Diploma Thesis

Usage of GPU Based Streaming Architecture for Solving Computational Problems

Accomplished at

Centre for Graphics and Media Technology (CGMT)
Nanyang Technological University
Singapore

University of Applied Sciences
Fachhochschule Gießen-Friedberg
Section Friedberg, Germany
Departments IEM, MND, MNI
Course of Study Media Computer Sciences

Summer Term 2005

Graduand:	André Schröder
Examiner:	Prof. Dr.-Ing. Dipl.-Math. Monika Lutz Fachhochschule Gießen-Friedberg, Germany
Second Examiner:	Assoc. Prof. Dr.-Ing. Wolfgang Müller-Wittig Nanyang Technological University, Singapore

Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben kann.

Friedberg, den 1. Mai 2005 André Schröder

Table of Contents

List of Abbreviations

2D	two-dimensional
3D	three-dimensional
A	adenine
API	application programming interface
ARB	Architectural Review Board
ASCII	American Standard Code for Information Interchange
ATI	ATI Technologies Inc.
BAC	Bacterial Artificial Chromosome
BeOS	Be Inc. Operating System
BLAST	Basic Local Alignment Search Tool
BLOSUM	Blocks Substitution Matrix
bp	base pair
C	cytosine
Cg	C for graphics
CPU	central processing unit
CTT	Copy-To-Texture
CUPS	cell units per second
DNA	deoxyribonucleic acid
EMBL	European Molecular Biology Laboratory
EST	expressed cDNA sequence tag
FAI	framebuffer-attachable images
FASTA	FAST-All: fast sequence comparison tool
FBO	framebuffer objects
FFT	fast Fourier transformation
Fig	figure
FPGA	field-programmable gate array
G	guanine
Gbp	giga base pairs
GFLOPS	giga floating point operations per second
GLEW	OpenGL Extension Wrangler Library
GLSL	OpenGL Shading Language
GLUT	OpenGL Utility Toolkit
GLX	OpenGL extension for Linux
GPGPU	general-purpose computation on graphics processing units
GPU	graphics processing unit

HDRI	high dynamic range image
HGP	Human Genome Project
HLSL	High Level Shading Language
e.g.	for example (from Latin: exempli gratia)
i.e.	that is (from Latin: id est)
IUPAC	International Union of Pure and Applied Chemistry
MacOS	Macintosh operating system
MIMD	multiple instructions / multiple data
MPSS	massively parallel signature sequencing
mRNA	messenger RNA
MUL	multiplication in assembler language
NCBI	U.S. National Center for Biotechnology Information
NH3+	amino group
NV	NVIDIA
NV1	NVIDIA graphics processing unit model 1
OpenGL	Open Graphics Library: 2D and 3D graphics API
OS/2	IBM operation system
PAM	Point Accepted Mutation
PC	personal computer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PDA	Personal Digital Assistant
PDB	Protein Databank
PIR	Protein Information Resource
PROSITE	database of protein families and domains
RGBA	red, green, blue, and alpha (color components)
RNA	ribonucleic acid
RTT	Render-To-Texture
SAGE	serial analysis of gene expression
SCA	sequence comparison algorithm
SDK	software development kit
SGI	Silicon Graphics Inc.
Sh	libsh: GPGPU language
SIMD	single instruction / multiple data
SNP	single base nucleotide polymorphism
SSearch	Sequence Similarity Search
T	thymine
U	uracyl
WGL	OpenGL extension for Microsoft Windows

1 Introduction

1.1 Motivation

Until 1995 a personal computer's graphics card was only a pixel buffer preparing incoming images to be displayed on the screen. Since Central Processing Units (CPUs) were too slow to handle massive amounts of three-dimensional (3D) data as they are produced with 3D games, the game market's need for faster solutions made arise a revolutionary development of graphic adapters: the Graphics Processing Unit (GPU). In 1995 the 3Dfx Voodoo was introduced and the CPU was neither responsible for 3D transformations any longer nor for rendering of anti-aliased, textured, and shaded geometric primitives having more time to handle the logic of applications. Still driven by the game market the GPUs' performance grew faster than the microchips' development was predicted by Moore's Law (Moore 1965). The initially simple graphical auxiliary adapter arose to a ubiquitous flexible high performance device supporting computer aided design processes, scientific visualization, computer generated movies, 3D games, entertainment business and even nongraphical application development (Fernando et al. 2004, Luebke 2004, Zeller 2004b). Thus, today's graphics cards have become an indispensable coprocessor of the CPU and therefore are a central element of nowadays' PCs.

In the beginning, the GPU's functionality was limited by a fixed function rendering pipeline giving the programmers only little space for variation. In contrast to that, recent graphics architectures provide tremendous memory bandwidth and speed combined with almost fully programmable vertex and pixel processing units. Although this programmability was initially meant to serve the needs of real-time shading, it was proven that it was suitable for general purpose computation as well. The latest models even provide

32bit floating point numbers creating possibilities in the field of scientific applications that require high precision. Some examples are Fast Fourier Transform (Moreland & Angel 2003), linear algebra (Krüger & Westermann 2003), raytracer (Purcell et al. 2002, Carr et al. 2002), physical simulation (Harris et al. 2002), and cloud dynamics (Harris et al. 2003).

Commodity GPUs are a cost effective chance to equip PCs with a device that tops CPU performance. They can be an alternative to expensive special-purpose hardware. Thus, there are many areas of research where possibilities of this processing unit are closely investigated to make use of its advantages. One of these areas is the field of bioinformatics. With the genomic research's speed of producing faster sequencing methods resulting in faster data gathering, the amount of stored information about Deoxyribonucleic Acid (DNA), genes, proteins, and other elements of biological interest grew to enormous dimensions. This automatically causes a need for more effective data analysis techniques. An essential technique of those is sequence comparison. Scanning whole databases is a highly computation intensive task requiring high performance systems to complete in an acceptable time. GPUs can be a low cost alternative to specialized hardware such as a field-programmable gate array (FPGA) as it has been used by Hirschberg et al. (1996), although they are not quite reaching the same performance.

Furthermore, the programming complexity of GPUs and the required expertise are much lower compared to FPGAs. This is mainly due to the rose of high level programming languages. Nowadays, GPUs are not programmed with only assembly languages any longer but with languages that can be compared to C/C++. Hence, more programmers can start implementing their algorithms in an environment that is similar to the one they are familiar with. Since one still has to deal with computer graphics primitives like textures, triangles, and pixels, the next abstraction layer called general-purpose programming language treats the GPUs as a streaming architecture lifting the use of GPUs up to a higher usability level. With that, the programmer will be able to use the GPU without taking care of particularities related to graphics card technology, yet having to accept a loss of performance compared to lower level languages.

1.2 Purpose of this Thesis

The objective of this diploma thesis is to describe the use of GPUs for general-purpose computation. Furthermore, a computational problem from the field of bioinformatics shall be mapped to the GPU, implemented and investigated. The algorithm that is investigated in the practical part is based on the Smith-Waterman algorithm (Smith & Waterman 1981) which was enhanced by Gotoh (1982). It is used for DNA-DNA and protein-protein comparisons. While describing bioinformatics the focus will therefore lie on genetics and sequence comparison.

In chapter 2, the context that produced the Smith-Waterman algorithm is described. As it origins from bioinformatics, an overview over bioinformatics is given first, followed by subchapters that lead to the concrete cases where the algorithm is used. Therefore, genetics with focus on DNA and proteins is discussed to show the origin and properties of the sequences that are compared by the algorithm. Projects that investigated on the human genome produced huge amounts of data which are stored in databases. These databases use different algorithms for searching and data evaluation including the Smith-Waterman. Thus, the progression and features of the databases are presented followed by the basics of sequence alignment. In the context of sequence alignment several important algorithms like Basic Local Alignment Search Tool (BLAST) (Altschul et al. 1990) and the Smith-Waterman in detail are described.

The subject of chapter 3 is general-purpose programming on GPUs (GPGPU). It lays down the overall framework of the consequent implementation chapter by describing the graphics hardware architecture and introducing the terminology and concepts. The history of GPUs and general-purpose programming on GPUs introduce to this subject. The second part of this chapter refers to the programming itself by presenting CPU-GPU analogies and the methods of how to control the GPU. In this context, 3D application programming interfaces (API) as well as GPU programming languages are presented as elementary tools. Hereby the focus lies on the OpenGL API and OpenGL Shading Language (GLSL) since this combination is used in the implementation part of this thesis. The discussion of the principles of efficient computing on GPUs and shader optimization form the end of the GPGPU chapter.

Chapter 4 deals with the implementation of the Smith-Waterman algorithm on graphics hardware. First, the concept of efficiently mapping the algorithm onto GPUs is presented. This is followed by a description of the specific implementation and the application structure. Furthermore, test scenarios of the created application and their results are analysed and evaluated. Test scenarios consist of artificial random sequence tests as well as scans of the Swiss-Prot database that contains over 170.000 protein sequences to see the performance in a practical context. A comparison to values of a reference CPU implementation shows whether the GPU implementation excels the reference implementation or not. Finally, a conclusion is drawn and possible improvement suggestions are depicted.

2 Bioinformatics

2.1 Introduction

The algorithm chosen for implementing origins from the field of bioinformatics. More precisely, it origins from sequence analysis which is a part of genetics. Implementing the algorithm in this thesis goes along with a common need to develop more and more improved sequence comparison algorithms and implementations. To understand where this need comes from this chapter first describes the fundamentals of bioinformatics, followed by the history and progression of genetics in the past years. It is explained what bioinformatics is and what the major research areas and challenges are. As bioinformatics is always related to the analysis, evaluation and storing of information that occurs in the context of biology, a closer look on these areas is taken. Analysis and evaluation are those parts in which algorithms like the Smith-Waterman are used. Because the information itself is stored in huge databases, the development and properties of those databases are discussed as well. In this context, major databases, the kind of information they store, and the data formats they use are presented. One of these databases, the Swiss-Prot database, will be used for tests of the implementation. Finally, the subjects of databases and genetics are brought together in the section about sequence comparison algorithms. These algorithms search databases to find patterns in genetic sequences. The major algorithms are presented with a closer look at the Smith-Waterman. Since these algorithms including the Smith-Waterman use so-called scoring matrices for comparison evaluation, their meaning and derivation from genetics is discussed as well.

The following chapter starts with the basic introduction into bioinformatics.

2.2 Bioinformatics

Bioinformatics is also called computational biology (BCHM 2001, Wikipedia 2005, Leòn 1999). The name already implies that this scientific field is about processing information created in a biological and especially genetic context. In fact, bioinformatics uses techniques from applied mathematics, informatics, statistics, and computer science to aid research in the field of biology. It compromises all aspects of gathering, storage, organization, access, analysis, interpretation, and preparation of biological data. To guarantee efficient processing of huge amounts of data such as occur in databases in a relatively short time powerful computers and state of the art methods are necessary. Thus, bioinformatics is responsible for the development and implementation of algorithms, applications and systems to provide the required performance. The company Celera Genomics emphasizes on its website[1] that "Bioinformatics played a critical role in managing the data from the human genome, and the conversion of that raw sequence data into the assembled genome sequence". More about genomics and Celera Genomics is described in the chapters "Genomics: DNA, Amino Acids and Proteins" and "Genetics: Sequencing".

The major research areas of bioinformatics are
- Sequence Analysis
- Expression Analysis
- Protein Structure Prediction
- Modelling of Biological Systems.

Sequence analysis mainly tries to find similarities between different DNA or protein sequences which involves the usage of algorithms like the Smith-Waterman method. As genes or proteins with similar sequences (homologous genes) are likely to have similar functions, it is a common method to look for those in a pool of well known sequences which are homologous to the yet unknown one. This can speed up the research process of new proteins and genes. The comparison is mostly performed between different species but also within a species. Another range of application is the matching of DNA sequence fragments. A technique called shutgun DNA sequencing produces thousands of small DNA fragments that are scanned and gathered. As they overlap, sequence analysis can find the right overlapping partners to build up a complete sequence (Istrail et al. 2003). A closer look on sequence analysis is taken in chapter "Sequence Alignment".

1 http://www.celera.com/celera/discovery_platforms

The expression of the genetic information stored in DNA involves transcription of DNA into messenger RNA and translation of linear mRNA nucleotide sequences into sequences of amino acids in proteins (Shamir 2002, Wikipedia 2005, Wikipedia 2005b). The flow is: DNA→ RNA → Protein. In other words a gene expression is the process by which a gene's information is converted into proteins that form the structures and functions of a cell . The task of expression analysis is to measure the amount of mRNA in the cell i.e. the expression level of each gene. This can be performed with different techniques such as microarrays, expressed cDNA sequence tag (EST) sequencing, serial analysis of gene expression (SAGE), tag sequencing, massively parallel signature sequencing (MPSS), or by measuring protein concentrations with high-throughput mass spectroscopy.

A protein has several physical structural levels: the primary, secondary, tertiary and quaternary structure (see chapter "Proteins and Sequences"). The prediction of the structure of a protein by analysing its amino acid sequence is a field of bioinformatics where a lot of problems still have to be solved to get reliable results. Like homologous sequences can be used to predict the functions of a gene, they can also be used to predict the structure. This is called homology modelling and the only way to get reliable results so far.

The modelling of biological systems deals with their observation and simulation to understand the biological processes running in them. Therefore relationships and interactions between different parts of biological systems are studied. Systems like cells, organels and organisms are of interest as well as subsystems like gene regulatory networks and signal transduction pathways. This is like a process of reverse engineering by analysing real life with the purpose of creating artificial life in simulations. Apart from that simulations are used to understand evolutionary processes.

This was a rough overview over bioinformatics. The further chapters will deal only with sequence analysis where the Smith-Waterman algorithm origins.

2.3 Genetics: DNA, Amino Acids and Proteins

The Smith-Waterman algorithm is used for sequence comparison and alignment which is a part of sequence analysis. To understand sequence analysis it is important to understand where the sequences that are dealt with come from. This is most important for the creation of scoring matrices which are discussed later. This chapter describes the origin of the sequences that are analyzed with sequence alignment algorithms.

The construction information and hereditary material for almost every lifeform is encoded in its genome (LHNCfBC 2005, Shamir 2002, HGMIS 2003). The genome is the full set of an organism's DNA sequences. DNA is a sequence of four elementary chemical bases arranged in a double helix: adenine (A), cytosine (C), guanine (G), and thymine (T). Each is attached to a sugar (deoxy-ribose type) and a phosphate. This structural unit is called a nucleotide. They occur as nucleic base pairs whereat adenine always pairs with thymine and cytosine with guanine (see figure 1).

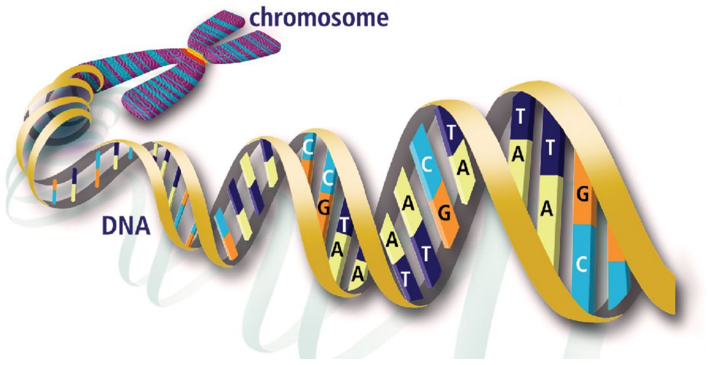


Fig. 1 - Artistical illustration of a chromosome consisting of DNA strands with the shape of a double helix (image courtesy of U.S. Department of Energy's Joint Genome Institute, Walnut Creek, CA, http://www.jgi.doe.gov).

The order of the bases can be seen as an instruction code to build the elements an organism consists of. The human genome consists of 3 billion nucleic base pairs of which 99,9 % are identical in every person. In contrast, the genome of a bacterium consists of only several hundred thousand base pairs. As DNA only occurs in a double helix with base pairs, half of the information is redundant. Only one strand is necessary to get the full information. The sugars have an asymmetrical structure and therefore the strand has a reading direction. They are connected with their fifth carbon group upstream and with their third downstream. Hence, the direction goes from 5' (five prime) to 3' (three prime). Because each strand has the reverse direction of his partner strand, a base sequence can be written down as the following example:

```
5' GTTAGTTTCC 3' and
3' CAATCAAAGG 5'
```

In the human genome DNA is organized in 23 distinct chromosomes which consist of up to 250 million base pairs (HGMIS 2003, LHNCfBC 2005, Alberts et al. 1994). Chromosomes contain up to 3,000 genes. Genes, specific DNA sequences, act as instruction sets for the production of proteins and thus are the basic physical and functional units of heredity. According to research results of the Human Genome Project, humans have about 30,000 genes. The fact that an insect like the fruit fly (Drosophila melanogaster) already has 13,000 genes shows how complex genetics even with quite primitive lifeforms is. In relation to the complete genome, the encoding parts (euchromatin) for protein synthesis instruction, i.e. those parts of the DNA that contain genes, make up only 2 %. The remaining non-coding part is so called junk-DNA.

With the expression of genes, DNA sequences are translated into amino acid sequences that form proteins (Alberts et al. 1994). This procedure is divided into two major steps: transcription and translation (see figure 2).

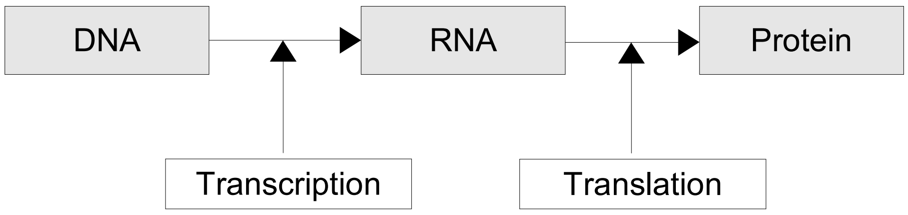


Fig. 2 - From DNA to Protein (The central dogma)

First, the enzyme RNA polymerase creates a complementary copy of a specific part of the DNA: the ribonucleic acid (RNA). The following translation of the RNA's nucleic sequence into the sequence of amino acids is based on the so-called genetic code (see appendix “The Genetic Code”). Each of the 20 amino acids is encoded by one triplet of 3 nucleotides in series. The amino acids are brought together to form a protein.

Amino acids have a simple structure: each consists of a carbon atom (α carbon), a carboxyl group (COO^-), an amino group (NH_3^+), a hydrogen atom and a specific side chain commonly referred to as residue. The chemical properties of an amino acid are determined by its residue (see appendix “The Structure of Amino Acids” and “The Amino Acids”). As well as DNA amino acid chains have a reading direction. Thus, if written down with one-letter-code, the sequence ends are marked with C (carboxyl group) and N (amino group):

```
N PKRGACMLTNQFKRKSACQ C
```

Proteins are polymers consisting of up to several thousand amino acids joined by peptide bonds (Cooper 2000, HGMIS 2003, Wikipedia 2005c, LHNCfBC 2005). While the genetic information lies in the DNA's and RNA's nucleic acids, the structural and life functions encoded in those sequences are performed by proteins. They make up the majority of cellular structures and are involved in most life functions. The function they have depends on their constellation of amino acids. Because of this importance, they got their name “protein” (Greek: πρωτεϊνη) derived from proteios (of first rank), first mentioned by Jöns Jacob Berzelius in 1838. The sum of all proteins in a cell is called a cell's proteome and, in contrast to the genome, varies continuously depending on their production. The field of research related to proteins is called proteomics.

The unique structure of each protein is commonly divided into four levels (Cooper 2000, Rupp 2000):

- The primary structure: the amino acid sequence.
- The secondary structure: mainly α-helix and β-sheet, regular arrangements of the amino acid subsequences formed by hydrogen bonds.
- The tertiary structure: the overall shape of a protein formed by the sum of secondary structures connected by looped regions.
- The quaternary structure: interactions between different polypeptide chains in proteins composed of more than one polypeptide, e.g. hemoglobin.

The higher level structures are formed by a process called folding and are basically a consequence of the primary structure (see figure 3 and appendix “Protein Structure”).

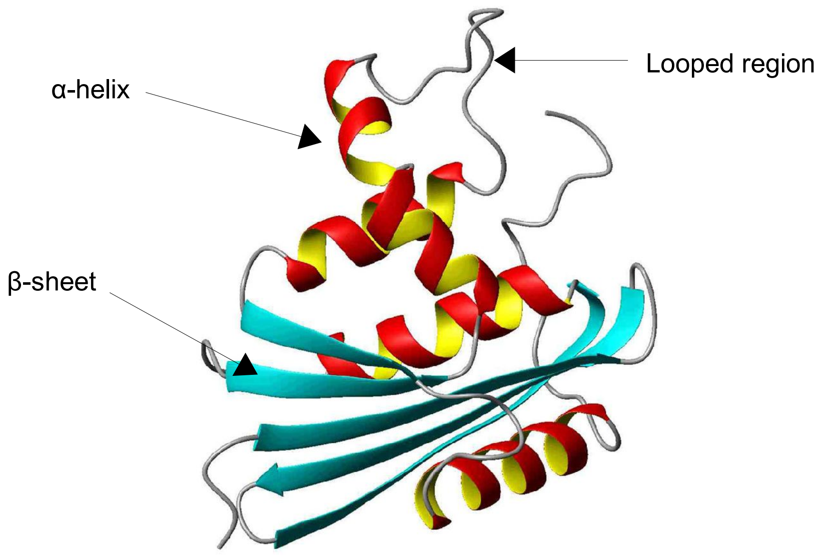


Fig. 3 - 3D model of the glycoprotein Ribonuclease H – tertiary structure (image courtesy of E. Meiering, Waterloo University, http://sciborg.uwaterloo.ca/~meiering/research.html).

Sequence alignment and comparison are the computational part of sequence analysis. The practical part consists of reading the sequence elements, nucleotides if DNA is scanned and amino acids in the case of proteins. The whole process of preparing and reading the sequences is called sequencing and presented in the following chapter.

2.4 Genetics: Sequencing

This chapter explains how the sequence information that is processed with sequence comparison algorithms is gathered by sequencing. Moreover, it is shown how genetic databases grew to enormous dimensions in the last decade, caused by genome projects. Thereby a need for more advanced algorithms and analysis systems was created.

Sequencing is the “determination of the order of nucleotides (base sequences) in a DNA or RNA molecule or the order of amino acids in a protein” (HGMIS 2003b, Glossary).

The most commonly used sequencing method is the Sanger method developed by Fredrick Sanger and co-workers (Sanger et al. 1977). With this method small DNA fragments with a length of up to 1000 (usually 500 to 800) base pairs (bp) can be created by a controlled interruption of enzymatic replication. DNA polymerase copies sequences of single stranded DNA and stops the process when it reaches a dideoxyribonucleotide. The latter is randomly incorporated into sequences, thus resulting in many different sequence lengths. The fragments, which are all labled with a flurophore, are separated by chromatography and detected with laser and a photometer in a high-throughput machine.

The Sanger method was also used by the publically funded Human Genome Project (NHGRI 2004). A primary goal of this project was to sequence the whole human genome with its about 3 billion base pairs. Before the sequencing process started, the whole human genome was cloned using bacteria, the so-called BAC-based method (Bacterial Artificial Chromosome). First, the DNA is split into pieces of about 200,000 bp. The position of each resulting sequence in the human genome is stored (mapped) to ensure that it is pursuable where the information comes from. Then all the sequences are cloned using bacteria. Each fragment is inserted into one bacterium e.g. Escherichia coli (E.coli). The bacteria are multiplied in liquid broth and then diluted so that single colonies can be separated on agar plates. Each colony represents one clone containing one specific DNA fragment, which then can be purified from the bacteria and used for sequencing. The sum of different Bacterial Artificial Chromosome (BAC) clones that contains a whole genome is called a BAC library. Finally, the Sanger method was used for sequencing.

Competing with the non-profitable Human Genome Project (HGP), there also existed a commercial attempt to sequence the whole human genome executed by the private company Celera, finishing the genome almost at the same time, but much faster. They used the Sanger method as well, but in contrast to the directed method of the HGP they had a random approach, called shotgun sequencing (Venter et al. 2003, Golding & Morton 2004) which was not invented until some years after the HGP had started. Randomly chosen pieces of the entire genome are sequenced in this method. Thus, a mapping process does not occur beforehand. This method requires more work afterwards, but on the other hand many steps of this process can be automated. Therefore, shotgun sequencing is faster than the BAC-based approach whereas the latter is more accurate.

At this point the practical part ends and the information analysis starts.

The sequencing part of both methods results in a huge amount of fragmented information about the entire genome, stored in databases. The BAC clone fragments are existent in groups whereas the shotgun fragments are totally unrelated. With the assembly process all fragments are put together to form the whole DNA sequence. As the fragments overlap it is a matter of time and good matching algorithms to bring them in correct order whereby the permutation of BAC clone fragments is much less due to the grouping. For this matching fast sequence alignment algorithms are used like they are presented in the chapter “Sequence Alignment”. Filling the gaps and finding repetitive areas is that part of the assembly process that demands most patience and time. In reality, with today's techniques not all gaps are solved and 1 % of the euchromatin remains unrecoverable.

Robert W. Holley (Holley et al. 1965) and his team were the first to determine the nucleic sequence of an RNA molecule in 1965. It took them about one year to determine all of only 77 nucleotides that alanine transfer RNA of yeast consists of. Since then, technological improvements and automation have increased speed and lowered costs enormously. Figure 4 shows the cost and quantity progression from 1996 till 2004. Whereas the cost per base fell extremely, the amount of annually produced sequence bases raised reciprocally.

How well technology evolved since then was shown by the Human Genome Project (HGP) and Celera Genomics by sequencing the whole human genome. The former started in 1990 with a 3 billion dollar budget and the intention to complete within 15 years. The mapping process was finished in 1994 and in 2001 the initial working draft of the complete sequence was published. This version still contained several hundred thousand gaps and it took until 2003 to finish 95 % of the gene-containing part with an accuracy of 99.99 %. Celera Genomics was founded in 1998 by Craig Venter with the intention to sequence the human genome within only 3 years. On 17th of June 2000, the

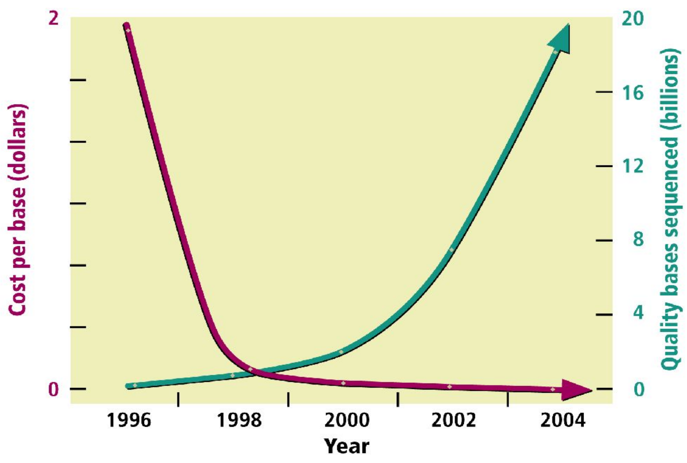


Fig. 4 - Cost and Quantity Progression from 1996 till 2004 (Image courtesy of The Human Genome Program of the U.S. Department of Energy Office of Science, http://www.ornl.gov/sci/techresources/Human_Genome/research/instrumentation.shtml).

scientists of this company finished sequencing the 2.91-billion base pairs of the euchromatic portion of the human genome (Venter et al. 2001). In fact, they produced it with 5.11-fold coverage (14.8-billion bp in total) by shotgun sequencing within 9 months. For assembling they combined both the Celera data with the HGP data and announced the completion of its first draft of the human sequence on 25th of June 2000. The assembly of a total of 2.586 Gbp of sequence took about 20,000 CPU hours with a quite powerful compute infrastructure. This obviously shows how important computational support for biological and especially genetic sciences can be. This part can be accelerated by using faster algorithms, faster architecture, or both. Mapping that uses algorithms on fast parallel architectures, graphics hardware in this case, is an approach that is done within this thesis.

Research does not stop after assembling the sequences. In fact, the most interesting work comes on further research (Golding & Morton 2004, Shamir 2002). The next challenging step is to find the location of all genes within each chromosome. There are some indicators that allow a vague prediction of the presence of a gene, but still these areas have to be examined carefully. For other genes there are no helpful hints at all. Hints are start and stop codons and the similarity of a sequence part to an already discovered gene. Once a gene is discovered, it is possible to predict proteins, but also the opposite direction is possible. With mass spectrometry methods proteins can be sequenced in order to use this information for predicting gene patterns. All in all it is to explore how DNA and proteins work with each other and how they affect the living systems they are included in.

All data about DNA sequences, genes, and amino acid sequences is stored in databases. In case of ongoing and commercial research it is most likely that the information is kept privately, but especially in cases of non-profit and publically funded research it is stored in open databases, accessible for everybody. Independent of the purpose of the database, it must be structured in such a way that information can be found and used effectively. In genetics research, as mentioned previously, common tasks are to compare one sequence to a set of other sequences in order to find similarities or, on the opposite, to search a long sequence like a genome for parts that contain certain short sequences. One example for the former is the comparison of one protein with a database of proteins in the context of function prediction. The latter occurs in the investigation of a genome to find regions that resemble known genes for gene prediction purposes.

Both, databases and sequence comparison algorithms are presented in the following two sections, since the Smith-Waterman algorithms is one of these. Moreover, one of these databases will be used to test the scan performance of the implemented algorithm. Therefore, the database entry formats will be presented as well, because every sequence alignment tool has to deal with them if it loads sequences.

2.5 Databases

Swiss-Prot, the database used for testing the Smith-Waterman GPU implementation, is only one of a huge number of genetic databases. This chapter will give a short overview over the major databases and presents the database entry format FASTA that is loaded by the Smith-Waterman implementation.

Each database has a focus on a specific type of information. There are databases focusing on genes, proteins, complete sequences, sequence fragments, certain species like mouse, fly, yeast or bacteria, and many more criteria. Darryl Leòn (Leòn 1999) categorizes databases into the following main groups:

Genomic Information

- General Databases
- Human Genome Databases
- Non-Human Genome Databases

Proteomic Information

- Protein Databases
- Motif Databases

Other Databases

- SNPs Databases (SNP: Single base Nucleotide Polymorphisms)
- Expression Databases
- Pathway Databases

For each group he gives a representative list of the most important ones with their description. The whole list can be found on his website[2]. According to Golding and Morton (2004) the major nucleotide (DNA, RNA, ...) databases are

– EMBL	*European Molecular Biology Laboratory*
– NCBI	*U.S. National Center for Biotechnology Information*
– DDBJ	*DNA Data Bank of Japan*.

Referring to the previous subchapter, DNA sequence information generated by the Human Genome Project is freely accessible to scientists through the online GenBank / NCBI[3], a database run by the National Institutes of Health and the National Library of Medicine's National Center for Biotechnology Information. Furthermore, as the major protein databases the following are mentioned:

– Swiss-Prot	
– PIR	*Protein Information Resource*
– PDB	*Protein Databank*
– PROSITE	

Web address of the mentioned databases can be found in the chapter “Links”. In addition to the sequences, most databases contain other useful data as well, including organism, tissue, function, and bibliographic information. The amount of additional included information depends on the format of the database entries. A representative of a minimum of information for entries is the FASTA format which contains only one description line and the sequence in one-letter-code. The following entry is written in FASTA format:

2 http://home.san.rr.com/dna/darryl/home.html
3 http://ncbi.nih.gov/Genbank/index.html

```
>sp|P53765|UNG_EHV2 Uracil-DNA glycosylase (EC 3.2.2.-) (UDG) -
Equine herpesvirus 2 (strain 86/87) (EHV-2).
MERWLQLHVWSKDQQDQDQEHLLDEKIPINRAWMDFLQMSPFLKRKLVTLLETVAKLRTSTVVYPGEE
RVFSWSWLCEPTQVKVIILGQDPYHGGQATGLAFSVSKTDPVPPSLRNIFLEVSACDSQFAVPLHGCL
NNWARQGVLLLNTILTVEKGKPGSHSDLGWIWFTNYIISCLSNELDHCVFMLW-
GSKAIEKASLINTNKHLVLKSQHPSPLAARSNRPSLWPKFLGCGHFKQANEYLELHGKCPVDWNLD
```

The description line is marked with a “>” character. It contains information like the database entry number, the sequence name abbreviation, and the detailed sequence name. A line break terminates the description. The following lines of characters form an amino acid sequence in one-letter-code. For sequence comparison, only the sequence itself is used, but it should always be pursuable where the sequence comes from. Otherwise the result would be useless.

This example entry was taken from Swiss-Prot. The same format will be used in the implementation part of this thesis and, furthermore, it is also Swiss-Prot that will be used for performance tests. The original sequence entry in UniProt format has much more detail and complexity. It can be found in the appendix section “Swiss-Prot Entry Example”. The UniProt format itself is defined on the Expasy website[4], the FASTA format is described by Pearson & Lipman (1988). The most frequently mentioned formats are EMBL, FASTA, GCG, GenBank, UniProt, IUPAC, and the trivial plain format which consists of only the sequence without any additional information.

The study of genomics has caused a true explosion of databases. Figure 5 shows the growth of the EMBL database with nucleic sequences from 1982 till 25th of march 2005. Figure 6 is from the same date and presents the growth of protein entries of Swiss-Prot. These diagrams again show, how fast the amount of data grows, emphasizing the need for fast solutions that are able to handle such huge amounts of data in a short time.

The actual algorithmic solutions for database scans are shown in the following chapter, also introducing the Smith-Waterman.

4 http://au.expasy.org/sprot/userman.html#entrystruc

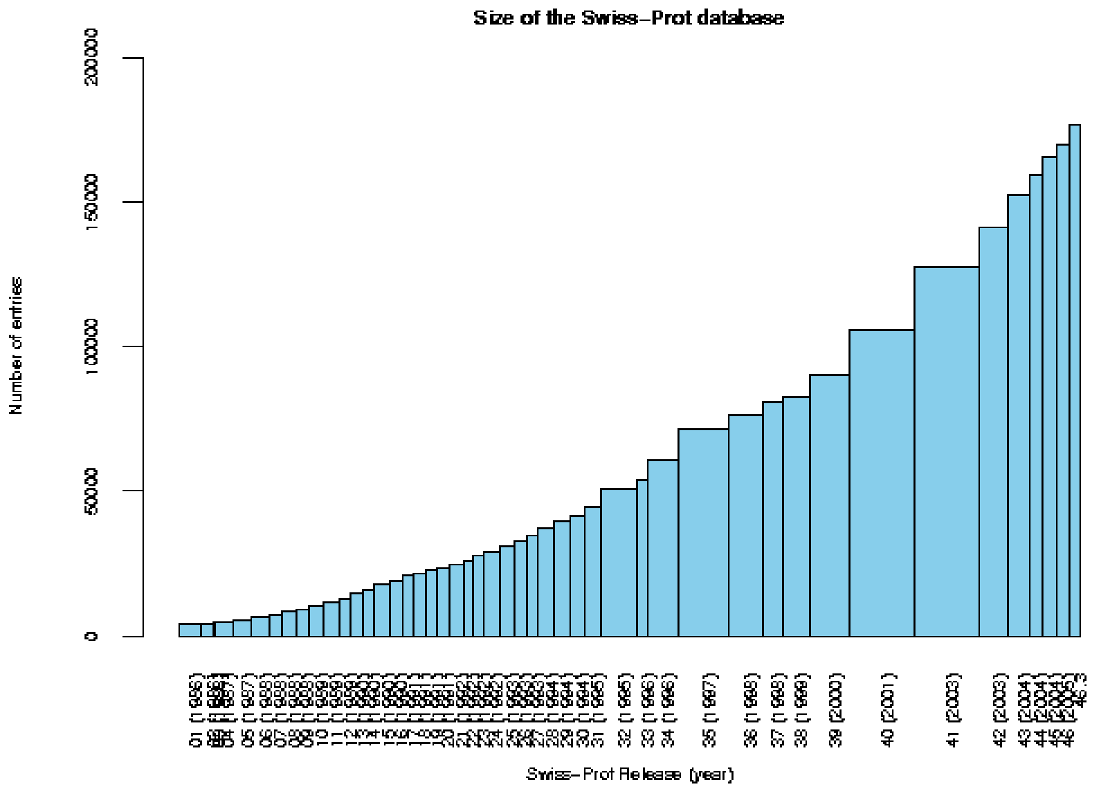


Fig. 5 - Swiss-Prot database growth. 176,469 entries (protein sequences) in release 46.3 of 15 march 2005. (http://au.expasy.org/sprot/relnotes/relstat.html, as of 25.03.2005)

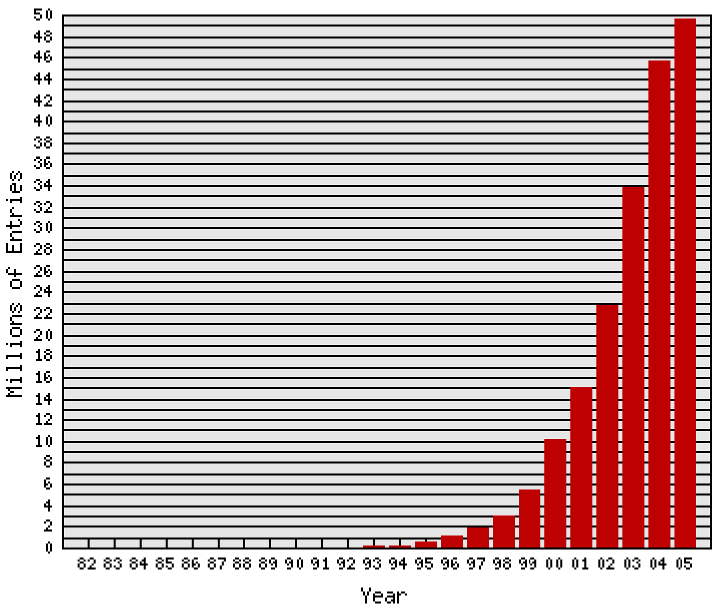


Fig. 6 - EMBL database growth. 49,654,087 entries (nucleic sequences) on 25th of march 2005, http://www3.ebi.ac.uk/Services/DBStats/, as of 25.03.2005.

