Cent. Eur. J. Comp. Sci. • 2(4) • 2012 • 367-388
DOI: 10.2478/s13537-012-0033-5

VERSITA

Central European Journal of **Computer Science**

Graphics card processing: accelerating profile-profile alignment

Research Article

Muhammad Kashif Hanif*, Karl-Heinz Zimmermann

Institute of Computer Technology,
Hamburg University of Technology,
Schwarzenbergstrasse 95e, 21073 Hamburg, Germany

Received 27 April 2012; accepted 29 October 2012

Abstract: Alignment is the fundamental operation in molecular biology for comparing biomolecular sequences. The most widely used method for aligning groups of alignments is based on the alignment of the profiles corresponding to the groups. We show that profile-profile alignment can be significantly speeded up by general purpose computing on a modern commodity graphics card. Wavefront and matrix-matrix product approaches for implementing profile-profile alignment onto graphics processor are analyzed. The average speed-up obtained is one order of magnitude even when overheads are considered. Thus the computational power of graphics cards can be exploited to develop improved solutions for multiple sequence alignment.

Keywords: alignment • progressive alignment • graphics processor card • basic linear algebra subprograms • performance

1. Introduction

Sequence alignment is the technique in molecular biology used to compare sequences and to arrange sequences of biomolecules for identifying regions of similarity that are eventually consequences of structural, functional, or evolutionary relationships [6, 10, 20, 22]. The problem of sequence alignment can be tackled by two computational approaches: optimal methods following the paradigm of dynamic programming and heuristic methods. The optimal methods include two basic algorithms. The Needleman-Wunsch algorithm provides a global alignment of two sequences aligning every residue in both sequences. It is most useful when the sequences are similar and of roughly equal length [15]. The Smith-Waterman algorithm yields a local alignment of two sequences in which only part of the residues participate. This method is more utilizable for dissimilar sequences that are suspected to contain regions of similarity [21].

Today, the size of biomolecular sequence databases grows exponentially due to the recent availability of high-throughput sequencing technologies [19]. This upsurge demands for fast alignment techniques rendering the more time-consuming optimal alignment techniques less useful for searching similarities in larger data sets. This is the reason why fast heuristic techniques such as BLAST [2] and FASTA [17] are preferred that are an order of magnitude faster than the

* *E-mail: muhammad.hanif@tuhh.de*

optimal algorithms. However, a downside of heuristic approaches is that they are less sensitive (i.e., missing more homologous) than the optimal ones.
The simultaneous alignment of several sequences results in an NP-complete combinatorial optimization problem [20, 22]. Therefore, a variety of heuristic methods have been developed for the alignment of three or more sequences. The most popular method to generate a multiple sequence alignment is based on trees which are used to describe a relationship between the sequences based on their pairwise comparison. In these progressive methods, the most similar sequences are aligned first and then less related sequences or groups of sequences are successively added to the alignment. Some progressive methods additionally assess the sequences according to their relatedness in order to improve alignment accuracy. The most widely used multiple sequence alignment programs are Clustal [5] and T-coffee [16].
Today, new desktop and notebook computers contain hardware for 3D graphics acceleration called Graphics Processing Units (GPUs). Modern GPUs include many independent floating-point arithmetic units for computing 3D models and other graphical tasks such as video-related functions. This makes GPUs amenable for general-purpose (GPGPU) computations that are traditionally treated by personal computers or workstations. GPUs have already been employed to general purpose computing in several areas such as molecular dynamics, physics simulations, and scientific computing[1]. Manavski et al. [13] and Munekawa et al. [14] have accelerated the Smith-Waterman algorithm on a GPU gaining moderate performance boosts. Methods to reduce the amount of data transfer and data fetches help to further increase the speed-up. Schatz et al. [18] have provided an implementation of a local sequence alignment algorithm (MUMmer) on a GPU attaining a ten-fold speed-up over a serial CPU version. Similarly, Dzivi [7] has directly implemented the Needleman-Wunsch algorithm and gained performance peaks of an eighty-fold speed-up. All these algorithms follow the wavefront approach utilizing the fact that the anti-diagonals in the corresponding forward table are independent of each other.
Recently, Bassoy et al. [3] have transformed the sequence-profile algorithm into matrix form as a vector-matrix and matrix-matrix product attaining maximum speed-up of 278.5 using NVidia BLAS3 when compared with a native Intel CPU implementation. This huge performance boost is due to the conversion of the alignment problem into a form that matches the vector-processing architecture of commodity GPUs.
In this paper, we provide GPGPU programs for performing profile-profile alignments. This method of alignment is part of progressive alignment allowing to combine two groups of alignments into a single alignment. For this, each alignment is represented by a statistical representative called profile. The basic operation used in profile-profile alignment are scalar products of fixed-length vectors that facilitate the generation of efficient GPGPU code; there are exceptions to this approach [8]. Our implementations run on recent hardware available from NVidia using a new software development kit (CUDA) for GPGPU programming. The performance of our implementations is assessed by comparing it with CPU based computations. The speed-ups achieved by Bassoy et al. [3] for profile-sequence alignment encouraged the implementation of profile-profile alignment algorithm. Although, computation times for profile-profile alignment on CPU might be tolerable since globular proteins are considered in this article. However, computation time on a CPU will be a major factor for genome based profiles which have much longer lengths.

