33rd Annual International Conference of the IEEE EMBS
Boston, Massachusetts USA, August 30 - September 3, 2011

GPU-Accelerated Protein Sequence Alignment

Laiq Hasan Marijn Kentie Zaid Al-Ars
Computer Engineering Lab, TU Delft, The Netherlands
E-mail: L.HASAN@TUDELFT.NL

***Abstract*— *Smith-Waterman (S-W)* algorithm is an optimal sequence alignment method and is widely used for genetic databases. This paper presents a *Graphics Processing Units (GPUs)* accelerated S-W implementation for protein sequence alignment. The paper proposes a new sequence database organization and several optimizations to reduce the number of memory accesses. The new implementation achieves a performance of 21.4 GCUPS, which is 1.13 times better than the state-of-the-art implementation on an NVIDIA GTX 275 graphics card.**

***Index Terms*— Bioinformatics, Protein Sequence Alignment, Database Organization, Smith-Waterman Algorithm, GPUs**

I. Introduction

Sequence alignment is used to identify regions of similarity between DNA or protein sequences. This similarity may be a consequence of functional, structural or evolutionary relationships between the sequences. Various methods are available for local and global sequence alignment [1]. Heuristics based approaches like BLAST, FASTA and HMMER [2–4] are fast, but they do not guarantee an optimal alignment. Although slow in aligning long sequences, the *Smith-Waterman (S-W)* algorithm [5], based on *dynamic programming (DP)* [6], is a method that finds an optimal local alignment between two DNA or protein sequences, i.e. the *query sequence* and the *database sequence*. To develop efficient and optimal sequence alignment solutions, the S-W algorithm has recently been implemented on emerging accelerator platforms such as FPGAs, Cell/BEs and GPUs [7–16]. The fastest GPU implementation, i.e. 'CUDASW++ 2.0' [16], achieves a performance of 17 GCUPS on a single GTX 280 GPU, outperforming CPU-based BLAST in its benchmarks.

The S-W algorithm with affine gap penalties [17] is given by Equation 1, where α, β are the gap opening and extension penalties, respectively. Further, $H_{0,0} = D_{0,0} = E_{0,0} = H_{i,0} = D_{i,0} = E_{i,0} = H_{0,j} = D_{0,j} = E_{0,j} = 0$, for $1 \leq i \leq M$ and $1 \leq j \leq N$, where M and N are the lengths of the sequences to be aligned.

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + S_{i,j} \\ D_{i,j} \\ E_{i,j} \end{cases} \tag{1}$$

$$\text{where} \quad D_{i,j} = \max \begin{cases} H_{i-1,j} - \alpha \\ D_{i-1,j} - \beta \end{cases}$$

$$\text{and} \quad E_{i,j} = \max \begin{cases} H_{i,j-1} - \alpha \\ E_{i,j-1} - \beta \end{cases}$$

In this paper, we present a GPU-accelerated S-W implementation for protein sequence alignment. The implementation pre-converts the reference protein database to a custom GPU format. Like other GPU implementations, the time consuming matrix fill step of the S-W algorithm is implemented and accelerated on the GPU. The performance is enhanced by restructuring the entire database and optimizing its organization. Furthermore, memory accesses are optimized to eliminate bandwidth bottlenecks. The results demonstrate that the new implementation achieves a performance of 21.4 GCUPS, which is 1.13 times better than the state-of-the-art implementation on an NVIDIA GTX 275 graphics card.

The remainder of the paper is organized as follows: Section II presents the GPU-accelerated S-W implementation for protein sequence alignment. Section III discusses the results and compares the performance of our implementation with other solutions. Section IV concludes the paper.

