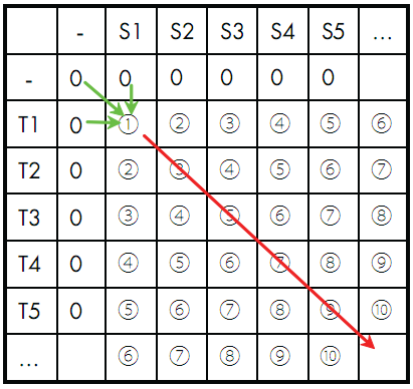


Figure 2.10: Parallelism of the Smith-Waterman algorithm [81].

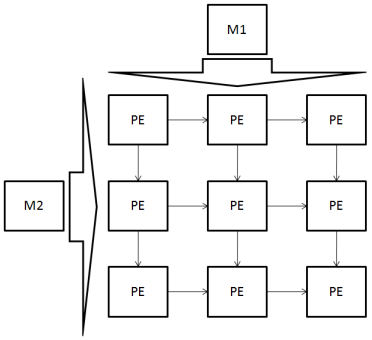


Figure 2.11: Systolic array for matrix multiplication.

first only the element marked '1' can be calculated, after that both elements marked '2' can be calculated, etc. Note how the cells sharing the same number form anti-diagonals: all cells on these anti-diagonals can be filled in parallel.

This is where the *systolic array* comes in. Systolic array approaches are used in many hardware implementations [36] [46] [72] [74] [81]. A systolic array is a matrix of simple *processing elements* (PEs). The operands of the calculation to be performed are streamed into the array, the PEs perform the required calculation on their two values and pass the result to their neighbors. For example, a matrix multiplication could be performed by streaming two matrices row-by-row and column-by-column into the array; Figure 2.11 illustrates this. Returning to Smith-Waterman, using such a two-dimensional array would result in PEs being idle a lot of the time [36] [48], wasting hardware resources; as such an one-dimensional *linear systolic array* is used.

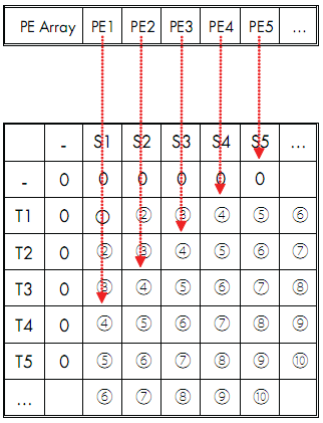


Figure 2.12: Linear systolic array performing Smith-Waterman matrix fill [81].

Figure 2.12 shows a linear array being used to fill the aforementioned anti-diagonal lines. The PEs are each assigned one of the query sequence letters, and the database sequences are streamed through horizontally; the PEs pass intermediate values to one another as the sequence is streamed through. The figure illustrates the processing order: each PE effectively handles a matrix column. Each PE performs alignment of its query sequence letter and the current letter of the database sequence at its position. The PEs keep a current maximum cell value and compare this to that of their predecessor: the largest of the two is saved and passed on. This way, once the entire query sequence has

been streamed through, the last PE will hold the alignment score.

If there are enough hardware resources available, multiple arrays can be synthesized to handle database sequences in parallel. Unfortunately, more likely the opposite will be true: not enough hardware resources to create an array of the same length as the target sequence [72] [81]. In this case, the computation will need to be split up into multiple passes. Part of the query sequence is assigned to the array and the database is run through it. Each output value of the last PE is saved. Then when the next part of the query sequence is assigned, these values can be used as inputs. In effect, if there are N PEs, the Nth column of the matrix is saved and then used in the next pass.

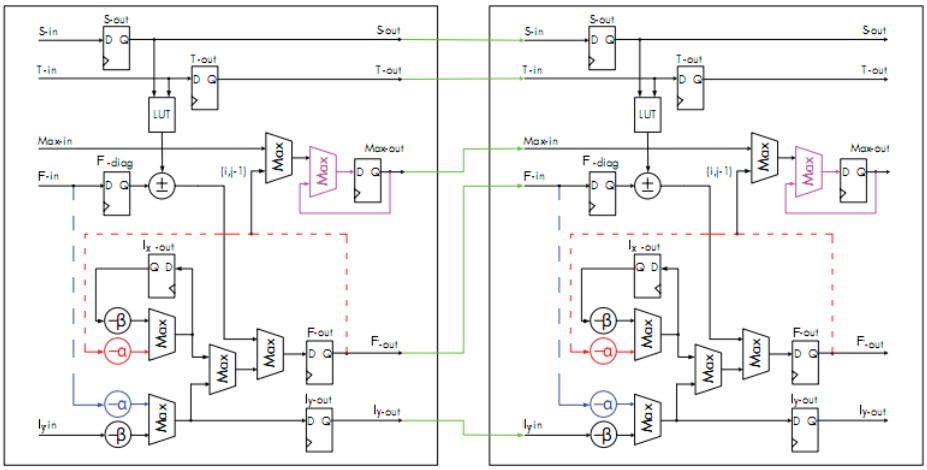


Figure 2.13: Two connected Smith-Waterman processing elements.

Source: Adapted from [81].

A straightforward design of the actual PEs is shown in Figure 2.13, which shows two of them connected as they would be in the array. Note the *Max-out* output, which handles the aforementioned maximum value; it is updated as needed from the predecessor's output or from the result of the PE's calculations. This particular PE operates with affine gap penalties (Formulas 2.6, 2.7 and 2.8). Note how F and I_y are passed on between the elements; F is delayed a clock cycle in *F-diag* so it will in effect be the value of the PE's upper left matrix element instead of its left neighbor. The I_x-values are reused in one PE as the next element to be processed by this PE will be the one below the current one, see Figure 2.12.

GPU accelerated sequence alignment

3

A somewhat recent development in the field of bioinformatics is the application of *graphics processing units* (GPUs) to computational problems such as sequence alignment. Originally intended for video gaming, GPUs have since become powerful and flexible vector processors. They are vendor-supported as a viable platform for general-purpose computing and simulations such as fluid dynamics, electrical fields and rigid-body physics [6].

3.1 Overview of GPU evolution and programmability

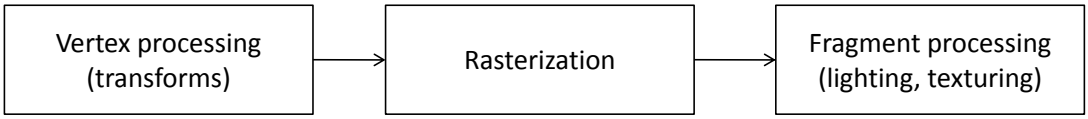


Figure 3.1: The graphics pipeline.

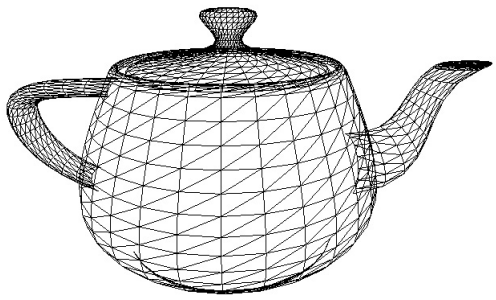


Figure 3.2: A teapot mesh, made up of triangular polygons.

This section takes a look at the changes GPUs have undergone in the last decade and a half, and how these changes resulted in applicability to general purpose computation.

3.1.1 The fixed-function GPU

Depending on one's exact definition of the word 'GPU', it is arguable whether the early, non-programmable graphics accelerators can be qualified as such. However, they are clearly related to the cards in use today. In this context a graphics accelerator is a piece of hardware that aids in the display of three-dimensional graphics, offloading the CPU from tasks such as matrix transformations and texture mapping.

Although present in the high-end workstations of the early 90s, graphics accelerators only gained mainstream appeal with their application to PC gaming in the mid-90s. For the first time offered at an affordable price point, cards with chips such as the 3DFX Voodoo (1996) greatly increased the graphics quality and frame rates for video games and set the tone for the devices that would follow.

Three-dimensional computer graphics are drawn using the steps shown in Figure 3.1. 3D shapes are specified using points called *vertices* that form triangles. These triangles are called *polygons*, see Figure 3.2. In the first pipeline step, vertex processing, the vertices are transformed to their final on-screen positions. This involves multiple position transformations, which are applied using matrix transforms. The vertices of, for instance, the player character are first transformed to their correct position in the world. Then, a perspective transformation is applied to create the illusion of depth: distant objects become smaller and parallel lines converge to a vanishing point. The second step, rasterization, is the conversion of the point data to pixels or *fragments*. In this step care must be taken when dealing with overlapping objects, especially when they are translucent. Algorithms such as *Z-buffering* aid in this. Finally, the pixels are assigned a color. This color depends on the object's material properties and the positions of lights in the scene. Additionally, objects can be textured with an image, such as a photo of bricks in case of a wall.

These earliest graphics accelerators implemented only the rasterization and texture mapping steps in hardware. Vertices were provided fully transformed to their on-screen positions and for each a color was supplied, the result of lighting calculations performed on the CPU. The colors of the vertices were then interpolated across their polygons' resultant pixels.

The next big improvement in accelerators was the introduction of *Hardware Transform & Lighting* with the NVIDIA Geforce 256 in 1999. As the name suggests, this meant that the rest of the pipeline was implemented in hardware too; programs now specified transformation matrices and light positions to the graphics card, which took care of the rest.

The graphics pipeline as discussed here is known as the *fixed-function pipeline*. Although various pixel blending modes and support for techniques such as environment and bump mapping meant that developers had a collection of effects at their disposal, the fundamental operation of the pipeline was set in stone.

3.1.2 Shaders and programmability

In 2001 the NVIDIA Geforce 3 graphics card was introduced, and this can be seen as the first true GPU. The vertex and fragment processing stages were now flexible and could be customized using small programs called *shaders*. For example, a vertex shader could be used to apply a sine wave transform to a plane mesh, simulating rippling water. Pixel shaders could be used to calculate lighting per-pixel instead of it being interpolated from per-vertex values, and allowed for effects such as heat haze. Throughout the years various new cards improved performance and flexibility by offering more texture reads per shader, various new instructions and greatly improved support for flow control.

Finally, in 2006 the distinction between vertex and pixel shaders was dropped on the

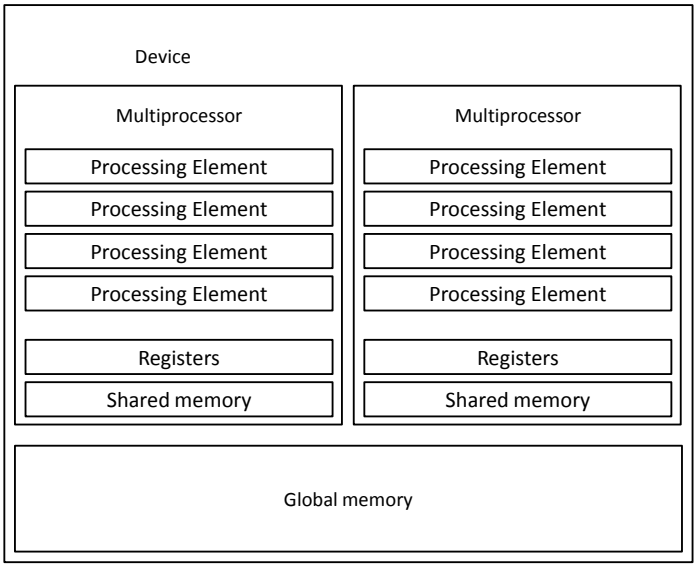


Figure 3.3: Layout of a modern GPU.

hardware level. *Unified shader architecture* GPUs have just one type of processor that can run instructions relevant to any type of graphics operation. This change simplified the hardware and introduced the flexibility required for general purpose computing. Modern GPUs are highly parallel floating point *stream processors* [71]. Graphics work is inherently parallel: vertices and pixels can be processed completely independently. To that end, GPUs consist of a large amount of *processing elements* that are little more than parallel ALUs, optimized for floating point vector work. These processing elements are grouped into *multiprocessors* that offer facilities such as shared memory. Figure 3.3 shows the architecture of a modern GPU. The amounts of multiprocessors per GPU and processing elements per multiprocessor vary; a currently reasonably high-end Geforce GTX 275, see Figure 3.4, offers 30 multiprocessors with 8 processing elements each, for a total of 240 processing elements.

An array of data, such as pixels (each of which is a 4-component floating point value), is streamed through the processors. Each multiprocessor's elements perform the same operation on their pixels: the multiprocessors are *Single Instruction Multiple Thread* (SIMT) processors. This streaming paradigm is reflected in architectural tradeoffs: memory latency is rather high, but is compensated for by the streaming's predictable access patterns. Flow control is allowed, but as all of a multiprocessor's elements must remain in sync, it is accomplished by temporarily halting multiprocessors that do not agree on the path taken.

Figure 3.4: A NVIDIA Geforce GTX 275 graphics card.

3.1.3 General purpose computing and CUDA

Although general purpose computing on GPUs has been experimented with since the introduction of shaders, it had always been rather convoluted: users were stuck with graphics APIs such as OpenGL and Direct3D. Problems had to be expressed in terms of graphics primitives such as vertices and textures, then rendered to another texture and this result had to be read back. Fortunately, things became easier with the release of NVIDIA's *Compute Unified Device Architecture* (CUDA). CUDA offers a more general parallel programming model than before and allows the GPU to be programmed in a C-like fashion. Nowadays alternatives to CUDA exist, that run on the same hardware. Examples are OpenCL [18], DirectCompute [8] and the gaming-aimed Direct3D Compute Shaders [7]. These are largely similar, but as CUDA is the most widely used and mature, this section will focus on that.

3.1.3.1 The CUDA thread hierarchy

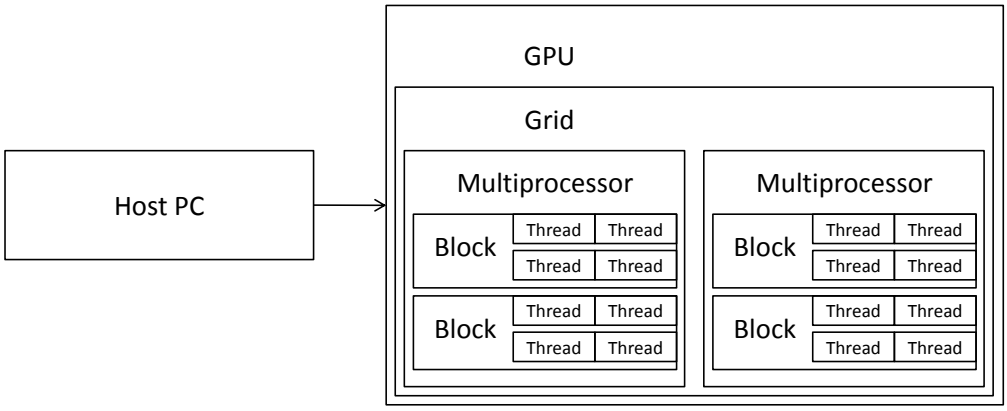


Figure 3.5: The CUDA thread hierarchy.

CUDA accelerated programs are laid out as shown in Figure 3.5. The PC system, the *host*, invokes a *kernel* by calling it from C/C++ code. This kernel is the code that each processing element will run in parallel. A kernel is launched as a grid of threads; the threads are bundled as blocks. This block abstraction allows for hardware scaling: if GPU x has twice the multiprocessors of GPU y, it can run twice as many blocks in parallel with no further code changes. Furthermore, on multi-GPU systems a grid can be launched on each GPU.

Each block consists of a number of threads; how many threads make up a block can be chosen by the programmer. The threads in a block are executed by the same multiprocessor and can communicate using its shared memory. A multiprocessor can run

multiple blocks at the same time by interleaving them; it can also interleave bundles of threads from one block. On the hardware level, a block's threads are executed as *warps*. A warp is executed in parallel by a multiprocessor's processing elements. For example, for NVIDIA G80 and GT200 GPUs, a warp consists of 32 threads; each multiprocessor has eight processing elements that run four threads in a quad-pumped fashion. The execution of threads as warps and half-warps is important as it ties into optimization factors, such as bundling memory accesses and preventing control flow divergence.

How many threads to run in total, and how to split these up into blocks, depends on various factors. First of all, it makes sense to run a thread for each parallel subproblem. It is also advantageous to run as many threads as possible at any time, to prevent processing elements from idling, or delays while waiting for memory accesses. However there is a hard limit on the number of threads per block, and all threads in a block have to share a multiprocessor's resources such as registers. The number of threads should be a multiple of the warp size to prevent warps from leaving processing elements idle [70]. Yet there is also a limit on the total number of threads active on a multiprocessor at any one time, which might make it more attractive to have multiple smaller blocks using this full capacity than one large block that does not. The documentation sets some guidelines, but recommends experimenting with the parameters as they depend on the specific GPU and kernel in question.

3.1.3.2 The CUDA memory hierarchy

Besides the threading model, another thing that sets CUDA programming apart from working with a general purpose CPU is its memory layout. The physical memories are shown in Figure 3.3, but some of these offer various memory spaces. The complete set of CUDA memory spaces is:

Registers
: Each multiprocessor offers a bank of registers, shared between its processing elements. Depending on the capabilities of the device, there are between 8k and 32k registers per multiprocessor.

Global memory
: Global memory is the GPU's off-chip RAM. On modern GPUs the amount of global memory tends to be around one gigabyte. Accessing it has a high latency, which can be hidden by switching between threads. Furthermore, memory accesses can be *coalesced*, which means memory accesses of multiple threads in a warp can be bundled. This takes place if certain criteria are met; most importantly, each thread in a half-warp (16 threads) must access a neighboring address. Access sizes can be 32, 64 and 128 bytes. For example, if the threads in a half-warp read neighboring 4-byte values, one 64-byte access will be issued. If accesses are not coalesced, they must take place sequentially, resulting in slower performance and wasted bandwidth; the minimum access size will still be 32 bytes. If the 16 threads in the half-warp each read aforementioned 4-byte values at unrelated addresses, 16 32-byte accesses will take place, wasting $16 \times 32 - 64 = 448$ bytes of bandwidth.

Local memory
Local memory is a per-thread portion of global memory; it is used to spill registers and for large variables. Has the same speed as global memory. 16-48 kilobytes per thread.

Shared memory
Shared memory is on-chip fast memory shared between all threads in a block. Depending on the capabilities of the GPU, this can be 16 or 512 kilobytes per multiprocessor. Shared memory is divided into banks, and performance is improved if all threads in a half-warp access different banks. However, if multiple threads within a half-warp access the same 32-bit address, these accesses are bundled into a single *broadcast* to the relevant threads.

Constant memory
The constant memory is a 64 kilobyte portion of read-only global memory that is cached on each multiprocessor. Accessing data is as fast as if it were a register, as long as all threads in a half-warp read from the same address.

Texture cache
Textures are cached 'windows' into global memory, optimized for spatially local reads. Textures can be one, two or three-dimensional and they offer various functions of the texture mapping hardware such as interpolation and clamping modes. Each multiprocessor can cache up to 8 kilobytes of texture data.

As can be seen, there are six memory types to deal with; compared to general purpose CPU work where there are usually only registers and RAM, CUDA programming requires care to optimize the kernel's memory access patterns to best exploit the hardware.

3.2 GPU accelerated Smith-Waterman

This section focuses on using GPUs to accelerate optimal sequence alignment against a database using the Smith-Waterman algorithm (Section 2.1.2). As with FPGA implementations (Section 2.4), usually only the time consuming matrix fill step is implemented on the GPU. The sequences with the highest scores are subjected to a second pass on the CPU, where they are aligned again and have traceback performed.

3.2.1 The OpenGL approach

The first known implementations of Smith-Waterman on a GPU are described in [59] and [61]. These approaches are similar and use OpenGL (CUDA did not exist yet) to search protein databases. They closely match the FPGA based systolic array approach. First the database and query sequences are copied to GPU memory as textures. The score matrix is then processed in the familiar anti-diagonal fashion. For each element of the current anti-diagonal, a pixel is drawn. Drawing this pixel executes a pixel shader that calculates the score for the cell. The results are written to a texture, which is then used as input for the next pass, similar to how the systolic array elements pass on values.

The implementation of [59] searched 99,8% of Swiss-Prot (almost 180,000 sequences at the time) and managed to obtain a maximum speed of 650 MCUPS compared to around 75 for the compared CPU version. The performance in *cell-updates per second* (CUPS) is the total number of Smith-Waterman score matrix cells that are updated per second; the formula is shown in Equation 3.1.

$$CUPS = query\_length * total\_database\_size/run\_time \tag{3.1}$$

Another implementation discussed in [61] offers the ability to run in two modes, one with and one without traceback. The version with no traceback managed to perform at 241 MCUPS, compared to 178 with traceback and 120 for the compared CPU implementation. Note that a database of just 983 sequences was used. Both implementations were benchmarked using a Geforce 7800GTX graphics card.

3.2.2 The CUDA approach

The first known CUDA implementation, 'SW-CUDA', is discussed in [65]. The approach is different from the systolic array method: each of the GPU's processing elements performs a complete alignment instead of them being used to stream through a single matrix. The advantage of this is that no communication between the processing elements is required, which cuts down on memory reads and writes. The database is stored in global GPU memory, sorted by length so the threads in a warp will have similar execution times. To speed up substitution matrix accesses, a *query profile* is generated: an expanded substitution matrix that has the query sequence elements as its columns, see Figure 3.6. Its rows are still the protein alphabet. Query profile accesses are less random than accessing a substitution matrix: when processing a database sequence letter, a bundle of alignment scores with the query sequence can be prefetched. Each kernel iteration processes four cells, making use of the memory's ability to fetch and store vectors. This implementation managed to perform at 1.9 GCUPS on a single Geforce 8800GTX GPU when searching Swiss-Prot, compared to around 0.12 GCUPS for the compared software implementation. Furthermore, it was shown to scale almost linearly with the amount of GPUs used by simply splitting up the database.

	A	L	R	K
A	4	-1	-1	-1
R	-1	-2	5	2
N	-2	-3	0	0
D	-2	-3	-2	-1
...	...	...	...	...
V	0	1	-3	-2

Figure 3.6: Query profile for sequence ALRK using the BLOSUM62 matrix (Figure 1.6).

3.3 Other GPU-based sequence alignment approaches

Although this section has so far focused on database searches using Smith-Waterman, there exist additional GPU based sequence alignment tools which merit a mention.

Pairwise sequence alignment

MUMmerGPU [76] is a CUDA accelerated version of the MUMmer tree-based approach to pairwise alignment. It is targeted towards aligning a set of small DNA query sequences with a large reference sequence. The data used is stored in global memory and split up into multiple horizontal and vertical parts mapped as textures, for improved spatial locality and, as such, cache performance. It is able to accomplish a more than three times speedup compared to the benchmarked software implementation.

CUDAlign [75] is a GPU based pairwise sequence alignment program aimed at aligning two large sequences of more than a million bases (tests were performed on 47-million base sequences). It only operates on DNA, not proteins.

Multiple sequence alignment

An early OpenGL-based multiple sequence alignment method is discussed in [60]. Multiple pairwise alignments are performed in parallel to keep the processing elements busy, and sequences are sorted by length to prevent superfluous computations. The authors later implemented a CUDA based method, discussed in [64], which works similar to ClustalW (Section 2.2.3) and managed to obtain a speedup of more than 36 times compared to a CPU implementation.

Heuristics

CUDA-BLASTP [5] is a GPU implementation of BLAST (Secton 2.2.2) and is claimed to be ten times faster than CPU BLAST. However, not much further information on this method is available in the literature.

In [41] and [51] the Viterbi algorithm (Section 2.3) part of HMMER is GPU accelerated.

3.4 Optimizing GPU accelerated Smith-Waterman

Since the approach detailed in [65], various improvements have been suggested. As detailed in Section 3.1.3, GPUs are sensitive to how the various memories are used; optimizing this is a good starting point for speed improvements. In [31], it is observed that storing the query profiles in texture memory can lead to a large amount of cache misses. To combat this, it is opted to store the substitution matrix in constant memory, ordered alphabetically so the values can be looked up by ASCII value.

