Bio-Sequence Database Scanning on a GPU

Weiguo Liu, Bertil Schmidt, Gerrit Voss, Andre Schroder, and Wolfgang Muller-Wittig

Nanyang Technological University
School of Computer Engineering
Centre for Advanced Media Technology
Singapore 639798
{liuweiguo,asbschmidt}@ntu.edu.sg

Abstract

Protein sequences with unknown functionality are often compared to a set of known sequences to detect functional similarities. Efficient dynamic programming algorithms exist for this problem, however current solutions still require significant scan times. These scan time requirements are likely to become even more severe due to the rapid growth in the size of these databases. In this paper, we present a new approach to bio-sequence database scanning using computer graphics hardware to gain high performance at low cost. To derive an efficient mapping onto this type of architecture, we have reformulated the Smith-Waterman dynamic programming algorithm in terms of computer graphics primitives. Our OpenGL implementation achieves a speedup of approximately sixteen on a high-end graphics card over available straightforward and optimized CPU Smith-Waterman implementations.

1. Introduction

Scanning protein sequence databases is a common and often repeated task in molecular biology. The need for speeding up these searches comes from the rapid growth of the bio-sequence banks: every year their size is scaled by a factor 1.5 to 2. The scan operation consists of finding similarities between a particular query sequence and all sequences of a bank. This allows biologists to identify sequences sharing common subsequences, which from a biological viewpoint have similar functionality. Comparison algorithms whose complexities are quadratic with respect to the length of the sequences detect similarities between the query sequence and a subject sequence. One frequently used approach to speed up this time consuming operation is to introduce heuristics in the search algorithm. The drawback is that the more efficient the heuristics, the worse is the result. Another approach to get high quality results in a short time is to use high performance computing. In this paper, we investigate how commodity computer graphics hardware can be used as a computational platform to accelerate database scanning.

The fast increasing power of the GPU (*Graphics Processing Unit*) and its streaming architecture opens up a range of new possibilities for a variety of applications. With the enhanced programmability of commodity GPUs, these chips are now capable of performing more than the specific graphics computations they were originally designed for. Recent work on GPGPU (General-Purpose computation on GPUs) shows the design and implementation of algorithms for non-graphics applications. Examples include scientific computing [10], computational geometry [1], image processing [22], and bioinformatics [8, 3]. The evolution of GPUs is driven by the computer game market. This leads to a relatively small price per unit and to very rapid developments of next generations.

Currently, the peak performance of high-end GPUs such as the GeForce 7800 GTX is approximately ten times faster than that of comparable CPUs. Further, GPU performance has been increasing from two to two-and-a-half times a year (see Figure 1). This growth rate is faster than Moore's law as it applies to CPUs, which corresponds to about one-and-a-half times a year [12]. Consequently, GPUs will become an even more attractive alternative for high performance computing in the near future.

However, there are still a number of challenges to be solved in order to enable scientists other than computer graphics specialists to facilitate efficient usage of these

1-4244-0054-6/06/$20.00 ©2006 IEEE

resources within their research area. The biggest challenge in order to solve a specific problem using GPUs is reformulating the proposed algorithms and data structures using computer graphics primitives (e.g. triangles, textures, vertices, fragments). Furthermore, restrictions of the underlying streaming architecture have to be taken into account, e.g. random access writes to memory is not supported and no cross fragment data or persistent state is possible (e.g. all the internal registers are flushed before a new fragment is processed). In this paper we show how bio-sequence database scanning based on the Smith-Waterman dynamic programming algorithm can benefit from this type of computing power.

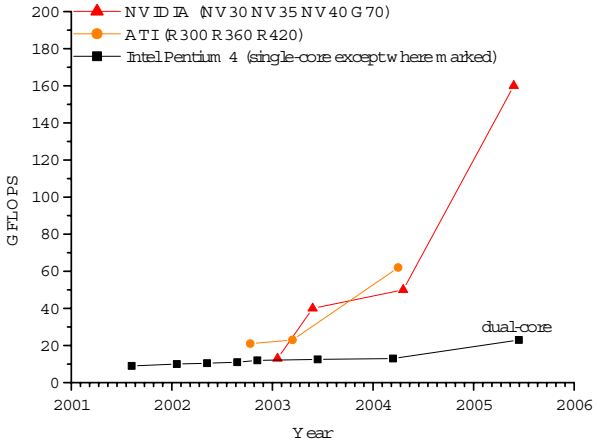

Figure 1. Peak performance comparison (measured on the multiply-add instruction) of GPUs and CPUs in recent years. Figure taken from Owens et al. [14].

