9

An Overview of Hardware-Based Acceleration of Biological Sequence Alignment

Laiq Hasan and Zaid Al-Ars
TU Delft
The Netherlands

1. Introduction

Efficient biological sequence (proteins or DNA) alignment is an important and challenging task in bioinformatics. It is similar to string matching in the context of biological data and is used to infer the evolutionary relationship between a set of protein or DNA sequences. An accurate alignment can provide valuable information for experimentation on the newly found sequences. It is indispensable in basic research as well as in practical applications such as pharmaceutical development, drug discovery, disease prevention and criminal forensics. Many algorithms and methods, such as, dot plot (Gibbs & McIntyre, 1970), *Needleman-Wunsch (N-W)* (Needleman & Wunsch, 1970), *Smith-Waterman (S-W)* (Smith & Waterman, 1981), FASTA (Pearson & Lipman, 1985), BLAST (Altschul et al., 1990), HMMER (Eddy, 1998) and ClustalW (Thompson et al., 1994) have been proposed to perform and accelerate sequence alignment activities. An overview of these methods is given in (Hasan et al., 2007). Out of these, S-W algorithm is an optimal sequence alignment method, but its computational cost makes it inappropriate for practical purposes. To develop efficient and optimal sequence alignment solutions, the S-W algorithm has recently been implemented on emerging accelerator platforms such as *Field Programmable Gate Arrays (FPGAs)*, *Cell Broadband Engine (Cell/B.E.)* and *Graphics Processing Units (GPUs)* (Buyukkur & Najjar, 2008; Hasan et al., 2010; Liu et al., 2009; 2010; Lu et al., 2008). This chapter aims at providing a broad overview of sequence alignment in general with particular emphasis on the classification and discussion of available methods and their comparison. Further, it reviews in detail the acceleration approaches based on implementations on different platforms and provides a comparison considering different parameters. This chapter is organized as follows:
The remainder of this section gives a classification, discussion and comparison of the available methods and their hardware acceleration. Section 2 introduces the S-W algorithm which is the focus of discussion in the succeeding sections. Section 3 reviews CPU-based acceleration. Section 4 provides a review of FPGA-based acceleration. Section 5 overviews GPU-based acceleration. Section 6 presents a comparison of accelerations on different platforms, whereas Section 7 concludes the chapter.

1.1 Classification

Sequence alignment aims at identifying regions of similarity between two DNA or protein sequences (the query sequence and the subject or database sequence). Traditionally, the methods of pairwise sequence alignment are classified as either global or local, where pairwise means considering only two sequences at a time. Global methods attempt to match as many

characters as possible, from end to end, whereas local methods aim at identifying short stretches of similarity between two sequences. However, in some cases, it might also be needed to investigate the similarities between a group of sequences, hence multiple sequence alignment methods are introduced. Multiple sequence alignment is an extension of pairwise alignment to incorporate more than two sequences at a time. Such methods try to align all of the sequences in a given query set simultaneously. Figure 1 gives a classification of various available sequence alignment methods.

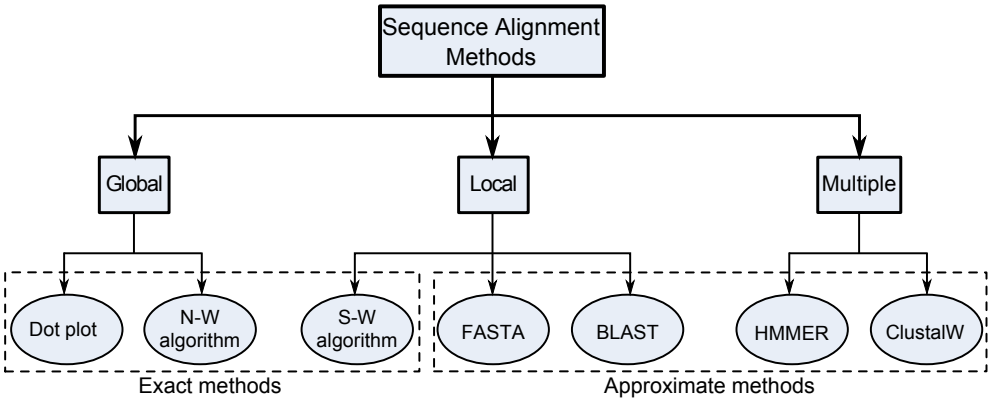

Fig. 1. Various methods for sequence alignment

These methods are categorized into three types, i.e. global, local and multiple, as shown in the figure. Further, the figure also identifies the exact methods and approximate methods. The methods shown in Figure 1 are discussed briefly in the following subsection.

1.2 Discussion of available methods

Following is a brief description of the available methods for sequence alignment.

Global methods

Global methods aim at matching as many characters as possible, from end to end between two sequences i.e. the *query sequence (Q)* and the *database sequence (D)*. Methods carrying out global alignment include dot plot and N-W algorithm. Both are categorized as exact methods. The difference is that dot plot is based on a basic search method, whereas N-W on *dynamic programming (DP)* (Giegerich, 2000).

Local methods

In contrast to global methods, local methods attempt to identify short stretches of similarity between two sequences i.e. Q and D. These include exact method like S-W and heuristics based approximate methods like FASTA and BLAST.

Multiple alignment methods

It might be of interest in some cases to consider the similarities between a group of sequences. Multiple sequence alignment methods like HMMER and ClustalW are introduced to handle such cases.