In [62] the database sequences are stored in vertical columns of global memory, so the reading of the database across multiple threads can be coalesced. Furthermore, writes to global memory are first batched in shared memory for better coalescing. Finally, for sequences of more than 3,072 amino acids an ‘inter-task parallelization’ method similar to the systolic array and OpenGL approaches is used as this, while slower, requires

less memory. This 'CUDASW++' solution manages a maximum speed of about 9.5 GCUPS searching Swiss-Prot on a Geforce GTX 280. The authors examined the 'SWAT' optimization, which is a heuristic that allows certain cells to be skipped in some cases. However, this actually decreased performance due to warp control divergence.

An improved version, 'CUDASW++ 2.0' [63] has been released. It differs from the original by using query profiles in texture memory like the implementation described in [65] and by more closely packing data in vectors. Additionally, an alternative approach is evaluated: porting a SIMD CPU algorithm [44] to the GPU, viewing collections of processing elements as part of a single vector. This approach resulted in performance similar to the alignment-per-element method used before. CUDASW++ 2.0 is currently the fastest GPU implementation of Smith-Waterman and manages 17 GCUPS on a single GTX 280 GPU; it outperforms BLAST in its benchmarks.

As generally each processing element processes its own database sequences completely independently from the others, computation can be sped up by using multiple GPUs, multiplying the amount of processors. The database is simply split up across the different GPUs and generally a somewhat linear performance increase is observed [63] [65]. Additionally, in [56] multiple PCs are networked using *Message Passing Interface* (MPI) so their GPUs can work in conjunction.

A GPU-accelerated protein database search tool

4

This Chapter discusses the steps taken to produce a Smith-Waterman based database search tool for GPUs. Section 4.1 goes into the goals set for the implementation, and the design decisions that would shape it. A simple implementation is presented in Section 4.2. Next various program and practical database optimizations are discussed in Sections 4.3 and 4.4; the speed gains resulting from these optimizations are summarized in Section 4.5. Finally, a user friendly web interface for the program is presented in Section 4.6. For a summary of the implementation's final features and limitations please see its user's guide (Appendix B)

4.1 Requirements and design decisions

4.1.1 Goals and requirements

The following goals were decided on:

- A GPU-based bioinformatics database search tool will be implemented and tested to compare with other CPU and GPU implementations, theoretical limits will be looked at, etc.
- The program will be based on the Smith-Waterman algorithm so optimal alignments can be found. Affine gap penalties will be used.
- The program will be completely usable (i.e. not just a proof of concept) and fully-featured with user configurable settings and lenient limits on input data.
- The program will offer an easy to use interface that makes it convenient for researchers to use.

4.1.2 Fundamental design decisions

From the discussed goals, knowledge of GPU programming and the existing implementations looked at in Section 3.2, a set of basic decisions were made that would govern the rest of the implementation.

4.1.2.1 Implementation languages and toolkits

NVIDIA CUDA [6] was decided on as the toolkit to be used in the implementation phase. As of writing, CUDA is the most mature GPU programming toolkit and has been successfully used for the previously discussed GPU-based bioinformatics tools. The host-side language used in conjunction with CUDA is C++.

4.1.2.2 Target database

It was decided to focus on protein (not DNA) searches and the Swiss-Prot database [29] in particular. The reasons for this are twofold. First of all, as protein searches are more complicated due to the use of substitution matrices and a larger alphabet, supporting DNA searches once proteins can be searched is relatively simple. The second reason is that modern GPUs have around one gigabyte of on-board memory, while INSDC databases [16] such a GenBank are 200+ gigabytes in size; searching those would require working around memory limitations. In contrast, the August 2010 release of Swiss-Prot weighs in at 232 megabytes. It should also be noted that existing GPU implementations mostly focus on Swiss-Prot as well, making for easier performance comparisons.

4.1.2.3 Returning alignment scores

The choice was made to only have the search tool return maximum Smith-Waterman scores, not the actual alignments. This approach is generally used by the FPGA and GPU projects discussed in Section 2.4 and Chapter 3. Doing this significantly simplifies the implementation as the algorithm's traceback step can be skipped; additionally no traceback data structures, such as a pointer list, need to be kept, decreasing memory consumption.

To be able to meet the goal of being a fully-featured search tool, however, the ability to produce full alignments was deemed a necessity. To accomplish this, the decision was made to add the feature of exporting a user-configurable number of top-scoring sequences to a new database file. This database can then be searched by a third-party (CPU) tool that generates full alignments, such as the FASTA suite's SSearch program. This approach does of course lead to redundancy as some sequences are aligned twice. However, the number of such sequences is relatively tiny: by default 20 top scoring sequences are returned while all of Swiss-Prot contains more than 500,000. For practical information on using an external tool to find full alignments, see the implementation's user's guide (Appendix B).

4.1.2.4 Algorithm parallelism

As discussed in Section 2.4 and Chapter 3, multiple processing elements can be used to search a single sequence, passing data from element to element in a systolic array fashion; another approach is to have different processors perform alignments for different sequences in parallel. These approaches can even be blended, as discussed in Section 3.4's description of [62], where multiple processing units are used for long sequences.

The second approach, that of performing a complete individual alignment on each processing element, was chosen for various reasons. First of all, as a GPU has a fixed amount of processing elements that offer a full instruction set, there is no trade-off between the number of processing elements versus their features, which is the case when working with an FPGA. Secondly, it eliminates the need for inter-processing-element communications or, even worse, inter-multiprocessor communications. As described in Section 3.1.3, the processing elements of one CUDA multiprocessor can communicate using shared memory, while slow global memory must be used to transfer data between

multiprocessors. Apart from memory accesses being a major bottleneck (as reaffirmed in Section 4.3), each memory type has its own set of caveats and guidelines for optimal usage. In short, avoiding the systolic-array style approach to parallelism and, as such, the need for inter-processor communication, simplifies the implementation. Finally, apart from avoiding the need for communication, performing one alignment on each processing element results in a kernel where each processing element is doing the exact same thing independently. This not only bodes well for converging execution paths, but also simplifies implementation and testing, as there is little parallelism to deal with in the algorithm itself.

GPUs might contain hundreds of processing elements (240 for the NVIDIA GTX275 used in development), but Swiss-Prot contains more than 500,000 sequences as of writing. This means that it should be possible to keep all processing elements well occupied. A disadvantage of the chosen approach to parallelism is that of different sequence lengths: one processing element could be aligning with a database sequence of tens of thousands of proteins while another might be working on a sequence of just a few. As a result, the element that finishes first might be idle while the long alignment is being handled.

Furthermore, unless care is taken when assigning sequences to processing elements, this effect might be compounded by some processing elements having to handle multiple sequences, resulting in a larger total length than the workload for others. Section 4.4 discusses this problem in more detail and describes how the implementation works around it.

4.1.2.5 User interface

For ease of development and usage flexibility, the search tool was decided to be implemented as a command-line application. This way the program could easily be tested or incorporated into a script, while also being easier to port to different platforms. To offer a more attractive, easy to use interface, a web-based interface was decided on. Such an interface not only works on any platform that supports the necessary web server, but also allows users to run alignments from client computers that do not have a powerful GPU. Finally, it can offer links to database (Swiss-Prot) annotations on used sequences.

PHP [19] was chosen as the language for the web interface since this language is commonly used for all sorts of web projects [20] and is supported by the popular Apache [22] and IIS [25] web servers.

4.2 A straightforward implementation

This section describes a straightforward, largely unoptimized implementation of the database search tool. First, a simple general algorithm is presented, with no GPU specific traits. Next, used input file types are discussed. Finally, the algorithm is ported to the GPU. This implementation is then optimized in the next section.

4.2.1 Basic algorithm

An example implementation is shown in Figure 4.1. This version of the Smith-Waterman algorithm uses linear gap penalties (see Equation 2.5) and does not include any GPU-specific code. In fact, this specific implementation will perform alignments for the complete database, effectively offering no parallelism at all.

Although simple, this implementation already includes some specific design decisions. First of all, instead of a score matrix for each alignment, only a sole score column is kept: as no traceback is performed, values do not need to be saved for the duration of the algorithm and can be overwritten. This column stores the values to the left of the currently processing column: $F(i, j-1)$ in Equation 2.5. The size of this temporary data column is set to the query sequence, not database sequence size. This way the column can have one fixed size for all database sequences, while usually requiring less memory as it is unlikely that the query sequence will be as long as the longest database sequence. The column is set to zero whenever a new database sequence is started. Apart from the temporary score column for values to the current cell's left, variables are used to keep the values of the top and top-left cells required by the algorithm ($F(i-1, j)$ and $F(i-1, j-1)$ in Equation 2.5).

Note how the query, not database, sequence is accessed in the inner loop. The inner sequence is accessed for every single outer sequence letter; it would be attractive to store it in a fast (small) memory. The (sole) query sequence is a much better candidate for this than a database of hundreds of thousands of sequences.

Figure 4.2 shows the implementation extended with support for affine gap penalties (see Equation 2.6, 2.7, 2.8). This slightly complicates matters: a second temporary data column is added for I_x values. Additionally, an upper I_y value is kept.

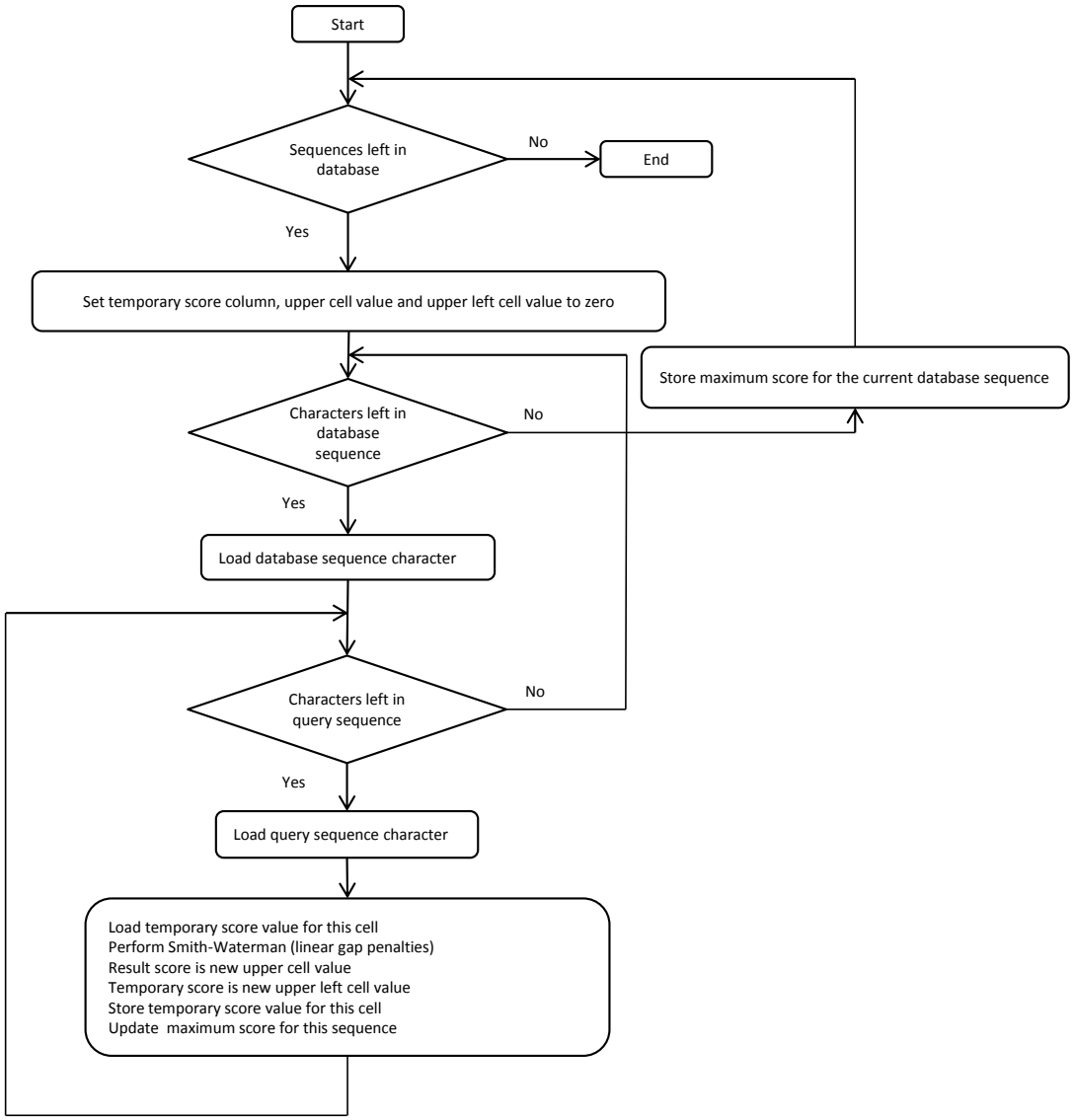


Figure 4.1: Basic Smith-Waterman algorithm.

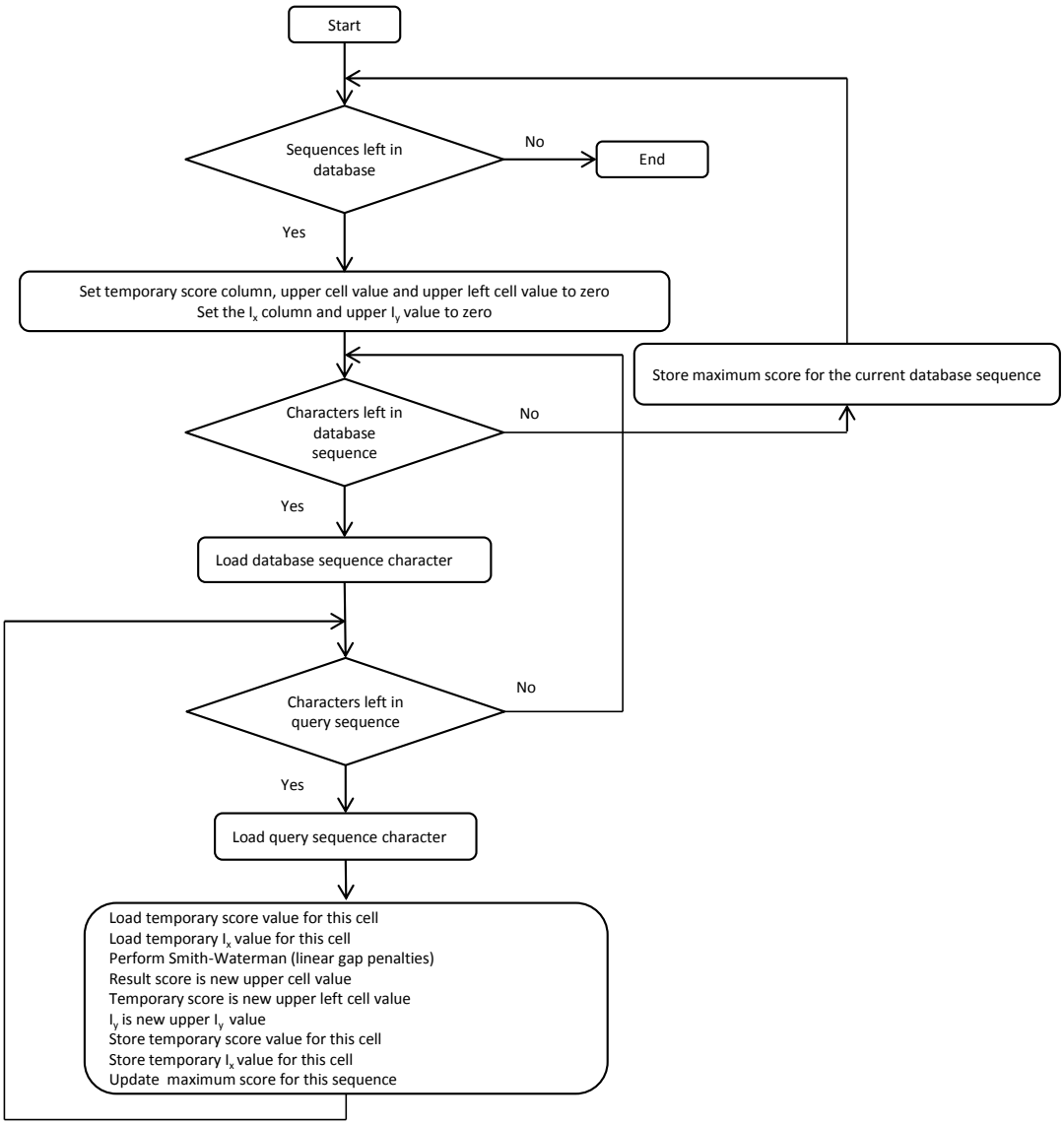


Figure 4.2: Smith-Waterman with affine gap penalties.

```
1 >K1HUAG | 1091 |  Ig kappa chain V-I region (Ag) - Human @P:51-90
2 DIQMTQSPSSLSASVGDRVTITCQASQDINHYLNWYQQGPKKAPKILIYDASNLETGVPs
3 rfsgsgfgtdftftisgLQPEDIATYYCQQYDTLPRTFGQGTKLEIKR*
4
5 >CCHU | 1 | Cytochrome c - Human @P:25-85
6 MGDVEKGKKIFIMKCSQCHTVEKGGKHKTGPNLHGLFGRKTGQAPGYSYTAANKNKGIIW
7 GEDTLMEYLENPKKYIPGTKMIFVGIKKKEERADLIAYLKKATNE
```

Listing 4.1: FASTA format database

4.2.2 File formats

The discussed figures gloss over the code where the query sequence, database and substitution matrix are loaded. Loader classes were written for FASTA format databases and substitution matrices; furthermore, a database format specifically tailored towards GPU use was designed and an application to convert FASTA databases to this format was developed.

The FASTA sequence format is a simple plain-text format in which popular databases such as Swiss-Prot can be downloaded; a simple two-sequence example database is shown in Listing 4.1. Each entry consists of a larger-than sign, a comment that might include a database-specific accession identifier and then a multi-line sequence that goes on until a next larger-than sign or the end of the file is encountered. Query sequences are simply files that contain a single sequence.

Sequences and their descriptions are loaded into records, stripping newlines and whitespace from sequences and converting them to all-upper-case. Various information such as sequence lengths, database size and the total amount of symbols in the database is recorded.

FASTA files support the symbols {UO*}, which are not present in FASTA format substitution matrices. These three symbols are replaced by {CKX} upon loading the file, in line with the FASTA application suite [2]. In practice this means that the selenocysteine protein is replaced by 'normal' cysteine, pyrrolycine is replaced by lycine and that a translation stop (stop codon) is replaced by the wildcard-like 'any' symbol.

Furthermore, to allow for quick array-based substitution matrix indexing, internally the symbols are replaced by their numeric position in an (arbitrarily ordered) array of the letters.

Query sequences are loaded directly, but FASTA-format sequence databases are converted to another format to better match the GPU's capabilities. First of all, sequence descriptions are useless on the GPU: only the sequence index in the database is needed to assign it a score. Secondly, substitution matrix lookups can be simplified by using numeric values instead of letters for sequence symbols. Finally there are the matters of *coalescing* and preventing *divergence*. Coalescing is described in Section 3.1.3. The issue of divergence involves the single-instruction-multiple-thread nature of GPU multiprocessors: if threads in a half-warp have diverging execution paths, these will be executed in turn. This can result in major slowdowns.

To take care of these issues, FASTA format databases are converted to a custom

'GPUDB' format. This is done by the `dbconv` tool. A database needs to be converted only once, after which it is stored on disk in the new format. The conversion process is illustrated in Figure 4.3.

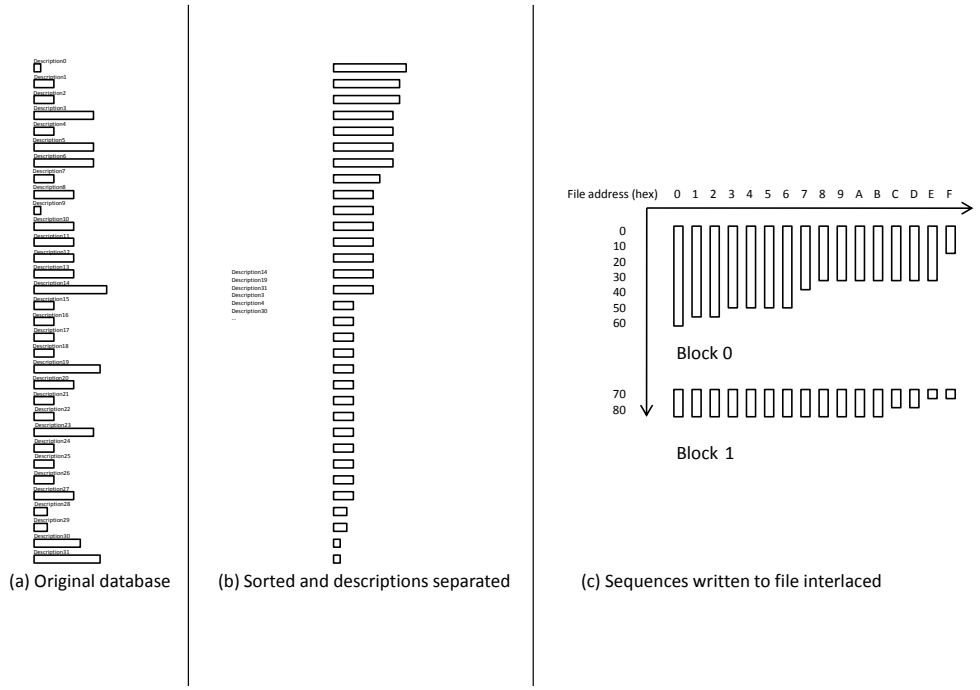


Figure 4.3: The first form of database conversion.

First the FASTA file is loaded as always, as shown in Figure 4.3a. Next it is sorted by sequence length (Figure 4.3b): this way a half-warp processing neighboring sequences will have less threads idle while waiting for longer sequences to finish. They must be idle instead of processing another sequence, as this would both lead to divergence and prevent coalescing. The sorted sequences are then written to file in an interlaced fashion (Figure 4.3c): the database is split up into half-warp sized *blocks* of 16 sequences. The sequences of a block are written in an alternating fashion: one `char`-format symbol of the first sequence is written, then one of the second one, etc. This way, the first character of all 16 sequences is written, then the second character, etc. In this simple version, all sequences in a rectangular block are padded to the length of the block's longest sequence with blank symbols. As each interlaced block has the size of a half-warp (16 sequences), all of a half-warp's threads will load database symbols from neighboring addresses: coalescing takes place.

Apart from the sequences, the file contains a header that includes the number of sequences in the database and the total number of symbols in the database. Furthermore, it contains a list of offsets to each block and a list of sequence lengths so the GPU kernel knows where to start and when to stop aligning. Each portion of the file is aligned to a 256-byte boundary to guarantee optimal access by the GPU.

The sequence descriptions are written to a separate file, in the sorted order used by

```
1 #  Matrix made by matblas from blosum62.iij
2    A  R  N  D  C
3 A  4 -1 -2 -2  0
4 R -1  5  0 -2 -3
5 N -2  0  6  1 -3
6 D -2 -2  1  6 -3
7 C  0 -3 -3 -3  9
```

Listing 4.2: FASTA format matrix

the GPU database. These descriptions are never uploaded to the GPU; the search tool uses them when displaying alignment scores and when exporting sequences back to a FASTA format database.

Besides sequences, another type of input data is required: a substitution matrix. FASTA format substitution matrices are again plain-text format; a few rows and columns of the BLOSUM62 matrix are shown in Listing 4.2. Substitution matrices are loaded into a 'map' data structure that allows matrix entries to be accessed by their letters, such as '(R,C)'. Functions are offered to convert this map to array-based formats or query profiles in GPU memory.

4.2.3 The GPU implementation

Now that the database has been converted to a format suitable for GPU use, the implementation can be modified to follow suit. A diagram of the GPU implementation is shown in Figure 4.4. As can be seen, the code must be split up into a host and device part, with some redundancy as data structures will need to be copied around. Furthermore, work must be split up among the GPU cores. A simple but fine approach is to just have each half-warp keep processing database blocks until none are left. The resulting host (CPU) code is shown in Figure 4.5; the device (GPU) code is shown in Figure 4.6.