The rest of this paper is organized as follows. In Section 2, we introduce the basic sequence alignment algorithm. Section 3 highlights previous work on parallelization of this algorithm on different parallel architectures. Important features of the GPU architectures are described in Section 4. Section 5 presents our mapping of the algorithm onto the GPU architecture. A performance evaluation is given in Section 6. Finally, Section 7 concludes the paper with an outlook to further research topics.

2. Smith-Waterman Algorithm

Surprising relationships have been discovered between protein sequences that have little overall similarity but in which similar subsequences can be found. In that sense, the identification of similar subsequences is probably the most useful and practical method for comparing two sequences. The Smith-Waterman algorithm [19] finds the most similar subsequences of two sequences (the local alignment) by dynamic programming (DP). The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment into another. Two elementary operations are used: substitution and insertion/deletion (also called a gap operation). Through series of such elementary operations, any segments can be transformed into any other segment. The smallest number of operations required to change one segment into another can be taken into as the measure of the distance between the segments.

Consider two strings S_1 and S_2 of length l_1 and l_2. To identify common subsequences, the Smith-Waterman algorithm computes the similarity $H(i,j)$ of two sequences ending at position i and j of the two sequences S_1 and S_2. The computation of $H(i,j)$, for $1 \leq i \leq l_1$, $1 \leq j \leq l_2$, is given by the following recurrences:

$$H(i,j) = \max\{0, E(i,j), F(i,j), H(i-1,j-1) + sbt(S_1[i], S_2[j])\}$$
$$E(i,j) = \max\{H(i,j-1) - \alpha, E(i,j-1) - \beta\},$$
$$F(i,j) = \max\{H(i-1,j) - \alpha, F(i-1,j) - \beta\},$$

where sbt is a character substitution cost table. Initialization of these values are given by $H(i,0) = E(i,0) = H(0,j) = F(0,j) = 0$ for $0 \leq i \leq l_1$, $0 \leq j \leq l_2$. Multiple gap costs are taken into account as follows: α is the cost of the first gap; β is the cost of the following gaps. This type of gap cost is known as *affine gap penalty*. Some applications also use a *linear gap penalty*, i.e. $\alpha = \beta$. For linear gap penalties the above recurrence relations can be simplified to:

$$H(i,j) = \max\{0, H(i,j-1) - \alpha, H(i-1,j) - \alpha, H(i-1,j-1) + sbt(S_1[i], S_2[j])\},$$

Each position of the matrix H is a similarity value. The two segments of S_1 and S_2 producing this value can be determined by a trace-back procedure. Figure. 2 illustrates an example.

3. Related Work

A number of parallel architectures have been developed for bio-sequence database scanning with the Smith-Waterman algorithm. In addition to architectures specifically designed for sequence analysis, existing programmable sequential and parallel architectures

	\	A	T	C	T	C	G	T	A	T	G	A	T
\	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	1	0	0	2	1	0
T	0	0	2	1	**2**	1	1	4	3	2	1	1	3
C	0	0	1	4	3	**4**	**3**	3	3	2	1	0	2
T	0	0	2	3	6	5	4	**5**	4	5	4	3	2
A	0	2	2	2	5	5	4	4	**7**	6	5	6	5
T	0	1	4	3	4	4	4	6	5	**9**	8	7	8
C	0	0	3	6	5	6	5	5	5	8	**8**	7	7
A	0	2	2	5	5	5	5	4	7	7	7	**10**	9
C	0	1	1	4	4	7	6	5	6	6	6	9	9

Figure 2. Example of the Smith-Waterman algorithm to compute the local alignment between two DNA sequences ATCTCGTATGAT and GTCTATCAC. The matrix $H(i,j)$ is shown for the linear gap cost $\alpha = 1$, and a substitution cost of $+2$ if the characters are identical and -1 otherwise. From the highest score ($+10$ in the example), a traceback procedure delivers the corresponding alignment, the two subsequences TCGTATGA and TCTATCA.

have been used for solving sequence alignment problems.