1.3 Comparison

The alignment methods can be compared on the basis of their temporal and spatial complexities and parameters like alignment type and search procedure. A summary of the comparison is shown in Table 1. It is interesting to note that all the global and local sequence alignment methods essentially have the same computational complexity of $O(L_Q L_D)$, where L_Q and L_D are the lengths of the query and database sequences, respectively. Yet despite this, each of the algorithms has very different running times, with BLAST being the fastest and dynamic programming algorithms being the slowest. In case of multiple sequence alignment methods, ClustalW has the worst time complexity of $O(L_Q^2 L_D^2)$, whereas HMMER has a time complexity of $O(L_Q L_D^2)$. The space complexities of all the alignment methods are also essentially identical, around $O(L_Q L_D)$ space, except BLAST, the space complexity of which is $O(20^w + L_Q L_D)$. In the exact methods, dot plot uses a basic search method, whereas N-W and S-W use DP. On the other hand, all the approximate methods are heuristic based. It is also worthy to note that FASTA and BLAST have to make sacrifices on sensitivity to be able to achieve higher speeds. Thus, a trade off exists between speed and sensitivity and we must come to a compromise to be able to efficiently align sequences in a biologically relevant manner in a reasonable amount of time.

Method	Type	Accuracy	Search	Time complexity	Space complexity
Dot plot	Global	Exact	Basic	$O(L_Q L_D)$	$O(L_Q L_D)$
N-W	Global	Exact	DP	$O(L_Q L_D)$	$O(L_Q L_D)$
S-W	Local	Exact	DP	$O(L_Q L_D)$	$O(L_Q L_D)$
FASTA	Local	Approximate	Heuristic	$O(L_Q L_D)$	$O(L_Q L_D)$
BLAST	Local	Approximate	Heuristic	$O(L_Q L_D)$	$O(20^w + L_Q L_D)$
HMMER	Multiple	Approximate	Heuristic	$O(L_Q L_D^2)$	$O(L_Q L_D)$
ClustalW	Multiple	Approximate	Heuristic	$O(L_Q^2 L_D^2)$	$O(L_Q L_D)$

Table 1. Comparison of various sequence alignment methods

1.4 Hardware platforms

Work has been done on accelerating sequence alignment methods, by implementing them on various available hardware platforms. Following is a brief discussion about such platforms.

CPUs

CPUs are well known, flexible and scalable architectures. By exploiting the *Streaming SIMD Extension (SSE)* instruction set on modern CPUs, the running time of the analyses is decreased significantly, thereby making analyses of data intensive problems like sequence alignment feasible. Also emerging CPU technologies like multi-core combines two or more independent processors into a single package. The *Single Instruction Multiple Data-stream (SIMD)* paradigm is heavily utilized in this class of processors, making it appropriate for data parallel applications like sequence alignment. SIMD describes CPUs with multiple processing elements that perform the same operation on multiple data simultaneously. Thus, such machines exploit data level parallelism. The SSE instruction set extension in modern CPUs contains 70 new SIMD instructions. This extension greatly increases the performance when exactly the same operations are to be performed on multiple data objects, making sequence alignment a typical application.

FPGAs

FPGAs are reconfigurable data processing devices on which an algorithm is directly mapped to basic processing logic elements. To take advantage of using an FPGA, one has to implement massively parallel algorithms on this reconfigurable device. They are thus well suited for certain classes of bioinformatics applications, such as sequence alignment. Methods like the ones based on systolic arrays are used to accelerate such applications.

GPUs

Initially stimulated by the need for real time graphics in video gaming, GPUs have evolved into powerful and flexible vector processors, ideal for accelerating a variety of data parallel applications. GPUs have in the last couple of years developed themselves from a fixed function graphics processing unit into a flexible platform that can be used for *high performance computing (HPC)*. Applications like bioinformatics sequence alignment can run very efficiently on these architectures.

2. Smith-Waterman algorithm

In 1981, Smith and Waterman described a method, commonly known as the *Smith-Waterman (S-W)* algorithm (Smith & Waterman, 1981), for finding common regions of local similarity. S-W method has been used as the basis for many subsequent algorithms, and is often quoted as a benchmark when comparing different alignment techniques. When obtaining the local S-W alignment, a matrix H is constructed using the following equation.

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

Where $S_{i,j}$ is the similarity score and d is the penalty for a mismatch. The algorithm can be implemented using the following pseudo code.

Initialization:

```
H(0,j) = 0
H(i,0) = 0
```

Matrix Fill:

```
for each i,j = 1 to M,N
    {
        H(i,j) = max(0,
                     H(i-1,j-1) + S(i,j),
                     H(i-1,j) - d,
                     H(i,j-1) - d)
    }
```

Traceback:

```
H(opt) = max(H(i,j))
traceback(H(opt))
```

The H matrix is constructed with one sequence lined up against the rows of a matrix, and another against the columns, with the first row and column initialized with a predefined value (usually zero) i.e. if the sequences are of length M and N respectively, then the matrix for the alignment algorithm will have $(M+1) \times (N+1)$ dimensions. The matrix fill stage scores each cell in the matrix. This score is based on whether the two intersecting elements of each sequence are a match, and also on the score of the cell's neighbors to the left, above, and diagonally upper left. Three separate scores are calculated based on all three neighbors, and the maximum of these three scores (or a zero if a negative value would result) is assigned to the cell. This is done for each cell in the matrix resulting in $O(MN)$ complexity for the matrix fill stage. Even though the computation for each cell usually only consists of additions, subtractions, and comparisons of integers, the algorithm would nevertheless perform very poorly if the lengths of the query sequences become large. The traceback step starts at the cell with the highest score in the matrix and ends at a cell when the similarity score drops below a certain predefined threshold. For doing this, the algorithm requires to find the maximum cell which is done by traversing the entire matrix, making the time complexity for the traceback $O(MN)$. It is also possible to keep track of the cell with the maximum score, during the matrix filling segment of the algorithm, although this will not change the overall complexity. Thus, the total time complexity of the S-W algorithm is $O(MN)$. The space complexity is also $O(MN)$.