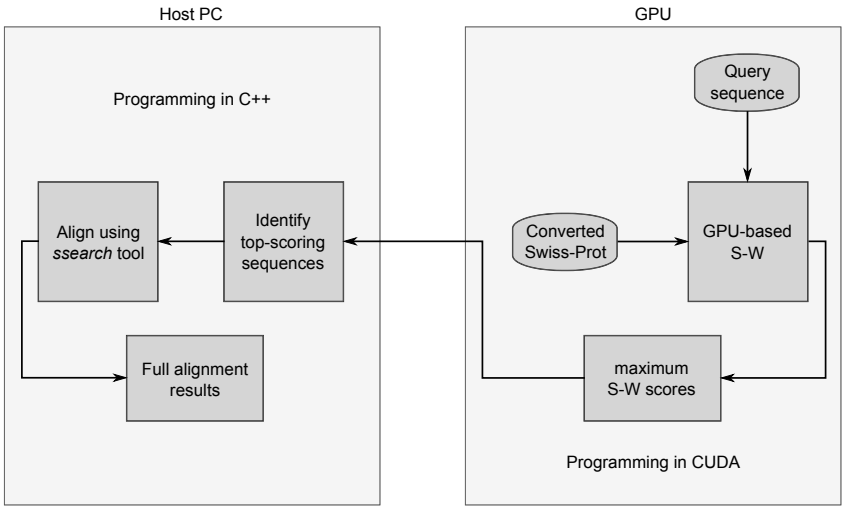


Figure 4.4: Overview of the GPU implementation.

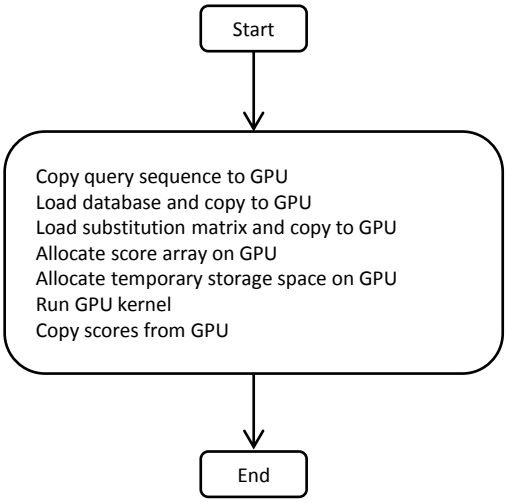


Figure 4.5: Host-side implementation.

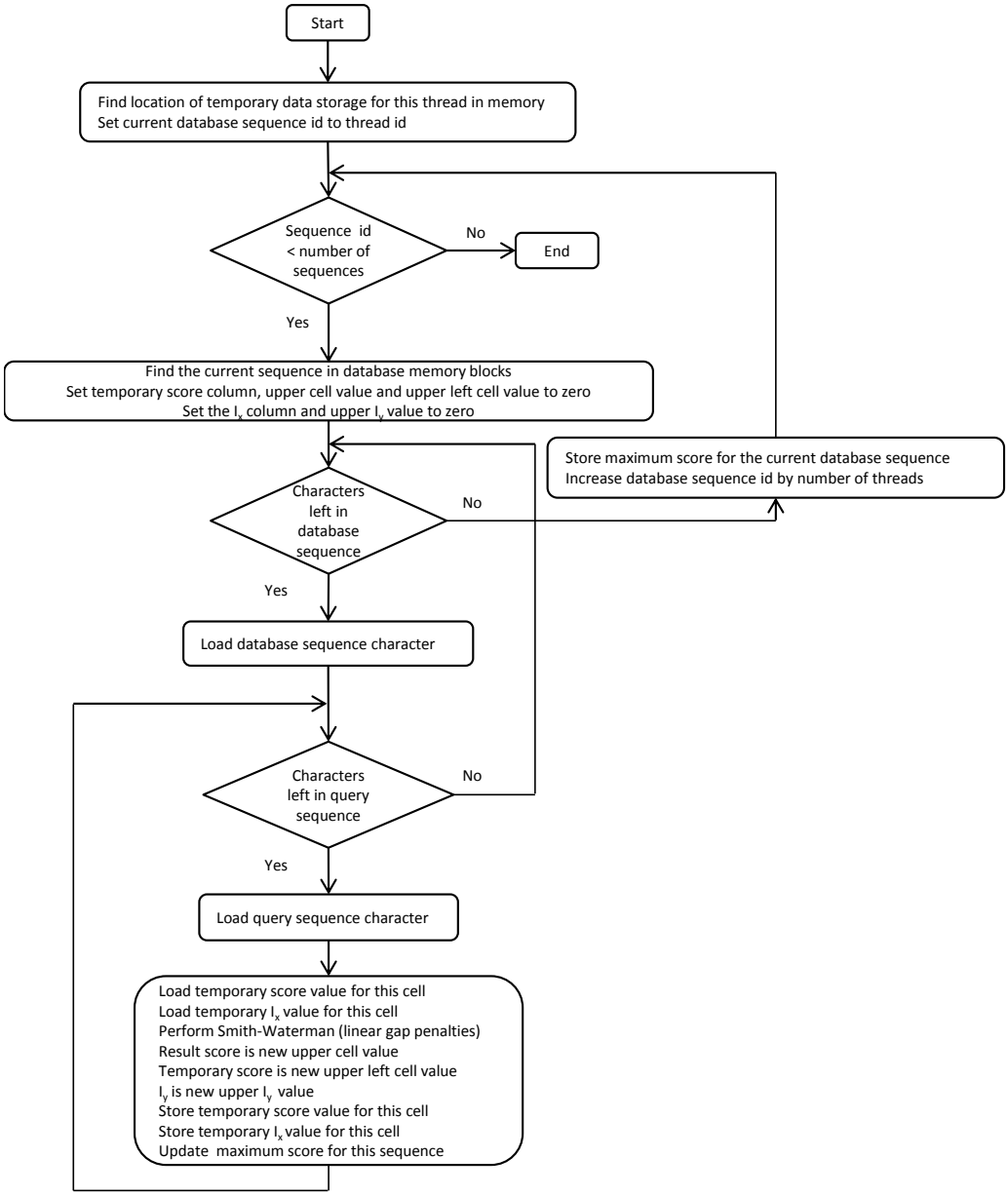


Figure 4.6: Device-side implementation.

The host code is mostly concerned with loading data structures, copying them to the GPU and copying back and presenting the results. One large temporary data area is created, with space for a temporary score and I_x column for each GPU thread. The GPU code bears some more explanation. First, the location where each thread can store its temporary data is determined; this is calculated using the thread id and the length of the temporary data for each thread. This thread id is an unique numeric identifier for each thread; threads in a (half-)warp will have neighboring ids. Next, the thread's current sequence is set to the thread id: this way each thread starts out processing a different (neighboring) sequence. When processing the sequence, the database block it belongs to is determined and it is located in memory, able to be loaded in a coalesced fashion thanks to the database conversion process. Once the database sequence has been processed, the sequence index is increased by the total number of GPU threads, again resulting in each thread processing a different, neighboring sequence. Execution is halted when no sequences are left.

It is important to note how the temporary storage is used in this example: each thread accesses a continuous chunk of $2 \times queryLength$ values, the first half of which is used for score values and the second half of which is used for I_x values. This is shown in Figure 4.7a for two threads and a sequence length of four. Note that the rest of the figure shows optimizations discussed in the next sections.

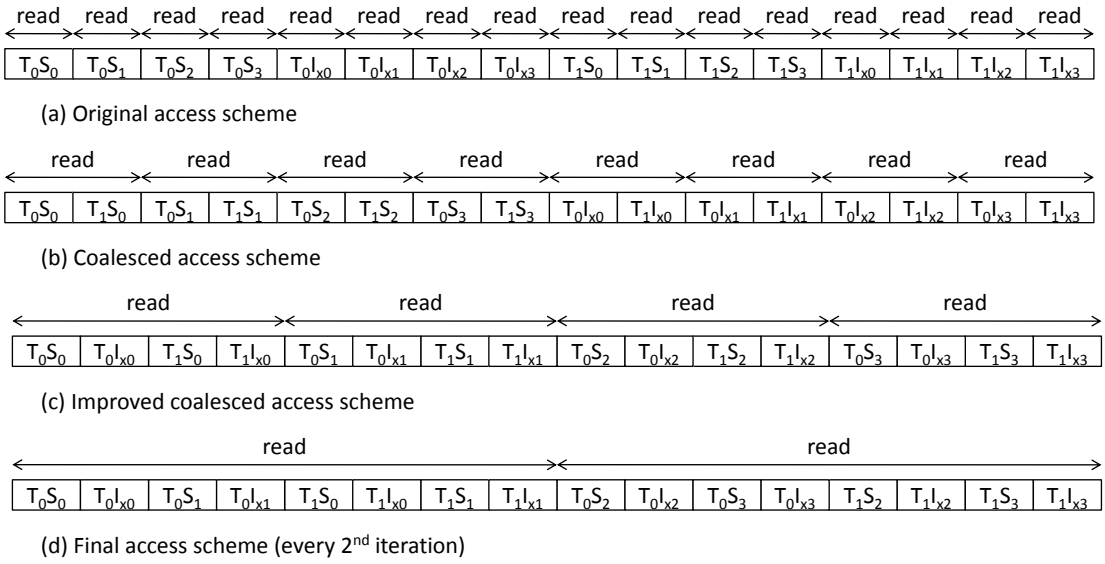


Figure 4.7: Temporary data access schemes. Query length is 4. To decrease the figure's size only two threads, T_0 and T_1, are used. They are in the same half-warp so that their reads can be coalesced. S_i values are temporary scores for query symbol i; I_{xi} are I_x values.

4.3 Optimizing the implementation

Although a fully functioning GPU algorithm has been presented, its simple implementation is extremely slow: more than 30 seconds to search a test database with a 218 residue query sequence. This test database was synthetic: randomly generated sequences of a specifically picked length. To be precise, 240000 sequences of 352 characters. The number of sequences was chosen so that searches would be fast enough for convenient benchmarking. Furthermore, $240000 = 240 \times 1000$. The NVIDIA GTX 275 GPU used during development has 240 processing elements; setting the number of sequences to a multiple of 240 was bound to result in favorable resource usage characteristics. The sequence length of 352 is the average sequence length of the August 2010 release of Swiss-Prot [28]. Finally, it should be noted that all sequences in the database were set to this length. Identical sequence lengths prevent two complications that might negatively influence benchmark results: padding space between sequences of varying lengths and different block sizes, which could result in differing workloads for multiprocessors.

Of course, an actual database such as Swiss-Prot will not have characteristics this favorable. However, it was decided to first focus on improving all-round performance and then comparing this to the Swiss-Prot case, attempting to get that performance as close as possible. That optimization phase is discussed in Section 4.4.

This section discusses the optimization steps taken to eventually reach a benchmark database search time of 0.780 seconds: an almost 40 times speedup.

4.3.1 Sequence and temporary data accesses

This section discusses optimizations related to data accesses: coalesced temporary data accesses, coalesced database reads and multi-character query sequence reads. Figure 4.8 shows the results of these optimizations. A sub-block worth (i.e. eight symbols) of database symbols are read in a coalesced fashion, and for each of these the query sequence is accessed in four-symbol chunks. The symbols are then aligned by aligning each loaded query symbol with the eight database symbols. The figure also shows how temporary data accesses only take place for every second query symbol as described in this section.

Temporary data

Benchmarking and commenting out various parts of the code revealed that memory accesses were the worst bottleneck of the algorithm shown in Figure 4.6. Each inner loop iteration involves reading and writing two temporary values, for four accesses total. As these reads and writes were both uncoalesced, 32 byte reads/writes were issued for each access. This meant that per half-warp $16 \times 32 \times 2 \times 2 = 2048$ bytes of bandwidth were used. Steps were taken to decrease this to one 128-byte coalesced read and write for every second inner loop iteration of the half-warp, for 128 bytes of bandwidth total on average. This is a 16 times improvement. 128 bytes is the largest allowed coalesced access size, and is faster than multiple smaller coalesced accesses.

First it was decided to use the 2-byte `unsigned short` data type for the temporary values, cutting the theoretically required bandwidth in half and allowing for better

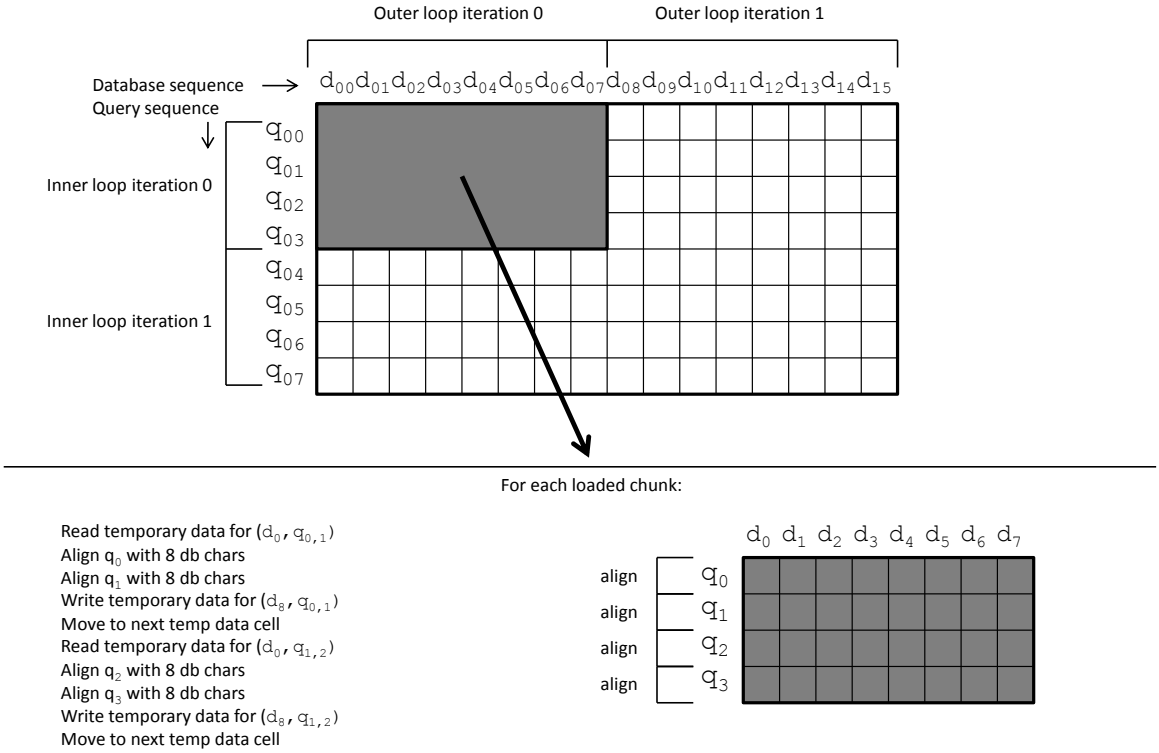


Figure 4.8: Sequence alignment with optimized data accesses.

coalescing later on. A disadvantage of using unsigned shorts is a maximum score value of $2^{16} - 1 = 65535$, however with the longest sequence in Swiss-Prot being 35213 proteins and most sequences being much smaller, this was deemed acceptable. Another disadvantage of the data type is its being unsigned. Although scores will never be less than zero for Smith-Waterman, the temporary I_x values might. This was taken care of by moving the `max(0,...)` operation from the calculation of the score to the calculation of I_x.

The next step was to actually allow for coalescing. This was done by rewriting the algorithm in such a way that each thread stores one temporary value in turn, as shown in Figure 4.7b. Instead of direct array accesses, a pointer into the temporary storage was started at the thread id, and increased by the total number of threads to move to the next cell. As each thread in a half-warp now read a 2-byte unsigned short coalesced, this meant that instead of one 32-byte access per thread, one such access took place per half-warp. This sixteen times bandwidth improvement resulted in an almost ten times speedup.

To again halve the number of memory accesses, the temporary score and I_x values

were interleaved. This is shown in Figure 4.7c. This was done by simply defining a `TempData` structure consisting of two `unsigned shorts` and using this to access the score and I_x values for an iteration in one go. At this point, a half-warp would access two unsigned short values in one read, for a total of $16 \times 2 \times 2$ bytes bandwidth. The result was 64-byte coalesced accesses.

To finally move to 128-byte accesses, the temporary values were interleaved in two-cell chunks. This is shown in Figure 4.7d. Apart from going from two 64 byte to one 128 byte access, this had the additional benefit of temporary reads/writes only being required for every second query sequence symbol processed.

The move to 32-byte coalesced accesses resulted in a huge speed boost of almost ten times; going from there to 128-byte accesses resulted in an improvement of about 10%. In the following item temporary data accesses are further optimized, as a side effect of larger database sequence reads.

Database sequences

Although database sequence accesses were already coalesced as a result of the database conversion process, there was still room for improvement. As the sequences were stored in a per-symbol alternating fashion, each half-warp would issue one 32-byte coalesced read to fetch 16 bytes of sequence data. Not only would larger coalesced reads be preferable, even within the 32-byte read half the bandwidth was wasted. Furthermore, just as important as the resultant coalescing, if not more so, is the fact that larger database fetches mean that more symbols can be processed in one algorithm iteration. By loading x symbols at a time, x database symbols can be aligned with a query symbol before a temporary data cell has to be read and written.

To be able to fetch multiple database sequence symbols in one access, `dbconv` was modified to write sequences in multi-character *sub-blocks*. For example, with a sub-block size of eight, 8 characters of sequence 0 are written, then 8 characters of sequence 1, etc. Sequences are padded to a full amount of sub-blocks with blank data that results in the alignment score not being modified; this was found to be faster than conditionally checking how many characters of the sub-block are valid.

A sub-block size of 8 results in coalesced memory accesses of $16 \times 8 = 128$ bytes and eight times as few temporary data read/writes as before, which in turn resulted in a two times total speed improvement for the complete algorithm. Although faster than the alternatives, such as 4-character sub-blocks for 64-byte accesses, these 8-byte sub-chunks do result in significantly increased register pressure: registers must be allocated to keep eight database characters for the duration of each iteration. Furthermore, the top(left) score and I_y variables now have to hold eight times as much data as before: the temporary data storage has shifted from main memory to the registers. Register pressure is discussed in more detail in Section 4.3.4. As this optimization cut the amount of temporary reads/writes by eight and the optimizations discussed in the part on temporary data did so by sixteen, these inner-loop memory accesses have been decreased by a total of 128 times over the starting case.

Query sequences

From the start, query sequences were stored in constant memory. Constant memory is as fast as reading from a register as long as all threads in a half-warp read the same 4-byte address. This is always true for the discussed implementation as each thread will always be at the exact same iteration within a database symbol alignment; the database sequence length does not matter for this. Constant memory can hold up to 64 kilobytes of data, which was deemed enough as the longest sequence in the August 2010 release of Swiss-Prot is about half that size. Constant memory does not support, or benefit from, coalescing, as this involves threads reading from different addresses.

As the discussed GPUs are 32-bit systems, reading just one character at a time from constant memory was rather wasteful. The algorithm was modified to read four symbols at a time; this resulted in a slight speed improvement. Moving to larger reads, such as eight characters at a time, resulted in increased register pressure with no speedup compared to the single-character case. The final speed improvement was only about 2%. Note that reading the query sequence would become unnecessary with the addition of query profiles (Section 4.3.2).

4.3.2 Substitution matrix accesses

Aligning proteins requires the use of a substitution matrix, as discussed in Section 1.2. This matrix is accessed every time two symbols are aligned, which means that its access speed is critical to the algorithm's performance. Substitution matrix access locations are dependent on both the query and database sequences, complicating the choice of memory used. Global memory is unattractive for usage this frequent to begin with, and the unpredictable nature of the accesses makes them very difficult to coalesce. Constant memory is fast, but only if all threads read the same address, which again does not apply. This is why initially the choice was made to store the matrix in per-multiprocessor shared memory.

Shared memory has a latency roughly 100 times lower than that of global memory. However, some disadvantages apply. First of all, data must be copied from global to shared memory in the kernel code; there is no way to directly upload shared data from the host. Fortunately, this only has to be done once per thread block, and the copying can be threaded by having each processing element copy part of the data. A bigger disadvantage for the use with substitution matrices is shared memory's organization into banks. Shared memory consists of a number of banks (16 or 32 depending on the GPU), with each alternating 32-bit integer belonging to a different bank. This way accesses are sped up if all threads in a (half) warp read neighboring values. However, if the threads access values in the same bank, these accesses are sequentialized, resulting in a performance penalty. As the used FASTA-format substitution matrices are 576 bytes large, each bank was used multiple times, resulting in frequent conflicts. An attempt was made to alleviate bank conflicts by storing a copy of the substitution matrix in each bank and assigning each processing element one of these. However, the logic to deal with this resulted in such register pressure that the result was a net slowdown.

As an alternative, the substitution matrix was stored in texture memory. Texture

memory is a cached window into global memory that offers lower latency and does not require coalescing for best performance. In short, it is well suited to random accesses. Data can also be copied directly to it from the host. Switching from shared to texture memory for the substitution matrix resulted in a total speedup of 25%.

Texture memory has the ability to fetch four values at a time; in graphics usage these will be a texture's color and alpha components. However, this mechanism can also be used to fetch four substitution matrix values from a query profile. Query profiles are described in Section 3.2. To reiterate, they are a substitution matrix with the query sequence, instead of the protein alphabet, used to fill the columns. This means that for a given database symbol, the substitution matrix columns that will be used next are known: multiple substitution scores can be fetched at once.

After giving the matrix loading class the capability to generate query profiles, the alignment algorithm shown in Figure 4.8 was modified to load query profile values for the four current query characters and pass them to the Smith-Waterman function, from which the substitution matrix lookup was removed. As the query symbols were only used to fetch substitution matrix scores, they did not have to be looked up anymore either; the loads from constant memory were removed as well. Only the position within the query is required to load query profile scores for a database character.

As the approach shown in Figure 4.8 aligns each query symbol with the loaded database symbols before advancing on to the next query symbol, query profile values for all eight database symbols had to be loaded. This meant that $8 \times 4 = 64$ values had to be stored in registers, resulting in significantly increased register pressure. For the synthetic test database, performance was 5% slower than before. Swiss-Prot performance was checked as well, however, and gained a 2% performance boost, perhaps due to the sequences being less random.

To decrease register pressure, the algorithm was rewritten to align the four current query symbols (loaded query profile values) with each loaded database symbol. Although this approach results in increased register pressure when no query profile is used, with query profile support it is much more favorable than the original method. This 'transposed loop' algorithm is shown in Figure 4.9. Note that the temporary data column is still accessed using two separate 128-byte reads/writes even though these take place in direct succession. Combining these accesses into 256-byte accesses (containing the data for all four query symbols) resulted in the loss of coalescing and a significant performance penalty. Switching to this query profile method resulted in a 10% performance improvement for the synthetic database and a 17% improvement for Swiss-Prot.

As the GTX 275 GPU used in development has 8KB of texture cache per multiprocessor and each query profile column stores values for 20 amino acids and 3 extra characters, a sequence length longer than $8 \times 1024/23 = 356$ residues will result in increased cache misses, as described in [31]. Searching the test database with a 218-residue query sequence resulted in 7×10^{-7}% of all lookups being misses, while with an 8000-residue sequence, this increased to 9×10^{-3}%. This is still a relatively tiny miss percentage. Furthermore, even for long sequences query profile performance was found to be better than the other tested substitution matrix schemes.