II. GPU-Accelerated implementation

A. General design

Being the most mature GPU programming toolkit to date, NVIDIA *Compute Unified Device Architecture (CUDA)* [18] is used for the GPU programming *(device code)* in conjunction with C++ for the PC programming *(host code)*. Like with other existing GPU implementations, protein sequences from the Swiss-Prot database [19] are considered for alignment. The reason is that protein alignment is more complex than the DNA version, which makes supporting DNA alignments later on relatively simple. Figure 1 shows a block diagram description of the implementation. The host code is mostly concerned with loading data structures, copying them to the GPU, and copying back and presenting the results. The query sequence, converted database and other data are copied to the GPU. Then the device code is launched, which aligns the query sequence with the database sequences using the S-W algorithm.

Like other GPU implementations, our implementation returns maximum S-W scores instead of the actual alignments. Skipping the algorithm's traceback step significantly simplifies and speeds up the implementation. Furthermore, as no data structures like pointer lists need to be kept, memory consumption is decreased as well. However, to be able to generate full alignments, a number of top-scoring sequences are exported to a new database file. The sequences in this file can then be aligned on the host PC using the *Smith-Waterman search (ssearch)* tool. This approach leads to some redundancy as some sequences are aligned twice, however,

the number of such sequences is relatively small. By default 20 top scoring sequences are returned, whereas the Swiss-Prot database contains more than 500,000.

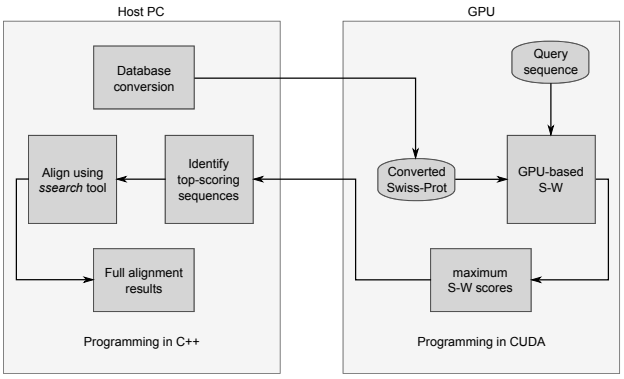

Fig. 1. Description of the GPU implementation

Each processing element in our implementation is used to independently generate a complete alignment between a query sequence and a database sequence. This eliminates the need for inter-processor communication and results in efficient resource utilization. The GPU used for implementation (i.e. NVIDIA GTX 275) contains 240 processors, while the latest release of Swiss-Prot contains more than 500,000 sequences. Hence, it is possible to keep all processors well occupied [20].

B. Database organization

The Swiss-Prot database is organized in FASTA format, where sequences are preceded by sequence descriptions that give names and other biological information about them. Instead of directly loading databases in FASTA format, the GPU implementation converts them to a custom GPU format to better match the device capabilities. A database only needs to be converted once, after which it is locally stored in the new format. The conversion process as shown in Figure 2 consists of the following steps.

1) Sorting: A CUDA program executes in parallel across a set of threads, where a thread is the basic unit in the CUDA programming model that executes an instance of the code and has access to registers and per thread local memory. The GPU processors execute threads in groups of 32 called *warps*. Performance on GT200-class GPUs can be optimized a great deal by having threads in a half-warp (16 threads) execute the same code path and access memory in a close vicinity. In practice the threads in a half-warp will have to wait for each other to finish their workload instead of continuing on independently. To reduce this waiting time, the database sequences are sorted by length to minimize length differences between neighboring threads, as shown in Figure 2(b). Sequence descriptions are stored in a separate file that is not uploaded to the GPU, saving memory and decreasing load times. Furthermore, sequence characters are replaced with numeric indexes to facilitate easier substitution matrix lookups.