2. Compute unified device architecture

The Compute Unified Device Architecture (CUDA) is the computing engine in NVidia GPUs that is accessible to software developers through standard programming languages like C [1]. CUDA treats the GPU as a compute device that is able to execute a high number of threads in a concurrent manner.
CUDA enables the programmer to write C-like functions called kernels. Each kernel is executed by a batch of threads that are organized as a grid of blocks. Each block consists of threads that execute in parallel. Threads in a block can efficiently communicate by using shared memory and in this way can synchronize their execution to coordinate memory access.
Each block can be organized as a one-, two-, or three-dimensional array of threads. The maximum number of threads per block is limited. All blocks of a kernel can be grouped into one- or two-dimensional arrays. With the invention of

[1] *http://gpgpu.org*

the Fermi architecture, blocks can have three-dimensions [1]. Each block is assigned to one streaming multiprocessor running until completion without pre-emption. Threads in different blocks within a kernel cannot communicate during execution. The blocks in a kernel are subject to a scheduler in order to assign them to the streaming multiprocessors. Each thread in a block executed by the device has registers that can only be accessed by the thread and shared memory that is local to the corresponding block. Moreover, data can be shared between blocks using global memory which is much slower than shared memory and can be accessed from both, the device and the host. The efficiency of GPU programs depends on the utilization of the allocated hardware, hardware configuration such as number of threads, blocks and memory space, and the amount of parallelism exhibited by the problem [1, 7].
CUDA also provides a BLAS (Basic Linear Algebra Subprograms) library called CUBLAS. It enables the programmer to use the GPU without direct operation of the CUDA drivers.
In particular, the GeForce GTX 560 Ti supports both graphics and general purpose applications [1]. The device runs at a clock rate of 1.8 GHz and provides a set of 8 streaming multiprocessors each of which organized as a SIMD (single instruction, multiple data) architecture. Data for a kernel application can be stored in global memory of 1,024 MB. Each streaming multiprocessor consists of 48 cuda cores. There are also two cached memories, texture memory and constant memory, which can only hold read-only data. These memories actually reside at the same level as the global memory. Each block contains 32,768 registers and can hold up to 1,024 threads. Shared memory of 48 KB is available for each block. The threads of a block are grouped into 32 threads known as warps. Each warp is executed by a streaming multiprocessor where a scheduler can switch from one warp to another to exploit the available resources.