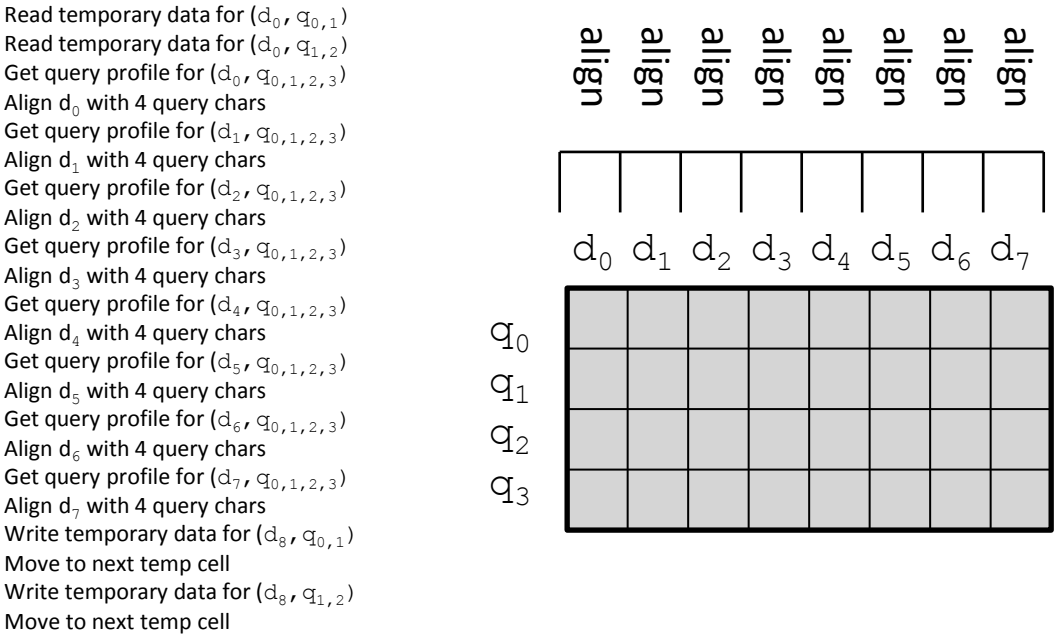


Figure 4.9: Transposed loops: each database symbol is now aligned with the current query symbols; note the vertical 'align' brackets. Query profile support is shown as well.

4.3.3 Miscellaneous optimizations

Various straightforward small-scale optimizations were included:

- The use of optimal data types for structures, to conserve memory and to allow for better coalescing. This includes using the `unsigned short` type for both temporary and final scores and for temporary I_x values.
- The use of numeric instead of letter values for proteins. By assigning each protein a numeric value, they can be looked up in a substitution matrix or query profile directly.
- The use of intrinsic instructions, where appropriate. Fortunately the `max()` operation is already a single instruction, and `__(u)mul24` was used instead of normal multiplications.
- Setting the temporary data to zero when processing a new database sequence was replaced with a multiplication by zero of the loaded value, if the current column is the first one.
- Code was reordered and data was given its own variables or made to reuse existing

ones for maximum speed. This was done by experimenting, and the results of this might be different using other GPUs or compilers.

- The gap penalty and gap extension penalty values were pre-added to make `gapPenaltyTotal`. This and the gap extension penalty were stored in constant memory instead of registers to decrease register pressure, as these values are constant and have to be propagated all the way to the most inner loop. The query sequence length was stored in constant memory as well.

- Integer divisions were replaced by shifts with the Log_2 of the divisor and integer modulus operations such as $x\%y$ with y a power of 2 were replaced by $x\&(y-1)$ in accordance with the CUDA programming guide.

4.3.4 Execution configuration and occupancy

CUDA code cannot just be blindly optimized: there is a trade-off between theoretically faster code and hardware demands that might result in decreased parallelism or the use of slower memory. Kernels are launched in a certain execution configuration of blocks and threads, as discussed in Section 3.1.3. The execution configuration will in part decide the occupancy, the ratio of the possible to maximum number of threads running on a multiprocessor. For CUDA devices with compute capability (feature set) 1.2 and higher, this maximum number of threads per multiprocessor is 1024. A block can contain at most 512 threads. So, for 100% occupancy, two blocks of 512 threads can be launched, or four blocks of 256 threads, etc.

Things are complicated by the fact that threads need to share some of the multiprocessor's resources. All active threads are allocated registers from the same pool, and all concurrently executing blocks have to share the same shared memory space. As the amount of registers required by a kernel increases, the number of concurrent threads that can be run on the multiprocessor decreases, decreasing the occupancy. Decreased occupancy results in worse performance: not only will blocks have to be processed in turn, memory latencies and register dependencies become harder to hide. However, higher occupancy does not always equal higher performance. Occupancy influences the execution configuration: if threads are more demanding, less can be executed per block. The optimal execution configuration depends on the GPU and kernel in question; some experimentation is in order. For the Smith-Waterman database search kernel, the best number of blocks was found to be four times the number of GPU multiprocessors ($4 \times 30 = 120$ on GTX 275). The optimal number of threads per block was found to be 64, 1/8 of the permitted maximum. This results in $120 \times 64 = 7680$ total threads on GTX 275. This configuration was settled on early in development and periodically re-checked; throughout the various optimizations this configuration kept performing the best.

The first completely working, but unoptimized, versions of the kernel required around 20 registers. This resulted in 50% occupancy, enough to hide memory and register access latencies [70]. Most of the discussed optimizations increased register pressure: more query and database symbols had to be stored in registers, as did query profile scores. If too few registers can be allocated, they are spilled into slow local memory, negating the optimization work. A solution is to allow the kernel to consume more registers than

the default 32, which is a compiler setting. However, this in turn decreases occupancy as the multiprocessor's hardware resources are spread thinner. Allowing each thread to use up to 64 registers decreased the occupancy to 25%, a less than optimal amount. However, the performance increases resulting from the optimizations that led to this increased register usage were found to be worth the hit. Implementing query profiles for the original algorithm pushed register usage to over 64 registers, resulting again in either spillage into local memory or an occupancy of just 12.5%. This resulted in an unacceptable performance penalty, but was resolved by transposing the query/database sequence alignment calculations as discussed in Section 4.3.2. The final register usage and occupancy are 63 and 25% respectively.

4.4 Improving practical database search performance

At this point, search performance of the test database had been improved to 0.780 seconds for the 218-residue test sequence, which translates to about 24 GCUPS on the NVIDIA GTX 275 graphics card used. However, Swiss-Prot performance, which is what actually matters, was lagging at around 12 GCUPS. The only possible explanation for this discrepancy was the difference in structure of the databases. The test database was specifically constructed with each sequence set to the same length, to counter any side effects that varying lengths might have had. Swiss-Prot's structure is quite different. The August 2010 version of the database has sequence lengths ranging from 2 to 35213 amino acids; a histogram is shown in Figure 4.10.

4.4.1 Filling database blocks

The database conversion process discussed in Section 4.2.2 sorted the database sequences by size and then bundled them in 16-sequence blocks. Even though the sequences were sorted, significant length differences could arise within a block, especially between the first and sixteenth sequence. For the used Swiss-Prot database, the longest sequence has a length of 35213; the sixteenth-longest is only 9535 residues long. To allow for coalescing, each database block is effectively rectangular, and shorter sequences were padded to the total block length: the length of the longest sequence in the block. This resulted in significantly wasted space and idle processing elements, waiting for their neighbor to finish its longer sequence before the whole half-warp could move on to the next block. Again looking at the Swiss-Prot example, the largest block had a size of $35213 \times 16 = 563408$ residues, of which only 237180 were actually used, which is less than half. In other words, twice the amount of sequence data could have been aligned in the time it took to process the block.

It was concluded that this discrepancy in sequence lengths and the resulting padding might be the reason for the performance difference between the two databases. As such, `dbconv` was rewritten to try and fill wasted space with other sequences. This is shown in Figure 4.11a. Each block now consisted of sixteen sequence groups: sets of concatenated sequences. Each such group consisted of the longest sequence left in the database, followed by the longest sequences that could be found to fill the leftover space

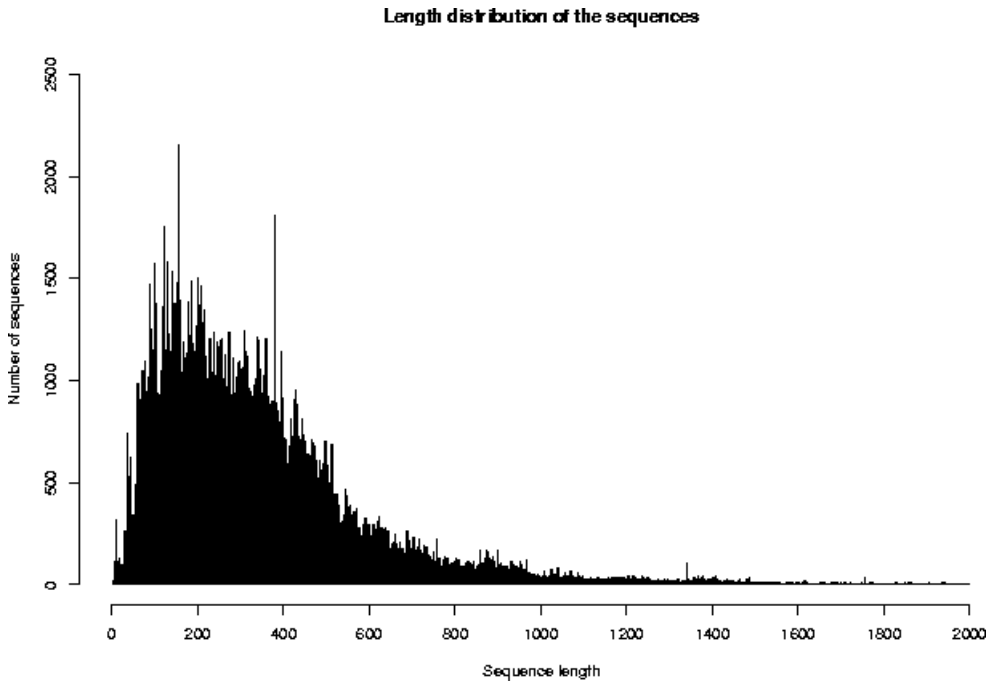


Figure 4.10: Swiss-Prot August 2010 sequence length histogram.

Source: [28].

resulting from the differing lengths. As the first sequence in the block determined the block size, this was always the sole sequence in its group.

To modify the GPU kernel to match this, further changes were made. Instead of looping through the sequences by length, they were now delimited by special terminating sub-blocks; one type to signify the end of a sequence, another type to signify the end of a complete sequence group. These work similar to the null character at the end of C-style strings: when the kernel loads a sub-block, its value is checked and acted upon if necessary. In the case of a sequence group terminator, resulting from there being no suitable sequences to further fill out the block, the kernel sits idle as before. However, upon encountering a sequence terminator, the current sequence score is reset to zero and the current entry into the score array is changed to match the new sequence's index. To facilitate this, for each block the database indexes of its groups' first sequences were written to file. This way when a block gets processed all threads can index into the member of the score array for their group's first sequence. When a new sequence is started, the index only needs to be increased by one. Database descriptions were written to file to match: in the order used to fill the groups instead of their original sorted order.

Upon implementing and testing all this, however, a complete lack of any speed difference was discovered. A reason for this might be the fact that each processing element executes multiple threads in an interleaved fashion; being idle in one thread might be made up for by being able to run other threads more often. Although the performance

difference between databases had to result from the varying sequence lengths, the presence of padding within the blocks seemed not to be the culprit.

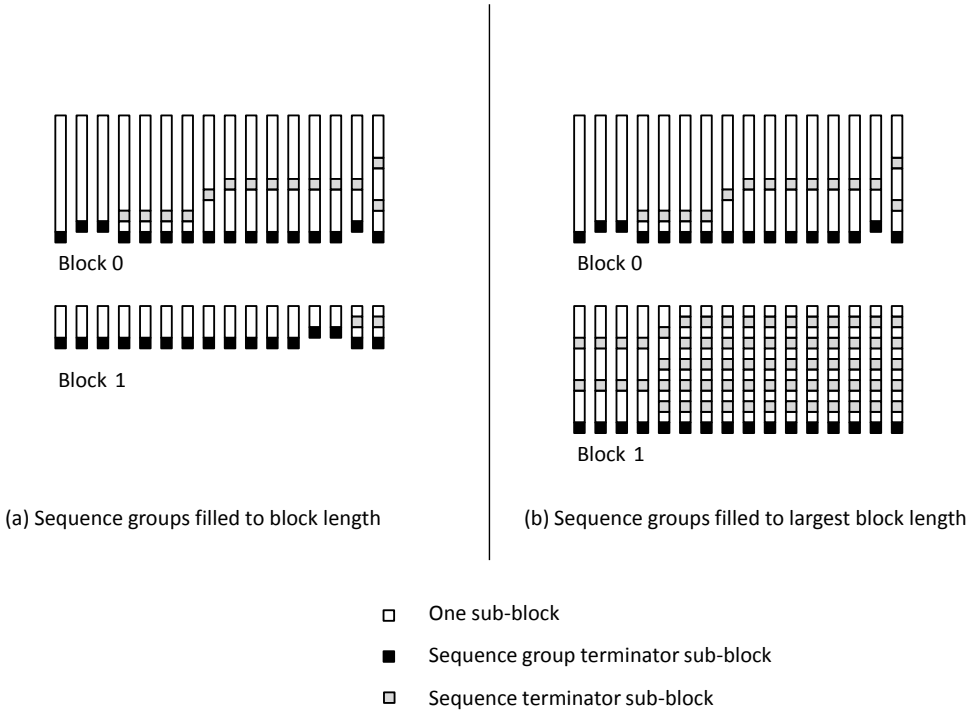


Figure 4.11: Optimized database conversion.

4.4.2 Same-length blocks

Although the sequence groups within a block now had the same length, the blocks themselves still had greatly varying sizes. As block size was determined by the longest sequence in the group, the largest block in the database was 35213 by 16 amino acids; the smallest 2 by 16. This was changed for the final `dbconv` implementation.

Each block is now padded to the length of the longest sequence in the database, filling the leftover space with other sequences similar to what was previously done for the last 15 sequences of each block. This is shown in Figure 4.11b. This resulted in a no less than 42% performance improvement in seconds over the original case (also see Section 5.1.2); this translates to a roughly 1.75 times as high performance in GCUPS, or 21 GCUPS in total. It seems that the fact that each thread block now requires the same amount of work by each multiprocessor is the reason for this speed improvement. At first, multiprocessors were working with thread blocks of wildly differing workloads, of which some might be finished before others, resulting in an inability to interlace their execution. This was further compounded by the fact that there were (much) more sequence blocks

than GPU threads: each half-warp of threads had to process multiple database blocks. If one half-warp ended up processing multiple large blocks in succession while others had to deal with fewer or shorter ones, execution time differences between thread blocks on a multiprocessor and between multiprocessors in general were unavoidable.

This performance of 21 GCUPS means a $24 - 21 = 3$ GCUPS, or 14% performance difference with the test database. This difference can be explained by the overhead in processing the terminating sub-blocks, the fact that not all blocks will be completely filled, and the decreased number of blocks. As each block is now larger than before, there are fewer of them. For example, the August 2010 release of Swiss-Prot is converted into 336 blocks of 16 sequence groups, for $336 \times 16 = 5376$ sequence groups total. As described in Section 4.3.4, on GTX 275 7860 threads are launched, which means that $7680 - 5376$ threads have no work to do. Although total time consumed is now less than if these threads did perform any work, it results in worse interlacing of threads and as such, worse hiding of memory and register accesses. In short, if more threads had work to do, the total runtime would go up but so would the performance in GCUPS. It was attempted to improve performance by, for instance, creating shorter and longer blocks so that there were exactly $7680/16 = 480$ of them, one for each GPU multiprocessor. However, this only resulted in decreased performance due to the now again different block sizes. A look was taken at other methods to schedule sequence blocks in such a way that each GPU thread block would perform the same amount of work. One possible method would be to create sequence blocks of differing lengths as before, but to have each thread block process a selection of database blocks resulting in an about equal workload for each thread. However, this would require some mechanism by which each half-warp can look up which, if any, block to process next. Furthermore, it would be difficult to guarantee equal work per multiprocessor in this case; the details of thread to multiprocessor mapping are vague. It was deemed unlikely that schemes such as this would increase performance at all, let alone bridge the small performance gap left.

Despite this sub-optimality, performance is still significantly higher than before. Furthermore, as the number of sequences in Swiss-Prot grows, so will the number of blocks needed and, in turn, the number of threads able to perform work. Section 5.1.5 discusses this further.

4.5 Summary of optimization steps taken

This section presents a recap of the project's development history, all the optimizations discussed in this chapter thus far, and the speedups obtained. They are presented in the order in which they were implemented, which is not always the same as the order in which they have been shown in this chapter.

Table 4.1 shows the steps taken in optimizing performance for the synthetic test database described in Section 4.3. The query sequence used was Swiss-Prot sequence P10649 which is 218 residues long. Table 4.2 illustrates the effects of the database optimizations discussed in Section 4.4, when searching the October 2010 release of Swiss-Prot with the same query sequence.

Note that due to shifting bottlenecks, the relative performance improvements could have been very different had this implementation order been different. As such, only

global conclusions, such as the importance of coalesced memory accesses and equal work per block can be drawn from these numbers. These performance figures were collected by running the alignment tool in NVIDIA's Compute Visual Profiler profiling application.

Description of optimization step	Execution time (s)	GCUPS	Performance compared to previous (%)
Naive implementation, subst. mat. in shared mem	34.12	0.54	-
Temporary values interleaved	3.443	5.35	991
Database: database sequences interleaved	3.120	5.90	110
Database: 8-character database sub-blocks	2.802	6.57	111
Database: sequence data on 256-byte boundary	2.802	6.57	100
Process 4 db characters at a time	1.471	12.52	191
Load 4 query characters at a time	1.453	12.67	101
Score and I_x temporary values in one access	1.380	13.35	105
Gap penalties in constant memory	1.301	14.16	106
Texture for substitution matrix	1.012	18.20	129
Process 8 db characters at a time	0.909	20.26	111
Access two temporary values at a time	0.873	21.10	104
Query profile with transposed loops	0.780	23.61	112

Table 4.1: Optimization steps for synthetic test database.

Description of optimization step	Execution time (s)	GCUPS	Performance compared to previous (%)
Optimized for synthetic database	2.84	12.23	-
Sequence groups of same length	2.84	12.23	100
Sequence blocks of same length	1.87	21.35	175

Table 4.2: Optimization steps for Swiss-Prot.

4.6 Web interface

The web interface was implemented as a series of PHP scripts. The search interface is shown in Figure 4.12; the results of a query are shown in Figure 4.13. For information on the installation and usage of the web interface, please see Appendix B.

As the GPU-based search tool is command line based, creating a web interface for it simply involved executing it with parameters that match the user's settings and then interpreting the results. The command line is generated using the input from the form shown in Figure 4.12; this form allows the user to upload a query sequence and offers a choice of databases and substitution matrices found on the computer running the script.

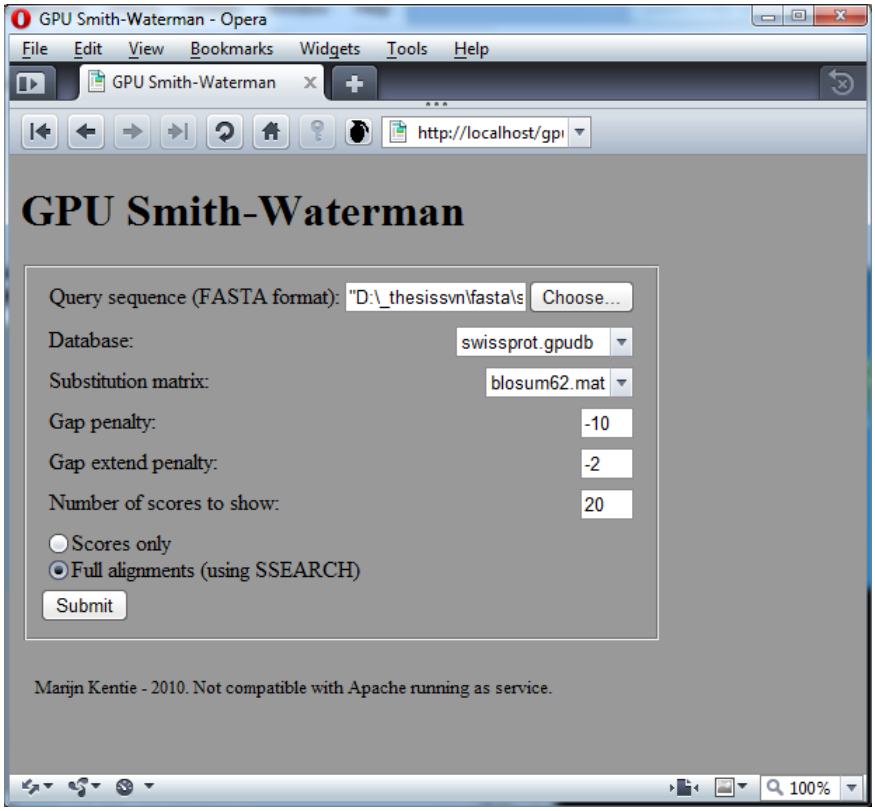


Figure 4.12: Web interface index page.

The result interpretation is done using regular expressions to isolate the lines containing descriptions and scores, and further expressions to split these into a table containing a Swiss-Prot link by sequence identifier and a score. If needed, the user can opt to have full alignments shown. In this case, the search tool is set to export its top scoring sequences to a new FASTA format database, on which then FASTA's CPU-based `ssearch` tool is run. The results are processed in a fashion similar to those of the GPU tool.

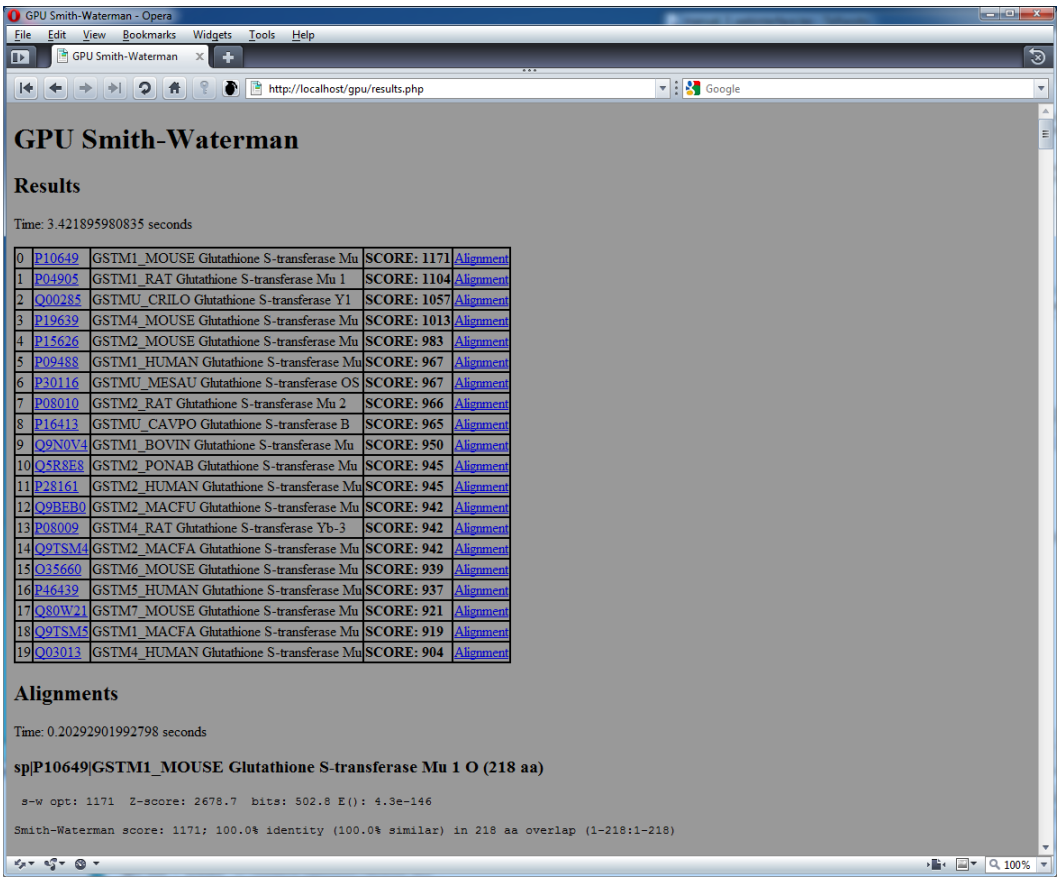


Figure 4.13: Web interface results page

5 Results and discussion

In this chapter the performance of the GPU Smith-Waterman implementation is evaluated. Section 5.1 details the practical and theoretical performance of the implementation, while Section 5.2 presents a comparison with other Smith-Waterman search offerings and looks at the relative costs incurred.