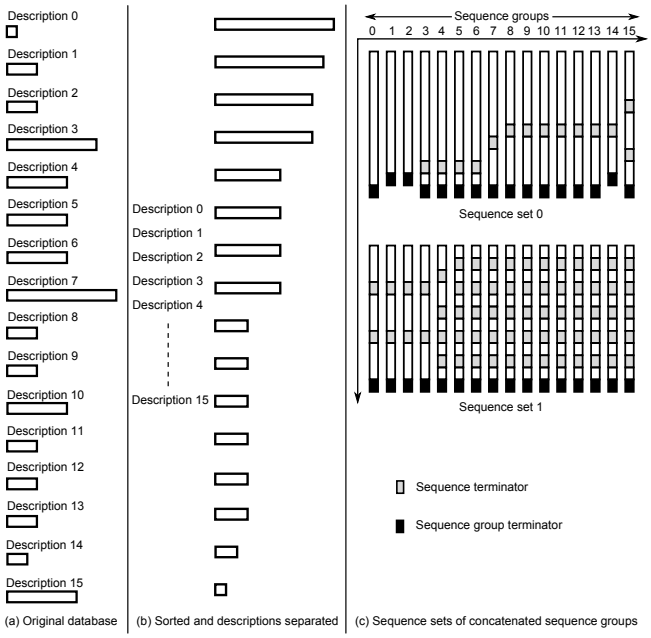

Fig. 2. The database conversion process

2) Concatenation: After sorting, groups of 16 sequences are taken and processed in *sequence sets* that will have a half-warp of threads working on them, as shown in Figure 2(c). Even though sorting by length has somewhat equalized workload within each sequence set, various sequence sets still have different sizes. To combat this, sequences within a sequence set are concatenated with leftover sequences to form *sequence groups*. The total length of each sequence group within a sequence set nearly equals or, ideally, matches the length of the longest sequence in that set. This results in an equal workload for each thread in a half-warp processing a sequence set.

Sequence terminators are inserted between the concatenated sequences; these tell the GPU kernel to initiate a new alignment. *Sequence group terminators* are inserted at the end of each sequence group signifying the end of a group of concatenated sequences, at which point a thread will wait for the rest of the threads in the half-warp to cease execution.

3) Interlacing: Once all database sequences have been processed into 16-wide sets of sequence groups, they are written to file. The sequence sets are written in an interlaced fashion, as shown in Figure 3. Each interlaced *subset* consists of eight characters from each sequence group.

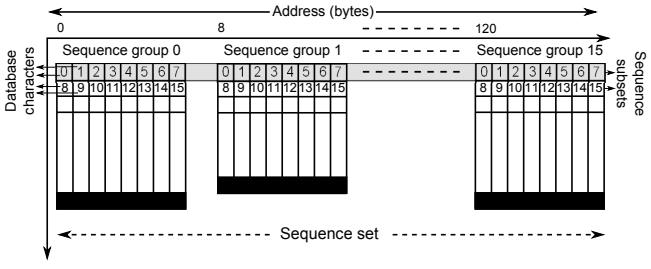

Fig. 3. Sequence storing as interlaced subsets

Eight characters of the set's first sequence group are written, then eight characters of the set's second group and so on. As there are 16 sequence groups in each sequence set, each thread in a half-warp is now able to load 8 bytes of sequence data from neighboring addresses. As a result, 128-byte coalesced loading takes place [20].

4) Equal length sequence sets: During code development, alignments were conducted with a synthetic (randomly generated) database, each sequence of which had the same length. The performance of this synthetic database is twice that of the Swiss-Prot database, which has sequences ranging in length from 2 to 35213 characters. The drop in performance for Swiss-Prot is the result of different workloads between different half-warps.

Though concatenation resulted in an equal workload distribution for threads within every sequence set, it still varies among different sequence sets. To resolve this, the length of each sequence group within every sequence set is made equal or nearly equal to the length of the longest sequence in the database, as shown in Figure 2(c). This results in an equal workload distribution for all GPU threads in general. The outcome of this is a 1.7 times increase in performance.

Evidently, equal workload across different threads improves performance; possibly a result of the GPU's thread scheduling not being optimal in the previous case. For example, the GPU thread scheduler might only schedule a new thread block once all the threads in a previous thread block have completed their execution.