Special-purpose architectures can provide the fastest means of running a particular algorithm with very high processing element (PE) density. Each PE is specifically designed for the computation of one cell in the DP matrix (see Figure 2). However, such architectures are limited to one single algorithm, and thus cannot supply the flexibility necessary to run a variety of algorithms required for analyzing DNA, RNA, and proteins. P-NAC was the first such machine and computed edit distance over a four-character alphabet [11]. More recent examples, better tuned to the needs of computational biology, include BioScan [18], BISP [4], and SAMBA [6].

An approach presented in [17] is based on instruction systolic arrays (ISAs). ISAs combine the speed and simplicity of systolic arrays with flexible programmability. Several other approaches are based on the SIMD concept, e.g. MGAP [2], Kestrel [5], and Fuzion [17]. SIMD and ISA architectures are programmable and can be used for a wider range of applications, such as image processing and scientific computing. Since these architectures contain more general-purpose parallel processors, their PE density is less than the density of special-purpose ASICs. Nevertheless, SIMD solutions can still achieve significant runtime savings. However, the costs involved in designing and producing SIMD architectures are quite high. As a consequence, none of the above solutions has a successor generation, making upgrading impossible.

Reconfigurable systems are based on programmable logic such as field-programmable gate arrays ($FPGAs$). They are generally slower and have lower PE densities than special-purpose architectures, e.g. [13, 23]. They are flexible, but the configuration must be changed for each algorithm, which is generally more complicated than writing new code for a programmable architecture. Solutions based on FPGAs have the additional advantage that they can be regularly upgraded to state-of-the-art-technology.

All these approaches can be seen as accelerators - an approach satisfying the demand for a low cost solution to compute-intensive problems. The main advantage of GPUs compared to the architectures mentioned above is that they are commodity components. In particular, most users have already access to PCs with modern graphics cards. For these users this direction provides a zero-cost solution. Even if a graphics card has to be bought, the installation of such a card is trivial (plug and play). Writing the software for such a card does still require specialist knowledge, but new high level languages such as Cg [21] offer a simplified programming environment.

4. GPU Architecture

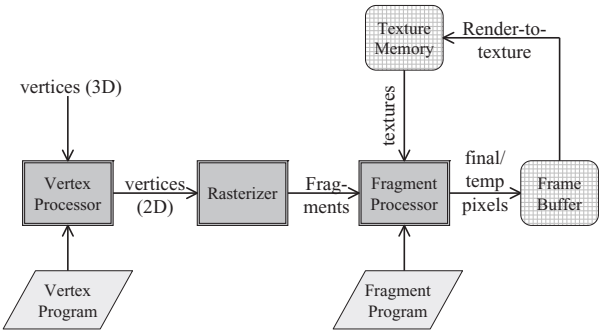

Figure 3. Illustration of the GPU graphics pipeline.

Computation on a GPU follows a fixed order of processing stages, called the graphics pipeline (see Figure 3). The pipeline consists of three stages: vertex processing, rasterization and fragment processing. The

vertex processing stage transforms three-dimensional vertex world coordinates into two-dimensional vertex screen coordinates. The rasterizer then converts the geometric vertex representation into an image fragment representation. Finally, the fragment processor forms a color for each pixel by reading texels (texture pixels) from the texture memory. Modern GPUs support programmability of the vertex and fragment processor. Fragment programs for instance can be used to implement any mathematical operation on one or more input vectors (textures or fragments) to compute the color of a pixel.

In order to meet the ever increasing performance requirements set by the gaming industry, modern GPUs use two types of parallelism. Firstly, multiple processors work on the vertex and fragment processing stage, i.e. they operate on different vertices and fragments in parallel. For example, a typical mid-range graphics card such as the nVidia GeForce 6800 GT has 6 vertex processors and 16 fragment processors. Secondly, operations on 4-dimensional vectors (the four channels Red/Green/Blue/Alpha (RGBA)) are natively supported without performance loss.

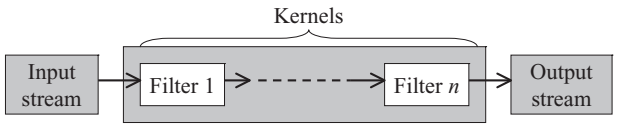

Figure 4. Streaming model that applies kernels to an input stream and writes to an output stream.

Several authors have described modern GPUs as *streaming processors*, e.g [20]. Streaming processors read an input stream, apply kernels (filters) to the stream and write the results into an output stream. In case of several kernels, the output stream of the leading kernel is the input stream for the following kernel (see Figure 4). The vast majority of general-purpose GPU applications use only fragment programs for their computation. In this case textures are considered as input streams and the texture buffers are output streams. Because fragment processors are SIMD architectures, only one program can be loaded at a time. Applying several kernels thus means to do several passes.