5.1 Performance of the implementation

5.1.1 Practical benchmarks

The completed implementation, with the optimizations discussed in Section 4.3, was benchmarked to determine its real-word performance. This was done by searching the Swiss-Prot database for various query sequences, which come from Swiss-Prot themselves. The set-up used was as follows:

System Intel Core 2 Quad Q6600 (2.4 GHz) with 4GB of RAM and an NVIDIA Geforce GTX 275 graphics card with 896 MB of memory. This graphics card had clock speeds of 633, 1134 and 1404 MHz for its core, memory and shaders respectively. The operating system used was Microsoft Windows 7 Professional 64 bit; the video drivers used were version 257.21. The CUDA toolkit used was version 3.1.

Database The database used was Swiss-Prot release October 2010.

Query sequences Swiss-Prot query sequences were chosen to match the sequences used by the other implementations discussed in Section 5.2.1. 14 sequences were chosen that vary in length from 144 to 5478 amino acids.

Program settings Substitution matrix: BLOSUM62. Gap penalty: -10. Gap extend penalty: -2. These parameters do not influence the execution time.

Measuring method The run-time of the application was timed using the C `clock()` instruction; the accuracy of this approach was verified using the CUDA profiling application. Loading the database and copying it to the GPU was not included in the measured time. This was done as in some cases the database load time is much larger than the actual alignment time, complicating the measurement of the performance for various query sequence lengths. Furthermore, it would be trivial to extend the application in such a way that the database only needs to be loaded once, after which various query sequences can be searched for with no overhead.

Both the execution time and performance in GCUPS were measured. The results of the benchmarks are shown in Table 5.1. This information is shown in graphical form in Figure 5.1. The performance hovers just above 21 GCUPS. Figure 5.2 again shows

the run time, but with the sequence length (horizontal) axis linearized. As can be seen, execution time increases linearly with sequence length, which is why the performance in GCUPS is almost constant.

Sequence	Length	Execution time (s)	Performance (GCUPS)
P02232	144	1.24	21.35
P05013	189	1.65	21.06
P14942	222	1.93	21.15
P07327	375	3.24	21.28
P01008	464	3.99	21.38
P03435	567	4.89	21.32
P27895	1000	8.60	21.38
P07756	1500	12.91	21.36
P04775	2005	17.27	21.35
P19096	2504	21.54	21.37
P28167	3005	25.88	21.35
P0C6B8	3564	30.67	21.37
P20930	4061	34.97	21.35
Q9UKN1	5478	47.15	21.36

Table 5.1: Swiss-Prot performance.

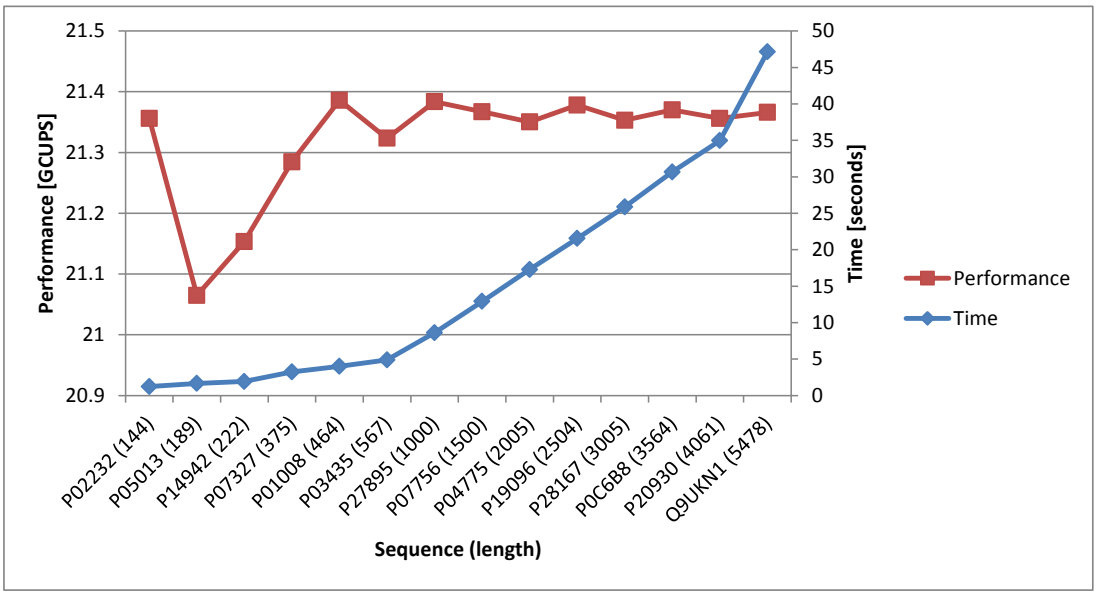


Figure 5.1: Swiss-Prot performance.

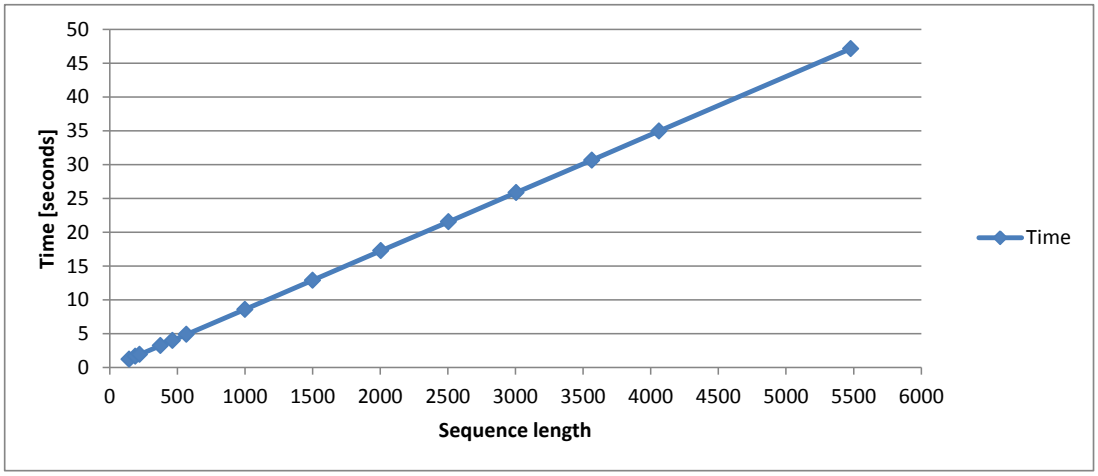


Figure 5.2: Swiss-Prot performance, linearized sequence length axis.

5.1.2 Comparison with less optimized versions

To show the performance impact of a few selected optimizations, the performance of the final implementation was compared with that of earlier versions. The execution time and performance of the final and compared versions is shown in Figure 5.3. The test system, sequences and database were the same as before. The first compared version lacks the database conversion optimized with equal length sequence blocks, as discussed in Section 4.4.2. The other used version lacks the database access optimizations described in Section 4.3.1: database sequences are read one character at a time. Compared to the final version, this results in eight times as many database sequence reads and eight times as many temporary data accesses. In short, global memory is a bottleneck again.

The database block optimization shows the around 75% performance improvement described in Section 4.4.2. Furthermore, it can be seen that global memory accesses are a major hindrance if not taken care of.

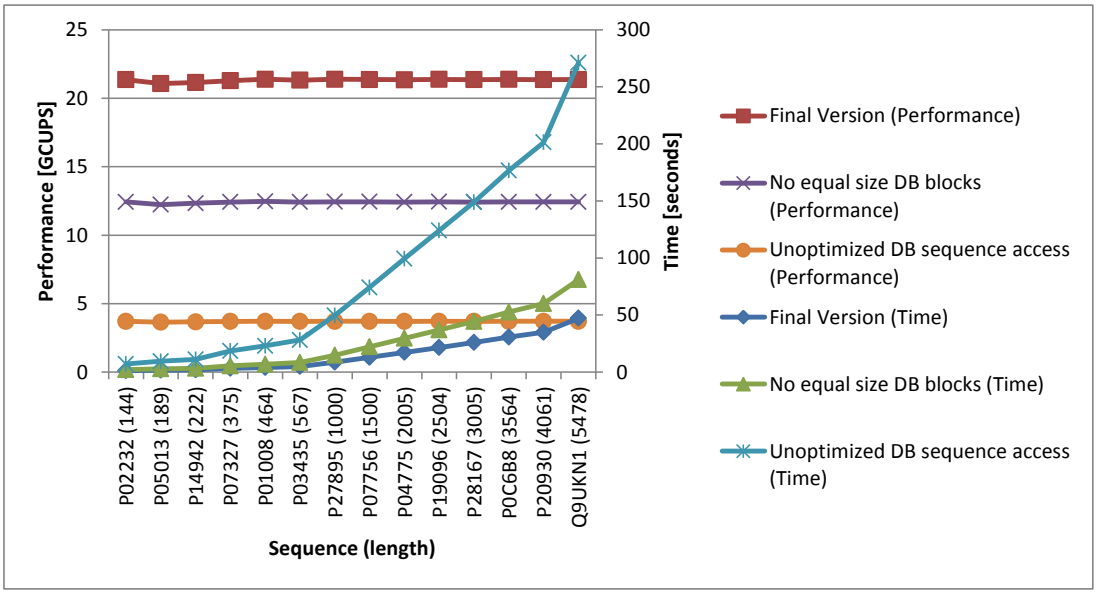


Figure 5.3: Performance of final and less optimized version of the GPU implementation.

5.1.3 Performance when used with SSearch

As described in Section 4.1.2, the SSearch tool can be used in conjunction with the GPU implementation to produce full alignments. The GPU implementation is used to search the database, after which the top scoring sequences are exported to a new FASTA format database. This is then searched using SSearch, which generates the alignments. Of course, doing this only makes sense if running the GPU search tool followed by running SSearch on the exported top scoring sequences is faster than directly running SSearch on Swiss-Prot. Figure 5.4 shows the execution time when:

- Using SSearch to search Swiss-Prot, using the same settings as the GPU implementation. The SSE2 accelerated, multi-threaded version of SSearch was used, running on the same system as the GPU implementation. It should be noted that SSearch rounds its execution time figures down to whole seconds.
- Using the GPU implementation to search Swiss-Prot. As SSearch includes the database load time in its execution time figures, the database load time of the GPU implementation, around 1.6 seconds, is included as well. Otherwise these numbers are identical to those in Table 5.1.
- The total time consumed when searching using the GPU implementation followed by using SSearch to produce full alignments.

As can be seen, the GPU implementation is consistently faster than SSearch, while the time consumed to produce alignments is negligible.

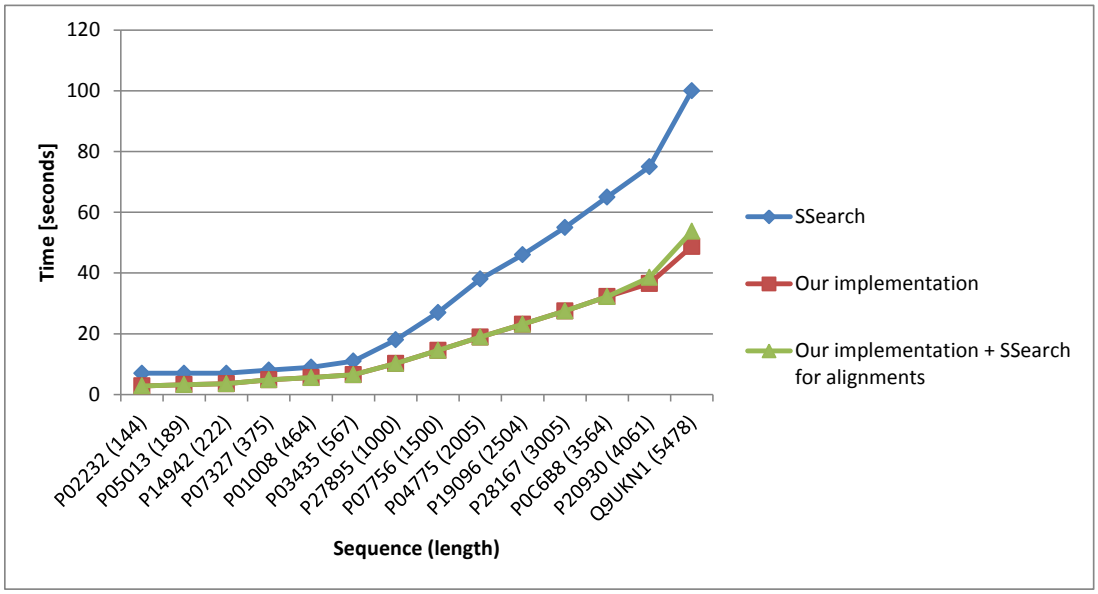


Figure 5.4: Performance comparison with SSearch.

5.1.4 Theoretical limits and bottlenecks

Although the optimizations discussed in Section 4.3 eliminated various real-world performance bottlenecks, it is interesting to take a look at how close the implementation comes to the GPU's theoretical maximum performance.

Database First of all, as discussed in Section 4.4, for an optimal synthetically created database the search performance goes up to 24 GCUPS compared to the current 21 GCUPS for the Swiss-Prot database, due to resource utilization and overhead factors.

Memory bandwidth The maximum theoretical memory bandwidth for a GPU is calculated using Equation 5.1 [70]:

$$Bandwidth = (mem\_speed * 10^6 * (bus\_width/8) * 2)/10^9 \text{ [GB/s]} \quad (5.1)$$

For GTX 275 this is $(1134*10^6*(448/8)*2)/10^9 \approx 127$ GB/s. During benchmarking with the test database about 50 GB/s of bandwidth was used in practice. This can be interpreted in two ways: the maximum possible bandwidth is not utilized; however, on the other hand, due to proper coalescing no more data is transferred than strictly required. In any case, memory bandwidth is not a bottleneck anymore.

Not just pure bandwidth determines performance due to memory factors: latency is an issue too. Many small sequential transfers will be slower than a single larger one. To further investigate the practical effect of memory accesses on performance, the saving and loading of temporary values (the most frequent memory accesses) were commented out. This resulted in a performance of 25.5 GCUPS, only a slight increase. It can be concluded that memory accesses are not a limiting factor anymore.

Arithmetic throughput With memory not being a limiting factor, arithmetic performance is a likely candidate. To test this, the actual Smith-Waterman formula was commented out of the kernel code. This resulted in a performance of 50 GCUPS with the synthetic test database; clearly arithmetic throughput imposes a limit on the practical performance. To further cement this, for the original code the CUDA profiler reported an instruction throughput of 1, which means that instructions are issued at the maximum possible speed. Unfortunately, it seems unlikely instruction throughput can be further optimized: the Smith-Waterman formula consists mostly of maximum operations and additions, which both already map to single instructions. Only low-level optimizations such as minimizing the amount of assembly-code level instructions and register transactions seem helpful.

5.1.5 Scalability and future prospects

5.1.5.1 Database growth

As touched upon in Section 4.4, as the Swiss-Prot database grows, so will the amount of sequence blocks and, as such, threads that have work to do. This will increase future performance effectively for free, just as a result of less threads being idle. However, two caveats apply. First of all, if the number of sequence groups grows to more than the number of GPU threads launched, some processing elements will have to perform multiple alignments, resulting in unequal work between processors and, as such, a performance penalty. However, this issue can be curtailed by increasing the size of each database block by concatenating more sequences to each group, lowering the amount of blocks needed. The second issue is the opposite, and this arises if the longest sequence in the database were to grow: all blocks would grow larger, resulting in less blocks to spread work across. However, this happening is unlikely: as shown in Figure 4.10, long sequences are rare and, in fact, the current longest sequence is significantly longer than

the second-longest one; it is not in danger of being overtaken. In support of all this, a performance increase of 3% was found after updating from the August to October release of Swiss-Prot, which contains 1668 more sequences and resulted in one additional sequence block being created.

5.1.5.2 Future GPUs

The GPU accelerated Smith-Waterman implementation was optimized for GT200-series GPUs as this is what was available for the project. Although the project will run just fine on newer GPUs such as the Geforce 400 series, performance will not be optimal. These newer GPUs offer many more processing elements, which might require the workload distribution to be re-evaluated. They also run two half-warps at a time, and offer a cache hierarchy. This means that memory layouts might need to be changed, or that for example texture memory is not the best option to store a query profile anymore. Furthermore, some instructions perform differently: for example a 24-bit integer multiplication is slower, not faster, than 32-bit one on these GPUs due to architectural reasons. In short, as both CUDA and the GPUs it runs on undergo rapid changes, it is difficult to guarantee future performance, other than the fact that with proper tailoring of the target application, each GPU generation should be inherently faster than the previous.

5.2 Comparison with other implementations

5.2.1 Benchmarks

Swiss-Prot performance was compared to the following other solutions:

BLAST The heuristic method discussed in Section 2.2.2 running on a 2.4 GHz Intel Core 2 Quad processor. Performance figures taken from [78].

CUDASW++ The original CUDASW++ implementation, released in 2009. Compiled for Windows. Running on the same system, with the same settings, as our implementation. CUDASW++ is described in [62].

CUDASW++ 2.0 The fastest GPU implementation so far, released early 2010. Again compiled for Windows and running on the same system as before. It was run in SIMT mode as its SIMD mode resulted in out of memory errors. CUDASW++ 2.0 is described in [63].

SWPS3 An optimized vectorized CPU implementation, faster than the SSE2 implementation of FASTA SSearch; only calculates alignment scores. Running on a Q6600. Performance figures taken from [78].

Unfortunately, FPGA performance figures fit for comparison could not be found; papers on FPGA approaches were either old, aligned DNA and not proteins, or did not offer detailed performance figures. The performance of the benchmarked solutions is shown in Figure 5.5. Averaged over all query sequences, our implementation is 68% faster than BLAST, 12.5% faster than CUDASW++ and 107% faster than SWPS3.

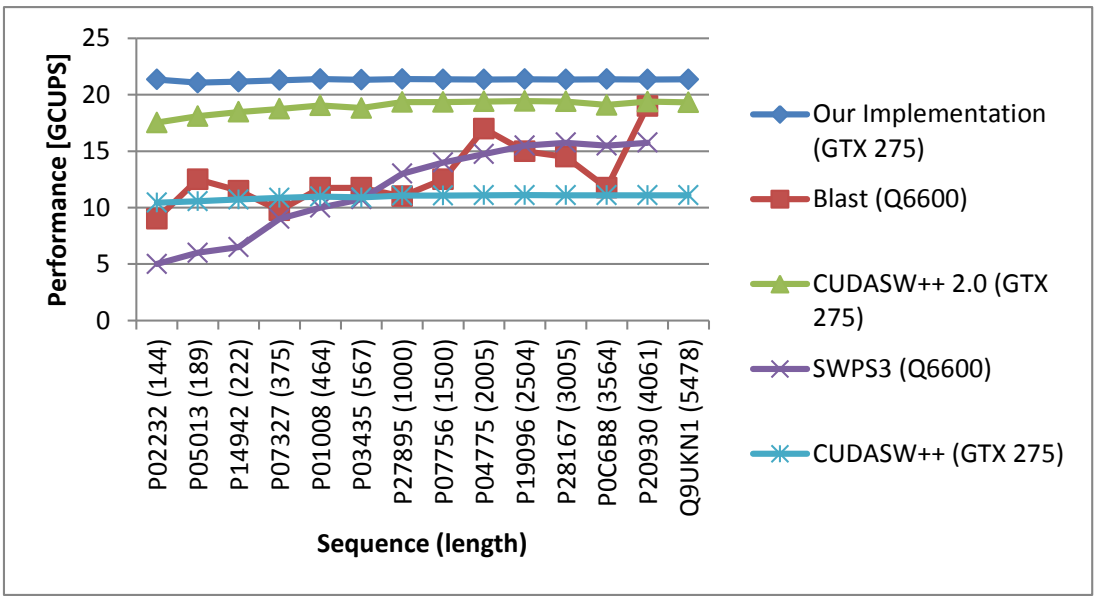


Figure 5.5: Swiss-Prot performance comparison.

Our implementation performs better than BLAST, meaning an optimal dynamic programming based sequence alignment algorithm can be run in less time than a heuristic approach: more accurate results in less time. Comparing to CUDASW++ 2.0, note that our implementation is a simpler design that uses just one search kernel, not two (see Section 3.4), resulting in easier to maintain source code. Furthermore, CUDASW++ 2.0 does not offer a user interface, the ability to use any substitution matrix (only four built in ones), or the ability to produce full alignments. Both CUDASW++ 2.0 and our implementation are sensitive to the structure of the database used: our implementation bases its work distribution on the longest sequence in the database, while CUDASW++ 2.0 uses a hand-picked point at which it switches from one kernel to the other. Finally, it should also be noted that CUDASW++ 2.0 supports the use of multiple GPUs for a speed increase, while our implementation does not. On a Geforce GTX 295, which is essentially two GTX 260 cards in one package, CUDASW++ 2.0 attained a performance of between 24 and 29 GCUPS.

5.2.2 Performance versus cost and power

In practice, raw speed might not be the most interesting metric by which to evaluate the performance of various bioinformatics search engines. The cost of a solution, both in purchase and upkeep, might be more important.

Comparing the purchase price of CPU and GPU hardware is not simple. There are

many different models of both, and their prices fluctuate based on what their manufacturers' competitors are doing. However, experience teaches us that the prices for both are often reasonably close, with medium-end CPUs and GPUs costing around 200 euros and the high end starting at 300 euros. Figures 5.6 and 5.7 show the pricing history of the GTX 275 video card and Q6600 CPU used in the benchmarks; these price figures were aggregated from dozens of online computer parts shops. As can be seen, in 2009 the prices of both hovered around the 200 euro mark: in this case, looking at the performance benchmarks, the GPU offers almost double the performance per euro. However, any system requires a CPU. It might make more sense to consider the cost of purchasing a GPU along with, not instead of, a CPU, making the GPU option less attractive. On the other hand, multiple GPUs can be used in a system (software support notwithstanding), offsetting the cost of the rest of the components, such as the CPU. Furthermore, if one is only interested in GPU accelerated applications, a simple, cheap CPU will suffice.

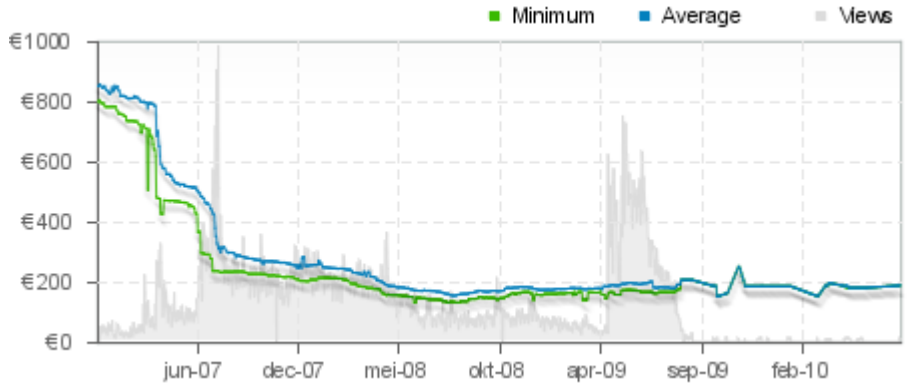


Figure 5.6: Intel Core 2 Quad Q6600 price history.

Source: [27].