C. Temporary data reads and writes

Memory bandwidth represented a serious bottleneck while developing the GPU implementation. A number of steps have been taken to optimize for high performance by reducing the number of memory accesses, the frequent temporary data accesses in particular. As no traceback is performed on the GPU, S-W matrix values do not need to be saved for the entire execution time and can be overwritten. As such, only a single column of S-W scores is kept. This score column stores values to the left of the currently processing column, i.e. $H_{i-1,1\leq j\leq N}$ in Equation 1. The size of this temporary data column is set to the size of the query sequence, not the database sequence, so that the column can have one fixed size for all database sequences. This usually requires less memory, as it is unlikely that the query sequence will be as long as the longest database sequence. The temporary data column is set to zero whenever a new database sequence is started. In addition to this temporary score column, variables are used to keep the values of the upper and upper-left cells required by the algorithm, i.e. $H_{i,j-1}$ and $H_{i-1,j-1}$ in Equation 1. To support affine gap penalties, another temporary data column is added for D values. Additionally, an upper E value is kept (see Equation 1).

Each S-W iteration involves reading and writing two temporary values (score and D), for four accesses in total. When both are non-coalesced, 32 byte reads/writes are issued for each access. This means that per half-warp 16 threads $\times$ 32 bytes $\times$ 2 values $\times$ 2 read/write $=$ 2048 bytes of bandwidth is used, resulting in a major memory bottleneck. The optimization steps mentioned below decrease this to one 128-byte coalesced read and write for every second iteration. This is a 16 times bandwidth improvement and requires only 1 instead of 64 accesses. 128 bytes is the largest allowed coalesced access size, and is faster than multiple smaller coalesced accesses. The optimizations are as follows:

- Smaller, 16-bit data type for the temporary values, cutting the theoretically required bandwidth in half and allowing for better coalescing.
- Each thread stores one data value in turn, resulting in an interlaced storage scheme. Instead of direct array accesses, a pointer into the temporary storage is started at the thread *id*, and increased by the total number of threads to move to the next element of the S-W H matrix. Each thread in a half-warp then reads a 2-byte coalesced value, meaning that instead of two 32-byte accesses per thread, two such accesses take place per half-warp. This sixteen times bandwidth improvement results in an almost ten times net speedup.
- To again halve the number of memory accesses, the temporary score and D values are interlaced. This is done by defining a data structure consisting of these values and using it to access the score and D values for an iteration in one go. At this point, a thread accesses two 2-byte values in one read, for a total of $16 \times 2 \times 2$ bytes bandwidth per half warp. The result is a 64-byte coalesced access.
- Finally, two temporary values are interlaced to move to 128-byte accesses. This has an additional benefit of temporary reads/writes only being required for every second query sequence symbol processed.

D. Substitution matrix accesses

Aligning proteins requires the use of a substitution matrix, which is accessed every time two symbols are aligned, making its access time critical to the implementation's performance. Substitution matrix (e.g. BLOSUM 62) accesses are random and are completely dependent on the database sequence, complicating the choice of memory used. GPU's Global memory is not a good choice for such a frequent usage due to its high access time. Also the random nature of substitution matrix accesses makes coalescing very difficult. As an alternative, the substitution matrix is stored in texture memory. Texture memory is a cached window into global memory that offers lower latency and does not require coalescing for best performance. It is thus well suited for random access. Texture memory has the ability to fetch four values at a time. This mechanism can be used to fetch four substitution matrix values from a *query profile*.