A typical GPGPU program is structured as follows [14, 7].

1. Data-parallel sections of the application are identified by the programmer. Each such section can be considered a kernel and is implemented as a fragment program. The input and output of each kernel is one or more data arrays, which are stored in textures in GPU memory.

2. To invoke a kernel, the range of the computation (or the size of the output stream) must be specified. The programmer does this by passing vertices to the GPU. A typical GPGPU invocation is a quadrilateral (quad) oriented parallel to the image plane, sized to cover a rectangular region of pixels matching the desired size of the output array.

3. The rasterizer generates a fragment for every pixel location in the quad, producing thousands to millions of fragments.

4. Each of the generated fragments is then processed by the active kernel fragment program. The fragment program can read from arbitrary texture memory locations but can only write to memory locations corresponding to the location of the fragment in the frame buffer.

5. The output of the fragment program is a value (or vector of values) per fragment. This output may be the final result of the application, or it may be stored as a texture and then used in subsequent passes. This feedback loop is realized by using the output buffer of a completed pass as input texture for the following one (known as *render-to-texture* (RTT)).

5. Mapping Onto the GPU Architecture

In this section we describe how the Smith-Waterman algorithm can be efficiently mapped onto a GPU. We take advantage of the fact that all elements in the same anti-diagonal of the DP matrix can be computed independent of each other in parallel (see Figure 5). Thus, the basic idea is to compute the DP matrix in anti-diagonal order. The anti-diagonals are stored as textures in the texture memory. Fragment programs are then used to implement the arithmetic operations specified by the recurrence relation.

Assuming, we are aligning two sequences of length M and K with affine gap penalties on a GPU. As a preprocessing step both sequences and the substitution matrix are loaded into the texture memory. We are then computing the DP matrix in $M + K - 1$ rendering passes. In rendering pass k, $1 \leq k \leq M+K-1$, the values $H(i,j)$, $E(i,j)$, and $F(i,j)$ for all i,j with $1 \leq i \leq M$, $1 \leq j \leq K$ and $k = i + j - 1$ are computed by the fragment processors. The new anti-diagonal is

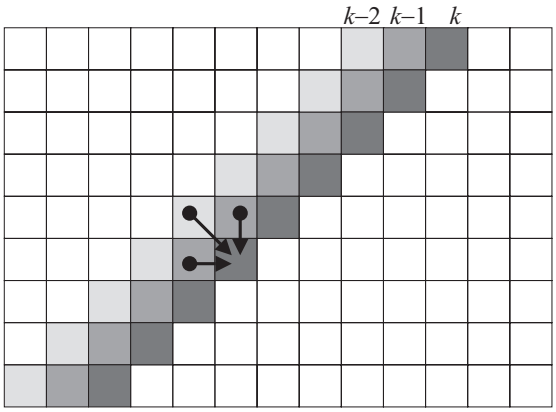

Figure 5. Data dependency relationship in the Smith-Waterman DP matrix: each cell (i, j) depends on its left neighbor $(i, j-1)$, upper neighbor $(i-1, j)$ and upper left neighbor $(i-1, j-1)$. Therefore all cells along anti-diagonal k can be computed in parallel from the anti-diagonals $k-1$ and $k-2$.

stored in the texture memory as a texture. The subsequent rendering pass then reads the two previous anti-diagonals from this memory.

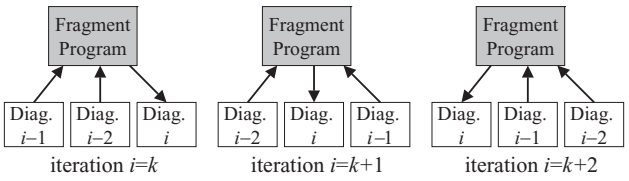

Figure 6. Cyclic change of the functions of buffers A, B, and C for computation of anti-diagonals in the DP matrix.

Since diagonal k depends on the diagonals $k-1$ and $k-2$, we store these three diagonals as separate buffers. We are using a cyclic method to change the buffer function as follows: Diagonals $k-1$ and $k-2$ are in the form of texture input and diagonal k is the framebuffer. In the subsequent iteration, k becomes $k-1$, $k-1$ becomes $k-2$, and $k-2$ becomes k. This is further illustrated in Figure 6. 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.