3. Alignment

Alignment is the basic operation in molecular biology for comparing sequences. It provides a means to arrange biomolecular sequences in order to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relatedness.
Aligned sequences are usually represented as rows of a matrix. Gaps are inserted between the residues such that identical or related residues can be aligned in corresponding columns (Fig. 1). Sequences can be represented in several text-based formats like the FASTA format that can be read by most web-based or GUI tools supporting alignment routines. Pairwise sequence alignment can be tackled by dynamic programming in $O(n^2)$ steps if both sequences have length $O(n)$.

```
MNSFSTSAFGPVAFSLGLLLVLPAAFP-APVPPGEDSKDVAAPHRQPLTS
|...|...|.|||| |||:||...||| :.|..|:.::| ..|:| |:.:
MKFLSARDFHPVAF-LGLMLVTTTAFPTSQVRRGDFTED-TTPNR-PVYT

SERIDKQIRYILDGISALRKETCNKSNMCESSKEALAENNLNLPKMAEKD
:.::...|.::|..|..:|||.||.::.|.::.:|||||||.||::...|
TSQVGGLITHVLWEIVEMRKELCNGNSDCMNNDDALAENNLKLPEIQRND

GCFQSGFNEETCLVKIITGLLEFEVYLEYLQNRF-ESSEEQARAVQMSTK
||:|:|:|:|.||:||.:||||:..||||::|.. ::.:::||.:|..|:
GCYQTGYNQEICLLKISSGLLEYHSYLEYMKNNLKDNKKDKARVLQRDTE

VLIQFLQKKAKNLDAITTPDPTTNASLLTKLQAQNQWLQDMTTHLILRSF
.||....::.|:|..|..|.|.:||.|..||::|.:||:..|...||:|.
TLIHIFNQEVKDLHKIVLPTPISNALLTDKLESQKEWLRTKTIQFILKSL

KEFLQSSLRALRQM
:|||:.:||:.||.
EEFLKVTLRSTRQT
```

Figure 1. Pairwise sequence alignment between Interleukin-6 proteins from human and mouse calculated by the EMBOSS pairwise alignment algorithm from EMBL-EBI taking the substitution matrix Blosum62, gap penalty 10.0, and gap extension penalty 5.0; the alignment has length 214 with 41.6% matches, 56.1% mismatches, and 2.3% indels.

Multiple sequence alignment corresponds to the simultaneous alignment of three or more sequences (Fig. 2). Such alignments are employed to establish evolutionary relationships that are useful for constructing phylogenies and revealing conserved and variable sites within protein families. However, multiple sequence alignment is very time consuming. In particular, the dynamic programming algorithm for simultaneously aligning k sequences of length $O(n)$ necessitates $O(2^k n^k)$ steps and thus is only feasible for a handful of sequences. Therefore, practical multiple sequence alignment techniques are based on heuristics.

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

Figure 2. Two multiple sequence alignments of DNA sequences. The first alignment is formed by AGCGG, ACGTCG, and TGCTGC, and the second by GAAA, GTCTA, and CTCAT.

Progressive alignment is the most widely accepted heuristic method for aligning multiple sequences. It works in three steps. First, the optimal alignments between each pair of sequences are computed. Second, the so-called guide tree is built that reflects similarities (or distances) among the sequences. In particular, ClustalW [5] uses fractional identities in the optimal local alignment to calculate distances between sequences, and MSAProbs [12] takes similarity scores to calculate the distances. Third, the sequences are combined into a multiple alignment by using the tree as a guide. For this, intermediate alignments are formed from the leaves to the root such that two neighboring sequences are aligned in pairs, a sequence and a neighboring alignment are combined by profile-sequence alignment, and two neighboring alignments are aggregated by profile-profile alignment.
Algorithms for progressive alignment should be able to cope with a larger number of sequences in practical time scales. Two typical implementations are ClustalW [5] and T-coffee [16]. These tools make use of several ad-hoc rules for weightening scores. T-coffee tends to be slower than ClustalW but eventually produces more accurate alignments for distantly related sequences.
Progressive alignment describes alignments by profiles. A profile is a statistical representative of an alignment and can be pictured by an $l \times m$ matrix $\boldsymbol{P} = (p_{ij})$, where l is the size of the extended alphabet $\Sigma' = \Sigma \cup \{-\}$, m is the length of the alignment, and the entry p_{ij} gives the relative frequency of the symbol (residue) j to occur in the i-th column of the alignment (Fig. 3). DNA and RNA alphabets consist of four respective nucleotides, and the amino acid alphabet has 20 (naturally occurring) amino acids [9].

$$\begin{pmatrix} 0.66 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.33 & 0.66 & 0.00 & 0.00 & 0.33 & 0.33 \\ 0.00 & 0.66 & 0.33 & 0.33 & 0.00 & 0.66 & 0.33 \\ 0.33 & 0.00 & 0.00 & 0.00 & 0.66 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.66 & 0.33 & 0.00 & 0.33 \end{pmatrix}$$

Figure 3. Profile of the first multiple sequence alignment given in Figure 2; the rows are labelled in turn by symbols A, C, G, T, and –.