Once hardware has been purchased, it still incurs a cost in the form of power usage. Again comparing the Q6600 and GTX 275, the CPU has a maximum power consumption of 100 W [15]. The GPU, on the other hand, consumes 219 W at most [12]. Although it is not guaranteed that this maximum power usage will be reached when performing sequence alignments, fact remains that the GPU is much more power hungry.

Finally, the ease with which a product can be updated to fix bugs or add additional features bears mention. Generally speaking, this will be easier for CPU than for GPU and FPGA solutions: CPU code is much easier to debug and inspect. This became all the more apparent during the implementation of our GPU solution: no debug text can be printed on the screen and errors such as out-of-bounds array accesses would lead to a hanging kernel, sometimes resulting in a corrupted screen or system hang, with no clue as to the actual problem. CPU programs can be interrupted at any time and have advanced debuggers available.

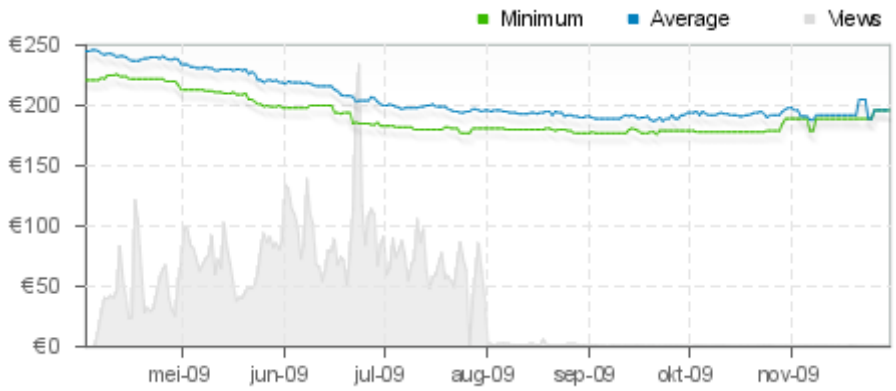


Figure 5.7: NVIDIA Geforce GTX 275 price history.

Source: [27].

Conclusions and recommendations

6

A fully-featured GPU-accelerated protein database search tool was implemented. It can search the Swiss-Prot database using user-provided score and substitution matrix settings. Although only alignment scores are returned, not full alignments, the top scoring sequences can be exported to be processed by a tool that returns these. Furthermore, a graphical web interface is offered to simplify usage of the tool.

6.1 Performance

Our implementation's performance, at around 21 GCUPS, was higher than that of the other CPU and GPU solutions compared: 107% faster than SWPS3 and 12.5% faster than CUDASW++ 2.0, making it the fastest GPU implementation in literature. For our implementation, this performance was capped by the GPU's arithmetic throughput, since memory bottlenecks had been taken care of during development. As the GPU was being throughput limited and the synthetic database designed to fully utilize the GPU resulted in a performance only 14% higher than when searching Swiss-Prot, it is doubtful that other schemes of parallelizing the alignments will result in a significant speed boost.

Besides performance, purchase price and power consumption were also discussed. It was concluded that, though the myriad available CPUs and GPUs complicate comparison, it is possible to purchase a better-performing GPU for the same amount of money one would pay for a CPU otherwise. However, the GPU used in the benchmarks did consume twice as much power as the CPU it was compared to, which makes their performance per Watt about equal.

CPUs are more flexible than GPUs and are of course simply required in a PC. As such purchase price and usage cost of a GPU might have to be added to those of a CPU, not compared to it. On the other hand, if one is only interested in GPU-accelerated tasks a simple, cheap, CPU suffices.

Finally, developing software for GPUs is more complex and time consuming than doing the same for CPU code, while at the same time the lower amount of GPU memory and having to communicate over a bus lead to flexibility limitations.

In conclusion, it is not easy to recommend GPU-based sequence alignment over CPU implementations without some side-notes. Although faster, their additional purchase and upkeep costs, in both power consumption and development support, make GPUs less attractive than they might have been otherwise. But if one must have better protein search performance than a CPU offers, while not wanting to spring for dedicated FPGA-based hardware, then GPUs might be an attractive option.

6.2 Optimizing the implementation

The main issue in extracting performance from the GPU was found to be memory bottlenecks, as the GPU's global memory is relatively slow and requires certain access schemes to be used properly. Performance was improved by taking steps such as laying out temporary data in memory in an optimal fashion, keeping more data in registers, and converting the database to an interlaced format more fitting for GPU use. Furthermore, the best memory area for each type of data was investigated and unnecessary usage of slow global memory was avoided. Memory accesses were decreased many times over the initial implementation, for example 128 times in the case of temporary data reads/writes. With all memory bottlenecks removed, an up to 40 times performance gain was achieved.

To close the performance gap between the test database used during development, which had all sequences of the same length, and the actual Swiss-Prot database, changes were made to the database conversion process. It was found that by concatenating database sequences in such as way that each GPU thread has a roughly equal amount of work to do, Swiss-Prot performance improved by 75%, bringing it close to the optimal case.

6.3 Recommendations for further research

With both our implementation and CUDASW++ 2.0 reaching a maximum performance of around 20 GCUPS on similar hardware, and keeping in mind the conclusions drawn, it is unlikely that more performance can be extracted from the GPUs used. Of course, faster chips keep coming out, which might offer interesting new avenues. At the same time, CPU performance is not static either, with CPUs gaining more and more processing cores for performance leaps in parallel applications. In any case, the differences between GPU generations are significant, and the tools used to develop for them are still in full flux too. As such, it bears regularly re-evaluating the performance of GPU solutions compared to their CPU counterparts. As differences between GPU generations are rather large, such as changes to the threading model or the introduction of different cache hierarchies, this will require careful optimization of these GPU solutions to guarantee maximum performance.

Bibliography

[1] *Bioinformatics Core Methods at cBio*, April 2010, http://cbio.mskcc.org/bic/methods/pairwise_alignment.html.

[2] *ChangeLog - FASTA v36*, September 2010, http://faculty.virginia.edu/wrpearson/fasta/fasta36.

[3] *CLC bio*, April 2010, http://www.clcbio.com.

[4] *Clustal: Multiple Sequence Alignment*, April 2010, http://www.clustal.org.

[5] *CUDA-BLASTP on Tesla GPUs*, June 2010, http://www.nvidia.com/object/blastp_on_tesla.html.

[6] *CUDA zone*, June 2010, http://www.nvidia.com/object/cuda_home_new.html.

[7] *Direct3D 11 Compute Shader More Generality for Advanced Techniques*, June 2010, http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=9f943b2b-53ea-4f80-84b2-f05a360bfc6a.

[8] *DirectCompute*, June 2010, http://www.nvidia.com/object/cuda_directcompute.html.

[9] *DNA Data Bank of Japan*, April 2010, http://www.ddbj.nig.ac.jp.

[10] *European Bioinformatics Institute*, April 2010, http://www.ebi.ac.uk.

[11] *FPGA vs. ASIC*, April 2010, http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm.

[12] *Geforce GTX 275*, September 2010, http://www.nvidia.com/object/product_geforce_gtx_275_us.html.

[13] *GenBank, RefSeq, TPA and UniProt: Whats in a Name?*, April 2010, http://www.ncbi.nlm.nih.gov/bookshelf/br.fcgi?book=handbook.

[14] *HMMER3: a new generation of sequence homology search software*, April 2010, http://hmmer.janelia.org.

[15] *Intel Core 2 Quad Processor Q6600*, September 2010, http://ark.intel.com/Product.aspx?id=29765.

[16] *International Nucleotide Sequence Database Collaboration*, April 2010, http://www.insdc.org.

[17] *National Center for Biotechnology Information*, April 2010, http://www.ncbi.nlm.nih.gov.

[18] *OpenCL*, June 2010, http://www.khronos.org/opencl/.

[19] *PHP: Hypertext Preprocessor*, September 2010, http://www.php.net.

[20] *PHP usage stats*, September 2010, http://www.php.net/usage.php.

[21] *PSI-BLAST/PHI-BLAST*, April 2010, http://www.ebi.ac.uk/Tools/blastpgp.

[22] *The Apache Software Foundation*, September 2010, http://www.apache.org.

[23] *The BLAST Sequence Analysis Tool*, April 2010, http://www.ncbi.nlm.nih.gov/bookshelf/br.fcgi?book=handbook.

[24] *The HMMER user's guide*, April 2010, ftp://selab.janelia.org/pub/software/hmmer3/3.0/Userguide.pdf.

[25] *The Official Microsoft IIS Site*, September 2010, http://www.iis.net.

[26] *TimeLogic*, April 2010, http://www.timelogic.com/.

[27] *Tweakers.net Pricewatch*, September 2010, http://tweakers.net/pricewatch.

[28] *UniProtKB/Swiss-Prot Release 2010_09 statistics*, September 2010, http://www.expasy.ch/sprot/relnotes/relstat.html.

[29] *Universal Protein Resource*, April 2010, http://www.uniprot.org.

[30] *Web BLAST page options*, April 2010, http://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml.

[31] Ali Akoglu and Gregory M. Striemer, *"scalable and highly parallel implementation of smith-waterman on graphics processing unit using cuda"*, Cluster Computing **12** (2009), no. 3, 341–352.

[32] Geoffrey J. Barton, *Protein Sequence Alignment and Database Scanning*, 1996.

[33] John Blyler, *Navigating the Silicon Jungle: FPGA or ASIC?*, Chip Design Magazine (2005).

[34] Boumphreyfr, *Illustration of tRNA building peptide chain*, March 2010, http://en.wikipedia.org/wiki/File:Peptide_syn.png.

[35] C. Chothia and A. M. Lesk, *The relation between the divergence of sequence and structure in proteins.*, The EMBO journal **5** (1986), no. 4, 823–826.

[36] E. T. Chow, J. C. Peterson, M. S. Waterman, T. Hunkapiller, and B. A. Zimmermann, *A systolic array processor for biological information signal processing*, ICS '91: Proceedings of the 5th international conference on Supercomputing (New York, NY, USA), ACM, 1991, pp. 216–223.

[37] Jacques Cohen, *Bioinformatics - an introduction for computer scientists*, ACM Computing Surveys **36** (2004), 122–158.

[38] Craig Freudenrich, *How DNA Works?*, March 2010, `http://science.howstuffworks.com/cellular-microscopic-biology/dna.htm`.

[39] Steven Derrien and Patrice Quinton, *Hardware Acceleration of HMMER on FPGAs*, J. Signal Process. Syst. **58** (2010), no. 1, 53–67.

[40] Andrea Di Blas et al., *The UCSC Kestrel Parallel Processor*, IEEE Trans. Parallel Distrib. Syst. **16** (2005), no. 1, 80–92.

[41] Z Du, Z Yin, and D.A. Bader, *"a tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda"*, "9th IEEE International Workshop on High Performance Computational Biology (HiCOMB)" (2010).

[42] SR Eddy, *Profile hidden Markov models*, Bioinformatics **14** (1998), no. 9, 755–763.

[43] Philippe Faes et al., *Scalable hardware accelerator for comparing DNA and protein sequences*, InfoScale '06: Proceedings of the 1st international conference on Scalable information systems (New York, NY, USA), ACM, 2006, p. 33.

[44] Michael Farrar, *"striped smith–waterman speeds database searches six times over other simd implementations"*, Bioinformatics **23** (2007), no. 2, 156–161.

[45] Robert D. Finn et al., *The Pfam protein families database*, Nucl. Acids Res. **38** (2010), no. suppl_1, D211–222.

[46] M. Gok and C. Yilmaz, *"efficient cell designs for systolic smith-waterman implementations"*, aug. 2006, pp. 1 –4.

[47] Tony Han and Sri Parameswaran, *Swasad: An Asic Design For High Speed Dna Sequence Matching*, ASP-DAC '02: Proceedings of the 2002 Asia and South Pacific Design Automation Conference (Washington, DC, USA), IEEE Computer Society, 2002, p. 541.

[48] L. Hasan and Z. Al-Ars, *An efficient and high performance linear recursive variable expansion implementation of the smith-waterman algorithm*, Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE, sept. 2009, pp. 3845 –3848.

[49] Jakob Hull Havgaard, Rune B. Lyngso, Gary D. Stormo, and Jan Gorodkin, *Pairwise local structural alignment of RNA sequences with sequence similarity less than 40%*, Bioinformatics **21** (2005), no. 9, 1815–1824.

[50] Liisa Holm and Chris Sander, *Protein Structure Comparison by Alignment of Distance Matrices*, Journal of Molecular Biology **233** (1993), no. 1, 123 – 138.

[51] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan, *Clawhmmer: A streaming hmmer-search implementatio*, SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (Washington, DC, USA), IEEE Computer Society, 2005, p. 11.

[52] Alexander Isaev, *Introduction to Mathematical Methods in Bioinformatics (Universitext)*, Springer, 6 2004.

[53] Henry Jakubowski, *Biochemistry Online*, March 2010, `http://employees.csbsju.edu/hjakubowski/classes/ch331/bcintro/default.html`.

[54] John Schmidt, *DNA sequencing gel. Sequence visualized by autoradiography.*, March 2010, `http://en.wikipedia.org/wiki/File:Sequencing.jpg`.

[55] W. James Kent, *BLAT - The BLAST-Like Alignment Tool*, Genome Research **12** (2002), no. 4, 656–664.

[56] Ali Khajeh-Saeed, Stephen Poole, and J. Blair Perot, *Acceleration of the smith-waterman algorithm using single and multiple graphics processors*, J. Comput. Phys. **229** (2010), no. 11, 4247–4258.

[57] John W. Kimball, *Kimball's Biology Pages*, March 2010, `http://biology-pages.info`.

[58] Stefan M. Larson et al., *Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology.*

[59] Weiguo Liu et al., *"bio-sequence database scanning on a gpu"*, In 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006) (HICOMB Workshop), Rhode Island, Greece, 2006.

[60] ———, *Streaming algorithms for biological sequence alignment on gpus*, IEEE Transactions on Parallel and Distributed Systems **18** (2007), 1270–1281.

[61] Yang Liu et al., *GPU accelerated smith-waterman*, Computational Science ICCS 2006, Springer Berlin / Heidelberg.

[62] Yongchao Liu, Douglas Maskell, and Bertil Schmidt, *"cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units"*, BMC Research Notes **2** (2009), no. 1, 73.

[63] Yongchao Liu, Bertil Schmidt, and Douglas Maskell, *"cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions"*, BMC Research Notes **3** (2010), no. 1, 93.

[64] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell, *"msa-cuda: Multiple sequence alignment on graphics processing units with cuda"*, Application-Specific Systems, Architectures and Processors, IEEE International Conference on **0** (2009), 121–128.

[65] S. A. Manavski and G. Valle, *CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment*, BMC Bioinformatics **9 Suppl 2** (2008), S10.

[66] Lisa McMillan and Andrew Martin, *Automatically extracting functionally equivalent proteins from SwissProt*, BMC Bioinformatics **9** (2008), no. 1, 418.

[67] National Institute of Health, *Talking Glossary: Base Pair*, January 2006, `http://www.genome.gov/Glossary/index.cfm?id=16`.

[68] S. Needleman and C. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology **48** (1970), no. 3, 443–453.

[69] Lee Newberg, *Error statistics of hidden markov model and hidden boltzmann model results*, BMC Bioinformatics **10** (2009), no. 1, 212.

[70] NVIDIA, *Nvidia cuda best practices guide 3.0*, 2010.

[71] ———, *Nvidia cuda programming guide 3.0*, 2010.

[72] Tim Oliver, Bertil Schmidt, and Douglas Maskell, *Hyper customized processors for bio-sequence database scanning on FPGAs*, FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays (New York, NY, USA), ACM, 2005, pp. 229–237.

[73] W. R. Pearson, *Rapid and sensitive sequence comparison with FASTP and FASTA*, Meth. Enzymol. **183** (1990), 63–98.

[74] Kiran Puttegowda et al., *A Run-Time Reconfigurable System for Gene-Sequence Searching*, Proceedings of the 16th International Conference on VLSI Design (Washington, DC, USA), VLSID '03, IEEE Computer Society, 2003, pp. 561–.

[75] Edans Flavius O. Sandes and Alba Cristina M.A. de Melo, *Cudalign: using gpu to accelerate the comparison of megabase genomic sequences*, PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (New York, NY, USA), ACM, 2010, pp. 137–146.

[76] Michael Schatz et al., *High-throughput sequence alignment using graphics processing units*, BMC Bioinformatics **8** (2007), no. 1, 474.

[77] T. F. Smith and M. S. Waterman, *Identification of common molecular subsequences*, Journal of Molecular Biology **147** (1981), 195–197.

[78] Adam Szalkowski, Christian Ledergerber, Philipp Krahenbuhl, and Christophe Dessimoz, *Swps3 - fast multi-threaded vectorized smith-waterman for ibm cell/b.e. and x86/sse2*, BMC Research Notes **1** (2008), no. 1, 107.

[79] J. D. Thompson, D. G. Higgins, and T. J. Gibson, *CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice*, Nucleic Acids Res. **22** (1994), 4673–4680.

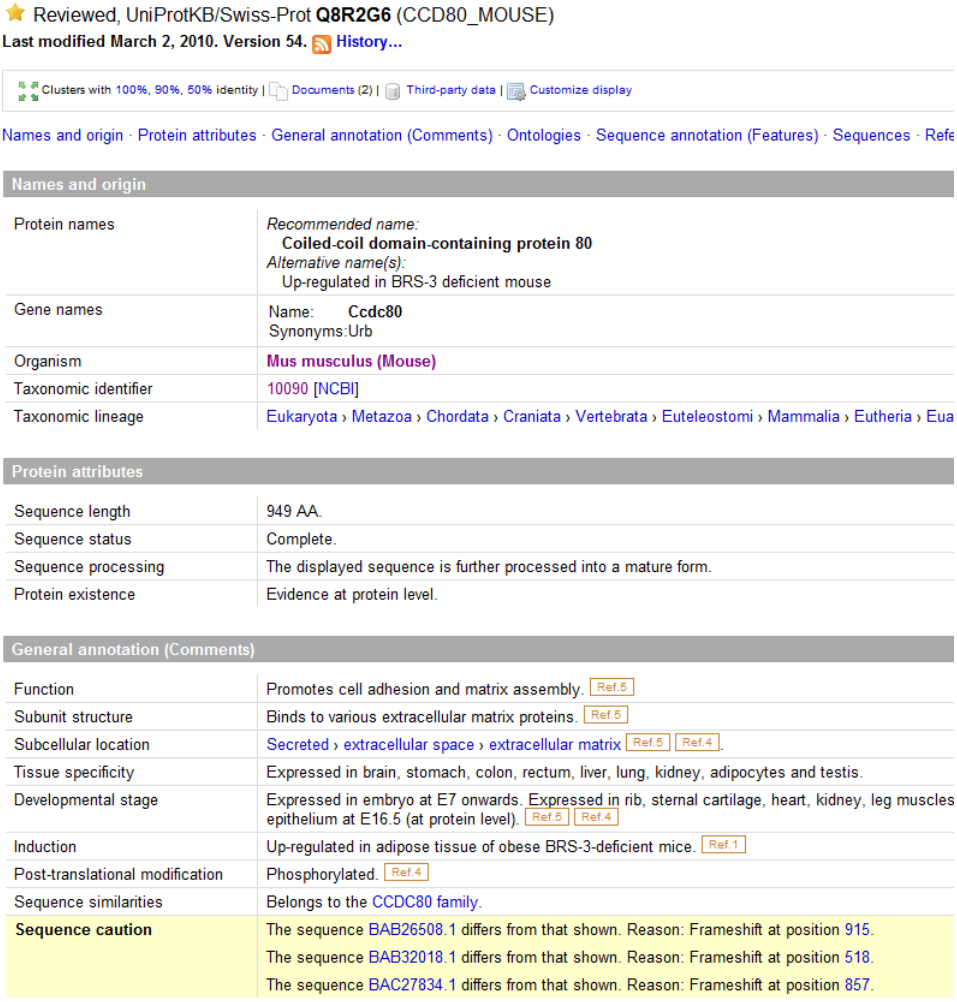
[80] W. J. Wilbur and D. J. Lipman, *Rapid similarity searches of nucleic acid and protein data banks*, Proc. Natl. Acad. Sci. U.S.A. **80** (1983), 726–730.

[81] Peiheng Zhang et al., *Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform*, HPRCTA '07: Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications (New York, NY, USA), ACM, 2007, pp. 39–48.

Example: sequence alignment using SwissProt, EMBL and NCBI BLAST

A

To put the myriad databases, search engines and tools into perspective, the following example shows some of the steps a bioinformatics researcher might take in his research. First, to show the relation between UniProt and the INSDC databases, a protein belonging to a mouse is looked up in UniProt. Then, one of the nucleotide sequences that codes for this protein is then retrieved from EMBL-Bank. Next, to show the duplication of data between the INSDC databases and the usage of the BLAST alignment tool, part of the mouse nucleotide sequence is searched for using NCBI's version of the BLAST tool; the original sequence is among the matches.



Reviewed, UniProtKB/Swiss-Prot **Q8R2G6** (CCD80_MOUSE)

Last modified March 2, 2010. Version 54. History...

Clusters with 100%, 90%, 50% identity | Documents (2) | Third-party data | Customize display

Names and origin · Protein attributes · General annotation (Comments) · Ontologies · Sequence annotation (Features) · Sequences · Refe

Names and origin

Protein names	*Recommended name:* **Coiled-coil domain-containing protein 80** *Alternative name(s):* Up-regulated in BRS-3 deficient mouse
Gene names	Name: **Ccdc80** Synonyms:Urb
Organism	**Mus musculus (Mouse)**
Taxonomic identifier	10090 [NCBI]
Taxonomic lineage	Eukaryota › Metazoa › Chordata › Craniata › Vertebrata › Euteleostomi › Mammalia › Eutheria › Eua

Protein attributes

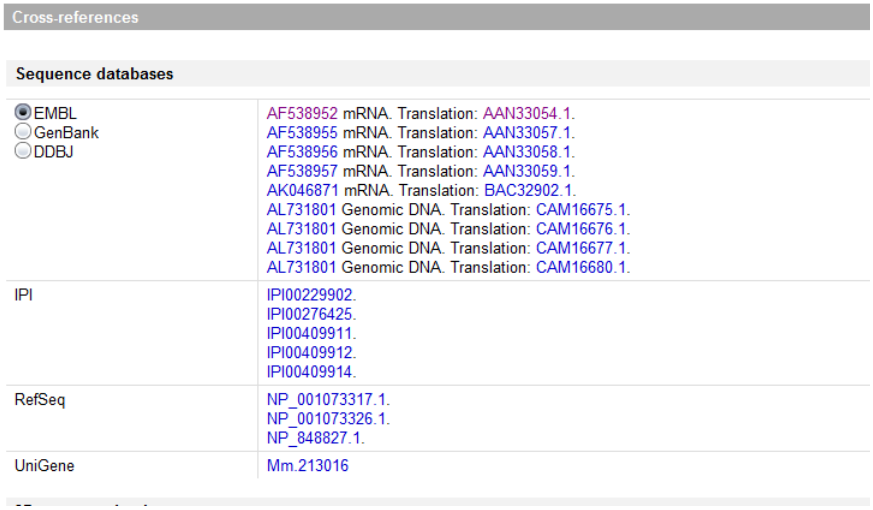
Sequence length	949 AA.
Sequence status	Complete.
Sequence processing	The displayed sequence is further processed into a mature form.
Protein existence	Evidence at protein level.

General annotation (Comments)

Function	Promotes cell adhesion and matrix assembly. Ref.5
Subunit structure	Binds to various extracellular matrix proteins. Ref.5
Subcellular location	Secreted › extracellular space › extracellular matrix Ref.5 Ref.4.
Tissue specificity	Expressed in brain, stomach, colon, rectum, liver, lung, kidney, adipocytes and testis.
Developmental stage	Expressed in embryo at E7 onwards. Expressed in rib, sternal cartilage, heart, kidney, leg muscles epithelium at E16.5 (at protein level). Ref.5 Ref.4
Induction	Up-regulated in adipose tissue of obese BRS-3-deficient mice. Ref.1
Post-translational modification	Phosphorylated. Ref.4
Sequence similarities	Belongs to the CCDC80 family.
Sequence caution	The sequence BAB26508.1 differs from that shown. Reason: Frameshift at position 915. The sequence BAB32018.1 differs from that shown. Reason: Frameshift at position 518. The sequence BAC27834.1 differs from that shown. Reason: Frameshift at position 857.