Another concern is the way to map each diagonal in the DP matrix into a quad. From Figure 7 we can see that drawing a diagonal quad that covers the diagonal in the buffer does not yield the desired result. This is because all pixels touched by the quad are rendered instead of rendering only the cells on the diagonal.

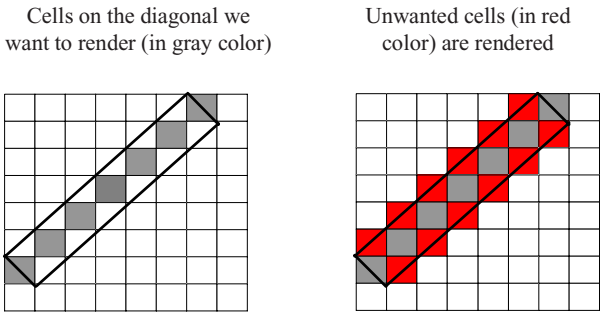

Figure 7. Drawing a quad over the anti-diagonal results in rendering too many pixels.

In our implementation, 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 8).

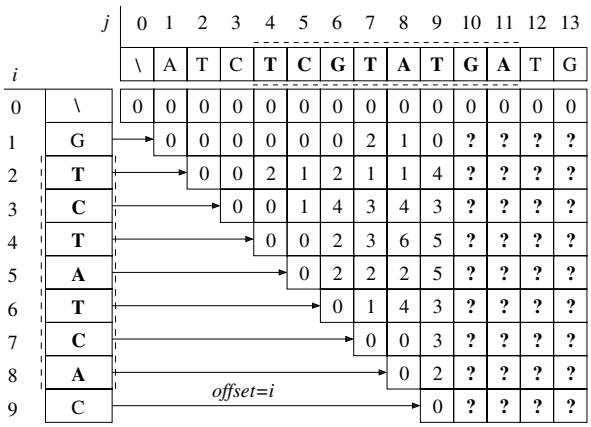

Figure 8. Matrix with shifted rows.

With this method, a $1 \times L(k)$ quad can be rendered in each iteration, where $L(k)$ denotes the length of diagonal k. Furthermore, it is possible to perform N pairwise comparisons at the same time by using two-dimensional buffers instead of one-dimensional buffers. This is shown in Figure 9 in which the buffer is filled from bottom up. Each buffer contains N diagonals of length $L(k)$, where $L(k)$ denotes the length of diagonal k. The computation is invoked by drawing an $N \times L(k)$ quad in each iteration.

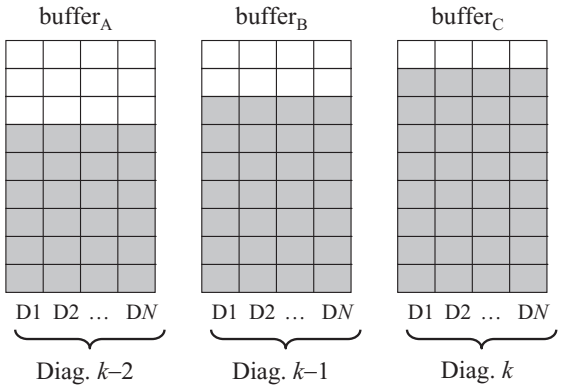

Figure 9. Buffers at iteration step k=8 after rendering N diagonals of length 8 each in parallel (in different fragment processors).