The dynamic programming algorithm for profile-profile alignment is specified by the routine ProfProfAlign. Its input is given by two profiles, an $l \times m$ matrix $\boldsymbol{P} = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_m)$ and an $l \times n$ matrix $\boldsymbol{Q} = (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n)$. In particular, a column consisting solely of blanks is associated with the profile column $-_p = (0, \ldots, 0, 1)^T$, where blank occurs with relative frequency 1. An alignment between the profiles $\boldsymbol{P}$ and $\boldsymbol{Q}$ is a pair of sequences

$$\boldsymbol{P}' = \boldsymbol{p}'_1 \ldots \boldsymbol{p}'_k \quad (1)$$

$$\boldsymbol{Q}' = \boldsymbol{q}'_1 \ldots \boldsymbol{q}'_k \quad (2)$$

Both sequences are of equal length and are derived from the corresponding profiles by inserting blank columns (Figs. 4 and 5).

$$\begin{array}{ccccccccc} \boldsymbol{p}_1 & \boldsymbol{p}_2 & \boldsymbol{p}_3 & \boldsymbol{p}_4 & \boldsymbol{p}_5 & \boldsymbol{p}_6 & \boldsymbol{p}_7 & -_p & -_p \\ -_p & \boldsymbol{q}_1 & -_p & \boldsymbol{q}_2 & \boldsymbol{q}_3 & \boldsymbol{q}_4 & -_p & \boldsymbol{q}_5 & \boldsymbol{q}_6 \end{array}$$

Figure 4. A profile-profile alignment between the profiles describing the multiple alignments in Fig. 2.

$$\begin{array}{ccccccccc} \mathrm{A} & \mathrm{G} & \mathrm{C} & \mathrm{G} & - & \mathrm{G} & - & - & - \\ \mathrm{A} & \mathrm{C} & \mathrm{G} & - & \mathrm{T} & \mathrm{C} & \mathrm{G} & - & - \\ \mathrm{T} & \mathrm{G} & \mathrm{C} & - & \mathrm{T} & \mathrm{G} & \mathrm{C} & - & - \\ - & \mathrm{G} & - & \mathrm{A} & \mathrm{A} & - & - & \mathrm{A} & - \\ - & \mathrm{G} & - & \mathrm{T} & - & \mathrm{C} & - & \mathrm{T} & \mathrm{A} \\ - & \mathrm{C} & - & \mathrm{T} & - & \mathrm{C} & - & \mathrm{A} & \mathrm{T} \end{array}$$

Figure 5. The overall multiple alignment resulting from the profile-profile alignment in Fig. 4.

The score of a profile-profile alignment $(\boldsymbol{P}', \boldsymbol{Q}')$ is defined as the sum of so-called column scores

$$d(\boldsymbol{P}', \boldsymbol{Q}') = \sum_{i=1}^{k} d(\boldsymbol{p}'_i, \boldsymbol{q}'_i), \tag{3}$$

where each column is scored by the Euclidean distance

$$d(\boldsymbol{p}'_i, \boldsymbol{q}'_i) = \sqrt{\sum_{j=1}^{k} (p'_{ij} - q'_{ij})^2}. \tag{4}$$

Note that the squared Euclidean distance amounts to a scalar product,

$$d(\boldsymbol{p}'_i, \boldsymbol{q}'_i)^2 = (\boldsymbol{p}'_i - \boldsymbol{q}'_i)^T \cdot (\boldsymbol{p}'_i - \boldsymbol{q}'_i). \tag{5}$$

Observe that the scoring model assumes that the aligned columns are statistically independent of each other. The problem of profile-profile alignment is to find an alignment with minimum score [10, 22].