In order to reduce the $O(MN)$ complexity of the matrix fill stage, multiple entries of the H matrix can be calculated in parallel. This is however complicated by data dependencies, whereby each $H_{i,j}$ entry depends on the values of three neighboring entries $H_{i,j-1}$, $H_{i-1,j}$ and $H_{i-1,j-1}$, with each of those entries in turn depending on the values of three neighboring entries, which effectively means that this dependency extends to every other entry in the region $H_{x,y} : x \leq i,\ y \leq j$. This implies that it is possible to simultaneously compute all the elements in each anti-diagonal, since they fall outside each other's data dependency regions. Figure 2 shows a sample H matrix for two sequences, with the bounding boxes indicating the elements that can be computed in parallel. The bottom-right cell is highlighted to show that its data dependency region is the entire remaining matrix. The dark diagonal arrow indicates the direction in which the computation progresses. At least 9 cycles are required for this computation, as there are 9 bounding boxes representing 9 anti-diagonals and a maximum of 5 cells may be computed in parallel.

The degree of parallelism is constrained to the number of elements in the anti-diagonal and the maximum number of elements that can be computed in parallel are equal to the number of elements in the longest anti-diagonal (l_d), where,

$$l_d = \min(M, N) \tag{2}$$

Theoretically, the lower bound to the number of steps required to calculate the entries of the H matrix in a parallel implementation of the S-W algorithm is equal to the number of anti-diagonals required to reach the bottom-right element, i.e. $M + N - 1$ (Liao et al., 2004).

Figure 3 shows the logic circuit to compute an element of the H matrix. The logic contains three adders, a sequence comparator circuit (*SeqCmp*) and three max operators *(MAX)*. The sequence comparator compares the corresponding characters of two input sequences and outputs a match/mismatch score, depending on whether the two characters are equal or not. Each max operator finds the maximum of its two inputs. The time to compute an element is 4 cycles, assuming that the time for each cycle is equal to the latency of one add or compare operation.

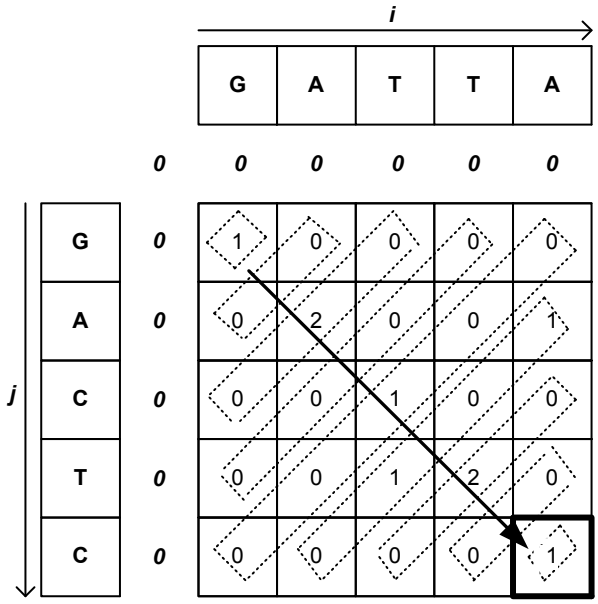

Fig. 2. Sample H matrix, where the dotted rectangles show the elements that can be computed in parallel

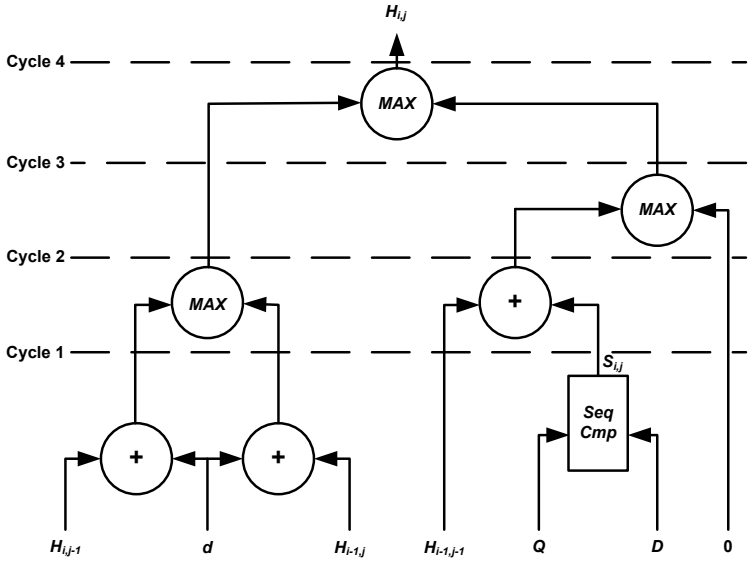

Fig. 3. Logic circuit to compute cells in the H matrix, where $+$ is an adder, *MAX* is a max operator and *SeqCmp* is the sequence comparator that generates match/mismatch scores

3. CPU-based acceleration

In this section CPU-based acceleration of the S-W algorithm is reviewed. Furthermore, an estimation of the performance for top-end and future systems is made.