2.6 Sequence Alignment

Introduction

The Smith-Waterman method is one of several commonly used algorithms for sequence alignment in a genetic context. This chapter presents it in the context of different types of algorithms. Furthermore, the classical Smith-Waterman local alignment with its enhancement by Gotoh is presented in detail as a preliminary step for its mapping onto parallel architecture in chapter 4. Furthermore, scoring matrices are introduced, since they are used by all of these algorithms.

Because genetic elements share common sequences, it is possible to use their similarities to derive the function or structure of an unknown sequence from a known one. With this regularity mathematical algorithms can be applied to analyze sequences. Sequence analysis therefore mainly tries to find similarities between different sequences by aligning them.

Some algorithms stretch sequences by inserting gaps in order to get the best alignment. With the study of evolution of sequences mismatches correspond to mutations, and gaps indicate insertions of deletions of sequence elements. A gap is a segment in a sequence consisting only of empty elements. In the following example two DNA sequences S_1 and S_2 are aligned resulting in S_1' and S_2' whereby gaps are indicated by “-” and matching elements are marked with “|”:

```
S1': tcctctgcctctgccatcat---caaccccaaagt
     |||| ||| ||||| |||||   ||||||||||||
S2': tcctgtgcatctgcaatcatgggcaaccccaaagt
```

The highest similarity corresponds to the lowest sum of distances between corresponding elements (Tompa 2000). For two sequences with equal length l and $\delta(x, y)$ being the distance function of two elements it is tried to minimize

$$\sum_{i=1}^{l} \delta(S_1'[i], S_2'[i]) \quad .$$

Alignment is commonly categorized into the following types (Jovanovic 2003, IISR 2003, Wikipedia 2005d):

Pairwise Alignment

The comparison of two sequences is called pairwise alignment. An example for an use case is a database search for homologous sequences for function or structure prediction of genes or proteins. The search of a database in this case is a series of pairwise alignments. Another case is the analysis of sequence evolution by mutation.

Multiple Alignment

Determining the similarity among more than two sequences is called multiple alignment. It is also used for structure or function prediction of proteins. This approach does not look for the best similarity to a concrete protein, though, but to a protein family. Within a family proteins share specific features. In some databases proteins are grouped by function, structure or evolutionary history. With the multiple alignment among a protein family a representative protein can be generated, called a motif. Afterwards, this motif can be pairwise compared to a new protein in order to find that group of proteins which it is most similar to. Thus, weak sequence similarity can be helpful in the process of predicting a protein's structure, function and origin. This technique is used in the previously mentioned motif databases.

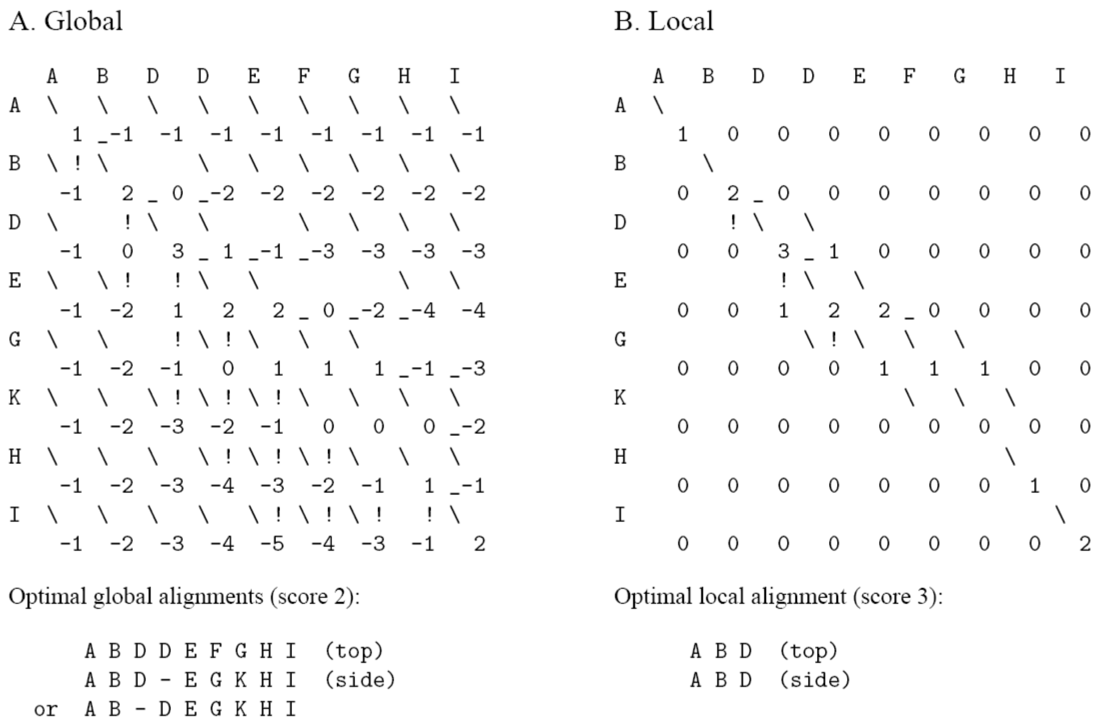
Global Alignment

With global alignment an attempt is made to get the best alignment of entire sequences whereby the number of matches is maximized and the number of gaps is minimized. For this kind of alignment the examined sequences should have similar lengths. It is therefore suitable for finding closely related sequences as needed in evolutionary investigations. The previous example is globally aligned. Because many tasks of global alignment can be done by local alignment, this technique is regarded deprecated.

Local Alignment

Local alignment tries to find only segments of sequences that show the highest similarity. Due to this fact it is, in contrast to the global approach, useful for the comparison of sequences with very different lengths such as searching a genome for the occurrence of a certain pattern. This method can find related regions in a different order in two sequences. Thus, another application is the matching process of overlapping DNA fragments after shotgun sequencing.

The difference between local and global alignment is shown in figure 7. The alignment of two sequences is evaluated in a matrix. In case of global alignment, the path of the optimal alignment through the matrix (lower right to upper left) starts in the lower right corner. For a local alignment the cell with the highest score is chosen as starting point for the alignment path. In this example of figure 7, the highest score is 3 in cell [3; 3] resulting in a maximum subsequence length of 3. The resulting optimal alignments are shown below the matrices. Further explanation to this technique is given in the chapter “Details on Smith-Waterman”.



A. Global

```
    A   B   D   D   E   F   G   H   I
A   \   \   \   \   \   \   \   \   \
      1 _-1  -1  -1  -1  -1  -1  -1  -1
B   \ ! \       \   \   \   \   \   \
     -1   2 _ 0 _-2  -2  -2  -2  -2  -2
D   \     ! \   \       \   \   \   \
     -1   0   3 _ 1 _-1 _-3  -3  -3  -3
E   \   \ !   ! \   \           \   \
     -1  -2   1   2   2 _ 0 _-2 _-4  -4
G   \   \     ! \ ! \   \   \
     -1  -2  -1   0   1   1   1 _-1 _-3
K   \   \   \ ! \ ! \ ! \   \   \   \
     -1  -2  -3  -2  -1   0   0   0 _-2
H   \   \   \   \ ! \ ! \ ! \   \   \
     -1  -2  -3  -4  -3  -2  -1   1 _-1
I   \   \   \   \   \ ! \ ! \ !   ! \
     -1  -2  -3  -4  -5  -4  -3  -1   2
```

Optimal global alignments (score 2):

```
     A B D D E F G H I  (top)
     A B D - E G K H I  (side)
  or A B - D E G K H I
```

B. Local

```
    A   B   D   D   E   F   G   H   I
A   \
      1   0   0   0   0   0   0   0   0
B       \
      0   2 _ 0   0   0   0   0   0   0
D         ! \   \
      0   0   3 _ 1   0   0   0   0   0
E             ! \   \
      0   0   1   2   2 _ 0   0   0   0
G               \ ! \   \   \
      0   0   0   0   1   1   1   0   0
K                       \   \   \
      0   0   0   0   0   0   0   0   0
H                               \
      0   0   0   0   0   0   0   1   0
I                                   \
      0   0   0   0   0   0   0   0   2
```

Optimal local alignment (score 3):

```
     A B D  (top)
     A B D  (side)
```

Fig. 7 - Global and local alignment paths (Image courtesy of W.R. Pearson (2001)).

Optimal Algorithms

Also referred to as rigorous algorithms, these algorithms find the optimal solution to a problem. By using a technique called dynamic programming the problem gets divided into smaller subproblems which are solved and reassembled afterwards in order to find a solution to the entire problem. This approach is very accurate, but compared to heuristic algorithms these are relatively slow.

Heuristic Algorithms

By using rules of thumb to reach a solution, heuristic algorithms are not that exact, but they are much faster than optimal approaches like dynamic programming. This kind of alignment is also not as tolerant as the optimal version and thus it might not find certain solutions.

The first use of the dynamic programming method was made by Needleman & Wunsch (1970) for pairwise global alignment and by Smith & Waterman (1981) for pairwise local alignment whereby the latter is based on the former. Both algorithms compute all possible alignments of two sequences therefore being very computation intensive and slow. As they can then choose the best of all possible alignments they are optimal algorithms. The Smith & Waterman algorithm is “the gold standard for protein-protein or nucleotide-nucleotide pairwise alignment” (Wikipedia 2005d) if an exact investigation needs to be performed. BLITZ is a database searching services of the European Bioinformatics Institute[5] which uses two different methods of dynamic programming both including Smith-Waterman: MPsrch 4 [6] and Scanps[7] (2.3) for protein vs. protein searches. SSearch (Sequence Similarity Search) is another well known Smith-Waterman tool which comes along the the FASTA package (Pearson & Lipman 1988). It incorporates code developed by Huang et al. (1990).

Due to the much better performance heuristic algorithms are commonly used for large scale searches. FASTA (Pearson & Lipman 1988) was the first tool of this category optimized for high speed search. Golding & Morton (2004) mention it to be 100 times faster than SSearch, Pearson (2001) speaks of 5 to 50 times speed improvement with results of similar quality. A simple “Comparison of Smith-Waterman, FASTA and BLAST” can be found in the appendix. Finally, concerning multiple alignment Golding & Morton (2004) mention Clustal (Higgins & Sharp 1989) to be the most popular program which in fact starts with a pairwise alignment as well.

The Smith-Waterman algorithm is the most accurate one of the presented methods. But since it is very computation intensive, mainly faster heuristic algorithms are use for large scale database searches. To be able to use the Smith-Waterman algorithm more often in practice, it is necessary to find ways to speed up it's execution. This thesis discusses a way to increase it's performance by using parallel streaming architecture. Commodity graphics cards are inexpensive streaming architectures that can be used in desktop PCs and other devices. The following section will explain the algorithm in detail before it is mapped to graphics hardware in chapter 4.

Smith-Waterman Algorithm in Detail

The method developed by Smith & Waterman (1981) is an optimal pairwise local alignment algorithm dealing with two sequences A=a_1, a_2, ... a_n and B=b_1, b_2, ... b_m. The similarity of two elements a and b is evaluated by the function s(a_i, b_j). It is the aim of the algorithm is to find two segments with maximum similarity. Another approach describes this process as minimizing the cost for transforming one sequence into another

5 http://www.ebi.ac.uk/searches/blitz_doc.html
6 http://www.ebi.ac.uk/MPsrch/
7 http://www.ebi.ac.uk/scanps/index.html

one. Therefore s(a_i, b_j) returns the cost for transforming element a into element b. For both similarity and cost the function s(a_i, b_j) refers to a specific substitution lookup table called scoring matrix. In addition, deletions and insertions cause gaps and are therefore treated with a penalty (weight) called gap cost. This is referred to as W_k for a gap with length k.

In order to find an optimal solution all possible alignments are set up in a matrix H with size (n+1)×(m+1). This matrix is initialized by setting row 0 and column 0 to zero:

$$H_{k0} = H_{0l} = 0, \quad 0 \le k \le n \text{ and } 0 \le l \le m$$

A matrix cell H_{ij} is calculated with

$$H_{ij} = max(H_{i-1,j-1} + s(a_i, b_j),\ \max_{k \ge 1}(H_{i-k,j} - W_k),\ \max_{l \ge 1}(H_{i,j-l} - W_l),\ 0),$$
$$1 \le i \le n \text{ and } 1 \le j \le m$$

and it therefore depends on the left, upper left and upper cells $H_{i-1, j}$, $H_{i-1, j-1}$, $H_{i, j-1}$. The elements a_i and b_j being associated is considered by the similarity

$$H_{i-1,j-1} + s(a_i, b_j) \quad .$$

The end of a gap with length k at element a_i is taken into account with the similarity

$$H_{i-k,j} - W_k$$

as well as the similarity for an ending gap with length l at element b_j

$$H_{i,j-l} - W_l \quad .$$

To avoid a negative similarity H_{ij} can not be lower than 0.

Because the Smith-Waterman solution requires a large number of steps of order M^2N, a modification was made by Osamu Gotoh (1982). A matrix cell is then described by

$$H_{i,j} = max(H_{i-1,j} + s(a_i, b_j),\ E_{i,j},\ E_{i,j},\ 0), \qquad 1 \le i \le n \text{ and } 1 \le j \le m$$

Each

$$E_{i,j} = max(H_{i-k,j} - W_k)$$

and

$$F_{i,j} = max(H_{i,j-l} - W_l)$$

gets substituted by a recursive expression

$$E_{i,j} = max(H_{i-1,j} + \alpha,\ E_{i-1,j} + \beta), \qquad 1 \le i \le n \text{ and } 0 \le j \le m$$
$$\alpha = W_1 \text{ and } \beta = W_{k>1}$$

and

$$F_{i,j} = max(H_{i,j-1} + \alpha,\ F_{i,j-1} + \beta), \qquad 0 \le i \le n \text{ and } 1 \le j \le m$$
$$\alpha = W_1 \text{ and } \beta = W_{l>1}$$

where α is the weight of the first gap element and β is the weight of the following gaps. This is the so-called affine gap penalty. A linear gap penalty is present if α=β. In this case the rule definitions for the evaluation of a cell can be simplified to

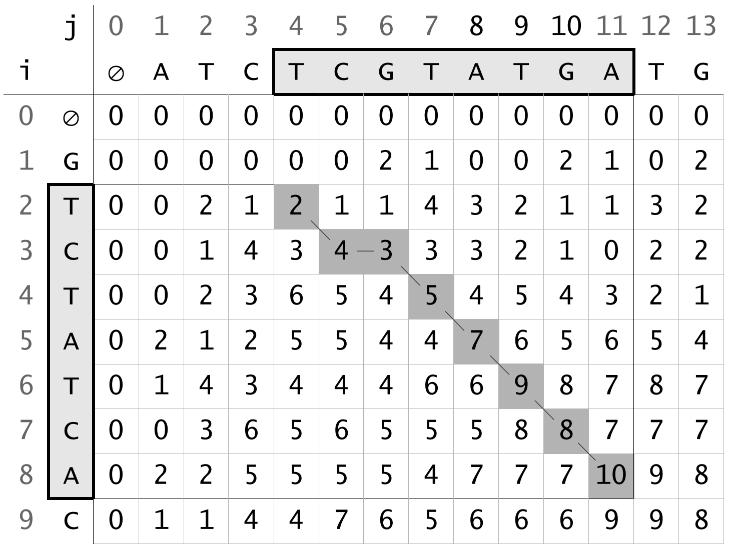
$$H_{i,j} = max(H_{i-1,j-1} + s(a_i, b_j),\ H_{i-k,j} - \alpha,\ H_{i,j-l} - \alpha,\ 0),$$
$$1 \le i \le n \text{ and } 1 \le j \le m$$

Due to the dependency of H_{ij} to $H_{i-1,\ j}$, $H_{i-1,\ j-1}$, $H_{i,\ j-1}$ the matrix needs to be computed in an order that satisfies this. The most obvious ways are to go by row from top down and within a row from left to right or to go by column from left to right and within a column top down. Figure 8 shows the former version. The Smith-Waterman matrix of two DNA sequences A=GTCTATCAC and B=ATCTCGTATGATG is shown in the process of evaluation. In this case, a linear gap penalty of α=β=1 was chosen. In addition, the substitution cost for matching elements is simply $s(a_i, a_i)=2$ and for non-matching $s(a_i, b_j)=-1$, a≠b. The cells are computed by row from top down and from left to right.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
i		∅	A	T	C	T	C	G	T	A	T	G	A	T	G
0	∅	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	0	0	0	0	0	2	1	0	0	2	1	0	2
2	T	0	0	2	1	2	1	1	4	3	2	1	1	3	2
3	C	0	0	1	4	3	4	3	3	3	2	1	0	2	2
4	T	0	0	2	3	6	5	$\boxed{4}$	→						
5	A	0													
6	T	0													
7	C	0													
8	A	0													
9	C	0													

Fig. 8 - Smith-Waterman matrix computation with two DNA sequences A=GTCTATCAC and B=ATCTCGTATGATG.

Having the full matrix the optimal alignment can be picked out. Beginning with the highest value of H it is backtracked along the path of cells of which the values were derived from. This is done until an H_{ij} with value zero is reached. The aligned subsequences are determined by projecting the path onto the compared sequences. A more detailed description of the backtrack process can be found in Gotoh (1982).



	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
i		∅	A	T	C	T	C	G	T	A	T	G	A	T	G
0	∅	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	0	0	0	0	0	2	1	0	0	2	1	0	2
2	T	0	0	2	1	**2**	1	1	4	3	2	1	1	3	2
3	C	0	0	1	4	3	**4**	**3**	3	3	2	1	0	2	2
4	T	0	0	2	3	6	5	4	**5**	4	5	4	3	2	1
5	A	0	2	1	2	5	5	4	4	**7**	6	5	6	5	4
6	T	0	1	4	3	4	4	4	6	6	**9**	8	7	8	7
7	C	0	0	3	6	5	6	5	5	5	8	**8**	7	7	7
8	A	0	2	2	5	5	5	5	4	7	7	7	**10**	9	8
9	C	0	1	1	4	4	7	6	5	6	6	6	9	9	8

Fig. 9 - Result of the traceback.

In figure 9 the backtracking result is shown. Beginning from the highest value (here: 10) the optimal alignment path is evaluated backwards until a zero cell is reached. The resulting optimally aligned subsequences are

```
TC-TATCA
|| ||| |
TCGTATGA
```

including one gap and one mismatch.

Scoring Matrices

The Smith-Waterman implementation in chapter 4 uses a BLOSUM62 scoring matrix (Blocks Substitution Matrix) to evaluate a sequence alignment. This chapter explains what scoring matrices are and why they are used by sequence alignment algorithms.

In the above example s(a, b) simply returned a score of 2 for a=b and -1 for a≠b. But in reality all substitutions are not equally likely. They have different consequences, and they therefore need to be weighed to account for this. An example for different consequences are synonymous and nonsynonymous substitutions in nucleic codons. Many amino acids are encoded not only by one but by several codons. Thus, a synonymous change of the triplet does not mean that it results in a different amino acid: GGG, GGA, GGC as well as GGU encode glycine. However, a nonsynonymous substitution in GGG would result in one of the following amino acids: valine (GUG), alanine (GCG), glutamine acid (GAG), tryptophan (UGG), and arginine (CGG, AGG). Following, the most common types of scoring matrices are presented (Golding & Morton 2004, Pearson 2001, IISR 2003).

Unitary Matrix

This trivial kind of matrix has only two different values, one along the diagonal and one for the rest of the matrix. The scoring matrix in the example above is of this type. All matrix elements are set to -1 except the diagonal (a=b) was set to 2. This matrix can be found in the appendix.

PAM

The term PAM (Point Accepted Mutation) was introducted by Dayhoff et al. (1978). This matrix is based on an explicit evolutionary model where mutations are counted on branches of a phylogenetic tree. In other words, the weights of the matrix are derived from how frequently each amino acid was replaced by another in evolution. The investigations for the original PAM250 by Dayhoff et al. (1978) comprises 1,572 changes in 71 groups of closely related proteins. An evolutionary distance of one PAM corresponds to mutation ratio of about 1 % meaning that 1 out of 100 amino acids got substituted. Thus, PAM250 is suitable for distantly related proteins. Proteins having diverged by 250 % still match with about 20 % due to back mutations and silent mutations resulting in a divergence of still 80 %. It is considered a good general matrix for protein database searching (IISR 2003).

BLOSUM

The principle of the Blocks Substitution Matrix was introduced by Henikoff & Henikoff (1992). This matrix is based on a rather implicit model of evolution and gets derived from groups of aligned sequences that differ by no more than X %. Thus, for the BLOSUM62 proteins that do not differ more than 62 % were used. For these purpose the BLOCKS[8] database is searched for highly conserved segments in series of alignments without gaps.

BLOSUM matrices are best for detecting local alignments and BLOSUM62 which can be found in the appendix is used in BLASTP for protein database searching. BLOSUM62 is best for the majority of weak protein similarities and BLOSUM45 is best for

8 http://blocks.fhcrc.org/

detecting long and weak alignments (IISR 2003). A list of equivalent PAMs and BLOSUMs is shown in table 1.

PAM		BLOSUM
PAM100	~	BLOSUM90
PAM120	~	BLOSUM80
PAM160	~	BLOSUM60
PAM200	~	BLOSUM52
PAM250	~	BLOSUM45

Table 1 - Equality of PAM and BLOSUM.

GONNET

Another method to obtain a scoring matrix is iterative and was developed by Gonnet et al. (1992). They use classical pairwise protein alignment for distance measurement and use the derived values to create a distance matrix. This matrix is then used in the next iteration step to refine the alignment of the same sequences estimating a new, more precise matrix. This is repeated until almost no changes in the resulting matrices occur anymore. The GONNET matrices are similar to PAM, but as they are derived from a larger set of sequences they are more sensitive and should therefore be used in preference to PAM (Golding & Morton 2004). GONNET250 is equivalent to PAM250.

2.7 Summary

This chapter introduced the basics of bioinformatics and sequence. In order to understand the algorithms dealt within this biological context, a genetics and its terminology is helpful. Basics in both genetics with focus on DNA and protein sequences and sequence alignment with focus on the Smith-Waterman algorithm were presented. Since genome projects exist, the amount of genetic data that is produced by research every year increases exponentially. The processing of this data needs algorithms that are fast enough to handle the data in an adequate time. On the one hand, the Smith-Waterman algorithm is an accurate algorithm that is very computation intensive. On the other hand, implementing the algorithm on parallel streaming architecture can speed up the alignment process a lot. This is why this thesis presents an implementation of the Smith-Waterman algorithm realized on inexpensive GPU-based streaming architecture.

The implementation of the algorithm described in chapter 4 is organized by a CPU application and computed on a GPU. The program will be able to load a scoring matrix from a file and to load sequences in FASTA format. In following tests a BLOSUM62 will be used during a whole database scan of Swiss-Prot.

The following chapter gives an overview over graphics hardware and GPU programming. It shows state-of-the-art programming methods and discusses features and the potential of this technique. This knowledge is necessary to to understand the techniques used for that Smith-Waterman implementation discussed in chapter 4.

3 GPGPU

3.1 Introduction

Using GPUs for general-purpose computation (GPGPU) can be a significant speed up compared to using only CPU power, but the way of programming and the way of thinking about the realization of an algorithm is different. The possible advantage in speed comes along with a series of limitations and disadvantages. Thus, this chapter gives an idea of how to work in this context.

After an explanation of why GPUs can be suitable for general-purpose computation, previous work is presented. Furthermore, graphics card technology represented by the hardware graphics rendering pipeline is discussed. Based on that knowledge computational concepts on GPUs and CPU-GPU analogies are shown. Finally, GPU programming itself is introduced and different programming languages are presented and compared. This chapter presents all techniques that are used for the implementation in chapter 4. The focus will therefore lie on these techniques.

3.2 GPUs are usable for General Purpose Computation

GPUs are fast. Table 2 shows a CPU – GPU performance comparison (floating point operations per second and data throughput) from 2004 (Harris 2004b, Houston 2005).

The value marked with * has been reached in a synthetic benchmark which consisted only of a long pixel shader with nothing but MUL (multiplication) instructions. The observed peak GFLOPS performances of the listed GPUs are 4.4 and 5.6 times the theoretical peak performance of the CPU.

Processor	Type	Case	Gflops	Throughput (peak)
Pentium4 3.4 GHz	CPU	theoretical:	13.6	5.96 GB/sec
Nvidia GeForce 6800 Ultra	GPU	observed*:	51.2	32.7 GB/sec
ATI Raden X800 XT	GPU	theoretical:	66.6	33.4 GB/sec

Table 2 - CPU - GPU performance comparison.

CPUs are designed for low latency computations whereas GPUs are optimized for high throughput. Increasing the throughput includes adding transistors to GPUs which consist of several processors and functional units. This is why the growth of the amount of transistors used on GPUs is greater than it was predicted for microprocessors by Gordon Moore (1965). Derived from observations of past years he predicted in his so-called Moore's Law that the number of transistors per square inch on integrated circuits would continue doubling every year. In fact, an exponential growth proved to be true but with an exponent of 1.5 like shown in table 3 (still referred to as Moore's Law).

Processor	Annual Growth	Decade Growth
CPU	1.5×	60×
GPU	> 2.0×	> 1000×

Table 3 - Growth rate of number of transistors on CPU and GPU.

In contrast, the relatively young technology of GPUs shows an annual growth of more than 2 which means that the number of transistors is multiplied by more than 1,000 within 10 years. That means that within the next years GPUs will become even more attractive as an computation alternative to CPUs. In addition, the time between new generations of GPUs is with 6 months much lesser than for CPUs, making improvements in technology available more frequently.

The fast development is driven by the game industry (Luebke 2004, Harris 2004b, Lefohn 2004, Fernando et al. 2004). Thus, GPUs are designed to render more and more realistic complex 3D scenes in realtime (see figure 10). They are also inexpensive, easily upgradable, and compatible with multiple operating systems and hardware architectures. They are already present in many different devices: desktop PCs, laptops, handheld PCs, PDAs and even cell phones.

Initially, GPUs had a fixed function render pipeline giving the software developers only little space to influence the result. But since 2001 they started becoming programmable,

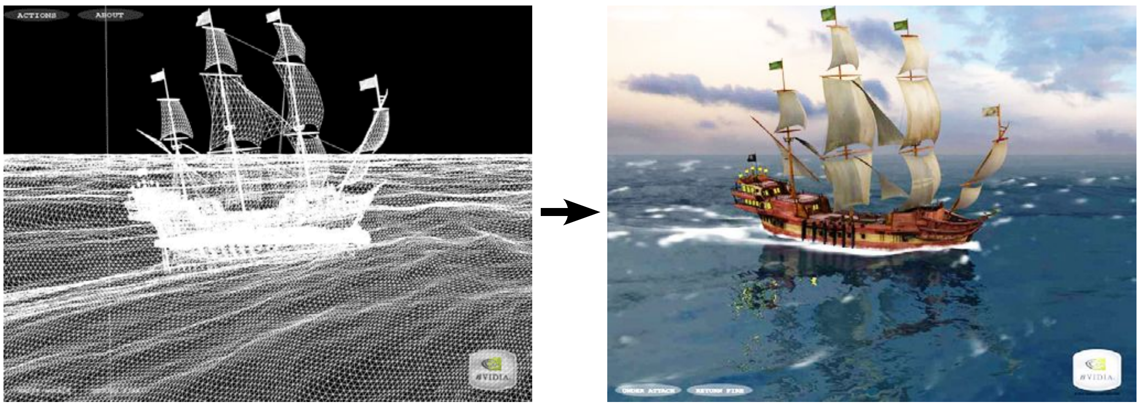


Fig. 10 - GPUs are designed to render 3D scenes: From triangles to pixels in real-time. (Figures Courtesy of NVIDIA)

starting with vertex programs executed by the vertex processors and later fragment programs executed by the fragment processors. 2002 16bit floating point numbers where added, enhanced to 32bit in 2004. GPUs therefore became suitable for general purpose computations beyond graphic applications including scientific computation. Furthermore, high level languages like NVIDIA's C for graphics (Cg), OpenGL's GLSL and Microsoft's HLSL. More about these languages can be found in the chapter “Programming the GPU”.

Regarding the architecture, GPUs are parallel streaming processors optimized for vector operations (Lindholm et al. 2001). Parallel architectures have advantages that can speed up suitable applications like in-game physics simulation enormously. Thus, it is already spoken of the “beginning of the desktop parallel computing age” (Lefohn 2004). Some example implementations are presented in the next chapter.

Of course, this technology brings along some disadvantages and limitations and the programming model is different as well, tied to computer graphics (Luebke 2004). Code written for a CPU cannot simply be ported to a GPU solution. However, because of its speed it is tried to make the inexpensive computational power available to developers as a sort of coprocessor. GPUs are made more flexible and new programming languages make application development easier.

3.3 A Brief History of GPUs and GPGPU

Before the rise of 3D support for PCs in 1995 the CPU had to handle all vertex transformation and pixel rendering (Fernando 2004, Fernando et al. 2004, Zeller 2004b). Although it was very comfortable that everything about vertex and pixel processing was programmable, the CPU was simply too slow to render complex 3D scenes in realtime. The 3dfx Voodoo which was released in 1996 is generally credited as the first graphics processor for PC architecture. It is a 3D accelerator card that was used in addition to a common graphics card and it was limited to processing two-dimensional (2D) triangles only. Thus, the geometry was still computed by the CPU. Later, models of other GPU developers like NVIDIA, Matrox and ATI Technologies Inc. (ATI) outmatched 3dfx cards and established themselves, combining the classical graphics card with 3D acceleration that included vertex processing as well. A big advantage of GPU technology was also, that 3D game developers didn't have to implement their own 3D rendering algorithms anymore. Instead they used 3D APIs like DirectX and OpenGL as communication interface between the application and the graphics card.

Table 4 gives a rough description of the release of further GPU features:

Year	Introduced feature
1995	Texture mapping and z-buffer
1998	Multitexturing
1999	Transform and lighting
2001	Programmable vertex shader
2002	Programmable pixel shader
2004	Shader model 3.0 and 64-bit color support

Table 4 - Introduction of new GPU features.

Figure 14 shows the performance development of GPUs in the last decade in a logarithmic graph. The per-pixel rate grows from far less than 100 Mpixels/s to almost 10,000 Mpixels/s. Below the graph the year related technological progress (bus type and speed, video memory) are listed together with the at that time available versions the 3D APIs Microsoft DirectX and OpenGL. The improvement of render quality that occured at the same time shall be emphasized by the figures 11 to 13 (Fernando et al. 2004). Two of these are games whereas the third one is a demo application. All three cases show quite similar scenarios with one or two characters being in the focus of interest. The second figure has a 3D background whereas the remaining two only have an environment image in background. Quality evolves from poorly shaded low polygon models to almost photorealistic characters with multitexturing, translucency and hair.

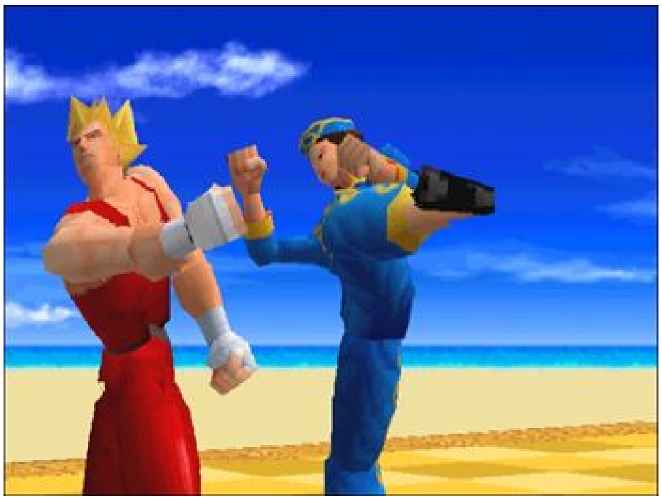


Fig. 11 - Virtua Fighter (SEGA Corporation)

1995: Virtua Fighter

(SEGA Corporation)
NVIDIA NV1
1M transistors
50K triangles/sec
1M pixel ops/sec
16-bit color
640×480
Nearest filtering

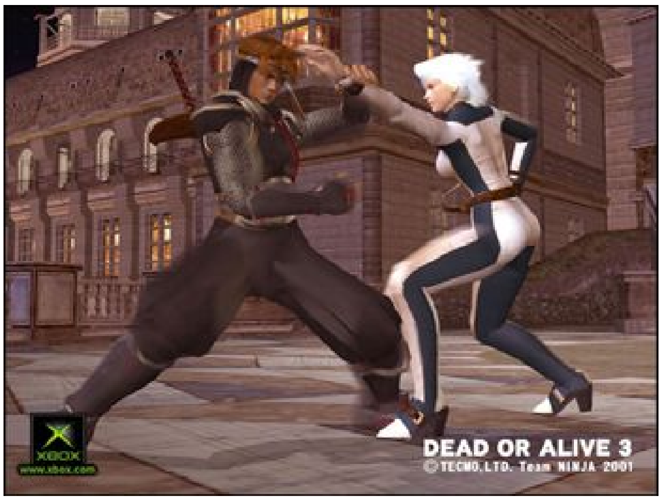


Fig. 12 - 2001: Dead of Alive 3 (Tecmo Corporation)

2001: Dead or Alive 3

(Tecmo Corporation)
Xbox (NV20)
20M transistors
100M triangles/sec
1G pixel ops/sec
32-bit color
640×480
Trilinear filtering

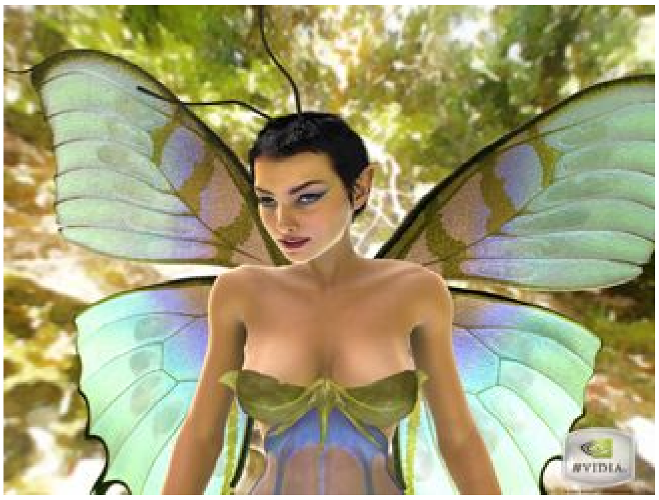


Fig. 13 - 2003: Dawn Demo (NVIDIA Corporation)

2003: Dawn Demo

(NVIDIA Corporation)
NVIDIA GeForce FX (NV30)
120M transistors
200M triangles/sec
2G pixel ops/sec
64-bit color
1024×768
8:1 Anisotropic filtering

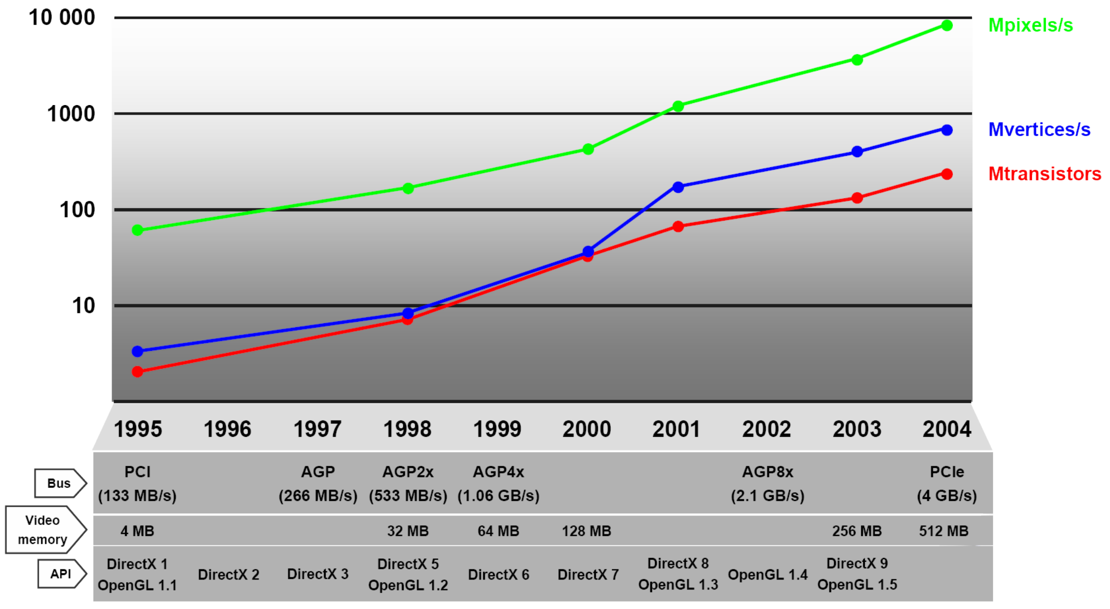


Fig. 14 - Performance evolution of the GPU (image courtasy of C. Zeller).