A query profile is shown in Figure 4. It is a type of substitution matrix where, instead of the protein alphabet, the query sequence is used along the top row. This means that for a given database character, the substitution matrix is not random anymore: multiple substitution scores can be loaded simultaneously when aligning the query with a database character. Furthermore, query sequence lookups are

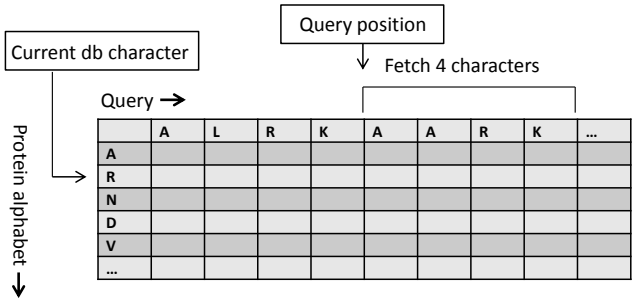

Fig. 4. Query profile

not required anymore; only the current position within the query is needed to index into the profile. A query profile is generated once for every query sequence. Each query profile column stores values for 23 characters. The number of columns and hence the memory requirement for a query profile depends on the length of the query sequence. The GTX 275 GPU used for our implementation has 8KB of texture cache per multiprocessor. This means that a query sequence having more than $\lfloor 8 \times 1024/23 \rfloor = 356$ characters will result in increased cache misses, as described in [14]. Tests were performed to quantify the texture cache miss rate, which was shown to be very low. For example, aligning an 8000 character query sequence resulted in 0.009% miss rate. For smaller sequences, the miss rate was reported to be even lower. Using this query profile method resulted in a 17% performance improvement with Swiss-Prot.

III. Discussion of results

The experimental setup used to test the implementation and measure its performance is as follows:

- Intel Core 2 Quad Q6600 (2.4 GHz) with 4GB of RAM
- NVIDIA Geforce GTX 275 graphics card with 896 MB of memory and clock speeds of 633, 1134 and 1404 MHz for its core, memory and shaders respectively
- 64 bit Microsoft Windows 7 Professional
- CUDA toolkit version 3.1
- Swiss-Prot release October 2010
- Substitution matrix BLOSUM62
- Gap penalty: -10 and gap extend penalty: -2 (these do not influence the execution time though)

The run time is measured using the C `clock()` instruction, the accuracy of which is verified using the CUDA profiling application. Table I displays the performance results, where the execution time in seconds and the performance in GCUPS are given for query sequences of varying lengths taken from Swiss-Prot and aligned against the same database.

Figure 5(a) shows that the execution time increases linearly with sequence length, resulting in an almost constant performance of around 21.4 GCUPS, shown in Figure 5(b).

The comparison of our optimized implementation with other solutions is shown in Figure 6 and is described in the following subsections.

A. Comparison with ssearch

Ssearch (SSE2) is an accelerated and multi-threaded version of ssearch, where ssearch is a CPU-based S-W alignment tool that can be found in the FASTA suite of applications [21]. The SSE2 optimizations, described in [22] utilize modern CPU's vector extensions for a performance increase. When run on the same system, using the same settings, our implementation performs 2.14 times better in terms of GCUPS than this accelerated and multi-threaded version of ssearch.

TABLE I
Performance results with Swiss-Prot

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

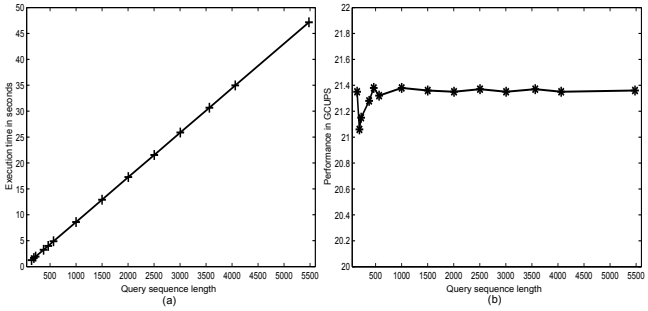

Fig. 5. (a) Execution time, (b) performance, for various query sequences

B. Comparison with a less optimized version

In the less optimized version, sequences are only sorted, concatenated and interlaced. However, no equal length sets were used, making the length of each sequence set depend on the longest sequence within that set. When run on the same experimental setup as the fully optimized version, this less optimized version results in a performance of around 12.5 GCUPS. The comparison in Figure 6 shows that our fully optimized GPU implementation performs around 1.7 times better than the less optimized version.