Algorithm 1 ProfProfAlign$(\boldsymbol{P}, \boldsymbol{Q})$

Require: two profiles $\boldsymbol{P} = \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ and $\boldsymbol{Q} = \boldsymbol{q}_1, \ldots, \boldsymbol{q}_n$
Ensure: $\boldsymbol{S} = (S_{ij})$ forward matrix
1: $S_{0,0} \leftarrow 0$ {initialization}
2: **for** $i \leftarrow 1$ to m **do**
3: $\quad S_{i,0} \leftarrow \sum_{k=1}^{i} d(\boldsymbol{p}_k, -_p)$
4: **end for**
5: **for** $j \leftarrow 1$ to n **do**
6: $\quad S_{0,j} \leftarrow \sum_{k=1}^{j} d(-_p, \boldsymbol{q}_k)$
7: **end for**
8: **for** $i \leftarrow 1$ to m **do** {computation and minimization}
9: $\quad$ **for** $j \leftarrow 1$ to n **do**
10: $\quad\quad S_{i,j} \leftarrow \min\{S_{i-1,j} + d(\boldsymbol{p}_i, -_p), S_{i,j-1} + d(-_p, \boldsymbol{q}_j), S_{i-1,j-1} + d(\boldsymbol{p}_i, \boldsymbol{q}_j)\}$
11: $\quad$ **end for**
12: **end for**
13: **return** $\boldsymbol{S}$

The routine ProfProfAlign evaluates an $m \times n$ table $\boldsymbol{S} = (S_{ij})$, which is called the forward algorithm. The backward algorithm retrieves the optimal alignments from the table. This is achieved by tracing back through the table from the last entry $S_{m,n}$ to the first entry $S_{0,0}$, the optimal decisions made at each step. The paths from the last entry $S_{m,n}$ to the first entry $S_{0,0}$ established in this way correspond one-to-one with the optimal alignments. We will focus on the parallelization of the algorithm calculating the forward table, since the trace back has very low inherent parallelism.

4. Acceleration of profile-profile alignment

The algorithm ProfProfAlign shows that the entries S_{ij} in the interior of the forward table can only be computed if the neighboring entries $S_{i-1,j}$, $S_{i,j-1}$, and $S_{i-1,j-1}$ are already known. In the following, two approaches will be presented that fit to the parallel architecture of the GPU.

4.1. Matrix approach

The algorithm ProfProfAlign will be redesigned and implemented onto a GPU by separating the data independent and data dependent parts. First, the data independent part is implemented on GPU by calculating Euclidean distances $d(\boldsymbol{p}_i, -_p)$, $d(-_p, \boldsymbol{q}_j)$, and $d(\boldsymbol{p}_i, \boldsymbol{q}_j)$ making use of scalar products. These values are pre-stored in three $m \times n$ matrices $\boldsymbol{V} = (V_{ij})$, $\boldsymbol{H} = (H_{ij})$, and $\boldsymbol{D} = (D_{ij})$. Second, the data dependent part uses these matrices to compute the entries of the forward table. This part (lines 15 to 19) requires to take the minimum of three values and is implemented onto CPU. This gives the algorithm ProfProfAlignScalProd.

Algorithm 2 ProfProfAlignScalProd($\boldsymbol{P}$, $\boldsymbol{Q}$)

Require: two profiles $\boldsymbol{P} = \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ and $\boldsymbol{Q} = \boldsymbol{q}_1, \ldots, \boldsymbol{q}_n$
Ensure: $\boldsymbol{S} = (S_{ij})$ forward matrix
1: $S_{0,0} \leftarrow 0$ {initialization}
2: **for** $i \leftarrow 1$ to m **do**
3: $\quad S_{i,0} \leftarrow \sum_{k=1}^{i} d(\boldsymbol{p}_k, -_p)$
4: **end for**
5: **for** $j \leftarrow 1$ to n **do**
6: $\quad S_{0,j} \leftarrow \sum_{k=1}^{j} d(-_p, \boldsymbol{q}_k)$
7: **end for**
8: **for** $i \leftarrow 1$ to m **do** {calculation}
9: $\quad$ **for** $j \leftarrow 1$ to n **do**
10: $\quad\quad V_{i,j} \leftarrow d(\boldsymbol{p}_i, -_p)$
11: $\quad\quad H_{i,j} \leftarrow d(-_p, \boldsymbol{q}_j)$
12: $\quad\quad D_{i,j} \leftarrow d(\boldsymbol{p}_i, \boldsymbol{q}_j)$
13: $\quad$ **end for**
14: **end for**
15: **for** $i \leftarrow 1$ to m **do** {minimation}
16: $\quad$ **for** $j \leftarrow 1$ to n **do**
17: $\quad\quad S_{i,j} \leftarrow \min\{S_{i-1,j} + V_{ij}, S_{i,j-1} + H_{ij}, S_{i-1,j-1} + D_{ij}\}$
18: $\quad$ **end for**
19: **end for**
20: **return** $\boldsymbol{S}$