The history of GPU programmability is short: it did not start before 2001 when NVIDIA and ATI made the vertex units of their new GPU models programmable. The new feature was the ability to upload a small vertex program usually referred to as vertex shader which then was executed. In 2002/2003 both companies introduced programmable fragment processors as well. In contrast to the initial 8-bit fixed point format, 32-bit floating point precision is standard for today's high-end cards and is very important for accurate calculations as they occur in scientific applications. In addition, these GPUs have up to 512 MB memory, 6 vertex shader units and 16 pixel shader units. The PCIe bus gives a high bandwidth between CPU and GPU with a peak of 4GB/s up- and download.

The idea of using the power of graphics hardware for general purpose computation or simply for applications they are not designed for is not new. The first approaches have been done on machines like the Ikonas (England 1978), the Pixel Machine (Potmesil & Hoffert 1989), and Pixel-Planes 5 (Rhoades et al. 1992) before the age of GPUs. Examples for papers and implementations using GPUs for general-purpose computation between 1990 and 2004 deal with robot motion planning (Lengyel et al. 1990), procedural texturing and shading (Olano & Lastra 1998, Peercy et al. 2000, Proudfoot et al. 2001, Rhoades et al. 1992), collision detection (Hoff et al. 2001; Govindaraju et al., 2003), ray tracing (Carr et al. 2002, Purcell et al. 2002), image based modelling (Yang et al. 2002; Hillesland et al. 2003), photon mapping (Purcell et al. 2003), multigrid solvers for boundary value problems (Goodnight et al. 2003), physically-based visual simulation

(Harris et al. 2002), simulation of cloud dynamics (Harris et al. 2003), simulation of dendritic ice crystal growth (Kim & Lin 2003), and database operations (Govindaraju et al. 2004). This list was a small excerpt of what has been done so far.

By mapping the Smith-Waterman algorithm to graphics hardware, this thesis ties up to the presented examples. The following chapters present the techniques used to implement general-purpose applications on GPUs including the consequent implementation. To understand these techniques a basic understanding of graphics hardware is required. The next chapter therefore introduces the hardware graphics pipeline.

3.4 Introduction to the Hardware Graphics Pipeline

Efficient GPU programming requires a basic knowledge of graphics hardware. It is the prerequisite to taking advantage of its features. The fundamentals are described in this chapter whereas the following chapter focuses on using this technology for general-purpose computing.

The rendering of a 3D scene consists of many different steps commonly referred to as the Graphics Pipeline (see figure 15). The entire process consists of three main stages: the application stage, the geometry stage, and the rasterization stage. The application stage organizes all 3D objects, their properties and transforming nodes depending on game- or application logic and animation. The geometry stage is responsible for all vertex transformations from world space to view space and to image space. This stage mainly consists of multiplications of transformation matrices with vectors and vector normalizations. Drawing the pixels is a task of the rasterization stage. First, the triangles get rasterized before their surface properties like shading, texture and transparency are evaluated. Geometry stage and rasterization stage can be processed by software or by hardware like GPUs. Thus, it is either spoken of Software Graphics Pipeline or Hardware Graphics Pipeline.

In the case of GPUs, the rasterization stage gets divided into two stages, thus having three stages in total: vertex processing, rasterization, and fragment processing (see figure 16) (Lindholm et al. 2001, Buck et al. 2004). The vertex and fragment stages became programmable with the introduction of vertex and fragment processors. In fact, there are several processors working parallely. Modern graphic cards have 6 vertex processors (6 vertex shader units) and 16 fragment processors (16 pixel pipelines). Each type of them is able to execute user defined assembly-level programs, also referred to as shaders or shader programs.

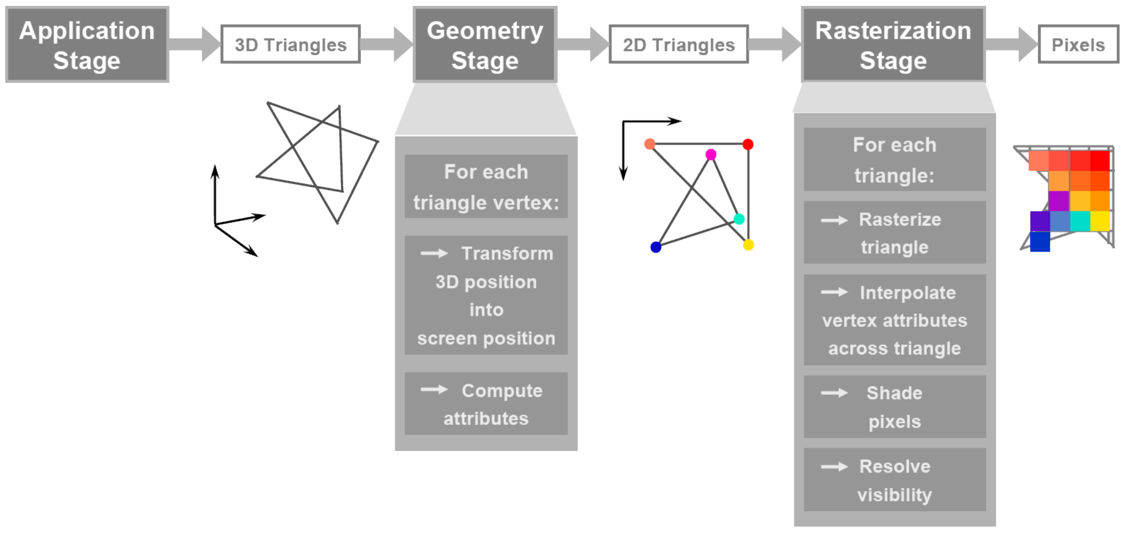


Fig. 15 - Graphics Pipeline Stages: Application Stage, Geometry Stage, and Rasterization Stage (Figure Courtesy of C. Zeller (2004b)).

These programs basically allow mathematical operations, texture fetching, and some special purpose operations. Due to high-level shading languages shaders can be written with C-like syntax and get translated into assembly-level code. Furthermore, there are two important major differences to classical programming:

- Elements are independent: while processing a vertex, it is not possible to access other vertices. While processing a fragment, it is not possible to access other fragments as well.
- Registers can only be read or written, but never both at a time (except temporary ones). The information the processors get from the previous stage can only be read whereas the output (e.g. framebuffer) can only be written.

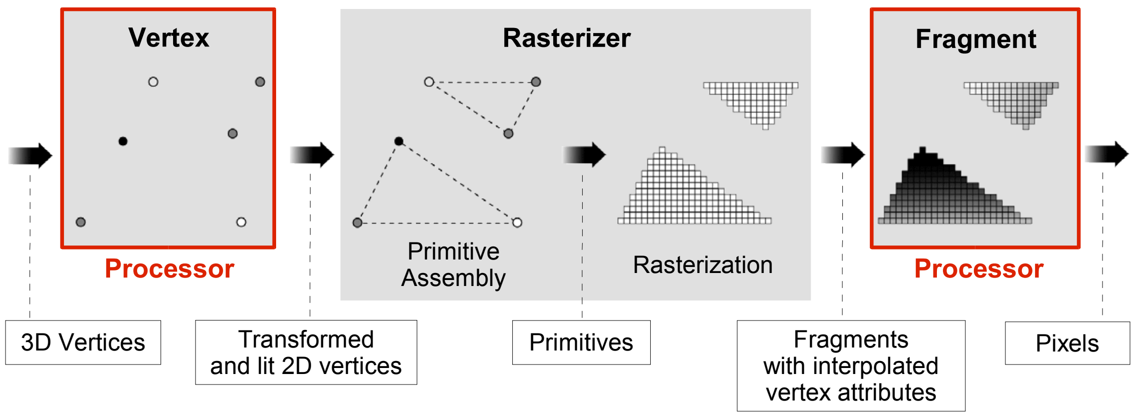


Fig. 16 - Three stages on the GPU (image courtesy of Randy Fernando (2004), modified).

Each stage is described in the following (Kessenich et al. 2004, van der Linden 2004, Luebke 2004):

Vertex Processor

The (programmable) vertex processor transforms vertices from world space into image space (no viewport mapping, see figure 17), generates and transforms texture coordinates, and computes the per-vertex lighting and material color. It gets the required information such as transformation, lighting and material from the application. Each processor handles only one vertex at a time and has no access to the others.

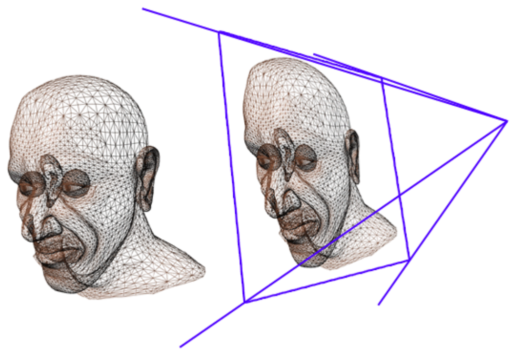


Fig. 17 - Transformation from world space to image space by vertex processors (image courtesy of Luebke (2004)).

Rasterizer

In contrast to the vertex and fragment processing, this step is not limited to single vertex or single fragment operations, but it is not programmable. This step creates fragments (preliminary pixels) by rasterizing primitives (see figure 18) and it is therefore responsible for primitive assembly as well. Furthermore, it computes perspective divisions, viewport mapping, clipping, and backface culling. Per-vertex information like color are interpolated between vertices and are output as per-fragment information. This also includes values like depth and stencil.

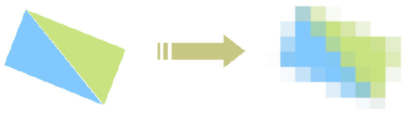


Fig. 18 - Rasterization of primitives (image courtesy of Luebke (2004)).

Fragment Processor

The fragment processor calculates the pixel color for every fragment that the rasterizer sends. A pixel color usually consists of 4 components: red, green, blue, and alpha (RGBA). In addition to per-fragment values global information like textures is accessible. Texture coordinates and z-depth (e.g. for fog) in contrast are per-fragment values. Since multitexturing is available, complex materials can be applied. In figure 19 bumpmapping is used to simulate a detailed surface structure of a man's face, a color map determines the surface color, and a gloss map makes areas on the surface more shiny than others

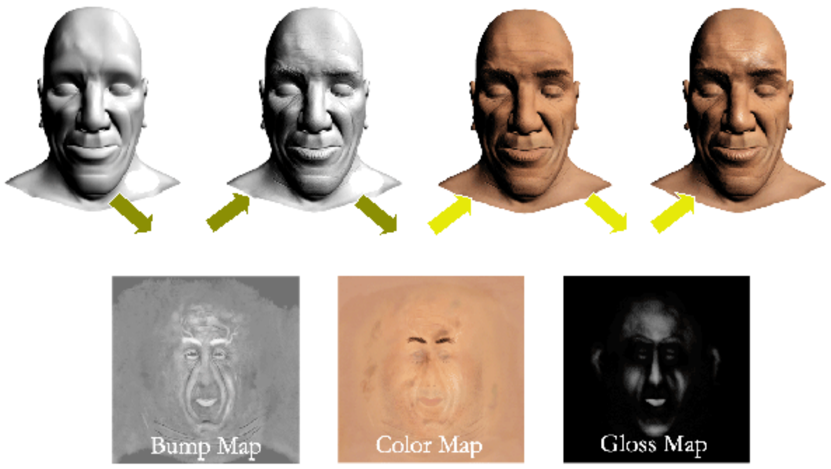


Fig. 19 - Multitexturing (image courtesy of Luebke (2004), modified contrast).

It is important to consider an adequate arithmetic intensity (compute-to-bandwidth ratio) (Buck 2004): a vertex needs a bandwidth of 32 bytes and shouldn't be computed with more then 100 to 500 32bit floating point operations (f32-ops). A fragment in contrast needs a bandwidth of only 10 bytes and can therefore be evaluated by more complex algorithms with 300 up to 1000 8-bit operations (i8-ops). 8-bit color components (RGBA

= 4 × 8 bit = 32 bit color) are used in classical GPUs before 16-bit, 24-bit (ATI GPUs) and 32-bit (Nvidia GPUs) floating point precision was available. With this precision high dynamic range images (HDRI) can be used as well as accurate general purpose computations can be done. A number format in between was 12-bit fixed point. The output for display devices remains 8 bit per component.

Figure 20 shows a simplified model of the graphics pipeline with today's shader model 3.0 graphics cards such as the Nvidia Geforce 6800 (Nvidia 2004) and ATI Radeon X800 (ATI 2004). The application runs on the CPU and passes geometry (commonly triangles, but more formats exist) and textures as well as additional information like transformation instructions to the GPU. Both are connected via a PCIe bus (Peripheral Component Interconnect Express) which was introduced in 2004. Until then the AGP bus (Accelerated Graphics Port, introduced in1997) has been used. Much older cards use the PCI bus (Peripheral Component Interconnect, introduced 1992). With Shader Model 3.0 both the vertex processor and the fragment processor are programmable and can have multiple textures as input. Shader Model 2.0 only allowed the fragment processor to get texture input. Especially for iterative algorithms it is very important that rendered values can be reused by loading a rendered buffer as texture.

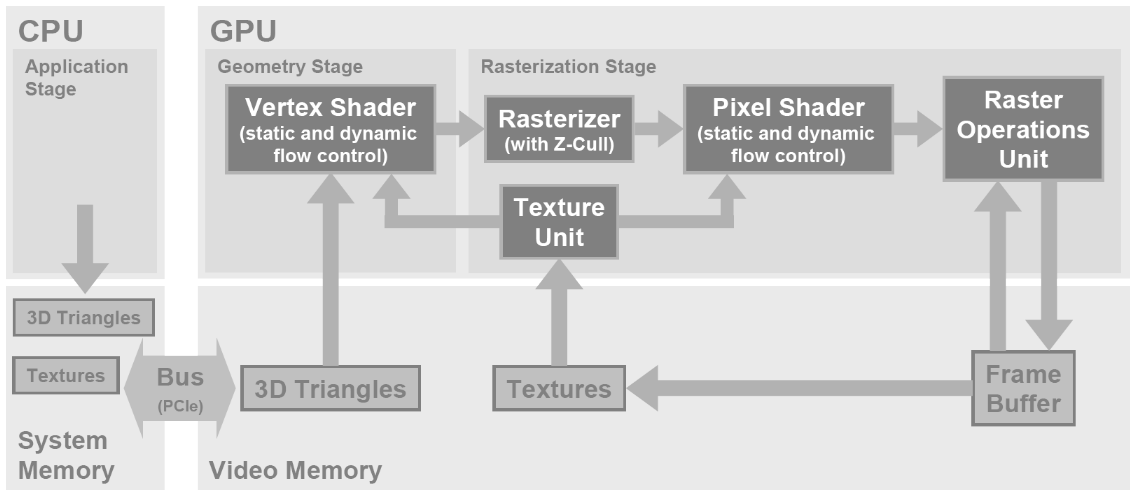


Fig. 20 - Graphics Pipeline with Shader Model 3.0 (image Courtesy of Cyril Zeller (2004b)).

How to use the graphics pipeline (with focus on fragment processing) and how to reuse buffers is described in the following chapters.

3.5 Mapping Computational Concepts to the GPU

The memory and processing model of GPUs is different to that of CPUs. General-purpose programming on GPUs requires a basic knowledge of computational concepts on graphics hardware. These as well as GPU-CPU analogies are presented in this chapter.

GPUs are designed for graphics applications and therefore they fit their characteristics (Owens 2004, Harris 2004, Lefohn 2004b). Graphics applications are arithmetic intensive tasks which are suitable to compute parts of it on parallel systems. GPUs feature lots of parallelism with their parallel feed forward pipelines, but these bring along some properties that need to be taken into account.

Purcell et al. (2002) describe modern GPUs as *streaming processors*. Streaming processors read an input stream, apply kernels (filters) to the stream and write results to an output stream. In case of several kernels, the output stream of leading kernels is the input stream for the following (see figure 21).

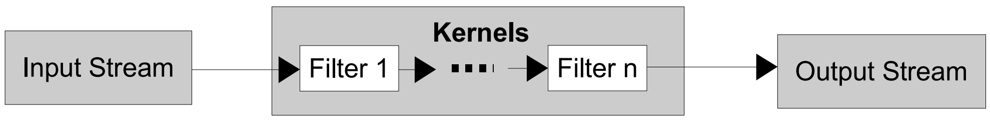


Fig. 21 - Streaming model that applies kernels to an input stream and writes to an output stream.

Streams

A stream is a collection of data records that require a similar computation and that can be operated on in parallel. Stream elements can be e.g. vertices, pixels of an image in case of image processing, voxels, or finite element method cells.

Kernels

A kernel is the computation / function that is applied on each stream element. This can be, for instance, a transformation or a partial differential equation. As elements are processed parallely, no computational dependency between stream elements must exist. It is neither possible to access other elements nor is it possible to read the output. The analogue to a kernel is a vertex or fragment program.

Two properties make stream processing so fast: explicit parallelism and explicit memory locality (Lefohn 2004b, Harris 2004, Lefohn 2004b). Stream elements are independent and no communication between stream elements or kernels is possible. There's no shared or static memory like global variables. Every temporary value in a kernel is local and each temporary register is zeroed after execution. As there are no read-modify-write buffers kernels are neither able to write into input streams nor to read from output streams.

Most general-purpose applications use only fragment programs for computation whereby the input comes from textures. Thus, textures are considered input streams and the buffers that is written into are output streams. Because fragment processors are SIMD[9] (Single Instruction Multiple Data) architectures, only one program can be loaded at a time. Applying several kernels thus means to do several passes (see figure 22).

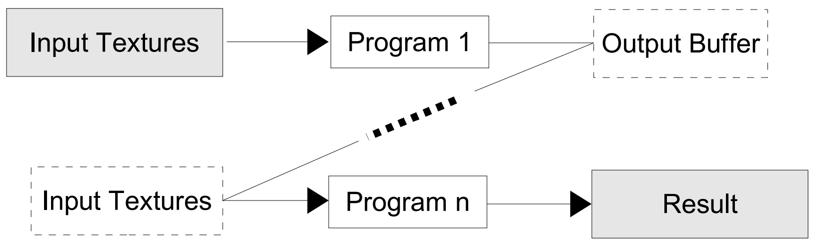


Fig. 22 - Multipass method for applying n instruction sets.

This method implies the *three basic concepts* most general-purpose GPU applications are based on (Fernando et al. 2004):

1. Textures are data input.
2. Input is processed by a vertex or fragment program.
3. Feedback is realized by using the output buffer of a completed pass as input texture for the following one.

Textures

The analogue of arrays on the CPU are textures on the GPU (see figure 23) (Fernando et al. 2004, Harris 2004). Textures are best suitable to represent one, two, or three dimensional grids and they are therefore used for grid simulation computation. That means that an algorithm that originally does not use grids needs to be remapped to a grid algorithm. Many algorithms map to grids. Some examples are physical matrix algebra, image and volume processing, global illumination, ray tracing, and fluid simulation. Textures are optimized to contain data with up to four components like RGBA color. They can therefore be used to store vectors with up to four components or up to four scalars which then can be interpreted as four grid layers. The analogue of an array read is a tex-

9 On parallel SIMD systems several processors apply instructions on independent elements synchronously (Alaghband 1997). MIMD (Multiple Instructions Multiple Data) systems in contrast have several processing units that work asynchronously and apply distinct instructions sets independently on streams.

ture lookup. The integer array offset therefore corresponds to a floating point texture coordinate with the same dimension of the texture. Texture-lookups are very fast if they can be pre-fetched. This is the case if the textures lookup coordinates are foreseeable, e.g. if the coordinates are constant or if the fragment texture coordinate is used. Dynamic texture-lookups in contrast are quite slow. If the texture coordinate is computed at runtime because it depends on values that result from a previous lookup, it cannot be prefetched.

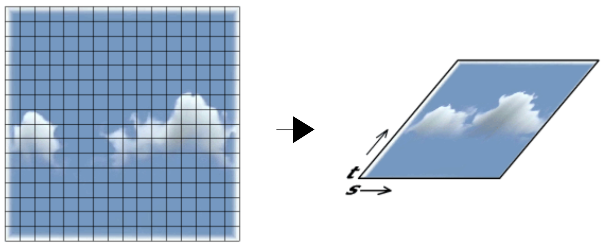


Fig. 23 - The analogue to an array in the CPU is a texture in the GPU (Image courtesy of Harris (2004)).

Programs

If an algorithm is applied to a grid using a CPU, nested loops are usually used to iterate over the grid whereby the loop body contains the instructions (Fernando et al. 2004). The analogue to the loop body is the kernel or fragment / vertex program. Figure 24 shows a simple C++ loop and its equivalent fragment program written in GLSL. The two nested loops iterate over a grid whereby the loop body is applied to each grid element. The fragment program is automatically executed for each pixel of the buffer area that is rendered. The application passes a stream of elements to the GPU which then applies the loaded kernel to each element on all available processors in parallel. If an application starts a rendering on the GPU it is always a for-each-call.

```
for(int i = 0; i < height; i++)
{
   for(int j = 0; j < height; j++)
   {
      Vec2f a = grid(i, j);
      a.x *= u;
      a.y *= v;
      grid(i, j) = a;
   }
}
```

C++

```
uniform sampler2DRect grid;
uniform vec2          uv;

void main(void)
{
   vec2 a = texture2DRect
            (grid, gl_TexCoord[0]);
   gl_FragColor = a * uv;
}
```

GLSL

Fig. 24 - The loop bodies of CPU programs that iterate over arrays correspond to kernels applied to streams.

GPU programs can either scatter or gather information (Harris 2004, Fernando et al.

2004). A vertex program cannot gather information, because it cannot access other vertices but the processed one. On the other hand it can change the vertex coordinates which influences the position the pixels that are rendered. Therefore, it can perform scattering algorithms. The fragment program in contrast can randomly access every input texel of the texture, but it can write only the pixel it is processing. Thus, it can perform only gathering algorithms (see figure 25).

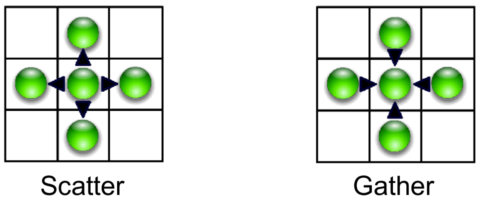


Fig. 25 - Scattering and gathering in a grid (image courtesy of Harris (2004)).

GPU programs have two levels of parallelism. First, multiple processors execute the program in parallel. Second, some arithmetic operations can be applied to multiple values in parallel. This is because GPUs are optimized for vector mathematics. Built-in vector functions are executed within one cycle. Thus, if four scalars are put into a vector, it is possible to execute four scalar operations in one cycle.

Feedback

Getting feedback is not as trivial as it is in CPU applications which normally have read-and-write variables (Harris 2004). GPU programs can only read input and write output within one pass. Thus, iterative algorithms need multiple passes whereby the result of a completed pass is the input for the upcoming pass (see figure 26). There are two techniques to realize that:

1. CTT (Copy-To-Texture) is the older and much slower method of both. A pass renders its result to framebuffer. To use the data in the framebuffer as input for the next pass it first needs to be copied into a texture first. Then it can be used as input. This process is very slow, because the buffer is transferred from GPU memory to CPU memory where it is copied to a texture and transferred back when the texture is loaded.

2. RTT (Render-To-Texture) is a method which is supported by Shader Model 2.0+ GPUs. This technique requires a buffer that can be bound as render target and as texture. Therefore, the render result can directly be used as input for the next pass without the need to copy it. Actually, there are two buffer models that support this technique: pbuffers and framebuffer objects.

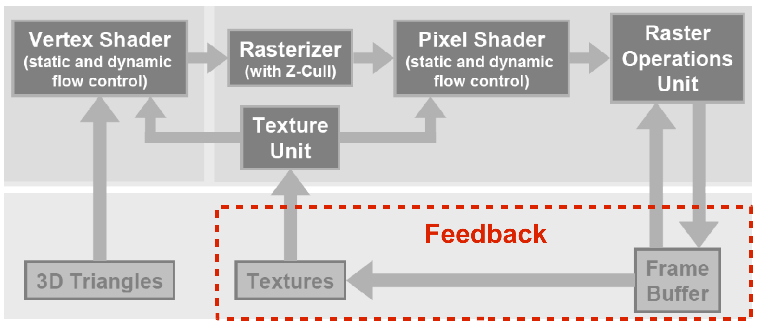


Fig. 26 - Feedback by using the output buffer as input for a following pass (image courtesy of Harris (2004), only partially shown).

Invoking Computation

Computation is invoked by drawing geometry (Harris 2004). Commonly a quad is drawn into the render buffer in form of a rectangle covering the pixels that shall be evaluated. A quad is a polygon consisting of four vertices. Therefore, all four vertices must be defined. In addition, texture coordinates are defined for the textures that are used. In graphic applications, this is called texture projection or texture mapping and it is used to apply images onto geometry. Each vertex has its individual coordinate on each texture. Texture coordinates for pixels in between are interpolated. If this is done, the fragment program automatically gets the texture coordinates that correspond to the relative location of the pixel in the rendered quad with respect to the mapping (see figure 27). After the rasterization stage, the fragment program is executed for each of the fragments the quad got converted into.

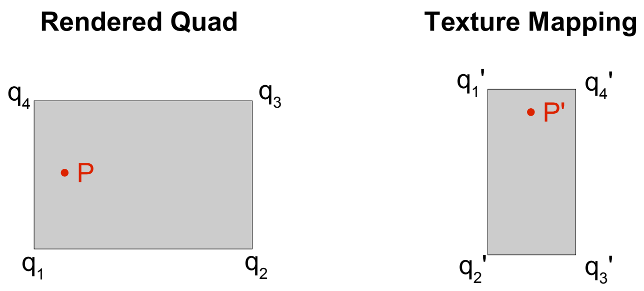


Fig. 27 - Point P in the rendered quad Q corresponds to sample P' in the mapped quad Q'.

Example

How to use the presented methods shall be clarified with the example of a reduction function. The purpose is to find the minimum value within a large quantity. The values are stored in a 9×9 matrix (see figure 28).

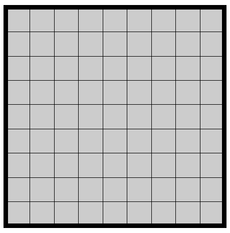


Fig. 28 - 9×9 matrix of which the minimum value is evaluated.

The algorithm computes the overall minimum value by iteratively evaluating the minimum value of 3×3 submatrices (see figures 29 and 30). Determining the minimum value of a submatrix is done by a fragment program. With one pass, the matrix is reduced by factor 9. This is done until a matrix of size 1×1 is reached. Therefore, $\log_9(m \times n) = \log_9 81 = 2$ passes are necessary to complete the example, where m is the width of the matrix and n is the height. In case of a GPU with 16 pipelines, only nine pipelines would be used in the first pass and only 1 pipeline in the second. Thus, this example is too small to be able to make optimal use of the parallelism.

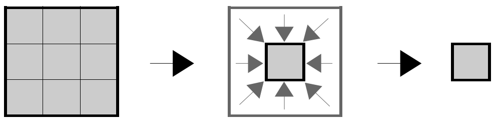


Fig. 29 - The reduction is done by calculating the minimum of 3×3 submatrices.

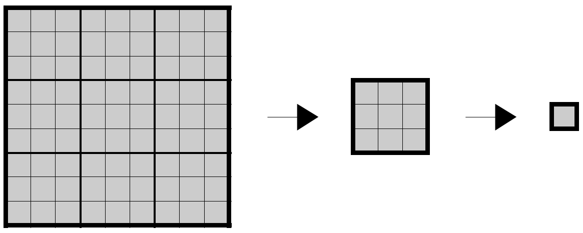


Fig. 30 - With this technique two passes are needed to complete 9×9 matrix.

Buffers can only be created with a size of $2^a \times 2^b$. It is assumed that a 16×16 buffer exists containing the 9×9 matrix as shown in figure 31. This buffer is used as input texture. The render buffer that is still empty has the same size.

For each pass, a quad Q is drawn into the render buffer with border lengths of a third of the matrix in the texture buffer. Q is mapped to the corresponding coordinates Q' of the texture. The fragment program is then executed for each pixel in the render buffer that is covered by the quad. If there is a following pass, the render buffer is bound as texture whereas the texture buffer is used as render target. The result can be found in the final 1×1 matrix.

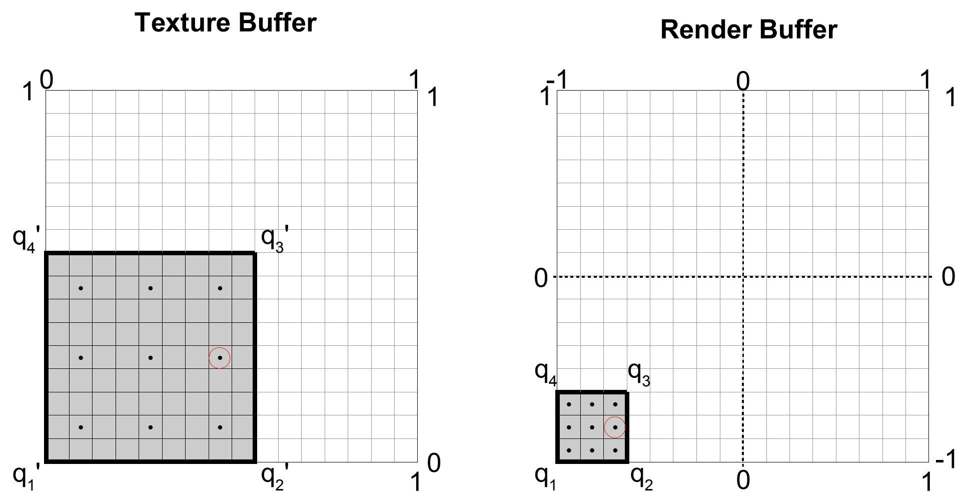


Fig. 31 - Quad Q = {q_1, q_2, q_3, q_4} is drawn into the render buffer and its coordinates are mapped to a quad Q' = {q_1', q_2', q_3', q_4'} in the texture buffer. Thus, the marked pixel in the render buffer corresponds to the marked sample point in the texture buffer.

How to implement these techniques on CPU and GPU side and how both work together is discussed in the following two chapters.

3.6 The 3D API

Introduction

Programming using a GPU consists of two parts: the CPU programming part and the GPU programming part. This chapter deals with the CPU side of the graphics pipeline: the application and the 3D API that is used to control the GPU on a high abstraction layer.

The interface between software and hardware is the hardware driver which is a piece of software as well (Fernando et al. 2004). The driver is hardware specific and contains information and functions to interact with the hardware. There is one more abstraction layer between application and hardware driver, the API. It makes programming independent from hardware specifications (see figure 32). Hardware and driver can therefore be changed without the need for recompiling a program as long as the new hardware supports the features requested by the application. In case of graphics applications, it is referred to APIs that are optimized for 3D graphics: 3D APIs. The two available APIs are DirectX[10] developed by the Microsoft Corporation, and OpenGL[11] maintained by the OpenGL Architectural Review Board (ARB). All three parts dynamically link to each other at runtime.

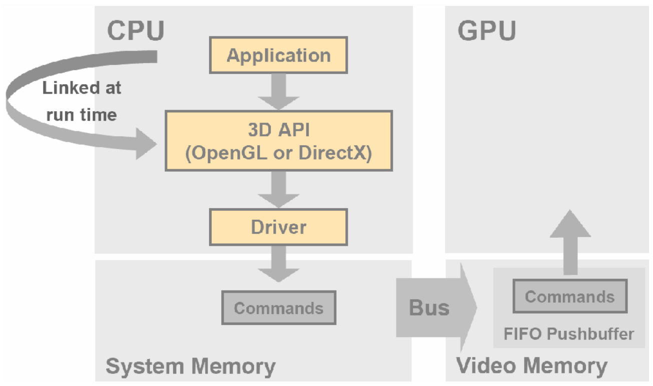


Fig. 32 - Layers of a graphics application: Application, 3D API, and Driver (Figure Courtesy of Cyril Zeller (2004)).

DirectX

Microsoft's API is C++ based and a new version is released about every year (Strzodka 2004, Buck 2004, Zeller 2004). It is compatible with the Microsoft Windows[12] operating system only. It uses window manager like Windows itself, GLUT, or Qt. It is very popular in the PC game industry as most PC games are written for Windows only. For shader programming, Microsoft's HLSL (High Level Shading Language), NVIDIA's and Microsoft's Cg, and DirectX pixel- and vertex-shader can be used. More on these languages can be found in chapter “Programming the GPU”. Many Tools and Shader Debugger for DirectX exist. Some selected can be found in chapter “Links”. The slow readback from GPU to CPU of about 50 MB/sec is a negative aspect. More information about DirectX can be found in the DirectX Documentation[13].

10 http://www.microsoft.com/windows/directx/
11 http://www.opengl.org/
12 http://www.microsoft.com/windows/
13 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/directx9cpp.asp

OpenGL

OpenGL is C based and does not evolve through complete reviews but through a system of OpenGL Extensions[14] (Strzodka 2004, Buck 2004, Zeller 2004). This system allows software developers to add and access technological innovations above and beyond the features specified in the official OpenGL standard, the so-called OpenGL core. If an extension proves itself, it can ultimately be integrated into the core. All known extension specifications can be found in the OpenGL Extension Registry[15] which is maintained by SGI. OpenGL is available for most common operating systems like Windows, Linux, Unix, MacOS, OS/2, and BeOS. It is very popular in the academic world and all non game–related graphics industries. Some fields of usage are Computer Aided Design, and scientific visualization. GPU programs can be written in GLSL, Cg, and as OpenGL fragment- and vertex-shader. Brook and Sh are available as streaming languages. The readback is faster than with DirectX, but there are also disadvantages like that float formats are specialized for ATI and NVIDIA. The specifications of OpenGL and GLUT can be found at the OpenGL website for OpenGL & utility library specifications[16].

The functionality of both DirectX and OpenGL is mainly equivalent. Both provide features like Render-To-Texture and multiple render targets. As OpenGL is used in the implementation part of this thesis, the focus lies on OpenGL and GLSL in the following.

Application Structure

A graphics application basically consists of two main parts: initialization and rendering loop (see figure 33) (Fernando 2004, Fernando et al. 2004, Zeller 2004, Lefohn 2004b).

The initialization sets up the API, checks hardware capabilities, and creates necessary resources like a window, buffers, and textures. It also determines display properties which are operating system specific like double buffering, pixel format, and windowed or fullscreen mode. For this purpose, a library like GLUT[17] can be used in addition to simplify this process and to hide the operating system specific part. Furthermore, geometry needs to be loaded or created and stored in index and vertex buffers.

For each frame, the rendering loop renders geometry to a buffer and displays the image on screen. Usually, double buffering is used in order to display animations without tearing. This technique uses two buffers: a front buffer which is shown on the screen and an

14 http://www.opengl.org/resources/features/OGLextensions/
15 http://oss.sgi.com/projects/ogl-sample/registry/
16 http://www.opengl.org/documentation/spec.html
17 http://www.opengl.org/resources/libraries/glut.html

invisible back buffer. The rendering loop first draws to the back buffer and then swaps both buffers to display the ready rendered frame. There are several methods for drawing complex geometry: display lists, vertex arrays (vertex array ranges and vertex array objects), and vertex buffer objects. General-purpose applications in contrast will rather use the immediate mode. In this case, each vertex of a triangle or quad that are drawn is defined manually.