Figure A.1: Taking a look at an UniProt entry for a mouse protein.

Note the detailed annotations; this is a Swiss-Prot sequence.



Cross-references

Sequence databases

(•) EMBL ( ) GenBank ( ) DDBJ	AF538952 mRNA. Translation: AAN33054.1. AF538955 mRNA. Translation: AAN33057.1. AF538956 mRNA. Translation: AAN33058.1. AF538957 mRNA. Translation: AAN33059.1. AK046871 mRNA. Translation: BAC32902.1. AL731801 Genomic DNA. Translation: CAM16675.1. AL731801 Genomic DNA. Translation: CAM16676.1. AL731801 Genomic DNA. Translation: CAM16677.1. AL731801 Genomic DNA. Translation: CAM16680.1.
IPI	IPI00229902. IPI00276425. IPI00409911. IPI00409912. IPI00409914.
RefSeq	NP_001073317.1. NP_001073326.1. NP_848827.1.
UniGene	Mm.213016

Figure A.2: The sequences the UniProt entry references.

These sequences code for the protein in question. Note the option of looking them up in any of the INSDC databases. Next we will look up the nucleotide sequence with accession number 'AF538952'.

```
FT                   /translation="MLFSGGQYSPVGRPEEVLLIYKIFLVIICFHVILVTSLKENGNSS
FT                   LLSPSAESSLVSLIPYSNGTPDAASEVLSTLNKTEKSKITIVKTFNASGVKSQRNICNL
FT                   SSLCNDSVFFRGEIVFQHDEDHNVTQNQDTANGTFAGVLSLSELKRSELNKTLQTLSET
FT                   YFIVCATAEAQSTVNCTFTVKLNETMNVCAMMVTFQTVQIRPMEQCCCSPRTPCPSSPE
FT                   ELEKLQCELQDPIVCLADQPHGPPLSSSSKPVVPQATIISHVASDFSLAEPLDHALMTP
FT                   STPSLTQESNLPSPQPTIPLASSPATDLPVQSVVVSSLPQTDLSHTLSPVQSSIPSPTT
FT                   PAPSVPTELVTISTPPGETVVNTSTVSDLEAQVSQMEKALSLGSLEPNLAGEMVNRVSK
FT                   LLHSPPALLAPLAQRLLKVVDAIGLQLNFSSTTISLTSPSLALAVIRVNASNFNTTTFA
FT                   AQDPTNLQVSLETPPPENSIGAITLPSSLMNNLPANDVELASRIQFNFFETPALFQDPS
FT                   LENLTLISYVISSSVTNMTIKNLTRNVTVALKHINPSPDDLTVKCVFWDLGRNGGKGGW
FT                   SSDGCSVKDKRMNETICTCSHLTSFGILLDLSRTSLPPSQMMALTFITYIGCGLSSIFL
FT                   SVTLVTYIAFEKIRRDYPSKILIQLCAALLLLNLIFLLDSWIALYNTRGFCIAVAVFLH
FT                   YFLLVSFTWMGLEAFHMYLALVKVFNTYIRKYILKFCIVGWGIPAVVVSIVLTISPDNY
FT                   GIGSYGKFPNGTPDDFCWINSNVVFYIITVVGYFCVIFLLNVSMFIVVLVQLCRIKKKKQ
FT                   LGAQRKTSIQDLRSIAGLTFLLGITWGFAFFAWGPVNVTFMYLFAIFNTLQGFFIFIFY
FT                   CAAKENVRKQWRRYLCCGKLRLAENSDWSKTATNGLKKQTVNQGVSSSSNSLQSSCNST
FT                   NSTTLLVNSDCSVHASGNGNASTERNGVSFSVQNGDVCLHDLTGKQHMFSDKEDSCNGK
FT                   SRIALRRTSKRGSLHFIEQM"
FT   polyA_signal    4523..4528
FT   polyA_signal    4660..4665
XX
SQ   Sequence 4684 BP; 1329 A; 1044 C; 929 G; 1382 T; 0 other;
     gccaaccggt caccgggacc cgcagatgca cacggagttt cctccctatt tcctctgaac       60
     ttcctgtcag gatgcttttc tctggtgggc agtacagccc tgttggcaga cctgaagagg      120
     ttttactgat atacaagata ttccttgtca tcatttgttt tcatgtcatt ctcgttacat      180
     ccctgaaaga aaacggtaat tccagtttgt tatcaccatc tgctgaatca tctcttgtca      240
     gtctcatccc ctactccaat ggtacaccag atgctgcttc agaagttttg tcgactttaa      300
     acaaaacaga aaaatctaaa atcactatag taaaaacctt caatgcatca ggagtcaaat      360
     cccagagaaa tatctgcaat ttgtcatctc tttgcaatga ctcagtattt tttagaggtg      420
     agatagtgtt tcaacatgat gaagaccaca atgttaccca gaatcaagat acagctaatg      480
     gcaccttcgc tggagtcctg tctctaagtg aactgaagcg atcagagctc aacaaaactc      540
     tacagacctt aagtgagact tactttatag tgtgtgctac agcagaggcc caaagcacgg      600
     taaactgtac attcacagta aaactcaatg agaccatgaa tgtgtgtgcc atgatggtta      660
     ctttccaaac tgtacagatt cggccaatgg aacagtgctg ctgttccccg aggactccct      720
     gcccttcctc accagaagag ttagaaaaac tacagtgtga actgcaggat cccattgtct      780
     gtcttgctga tcaaccgcat ggcccaccgt tatcgtcttc cagcaagcct gttgtacctc      840
     aggccaccat tatttcccat gttgctagtg acttctcttt ggctgaaccc cttgatcatg      900
```

Figure A.3: Nucleotide sequence AF538952 in EMBL-bank.

Note the protein translation for the sequence and the presence of the raw nucleotide sequence data.

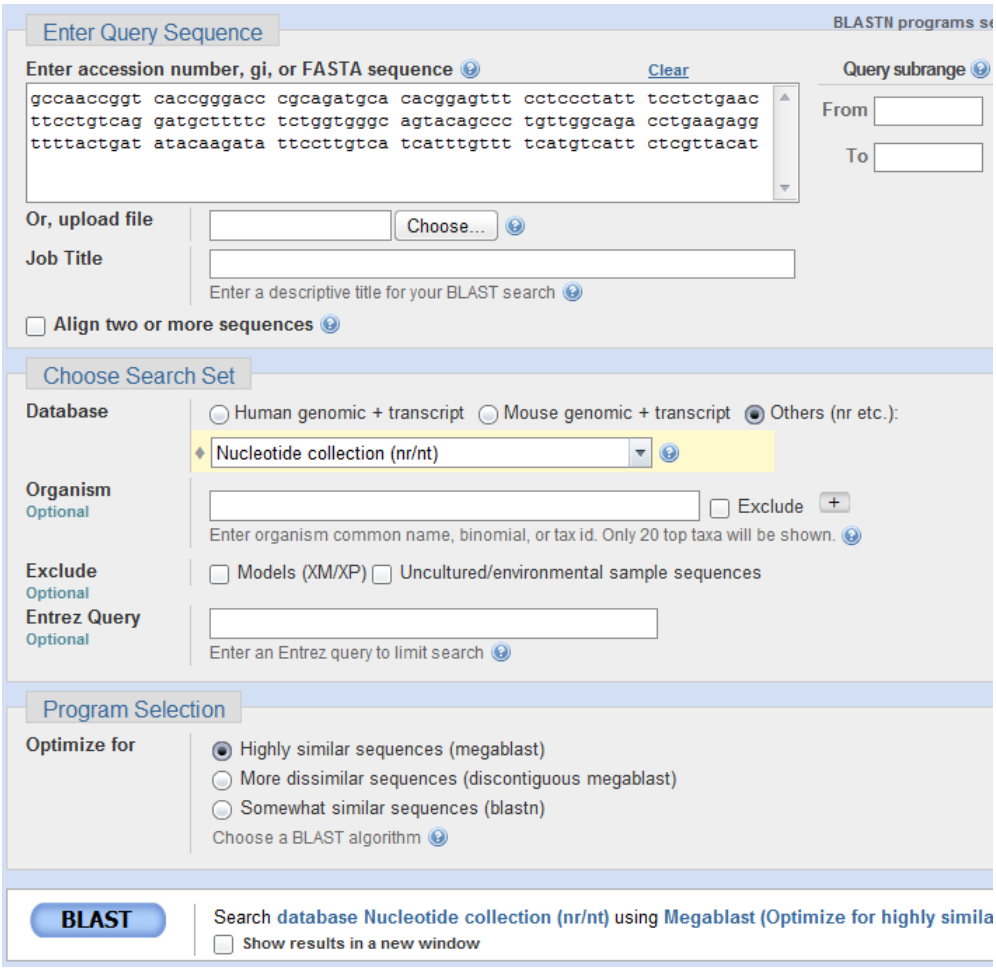
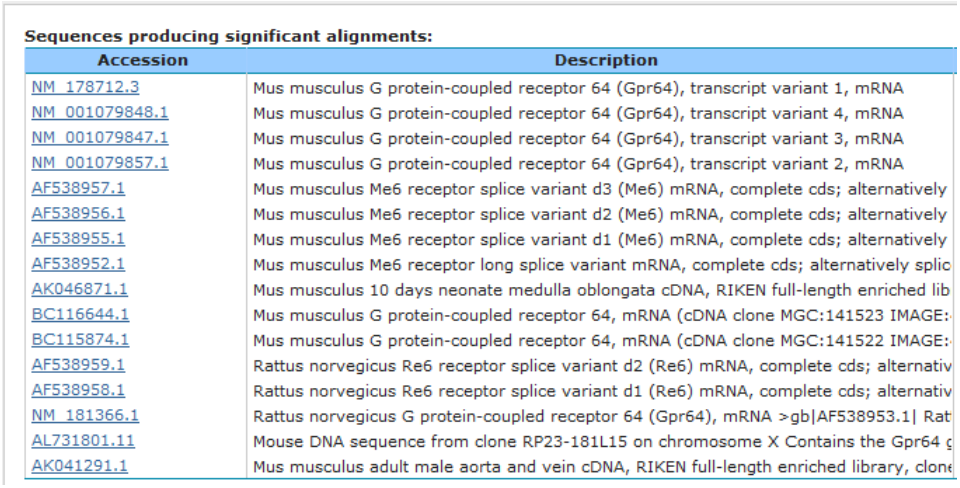


Figure A.4: Using NCBI's BLAST alignment tool to look the first portion of the AF538952 sequence.

These are the first three lines of the sequence from the previous picture; using BLAST, the idea is to find similar sequences.



Sequences producing significant alignments:

Accession	Description
NM_178712.3	Mus musculus G protein-coupled receptor 64 (Gpr64), transcript variant 1, mRNA
NM_001079848.1	Mus musculus G protein-coupled receptor 64 (Gpr64), transcript variant 4, mRNA
NM_001079847.1	Mus musculus G protein-coupled receptor 64 (Gpr64), transcript variant 3, mRNA
NM_001079857.1	Mus musculus G protein-coupled receptor 64 (Gpr64), transcript variant 2, mRNA
AF538957.1	Mus musculus Me6 receptor splice variant d3 (Me6) mRNA, complete cds; alternatively
AF538956.1	Mus musculus Me6 receptor splice variant d2 (Me6) mRNA, complete cds; alternatively
AF538955.1	Mus musculus Me6 receptor splice variant d1 (Me6) mRNA, complete cds; alternatively
AF538952.1	Mus musculus Me6 receptor long splice variant mRNA, complete cds; alternatively splic
AK046871.1	Mus musculus 10 days neonate medulla oblongata cDNA, RIKEN full-length enriched lib
BC116644.1	Mus musculus G protein-coupled receptor 64, mRNA (cDNA clone MGC:141523 IMAGE:
BC115874.1	Mus musculus G protein-coupled receptor 64, mRNA (cDNA clone MGC:141522 IMAGE:
AF538959.1	Rattus norvegicus Re6 receptor splice variant d2 (Re6) mRNA, complete cds; alternativ
AF538958.1	Rattus norvegicus Re6 receptor splice variant d1 (Re6) mRNA, complete cds; alternativ
NM_181366.1	Rattus norvegicus G protein-coupled receptor 64 (Gpr64), mRNA >gb|AF538953.1| Rat
AL731801.11	Mouse DNA sequence from clone RP23-181L15 on chromosome X Contains the Gpr64 g
AK041291.1	Mus musculus adult male aorta and vein cDNA, RIKEN full-length enriched library, clone

Figure A.5: The results of the BLAST search.

Note AF538952 and similar entries; BLAST did its job and managed to find, amongst others, the sequence we took the snippet from.

Implementation user's guide B

B.1 Introduction

GASW (short for *GPU Accelerated Smith-Waterman*) is a complete graphics processing unit based implementation of the Smith-Waterman sequence alignment algorithm. GASW can be used to search FASTA format protein databases such as Swiss-Prot with a recorded performance of up to around 21 GCUPS on NVIDIA GTX275.

Features:

- CUDA-based implementation of the Smith-Waterman sequence alignment algorithm for proteins, optimized for NVIDIA GT200-class GPUs.
- Aligns FASTA-format sequences with FASTA-format databases such as Swiss-Prot.
- Only returns alignment scores, not the actual alignments. However, top scoring sequences can be exported for alignment by tools that do offer this feature; see Section B.3.2.
- Supports FASTA-format substitution matrices and user configurable affine gap penalties.
- Lenient limits on database and query sequence lengths, see Section B.4.
- Comes with an easy to use web interface that allows the tool to be used remotely and with a graphical user interface, see Section B.5.

Requirements:

- GASW is a 32-bit Microsoft Windows application; it works fine on 64 bit operating systems. It should easily compile for Unix based operating systems as it contains no platform-dependent code; however this has not been tested.
- An NVIDIA CUDA-compatible GPU with the Shader Model 1.3 feature set is required. Effectively this means GT200-based GPUs and newer. GASW should work on GT400-series GPUs, but has not been optimized for these.

B.2 Program usage

This section explains the usage of the command-line database conversion and sequence alignment tools. Furthermore, it shows how to use a third party application to generate the actual alignments.

B.3 Converting databases to GPUDB format

Before a database can be searched using GASW, it must be converted to its own *GPUDB* format using the `dbconv` command-line program. This has to be done just once for each database. The conversion process involves separating the sequences and their descriptions and optimizing the sequence layout for GPU access. `Dbconv`'s usage is simple: its only parameter is the input database file, from which it produces `out.gpudb` (sequences) and `out.gpudb.descs` (descriptions) files. These files can be renamed; however both files for one database must have the same name and must reside in the same directory. Note that GASW comes with a small pre-converted test database named `prot_test.gpudb`, the result of converting the similarly included `prot_test.lseg` file.

B.3.0.1 Example: converting Swiss-Prot to GPUDB format

The following example shows how to convert a Swiss-Prot database to GPUDB format and how to rename it so it can be differentiated from possible other databases.

```
D:\gasw>dbconv ..\..\uniprot_sprot.fasta
Conversion parameters: block size 16, sub block size 8, alignment 256.
Loading...
..\..\uniprot_sprot.fasta: 183273162 symbols in 519348 sequence(s) in database.
Converting...
Writing out.gpudb
336 blocks
513975 sequences used to fill gaps
448 bytes of alignment padding inserted.
1822358 bytes of padding inserted.
1.033235 new vs original size ratio.
Writing out.gpudb.descs
Done.


D:\gasw>ren out.* swissprot.*

D:\gasw>dir
...
31-08-2010  16:28       191.464.944 swissprot.gpudb
31-08-2010  16:28        56.809.785 swissprot.gpudb.descs
...
```

B.3.1 GASW usage

Once a database has been converted to GPUDB format, the `gpu` command line program can be used to perform the actual search. Its usage is similar to that of FASTA's `ssearch` program:

```
gpu [options] <sequence> <database>
```

Where *sequence* is a FASTA-format single-sequence file, and *database* is a GPUDB-format database. Additionally, the program supports the following command-line options:

Switch	Required	Description	Default
-s	yes	Substitution matrix file: FASTA-format substitution matrix	
-f	no	Gap penalty	-10
-g	no	Gap extend penalty	-2
-b	no	Number of result scores to show/export	20
-o	no	Output file for result scores (see Section B.3.2)	

As noted a substitution matrix is required; GASW comes with the `blosum62.mat` Blosum62 matrix. The FASTA suite of programs offers various additional matrices in its `data` subdirectory. Once run, `gpu` will output a list of descriptions and scores for the top scoring sequences. The following example illustrates this.

B.3.1.1 Example: searching the test database

The following example shows how to search the `prot_test` database for the `mgstm1.aa` sequence. Both come with the program, as does the `blosum62` substitution matrix used.

```
D:\gasw>gpu -s blosum62.mat mgstm1.aa prot_test.gpudb
Sequence: mgstm1.aa
Database: prot_test.gpudb
Substitution matrix: blosum62.mat
Gap penalty: -10
Gap extend penalty: -2
Number of scores to show: 20
Output database file for top scoring sequences: (null)

mgstm1.aa: input sequence length is 218.
Loading database...
prot_test.gpudb: 2245 symbols in 23 sequence(s) in 1 block(s) in database.

Launching kernel.
Using 120 blocks of 64 threads: 7680 threads for 23 sequences in 1 blocks.
Processing 1 blocks per half warp.
Running...
Done. Seconds: 0.020000, GCUPS: 0.024470

Sorting results...
Results:
  0. GT8.7 | 266 | transl. of pa875.con, 19 to 675 @P:2  SCORE: 1171
  1. XURTG | 266 | glutathione transferase (EC 2.5.1.18  SCORE: 150
  2. HMIVV | 2581 | Hemagglutinin precursor - Influenza  SCORE: 38
  3. OKBO2C | 296 | Protein kinase (EC 2.7.1.37), cAMP-  SCORE: 34
  4. HAHU| 1114 | Hemoglobin alpha chain - Human, chimp  SCORE: 30
  5. RKMDS | 677 | Ribulose-bisphosphate carboxylase (E  SCORE: 27
  6. TPHUCS | 1322 | Troponin C, skeletal muscle - Huma  SCORE: 26
  7. K1HUAG | 1091 |  Ig kappa chain V-I region (Ag) -  SCORE: 25
  8. CCHU | 1 | Cytochrome c - Human @P:25-85            SCORE: 25
  9. K3HU | 1099 | Ig kappa chain C region - Human      SCORE: 24
 10. N2KF1U | 1021 | Long neurotoxin 1 - Many-banded kr  SCORE: 20
 11. FEPE | 25 | Ferredoxin - Peptostreptococcus asacch  SCORE: 19
 12. PADDING | PADDING | PADDING                         SCORE: 0
 13. PADDING | PADDING | PADDING                         SCORE: 0
 14. PADDING | PADDING | PADDING                         SCORE: 0
 15. PADDING | PADDING | PADDING                         SCORE: 0
 16. PADDING | PADDING | PADDING                         SCORE: 0
 17. PADDING | PADDING | PADDING                         SCORE: 0
 18. PADDING | PADDING | PADDING                         SCORE: 0
 19. PADDING | PADDING | PADDING                         SCORE: 0
```

The output shows the settings used; the time consumed by the alignment; and the top scoring sequences and their scores. The `PADDING` sequences are a result of the database conversion process; the original database contained just 12 sequences.

B.3.2 Performing alignment of top scoring sequences using SSearch

As described, GASW only calculates alignment scores and does not perform the actual sequence alignments. However, the top scoring database sequences can be exported to

a new database file which can then be searched by a program that does perform full alignments. An example of such a program is `ssearch`, which comes with the FASTA suite of programs. As the full alignments will only be performed for a relatively tiny amount of sequences, the overheard incurred by this somewhat redundant approach is negligible. If the command line used in the previous example is modified with the `-o` option, as such:

```
D:\gasw>gpu -s blosum62.mat -o out.lib mgstm1.aa prot_test.gpudb
```

The same output will be shown, however the listed sequences will be exported to the file `out.lib`. The number of sequences exported can be set using the `-b` option. The generated file can then be searched using `ssearch`. Care must be taken to provide the same options:

```
ssearch35sse2 -s blosum62.mat mgstm1.aa out.lib
...
The best scores are:                                      s-w bits E(12)
GT8.7 | 266 | transl. of pa875.con, 19 to 675 @P:2 ( 218) 1171 542.7 2.4e-158
XURTG | 266 | glutathione transferase (EC 2.5.1.18 ( 222)
150 71.8 1.4e-016
HAHU| 1114 | Hemoglobin alpha chain - Human, chimp ( 141)   30 17.2     2.2
HMIVV | 2581 | Hemagglutinin precursor - Influenza ( 567)   38 18.8     2.9
OKBO2C | 296 | Protein kinase (EC 2.7.1.37), cAMP- ( 350)   34 17.7     3.7
RKMDS | 677 | Ribulose-bisphosphate carboxylase (E ( 140)   27 15.8       5
CCHU | 1 | Cytochrome c - Human @P:25-85           ( 105)   25 15.3     5.2
K1HUAG | 1091 |  Ig kappa chain V-I region (Ag) -  ( 109)   25 15.3     5.5
K3HU | 1099 | Ig kappa chain C region - Human      ( 106)   24 14.8     6.5
TPHUCS | 1322 | Troponin C, skeletal muscle - Huma ( 159)   26 15.2     7.4
FEPE | 25 | Ferredoxin - Peptostreptococcus asacch (  54)   19 13.5     7.5
N2KF1U | 1021 | Long neurotoxin 1 - Many-banded kr (  74)   20 13.5     8.9
 More scores? [0]
 Display alignments also? (y/n) [n] y
```

Note how the same sequences and scores are shown. As the prompt for alignments was answered positively, the actual alignments are displayed:

```
...
                10         20         30         40         50         60
GT8.7   MPMILGYWNVRGLTHPIRMLLEYTDSSYDEKRYTMGDAPDFDRSQWLNEKFKLGLDFPNL
        ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
GT8.7   MPMILGYWNVRGLTHPIRMLLEYTDSSYDEKRYTMGDAPDFDRSQWLNEKFKLGLDFPNL
                10         20         30         40         50         60
...
```

Note that the web interface (Section B.5) automates the task of running `ssearch` on GASW output, saving command-line work and showing the alignments in a more attractive web page format.

B.4 Program limitations

Although care has been taken to be as lenient and flexible as possible, GASW is unfortunately subject to some limitations. Some are inherent to its design, some are unavoidable and some are the result of external factors. First and foremost, as it was written using

the NVIDIA CUDA programming interface, it is only compatible with certain GPUs as described in Section B.1. Furthermore, it only returns alignment scores, not the actual alignments of sequences. This problem can be circumvented by feeding the top scoring sequences into a program that does perform full alignments as discussed in Section B.3.2. Additionally, due to limited development resources, GASW only supports single-GPU operation.

Finally, Microsoft Windows Vista and later implement a *timeout detection and recovery* mechanism to recover from GPU hangs. By default, this mechanism resets the primary (desktop) GPU after two seconds are spent on a single task. As GASW only supports a single GPU, the primary one will always be used, which makes this timeout mechanism somewhat problematic. When using GASW to perform longer alignments, the GPU will be reset and the program aborted after two seconds. Fortunately, the timeout detection and recovery mechanism can be altered. By modifying the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers\TdrDelay` registry entry using `regedit`, the timeout value can be set. GASW comes with a registry file `timeout.reg`; running this file will set the timeout to 255 seconds, which should be enough for most work.

B.4.0.1 Sequence limitations

The following limits are imposed on sequence (database) lengths. Note that in some cases, multiple limits apply; which one of these is the final limiting factor will depend on the particular sequences and GPU used.