The algorithm ProfProfAlignScalProd can be reformulated by using matrix multiplications. For this, define the $l \times m$ matrix

$$\boldsymbol{P}_- = \left(-_p, \ldots, -_p\right) \tag{6}$$

and the $l \times n$ matrix

$$\boldsymbol{Q}_- = \left(-_p, \ldots, -_p\right), \tag{7}$$

and form the $l \times n$ matrix

$$\begin{aligned} \boldsymbol{H}_1 &= \boldsymbol{Q}_- - \boldsymbol{Q} \\ &= \left(-_p - \boldsymbol{q}_1, \ldots, -_p - \boldsymbol{q}_n\right), \end{aligned} \tag{8}$$

the $l \times m$ matrix

$$\begin{aligned} \boldsymbol{V}_1 &= \boldsymbol{P} - \boldsymbol{P}_- \\ &= \left(\boldsymbol{p}_1 - -_p, \ldots, \boldsymbol{p}_m - -_p\right), \end{aligned} \tag{9}$$

and the $l \times (m \times n)$ matrix

$$D_1 = (p_1 - q_1, \ldots, p_1 - q_n, \ldots, p_m - q_1, \ldots, p_m - q_n). \tag{10}$$

Then the squared Euclidean distances of the entries in the matrix H can be calculated by multiplying the newly formed matrix H_1 with its transpose and extracting the diagonal entries. This similarly holds for the matrices V and D.

$$H = \operatorname{diag}[H_1^T \cdot H_1] = \left(d(-_p - q_j)^2\right)_j, \tag{11}$$

$$V = \operatorname{diag}[V_1^T \cdot V_1] = \left(d(p_i - -_p)^2\right)_i, \tag{12}$$

$$D = \operatorname{diag}[D_1^T \cdot D_1] = \left(d(p_i - q_j)^2\right)_{i,j}. \tag{13}$$

These identities proved in the appendix give rise to algorithm ProfProfAlignMatProd.

Algorithm 3 ProfProfAlignMatProd(P, Q)

Require: two profiles $P = p_1, \ldots, p_m$ and $Q = q_1, \ldots, q_n$
Ensure: $S = (S_{ij})$ forward matrix
1: $S_{0,0} \leftarrow 0$ {initialization}
2: **for** $i \leftarrow 1$ to m **do**
3: $\quad S_{i,0} \leftarrow \sum_{k=1}^{i} d(p_k, -_p)$
4: **end for**
5: **for** $j \leftarrow 1$ to n **do**
6: $\quad S_{0,j} \leftarrow \sum_{k=1}^{j} d(-_p, q_k)$
7: **end for**
8: $H_1 \leftarrow Q_- - Q$ {calculation}
9: $V_1 \leftarrow P - P_-$
10: $D_1 \leftarrow \left(p_i - q_j\right)_{ij}$
11: $H \leftarrow \operatorname{diag}[H_1^T \cdot H_1]$
12: $V \leftarrow \operatorname{diag}[V_1^T \cdot V_1]$
13: $D \leftarrow \operatorname{diag}[D_1^T \cdot D_1]$
14: **for** $i \leftarrow 1$ to m **do** {minimation}
15: $\quad$ **for** $j \leftarrow 1$ to n **do**
16: $\quad\quad S_{i,j} \leftarrow \min\{S_{i-1,j} + V_i S_{i,j-1} + H_j, S_{i-1,j-1} + D_{ij}\}$
17: $\quad$ **end for**
18: **end for**
19: **return** S

4.2. Wavefront approach

The entries of the forward table depend on one or three previous entries (Fig. 6). The cells can be filled column by column, row by row, or anti-diagonal by anti-diagonal. The first two approaches limit the number of cells that can be simultaneously calculated, since entries in one column depend on other entries in the same or previous columns. The situation is similar for rows.