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

Doing Smith-Waterman Alignment
        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
                End Passes Loop
        deactivate fragment program
        read results from framebuffer
End Alignment
```

Figure 10. Our GPU implementation for the Smith-Waterman algorithm with multiple passes.

Figure 10 gives an overview of our GPU implementation. 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 diagonal. First, the quad and its texture coordinates are determined and the two buffers representing the diagonals D_{k-1} and D_{k-2} are bound as textures. The render target is set to the buffer that represents diagonal D_k 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.

Note that the purpose of our Smith-Waterman GPU implementation is the acceleration of sequence database scanning. This application requires the alignment of a query sequences to all subject sequences of a given database. All subject sequences are ranked by the maximum score in the DP matrix. Reconstruction of the actual alignment (the traceback) is merely performed for the highest ranked sequences. Therefore, only the highest score of each pairwise alignment is computed on the GPU. Ranking the compared sequences and reconstructing the alignments are carried out by the front end PC. Because this last operation is only performed for very few subject sequences, its computation time is negligible.

6. Performance Evaluation

We have implemented the proposed algorithm using the high-level GPU programming language *GLSL* (OpenGL Shading Language) [9] and evaluated it on the following graphics cards:

- *nVidia GeForce 6800 GTO*: 414 MHz engine clock speed, 1.10 GHz memory clock speed, 5 vertex processors, 16 fragment processors, 256 MB memory.

- *nVidia GeForce 7800 GTX*: 622 MHz engine clock speed, 1.83 GHz memory clock speed, 8 vertex processors, 24 fragment processors, 512 MB memory.

Tests have been conducted with these cards installed in a PC with an Intel Pentium4 3.0 GHz, 1GB RAM running Windows XP.

A performance measure commonly used in computational biology is *cell updates per second* (*CUPS*). A CUPS represents the time for a complete computation of one entry of the matrix H, including all comparisons, additions and maxima computations. We have scanned the Swiss-Prot protein databank (release 46.3, which contains 176,469 sequences with an average length of 361) for query sequences of various lengths using our implementation on a GeForce 6800 GTO and a GeForce 7800 GTX. They allow handling query sequences up to a length of 4096. This restriction is imposed by the maximum texture buffer size of these graphics cards. However, this limitation is not severe since 99.8% of the sequences in the Swiss-Prot database are of length <4096. Furthermore, it is expected that the texture

Table 1. Runtime comparison for scanning the Swiss-Prot database (release 46.3, March 2005). The query sequences have accession numbers O29181, P03630, P53765, Q8ZGB4, P58229, P39985, Q96HP0 and P36022 in the Swiss-Prot.

Query Sequence Length	OSEARCH (sec)	SSEARCH (sec)	GeForce 6800 GTO (sec)	GeForce 7800 GTX (sec)
63	90.9	48.6	32.4	19.5
127	184.1	99.7	44.4	25
255	375.1	198.2	68.8	36.3
361	532.7	295.8	89.1	45.8
511	731.9	391.9	117.2	59.2
1023	1472.5	848.7	213.6	105.1
2047	2985.7	1741.7	407.2	197.9
4095	5925.2	3610.8	793.2	383.1

buffer sizes will increase in next-generation graphics hardware.

We have also compared the performance between our GPU implementation and a widely used CPU program for database scanning - FASTA [16]. FASTA stands for FAST-All, reflecting the fact that it can be used for a fast protein comparison or a fast nucleotide comparison between a query sequence and a large database of known sequences. OSEARCH and SSEARCH [15] are two Smith-Waterman implementations in FASTA programs. OSEARCH is a straightforward Smith-Waterman implementation. SSEARCH [15] is an optimized implementation of Smith-Waterman algorithm. However, OSEARCH is more sensitive and accurate. We have run OSEARCH34 and SSEARCH34 on an Pentium4 3.0 GHz CPU running Red Hat Linux 7.3.

Table 1 reports the runtime of our GPU implementation, OSEARCH, and SSEARCH for different query sequence lengths. Figure 11 compares the corresponding MCUPS performance values. As can be seen, our GPU implementation achieve speedups of almost 16 compared to OSEARCH and 8 compared to SSEARCH while produce the same accuracy as OSEARCH, which has higher quality than SSEARCH.