3.1 Recent implementations

The first CPU implementations used a sequential way of calculating all the matrix values. These implementations were slow and therefore hardly used. In 2007, Farrar introduced a SSE implementation for S-W (Farrar, 2007). His work used SSE2 instructions for an Intel processor and was up to six times faster than existing S-W implementations. Two years later, a *Smith-Waterman implementation on Playstation 3 (SWPS3)* was introduced (Szalkowski et al., 2009), which was based on a minor adjustment to Farrar's implementation. SWPS3 is a vectorized implementation of the Smith-Waterman local alignment algorithm optimized for both the IBM Cell/B.E. and Intel x86 architectures. A SWPS3 version optimized for multi threading has been released recently (Aldinucci et al., 2010). The SSE implementations can be viewed as being semi parallel, as they constantly calculate sixteen, eight or less values at the same time, while discarding startup and finish time. Table 2 presents the performance achieved by these implementations on various CPU platforms.

Implementation	Peak performance	Benchmark hardware	Peak performance (per thread)
(Farrar, 2007)	2.9 GCUPS	2.0 GHz, Xeon Core 2 Duo single thread	3.75 GCUPS
(Szalkowski et al., 2009)	15.7 GCUPS	2.4 GHz Core 2 Quad Q6600, 4 threads	4.08 GCUPS
(Aldinucci et al., 2010)	35 GCUPS	2.5 GHz, 2x Xeon Core Quad E5420, 8 threads	4.38 GCUPS

Table 2. Performance achieved by various S-W CPU implementations (Vermij, 2011)

3.2 Performance estimations for top-end and future CPUs

With the data from Table 2, we make an estimate of the performance on the current top-end CPUs and take a look into the future. Table 3 gives the estimated peak performances based on the SIMD register width, the number of cores, clock speed and the known speed per core. We assumed linear scaling in the number of cores as suggested in Table 2, and the given performances may therefore not be reliable. Non-ideal inter-core communication, memory bandwidth limitations and shared caches could lead to a lower peak performance. Furthermore, no distinction in performance is made between Intel and AMD processors. Hence, Table 3 must be used as an indication to where the S-W performance could go on in current and future CPUs (Vermij, 2011).

4. FPGA-based acceleration

FPGAs are programmable logic devices. To map an application on flexible FPGA platforms, a program is written in a hardware description language like VHDL. The flexibility, difficulty

System	Released	SIMD register width	Cores (threads)	Clock speed	Peak performance (estimated)
Xeon Beckton	2010	128	8 (16)	2.26 GHz	32 GCUPS
Opteron Magny-Cours	2010	128	12 (12)	2.3 GHz	48 GCUPS
Opteron Interlagos	2011	128	16 (16)	2.3 GHz	64 GCUPS

Table 3. Estimated peak performance for current top-end and future CPUs (Vermij, 2011)

of design as well as the performance of FPGA implementations fall typically somewhere between pure software running on a CPU and an *Application Specific Integrated Circuit (ASIC)*. FPGAs are widely used to accelerate applications like S-W based sequence alignment. Implementations rely on the ability to create building blocks called *processing elements (PEs)* that can update one matrix cell every clock cycle. Furthermore, multiple PEs can be linked together in a two dimensional or linear systolic arrays to process huge data in parallel. This section provides a brief description of traditional systolic arrays followed by a discussion of existing and future FPGA-based S-W implementations.

4.1 Systolic arrays

Systolic array is an arrangement of processors in an array, where data flows synchronously across the array between neighbors, usually with data flowing in a specific direction (Kung & Leiserson, 1979), (Quinton & Robert, 1991). Each processor at each step takes in data from one or more neighbors (e.g. North and West), processes it and, in the next step, outputs results to the opposite neighbors (South and East). Systolic arrays can be implemented in rectangular or *2-dimensional (2D)* and linear or *1-dimensional (1D)* fashion. Figure 4 gives a pictorial view of both implementation types.

They best suit compute-intensive applications like biological sequence alignment. The disadvantage is that being highly specialized processors type, they are difficult to implement and build.

In (Pfeiffer et al., 2005), a concept to accelerate S-W algorithm on the basis of linear systolic array is demonstrated. The reason for choosing this architecture is outlined by demonstrating the efficiency and simplicity in combination with the algorithm. Nevertheless, there are two key methodologies to speedup this massively parallel system. By turning the processing from bit-parallel to bit-serial, the actual improvement is enabled. This change is performance neutral, but in combination with the early maximum detection, a considerable speedup is possible. Another effect of this improvement is a data dependant execution time of the processing elements. Here, the second acceleration prevents idle times to exploit the hardware and speeds up the computation. This can be accomplished by a globally asynchronous timing representing a self-timed linear systolic array. The authors have provided no performance estimation due to the initial stage of their work, that is why it cannot be compared with other related work.

In (Vermij, 2011), the working of a *linear systolic array (LSA)* is explained. Such an array works like the SSE unit in a modern CPU. But instead of having a fixed length of lets say 16, the FPGA based array can have any length.