Description	Limited by	Limit
Length of query sequence	GPU memory size	A temporary data matrix of size $num\_threads^1 * 4 * query\_length$ bytes is stored in global memory.
Length of query sequence	GPU memory size	A query profile of size $23 * query\_length$ bytes is stored in global memory.
Length of query sequence	Score of alignment	The maximum score supported is 65535.
Number of database sequences	GPU memory size	A score array of size $2 * num\_database\_sequences$ bytes is stored in global memory.
Number of database sequences	32-bit integer size	Disregarding memory limits, at most 4294967296 sequences are supported.
Total database size	GPU memory size	Amount of global GPU memory left after storing other data.
Total database size	GPU memory size	Disregarding other memory limits, a database size of at most 4 gigabytes is supported.
Longest sequence in database	Score of alignment	The maximum score supported is 65535.

Table B.1: Sequence limitations

[1] *num_threads* is the amount of GPU multiprocessors times 256, for example $30 * 256 = 7680$ for GTX275.

B.5 Web interface

To cut back on repetitive command-line work, especially when interested in full alignments, and to be able to show results in a more attractive manner, GUSW comes with a web interface. This interface offers all the options of the command-line program with the additions of being able to look up sequences in Swiss-Prot and the ability to invoke GUSW remotely: users can run queries from any computer, even if it does not have a powerful GPU.

B.5.1 Setting up the web interface

The web interface is a simple set of PHP scripts. It has been tested with the Apache 2.2.11 web server but should be compatible with any server that will run PHP scripts. Note that the web interface is not compatible with Apache running as a Windows service: services do not have access to display adapters, which prevents CUDA from being used.

Installing the web interface is a matter of copying the files from the `interface` directory of the GASW distribution to somewhere in the server's web directory. The next step is to edit `config.php` and set some parameters:

- `GPU_LOCATION` should be set to the location of GASW's `gpu.exe` file. Example: `C:/GASW/gpu.exe`.
- `SSEARCH_LOCATION` should be set to the location of the FASTA `ssearch.exe` program or one of its variants. Example: `C:/FASTA/bin/ssearch35.exe`.
- `DB_LOCATION` should be set to the location where the web interface should look for GASW GPUDB database files. Of course, `gpu.exe` should be able to access this location. Example: `C:/GASW/db`
- `MATRIX_LOCATION` should be set to the location where the web interface should look for substitution matrices. It is convenient to set this to the `data` directory of a FASTA installation. Example: `C:/FASTA/data`.

B.5.2 Using the web interface

Once the web interface has been properly installed, it can be accessed using any web browser. The interface index page is shown in Figure B.1. The interface offers the following options:

- Query sequence: the query sequence to be used in the alignment, must be a file in FASTA format. The file can be anywhere on the user's system; it will be temporarily uploaded to the server hosting the web interface during the alignment.
- Database: allows the database for the alignment to be chosen. Shows a list of all databases present in the server's `DB_LOCATION` path.
- Substitution matrix: allows the substitution matrix for the alignment to be chosen. Shows a list of all matrices present in the server's `MATRIX_LOCATION` path.
- Gap penalty / gap extend penalty: Smith-Waterman alignment parameters.
- Number of scores to show: The number of top scoring sequences that will be shown.
- Scores only / Full alignments. When 'Full alignments' is selected, the top scoring sequences will be run through SSearch and the actual alignments shown.

Once the query has been submitted and run, the result page shown in Figure B.2 will be shown. The table shows the top scoring sequences and their alignment scores.

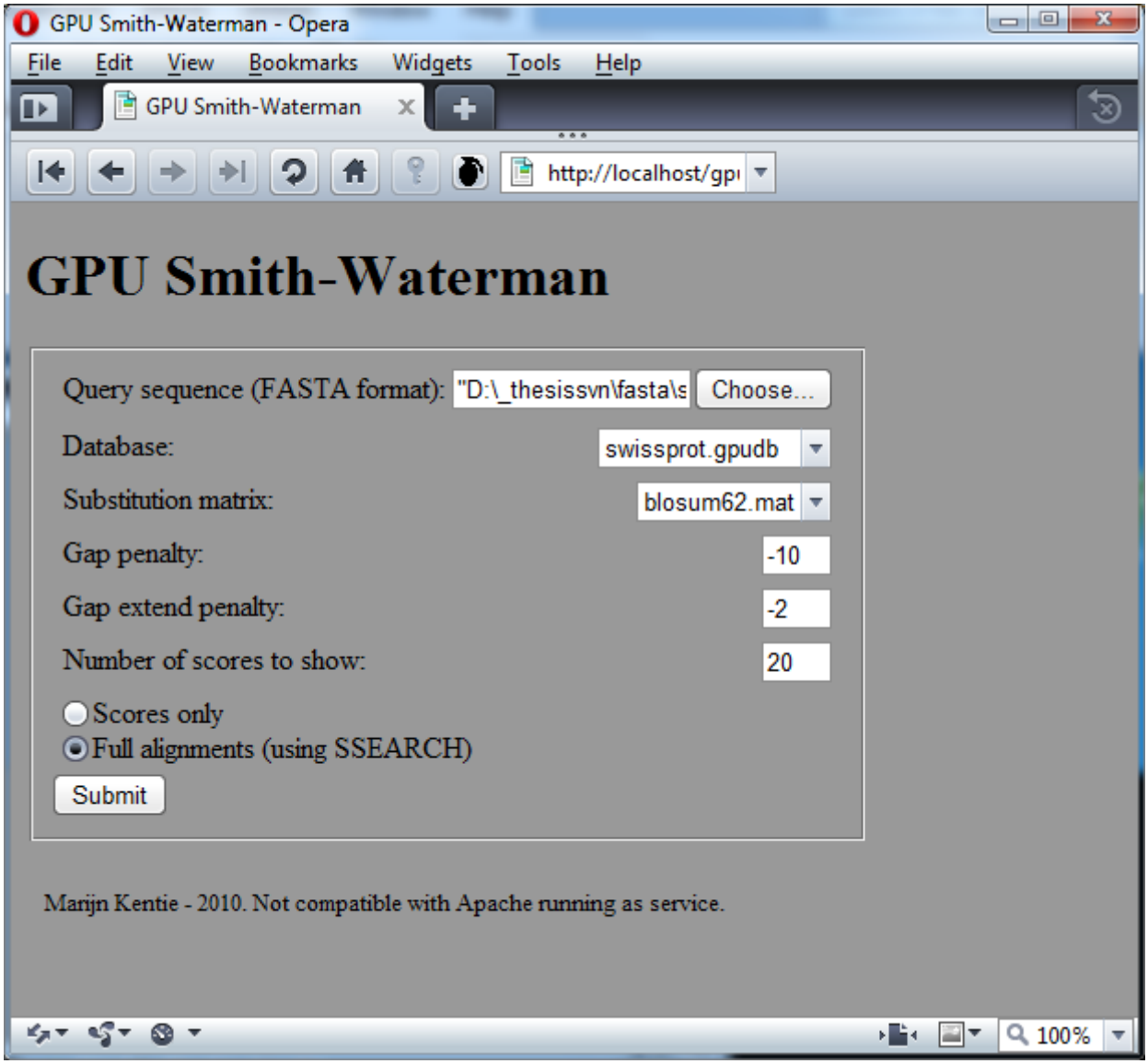


Figure B.1: GASW web interface index page.

The sequence identifiers are links that will show the Swiss-Prot page for the sequence as shown in Figure B.3. If the full alignment option was used, each table entry has an additional 'alignment' link that will show the SSearch output for the sequence, see Figure B.4.

B.6 Building GASW from source

GASW was originally developed using the NVIDIA CUDA toolkit version 3.1; the 32-bit version to be specific. As this version of the toolkit supports Microsoft Visual Studio

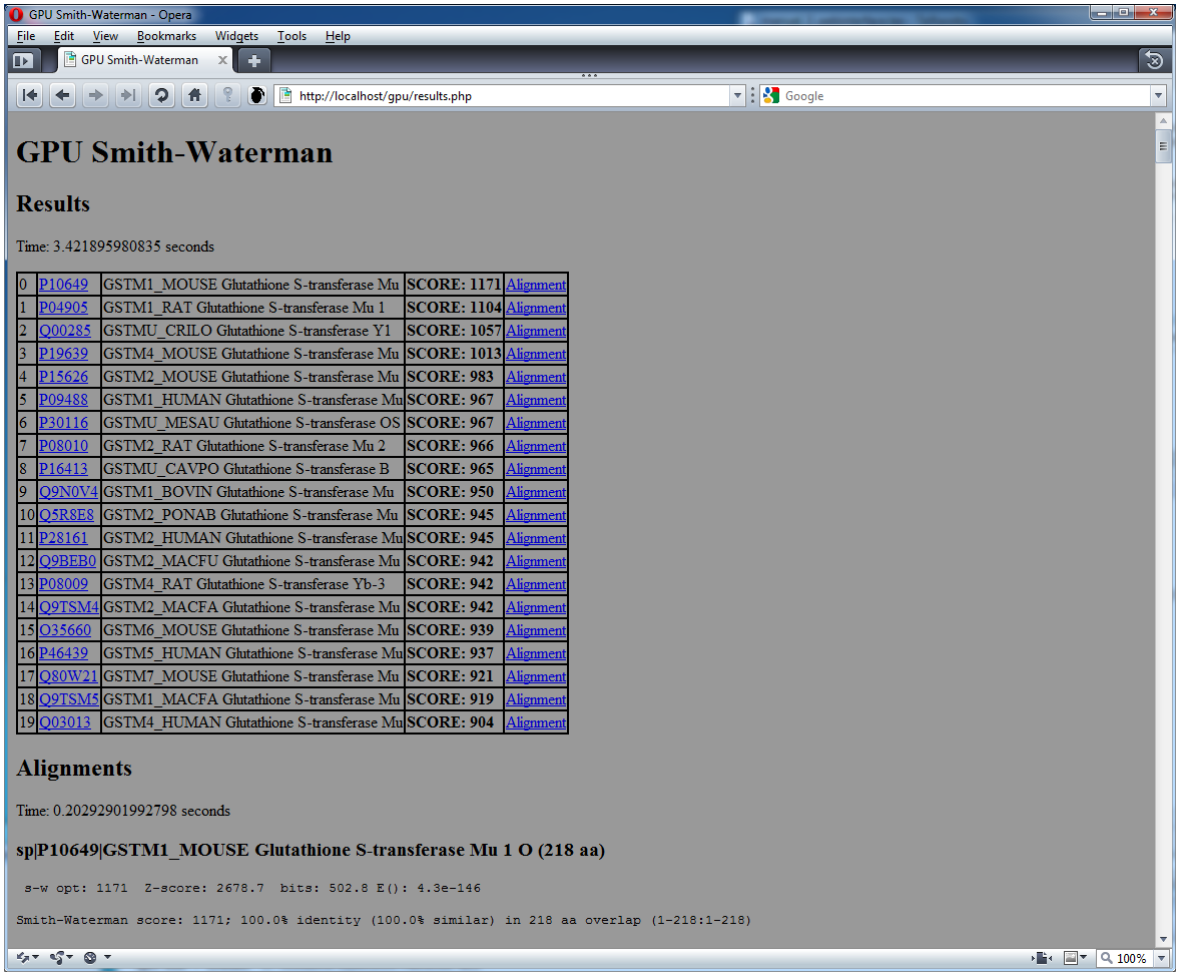


Figure B.2: GASW web interface results page

2008 at the latest, that version was used in the creation of GASW. GASW consists of a few subprojects:

- `dbconv`, the database converter that converts databases from FASTA to GASW's GPUDB format.
- `gpu`, the actual GPU accelerated Smith-Waterman implementation.
- `dbgen`, a small program to generate random FASTA databases for testing purposes.

Building `dbconv` and `dbgen` should be straightforward. To build `gpu` the CUDA toolkit must have been installed and Visual Studio must be able to find its include and library files. The `gpu` project uses the `cuda.vsprops` property sheet that has been added to the project to resolve these: the *C/C++* → *general* → *Additional Include Directories* option has been set to `$(CUDA_INC_PATH)` while the *Linker* → *Input* → *Additional Dependencies* option has been set to `$(CUDA_LIB_PATH)\cudart.lib`. These settings should work for

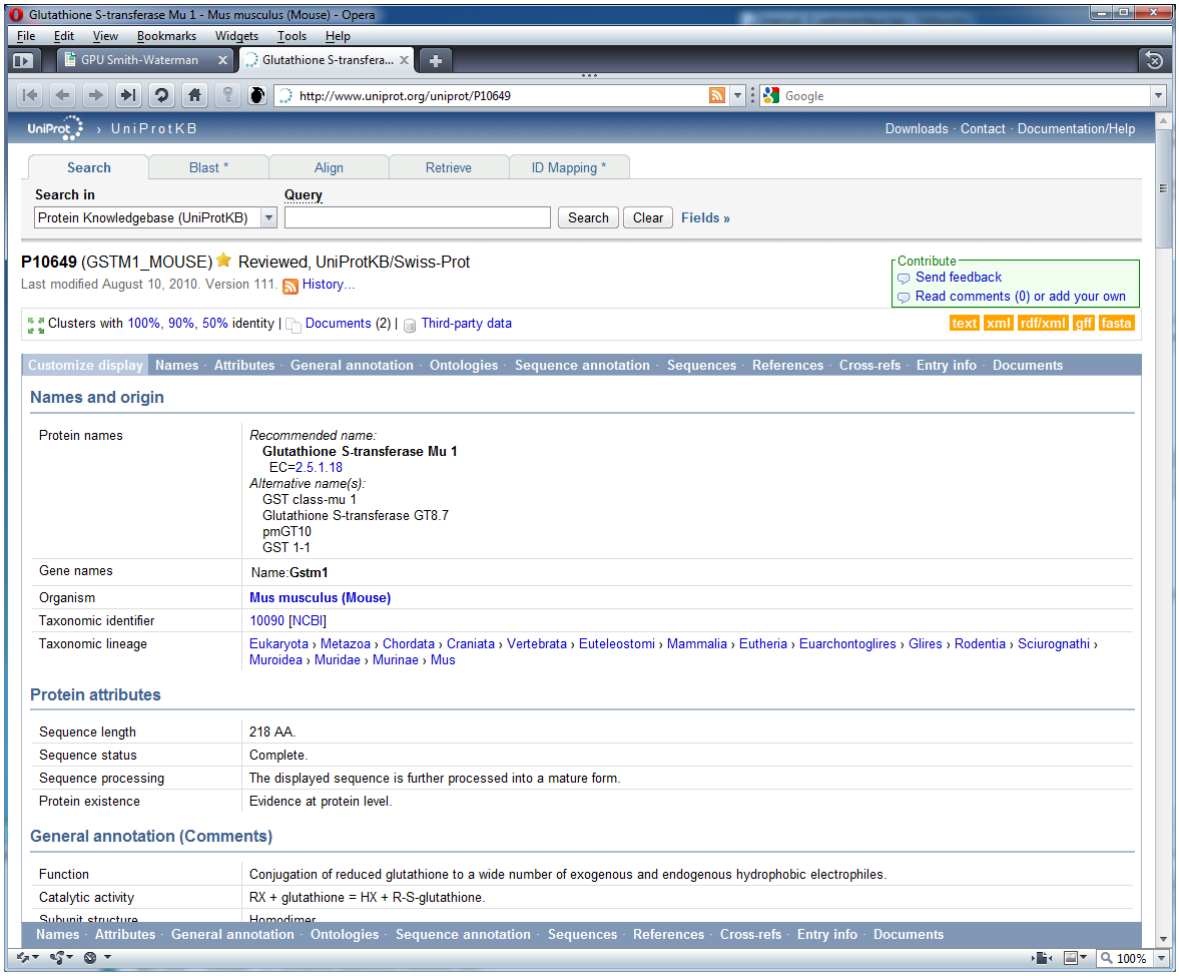


Figure B.3: The Swiss-Prot page for the top sequence.

any CUDA installation as long as the installer has set the proper environment variables. However, it is important to be aware of how the project finds these files in the event of compile errors.

The actual CUDA file in the `gpu` project, `main.cu`, must be compiled using a CUDA build rule. The CUDA 3.1 build rule comes with the source code (`cuda.rules`). If necessary, build rules can be added by right-clicking the `gpu` project in the solution explorer and selecting the *Custom Build Rules...* option. If all is well, opening the properties for `main.cu` should show the window in Figure B.5.

As GASW is optimized for GT200-class cards, the file should be built for SM_13 architectures. Furthermore, the default 32 register limit is too low for optimal performance; 64 (or less) registers should be used with the included code compiling to use 63. When more than 64 registers are used speed will suffer due to insufficient occupancy. However, it is best to set the register limit to some higher value: this way code that compiles to more than 64 registers can be modified instead of having the compiler silently

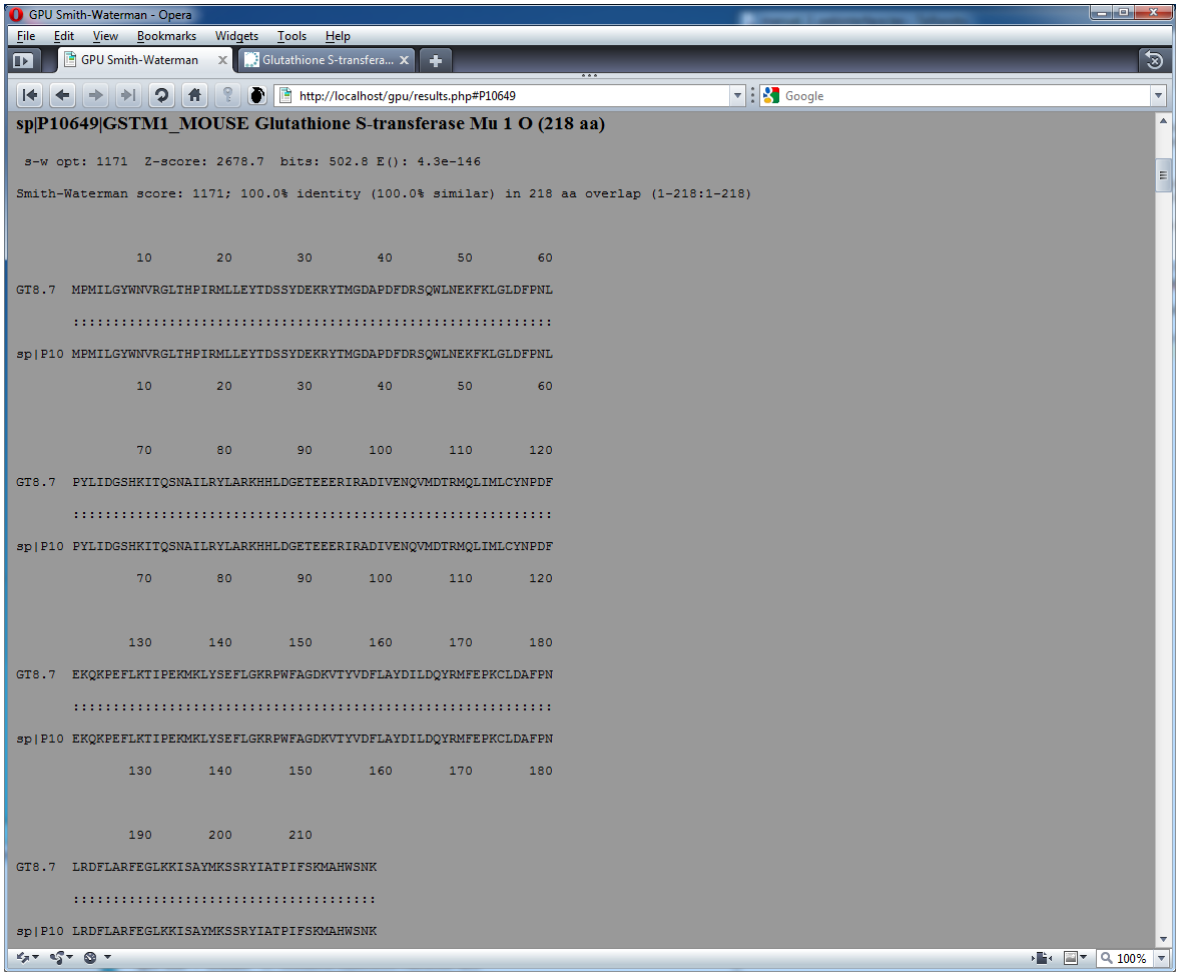


Figure B.4: The alignment for the top sequence.

spill registers to slow local memory.

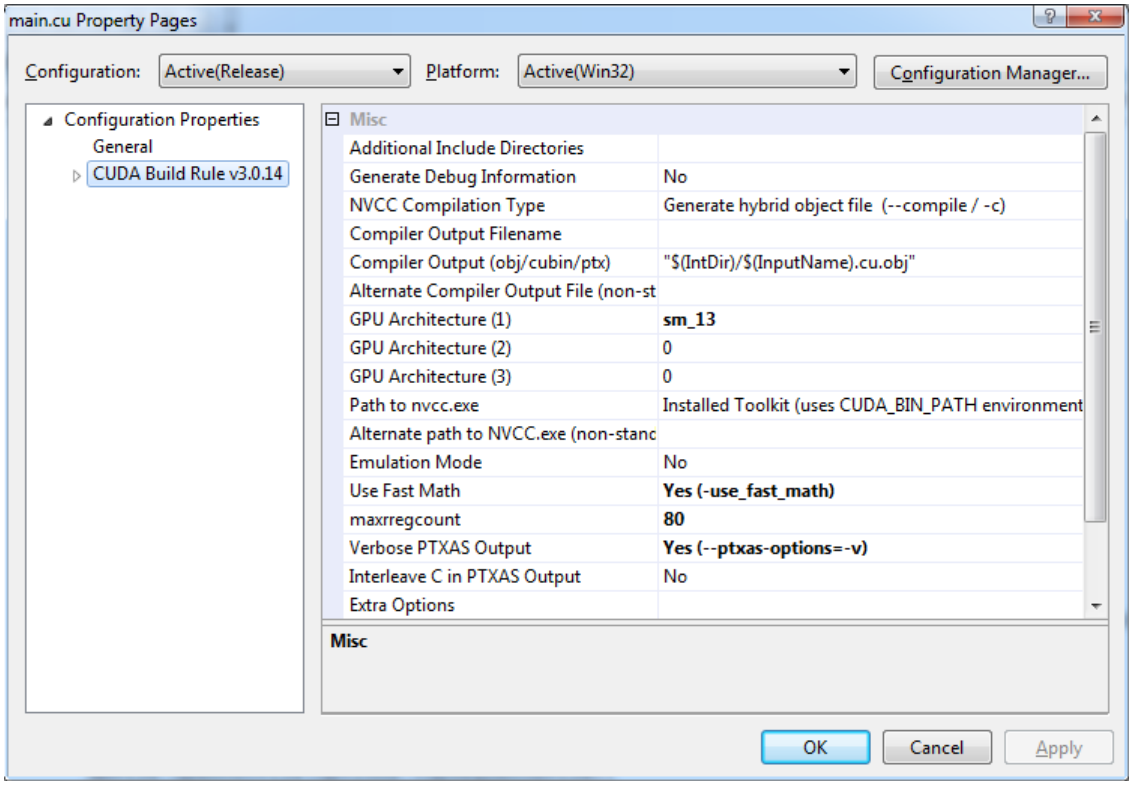
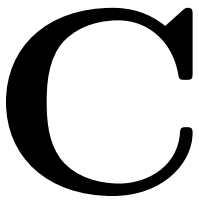


Figure B.5: CUDA build rules property sheet.

CD-ROM



The attached CD-ROM contains:

- A compiled version of GASW, the GPU accelerated protein database search tool discussed in this thesis. It includes data files needed to run test searches: a simple test database in FASTA and GPUDB format, a query sequence in FASTA format, and the BLOSUM62 substitution matrix. All can be found in the `gasw` directory.
- The source code for GASW, in the `source` directory.
- The synthetic benchmark database used during development and the October 2010 Swiss-Prot database. Both can be found in FASTA and GPUDB format in the `extra_db` directory.
- The GASW user's guide (Appendix B) in PDF format.
- This thesis in PDF format.