C. Comparison with CUDASW++ 2.0

When run on the same setup as our implementation, CUDASW++ 2.0 achieves a performance of around 19 GCUPS. Thus our fully optimized implementation performs 1.13 times better than CUDASW++ 2.0 in terms of GCUPS. Both approaches are sensitive to the structure of the database used. Like our implementation, CUDASW++ 2.0 also uses 16-bit score values, as discussed in Section II-C. Table II summarizes the optimization steps undertaken by our fully optimized implementation in comparison with CUDASW++ 2.0.

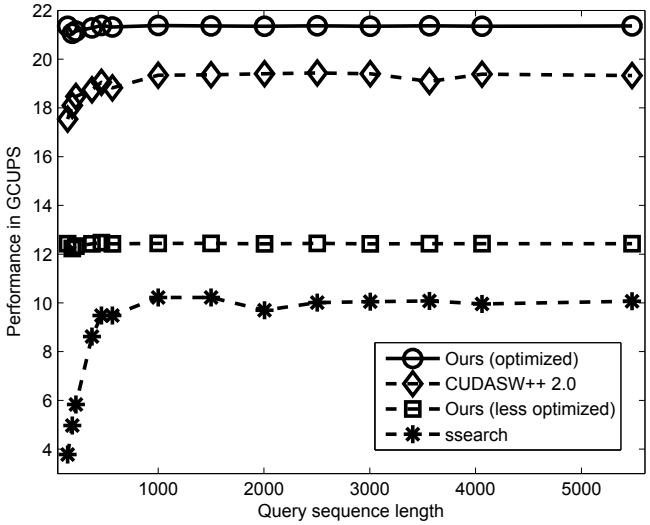

Fig. 6. Performance comparison

TABLE II
A COMPARISON WITH CUDASW++ 2.0

#	Optimization	Ours	CUDASW++ 2.0
1	Database sorting	+	+
2	Concatenation	+	−
3	Interlacing	+	+
4	Equal length sequence sets	+	−
5	Query profile	+	+

Additionally, our implementation also brings in the following improvements:

- In comparison with CUDASW++ 2.0, our implementation is simpler, as it uses just one search kernel instead of two, requiring no inter-processor communication.
- The optimized database organization scheme used in our implementation allows an equal workload for each thread block, while CUDASW++ 2.0 uses a hand-picked point at which it switches from one kernel to the other for its work distribution.
- Our implementation exports the top scoring sequences for full alignment with *ssearch*. CUDASW++ 2.0 does not provide this facility. Our implementation also provides a web interface that allows it to be used conveniently and remotely.

In comparison with CUDASW++ 2.0, our less optimized implementation performs 1.52 times slower in terms of GCUPS, as shown in Figure 6. This is because CUDASW++ 2.0 switches to its secondary systolic array based alignment stage for long sequences. Long sequences in a database inherently have the largest length differences, specifically true for Swiss-Prot. Thus, aligning them using systolic array based approach reduces the workload differences.

IV. CONCLUSION

This paper presented a GPU-accelerated Smith-Waterman based implementation for protein sequence alignment. The new implementation improves the performance by reducing the number of memory accesses and optimizing the database organization. The database is organized in equal length sequence sets resulting in an equal workload distribution for all the threads of each multiprocessor on the GPU. The performance achieved by our new implementation is 21.4 GCUPs. In comparison with the state-of-the-art implementation on an NVIDIA GTX 275 graphics card, our implementation reports a 1.13 times performance improvement in terms of GCUPS.

REFERENCES

[1] L. Hasan, Z. Al-Ars and S. Vassiliadis, "Hardware Acceleration of Sequence Alignment Algorithms - An Overview", *International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS'07)*, pages 96–101, Rabat, Morocco, September 2–5, 2007.