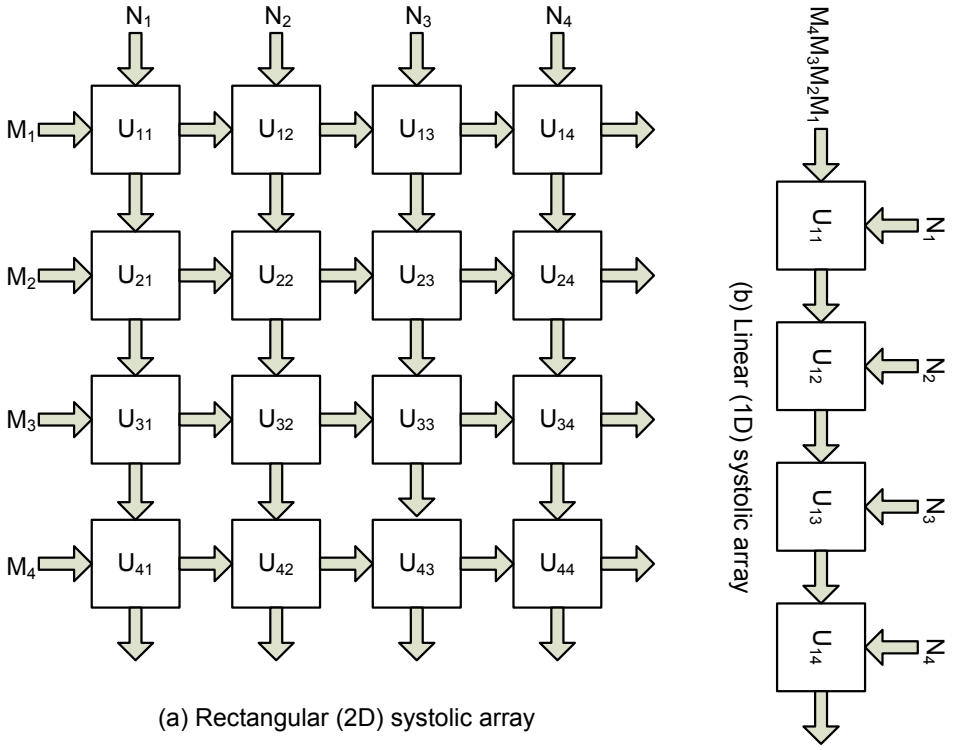

Fig. 4. Pictorial view of systolic array architectures

4.2 Existing FPGA implementations

In Section 3, we discussed some existing S-W implementations running on a CPU. A comparable analysis for FPGAs is rather hard. There are very few real, complete implementations that give usable results. Most research implementations only discuss synthetic tests, giving very optimistic numbers for implementations that are hardly used in practice. Furthermore, there is a great variety in the types of FPGAs used. Since every FPGA series has a different way of implementing circuitry, it is hard to make a fair comparison. In addition, the performance of the implementations relies heavily on the data widths used. Smaller data widths lead to smaller PEs, which lead to faster implementations. These numbers are not usually published. The first, third and fourth implementations shown in Table 4 make this clear, where the performance is given in terms of *Giga Cell Updates Per Second (GCUPS)*. Using the same FPGA device, these three implementations differ significantly in performance. The most reliable numbers are from Convey and SciEngines, as shown in the last two entries of Table 4. These implementations work the same in practice for real cases and are build for maximal performance (Vermij, 2011).

Reference	FPGA	Frequency	PEs	Performance (per FPGA)	Performance (per system)
(Puttegowda et al., 2003)	Virtex2 XC2V6000	180 MHz	7000	1260 GCUPS	—
(Yu et al., 2003)	Virtex2 XCV1000-6	—	4032	742 GCUPS	—
(Oliver et al., 2005)	Virtex2 XC2V6000	55 MHz	252	13.9 GCUPS	—
(Gok & Yilmaz, 2006)	Virtex2 XC2V6000	112 MHz	482	54 GCUPS	—
(Altera, 2007)	Stratix2 EP2S180	66.7 MHz	384	25.6 GCUPS	—
(Cray, 2010)	Virtex4	200 MHz	120	24.1 GCUPS	—
(Convey, 2010)	Virtex5 LX330	150 MHz	1152	172.8 GCUPS	691.2 GCUPS
(SciEngines, 2010)	Spartan6 LX150	—	—	47 GCUPS	6046 GCUPS

Table 4. Performance of various FPGA implementations (Vermij, 2011)

4.3 Future FPGA implementations

The performance of S-W implementations on FPGA can foremost be increased by using larger and faster FPGAs. Larger FPGAs can contain more PEs and therefore deliver higher performance in terms of GCUPS. The largest Xilinx Virtex 6 FPGA device has roughly 2.5 times more area than the largest Virtex 5 FPGA, so the peak performance of the former can be estimated at $2.5 \times 172.8 = 432$ GCUPS (using the numbers from the Convey implementation) (Vermij, 2011).

5. GPU-based acceleration

The parallelization capabilities of GPUs can be best exploited for accelerating biological sequence alignment applications. This section provides some brief background information about GPUs. Furthermore, it presents the current GPU implementations for S-W based sequence alignment.

5.1 GPU background

Compute Unified Device Architecture (CUDA) is the hardware and software architecture that enables NVIDIA GPUs (Fermi™, 2009) to execute programs written in C, C++, Fortran, OpenCL, DirectCompute and other languages. A CUDA program calls kernels that run on the GPU. A kernel executes in parallel across a set of threads, where a thread is the basic unit in the programming model that executes an instance of the kernel, and has access to registers and per thread local memory. The programmer organizes these threads in grids of thread blocks, where a thread block is a set of concurrently executing threads and has a shared memory for communication between the threads. A grid is an array of thread blocks that execute the same kernel, read inputs from and write outputs to global memory, and synchronize between interdependent kernel calls. Figure 5 gives a block diagram description of the GPU architecture.