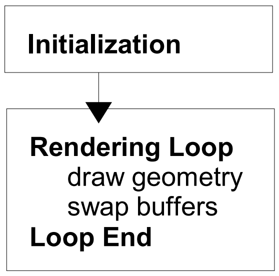


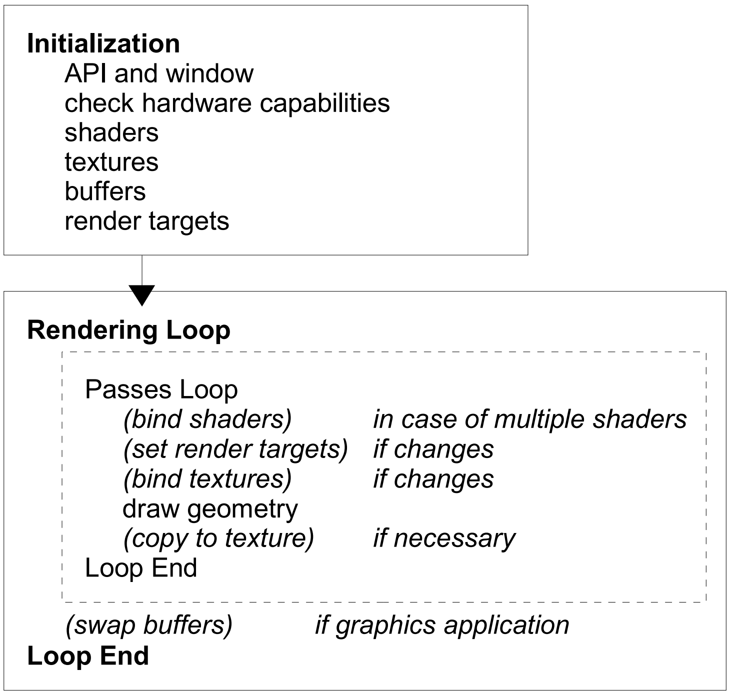
Fig. 33 - Simplified model of a graphics application with double buffering.

In case of a general-purpose application or any graphics application that uses general-purpose programming techniques, vertex shaders, fragment shaders, and render targets need to be initialized. Render targets are buffers in the video memory that are used to store intermediate images in case of a multipass rendering process. These can then be loaded as textures for a following pass. Multiple passes may be necessary due to structural reasons, if an algorithms needs severals steps, or because of hardware limitations. In graphics applications, multiple passes are necessary for 2D-postprocessing like fog, tone mapping, depth of field, and motion blur (Zeller 2004).

Rendering to an invisible buffer is called off-screen rendering. Several methods for off-screen rendering with multiple passes exist:

- CTT using window context buffers.
- RTT using pbuffers.
- FBO (Framebuffer Objects).

Figure 34 shows a simplified model of an application that uses off-screen rendering with multiple passes. The rendering consists of a second loop that handles all passes. If each pass uses a different shader, than the specific one has to be bound in each pass. The used textures and render targets have to be set for each pass as well if they change. The example of a reduction function shown in the last chapter uses only one shader, but render target and texture are exchanged with every pass.



```
Initialization
    API and window
    check hardware capabilities
    shaders
    textures
    buffers
    render targets

Rendering Loop
      Passes Loop
          (bind shaders)          in case of multiple shaders
          (set render targets)    if changes
          (bind textures)         if changes
          draw geometry
          (copy to texture)       if necessary
      Loop End
    (swap buffers)          if graphics application
Loop End
```

Fig. 34 - Simplified model of an application with multiple passes.

The following sections contain source code examples. Parts that have been left out for simplicity are marked with three dots "...". Most examples are taken from the implementation that was done together with this thesis. This is the reason why only compatibility to NVIDIA hardware is regarded. For tests, the NVIDIA Geforce 6800 GT was used. Furthermore, information was extracted from the OpenGL 2.0 Specification (Segal & Akeley 2004).

Copy-To-Texture

This example explains, how CTT can be used. The OpenGL framebuffer is a collection of logical buffers like color, depth, stencil, and accumulation buffers. A color buffer like the backbuffer can be defined as render target. Textures are loaded to the GPU as data input for a fragment program which processes the data and writes it to the back buffer. After rendering, the buffer content is copied into a texture so that it can be used as input for the next pass (see figure 35).

This method has many limitations. Using the window buffer means that the window size is automatically the maximum available buffer size and only 8-bit values can be used. The number of color buffers is limited to the front and back buffer (if double buffering is activated) plus auxillary and stereo buffers, in case that they are supported. In addition, CTT is very slow because a readback from the GPU memory (framebuffer) to the

CPU memory (texture variable) is very slow. In the following, it is explained how to implement this method.

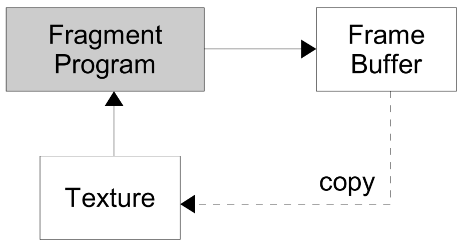


Fig. 35 - Copy-To-Texture model.

Initializing the API and Window

Using the GLUT library, initializing the window is as simple as in the following:

```
#include <GL/glut.h>
...
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_ALPHA | ...);
glutInitWindowSize(512, 512);
window = glutCreateWindow("MyApplication");
glutSetWindow(window);
```

An orthogonal projection is chosen for rendering with the screen limits {left, right, bottom, top} = {-1, 1, -1, 1} like cross hairs:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

gluOrtho2D(-1, 1, -1, 1);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Checking Hardware Capabilities

The OpenGL Extension Wrangler Library[18] (GLEW) is a cross-platform C/C++ library for determining which OpenGL, WGL, and GLX extensions are supported. The first method is to check for globally defined variables that are `true` if an extension exists. The variables name is created by the following pattern: `GLEW_{extension_name}` and `WGLEW_{extension_name}`.

```
if (GLEW_ARB_vertex_program)
{
  /* It is safe to use the ARB_vertex_program extension here. */
  ...
}
```

The second method is slower and checks for a string with the extension name:

18 http://glew.sourceforge.net/

```
if ( glewGetExtension("GL_ARB_fragment_program") &&
    wglewGetExtension("ARB_buffer_region")       )
{
  /* Extensions are supported. */
  ...
}
```

Shaders

If no vertex or fragment shader[19,20,21] is loaded, the fixed function equivalent is used. Shaders can be loaded as follows:

```
/* create program object */
GlhandleARB hProgram = glCreateProgramObjectARB();

/* create shader object and load, compile, attach, and link shader
*/
GlhandleARB hShader  = glCreateShaderObjectARB
(GL_FRAGMENT_SHADER_ARB);

const char *programCode = "...“; // shader source code
glShaderSourceARB (hShader, 1, &programCode, NULL);
glCompileShaderARB(hShader);
glAttachObjectARB (hProgram, hShader); // more shaders can be added
glLinkProgramARB  (hProgram);

/* check if linking was successful */
GLint progLinkSuccess;
glGetObjectParameterivARB(hProgram, GL_OBJECT_LINK_STATUS_ARB,
                          &progLinkSuccess);
if (!progLinkSuccess){
  /* Shader could not be linked */
  ...
}
```

Activating and deactivating is done like this:

```
glUseProgramObjectARB(hProgram); // activate
... /* draw geometry */
glUseProgramObjectARB(0);         // deactivate
```

The program is deleted with:

```
glDeleteObjectARB(hProgram);
```

Shaders can be loaded as character string or as precompiled binary. The chapter “Programming the GPU” discusses the implementation of shaders.

Shader Input

Shader input can be vertex attribute variables that are defined per-vertex like normals or uniform variables that are used as a global constants like a texture. The different types of inputs and their meanings are described in the chapter “Programming the GPU”.

19 http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_shader.txt
20 http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_shader.txt
21 http://oss.sgi.com/projects/ogl-sample/registry/ARB/shader_objects.txt

User-defined uniform variables are set for an active shader before drawing geometry (glBegin... glEnd). It is assumed that a vector with 3 components is used as uniform input for a shader. This variable is declared in the shader as:

```
uniform vec3 inputVector;
```

First, it is important to get the location of the variable in the shader program, then values can be send to this location.

```
// get variable location
int inputVectorloc = glGetUniformLocationARB(hProgram,
“inputVector”);

// define values
glUniform3fARB(inputVectorloc, 1.0f, 4.0f, 3.0f);
```

The last function, loading the values, can only be used for float vectors with three components. This function is specific for each vector or matrix type. The naming of the function is defined by the patterns:

```
void glUniform{1234}{if} ( int location, T value );
void glUniform{1234}{if}v( int location, sizei count, T value );
void glUniformMatrix{234}{f}v( int location, sizei count,
                             boolean transpose, const float
*value );
```

For loading samplers, only the following pattern can be used:

```
void glUniform1i{v}( int location, {sizei count,} T value );
```

It works accordingly for user-defined attribute variables. First, the location of the variable needs to be queried, then the values are loaded. In this case, the values are loaded when the geometry is drawn (glBegin...glEnd).

```
int glGetAttribLocation( uint program, const char *name);
```

The values are then loaded for each vertex with functions of the following patterns:

```
void glVertexAttrib{1234}{sfd} ( uint index, T values );
void glVertexAttrib{123} {sfd}v( uint index, T values );
void glVertexAttrib4{bsifd ubusui}v( uint index, T values );
```

The following example draws three vertices with a float vector `inputVector` consisting of 3 components as attribute variable.

```
int inputVectorLoc = glGetAttribLocationARB(my_program, “inputVec-
tor”);

glBegin(GL_TRIANGLES);
      glVertexAttrib3f(inputVectorLoc, 1.0f, 2.0f, 3.0f);
      glVertex3f(-1.0f, -1.0f, -0.5f);

      glVertexAttrib3f(inputVectorLoc, 4.0f, 5.0f, 6.0f);
      glVertex3f( 1.0f, -1.0f, -0.5f);

      glVertexAttrib3f(inputVectorLoc, 7.0f, 8.0f, 9.0f);
      glVertex3f(-1.0f,  1.0f, -0.5f);
glEnd();
```

Textures

The following lines create a 32-bit 2D RGBA texture (8 Bit for each channel):

```
/* create texture ID */
GLuint iTexID;

glActiveTexture(GL_TEXTURE0  );
glEnable       (GL_TEXTURE_2D);
glGenTextures  (1, &iTexID   );

/* bind texture */
glBindTexture(GL_TEXTURE_2D,  iTexID);

/* set sampling parameters */
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S    , GL_CLAMP  );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T    , GL_CLAMP  );
```

The texture is created with the same size as the window: 512×512. The texture width and height must be a power of 2 each. It can be filled with any data or an image in form of a float array of size 4×512×512 whereby the elements have a value of 0 to 255 devided by 255. If no data shall be loaded, 0 can be used instead.

```
/* load data array */
float* data = new float(4*512*512);
... /* fill data */

/* fill texture with data array */
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 512, 512,
             0, GL_RGBA, GL_FLOAT, data);
```

To use the texture as input for the fragment program, it needs to be defined as uniform input. Assuming the shader program has a texture variable declaration `uniform sampler2D inputTex`, `iTexID` can be loaded with

```
GLint iTexLoc = glGetUniformLocationARB(hProgram, “inputTex”);
glUniform1iARB(iTexLoc, iTexID);
```

Setting the Render Target

In this example, the back buffer shall be used as framebuffer:

```
glDrawBuffer(GL_BACK_LEFT);
```

Rendering

A full screen quad is drawn in a distance of 0.5f in immediate mode. According to the initialization with `gluOrtho2()`, `the` full screen coordinates for the vertices are lower left (-1, -1), lower right (1, -1), upper right (1, 1), and upper left (-1, 1). The texture coordinates are defined with normalized coordinates in form of floating point values between 0 and 1 (see figure 31 in the chapter “Mapping Computational Concepts to the

GPU"). Thus, the according values for 1:1 mapping are lower left (0, 0), lower right (1, 0), upper right (1, 1), and upper left (0, 1).

```
glBegin(GL_QUADS);
{
  glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
  glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
  glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
  glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
}
glEnd();
```

It is important that the texture coordinates have to cover all required texels. Texels are the textures pixels. In case of a 2×2 texture, the texture coordinates for a quad covering the leftern two pixels are (0, 0), (0.5, 0), (0.5, 0.5), and (0, 0.5).

Everything between `glBegin()` is pushed into a FIFO (first-in/first-out) buffer and rendered after `glEnd()`. The following command can be added to wait until the rendering process is completed :

```
glFinish();
```

The result will be in the back buffer then.

Copy-To-Texture

In order to use the back buffer content for the next pass, it is copied into the texture. Therefore, it is defined which buffer shall be read:

```
glReadBuffer(GL_BACK_LEFT);
```

Then the buffer content is copied to the active bound texture:

```
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, 512, 512);
```

Reading the Result

In case of general-purpose programming, the complete framebuffer or a part of it needs to be read back into an array to make the result available for further CPU computation. This works similar to copying the framebuffer to a texture, but in this case the target is an array. The whole buffer can be read back like this:

```
float* result = new float(4*512*512);
glReadPixels(0, 0, 512, 512, GL_RGBA, GL_FLOAT, result);
```

Swapping the Buffers

In graphic applications the rendered image is shown after all passes have been rendered. Therefore, front and back buffer need to be swapped:

```
glutSwapBuffers();
```

Render-To-Texture

RTT[22] was state-of-the-art until the framebuffer object extension was released on 31st of January 2005. RTT uses a buffer system called pbuffers[23] (pixel buffers) which is available only under windows yet. A cross-platform solution for RTT[24] is under development. Pbuffer subbuffers can be used each either as framebuffer or as texture. Therefore, the rendered result of a pass can be bound as texture in a following pass like shown in figure 36.

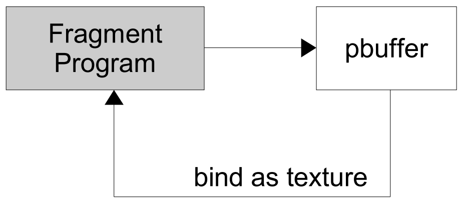


Fig. 36 - Render-To-Texture model.

RTT is much faster than CTT, but it has some disadvantages as well (Harris 2005): switching between different pbuffers is slow, depth buffers cannot be shared between pbuffers, RTT does not work with multi-sample anti-aliasing which is very important for games. A big advantage is that buffer size and pixel format[25] are independent from the current display mode. Sizes bigger than the window size are possible. Furthermore, 128-bit floating point colors can be used on NVIDIA hardware like NV40 and 96-bit on ATI hardware like R420. That means 32-bit and 24-bit floating point precision.

A pbuffer is an off-screen pixel buffer that has the available standard properties of the on-screen buffer like front-, back-, stereo-, aux-, and stencil buffer. Each pbuffer has its own device context. Before being able to use a pbuffer, it must be switched from the window device context to a pbuffer device context. Switching between pbuffers means switching contexts as well, which is quite slow (see figure 37).

22 http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_render_texture.txt
23 http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pbuffer.txt
24 http://www.opengl.org/resources/features/GL_EXT_render_target.txt
25 http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_pixel_format.txt

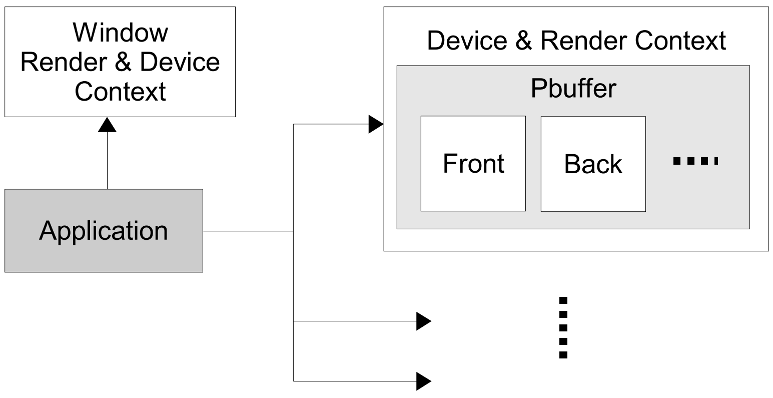


Fig. 37 - Pbuffer model. A pbuffer contains the standard buffers, but it needs its own device and render context. Thus, if an application uses several pbuffers it needs to provide several contexts as well and has to switch between them.

According to Wynn (2001), the usage of pbuffers has four phases:

1. Pbuffer extension initialization
2. Pbuffer creation
3. Pbuffer binding
4. Pbuffer destruction

All four phases for creating an 1024×1024 RGBA pbuffer with 32-bit floating point precision, double buffer and auxiliary buffers on NVIDIA hardware are explained in the following.

Pbuffer Extension Initialization

The support of the required extensions can be checked with GLEW as described above.

- WGL_ARB_pixel_format
- WGL_ARB_pbuffer
- WGL_NV_render_texture_rectangle[26]

Pbuffer Creation

First, the current window device context and rendering context are stored.

```
HDC hWindowDC = wglGetCurrentDC();
if (NULL == hWindowDC) wglGetLastError();

HGLRC hWindowRC = wglGetCurrentContext();
if (NULL == hWindowRC)  wglGetLastError();
```

26 http://oss.sgi.com/projects/ogl-sample/registry/NV/render_texture_rectangle.txt

The creation of a pbuffer consists of first choosing a pixel format and then creating the buffer. The pixel format can be initialized like this:

```
int          format  = 0;
unsigned int nformats;

/* generate pixel format attributes list */
GLint *attribList = new Glint[50];
GLint *ap         = attribList;

*ap++ = WGL_RED_BITS_ARB;        *ap++ = 32;
*ap++ = WGL_GREEN_BITS_ARB;      *ap++ = 32;
*ap++ = WGL_BLUE_BITS_ARB;       *ap++ = 32;
*ap++ = WGL_ALPHA_BITS_ARB;      *ap++ = 32;
*ap++ = WGL_STENCIL_BITS_ARB;    *ap++ = 0;
*ap++ = WGL_DRAW_TO_PBUFFER_ARB; *ap++ = true;
*ap++ = WGL_SUPPORT_OPENGL_ARB;  *ap++ = true;
*ap++ = WGL_DOUBLE_BUFFER_ARB;   *ap++ = true;
*ap++ = WGL_PIXEL_TYPE_ARB;      *ap++ = WGL_TYPE_RGBA_ARB;
*ap++ = WGL_FLOAT_COMPONENTS_NV; *ap++ = true;
*ap++ = WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV;
                                 *ap++ = true;
*ap++ = WGL_AUX_BUFFERS_ARB;     *ap++ = 1;
*ap++ = 0;

/* choose pixel format */
if(!wglChoosePixelFormatARB(hWindowDC, attribList, NULL,
                            1, &format, &nformats      ) )
{
   /* wglChoosePixelFormatARB() failed */
   wglGetLastError();
   ...
}

if (nformats == 0)
{
    /* no pixel formats were found\n");
    ...
}
```

If a pixel format was found, the pbuffer can be created. For 32-bit support the texture target is not WGL_TEXTURE_2D but WGL_TEXTURE_RECTANGLE_NV. Accordingly, textures need to be created in this format as well. More information is provided be below in section “Textures”. Note, that the pbuffer width and hight must be a power of 2.

```
/* clear attribute list */
ap  = attribList;
*ap = 0;

/* generate pbuffer attributes list */
*ap++ = WGL_TEXTURE_FORMAT_ARB;  *ap++ = WGL_TEXTURE_FLOAT_RGBA_NV;
*ap++ = WGL_TEXTURE_TARGET_ARB;  *ap++ = WGL_TEXTURE_RECTANGLE_NV;
*ap++ = 0;

/* create pbuffer */
HPBUFFERARB hPbuffer = wglCreatePbufferARB(hWindowDC, format,
                                           1024, 1024, attribList);
if (hPbuffer == NULL)
```

```
{
    /* creating pbuffer failed */
    ...
}

delete attribList;
```

In the last step the device context must be created.

```
HDC hDC = wglGetPbufferDCARB(hPbuffer);
if (hDC == NULL) { /* no device context */ ... }

HGLRC hRC = wglCreateContext(hDC);
if (hRC == NULL) { /* rendering context */ ... }

if (!wglShareLists(hWindowRC, hRC))
{ /* cannot share data between rendering contexts */ ... }
```

If these steps succeeded, a valid pbuffer is available.

Pbuffer Binding

Binding the pbuffer makes its GL rendering context the current context that will interpret all OpenGL commands and state changes.

```
wglMakeCurrent(hDC, hRC)
```

Pbuffer Destruction

The pbuffer is destructed by destroying all related resources.

```
WglDeleteContext      (hRC           );
wglReleasePbufferDCARB(hPbuffer, hDC);
wglDestroyPbufferARB  (hPbuffer      );
```

To restore the window device context, the stored variables are used.

```
wglMakeCurrent(hWindowDC, hWindowRC)
```

Handling a Display Mode-Switch

The memory associated with a pbuffer is not guaranteed to remain valid when a display-mode switch occurs (Wynn 2001). To ensure the validity of a pbuffer, the following function can be used:

```
int flag = 0;
wglQueryPbufferARB(hPbuffer, WGL_PBUFFER_LOST_ARB, &flag);

if(flag > 0) { /* pbuffer is lost and must be recreated */ ... }
```

If a pbuffer was successfully bound, it can be used like the standard window framebuffer with some additional features. The ability to bind a pbuffer color buffer as texture makes RTT possible. Therefore, buffers can be used for rendering or bound as texture. Buffers can only be used either for writing or reading but never for both at the same time. Thus, it cannot be used as texture and as render target simultaneously.

Textures

As described above, GL_TEXTURE_RECTANGLE_NV is used for 32-bit support instead of GL_TEXTURE_2D. Textures have to be initialized accordingly. With using GL_TEXTURE_RECTANGLE_NV, it is neither necessary to use a texture width and height with values of power of 2, nor are the texture coordinates defined as normalized floating point values between 0 and 1 but as texel counts. Additional textures with other formats can be defined and used, too.

```
/* create texture ID */
GLuint iTexID;

glActiveTexture(GL_TEXTURE0);       // 0 if first texture
glEnable(GL_TEXTURE_RECTANGLE_NV);

// create render texture object
glGenTextures(1, &iTexID);
glBindTexture(GL_TEXTURE_RECTANGLE_NV, iTexID);

// Use NEAREST as the default texture filtering mode.
glTexParameteri( GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
glTexParameteri( GL_TEXTURE_RECTANGLE_NV, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
```

To use the texture as input for the fragment program, it needs to be defined as uniform input. Assuming the shader program has a texture variable declaration uniform sampler2DRect inputTex, iTexID can be bound with:

```
GLint iTexLoc = glGetUniformLocationARB(hProgram, “inputTex”);
glUniform1iARB(iTexLoc, iTexID);
```

If a GL_TEXTURE_RECTANGLE_NV is used in the application, a sampler2DRect has to be used in the shader.

Binding Buffers as Textures

In order to use a buffer as texture, it needs to be bound to an active texture first. In this case, this is GL_TEXTURE0.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE_NV, iTexID);

if(!wglBindTexImageARB(hPbuffer, GL_BACK_LEFT) )
{
  /* binding failed */
  ...
}
```

After the rendering process, the buffer can be released from the texture binding.

```
glActiveTexture(GL_TEXTURE0);
if(!(wglReleaseTexImageARB(hPbuffer, GL_BACK_LEFT) ))
```

```
{
  /* release failed */
  ...
}
```

Setting the Render Target

Setting the render target is done as described above for CTT. Buffers that are not bound as texture can be used as render target.

Rendering

The rendering with RTT works as with CTT except that texture coordinates are unnormalized (texel indexed) if a texture of type GL_TEXTURE_RECTANGLE_NV is used. In the following example, a rectengular 512×512 region in the center of the 1024×1024 buffer is defined for texture mapping. Any other size and location could be chosen as well, as long as it fits into the buffer.

```
glBegin(GL_QUADS);
{
  glTexCoord2f(255, 255); glVertex3f(-1, -1, -0.5f);
  glTexCoord2f(767, 255); glVertex3f( 1, -1, -0.5f);
  glTexCoord2f(767, 767); glVertex3f( 1,  1, -0.5f);
  glTexCoord2f(255, 767); glVertex3f(-1,  1, -0.5f);
}
glEnd();
```

After rendering, a new render target can be chosen and the previous one can be used as texture input.

Reading the Result

This is done as described for CTT.

Framebuffer Objects

The framebuffer object extension[27] EXT_framebuffer_object was released on 31st of January 2005 (Harris 2005). Actually, only beta drivers for developers are available from NVIDIA[28].

This buffer model is window system independent and it only requires a single OpenGL context, which is why no time consuming context switches are necessary. Furthermore, there is no need for complicated pixel format selection. The format of the framebuffer is determined by the texture or renderbuffer format. In addition, renderbuffer images and texture images like depth buffers can be shared among framebuffers to save memory.

27 http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt
28 http://developer.nvidia.com

Apart from that, this model is similar to the Direct3D render target model. This makes porting code between both systems easier.

An FBO is a collection of framebuffer-attachable images (FAI) plus a state that defines where the output of the GL rendering is directed (see figure 38) (Harris 2005). FAIs are 2D arrays of pixels like color, depth, and stencil buffer that can be attached (bound) to a framebuffer. For each attached image, an attachment point in the FBO exists which is simply a state that references such an image. There are two types of objects that can contain FAIs: texture objects and renderbuffer objects. A renderbuffer object (RBO) contains a single renderbuffer image (RBI) which is a 2D array of pixels (no mipmaps, cubemap faces etc.). These are used as renderbuffers for off-screen rendering and cannot be bound as texture. In contrast, a texture object contains several texture images that can be used as both texture and render target. Thus, it is these that are the RTT substitution. When an FBO is bound, its FAIs are the source and destination for fragment operations. With this system, a principle called framebuffer completeness comes along. For consistency of all attachments the following requirements must be fullfilled:

- Texture format and attachment point consistency. Example: a depth texture should not be bound as color attachment.
- All attached images must have the same width and height.
- All color attachments must have the same format.

Otherwise drawing geometry (glBegin) will generate an error[29].

Harris (2005) presents three ways of switching between FBO rendering destinations ordered by increasing performance:

1. Multiple FBOs
   - Separate FBO for each texture (render target).
   - FBOs are switched using BindFramebuffer().
     This can be two times faster than wglMakeCurrent() (in beta NVIDIA drivers).
2. Single FBO, multiple texture attachments
   - Textures have the same format and dimensions.
   - Only one color attachment is used as render target.
   - FramebufferTexture() is used to switch between textures.
3. Single FBO, multiple texture attachments
   - Textures are attached to different color attachments.

29 INVALID_FRAMEBUFFER_OPERATION

– glDrawBuffer() is used to switch rendering to different color attachments.

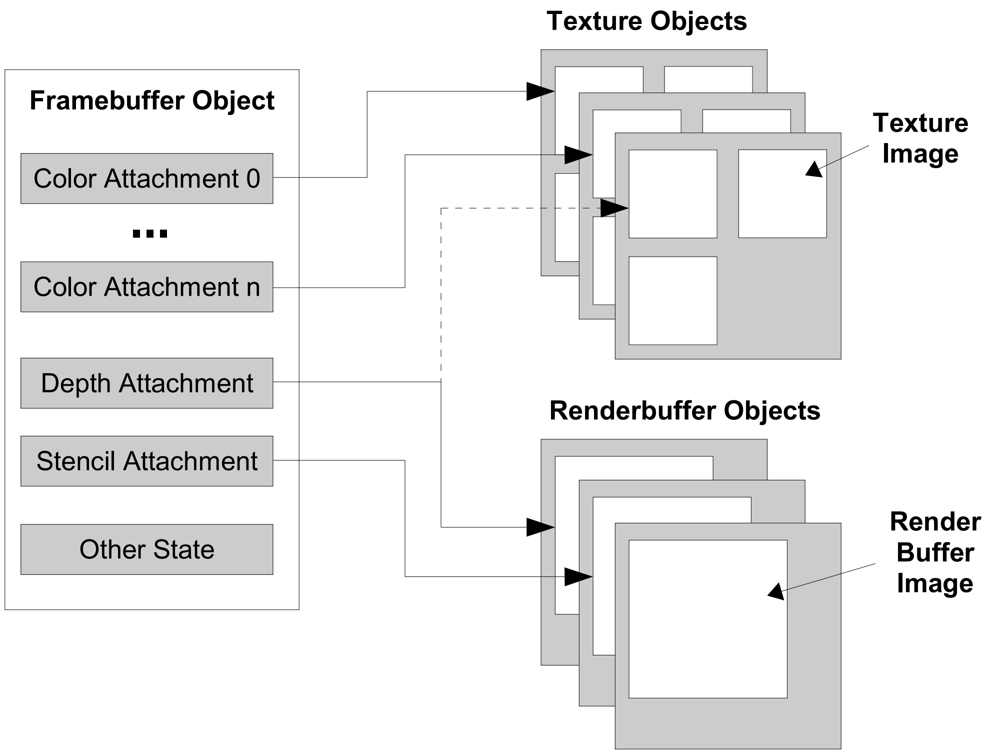


Fig. 38 - Framebuffer Object Architecture. This figure is adopted from Harris (2005).

Example

The source code of the following example was taken from Harris (2005):

```
#define CHECK_FRAMEBUFFER_STATUS() \
{ \
  Glenum status; \
  status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT); \
  switch(status) \
  { \
    case GL_FRAMEBUFFER_COMPLETE_EXT: \
      break; \
    case GL_FRAMEBUFFER_UNSUPPORTED_EXT: \
      // different formats */\
      break; \
    default: \
      /* programming error; will fail on all hardware */\
      assert(0); \
  } \
}

Gluint fb, depth_rb, tex;

// create objects
glGenFramebuffersEXT (1, &fb      ); // frame buffer
```

```
glGenRenderbuffersEXT(1, &depth_rb); // render buffer
glGenTextures        (1, &tex     ); // texture
glBindFramebufferEXT (GL_FRAMEBUFFER_EXT, fb);

// initialize texture
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA8, width, height,
              0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
...   // texture parameters here

// attach texture to framebuffer color buffer
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_2D, tex, 0);

// initialize depth renderbuffer
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
                         GL_DEPTH_COMPONENT24,
                         width, height);

// attach renderbuffer to framebuffer depth buffer
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                             GL_DEPTH_ATTACHMENT_EXT,
                             GL_RENDERBUFFER_EXT, depth_rb);
CHECK_FRAMEBUFFER_STATUS();
...

// render to the FBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
... // draw geometry

// render to the window, using the texture
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0  );
glBindTexture       (GL_TEXTURE_2D     , tex);
```

Multiple Textures

If more than one texture is used as input for a shader, the texture coordinates are defined with `glMultiTexCoord2f()` instead of `glTexCoord2f()`:

```
glBegin(GL_QUADS);
{
  glMultiTexCoord2f (GL_TEXTURE0, ... , ... );  // first texture
  ...
  glMultiTexCoord2f (GL_TEXTUREn, ... , ... );  // last texture
  glVertex3f( ... , ... , -0.5f);

  ... // more vertices
}
glEnd();
```

Multiple Render Targets

How to use multiple render targets is specified in the ARB_draw_buffers extension[30]. If

30 http://oss.sgi.com/projects/ogl-sample/registry/ARB/draw_buffers.txt

multiple render targets are used, the fragment program does not use `gl_FragColor` but `gl_FragColor[i]`, where i is one of n specified render targets.

```
int n  = 2;  // number of render targets

/* generate pixel format attributes list */
enum *bufs  = new Glint[n];
enum *pbufs = bufs;

*pbufs ++ = GL_FRONT_LEFT;  // is gl_FragColor[0]
*pbufs ++ = GL_BACK_LEFT;   // is gl_FragColor[1]

glDrawBuffersARB(n, bufs);

delete [] bufs;
```

The constants describing the buffers may be `NONE`, `FRONT_LEFT`, `FRONT_RIGHT`, `BACK_LEFT`, `BACK_RIGHT`, and `AUX0` through `AUXn`, where n+1 is the number of available auxiliary buffers. It is also possible to use only `gl_FragColor` as color output in the fragment program. In this case, it has to be defined into which of the specified buffers is written. This is the command to specify that:

```
void glDrawBuffer(enum buf);
```

The parameter `buf` can be one of the symbolic constants shown in table 5.

Symbolic constant	**Front left**	**Front right**	**Back left**	**Back right**	**Aux i**
GL_NONE					
GL_FRONT_LEFT	•				
GL_FRONT_RIGHT		•			
GL_BACK_LEFT			•		
GL_BACK_RIGHT				•	
GL_FRONT	•	•			
GL_BACK			•	•	
GL_LEFT	•		•		
GL_RIGHT		•		•	
GL_FRONT_AND_BACI	•	•	•	•	
GL_AUXi					•

Table 5 - Arguments to DrawBuffer() and the buffers that they indicate.

Detailed online descriptions of the presented methods can be found in the OpenGL specification, in the respective OpenGL extensions, on developer websites of GPU vendors such as NVIDIA and ATI, and on other websites like ShaderTech[31] and GPGPU[32] that deal with general-purpose computation on GPUs. More links can be found in the "Links" addendum.

31 http://www.shadertech.com
32 http://www.gpgpu.org/

3.7 Programming the GPU

Introduction

The implementaion of the Smith-Waterman algorithm in chapter 4 involves a GPU program written in the high-level GPU programming language OpenGL Shading Language (GLSL) that is used in an OpenGL context. This chapter therefore gives an introduction into GPU programming languages, especially into GLSL. Only the GPU side of the application is regarded.

Before the advent of high-level shading languages, GPUs were programmed using assembly code (van der Linden 2004, Fernando 2004, Fernando et al. 2004). When graphics hardware became capable of executing shader programs with a length of thousands of instructions, programming became too complicated and high-level languages had to be developed. Programming with high-level languages like C has definite advantages: easier programming, easier code reuse, easier debugging.

The assembly code

```
...
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

can be described by the high level language Cg with

```
...
float3 cSpecular = pow(max(0, dot(Nf, H)), phongExp).xxx;
float3 cPlastic  = Cd * (cAmbient + cDiffuse) + Cs * cSpecular;
...
```

Actually, there are three main levels of languages:

- Low level:
  Using fragment program extensions based on assembler.

– High level:
  Shader languages based on C/C++.
– General-purpose programming languages:
  Languages based on C/C++ that hide graphics elements.

High-Level Languages

The features of high level languages come from three sides (Fernando et al. 2004):

1. Syntax and semantics are based on C/C++.
2. Concepts of offline shading languages like the Renderman Shading Language are incorporated.
3. The graphics functionality is based on the APIs OpenGL and DirectX.

Being optimized for graphics programming, they have a native support for vector types and vector operations like dot products, vector normalization, and matrix multiplies.

The three commonly used shading languages are Cg (Mark et al. 2003), HLSL (Microsoft 2003), and GLSL (Kessenich et al. 2004). HLSL is maintained by the Microsoft Corporation, Cg was developed in corporation of Microsoft and NVIDIA, and GLSL is developed by the OpenGL ARB. HLSL and Cg are roughly the same language whereas HLSL only compiles to DirectX shaders and Cg is able to handle both DirectX and OpenGL shaders. An introduction to GLSL being representative in its concepts for all three languages is given below under topic “GLSL”.

General-Purpose Programming Languages

High-level GPU programming languages are optimized for implementing shaders, but they do not assist the developer with controlling the GPU from the CPU with a 3D API (Buck 2004b). Developers need to have deep knowledge of the latest API versions as well as features and limitations of the used graphics hardware. Furthermore, general-purpose algorithms need to be implemented by using graphics primitives like colors and vertices or textures, buffers, and geometry. General-purpose programming languages are designed to avoid these problems and to make programming easier. They abstract programming from any graphics API and hardware specifications and limitations and they simplify common operations. Due to the API and hardware independence, programs are portable between different hardware and different APIs.

Brook for GPUs[33] (Buck et al. 2004) is a system for general-purpose computation on programmable graphics hardware that uses the GPU as a streaming coprocessor as described in chapter “Mapping Computational Concepts to the GPU”. It can be regarded as C with stream extension. Data are handled in streams whereby kernels are functions applied to streams. It is designed for cross platform programming and it supports ATI and NVIDIA hardware, OpenGL and DirectX, Windows and Linux. Brook et al. claim the generated code's efficiency to be within 80 % of a hand-coded GPU version. Figure 39 shows a relative performance comparison of a hand-coded Fast Fourier Transformation (FFT) GPU implementation with a Brook version. Both run on ATI hardware and the Brook version has almost the same performance. In contrast, the code generated by C results in much less performance on a Pentium 4 than the hand-coded assembly version of the FFT does.

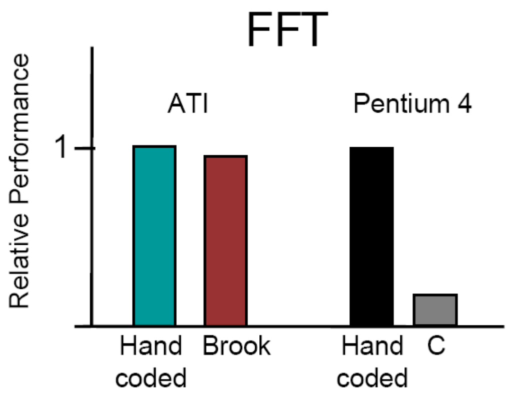


Fig. 39 - Comparison of hand coded FFTs to generated code (iigure courtesy of Ian Buck).