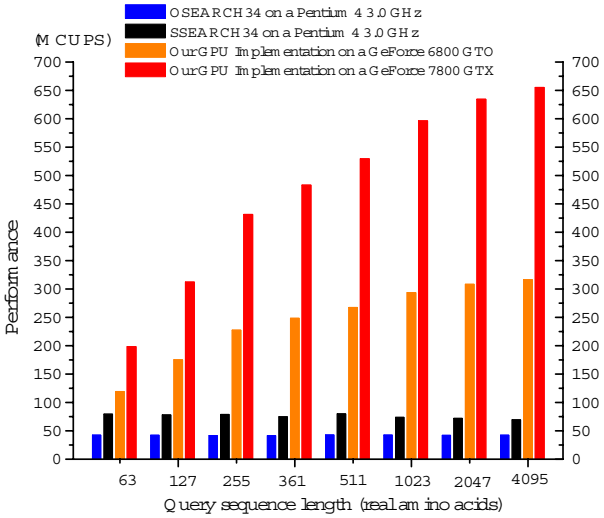

Figure 11. Performance comparison (in MCUPS) for scanning the Swiss-Prot database (release 46.3, March 2005). The query sequences have accession numbers O29181, P03630, P53765, Q8ZGB4, P58229, P39985, Q96HP0 and P36022 in the Swiss-Prot.

7. Conclusions and Future Work

In this paper we have demonstrated that the streaming architecture of GPUs can be efficiently used for biological sequence database scanning. To derive an efficient mapping onto this type of architecture, we have reformulated the Smith-Waterman algorithm in terms of computer graphics primitives. The evaluation of our implementation on a high-end graphics card shows a speedup of almost sixteen compared to a Pentium IV 3.0GHz. To our knowledge this is the first reported implementation of the Smith-Waterman algorithm on graphics hardware.

The very rapid growth of genomic databases demands even more powerful parallel solutions in the future. Because comparison and alignment algorithms that are favored by biologists are not fixed, programmable parallel solutions are required to speed up these tasks. As an alternative to inflexible special-

purpose systems, hard-to-upgrade SIMD systems, and expensive supercomputers, we advocate the use of graphics hardware platforms. The main advantage of graphics hardware compared to previously used architectures for biological sequence analysis is that they are commodity components. This facilitates easy upgrading to next-generation GPUs at a very reasonable price.

References

[1] P. Agarwal, S. Krishnan, N. Mustafa, and S. Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *Proc. 11th European Symposium on Algorithms*, 2003.

[2] M. Borah, R. Bajwa, S. Hannenhalli, and M. Irwin. A simd solution to the sequence comparison problem on the mgap. In *Proc. ASAP'94*, pages 144–160, 1994.

[3] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In *Proceedings of 10th Panhellenic Conference on Informatics*, 2005.

[4] E. Chow, T. Hunkapiller, J. Peterson, and M. Waterman. Biological information signal processor. In *Proc. ASAP'91*, pages 144–160, 1991.

[5] A. e. a. Di Blas. The ucsc kestrel parallel processor. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):80–92, 2005.

[6] P. Guerdoux-Jamet and D. Lavenier. Samba: hardware accelerator for biological sequence comparison. *CABIOS*, 12(6):609–615, 1997.

[7] M. Harris. *GPU Gems 2*, chapter 31, pages 493–508. Addison Wesley, 2005.

[8] D. Horn, Houston, and P. M., Hanrahan. Clawhmmer: a streaming hmmer-search implementation. In *Proceedings of Supercomputing 2005*, 2005.

[9] K. J., B. D., and R. R. The opengl shading language, document revision 59. Technical report, http://www.opengl.org/documentation/oglsl.html, 2005.

[10] K. J. and W. R. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph*, 22:908–916, 2003.

[11] D. Lopresti. P-nac: A systolic array for comparing nucleic acid sequences. *Computer*, 20(7):98–99, 1987.

[12] D. Manocha. General-purpose computations using graphics processors. *Computer*, 38(8):85–88, 2005.

[13] T. Oliver, B. Schmidt, and D. Maskell. Hyper customized processors for bio-sequence database scanning on fpgas. In *13th ACM International Symposium on Field-Programmable Gate Arrays*, 2005.

[14] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005*, pages 21–51, 2005.

[15] W. Pearson. Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms. *Genomics*, 11:635–650.

[16] W. Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods Enzymol*, 183:63–98, 1990.

[17] B. Schmidt, H. Schroder, and M. Schimmler. Massively parallel solutions for molecular sequence analysis. In *Proc. 1st IEEE Int. Workshop on High Performance Computational Biology*, 2002.

[18] R. e. a. Singh. Bioscan: a network sharable computational resource for searching biosequence databases. *CABIOS*, 12(3):191–196, 1996.

[19] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[20] P. T.J., B. I., M. W.R., and H. P. Ray tracing on programmable graphics hardware. *ACM Trans Graph*, pages 703–712, 2002.

[21] M. W.R., G. R.S., A. K., and K. M.J. Cg: A system for programming graphics hardware in a c-like language. *ACM Trans Graph*, 22:896–907, 2003.

[22] F. Xu and K. Mueller. Ultra-fast 3d filtered backprojection on commodity graphics hardware. In *IEEE International Symposium on Biomedical Imaging'04*, 2004.

[23] Y. Yamaguchi, T. Maruyama, and A. Konagaya. High speed homology search with fpgas. In *Proc.PSB'02*, pages 271–282, 2002.