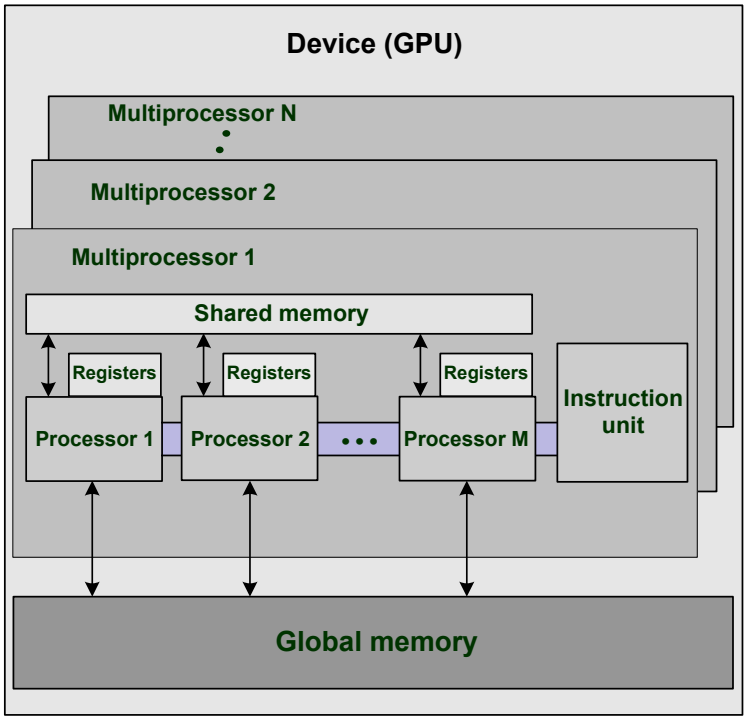

Fig. 5. Block diagram description of a GPU architecture

5.2 Current implementations

The first known implementations of S-W based sequence alignment on a GPU are presented in (Liu, Schmidt, Voss, Schroder & Muller-Wittig, 2006) and (Liu, Huang, Johnson & Vaidya, 2006). These approaches are similar and use the OpenGL graphics API to search protein databases. First the database and query sequences are copied to GPU texture memory. The score matrix is then processed in a systolic array fashion, where the data flows in anti-diagonals. The results of each anti-diagonal are again stored in texture memory, which are then used as inputs for the next pass. The implementation in (Liu, Schmidt, Voss, Schroder & Muller-Wittig, 2006) searched 99.8% of Swiss-Prot (almost 180,000 sequences) and managed to obtain a maximum speed of 650 MCUPS compared to around 75 for the compared CPU version. The implementation discussed in (Liu, Huang, Johnson & Vaidya, 2006) offers the ability to run in two modes, i.e. one with and one without traceback. The version with no traceback managed to perform at 241 MCUPS, compared to 178 with traceback and 120 for the compared CPU implementation. Both implementations were benchmarked using a Geforce GTX 7800 graphics card.

The first known CUDA implementation, 'SW-CUDA', is discussed in (Manavski & Valle, 2008). In this approach, each of the GPU's processors performs a complete alignment instead of them being used to stream through a single alignment. The advantage of this is that no communication between processing elements is required, thereby reducing memory reads and writes. This implementation managed to perform at 1.9 GCUPS on a single Geforce GTX 8800 graphics card when searching Swiss-Prot, compared to around 0.12 GCUPS for the compared

CPU implementation. Furthermore, it is shown to scale almost linearly with the amount of GPUs used by simply splitting up the database.
Various improvements have been suggested to the approach presented in (Manavski & Valle, 2008), as shown in (Akoglu & Striemer, 2009; Liu et al., 2009). In (Liu et al., 2009), for sequences of more than 3,072 amino acids an 'inter-task parallelization' method similar to the systolic array and OpenGL approaches is used as this, while slower, requires less memory. The 'CUDASW++' solution presented in (Liu et al., 2009) manages a maximum speed of about 9.5 GCUPS searching Swiss-Prot on a Geforce GTX 280 graphics card. An improved version, 'CUDASW++ 2.0' has been published recently (Liu et al., 2010). Being the fastest Smith-Waterman GPU implementation to date, 'CUDASW++ 2.0' managed 17 GCUPS on a single GTX 280 GPU, outperforming CPU-based BLAST in its benchmarks.
In (Kentie, 2010), an enhanced GPU implementation for protein sequence alignment using database and memory access optimizations is presented. Each processing element in this 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 protein sequences. Hence, it is possible to keep all processors well occupied while aligning query sequences with the sequences in the Swiss-Prot database. The results demonstrate that the implementation presented in (Kentie, 2010) achieves a performance of 21.4 GCUPS on an NVIDIA GTX 275 graphics card. Table 5 summarizes these GPU implementations. Besides NVIDIA, ATI/AMD (AMD, 2011) also produces graphics cards but to our knowledge no S-W implementations on such cards are available.

Implementation	Device	Database searched	Performance
(Liu, Schmidt, Voss, Schroder & Muller-Wittig, 2006)	GTX 7800	Swiss-Prot	650 MCUPS
(Liu, Huang, Johnson & Vaidya, 2006)	GTX 7800	983 protein sequences	241 MCUPS
(Manavski & Valle, 2008)	GTX 8800	Swiss-Prot	1.9 GCUPS
(Liu et al., 2009)	GTX 280	Swiss-Prot	9.5 GCUPS
(Liu et al., 2010)	GTX 280	Swiss-Prot	17 GCUPS
(Kentie, 2010)	GTX 275	Swiss-Prot	21.4 GCUPS

Table 5. Summary of the existing GPU implementations

6. Comparison of acceleration on different platforms

This section compares the performance of S-W implementations on various platforms like CPUs, FPGAs and GPUs. The comparison is based on parameters like cost, energy consumption, flexibility, scalability and future prospects. For the CPU, we consider a four way, 48 core Opteron machine. For GPUs, a fast PC with 4 high end graphics cards, and for FPGAs the fastest S-W system known, the one from SciEngines (SciEngines, 2010). The results are shown in Figure 6. Following is a discussion per metric (Vermij, 2011).