Sh[34] (McCool et al. 2002) is an embedded C++ library language for the dynamic metaprogramming of GPUs. It is developed by the University of Waterloo. It provides a high-level embedded programming interface for both shaders and general-purpose streaming programming. Thus, it is a streaming language as well. If an application is implemented in C++ with th Sh library both textures and parameters are handled by Sh. They simply need to be declare and can then be used immediately. Sh automatically uses pbuffers and ATI's uberbuffers. Furthermore, arrays are simulated with textures and textures can encapsulate interpretation code. Sh is Open Source and has a free, libpng license. All source is available, including the optimizer and backends.

33 http://brook.sourceforge.net
34 http://libsh.sourceforge.net

GLSL

The OpenGL Shading Language (GLSL) was developed for programming GPU shaders that are used in an OpenGL context. GLSL was used as shader language for the Smith-Waterman implementation, which is the last part of this thesis, because it was implemented using the OpenGL API. Therefore, the fundamentals and principles of its specification (Kessenich et al. 2004) shall be explained in the following.

Vertex and fragment programs have different specifications. Each has specific input and output variables. Since vertex processing is executed before fragment processing, it is possible to pass information from vertex to fragment shader only. Output values of the vertex shader therefore influence some input variables of the fragment shader. The shader input is classified in three types: uniform, attribute, and varying. The uniform qualifier can be used with both shader types and defines variables that are constant for all processed elements (vertices or fragments). Typical examples for uniform input are transformation matrices for vertex processing and textures for fragment processing. Variables qualified with attribute or varying in contrast are different for each primitive. Per-vertex input is described with the attribute qualifier whereas per-fragment input is described with varying. Per-vertex output like colors and texture coordinates that is used as per-fragment input is qualified with varying as well. The rasterizer generates varying input for the fragment processor by interpolating values from vertices that are part of the processed primitive (e.g. triangle). Thus, varying input that is used by the fragment processor must be set in the vertex processor. The figures 40 and 41 list the different in- and output variables that vertex and fragment processors can use. The in- and output consist of built-in variables that are always present and of user-defined variables that depend on the needs of each application. With Shader Model 3.0, both processor types can have texture input. The output of both contains built-in and user-defined variables as well, but also special variables. Special is no separate qualifier but this output is used for fixed pipeline computation and is therefore not used as input for further processing that can be influenced by the developer. Special output from the vertex processor is only used by the rasterizer whereas special output of the fragment processor is the graphics pipeline output and is written into its render targets. All output variables can be read, but they only have a defined value if any value has been assigned before.

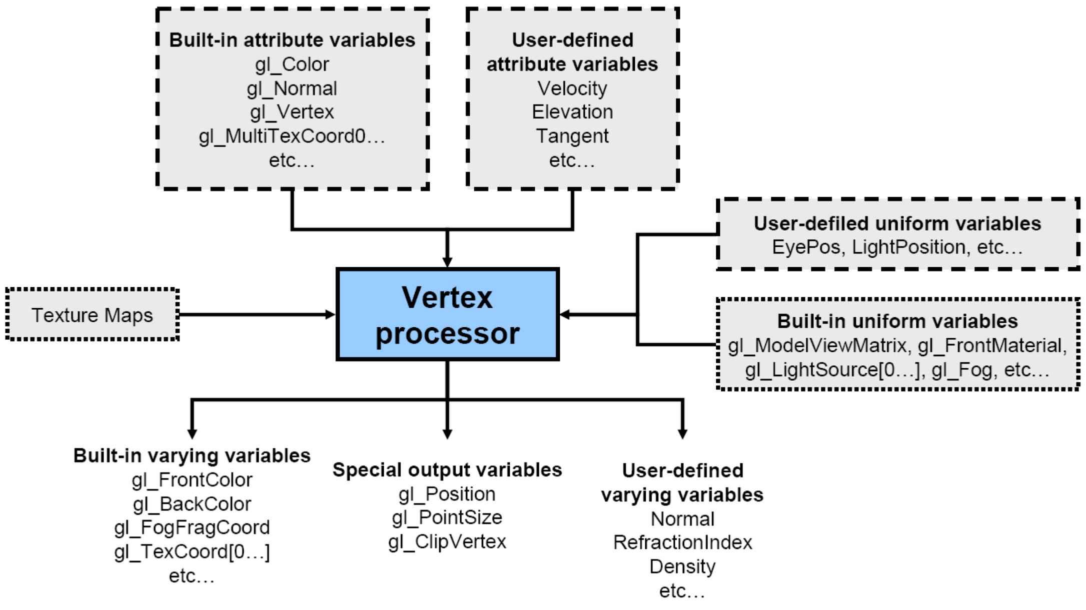

Built-in attribute variables
gl_Color
gl_Normal
gl_Vertex
gl_MultiTexCoord0…
etc…
User-defined attribute variables
Velocity
Elevation
Tangent
etc…
User-defiled uniform variables
EyePos, LightPosition, etc…
Texture Maps
Vertex processor
Built-in uniform variables
gl_ModelViewMatrix, gl_FrontMaterial, gl_LightSource[0…], gl_Fog, etc…
Built-in varying variables
gl_FrontColor
gl_BackColor
gl_FogFragCoord
gl_TexCoord[0…]
etc…
Special output variables
gl_Position
gl_PointSize
gl_ClipVertex
User-defined varying variables
Normal
RefractionIndex
Density
etc…

Fig. 40 - Vertex shader input and output variables. Image courtesy of Ho (2005).

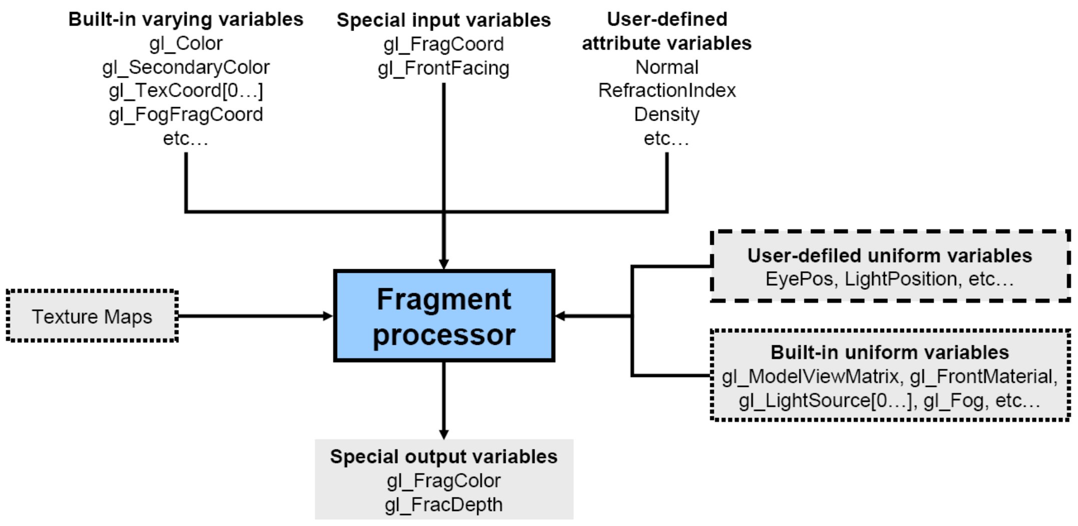


Fig. 41 - Fragment shader input and output variables. Image courtesy of Ho (2005).

The syntax of GLSL programs is closely oriented to C/C++ whereby a native support for vector and matrix types exists. Besides the basic types shown in table 6 structures (struct) and arrays are support as well, but no point,er types exist. The period or swizzle operator "." is used to access vector components and to generate new vectors. It is used by adding the operator to the variable name followed by components names. Three synonymous component name sets are described in table 6.

Component Name	**Description**
{x, y, z, w}	useful when accessing vectors that represent points or normals
{r, g, b, a}	useful when accessing vectors that represent colors
{s, t, p, q}	useful when accessing vectors that represent texture coordinates

Table 6 - Vector component names.

The swizzle operator is used to access components of vectors. Result of the swizzle access is a float if only one component is accessed and it is a vector if multiple components are selected:

```
vec4 v4;

vec4  a = v4.rgba; // is a vec4 and the same as just using v4,
vec3  b = v4.stp;  // is a vec3,
float c = v4.b;    // is a float,
vec2  d = v4.xy;   // is a vec2,
vec4  e = v4.xgba; /* is illegal - the component names do not
                      come from the same name set */
```

The swizzle operator can be used to assign one or more values to a vector whereby the order of components can be mixed:

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
pos.xw = vec2(5.0, 6.0);       // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0);       // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0);       // illegal - 'x' used twice
pos.xy = vec3(1.0, 2.0, 3.0);  // illegal - mismatch vec2 and vec3
```

Many optimized built-in functions including vector and matrix computations exist. Some sample functions are shown below.

```
float   dot (genType x, genType y)      // dot-product
vec3    cross (vec3 x, vec3 y)          // cross-product
genType normalize (genType x)           // normalization
genType reflect (genType I, genType N)  // vector I reflected by
                                        // surface orientation N
mat matrixCompMult (mat x, mat y)       // matrix multiplication
```

User-defined functions are possible as well and they are declared, defined and used very similar to C/C++:

```
// prototype
returnType functionName (type0 arg0, type1 arg1, ..., typen argn);

// definition
returnType functionName (type0 arg0, type1 arg1, ..., typen argn)
{
  ...
  return returnValue;
}
```

Arguments can be further specified with the qualifiers `in`, `out`, `inout`, and/or `const`. Due to the `out` qualifier it is possible to return more than one value. Further details are described in table 7. Recursion is not allowed in shaders and results in undefined behaviour. The only function that always needs to be defined is the main-function that is used as entry point for the shader. Not every shader needs to have a main function but at least one in a set of shaders that are linked together:

```
void main()
{
  ...
}
```

Type	Description
void	*for functions that do not return a value*
bool	*a conditional type, taking on values of **true** or **false***
int	*a signed integer*
float	*a single floating-point scalar*
vec2	*a two component floating-point vector*
vec3	*a three component floating-point vector*
vec4	*a four component floating-point vector*
bvec2	*a two component boolean vector*
bvec3	*a three component boolean vector*
bvec4	*a four component boolean vector*
ivec2	*a two component integer vector*
ivec3	*a three component integer vector*
ivec4	*a four component integer vector*
mat2	*a 2×2 floating-point matrix*
mat3	*a 3×3 floating-point matrix*
mat4	*a 4×4 floating-point matrix*
sampler1d	*a handle for accessing a 1d texture*
sampler2d	*a handle for accessing a 2d texture*
sampler3d	*a handle for accessing a 3d texture*
samplercube	*a handle for accessing a cube mapped texture*
sampler1dshadow	*a handle for accessing a 1d depth texture with comparison*
sampler2dshadow	*a handle for accessing a 2d depth texture with comparison*

Table 7 - GLSL basic types.

The fundamental building blocks of the OpenGL Shading Language are:

- Statements and declarations
- Function definitions
- Selection (`if-else`)
- Iteration (`for`, `while`, and `do-while`)
- Jumps (`discard`, `return`, `break`, and `continue`).

They are roughly used like in C/C++.

Type Qualifier	**Description**
< none: default >	*local read/write memory, or an input parameter to a function*
const	*a compile-time constant, or a function parameter that is read-only*
attribute	*linkage between a vertex shader and OpenGL for per-vertex data*
uniform	*value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application*
varying	*linkage between a vertex shader and a fragment shader for interpolated data*
in	*for function parameters passed into a function*
out	*for function parameters passed back out of a function, but not initialized for use when passed in*
inout	*for function parameters passed both into and out of a function*

Table 8 - GLSL Type Qualifiers.

Textures are accessed by texture sampling. Regarding how the graphics pipeline is realized by graphics hardware, there are two kinds of texture sampling: with prefetch and without prefetch. If the sample coordinates are not dynamically evaluated in the fragment program, the sampling result is prefetched an can be accessed directly. With prefetching the texture lookup needs much less time, of course. An example for a prefetch situation is using the varying texture coordinates. In case of dynamic sampling, the sampling coordinates are not foreseeable and can not be prefetched. This is the case, if the texture coordinates are evaluated at runtime, e.g. when they result from a previous texture lookup. The following example shows both types combined in a fragment shader.

```
uniform sampler2D     texA;
uniform sampler2DRect texB;

void main(void)
```

```
{
  vec2 texCoord0 = gl_TexCoord[0].xy,  // foreseeable
       texCoord1;                      // defined by lookup

  texCoord1 = texture2D(texA, texCoord0).xy;

  // dynamic texture lookup and result return
  gl_FragColor = texture2DRect(texB, texCoord1);
}
```

The shown example uses the first texture lookup to determine the coordinates for the second lookup. It is important to mention that the `texture2D()` function uses normalized texture coordinates whereby `texture2dRect()` uses texel indices just like `glTexCoord2f()` as explained in chapter "The 3D API". The result is then returned as color to the fixed pipeline.

The presented concepts are the basis for understanding the GPU based part of the implemented algorithm. The scenario of the last example also occurs in the fragment program of the Smith-Waterman GPU implementation in chapter 4.

3.8 Efficient Parallel Computing on GPUs

Introduction

While programming shaders, basic features of GPUs and some basic principles of GPU programming should be considered to make the computation as efficient as possible. This chapter presents the basics of efficient GPU programming based on Woolley (2005), Harris (2004b), Harris (2005), and Fernando et al. (2004). Some of these techniques are used in the algorithm implementation in chapter 4.

Parallelism

Parallelism can be classified into small-scale and large-scale parallelism (Woolley 2005).

Small-scale parallelism means the native vector processing support of GPU processors. A vector operation is done in the same time as a scalar operation whereat a vector consists of up to 4 scalars. Packing scalars into vectors can reduce the amount of instructions. Furthermore, using the swizzle operator is faster than explicitly declaring variables. It should therefore be used as efficiently as possible.

```
vec2 a = vec2(1.0, 2.0);
```

```
// explicitly building a vector results in several assembly
// MOV instrucions
vec2 b = vec2(a.y, a.x);

// using the swizzle operator is native and therefore free
vec2 c = a.yx;
```

In addition, on both NVIDIA and ATI GPUs a multiply-and-add operation is executed as one instruction. Thus, looking out for this combination can reduce instructions as well. The following example uses these techniques to optimize a given instruction set. It was originally written by Goodnight et al. (2003). The following code can be optimized:

```
// two substractions, two multiplies and two multiply-and-adds
float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f),
                       params.x*center.y - 0.5f*(params.x-1.0f));

// four scalar additions
float4 neighbor = float4(center.x - 1.0f,
                         center.x + 1.0f,
                         center.y - 1.0f,
                         center.y + 1.0f);
```

This is the optimized code:

```
// one elementwise substraction
float2 offset = center.xy - 0.5f;

// one multiply-and-add
offset = offset * params.xx + 0.5f;

// one vector addition
float4 neighbor = center.xxyy + float4(-1.0f,1.0f,-1.0f,1.0f);
```

The amount of instructions in the first line was reduced from six to two. The more the vector nature of GPUs is utilized, the more efficient programs get.

Large-scale parallelism refers to the datalayout. If grids (textures) consist of less than four components, they can maybe be packed into one texture. In the ideal case, four grids are packed into one 4-layered grid as shown in figure 42. If data is processed in vectors, the required memory bandwidth, the amount of texture lookups, and the amount of other instructions can be be reduced. The three matrices H, E, and F needed for the Smith-Waterman algorithm are packed exactly like this.

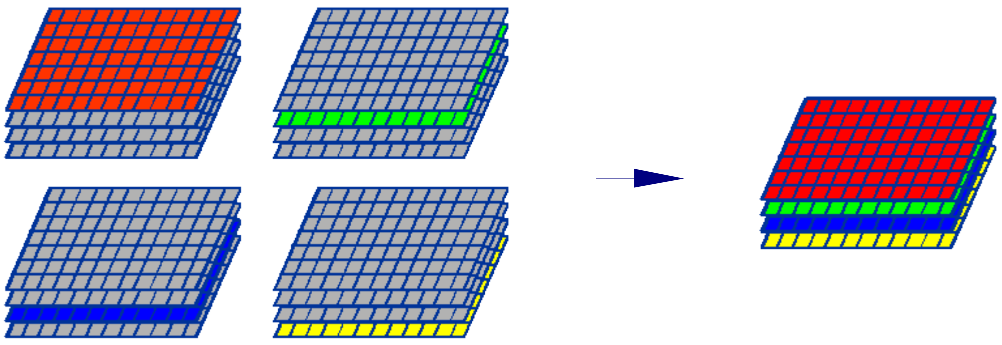


Fig. 42 - Efficient datalayout: four float grids can be packed into one 4-component grid (Figure courtesy Aaron Lefohn).

Precomputation

There are three stages where values can be computed: on the CPU, in the vertex program, and in the fragment program. Everything that can be computed in an earlier stage should be moved. Everything that is invariant for all computed primitives can be precomputed on the CPU and used as uniform input. This can be a scalar, a vector, or even a matrix as a lookup table. Functions that are computation intensive to evaluate can be precomputed, stored as matrix and loaded as uniform lookup table (texture). Complex per-vertex computation can maybe be precomputed and used as attribute input. As most computation is usually done on the fragment processor, everything that varies linearly across geometry like texture coordinates should be precomputed in the vertex shader. The rasterizer does the linear interpolation and makes the reslting values available for the fragment program. That reduces the amount of instructions. Additionally, the precomputation of texture coordinates enables the texture unit to prefetch data that is sampled in the fragment program to save time.

Branching

Using conditions (branching) can hurt performance. Modern GPUs like the NV40 are able to do SIMD branching in fragment shaders and fully MIMD branching in vertex shaders. In contrast, older GPU simulate branching by computing all branches and keeping only the desired value. SIMD branching should be used only if coherent regions containing at least 1000 pixels that take the same branch exist. But also in this case, the branching overhead should be regarded, because it still hurts performance. Branching should not be used for less than five instructions, but rather for early exits. If it is possible, it is better to move decisions up the pipeline or to replace a condition by math. A simple branch like