[2] S. F. Altschul et al., "A Basic Local Alignment Search Tool", *Journal of Molecular Biology*, vol. 215, pages 403–410, 1990.

[3] W. R. Pearson and D. J. Lipman, "Rapid and Sensitive Protein Similarity Searches", *Science*, vol. 227, pages 1435–1441, 1985.

[4] S. R. Eddy, "Profile Hidden Morkov Models", *Bioinformatics Review*, vol. 14(9), pages 755–763, July 1998.

[5] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences", *Journal of Molecular Biology*, vol. 147, pages 195–197, 1981.

[6] R. Giegerich, "A Systematic Approach to Dynamic Programming in Bioinformatics", *Bioinformatics*, vol. 16, pages 665–677, 2000.

[7] A. B. Buyukkur and W. Najjar, "Compiler Generated Systolic Arrays for Wavefront Algorithm Acceleration on FPGAs", *International Conference on Field Programmable Logic and Applications (FPL08)*, Heidelberg, Germany, September 2008.

[8] L. Hasan and Z. Al-Ars, "An Efficient and High Performance Linear Recursive Variable Expansion Implementation of the Smith-Waterman Algorithm", 31^{st} *Annual International Conference of the IEEE EMBS*, pages 3845–3848, Minneapolis, Minnesota, USA, September 2009.

[9] J. Lu, M. Perrone, K. Albayraktaroglu and M. Franklin, "HMMer-Cell: High Performance Protein Profile Searching on the Cell/B.E. Processor", *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2008)*, pages 223–232, Austin, Texas, USA, April 20–22, 2008.

[10] A. Schroder et al., "Bio-Sequence Database Scanning on a GPU" *HICOMB*, 2006.

[11] Y. Liu, W. Huang, J. Johnson and S. Vaidya, "GPU Accelerated Smith-Waterman", *International Conference on Computational Science, ICCS 2006*, University of Reading, UK, May 28–31 2006.

[12] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases", *2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*, Rome, Italy, May 23–29, 2009.

[13] S. A. Manavski and G. Valle, "CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment", *BMC Bioinformatics*, vol. 9, Suppl 2:S10, 2008.

[14] A. Akoglu and G. M. Striemer, "Scalable and Highly Parallel Implementation of Smith-Waterman on Graphics Processing Unit using CUDA", *Cluster Computing*, vol. 12(3), pages 341–352, 2009.

[15] Y. Liu, D. Maskell and B. Schmidt, "CUDASW++: Optimizing Smith-Waterman Sequence Database Searches for CUDA-enabled Graphics Processing Units", *BMC Research Notes*, vol. 2(1):73, 2009.

[16] Y. Liu, B. Schmidt and D. Maskell, "CUDASW++2.0: Enhanced Smith-Waterman Protein Database Search on CUDA-enabled GPUs based on SIMT and Virtualized SIMD Abstractions", *BMC Research Notes*, vol. 3(1):93, 2010.

[17] O. Gotoh, "An improved algorithm for matching biological sequences", *Journal of Molecular Biology*, vol. 162, pages 705–708, December 1982.

[18] Fermi™ "NVIDIA's Next Generation CUDA™ Compute Architecture", *White paper NVIDIA Corporation*, 2009.

[19] "http://www.uniprot.org", *Universal Protein Resource*, October 2010.

[20] M.A. Kentie, "Biological Sequence Alignment Using Graphics Processing Units", *M.Sc. Thesis CE-MS-2010-35*, Computer Engineering Laboratory, Technical University Delft, The Netherlands, 2010.

[21] "UVa Fasta Server", *http://fasta.bioch.virginia.edu*, February 2011.

[22] M. Farrar, "Striped Smith-Waterman Speeds Database Searches Six Times over other SIMD Implementations", *Bioinformatics*, vol. 23(2), pages 156–161, 2007.