Performance/Euro

FPGAs can deliver the best amount of GCUPS per Euro, followed closely by GPUs. The gap between GPUs and CPUs can be explained by the extra money needed for a 4 way CPU system, while plugging 4 GPUs on a commodity motherboard is free. This result explains

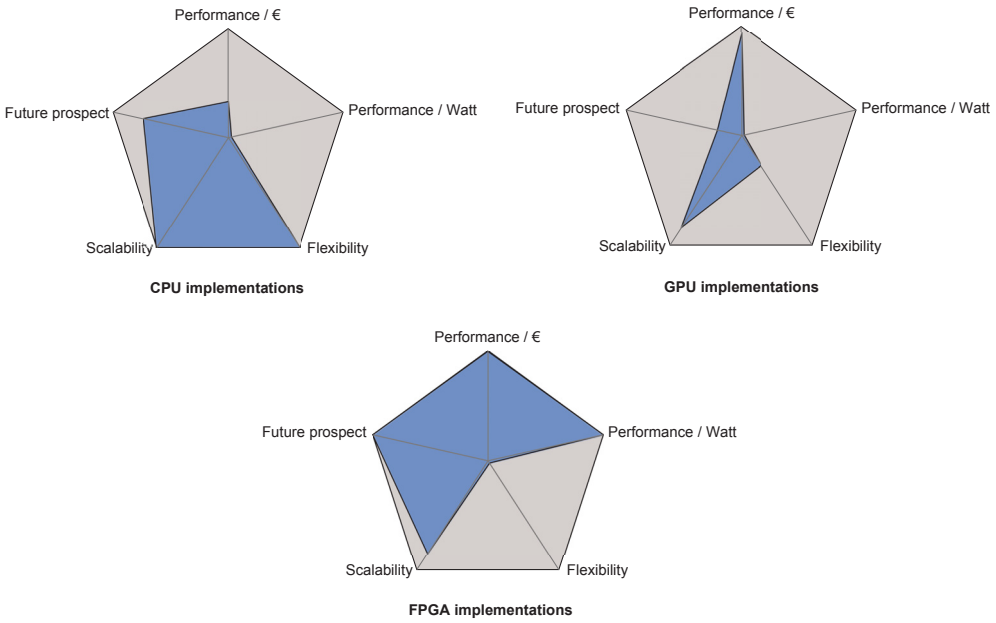

Fig. 6. Analysis of various S-W metrics for implementations on different platforms (Vermij, 2011)

why FPGAs are used for high performance computing. S-W might not be the algorithm of choice to show a major performance per Euro gain from using FPGAs. Nevertheless, it shows the trend.

Performance/Watt

It is clear that here, in contrast to the previous metric, FPGAs are the absolute winner. Full systems can deliver thousands of GCUPS for around 1000 Watts. This is another important reason for using FPGAs for sequence alignment. Note that, while not visible in the graphs, CPUs score around twice as good as GPUs.

Flexibility

This metric represents the effort needed to change a fast linear gap implementation to use affine gaps. A skilled engineer would manage to do this in a day for a CPU implementation, in a couple of days for GPU implementations, and many weeks for their FPGA counterpart.

Scalability

Here we took CPUs as baseline. Given a suitable problem, CPUs are very scalable as they can be connected together using standard networking solutions. By the same token, GPUs are also scalable, as they can take advantage of the scalability of CPUs, but will introduce some extra latency. Therefore they score a bit lower than CPUs. Depending on the used platform, FPGAs can also be made scalable.

Future prospect

In the past few years, there is a trend for CPUs to replace GPUs in high speed systems. This trend is expected to continue, thereby reducing the market share of GPUs in favour of CPUs. CPUs therefore score highly on this metric while GPUs score rather low. In the very specific HPC areas, however, where memory bandwidth requirements are low and the problem is very composable, FPGAs will likely continue to be the best choice. S-W partially lies in this category.

7. Conclusions

This chapter provided a classification, discussion and comparison of the available sequence alignment methods and their acceleration on various available hardware platforms. A detailed introduction about the S-W algorithm, its pseudo code, data dependencies in the *H* matrix and logic circuit to compute values of the cells in the *H* matrix are provided. A review of CPU-based acceleration of S-W algorithm was presented. Recent CPU implementations were discussed and compared. Further, performance estimations for top end current and future CPUs were provided. FPGA-based acceleration of S-W algorithm and a discussion about systolic arrays was given. Existing FPGA implementations were discussed and compared. Further, an insight into the future FPGA implementations was touched upon. GPU-based acceleration of S-W algorithm and GPU background were presented. Current GPU implementations were discussed and compared. Furthermore, this chapter presented a comparison of S-W accelerations on different hardware platforms. The comparison was based on the following parameters.

- Performance per euro
- Performance per unit watt
- Flexibility
- Scalability
- Future prospects

8. References

Akoglu, A. & Striemer, G. M. (2009). Scalable and highly parallel implementation of Smith-Waterman on graphics processing unit using CUDA, *Cluster Computing* Vol. 12(No. 3): 341–352.

Aldinucci, M., Meneghin, M. & Torquati, M. (2010). Efficient Smith-Waterman on multi-core with fastflow, *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, Pisa, Italy, pp. 195–199.

Altera (2007). Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform, *Altera White Paper*, Altera, pp. 1–18.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W. & Lipman, D. J. (1990). A basic local alignment search tool, *Journal of Molecular Biology* Vol. 215: 403–410.

AMD (2011). ATI/AMD.
URL: *http://www.amd.com/us/products/Pages/graphics.aspx*

Buyukkur, A. B. & Najjar, W. (2008). Compiler generated systolic arrays for wavefront algorithm acceleration on FPGAs, *Proceedings of International Conference on Field Programmable Logic and Applications (FPL08)*, Heidelberg, Germany, pp. 1–4.