```
x = (t == 0)? a : b;
```

can be replaced with

```
x = lerp(t = {0,1}, a, b);
```

There are several techniques that can be used to move the decision up the pipeline: pre-computation, static branch resolution, z-cull, and occlusion queries. If definite areas are computed differently to others, not a branch in the fragment program but different programs should be used to avoid branching. A typical case is the boundary condition as shown in figure 43. The border can be evaluated by a different shader than the rest. This requires multiple passes by drawing multiple primitives.

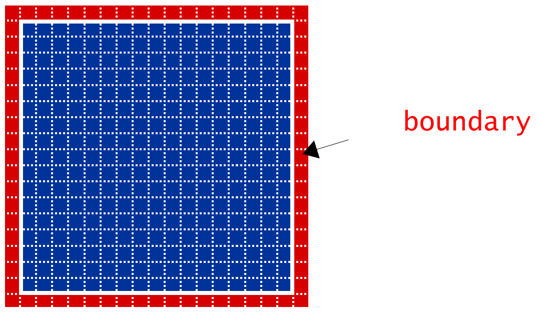


Fig. 43 - Boundary conditions are a typical case for static branch resolution.

Z-cull is useful for avoiding branches in multiple pass computations. First, the z-buffer is cleared to 1. Geometry is then drawn to z=0 and pixels that should be modified in later passes are discarded. In following passes, the depth test is enabled with e.g. `GL_LESS` to disregard pixels with a lower z-value than the drawn one. If geometry is then drawn to z=0.5, only those pixels with depth=1 are processed.

Optimizing

In a chain like the graphics pipeline, a system is only as fast as its slowest element. The slowest element is called a bottleneck. Optimizing the pipeline means iteratively finding and eliminating bottlenecks until the performance is acceptable.

Profiling tools like NVIDIA NVPerfHUD[35] for DirectX are a comfortable way to find out which shader needs most attention. One example tool for analyzing shaders is NVIDIA NVShaderPerf[36].

35 http://developer.nvidia.com/docs/IO/8343/How-To-Profile.pdf
36 http://www.developer.nvidia.com/page/tools.html

Optimizing always means to find the bottleneck first. Optimizing any other part of the pipeline does not improve performance, because the speed is still limited by the slowest part. The bottleneck can be found by systematically varying the workload of each element of a stage (see figure 44). This should be done backwards the pipeline, because the most workload usually lies with fragment processing. Elements to be varied in general-purpose applications can therefore be:

1. Fragment program instructions
   If reducing the amount of instructions speeds up the pipeline, the bottleneck is found.
2. Texture size
   If reducing the size of amount of texture speeds up the pipeline, the problem lies in the texture bandwidth.
3. Vertex program instructions
   If reducing the amount of instructions speeds up the pipeline, the bottleneck is found.

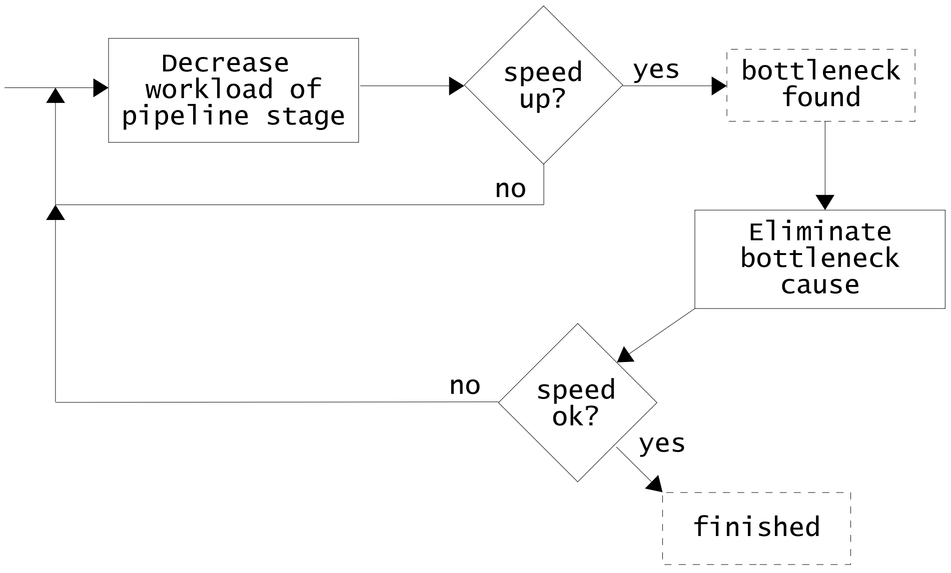

Decrease workload of pipeline stage
speed up?
yes
bottleneck found
no
Eliminate bottleneck cause
no
speed ok?
yes
finished

Fig. 44 - Iteratively testing for a bottleneck by decreasing the workload of pipeline stages one after another.

In graphics applications, more factors like the screen resolution and the amount of rendered vertices can be varied. It might also be the case that the real bottleneck is caused by driver overhead like context switching or simply by the CPU workload. It is therefore important to carefully check each single stage.

Optimization does not necessarily mean reducing workload. If the application has reached an acceptable performance, some stages might be unchallenged, meaning that their workload can be increased without decreasing the applications' performance. This can be used to make computations more precise or to add features.

3.9 Summary

This chapter gave an overview over general-purpose GPU programming regarding the aspects hardware, CPU-GPU analogies, APIs, GPU programming languages, and commonly used basic methods.

The part about hardware and CPU-GPU analogies showed that GPUs can be seen as streaming processors applying kernels to streams of independent elements. These streams flow through several processing stages called the graphics pipeline: application, vector processing, rasterization, and fragment processing. Each step applies kernels to an input stream and writes it to an output stream. The different stages are connected by these streams. The information is thereby transformed from geometry and textures to pixels. Textures are analogue to arrays on the CPU and drawing geometry is used for invoking computation. OpenGL and DirectX were presented and compared as 3D APIs. Focusing on OpenGL, it was shown how to control the GPU from the CPU and how GPU programs get feedback, supported by code examples. The basic principles for feedback are: Copy-To-Texture, Render-To-Texture with pbuffers, and framebuffer objects. GPU programming languages were discussed and the OpenGL Shading Language was presented in its basics to show the functionality. Finally, fundamental GPU programming rules were explained to implement shaders as efficiently as possible. This basically consists of using the GPU's native vector support, of using a packed data layout to safe bandwidth and of finding and eliminating bottlenecks in the pipeline by iteratively varying each step's workload.

This chapter provides the knowledge needed for the following presentation of a GPU-based implementation of the Smith-Waterman algorithm for local sequence alignment.

4 Smith-Waterman GPU Implementation

4.1 Motivation

On the one hand, there is an existing need for cheap and easy to use high performance systems that are able to handle huge amounts of information in a short time. On the other hand, commodity graphics hardware becomes more and more capable of doing parallelized general-purpose computation with a performance much higher than that of CPUs. In bioinformatics, the scanning of genetic databases for comparison purposes is a very computation intensive task that is commonly executed by expensive specialized hardware and/or PC clusters that need a lot of expertise to handle. In addition, algorithms are optimized and simplified to reduce computation time whereby accuracy decreases as well. Software tools like BLAST are used, because they show a high performance due to their use of heuristics. However, the Smith-Waterman algorithm is the standard for searches where a high accuracy is needed, although it is more computation intensive. Running this algorithm on parallel systems is an opportunity to speed up database searches by far. The following chapters explain how the algorithm was mapped to graphics hardware, how it was implemented, and it presents performance tests and results. Furthermore, possible improvements of the application are discussed.

4.2 Concept

Introduction

The concept of the Smith-Waterman algorithm implementation consists of three parts:

1. Mapping of the algorithm to a parallel system like graphics hardware.
2. Defining of the functionality of the application.
3. Defining test cases and evaluation.

The following sections describe each part.

Smith-Waterman Algorithm on Parallel Hardware

Primary task of the algorithm concept is to design the algorithm in a way that it provides maximum parallelism.

As shown in chapter 2.6, the Smith-Waterman algorithm compares two sequences A and B with lengths m and n, evaluates their similarity and generates their optimal local alignment. Therefore, a matrix H which incorporates every possible alignment is set up. Each cell $H_{i,j}$ depends on the neighbour cells $H_{i-1,j}$, $H_{i-1,j-1}$, and $H_{i,j-1}$ which need to have a concrete value (see figure 46). Using a CPU, it is possible to iterate over the matrix with nested loops row by row from top down or column by column from left to right.

Mapping this procedure to a parallel system by parallely computing an entire row or an entire column is not possible, because the evaluated cells depend on each other. Like shown in figure 45, computing randomly chosen cells of a row results in using cells that do not have a value yet. In this example, it is assumed that only four processors work in parallel. The evaluated cells are marked with a question mark.

In which way the matrix can be partially computed in parallel is given by the cell dependencies. Figure 46 shows that in the first iteration only cell [1, 1] fulfills the dependency. Every serial or parallel system starts at this point. In the second iteration the two cells [1, 2] and [2, 1] fulfill the dependency. If it is continued by evaluating all cells that fulfill the dependency, the resulting pattern is a diagonal that moves from upper left to the lower right of the matrix (see figure 47). Therefore, a parallel computation is only possible with diagonals. Since one row and one column with zeros need to be added, a $(m+1)\times(n+1)$ matrix is needed instead of an $m\times n$ matrix. The $m\times n$ submatrix contains the cells that need to be evaluated. The amount of necessary iterations is m+n-1 which corresponds to the amount of diagonals in the submatrix. In the example of figure 47, the amount of iterations is 13+9-1=22. The length of the rendered diagonal D_x is L(x) where x is the iteration count.

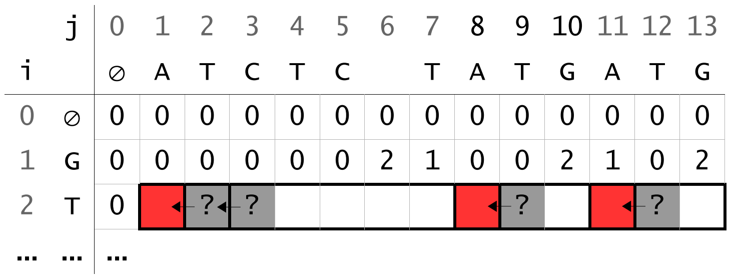


Fig. 45 - Parallel computation of lines or rows is not possible.

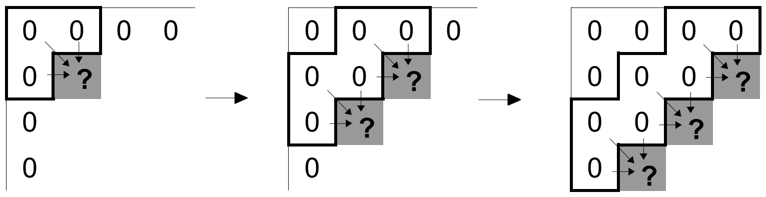


Fig. 46 - Only cells within anti-diagonals can be computed in parallel.

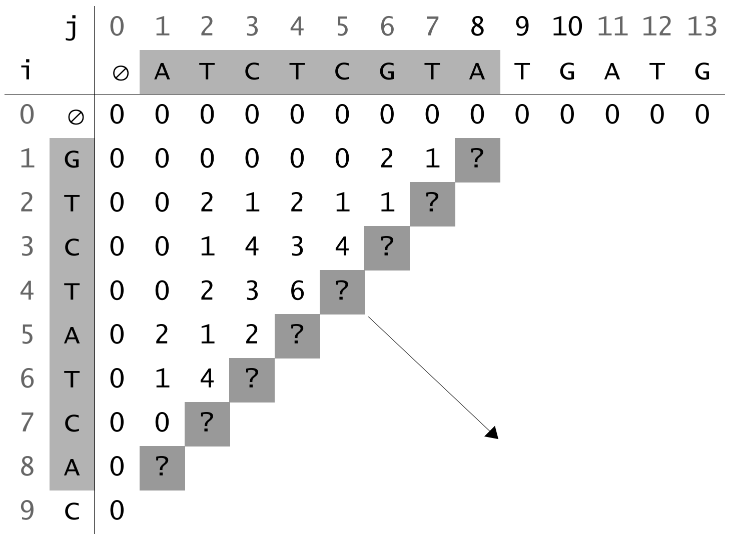


Fig. 47 - Anti-diagonals are computed along the diagonal of the matrix.

The overall basic idea is to compute the matrix in m+n-1 steps on the GPU, diagonal by diagonal. This is the only part of the algorithm that can be efficiently executed on a parallel system. The backtracking part is still performed on the CPU.

Application

The application controls the GPU rendering through a 3D API. It needs to have a rendering loop that invokes the computation. Furthermore, it has to load a fragment program from a file and to bind it. The information that is processed are nucleotide or amino acid sequences. The application must be able to load the query sequence and the sequence set to which the query sequence is compared to. Text files with FASTA format shall be used. For testing purposes the application shall also be able to generate random sequences. Since the algorithm needs any type of scoring matrix, a feature for loading text files containing a scoring matrix should be implemented as well. No graphical user-interface is necessary. It is sufficient to use the application as a command line tool. Controlling the application shall be solved by passing command line parameters. Results of the computation shall be those n sequences ordered by decreasing similarity score order which are most similar to the query sequence. The demanded functionality and structure is illustrated in figure 48.

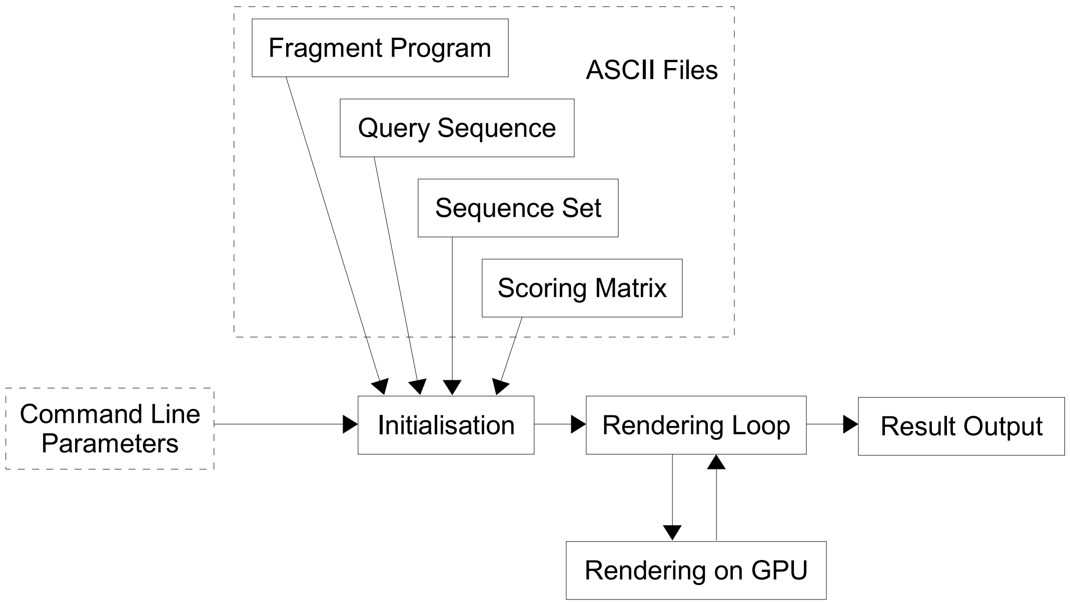


Fig. 48 - Application structure.

Test Cases and Evaluation

After implementation, the application can be tested for its performance. This can be done by running tests with random generated sequences and by scanning real databases.

Full control over the sequence length only exists with artificially generated sequences. These shall be used to test the performance development by varying sequence lengths and by varying the size of the sequence set that the query sequence is compared to. The maximum comparison speed can be determined like this.

Scanning real existent databases shows the application's performance in practical use cases. For this purpose, the Swiss-Prot database shall be scanned.

For evaluation, the results of performance test shall be visualized using graphs. Furthermore, the performance shall be compared to a reference CPU implementation of the same algorithm.

The following chapter describes the implementation that was realized according to these conceptual ideas.

4.3 Implementation

Application Structure

Besides the realization of the algorithm, the application was designed to fulfill the testing abilities proposed in the concept. Therefore, it has two execution modes: test mode and compare mode (see figure 49). For the test mode, many parameters such as query sequence length, length of the sequences in the sequence set, number of sequences in the set, step width for automatically increased parameters, and the number of repetitions per alignment can be specified. There are more parameters which are described in the appendix “SCA Program Arguments”. SCA means Sequence Comparison Algorithm and refers to the implementation that is presented in the following.

In test mode, a loop is entered (referred to as test loop) which invokes a number of sequence alignments in series, depending on the program arguments. The results of each alignment are gathered for statistics in an array. In this context, the alignment result regarding the sequence similarity is not of interest, but the performance is. Therefore, data such as render time and cell units per second (CUPS) is gathered. CUPS means the amount of matrix cells that are evaluated per second. It is possible to run several iterations for each alignment to get average values. The statistical data are written to text files and can be used for evaluation and visualization purposes.

The compare mode is simpler. The specified query sequence and the sequence set are loaded from files first. Subsequently, the alignment is evaluated whereas the number of iterations for averaging can be specified as well. Afterwards, those sequences with the highest similarity score are available as result. Render time and CUPS values are available, too.

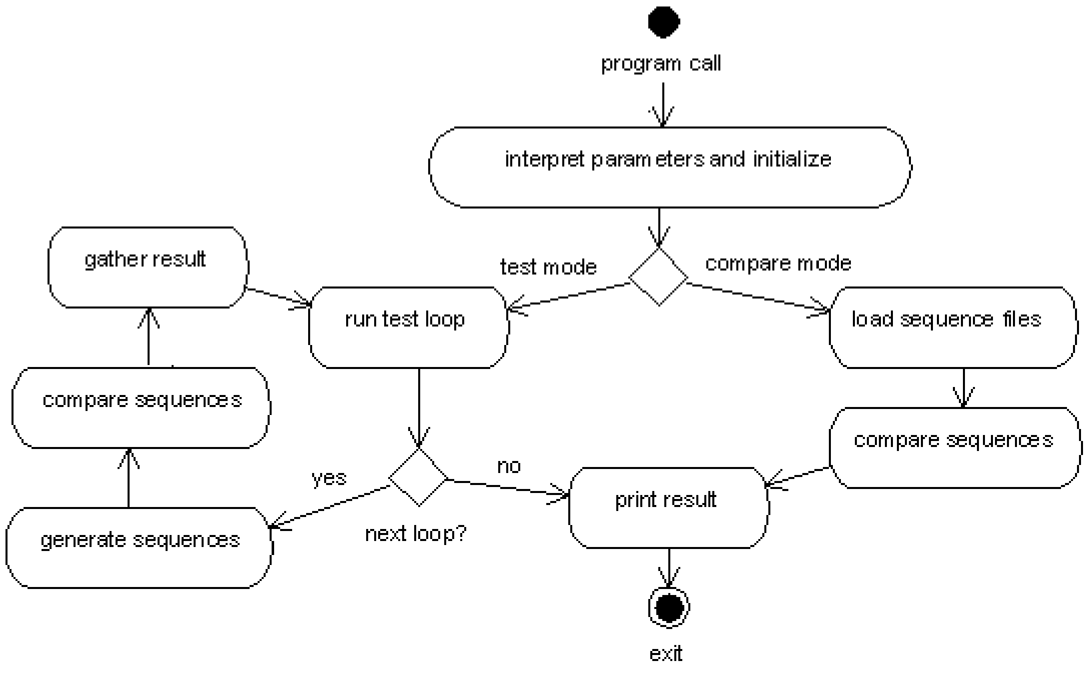


Fig. 49 - Activity diagram.

Figure 50 shows how the application was realized. The `SCA_Main` class manages the program operations. It is responsible for interpreting user-defined parameters, it initializes the window, it manages the two different run modes, it starts the sequence alignment and it prints the results to the standard output and to files. GLUT is used to initialize the window and GLEW is used for OpenGL extension queries.

The `Tokenizer` splits the program argument string into tokens and returns a token vector that can be used by the main program to interpret the arguments. First, the entire string is split into elements separated by spaces, then these elements are split at the position of distinct separators (see figure 51). Each token consists of a number that indicates whether a single parameter or a parameter-value pair is stored. Both parameter and value are stored as strings. Furthermore, the `Tokenizer` provides a function to load a scoring matrix file into a float array.

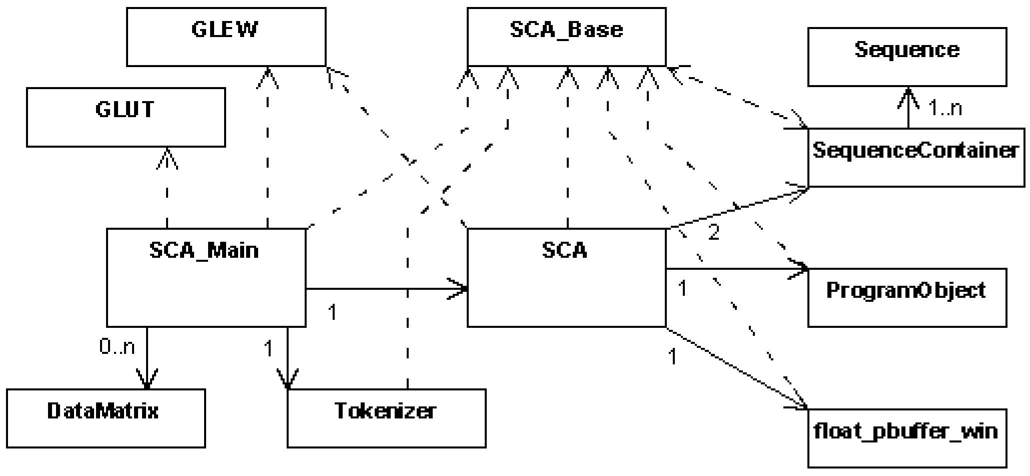


Fig. 50 - Class diagram.

The functionality of the `DataMatrix` class is similar to the `std::vector` except that it is designed like a matrix and that it has additional functionality. It is used by the main class to store alignment results of the test loop. Furthermore, it provides a function to print itself to an output stream like a file.

`SCA` is the class that incorporates the sequence alignment functionality. After creating an `SCA` object, it needs to be initialized. Required parameters like the filenames for the query sequence, the sequence set, the fragment program, and the scoring matrix are passed to the `initialize()` function (see figure 52). The initialization of the sequences depends on the mode that was chosen. In compare mode, the sequences are loaded from files whereas the sequences in test mode are created with random content. Sequences are kept in a `SequenceContainer` object. This object contains a vector with `Sequence` objects. In case of the query sequence, it is only one object. In case of the sequence set, it depends on the set's number of sequences. Each sequence object contains both the FASTA description line and the sequence itself. The pbuffer is prepared for RTT with 32-bit floating point precision and auxiliary buffers as described in the chapter “The 3D API”. The functionality is encapsulated in a pbuffer class that was taken from the NVIDIA software development kit[37] code example “Simple Float Pbuffer”[38]. This class was modified to fit the requirements. Pbuffer width and height must be a power of two each. These values are limited by the graphics hardware. 2048×2048 was the maximum pbuffer size that could be allocated using a NVIDIA GeForce 6800 GT graphics card. After setting the OpenGL projection mode, the textures are initialized. One texture for each the query sequence and the sequence set are created and filled with

37 http://developer.nvidia.com/object/sdk_home.html
38 http://download.developer.nvidia.com/developer/SDK/Individual_Samples/DEMOS/OpenGL/simple_float_pbuffer.zip

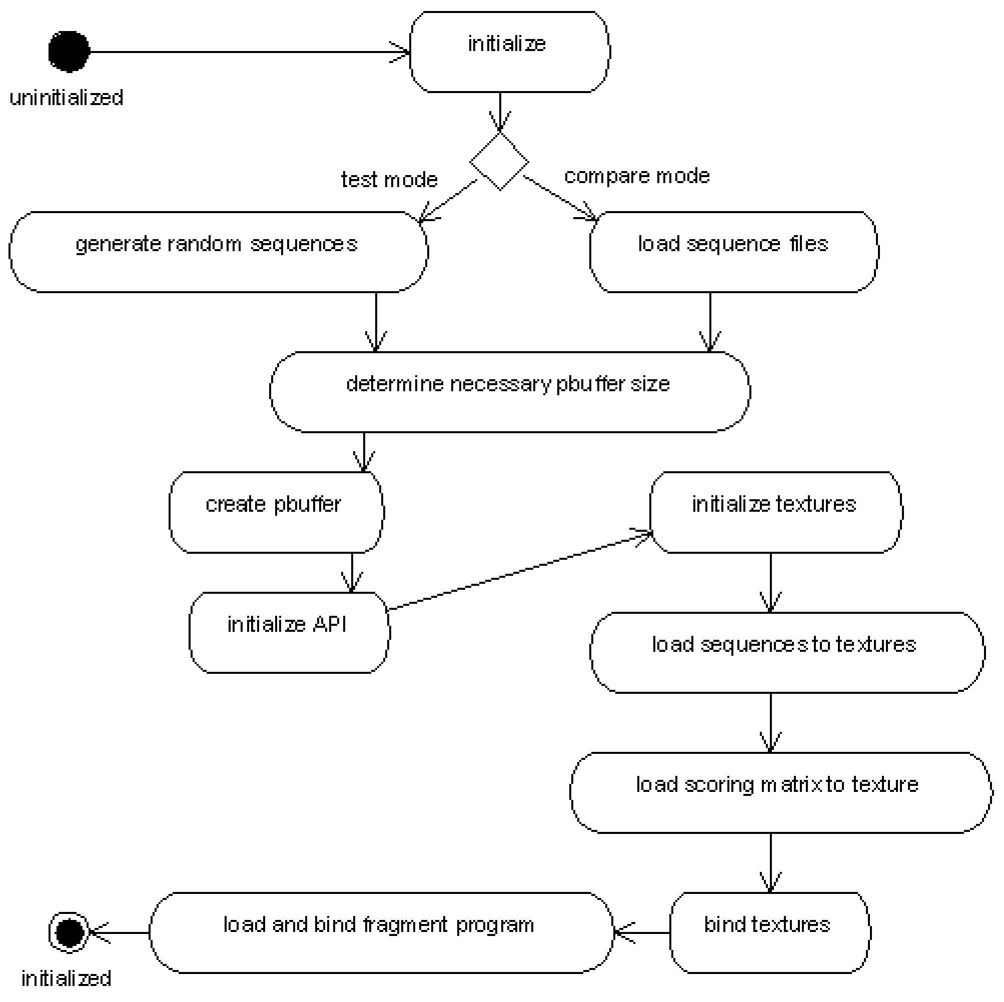

uninitialized
initialize
test mode
compare mode
generate random sequences
load sequence files
determine necessary pbuffer size
create pbuffer
initialize textures
initialize API
load sequences to textures
load scoring matrix to texture
load and bind fragment program
bind textures
initialized

Fig. 52 - Initialization activity diagram.

sequence data. Furthermore, the scoring matrix is loaded to a texture. In addition, two textures that are bound to the pbuffer are initialized later. They will contain the information of the diagonals D_{x-1} and D_{x-2}. A `ProgramObject` instance is used to load the fragment program from a file to initialize the shader program and to bind it. The initialization is complete at this point and the sequence alignment is invoked. A description of the algorithm can be found below in section "Algorithm Implementation".

After the alignment process, the `SCA` object provides information which can be accessed by get-methods and which can be used for evaluation and statistics. An example for the file results.txt that is generated from this information can be found in the appendix "SCA: results.txt". Furthermore, the n most similar sequences can be printed by the `SCA` object. The object prints n+k sequences if k more sequences have the same similarity value as sequence n.

Algorithm Implementation

How the Smith-Waterman algorithm in the SCA class is realized is described in the following. The concept defined that only diagonals of the matrix can be evaluated on a parallel system. But drawing a diagonal quad that covers the diagonal in the buffer does not yield the desired result. As shown in figure 53, all pixels touched by the quad are affected instead of rendering only the aimed pixels. Hence, the cells of each diagonal are brought into columns by shifting each row of the matrix to the right by its row number i (see figure 54).

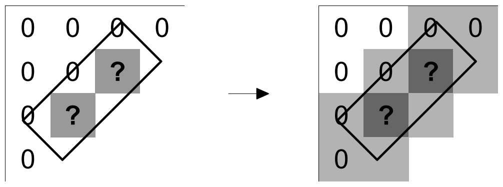


Fig. 53 - Drawing a quad over the anti-diagonal results in rendering too many pixels..

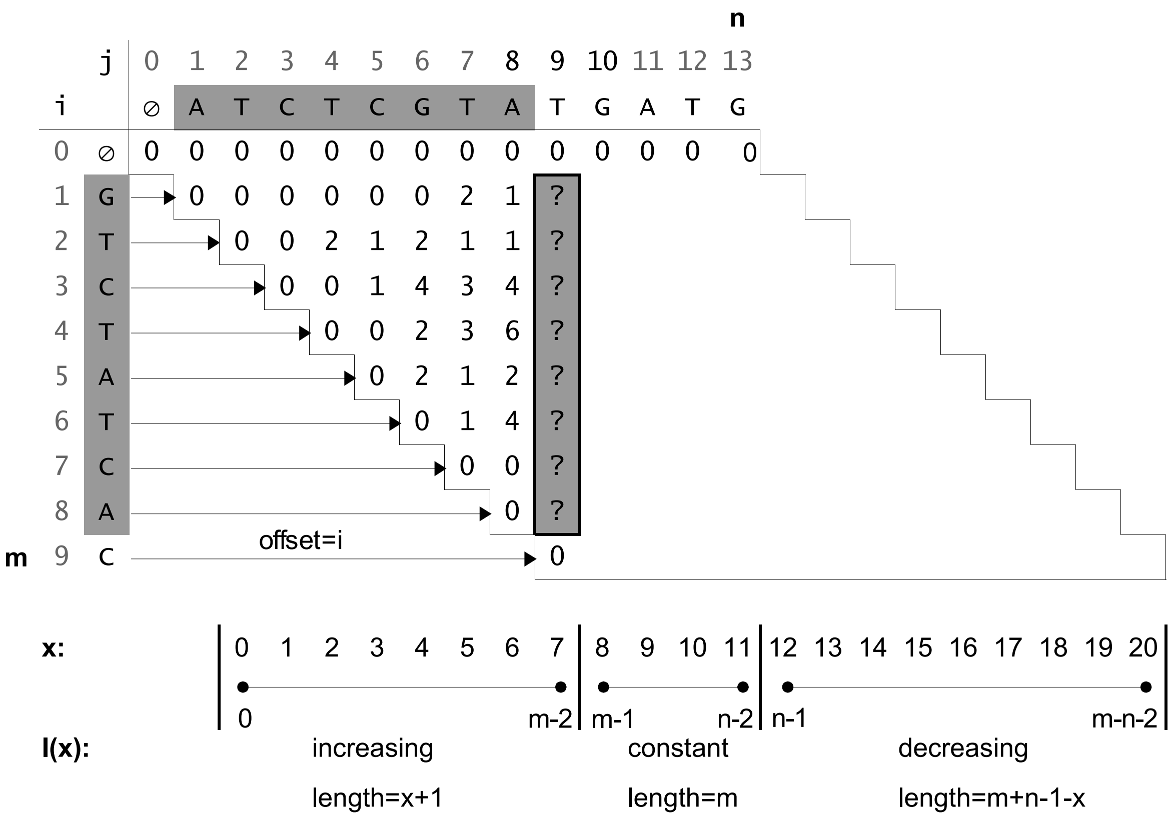

n
j 0 1 2 3 4 5 6 7 8 9 10 11 12 13
i ∅ A T C T C G T A T G A T G
0 ∅ 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 G 0 0 0 0 0 0 2 1 ?
2 T 0 0 2 1 2 1 1 ?
3 C 0 0 1 4 3 4 ?
4 T 0 0 2 3 6 ?
5 A 0 2 1 2 ?
6 T 0 1 4 ?
7 C 0 0 ?
8 A 0 ?
m 9 C 0
offset=i
x: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
0 m-2 m-1 n-2 n-1 m-n-2
l(x): increasing constant decreasing
length=x+1 length=m length=m+n-1-x

Fig. 54 - Matrix with shifted rows.

With this method, a 1×L(x) quad can be rendered in each iteration. Due to the offset, a (m+n+1)×(n+1) buffer is necessary to contain the whole matrix. Since a buffer cannot be used for reading and writing at the same time, two buffer are necessary. This means that the rendered result must be copied into the texture buffer to have two consistent matrices. This procedure is very time consuming.

Since $H_{i,j}$ depends on $H_{i-1,j}$, $H_{i,j-1}$, and $H_{i-1,j-1}$ the actual rendered diagonal D_x depends on the two previous diagonals D_{x-1} and D_{x-2}. If these three diagonals are used as separate one-dimensional buffers, D_{x-1} and D_{x-2} can be in the form of texture input and D_x is the framebuffer. In the following iteration, D_x becomes D_{x-1}, D_{x-1} becomes D_{x-2} and D_{x-2} becomes D_x. Figure 55 visualizes the idea of cyclic buffer function change. An arrow pointing towards the fragment program means that the buffer is used as texture. An arrow pointing from the fragment program to a buffer means that the buffer is used as framebuffer.

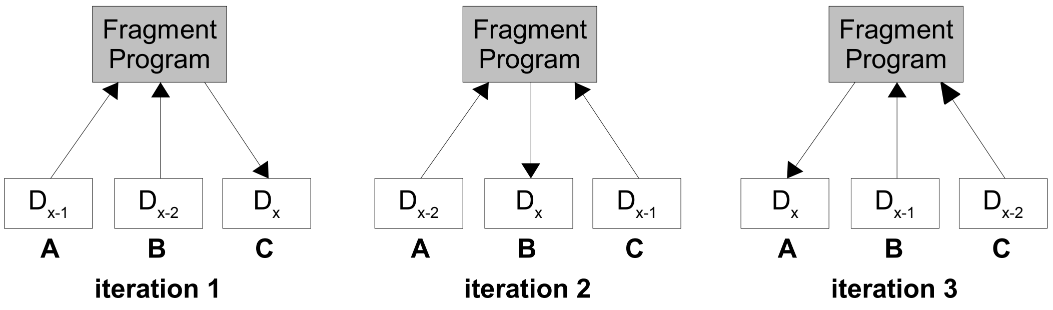


Fig. 55 - The functions of buffers A, B, and C are changed cyclic.

With this method comes the disadvantage that the information about the entire matrix is not kept. Only three diagonals are know at the same time. That means the information cannot be used to evaluate the optimal local alignment of both sequences. However, it is most import that the algorithm runs as fast as possible and returns a similarity value. This implementation is meant to scan an entire database containing several hundred thousand sequences in a short time. Afterwards, the sequences with the highest similarity scores can be picked out and the local alignment can be computed on the CPU which can store the entire matrix. The additional computation time for aligning a few sequences is relatively short compared to the entire database scanning time. In the GPU method, the highest score in the matrix which corresponds to the similarity value can still be evaluated. The maximum score max($H_{i,j}$, $H_{i-1,j}$, $H_{i,j-1}$) is stored in each cell. Since one color pixel can contain a maximum of four color components, $H_{i,j}$, $E_{i,j}$, $F_{i,j}$, and max ($H_{i,j}$, $H_{i-1,j}$, $H_{i,j-1}$) can be stored in the RGBA channels of one cell, thus storing four grids in one buffer (see figure 56).

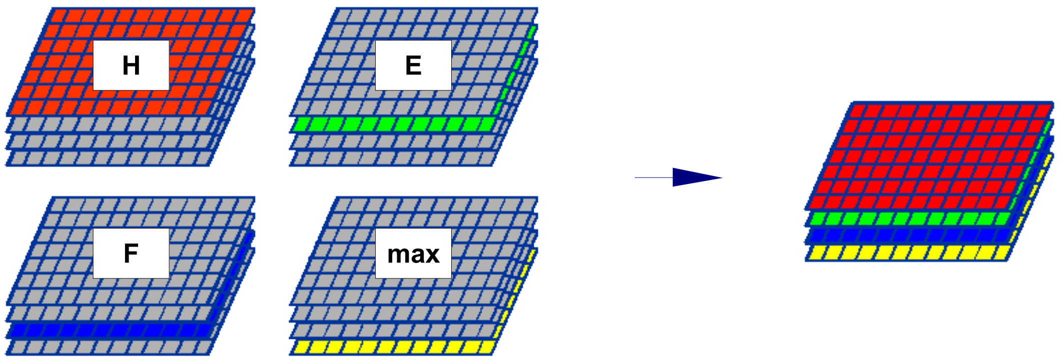


Fig. 56 - Data layout.

Furthermore, it is possible to perform N comparisons at the same time by using 2D buffers instead of one-dimensional buffers. This is shown in figure 57 in which the buffer is filled from bottom up. Each buffer contains N diagonals with a length L(x). Therefore, computation is invoked by drawing an N×L(x) quad. The maximum number of sequences is $N_{max}=PW_{max}=2^{l_{max}}$ where PW_{max} is the maximum pbuffer width. In case of the NVIDIA GeForce GT, N_{max} is 2048. A width of 4096 could be allocated as well, but then the application crashed. The same fact limits the buffer height to 2048. Since one additional matrix row with zeros is necessary for the algorithm, the maximum sequence length that can be used is $L(x)_{max}=PH_{max}-1=2^{k_{max}}-1$ where PH_{max} is the maximum pbuffer height.

The 1:1 mapped texture coordinates can be chosen in such a manner that no computation of texture coordinates is necessary in the shader (see figure 58). Due to this fact, the sampled values can be prefetched. The adapted texture mapping is solved by a cell count offset to the drawn quad whereby the quad size is identical. The offsets are different for both m<n and m>n where m is the length of the query sequence and n the maximum sequence length in the sequence set. If m<n, the diagonals with equal lengths are on the same height after the matrix shifting, otherwise they are one cell apart each (see figure 59). Table 9 shows the offsets depending on the iteration and the relation of m to n.

After all passes have been rendered, the results can be found in row min{m, n} of the buffer.

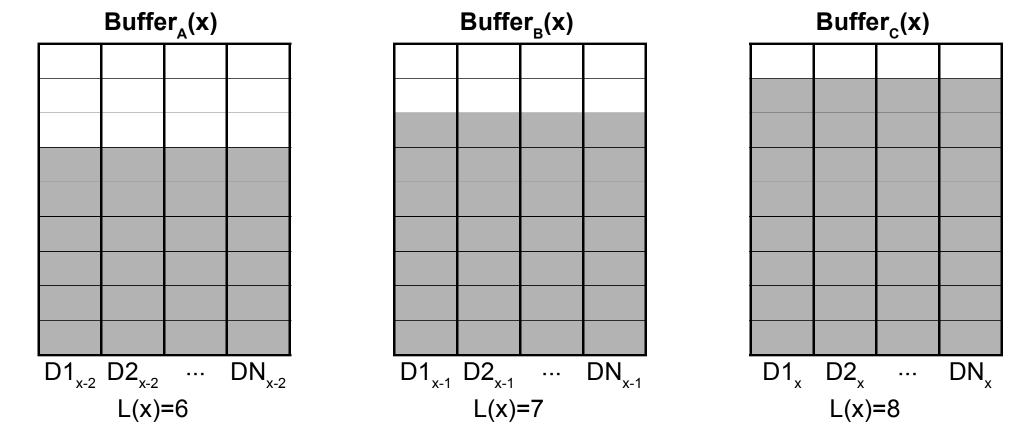


Fig. 57 - Buffers at iteration x=8 after rendering N diagonals D_x in parallel.

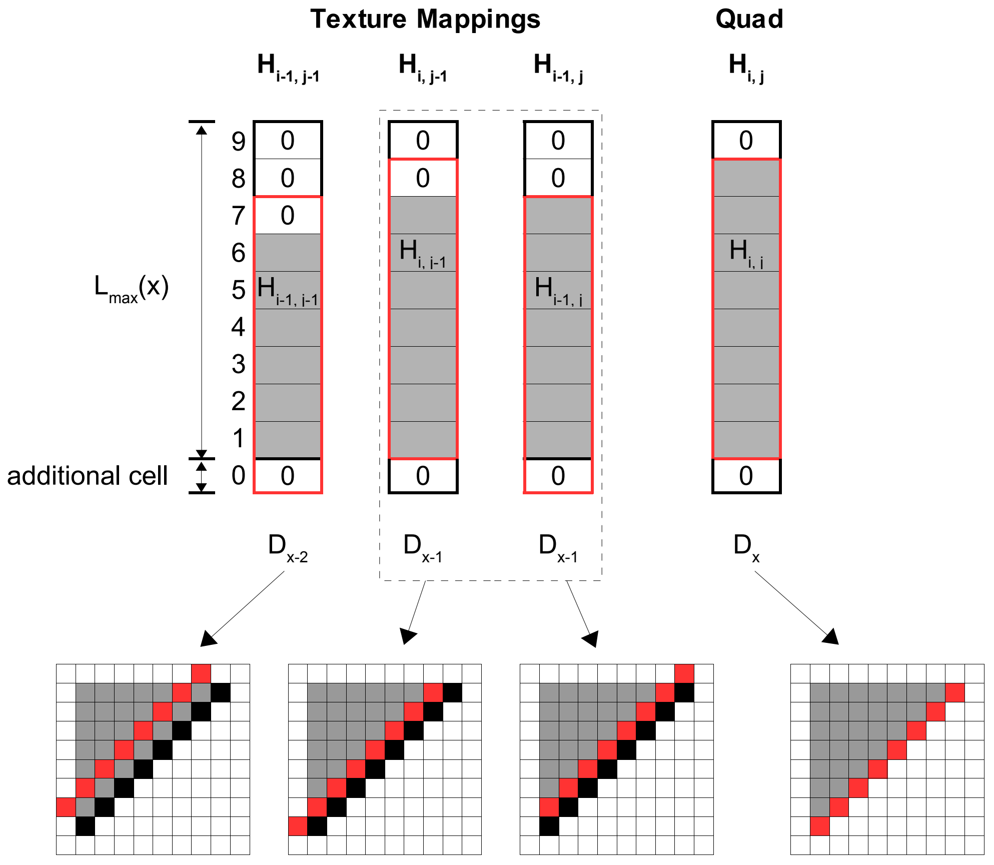


Fig. 58 - Texture mappings at iteration x=8. D_{x-1} is mapped with two texture coordinates. The quad and the texture mappings are marked in red.

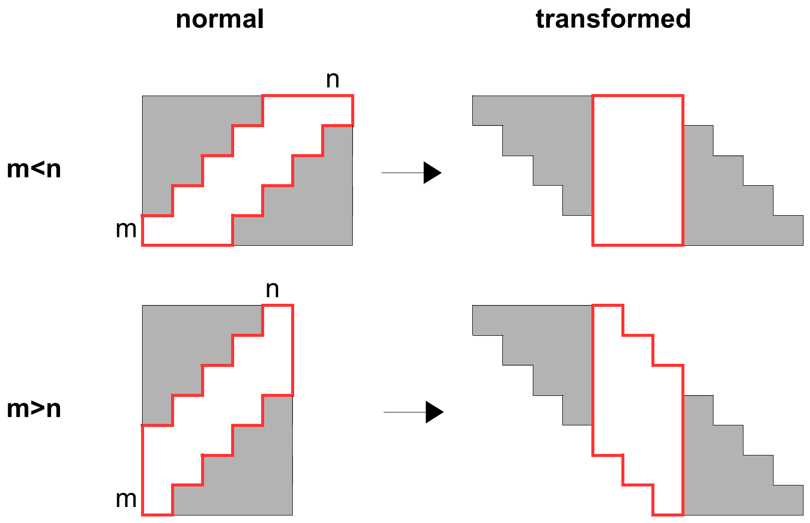


Fig. 59 - Different relations of m to n result in different matrix shapes after shifting.

	x	D_{x-2} ($H_{i-1, j-1}$)	D_{x-1} ($H_{i, j-1}$)	D_{x-1} ($H_{i-1, j}$)
m<n	all	-1	0	-1
m>n	<n	-1	0	-1
	n	0	1	0
	>n ^ <m	1	1	0
	m	0	0	-1
	>m	-1	0	-1

Table 9 - Mapping offsets.

Since the input sequences are textures, the quad is mapped to them as well (see figure 60). For sequence A, which is the vertical sequence in the matrix, the same offset in y is used as for the framebuffer quad. The quad of sequence B is mapped differently. It is mapped with the opposite direction as the quad in sequence A. Its offset is increased if x>m and m<n or if x>n and m>n. The red arrows in figure 60 show how the quad is mapped. The black dots indicate which texels are sampled in the sequence textures if the matrix cell marked with a black dot is evaluated. Assumed that the matrix cells are evaluated in series, the black arrows indicate the movements of the black dots.

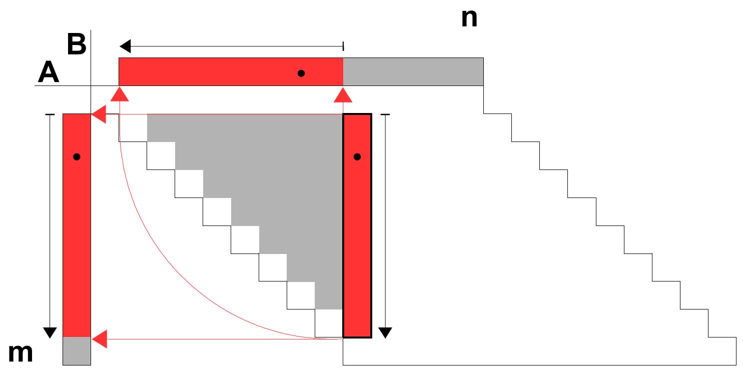


Fig. 60 - Mapping the quad to the sequence textures.

In order to evaluate the similarity s(a, b) of the elements a_i and b_j, a scoring matrix is used as lookup table. The elements of sequence A and sequence B are stored in the form of numbers or characters which are part of an ordered alphabet. If the used scoring matrix S is ordered likewise, the values of the elements a_i and b_j can be used as sampling coordinates. That means that if the elements are taken from an ordered alphabet of 20 elements which are encoded with the number 0 to 19, a 20×20 scoring matrix is used. The scoring matrix cell [a, b] contains the similarity of the elements a and b whereby [a, b] equals [b, a]. No further computation is necessary. The only disadvantage of this method is that the coordinates of the texture lookup are evaluated dynamically. That prevents texture prefetching and slows down the algorithm.

Figure 61 gives an overview of the entire alignment function. Before entering the loop that handles the passes which need to be rendered, the fragment program is activated. The following loop is executed for each of the m+n-1 diagonals. First, the quad and its texture coordinates are determined and the two buffers representing the diagonals D_{x-1} and D_{x-2} are bound as textures. The render target is set to the buffer that represents diagonal D_x in this pass. Subsequently, all necessary uniform variables are set, the quad is drawn, and its texture mappings are set. When the rendering has finished, the buffers are released from their texture bindings and the loop restarts. This is done until all diagonals are completed. After exiting the loop, the fragment program is deactivated and the result is read from the framebuffer. Furthermore, the results are stored and statistical information is generated. The function returns.

```
Alignment Function
  activate fragment program
    Passes Loop (for each diagonal)
      determine quad coordinates
      determine texture mappings
      bind buffers $D_{x-1}$ and $D_{x-2}$ as textures
      set render target to buffer $D_x$
      set uniform variables
      draw vertices and texture coordinates
      release texture buffers
    Loop End
  deactivate fragment program
  read result from framebuffer
  evaluate statistical information
Function End
```

Fig. 61 - Simplified model of an application with multiple passes.

Fragment Program

The fragment program determines the following variables:

$$E_{i,j}=max(H_{i-1,j}+\alpha,\ \ E_{i-1,j}+\beta)$$

$$F_{i,j}=max(H_{i,j-1}+\alpha,\ \ F_{i,j-1}+\beta)$$

$$H_{i,j}=max(H_{i-1,j-1}+s(a_i,b_j),\ \ E_{i,j},\ \ F_{i,j},\ \ 0)$$

$$max_{i,j}=max(H_{i,j},\ \ H_{i-1,j},\ \ H_{i,j-1})\ .$$

These are written to the framebuffer in the form of a color: red=$H_{i,j}$, green=$E_{i,j}$, blue=$F_{i,j}$, alpha=$max_{i,j}$ then. The GLSL fragment program is shown and commented in the following.

Declaration of uniform variables:

```
uniform sampler2D     texUnitSeqRef,  // query sequence
                      texUnitSeqComp; // sequence set
uniform sampler2DRect texUnitDiagN1,  // diagonal Dx-1
                      texUnitDiagN2,  // diagonal Dx-2
                      texUnitBlosum;  // scoring matrix
uniform vec4 params;                  // various parameters

void main(void)
```

```
{
```

The texture coordinates and the various parameters are renamed for a better readability. This will be optimized by the compiler and does not affect performance.

```
   vec2 tC0Ref    = gl_TexCoord[0].xy, // query sequence
        tC1Comp   = gl_TexCoord[1].xy, // sequence set
        tC2DiagN1 = gl_TexCoord[2].xy, // diagonal Dx-1
        tC3DiagN2 = gl_TexCoord[3].xy; // diagonal Dx-2

     const float step  = params.x,  // buffer step width
                 alpha = params.z,  // gap penalty
                 beta  = params.w;  // gap penalty
```

$H_{i-1,j}$, $H_{i,j-1}$, $H_{i-1,j-1}$, a_i, and b_j are sampled from textures. Against the concept, the coordinates for $H_{i,j-1}$ are evaluated dynamically in this program.

```
     vec4 hi_1j   = texture2DRect(texUnitDiagN1, tC2DiagN1   ),
          hij_1   = texture2DRect(texUnitDiagN1,
                                  tC2DiagN1 - vec2(0.0, step)),
          hi_1j_1 = texture2DRect(texUnitDiagN2, tC3DiagN2   );

     const float s1i = (texture2D(texUnitSeqRef , tC0Ref )).x,
                 s2j = (texture2D(texUnitSeqComp, tC1Comp)).x;
```

Dynamic sampling of s(ai, bj):

```
     /* determine floating point coordinates */
     vec2 tCBlosum   = vec2(s1i, s2j);
          tCBlosum  *= 255.;
          tCBlosum  += 0.5;
```

```
      /* lookup in scoring matrix */
     const float sbt = (texture2DRect(texUnitBlosum, tCBlosum)).x;
```

$E_{i,j}$, $F_{i,j}$, $H_{i,j}$, and $max_{i,j}$ are evalutated:

```
     const float E = max(hij_1.x - alpha, hij_1.y - beta),
                 F = max(hi_1j.x - alpha, hi_1j.z - beta);
```

```
     hi_1j_1.x += sbt;
     const float H = max(0.f, max(E,       max(F, hi_1j_1.x))),
           maximum = max(H,   max(hi_1j.w, hij_1.w          ));
```

Return result:

```
     gl_FragColor  = vec4(H, E, F, maximum);
}
```

NVShaderPerf translated this program to 26 assembly instructions. The number of instructions for determining $E_{i,j}$, $F_{i,j}$, $H_{i,j}$, and $max_{i,j}$ can be reduced to 23 like this:

```
     vec4 maxVec = max(vec4(hij_1.x - alpha, hij_1.y - beta,
                            hi_1j_1.x      , hi_1j.w        ),
                       vec4(hi_1j.x - alpha, hi_1j.z - beta,
                            0.f            , hij_1.w       ));

     const float H       = max(maxVec.x, max(maxVec.y, maxVec.z)),
                 maximum = max(H, maxVec.w);
     gl_FragColor  = vec4(H, maxVec.x, maxVec.y, maximum);
```

However, a test proved that the latter is slower than the previous version.

The presented implementation computes correct results. The next chapter shows its performance.

The presented implementation produces correct alignment scores. An example output of the application can be found in the appendix “SCA Example Output”. The performance test of the algorithm and their results are discussed in the following chapter.

4.4 Performance Test and Evaluation

This chapter discusses under which conditions the GPU implementation of the Smith-Waterman algorithm shows a better performance than a CPU implementation. For all GPU performance tests presented in this chapter, an NVIDIA GeForce 6800 GT (NV40 chip based) graphics card was used. It runs with 350 MHz core clock, 500 MHz memory clock (1 GHz effective due to double data rate random access memory). It is equipped with 6 vertex pipelines, 16 pixel pipelines, and 256 MB GDDR3 memory. The used operating system was Microsoft Windows XP. When the Swiss-Prot database is mentioned in this chapter, it refers to those sequences of the Swiss-Prot database (release 46.2, 1st of March 2005) which have a length below 2048 amino acids. That is 99.5 % of the entire amount of proteins.

There are two main factors that influence the performance of the GPU implementation:

1. The number of computed cells.
2. The relation between the average sequence length of the sequence set and its longest sequence.

As the graphics hardware is optimized for high-throughput computation, the performance is the faster, the more cells are processed. The amount of cells depends on the length of the query sequence, the average sequence length of the sequence set, and the number of sequences in the set. The larger each value, the less overhead occurs and the better is the performance. With the used hardware, the query sequence can be 2047 amino acids long, the sequences in the sequence set can be 2048 sequences long each, and the sequence set can contain up to 2048 sequences. The figures 62 and 63 show a

3D and a 2D visualization of a performance test. A sequence set of 2048 sequences was used. During a series of 16×16=256 tests, both the length of the query sequence as well as the lengths of the sequences in the sequence set were increased from 127 to 2047. A minimum of 127 MCUPS and a maximum of 155 MCUPS were reached. The red dot in the diagram approximately marks a 362×362 combination of sequences lengths. This corresponds to the Swiss-Prot average protein length of 362 amino acids. A value of about 148 MCUPS was reached which is 95 % of the peak value. The sequence length distribution of the Swiss-Prot database is shown in figure 64. Around 41 % of proteins of the Swiss-Prot database have a length between 256 and 512 amino acids. Queries with sequences of these lengths can be evaluated with an average of about 148.5 MCUPS which corresponds to 96 % of the peak value.

A CPU implementation does not have a dependency on the sequence lengths and no varying test scenarios need to be run. An optimized CPU implementation reached 75 MCUPS under Microsoft Windows XP on an Intel Pentium4 3,4 GHz CPU. Regarding only its peak value, the GPU implementation is about two times faster.

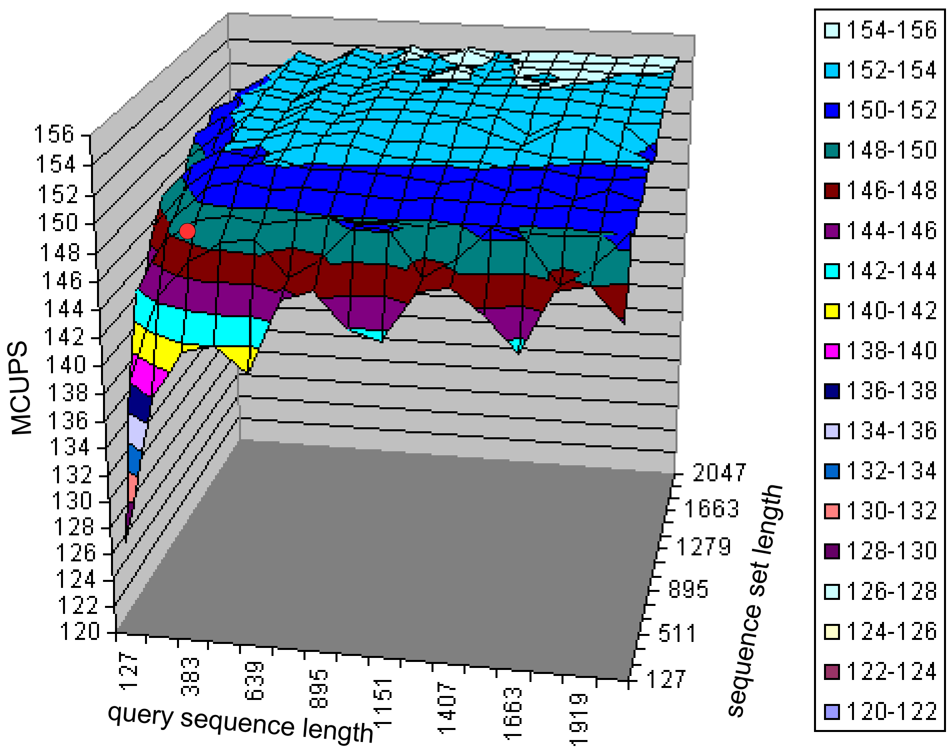


Fig. 62 - 3D performance diagram of alignment with 2048 sequences.

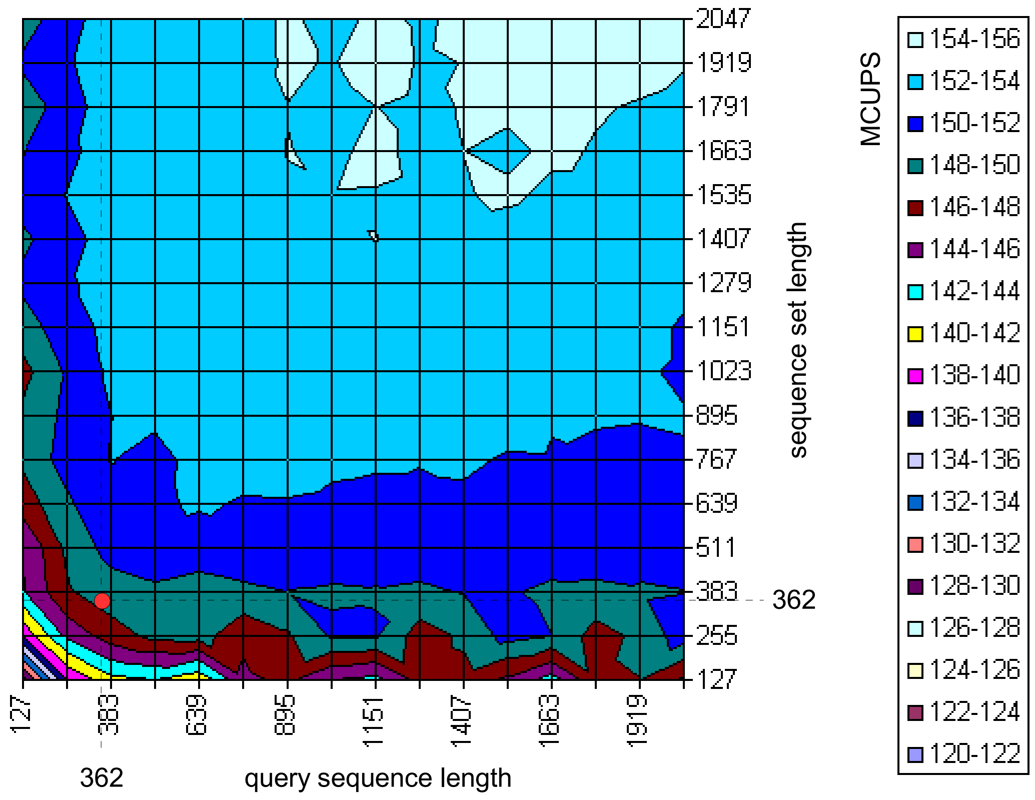


Fig. 63 - 2D performance diagram of alignment with 2048 sequences.

The presented values are only related to test cases in which all sequences in the sequence set have equal lengths. In practice, the lengths of sequences in a database vary. This is the second factor that influences the performance. The application always draws a quad of which each pixel is rendered. In case of varying sequence lengths, cells that are beyond the length of short sequences are rendered as well. Figure 65 shows 16×16 buffer filled with 16 sequences of linear increasing length. It is filled by 53 % which corresponds to the mentioned relation of the average sequence length in the set to the longest sequence. Short sequences are treated like the longest one. That means that 47 % of the computation is superfluous. As only 53 % of the computed cells in this examples are relevant, the reached CUPS value can only be assessed by the filling ratio. A peak value of 155 MCUPS therefore corresponds to only 82 relevant MCUPS in this case. However, if the Swiss-Prot database is ordered by length and cut into groups of 2048 proteins, the average sequence length within these groups is 97.5 % of the longest sequence of the respective group (see figure 66). CPU implementations of this algorithm only evaluate relevant cells and are more efficient in case of varying sequence lengths. If a GPU reaches 155 MCUPS and the buffer is filled with less than 53 % by a sequence set, the reference CPU is faster.

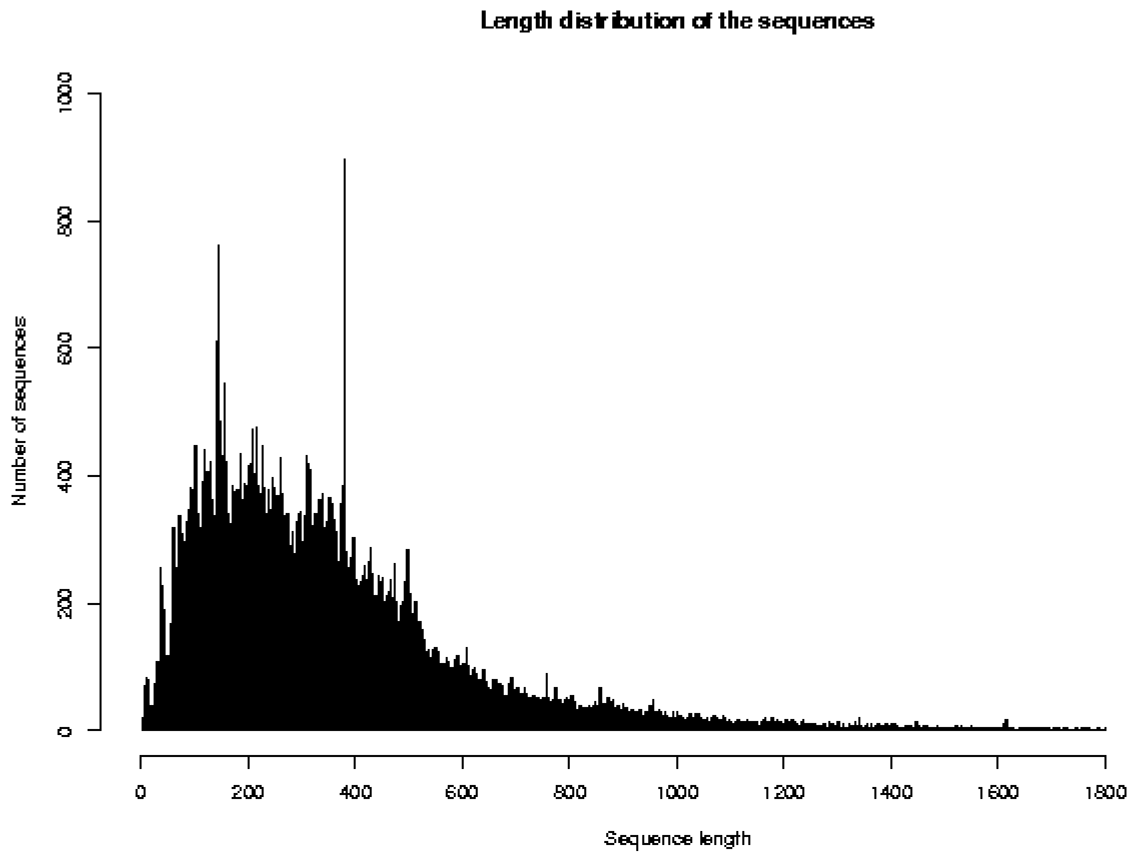


Fig. 64 - Swiss-Prot database protein sequence length distribution (release 46.6, 26.04.05, 180652 sequence entries).

Most disadvantageous scenarios for the GPU implementation are combinations of short and strongly varying sequence lengths. The 2048 shortest sequences of the Swiss-Prot database have an average length of 15 amino acids which corresponds to 64 % of the longest sequence. If the query sequence consists of only 31 amino acids, 19.8 CUPS can be reached with the GPU version. This corresponds to only 23.7 % of the CPU's performance. Using if-branches in the fragment program to speed up the rendering process in order to disregard superfluous cells only works if about 60 to 70 % or less are relevant. Otherwise, it slows down the computation. Since this border is far below the Swiss-Prot average of 97.5 %, an if-condition does not improve performance.

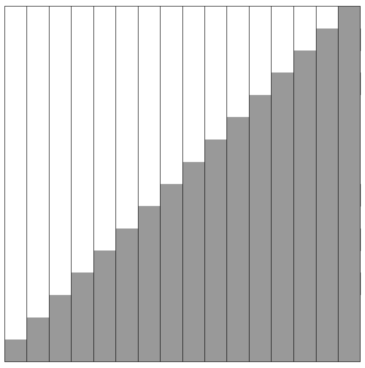


Fig. 65 - Buffer half filled.

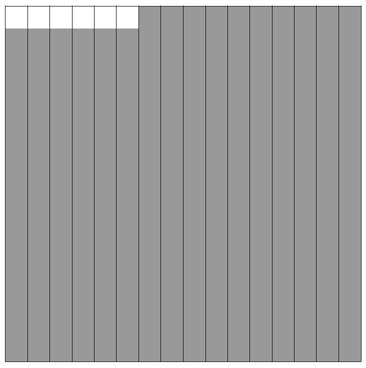


Fig. 66 - Buffer almost completely filled.

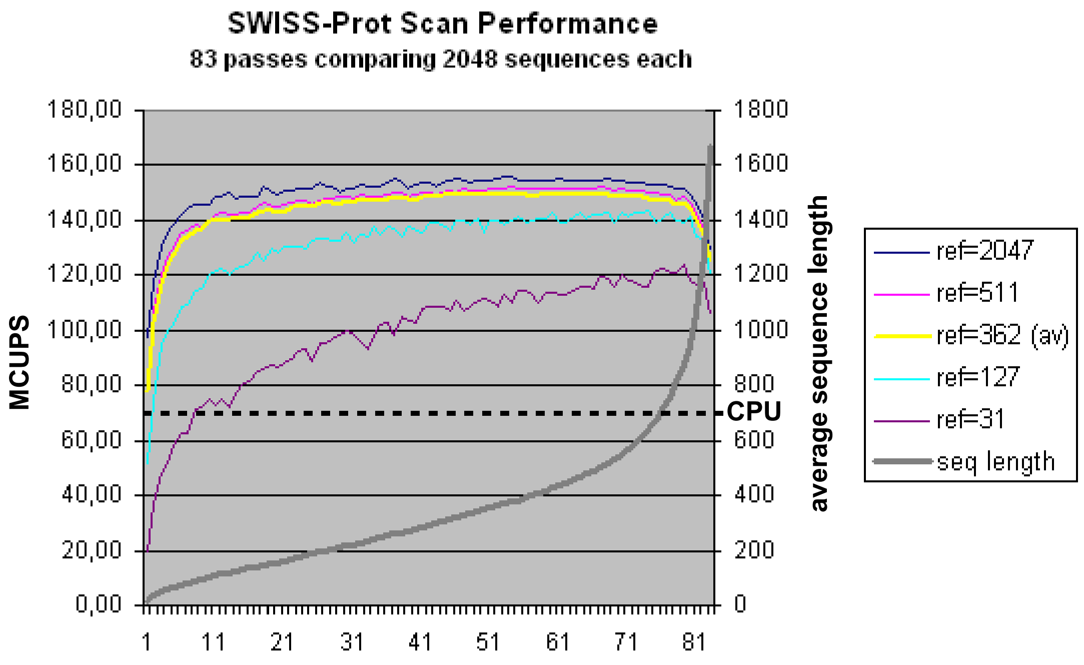


Fig. 67 - Swiss-Prot scan performance.

To test and compare the efficiency in practice, the Swiss-Prot database was compared multiple times with different query sequence lengths. Figure 67 visualizes the results of five different runs over the entire database. For each a different query sequence length was used: 31, 127, 511, 2047, and the average length of 362 amino acids. In the figure, the query sequence length is referred to as “ref”. All 169260 proteins were ordered by increasing length before they were loaded in sets of 2048 amino acids. To scan the entire database, 83 seperate renderings were necessary. Using a query sequence of 2047 amino acids, the GPU was 100 % faster than the CPU. It reached an average of 150 MCUPS. Using the shortest query length of 31 amino acids, the GPU was still 31 % faster than the CPU. The most import result is that of the average sequence length of 362 amino acids. The GPU was 91 % faster than the CPU.

The performance tests using the mentioned hardware prove that the GPU implementation is up to twice as fast as the CPU implementation. Both implementations were run on current high-end commodity systems.

4.5 Conclusions and Future Work

A GPU implementation of the Smith-Waterman algorithm for sequence comparison was presented. Regardless of that the GPUs' performance advantage applies accordingly to certain conditions only, the tests show an attractive performance improvement in practical use cases. The performance of the GPU version is the better the more sequences are used in parallel, the longer the sequences are, and the closer the sequence length ratio is to 1. If a database is scanned, large sequence sets can be used and sequences can be ordered by length so that the sequence length variation is within a few percent only. In case of the tested Swiss-Prot database release, the average sequence length in each sequence group was 97 % of the longest sequence which is very advantageous for the application. Using a query sequence with the Swiss-Prot average sequence length of 362 amino acids, the GPU's CUPS average value for scanning the entire database is 91 % higher than the CPU reference value. Using the Smith-Waterman GPU implementation for scanning databases fits the requirements for efficient computation. The used GPU is up to twice as fast as the used CPU. In addition, GPUs are much cheaper than specialized hardware and thus they are an attractive alternative. This implementation provides a way to inexpensively speed up the Smith-Waterman local alignment method on PCs by using a GPU.

However, the implementation can be further enhanced to increase performance. In this implementation, the texture coordinates used to sample $H_{i,\,j-1}$ are calculated in the shader instead of pre-calculating them by texture mapping. Changing this enables texture prefetching and could cause an increase in performance. Since if-branches do not help this application to increase performance by not evaluating values for pixels that do not need to be evaluated, z-cull could be used to disregard these pixels. The z-buffer needs to be filled with the pattern of the sequences in the sequence set first. Relevant parts get a different depth than superfluous parts and the depth-test can be used to render only relevant pixels. Assuming that a depth of 1 is assigned to relevant pixels and superfluous pixels get a depth of 0, the quad can be drawn to z=0.5, disregarding pixels with lower depth. In the shader, the depth for the pixel must be set to 1 again in order to remain at that depth. Another approach to speed up computation itself is to increase the level of parallelism. In the implemented shader, four different types of values are written to the buffer: H, E, F, and the maximum value. Each of them is evaluated in a different way. It could be faster to compute only one type of values in pairs of four. It is a try to efficiently use the native vector support in the shader. A buffer for each H, E, F, and the maximum value is necessary for this method. But as still three diagonals are used, three buffers of each type must be provided. This method requires more buffers than are available with pbuffers. However, it could be realized with FBOs. Furthermore, a large number of textures must be available in the fragment shader. A different method is to use one-component buffers instead of RGBA buffers. Rendering to single-float buffers is much faster than rendering to float buffers with more than one component. A GPUBench[39] report of the GeForce 6800 GT shows that a simple fragment program that fetches once from a single-float texture, performs a few ADD operations, and outputs the result to a single-float buffer is about seven times as fast as if multiple float components are used. Using single-float buffers for each H, E, F, and the maximum value as well as using four single-float rendertargets in a shader might increase performance. An approach to increase performance in case of sequence sets with a strong variation in length is to pack sequences intelligently into the sequence set buffer. More than one sequence can be stored in one buffer column if the sequences are short enough. There must be at least one zero cell between the sequences in a column to reset the scoring values while rendering. This can be solved by an if-branch which is slow in performance or by z-culling. The entire buffer must be cleared to zero first and all cells that have to be avoided need to be marked with a different z-depth than the normal cells.

Another necessary enhancement for the application is the ability to load an entire database at a time to avoid seeking in the source file. This enables a faster preparation of sequence data for rendering. Furthermore, the SCA object is created and destroyed for each alignment whereat the switching of device contexts needs a lot of time. This can be avoided by keeping one SCA object until all alignments are finished.

Using graphics hardware for computation still needs a lot of experience and knowledge regarding hardware and API. All data that is used must be mapped to fit graphical primitives like textures and vertices. GPGPU languages might change this in future if they prove themselves and win recognition. They abstract GPU programming from the

39http://graphics.stanford.edu/projects/gpubench/

graphics level and support an easier programming.

Much more than software development, hardware development influences the performance and features of such applications. The annual performance growth of GPUs has been higher than that of CPUs in the past years. If this development continues in the following years, the speed advantage of GPUs towards CPUs will grow further, making them even more attractive. The functionality and programmability of GPUs was increased by their vendors since they were introduced. More programmability will give developers more independence in implementing GPU programs in the near future. Algorithms that cannot be implemented yet will be possible then.

5 Summary

This diploma thesis gives an introduction into general-purpose programming on graphics hardware. Furthermore, an efficient GPU-based implementation of the Smith-Waterman algorithm is presented. Since the algorithm origins from bioinformatics, this subject, genetics and sequence comparison algorithms are discussed as well.

Research in genetics has produced huge amounts of data within the past decade. The main producers of this data are genome projects which investigate genomes of different species. A genome is the entire hereditary information of a biological lifeform. The information is stored in DNA which is a long chain of base pairs. Parts of the DNA are responsible for producing proteins which build up the structure of cells and which execute basic functions of life. Proteins themselves consist of amino acid chains. It is researched both on the functions of DNA and proteins as well as on their connection. Information about these sequences is kept in databases which grew fast in the past years. In the context of research, sequences are analyzed to find similarities for example. This mainly means to compare one sequence, may it be DNA or a protein, to other sequences of a database in order to find the most similar ones. Different algorithms are used to evaluate the similarity. One of these is the Smith-Waterman algorithm which is a very accurate one, but it is very computation intensive. This is why it is tried to find ways to speed it up.

One approach is to use parallely working architectures to compute the alignments because they provide high performance. Besides expensive and hard to use specialized hardware, commodity graphics hardware of desktop computers and other devices can be used for this purpose. They provide a high-throughput and since some years they are partially programmable. The processing of geometry is divided into several stages, the so-called graphics pipeline. In the vertex stage, geometry is transformed from world-space to image space. The rasterizer produced preliminary pixels, the so-called fragments, from primitives like triangles. The fragment processor receives these fragments, computes a pixel color for each fragment and writes it to the framebuffer. The vertex and fragment processors are those parts which are programmable. They can execute general-purpose programs as well. This system is described as streaming architecture. Each information that flows through the pipeline is part of a stream. Therefore, the programmable processors are streaming processors that apply kernels to the stream. Kernels are programs that are applied to each stream element. Since several streaming processors work in parallel and only one element of the stream can be accessed at a time, the elements must be independent of each other. The programs are written in GPU programming languages that depend on specific APIs which are an abstraction layer between the application an the hardware driver. The execution of a program is controlled by API commands. A general-purpose application has to map its algorithm onto the available graphical elements like textures, vertices, and polygons. Information is stored in buffers and passed to the GPU processors in the form of textures. Texture are input streams for these streaming processors. To reuse a computed result it has to be passed to the processor as texture again. The state-of-the-art technique is called Render-To-Texture. It uses pixel buffers to store the information. The pbuffers can be used as texture and as framebuffers but never both at a time. If information is rendered to a framebuffer, this information can be reused by using the buffer as texture input. At the same time, a different buffer must be used as framebuffer. Computation is invoked by drawing geometry into the framebuffer which is usually a quad with an orthogonal projection. All pixels covered by the quad are evaluated.

The Smith-Waterman algorithm compares two sequences with length m and n by computing an m×n matrix. After transforming the matrix, the cells of its diagonals can be evaluated in parallel on the GPU. Only three diagonals are known at a time, but the query sequence can be compared to many sequence in parallel. This method was implemented. The application is able to load sequences and to scoring matrices from files. Any further information is passed as command line parameter. 2048 proteins can be compared at a time whereas the sequences can have a maximum length of 2047 amino acids. In tests, 155 MCUPS were reached which is twice as much as the value of a reference CPU implementation. Since the performance depends on the lengths of the sequences in the database, the Swiss-Prot database was scanned as practical use case analysis. An average performance improvement of 91 % compared to the CPU reference value of 75 MCUPS was reached. A query sequence of 362 amino acids was used which corresponds to the average sequence length in the Swiss-Prot database.

It was proven that a GPU implementation that is run on commodity graphics hardware is up to twice as fast as a CPU implementation that was run on adequate PC hardware. Due to their high performance, GPUs are an attractive coprocessor to the CPU and with that an inexpensive alternative to specific hardware. In future, their performance and programmability will increase so that they will become more attractive for general-purpose programming. The GPU might evolve to a general-purpose coprocessor that is taken for granted, executing all high-throughput tasks redirected from the CPU.

6 References

In the following, all used references are listed. Since most papers are available in the internet nowadays, most references additionally have an URL line, although it is not an HTML page.

Akeley 2003
Akeley K: Data Storage and Transfer in OpenGL. NVIDIA U Presentation, 25 July 2003. NVIDIA Corporation, 2003.
URL: http://developer.nvidia.com/docs/IO/8229/Data-Xfer-Store.pdf, 22.12.2004

Alaghband 1997
Alaghband G: Parallel Computing and Architectures: SIMD Architectures. Copyright: Gita Alaghband, 1997
URL: http://carbon.cudenver.edu/~galaghba/simd.html, 22.12.2004

Alberts et al. 1994
Alberts B, Bray D, Lewis J, Raff M, Roberts K, Watson JD: Molecular Biology Of The Cell. Garland Publishing Inc., New York, 1994.

Altschul et al. 1990
Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ: Basic local alignment search tool. J Mol Biol 215, 403-10, 1990

ATI 2004
ATI: Radeon X800 graphics card online product site, 2004.
URL: http://www.ati.com/products/radeonx800, 11.02.2005

BCHM 2001
Belgian Clearing-House Mechanism: Glossary of terms related to the Convention on Biological Diversity, 2001. Copyright: Belgian Clearing-House Mechanism, 2001.
URL: http://bch-cbd.naturalsciences.be/belgium/glossary/glos_b.htm, 12.02.2005

Buck 2004
Buck I: SIGGRAPH 2004 GPGPU Course: GPU Computation Strategies & Tricks. Copyright: Ian Buck, Graphics Lab, Stanford University, 2004
URL: http://www.gpgpu.org/s2004/slides/buck.StrategiesAndTricks.ppt, 15.12.2004

Buck 2004b
Buck I: IEEE Visualization 2004: GPGPU - High level languages. Copyright: Ian Buck, Graphics Lab, Stanford University, 2004
URL: http://www.gpgpu.org/vis2004/D.buck.vis04.languages.pdf, 15.12.2004

Buck et al. 2004
Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P: Brook for GPUs: Stream Computing on Graphics Hardware. In: Proceedings of SIGGRAPH, 2004
URL: http://graphics.stanford.edu/papers/brookgpu/brookgpu.pdf, 15.12.2004

Carr et al. 2002
Carr NA, Hall JD, Hart JC: The Ray Engine. In: Proceedings of Graphics hardware, Eurographics Association, 37-46, 2002.

Cooper 2000
Cooper GM: The Cell. Sinauer Associates Incorporated, Sunderland, USA, 2000.

Dayhoff et al. 1978
Dayhoff MO, Schwartz RM, Orcutt BC:. A model for evolutionary change in proteins. In: Dayhoff MO (editor): Atlas of Protein Sequence and Structure, National Biochemical Research Foundation, Washington DC, volume 5, 345-52, 1978.

England 1978
England JN: A system for interactive modeling of physical curved surface objects. In: Proceedings of SIGGRAPH 1978, 336-40, 1978.

Fernando 2004
Fernando R: EuroGraphics 2004 Tutorial 5: Programming Graphics Hardware – High-Level Shading Languages. Copyright: Randy Fernando, NVIDIA Developer Technology Group, 2004
URL: http:/download.nvidia.com/developer/presentations/ 2004/Eurographics/EG_04_ProgrammingLanguages.pdf, 7.12.2004

Fernando et al. 2004
Fernando R, Harris M, Wloka M, Zeller C: EuroGraphics 2004 Tutorial 5: Programming Graphics Hardware. Copyright: Randy Fernando, Mark Harris, Matthias Wloka , Cyril Zeller, NVIDIA Corporation, 2004
URL: http://download.nvidia.com/developer/presentations/ 2004/Eurographics/EG_04_TutorialNotes.pdf, 7.12.2004

LHNCfBC 2005
Lister Hill National Center for Biomedical Communications: Genetics Home Reference. The Basics: Genes and How They Work ; 2005 March 11. Copyright: Lister Hill National Center for Biomedical Communications, U.S. National Library of Medicine, National Institutes of Health, Department of Health & Human Services.
URL: http://ghr.nlm.nih.gov/dynamicImages/understandGenetics/basics.pdf, 19.03.2005.

Golding & Morton 2004
Golding B, Morton D: Elementary Sequence Analysis. Copyright: Brian Golding and Dick Morton, Department of Biology, McMaster University, Hamilton, Ontario 2004
URL: http://helix.biology.mcmaster.ca/3S03.pdf

Gonnet et al. 1992
Gonnet GH, Cohen MA, Benner SA: Exhaustive matching of the entrie protein sequence database. Science 256, 1443-5, 1992.

Goodnight et al. 2003
Goodnight N, Woolley C, Lewin G, Luebke D, Humphreys G: Eurographics/SIGGRAPH Workshop on Graphics Hardware 2003 - A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. Copyright: Nolan Goodnight, Cliff Woolley, Greg Lewin, Dave Luebke, Greg Humphreys, 2003
URL: http://www.cs.virginia.edu/~gfx/pubs/multigridGPU/multigrid.pdf, 30.12.2004

Gotoh 1982
Gotoh O: An Improved Algorithm for Matching Biological Sequences. J Mol Biol 162, 705-8, 1982.
URL: http://jaligner.sourceforge.net/references/gotoh1982.pdf, 03.02.2005

Govindaraju et al. 2003
Govindaraju N, Redon S, Lin MC, Manocha D: Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware, 25–32, 2003.

Govindaraju et al. 2004
Govindaraju NK, Lloyd B, Wang W, Lin M, Manocha DB: Fast computation of database operations using graphics processors. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, 215-26, 2004. URL: http://portal.acm.org/ft_gateway.cfm?id=1007594&type=pdf&coll=GUIDE&dl=GUIDE&CFID=42562879&CFTOKEN=6114514 DATUM

Harris et al. 2002
Harris MJ, Coombe G, Scheuermann T, Lastra A: Physically-Based Visual Simulation on Graphics Hardware. In: Proceedings of Graphics Hardware, 109-18, 2002.

Harris et al. 2003
Harris MJ, Baxter WV, Scheuermann T, Lastra A:. Simulation of cloud dynamics on graphics hardware. In: Proceedings of Graphics hardware, Eurographics Association, 92-101, 2003.

Harris 2004
Harris M: SIGGRAPH 2004 GPGPU Course: Mapping Computational Concepts to GPUs.
URL: http://www.gpgpu.org/s2004/slides/harris.Mapping.ppt, 12.01.2005

Harris 2004b
Harris M: EuroGraphics 2004 Tutorial 5: Programming Graphics Hardware – GPGPU: Beyond Graphics
URL: http://eg04.inrialpes.fr/Programme/Tutorial/PDF/Tutorial-5.5-GPGPU.pdf, 12.01.2005

Harris 2005
Harris M: GPGPU: General Purpose Computation on GPUs. Game Developers Conference 2005.
URL: http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_GPGPU.pdf, 11.04.2005

Henikoff & Henikoff 1992
Henikoff S, Henikoff JG: Amino acid substitution matrices from protein blocks. Proc. Natl. Acad. Sci. USA 89:10915-19, 1992.

HGMIS 2003
Human Genome Management Information System: Genomics and its Impact on Science and Society: The Human Genome project and Beyond. Oak Ridge National Laboratory, Oak Ridge, Tennessee, 2003.
URL: www.ornl.gov/hgmis/publicat/primer, 04.01.2005

HGMIS 2003a
Human Genome Management Information System: Human Genome Project Information. Oak Ridge National Laboratory, Oak Ridge, Tennessee, 2003.
URL: http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml, 21.03.2005

Higgins & Sharp 1989
Higgins DG, Sharp PM: Fast and sensitive multiple sequence alignments on a microcomputer. Comput Appl Biosci, 151-3, 1989

Hillesland et al. 2003
Hillesland K, Molinov S, Grzeszczuk R: Nonlinear optimization framework for imagebased modeling on programmable graphics hardware. In: Proceedings of SIGGRAPH 2003, 925–934, 2003.

Hirschberg et al. 1996
Hirschberg JD, Hughey R, Karplus K: Kestrel: A Programmable Array for Sequence Analysis. Proceedings of International Conference for Application-Specific Systems, Architectures, and Processors, IEEE CS, 25-35, 1996.
URL: http://www.ccm.ece.vt.edu/hokiegene/papers/hirschberg96kestrel.pdf, 18.03.2005

Ho 2005
Ho TC: Computer Graphics course - Programming with OpenGL Shading Language. Department of Computer Science and Information, National Chiao-Tung University, HsinChu, Taiwan, ROC, 2005.
URL: http://cggmpc18.csie.nctu.edu.tw/~danki/courses/opengl/9_GLSL.pdf, as of 17.04.2005.

Hoff et al. 2001
Hoff KE, Zaferakis A, Lin M, Manocha D: Fast and simple 2D geometric proximity queries using graphics hardware. In: Proceedings of the 2001 Symposium on Interactive 3D Graphics, 145–148, 2001.

Holley et al. 1965
Holley RW, Apgar J, Everett GA, Madison JT, Marquisee M, Merrill SH, Penswick JR, Zamir A: Structure of a ribonucleic acid. Science 147, 1462-5, 1965.

Houston 2005
Houston M: Parallel Visualization Clusters. OpenIB Alliance Meeting, Feb 8, 2005.
URL: http://www.openib.org/docs/oib_wkshp_022005/visualization-stanford-mhouston.pdf

Huang et al. 1990
Huang X, Hardison RC, Miller W: A space-efficient algorithm for local similarities. CABIOS 6, 373-81, 1990.

IISR 2003
Indian Institute of Spice Research. Training on Bioinformatics and Biotechnology – Tools & Applications. Part 1 – Bioinformatics, 2-23, 2003..
URL: http://www.iisr.org/bioinformatics/train6.pdf, 13.02.2005

Istrail et al. 2003
Istrail S, Sutton GG, Florea L, Halpern AL, Mobarry CM, Lippert R, Walenz B, Shatkay H, Dew I, Miller JR, Flanigan MJ, Edwards NJ, Bolanos R, Fasulo D, Halldorsson BV, Hannenhalli S, Turner R, Yooseph S, Lu F, Nusskern DR, Shue BC, Zheng XH, Zhong F, Delcher AL, Huson DH, Kravitz SA, Mouchard L, Reinert K, Remington KA, Clark AG, Waterman MS, Eichler EE, Adams MD, Hunkapiller MW, Myers EW, Venter JC: Whole-genome shotgun assembly and comparison of human genome assemblies. PNAS 101, 1916-21, 2003.
URL: http://sugp.caltech.edu/sorin/Sorin-Papers/IstrailEtAlPNAS.pdf, 23.02.2005

Jovanovic 2003
Jovanovic O: G4120: Introduction to Computational Biology - Introduction to Bioinformatics. Copyright: Oliver Jovanovic, Department of Microbiology, Comubia University, 2003.
URL: http://microbiology.columbia.edu/icb/fall2004/Lecture3.pdf, 26.02.2005

Kessenich et al. 2004
Kessenich J, Baldwin D, Rost R: The OpenGL Shading Language. Document Revision 59, 2004.
URL: http://www.opengl.org/documentation/oglsl.html, 04.01.2005

Kim & Lin 2003
Kim T, Lin MC: Visual simulation of ice crystal growth. In: Proceedings of the 2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 86–97, 2003.

Krüger & Westermann 2003
Krüger J, Westermann R: Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans Graph 22, 908-16, 2003.

Lefohn 2004
Lefohn A: IEEE Visualization 2004: GPGPU - Introduction and Overview. Copyright: Institute for Data Analysis and Visualization, University of California, Davis, 2004.
URL: http://www.gpgpu.org/vis2004/A.lefohn.intro.pdf, 27.12.2004

Lefohn 2004b
Lefohn A: IEEE Visualization 2004: GPGPU - The GPGPU Programming Model. Copyright: Institute for Data Analysis and Visualization, University of California, Davis, 2004.
URL: http://www.gpgpu.org/vis2004/C.lefohn.mapping.pdf, 27.12.2004

Leòn 1999
Leòn D: Bioinformatics: Database and Research Resources for your Laboratory, 1999. Copyright: Darryl Leòn, 1999.
URL: http://home.san.rr.com/dna/darryl/home.html, 12.02.2005

van der Linden 2004
van der Linden J: Game Technology: Vertex and Fragment Shaders. Department of Computer Science, The University of Auckland, 2004.
URL: http://www.cs.auckland.ac.nz/compsci777s2c/lectures/Jarno/lecture8.pdf, 05.01.2005

Lindholm et al. 2001
Lindholm E, Kligard MJ, Moreton H: A user-programmable vertex engine. In: Proceedings of SIGGRAPH 2001, 149-58, 2001.

Luebke 2004
Luebke D: SIGGRAPH 2004 GPGPU Course: Introduction. Copyright: David Luebke, 2004.
URL: http://www.gpgpu.org/s2004/slides/luebke.Introduction.ppt, 15.12.2004

Mark et al. 2003
Mark WR, Glanville RS, Akeley K, Kilgard MJ: Cg: A system for programming graphics hardware in a C-like language. ACM Trans Graph 22, 896-907, 2003.

McCool et al. 2002
McCool MD, Qin Z, Popa TS: SIGGRAPH/Eurographics Graphics Hardware Workshop, Saarbruecken, Germany, 57-68, 2002.
URL: http://www.cgl.uwaterloo.ca/Projects/rendering/Papers/metaAPIpaper.pdf (revised version), 13.04.2005

Microsoft 2003
Microsoft: High-level Shader Language, 2003.
http://msdn.microsoft.com/library/default.asp?url=/library/enus/directx9c/directx/graphics/reference/Shaders/HighLevelShaderLanguage.asp, 13.12.2004

Moore 1965
Moore GE: Cramming more components onto integrated circuits, Electronics 38, 1965.
URL: ftp://download.intel.com/research/silicon/moorespaper.pdf, 16.04.2005

Moreland & Angel 2003
Moreland K, Angel E: The FFT on a GPU. In: SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003 Proceedings, 112–9, 2003.
URL: http://www.cs.unm.edu/~kmorel/documents/fftgpu/, 23.02.2005

Lengyel et al. 1990
Lengyel J, Reichert M, Donald BR, Greenberg DP: Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. In: Proceedings of SIGGRAPH 1990, 327-35, 1990.

Lindholm et al. 2001
Lindholm E, Kilgard MJ, Moreton H: A user-programmable vertex engine. In: Proceedings of ACM SIGGRAPH 2001, 149–58, 2001.

Needleman & Wunsch 1970
Needleman SB, Wunsch CD: A general method applicable to the search for similarities in amio acid asequences of two proteins. J Mol Biol 48, 443-53, 1970.
URL: http://www.ccm.ece.vt.edu/hokiegene/papers/Gotoh_82.pdf, 23.02.2005

NHGRI 2004

National Human Genome Research Institute: Human Genome Project Completion: Frequently Asked Questions. 2004.
URL: http://www.genome.gov/11006943, 23.03.2005

NVIDIA 2004
NVIDIA Corporation: GeForce 6800: Product overview, 2004.
http://nvidia.com/page/geforce 6800.html., 13.03.2005

Olano & Lastra 1998
Olano M, Lastra A: A Shading Language on Graphics Hardware: The PixelFlow Shading System. In: Proceedings of SIGGRAPH 1998, 159-68. 1998.
URL: http://www.csee.umbc.edu/~olano/s2004c01/ch03.pdf, 03.03.2005

Owens 2004
Owens J: IEEE Visualization: GPGPU, The GPU Family Tree. Copyright: Institute for Data Analysis and Visualization, University of California, Davis, 2004.
URL: http://www.gpgpu.org/vis2004/B.owens.gpu-family-tree.pdf, 28.12.2004

Pearson 2001
Pearson WR: Protein sequence comparion and Protein evolution. Tutorial – ISMB2000. Department of Biochemistry and Molecular Genetics , University of Virginia, USA, 2001.
URL: www.people.virginia.edu/~wrp/papers/ismb2000.pdf, 12.02.2005

Pearson & Lipman 1988
Pearson WR, Lipman DJ: Improved tools for biological sequence comparison. Proc Natl Acad Sci 85, 2444-8.

Peercy et al. 2000
Peercy MS, Olano M, Airey J, Ungar PJ: Interactive Multi-Pass Programmable Shading. In: Proceedings of SIGGRAPH 2000, 425-32, 2000.

Rhoades et al. 1992
Rhoades J, Turk G, Bell A, State A, Neumann U, Varshney A: Real-Time Procedural Textures. In: Proceedings of Symposium on Interactive 3D Graphics 1992, 95-100, 1992.

Potmesil & Hoffert 1989
Potmesil M, Hoffert EM: The Pixel Machine: A Parallel Image Computer. In: Proceedings of SIGGRAPH 1989, 69-78, 1989.

Proudfoot et al. 2001
Proudfoot K, Mark WR, Tzvetkov S, Hanrahan P: A Real-Time Procedural Shading System for Programmable Graphics Hardware. In: Proceedings of SIGGRAPH 2001, 159-70, 2001.

Purcell et al. 2002
Purcell TJ, Buck I, Mark WR, Hanrahan P: Ray tracing on programmable graphics hardware. ACM Trans Graph, 703-12, 2002.
URL: http://graphics.stanford.edu/papers/rtongfx/rtongfx.pdf, 14.03.2005

Rupp 2000
Rupp B: Protein Structure Basics. Copyright: Bernhard Rupp, 2002.
URL: http://www-structure.llnl.gov/Xray/tutorial/protein_structure.htm, 26.02.2005

Sanger et al. 1977
Sanger F, Nicklen S, Coulson AR: DNA sequencing with chain-terminating inhibitors. Proc Natl Acad Sci 74, 5463-67, 1977.

Segal & Akeley 2004
Segal M, Akeley K: The OpenGL Graphics System: A Specification, Version 2.0, 2004.
URL: http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf, 11.02.2005

Shamir 2002
Shamir R: Analysis of Gene Expression Data. Lecture 1: March 14, 2002. School of Computer Science, Tel Aviv University, 2002.
URL: http://www.math.tau.ac.il/~rshamir/ge/02/scribes/lec01.pdf, 11.02.2005

Smith & Waterman 1981
Smith TF, Waterman MS: Identification of common molecular subsequences. J Mol Biol 147, 195-7, 1981.
URL: http://www.cmb.usc.edu/papers/msw_papers/msw-042.pdf, 02.02.2005

Strzodka 2004
Strzodka R: IEEE Visualization 2004: GPGPU, Developer Tools. Copyright: Caesar Research Center, Bonn, Germany, 2004.
URL: http://www.gpgpu.org/vis2004/I.strzodka.DeveloperTools.pdf, 28.12.2004

Tompa 2000
Tompa M: Multiple Sequence Alignment. In: Lecture Notes on Biological Sequence Analysis, Technical Report #2000-06-01, Department of Computer Science and Engineering, University of Washington, 27-34, 2000.
URL: www.cs.uml.edu/bioinformatics/resources/Lectures/tompa00lecture.pdf, 26.12.2004

Venter et al. 2001
Venter JC et al.: The Sequence of the Human Genome. Science 291, 1304-51, 2001.
URL: http://www.sciencemag.org/cgi/reprint/291/5507/1304.pdf, 21.03.2005

Wikipedia 2005
Wikipedia contributors. Bioinformatics. Wikipedia, the free encyclopedia; 2005 Feb 14, 03:26 UTC [cited 2005 Feb 14].
Available from: http://en.wikipedia.org/wiki/Bioinformatics

Wikipedia 2005b
Wikipedia contributors. Gene Expression. Wikipedia, the free encyclopedia; 2005 Feb 12, 23:06 UTC [cited 2005 Feb 20].
Available from: http://en.wikipedia.org/wiki/Gene_expression

Wikipedia 2005c
Wikipedia contributors. Protein. Wikipedia, the free encyclopedia; 2005 Feb 16, 23:30 UTC [cited 2005 Feb 20].
Available from: http://en.wikipedia.org/wiki/Protein

Wikipedia 2005e
Wikipedia contributors. Sequence Alignment. Wikipedia, the free encyclopedia; 2005 Feb 25, 10:36 UTC [cited 2005 Feb 26].
Available from: http://en.wikipedia.org/wiki/Sequence_alignment

Woolley 2004
Woolley C: SIGGRAPH 2004 GPGPU Course: Efficient Data Parallel Computing on GPUs. Copyright: Cliff Woolley, 2004.
URL: http://www.gpgpu.org/s2004/slides/woolley.EfficientDataParallelProg.ppt, 15.12.2004

Wynn 2001b
Wynn C: NVIDIA Corporation Technical Paper: Using P-Buffers for Off-Screen Rendering in OpenGL. Copyright: Chris Wynn, NVIDIA Corporation, 2001.
URL: http://developer.nvidia.com/attach/6534, 27.01.2005

Yang et al. 2002
Yang R, Welch G, Bishop G: Realtime consensus-based scene reconstruction using commodity graphics hardware. In: Proceedings of Pacific Graphics, 2002.

Zeller 2004
Zeller C: EuroGraphics 2004 Tutorial 5: Programming Graphics Hardware – Controlling the GPU from the CPU: The 3D API. Copyright: Cyril Zeller, NVIDIA Corporation, 2004.
URL: http://download.nvidia.com/developer/presentations/2004/Eurographics/EG_04_3DAPI.pdf, 11.01.2005

Zeller 2004b
Zeller C: EuroGraphics 2004 Tutorial 5: Programming Graphics Hardware – Introduction to the Hardware Graphics Pipeline. Copyright: Cyril Zeller, NVIDIA Corporation, 2004.
URL: http://download.nvidia.com/developer/presentations/2004/Eurographics/EG_04_IntroductionToGPU.pdf, 12.01.2005

Leòn 1999
Leòn D: Bioinformatics: Database and Research Resources for your Laboratory, 1999. Copyright: Darryl Leòn, 1999.
URL: http://home.san.rr.com/dna/darryl/home.html, 12.02.2005

A Appendix

The Genetic Code

1st Position		2nd Position				3rd Position
		U	C	A	G	
	U	UUU Phe	UCU Ser	UAU Tyr	UGU Cys	U
		UUC Phe	UCC Ser	UAC Ty	UGC Cys	C
		UUA Leu	UCA Ser	UAA *St*	UGA Stop	A
		UUG Leu	UCG Ser	UAG *S*	UGG Trp	G
	C	CUU Leu	CCU Pro	CAU His	CGU Arg	U
		CUC Leu	CCC Pro	CAC His	CGC Arg	C
		CUA Leu	CCA Pro	CAA Gln	CGA Arg	A
		CUG Leu	CCG Pro	CAG Gln	CGG Arg	G
	A	AUU Ile	ACU Thr	AAU Asn	AGU Ser	U
		AUC Ile	ACC Thr	AAC Asn	AGC Ser	C
		AUA Ile	ACA Thr	AAA Lys	AGA Arg	A
		AUG Met	ACG Thr	AAG Lys	AGG Ar	G
	G	GUU Val	GCU Ala	GAU Asp	GGU Gly	U
		GUC Val	GCC Ala	GAC Asp	GGC Gly	C
		GUA Val	GCA Ala	GAA Glu	GGA Gly	A
		GUG Val	GCG Ala	GAG Glu	GGG Gly	G

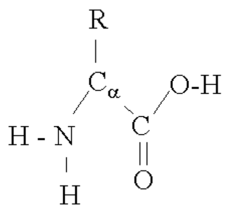
Table 10 - The genetic code table shows which amino acid a nucleic triplet is related to.

The Amino Acids

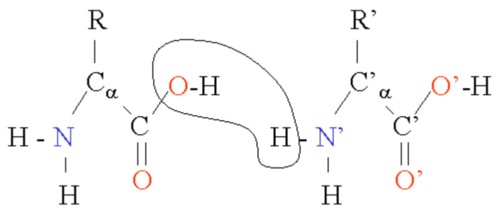
Amino Acid	Three-letter abbreviation	One-letter abbreviation
Alanine	ALA	A
Arginine	ARG	R
Aspartic acid	ASP	D
Asparagine	ASN	N
Cysteine	CYS	C
Glutamic acid	GLU	E
Glutamine	GLN	Q
Glycine	GLY	G
Histidine	HIS	H
Isoleucine	ILE	I
Leucine	LEU	L
Lysine	LYS	K
Methionine	MET	M
Phenylalanine	PHE	F
Proline	PRO	P
Serine	SER	S
Threonine	THR	T
Tryptophan	TRP	W
Tyrosine	TYR	Y
Valine	VAL	V

Table 11 - Amino Acid Code Table taken from Cooper (2000).

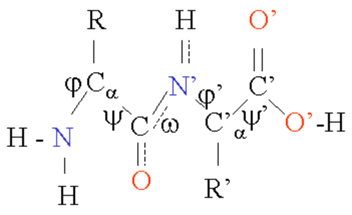
The Structure of Amino Acids



An amino acid consists of a carbon atom (α carbon), a carboxyl group (COO^-), an amino group (NH_3^+), a hydrogen atom and a specific side chain commonly referenced to as residue (R) (Cooper2000, pp. 50-52).



To form a chain they are joined between the α amino group and the α carboxyl group.



Images courtesy of Rupp (2000).

The Protein Structure

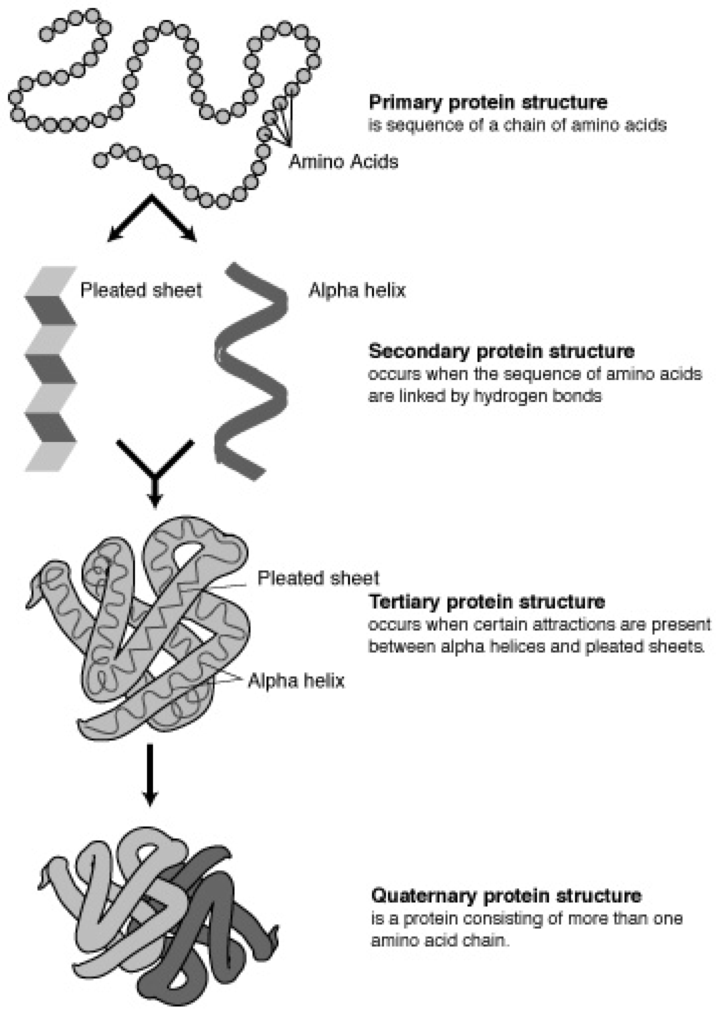

Primary protein structure
is sequence of a chain of amino acids
Amino Acids
Pleated sheet
Alpha helix
Secondary protein structure
occurs when the sequence of amino acids are linked by hydrogen bonds
Pleated sheet
Tertiary protein structure
occurs when certain attractions are present between alpha helices and pleated sheets.
Alpha helix
Quaternary protein structure
is a protein consisting of more than one amino acid chain.

Fig. 68 - The four levels of protein structure.
(Image courtesy of The National Human Genome Research Institute, Rockville Pike.
http://www.genome.gov/Pages/Hyperion//DIR/VIP/Glossary/Illustration/Images/protein.gif)

Swiss-Prot Entry Example