Convey (2010). Convey HC1.
URL: *http://www.convey.com*

Cray (2010). Cray XD1.
URL: *http://www.cray.com*

Eddy, S. R. (1998). Profile hidden morkov models, *Bioinformatics Review* Vol. 14: 755-763.

Farrar, M. (2007). Striped Smith-Waterman speeds database searches six times over other SIMD implementations, *Bioinformatics* Vol. 23(2): 156–161.

Fermi™(2009). Nvidia's next generation cuda™ compute architecture, *White paper NVIDIA corporation* .

Gibbs, A. J. & McIntyre, G. A. (1970). The diagram, a method for comparing sequences, its use with amino acid and nucleotide sequences, *European Journal of Biochemistry* Vol. 16(No. 22): 1–11.

Giegerich, R. (2000). A systematic approach to dynamic programming in bioinformatics, *Bioinformatics* Vol. 16: 665–677.

Gok, M. & Yilmaz, C. (2006). Efficient cell designs for systolic Smith-Waterman implementation, *Proceedings of International Conference on Field Programmable Logic and Applications (FPL06)*, Madrid, Spain, pp. 1–4.

Hasan, L., Al-Ars, Z. & Taouil, M. (2010). High performance and resource efficient biological sequence alignment, *Proceedings of* 32^{nd} *Annual International Conference of the IEEE EMBS*, Buenos Aires, Argentina, pp. 1767–1770.

Hasan, L., Al-Ars, Z. & Vassiliadis, S. (2007). Hardware acceleration of sequence alignment algorithms - an overview, *Proceedings of International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS'07)*, Rabat, Morocco, pp. 96–101.

Kentie, M. (2010). Biological sequence alignment using graphics processing units, *M.Sc. Thesis CE-MS-2010-35*, Computer Engineering Laboratory, TU Delft, The Netherlands, 2010.

Kung, H. T. & Leiserson, C. E. (1979). Algorithms for VLSI processor arrays, *in: C. Mead, L. Conway (eds.):* Introduction to VLSI Systems; Addison-Wesley.

Liao, H. Y., Yin, M. L. & Cheng, Y. (2004). A parallel implementation of the Smith-Waterman algorithm for massive sequences searching, *Proceedings of* 26^{th} *Annual International Conference of the IEEE EMBS*, San Francisco, CA, USA, pp. 2817–2820.

Liu, W., Schmidt, B., Voss, G., Schroder, A. & Muller-Wittig, W. (2006). Bio-sequence database scanning on a GPU, *Parallel and Distributed Processing Symposium*, IEEE, Rhodes Island, pp. 1–8.

Liu, Y., Huang, W., Johnson, J. & Vaidya, S. (2006). GPU accelerated Smith-Waterman, *Proceedings of International Conference on Computational Science, ICCS 2006*, Springer, Reading, UK, pp. 1–8.

Liu, Y., Maskell, D. & Schmidt, B. (2009). CUDASW++: Optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units, *BMC Research Notes* Vol. 2(No. 1:73).

Liu, Y., Schmidt, B. & Maskell, D. (2010). 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(No. 1:93).

Lu, J., Perrone, M., Albayraktaroglu, K. & Franklin, M. (2008). HMMER-cell: High performance protein profile searching on the Cell/B.E. processor, *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2008)*, Austin, Texas, USA, pp. 223–232.

Manavski, S. A. & Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment, *BMC Bioinformatics* Vol. 9(No. 2:S10).

Needleman, S. & Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins, *Journal of Molecular Biology* Vol. 48(No. 3): 443–453.

Oliver, T., Schmidt, B. & Maskell, D. (2005). Hyper customized processors for bio-sequence database scanning on FPGAs, *Proceedings of FPGA'05*, ACM, Monterey, California, USA, pp. 229–237.

Pearson, W. R. & Lipman, D. J. (1985). Rapid and sensitive protein similarity searches, *Science* Vol. 227: 1435–1441.

Pfeiffer, G., Kreft, H. & Schimmler, M. (2005). Hardware enhanced biosequence alignment, *Proceedings of International Conference on METMBS*, pp. 1–7.

Puttegowda, K., Worek, W., Pappas, N., Dandapani, A. & Athanas, P. (2003). A run-time reconfigurable system for gene-sequence searching, *Proceedings of 16th International Conference on VLSI Design*, IEEE, USA, pp. 561–566.

Quinton, P. & Robert, Y. (1991). *Systolic Algorithms and Architectures*, Prentice Hall Int.

SciEngines (2010). Sciengines rivyera.
URL: *http://www.sciengines.com*

Smith, T. F. & Waterman, M. S. (1981). Identification of common molecular subsequences, *Journal of Molecular Biology* Vol. 147: 195–197.

Szalkowski, A., Ledergerber, C., Krhenbhl1, P. & Dessimoz, C. (2009). SWPS3 - A fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2, *BMC Research Notes* Vol. 1(No. 1:107).

Thompson, J. D., Higgins, D. G. & Gibson, T. J. (1994). ClustalW: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, *Nucleic Acids Research* Vol. 22(No. 22): 4673–4680.

Vermij, E. (2011). Genetic sequence alignment on a supercomputing platform, *M.Sc. Thesis*, Computer Engineering Laboratory, TU Delft, The Netherlands, 2011.

Yu, C. W., Kwong, K. H., Lee, K. H. & Leong, P. H. W. (2003). A Smith-Waterman systolic cell, *International Workshop on Field Programmable Logic and Applications (FPL03)*, Springer, pp. 375–384.