```
ID   UNG_EHV2       STANDARD;      PRT;   255 AA.
AC   P53765;
DT   01-OCT-1996 (Rel. 34, Created)
DT   01-OCT-1996 (Rel. 34, Last sequence update)
DT   01-MAY-2005 (Rel. 47, Last annotation update)
DE   Uracil-DNA glycosylase (EC 3.2.2.-) (UDG).
GN   Name=46;
OS   Equine herpesvirus 2 (strain 86/87) (EHV-2).
OC   Viruses; dsDNA viruses, no RNA stage; Herpesviridae;
OC   Gammaherpesvirinae; Rhadinovirus.
OX   NCBI_TaxID=82831;
RN   [1]
RP   NUCLEOTIDE SEQUENCE.
RX   MEDLINE=95302501; PubMed=7783207;
RA   Telford E.A., Watson M.S., Aird H.C., Perry J., Davison A.J.;
RT   "The DNA sequence of equine herpesvirus 2.";
RL   J. Mol. Biol. 249:520-528(1995).
CC   -!- FUNCTION: Excises uracil residues from the DNA which can
arise as
CC       a result of misincorporation of dUMP residues by DNA poly-
merase or
CC       due to deamination of cytosine.
CC   -!- SIMILARITY: Belongs to the uracil-DNA glycosylase family.
CC
--------------------------------------------------------------------
------
CC   This Swiss-Prot entry is copyright. It is produced through a
collaboration
CC   between  the Swiss Institute of Bioinformatics  and the  EMBL
outstation -
CC   the European Bioinformatics Institute.  There are no  restric-
tions on  its
CC   use as long as its content is in no way modified and this
statement is not
CC   removed.
CC
--------------------------------------------------------------------
------
DR   EMBL; U20824; AAC13834.1; -.
DR   PIR; S55641; S55641.
DR   HSSP; P12295; 3EUG.
DR   InterPro; IPR003249; U_glycsylse_notp.
DR   InterPro; IPR002043; UDNA_glycsylse.
DR   InterPro; IPR005122; UDNA_glycsylseSF.
DR   Pfam; PF03167; UDG; 1.
DR   ProDom; PD001589; U_glycsylse_notp; 1.
DR   TIGRFAMs; TIGR00628; ung; 1.
DR   PROSITE; PS00130; U_DNA_GLYCOSYLASE; 1.
KW   DNA damage; DNA repair; Glycosidase; Hydrolase.
FT   ACT_SITE     90     90       Proton acceptor (By similarity).
SQ   SEQUENCE   255 AA;  29100 MW;  20104402C5297336 CRC64;
     MERWLQLHVW SKDQQDQDQE HLLDEKIPIN RAWMDFLQMS PFLKRKLVTL LETVAKL-
RTS
     TVVYPGEERV FSWSWLCEPT QVKVIILGQD PYHGGQATGL AFSVSKTDPV PPSLRNI-
FLE
     VSACDSQFAV PLHGCLNNWA RQGVLLLNTI LTVEKGKPGS HSDLGWIWFT NYIIS-
CLSNE
     LDHCVFMLWG SKAIEKASLI NTNKHLVLKS QHPSPLAARS NRPSLWPKFL GCGH-
```

```
FKQANE
     YLELHGKCPV DWNLD
//
```

This is entry number P53765 (Uracil-DNA glycosylase) [40] from the Swiss-Prot database, as of 25th of march 2005.

40 http://au.expasy.org/cgi-bin/niceprot.pl?P53765
Link: “View entry in raw text format (no links)”

Comparison of Smith-Waterman, FASTA and BLAST

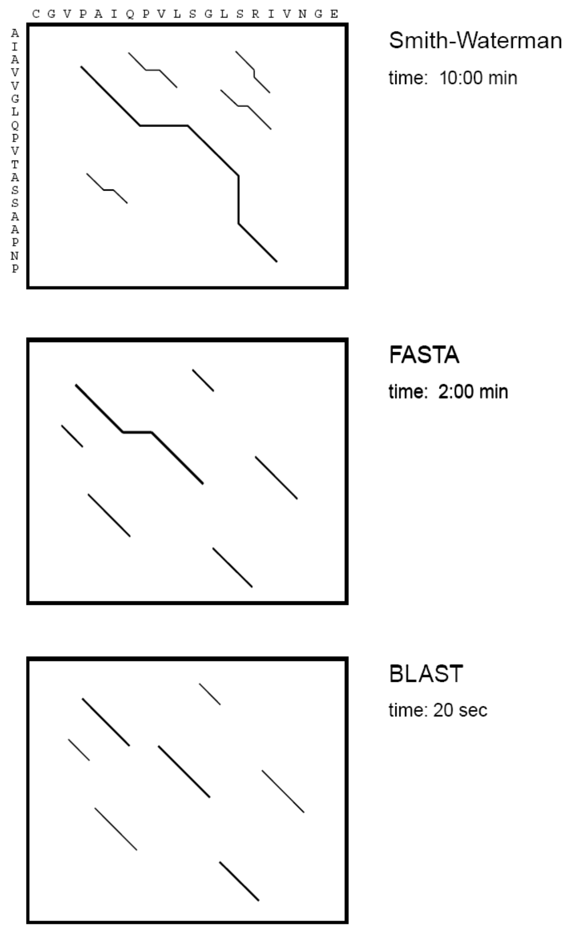

C G V P A I Q P V L S G L S R I V N G E
A I A V V G L Q P V T A S S A A P N P
Smith-Waterman
time: 10:00 min
FASTA
time: 2:00 min
BLAST
time: 20 sec

Fig. 69 - Performance and quality comparison of the Smith-Waterman algorithm, FASTA and BLAST.
(Image courtesy of W.R. Pearson (Pearson 2001))

Each rectangle represents a matrix. The query sequence can be found horizontally on the top margin whereas the comparison sequence is placed vertically on the left side. The black lines are aligned subsequences.

Obviously, the results' quality is reciprocally related to the time for computation. The Smith-Waterman implementation gives the best result, but it needs 30 times as much time as BLAST and 5 times as much time as FASTA. Although FASTA is much slower than BLAST its result is not much better. This is why BLAST is the more common tool.

Unitary Matrix

A	2																						
B	-1	2																					
C	-1	-1	2																				
D	-1	-1	-1	2																			
E	-1	-1	-1	-1	2																		
F	-1	-1	-1	-1	-1	2																	
G	-1	-1	-1	-1	-1	-1	2																
H	-1	-1	-1	-1	-1	-1	-1	2															
I	-1	-1	-1	-1	-1	-1	-1	-1	2														
K	-1	-1	-1	-1	-1	-1	-1	-1	-1	2													
L	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2												
M	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2											
N	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2										
P	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2									
Q	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2								
R	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2							
S	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2						
T	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2					
V	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2				
W	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2			
X	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2		
Y	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	
Z	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**K**	**L**	**M**	**N**	**P**	**Q**	**R**	**S**	**T**	**V**	**W**	**X**	**Y**	**Z**

Table 12 - Example for a unitary matrix for amino acid substitution

BLOSUM62 for Amino Acid Substitution

A	4																							
R	-1	5																						
N	-2	0	6																					
D	-2	-2	1	6																				
C	0	-3	-3	-3	9																			
Q	-1	1	0	0	-3	5																		
E	-1	0	0	2	-4	2	5																	
G	0	-2	0	-1	-3	-2	-2	6																
H	-2	0	1	-1	-3	0	0	-2	8															
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4														
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4													
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5												
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5											
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6										
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7									
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4								
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5							
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11						
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7					
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4				
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4			
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4		
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1	
*****	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1
	A	**R**	**N**	**D**	**C**	**Q**	**E**	**G**	**H**	**I**	**L**	**K**	**M**	**F**	**P**	**S**	**T**	**W**	**Y**	**V**	**B**	**Z**	**X**	*****

Table 13 - BLOSUM62 published by Steven and Jorga Henikoff (1992). It is used by NCBI's BLAST service at http://www.ncbi.nlm.nih.gov/BLAST/.

Vertex Shader Example

```
//
// Vertex shader for procedural bricks
//
// Authors: Dave Baldwin, Steve Koren, Randi Rost
//          based on a shader by Darwyn Peachey
//
// Copyright (c) 2002-2004 3Dlabs Inc. Ltd.
//
// See 3Dlabs-License.txt for license information
//

uniform vec3 LightPosition;

const float SpecularContribution = 0.3;
const float DiffuseContribution  = 1.0 - SpecularContribution;

varying float LightIntensity;
varying vec2  MCposition;

void main(void)
{
    vec3 ecPosition = vec3 (gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - ecPosition);
    vec3 reflectVec = reflect(-lightVec, tnorm);
    vec3 viewVec    = normalize(-ecPosition);
    float diffuse   = max(dot(lightVec, tnorm), 0.0);
    float spec      = 0.0;

    if (diffuse > 0.0)
    {
        spec = max(dot(reflectVec, viewVec), 0.0);
        spec = pow(spec, 16.0);
    }

    LightIntensity  = DiffuseContribution * diffuse +
                      SpecularContribution * spec;

    MCposition      = gl_Vertex.xy;
    gl_Position     = ftransform();
}
```

This vertex shader is part of the GLSL example for procedural bricks (ogl2brick)[41] offered by 3Dlabs[42]. The 3Dlabs license information can be found in appendix "3Dlabs-License.txt". The rendered result is shown in figure 70.

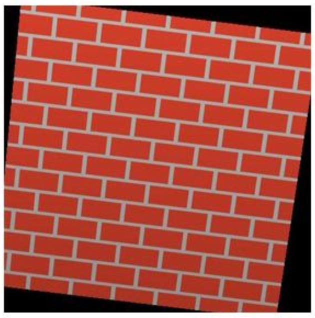


Fig. 70 - Procedural bricks.

41 http://developer.3dlabs.com/downloads/glslexamples/ogl2brick-2.0.zip
42 http://developer.3dlabs.com/downloads/glslexamples/

Fragment Shader Example

```
//
// Fragment shader for procedural bricks
//
// Authors: Dave Baldwin, Steve Koren, Randi Rost
//          based on a shader by Darwyn Peachey
//
// Copyright (c) 2002-2004 3Dlabs Inc. Ltd.
//
// See 3Dlabs-License.txt for license information
//

uniform vec3  BrickColor, MortarColor;
uniform vec2  BrickSize;
uniform vec2  BrickPct;

varying vec2  MCposition;
varying float LightIntensity;

void main(void)
{
    vec3  color;
    vec2  position, useBrick;

    position = MCposition / BrickSize;

    if (fract(position.y * 0.5) > 0.5)
        position.x += 0.5;

    position = fract(position);

    useBrick = step(position, BrickPct);

    color  = mix(MortarColor, BrickColor, useBrick.x * useBrick.y);
    color *= LightIntensity;
    gl_FragColor = vec4 (color, 1.0);
}
```

This fragment shader is part of the GLSL example for procedural bricks (ogl2brick) offered by 3Dlabs2. The 3Dlabs license information can be found in appendix "3Dlabs-License.txt". The rendered result is shown in figure 70.

3Dlabs-Licence.txt

SCA Program Arguments

The command line information output of the application shows the available arguments:

```
Smith-Waterman Sequence Comparison Algorithm
Usage: sca mode [options] [arguments]
options
  rtt     : Use Render-To-Texture (default).
  nortt   : Don't use Render-To-Texture.
  defaults: Use common default values for arguments.
modes
  -c / --compare
     Compare sequences with a reference sequence.
     Usage: sca -c options
     Arguments
       f1 / file2: path and file name of reference sequence
                   default is testRefSeq.txt
       f2 / file2: path and file name of comparison sequences
                   default is testCompSeq.txt
       fp        : path and file name of fragment program
                   default is sca_adv2.fp
     Examples: sca -c f1=refseq.txt f1=compseq.txt
               sca -c defaults fp=simple.fp
  -t / --test\n
     Performance test
     Usage: sca -t [options]
     Options
       ref / refLen     : maximum length of reference sequence *
       comp / compLen   : maximum length of comparison sequences *
       n / numSeq       : maximum number of sequences to be compared
       s / step         : the algorithm will start with sequences of
                          length step and will increase those by step
                          in each iteration until the maximum value
                          is reached. The number of sequences will
                          start with 1.
       i / iterations   : number of iterations per pass to get an
                          average
       stripe           : compute only a defined stripe in the
                          matrix of results in order to safe time.
                          This options needs the position of the
                          stripe stripePos and the stripe width
                          stripeWidth to be defined. Only
                          scenarios where
                            abs(stripePos - (i + j)) > stripeWidth/2)
                            with 0 < i < reflen, 0 < j < compLen
                          will be regarded.
       sp / stripePos   : position of stripe
       sw / stripeWidth: width of stripe
       r / result       : file to write results into
                          default is results.txt
       square           : render only where
                            i=j,
                            with 0 < i < reflen, 0 < j < compLen
       constN           : number of sequences to be compared remains
                          constant
       constSL          : the sequence lengths remain constant
       fp               : path and file name of fragment program
                          default is sca_adv2.fp
       filter           : sets all values in the result matrix 0
                          where i=j
       random/norandom : use random (default) or ordered sequences
                          for testing
     * optimal lengths are -1+2^n. The internal buffer size is then
       2^n.
     Examples: sca -t defaults
               sca -t ref=255 comp=255 s=32 square defaults
                   fp=simple.fp
               sca -t ref=255 comp=255 s=32 n=768 nortt constN
               sca -t defaults stripe sp=700 sw=120 i=5
                   r=resultXY.txt
```

This output is shown at the command line if no, insufficient, or wrong arguments are passed to the application. In such a case, the application quits first.

SCA: results.txt

```
-t ref=2047 comp=2047 n=2048 step=128 defaults constN blosum=blo-
sum62mt2.txt

BEST SCENARIO:
reference  seq length: 1919
comparison seq length: 1151
number of sequences  : 2048
CUPS (total)         : 154,709,768
CUPS (relevant)      : 154,709,768

256 RESULTS (best layer only)
      128         256         384         512         640         ...
128   0127046892  0137888732  0141501091  0141996855  0140254299  ...
256   0138176000  0144437310  0147072000  0148505643  0148845388  ...
384   0143747140  0147723722  0149165378  0149671480  0149261875  ...
512   0144152989  0148423048  0150740964  0151237502  0150853010  ...
640   0146950790  0149646350  0151517949  0151674165  0152293099  ...
768   0148432762  0150812530  0152040063  0151650666  0152360568  ...
896   0148365787  0150436048  0151986075  0152200611  0152986082  ...
1024  0147330126  0150197222  0152378434  0152138432  0153159988  ...
1152  0149163177  0150991771  0152633302  0152301647  0153389152  ...
1280  0150390047  0151771179  0152605284  0152415089  0153446231  ...
1408  0149613684  0151253948  0152963034  0152523814  0153556960  ...
1536  0150887135  0152055842  0152873215  0152325098  0153332686  ...
1664  0149875138  0151541637  0153012386  0152771775  0153629926  ...
1792  0148733057  0151030492  0153115176  0152471459  0153804813  ...
1920  0150383918  0151684359  0153251160  0152837689  0153928383  ...
2048  0149011058  0151098980  0153253226  0152494506  0153912760  ...


BEST SCENARIO of each outer loop iteration (different number of se-
quences):
1919  1151  2048  000154709768     000154709768
```

This is a part of the file results.txt that is generated after sequence alignments. The content was shortened in its width to keep a clear arrangement.

The first line shows the parameters that were passed to the application. Under topic "BEST SCENARIO" features of the fastest alignment scenario in this set of tests is shown. The matrix with values below shows the CUPS reached whereby the used query sequence length and comparison sequence lengths are listed in the first row and the first column. Figure 70 shows how the matrix can be visualized.

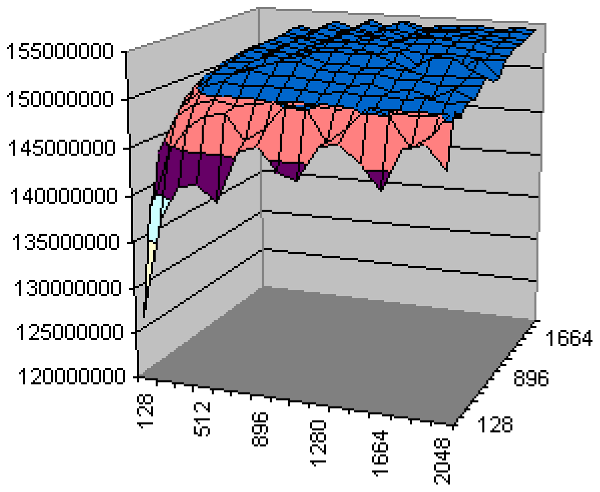


Fig. 71 - Visualisation in form of a 3D diagram.

SCA Output Example

```
sca -c file1=testRefSeq0031.txt file2=swissprot.seq skip=1 defaults


ITERATION 1/1

Loading file: testRefSeq0031.txt
Loading file: swissprot.seq
Skipping 167936 sequences

Successfully loaded sequence files!

refSeqLen       : 31
compSeqLen (max): 2047

sequences to compare: 1324

Creating pbuffer 2048x2048
PBuffer with size 2048x2048 created.
Loading file: blosum_simple.txt
Loaded BLOSUM (30x30)
Loaded fragment program file: sca_adv2.fp
rendering...
testTime: 658

finished!

render time  : 657ms
time for gpu : 430ms
cells     (total): 84017068
cells/sec (total): 127879860
cells     (relevant): 68477357 (81.5%)
cells/sec (relevant): 104227332


 2 RESULTS

(19) >PCX1_MOUSE (Q9QYC1) Pecanex-like protein 1 (Pecanex homolog)
(Fragment)

(18) >MRP3_HUMAN (O15438) Canalicular multispecific organic anion
transporter 2 (Multidrug resistance-associated protein 3) (Multi-
specific organic anion tranporter-D) (MOAT-D)
```

In this example, a query sequence that is loaded from file `testRefSeq0031.txt` is compared to the Swiss-Prot database in file `swissprot.seq`. Skip is set to true which means that skip.txt is read. The number stored in `skip.txt` defines, how many sequences have to be skipped before loading 2048 sequences. In this case, 167936 sequences were skipped. Because the default parameter was used, all parameters that are not defined in this program call are used with their default values. That means, 2048 sequences are loaded from the database, `blosum_simple.txt` is used as scoring matrix file, and `sca_adv2.fp` is used as fragment program file.

The query sequence has a length (refSeqLen) of 31 amino acids. The maximum length of sequences in the loaded sequence set is 2047 amino acids which corresponds the maximum length that can be used. Because the remaining sequences were longer than 2047 amino acids, they could not be loaded.

A buffer width and height of 2048 pixels each were chosen, because this number corresponds to the next power of 2 of both 2047 and 1324. The rendering of 1324×2047×31 matrix cells took 657 ms. That makes 127879860 CUPS out of which only 104227332 were relevant. Thus, 18.5 % redundancy occurred. The cause for redundancy is discussed in the chapter “Performance Test and Evaluation”.

Those two sequences that are most similar to the query sequence are printed at the end of the output. The number 18 and 19 in brackets are the computed similarity value.

B Links

Bioinformatics

Various

NHGRI National Human Genome Research Institute
http://www.genome.gov/

Genetic Databases

EMBL *European Molecular Biology Laboratory*
http://www.ebi.ac.uk/embl/index.html

NCBI *U.S. National Center for Biotechnology Information*
http://ncbi.nih.gov/

DDBJ *DNA Data Bank of Japan*
http://www.ddbj.nig.ac.jp/

Swiss-Prot
http://au.expasy.org/sprot/

PIR *Protein Information Resource*
http://pir.georgetown.edu/home.shtml

PDB *Protein Databank*
http://www.rcsb.org/pdb/

PROSITE
http://au.expasy.org/prosite/

GPGPU

Various

ATI *ATI Developer Website*
http://www.ati.com/developer/
http://www.ati.com/developer/tools.html
http://www.ati.com/developer/radeonSDK.html

Babelshader *Shader converter*
http://graphics.stanford.edu/~danielrh/babelshader.html

Brook *GPGPU-Language (Stanford University)*
http://brook.sourceforge.net

Cg *C for Graphics*
http://developer.nvidia.com/page/cg_main.html

DirectX *Windows Multimedia API Suite*
http://www.microsoft.com/windows/directx/

GLEW *The OpenGL Extension Wrangler Library*
http://glew.sourceforge.net/

GLUT *The OpenGL Utility Toolkit*
http://www.opengl.org/resources/libraries/glut.html

GLSL *The OpenGL Shading Language*
http://www.3dlabs.com/support/developer/ogl2/whitepapers/

GPGPU *General-Purpose Computation Using Graphics Hardware*
http://www.gpgpu.org/
http://www.gpgpu.org/developer/

GPUBench GPU benchmarking with focus on GPGPU
http://graphics.stanford.edu/projects/gpubench/

GPU Gems *Programming Techniques, Tips, and Tricks for Real-Time Graphics*
http://developer.nvidia.com/object/gpu_gems_home.html

HLSL *The D3D Shading Language*
http://msdn.microsoft.com/library/default.asp?url=/library/enus/
directx9_c/directx/graphics/reference/highlevellanguageshaders.asp

IEEE Visualization 2004: GPGPU Tutorial
http://www.gpgpu.org/vis2004/

imdebug *The Image Debugger*
http://www.cs.unc.edu/~baxter/projects/imdebug/

Nvidia *Nvidia Developer Website*
http://developer.nvidia.com/page/home

http://www.developer.nvidia.com/page/tools.html
http://www.developer.nvidia.com/object/sdk_home.html

OpenGL *The Industry's Foundation for High Performance Graphics*
http://www.opengl.org/

OpenGL & Utility Library Specifications
http://www.opengl.org/documentation/spec.html

All About OpenGL Extensions
http://www.opengl.org/resources/features/OGLextensions/

OpenGL Extension Registry
http://oss.sgi.com/projects/ogl-sample/registry/

Sh *GPGPU-Language (University of Waterloo)*
http://libsh.sourceforge.net

ShaderTech *Site with focus on GPU programming*
http://www.shadertech.com

Shadesmith Shader Debugger
http://graphics.stanford.edu/projects/shadesmith

Siggraph 2004: GPGPU Course
http://www.gpgpu.org/s2004

Sample codes and utilities

http://gpgpu.sourceforge.net
http://www.gpgpu.org/developer
http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/OpenGL/
HW_Image_Processing.html
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html

OpenGL Tools

NVShaderPerf (Nvidia)
HLSL, OpenGL fragment shader performance analysis
http://www.developer.nvidia.com/page/tools.html

RenderMonkey (ATI)
HLSL, GLSL shader IDE and performance analysis
http://www.ati.com/developer/tools.html

Babelshader (D. Horn)
Translator: DirectX pixelshader to OpenGL fragment shader
http://www.graphics.stanford.edu/~danielrh/babelshader.html

OpenGL Panther Tools (Apple)
OpenGL vertex and fragment shader IDE, profiling tools
developer.apple.com/opengl/panther.html

OpenGL Shader Designer (Typhoon Labs)
GLSL shader IDE
http://www.typhoonlabs.com

DirectX Tools

FX Composer, NVPerfHUD (NVIDIA)
HLSL shader IDE and performance analysis, real-time statistics
http://www.developer.nvidia.com/page/tools.html

RenderMonkey (ATI)
HLSL, GLSL shader IDE and performance analysis
http://www.ati.com/developer/tools.html

EffectEdit (Microsoft)
Interactive HLSL renderer
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/
directx/graphics/TutorialsAndSamples/Samples/EffectEdit.asp

ShaderWorks (Mad Software)
HLSL shader IDE
http://www.shaderworks.com

Shader Debugger

Visual debugging with the shader IDEs (Windows,MacOS)
FX Composer (DX), RenderMonkey (DX&GL), EffectEdit (DX),
ShaderWorks (DX), Panther Tools (GL), Shader Designer (GL)

Shader Debugger Tool (Microsoft)
HLSL debugger extension for Visual Studio IDE
msdn.microsoft.com/library/default.asp?url=/library/enus/
directx9_c/directx/graphics/tools/shaderdebugger.asp

Imdebug – The Image Debugger (B. Baxter)
Analysis of images output by shaders, easy integration
www.cs.unc.edu/~baxter/projects/imdebug/
Shadesmith (T. Purcell, P. Sen)
Interactive OpenGL fragment shader debugger
graphics.stanford.edu/projects/shadesmith/

Microsoft Shading and Debugging Tool
http://www.msdn.microsoft.com/library/default.asp?url=/library/en-
us/directx9_c/directx/graphics/Tools/Tools.asp

Dicussion

http://www.gpgpu.org/forums
http://www.shadertech.com

Conferences

CGI2005 *Computer Graphics International 2005*
http://www.cs.stonybrook.edu/~cgi05/

EGPGV04 *Eurographics Symposium on Parallel Graphics and Visualization 2004*
http://www-id.imag.fr/EGPGV04/

Eurographics E*uropean Association for Computer Graphics*
http://www.eg.org/

EUROVIS 2005 *Eurographics / IEEE VGTC Symposium on Visualization*
http://www.comp.leeds.ac.uk/eurovis/

GDC *Game Developers Conference*
http://www.gdconf.com/

GraphiCon *International Conference on Computer Graphics & Vision*
http://www.graphicon.ru/

Graphics Hardware
http://www.graphicshardware.org/

Graphite 2005
http://www.cs.otago.ac.nz/graphite/

IEEE Visualization
http://vis.computer.org/

IEEE VR2005 *IEEE Virtual Reality Conference*
http://www.vr2005.org/

Joint Eurographics *IEEE TCVG Symposium on Visualization*
http://www.inf.uni-konstanz.de/cgip/vissym04/index.shtml

Pacific Graphics 2004
http://graphics.snu.ac.kr/pg2004/index.html

SEAGRAPH *Conference for Computer Graphics South East Asia*
http://www.seagraph.org/

SIGGRAPH
http://www.siggraph.org/

TP.CG.04 *Theory and Practice of Computer Graphics 2004 Conference*
http://www.eguk.org.uk/TPCG04/