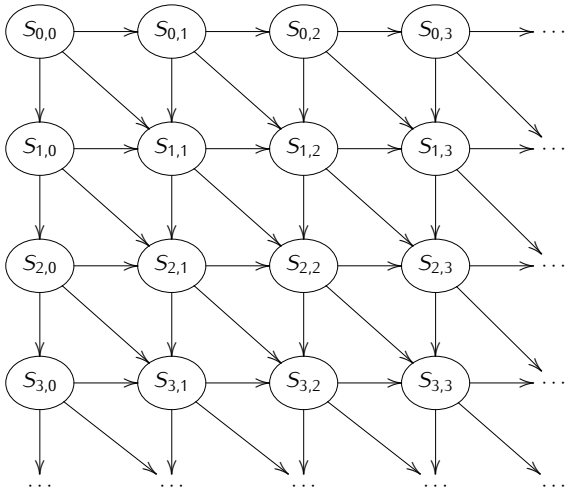

Figure 6. Data dependencies in the forward table.

However, an anti-diagonal consists of all cells S_{ij} such that $i + j$ is constant. Therefore, the elements on the same anti-diagonal are independent of each other and only depend on the previous two anti-diagonals. The approach of calculating all entries in each anti-diagonal at once is called wavefront method [11]. Most of the GPU implementations of sequence alignment follow this paradigm [4, 7, 13, 14, 18].

In the basic wavefront approach, the forward table is divided into rectangular blocks of the same size (Fig. 7). Each block inherits the data dependencies from the forward table. Moreover, there are data dependencies between the blocks (Fig. 8). For instance, at the beginning all entries of the block $B_{1,1}$ can be computed. Then the cells of the blocks $B_{1,2}$ and $B_{2,1}$ can be filled depending on the availability of the boundary entries of block $B_{1,1}$. Moreover, block $B_{2,2}$ needs both the boundary entries of $B_{1,2}$ and $B_{2,1}$ as well as the last entry of $B_{1,1}$. Thus the complete forward matrix exhibits parallelism at two levels: intra-block and inter-block. Both forms show a similar anti-diagonal pattern of parallelism.

Suppose each block has r rows and c columns. Then by the wavefront approach, there are $r + c - 1$ anti-diagonals such that the block can be computed in $r + c - 1$ parallel steps. Each block takes $r + c - 1$ boundary cells to compute $r \cdot c$ cell entries. Thus the communication-to-computation ratio becomes $(r + c - 1)/(r \cdot c)$. This ratio decreases when the block size increases.

Figure 7. Portion of a decomposition of the forward table into blocks of size 2×2.

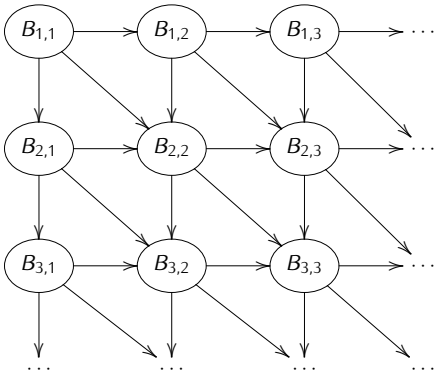

Figure 8. Block structure of the forward table.

A typical implementation of a wavefront algorithm on a GPU launches two kernels, one for initialization and one for filling the forward table [7]. The kernel for the initialization calculates the values of the boundary cells, while the kernel for the computation of the forward table provides a grid of blocks such that the blocks (of threads) correspond one-to-one with the blocks in the decomposition of the forward table. Moreover, the threads in a block are associated one-to-one with the cells of the corresponding block in the forward table each of which calculating the value of the cell. The kernel naturally emulates the anti-diagonal parallelism inside each block. For this, the threads in each block need to synchronize due to the dependencies among the anti-diagonals using the GPU function syncthreads.
However, CUDA does not provide global barrier synchronization between blocks and blocks exhibit producer-consumer relationship (due to data dependencies). Therefore, the anti-diagonal parallelism among blocks needs to be implemented on the host CPU.

5. Implementation and results

The above profile-profile alignment algorithms have been implemented on an Intel Core 2 Duo 6600 CPU (2.40 GHz) running openSUSE 11.4 linux distribution using the Intel Math Kernel Library (MKL) 10.3 and CUDA version 4.0 on an NVidia GeForce GTX 560 Ti graphics card. The tests are conducted using a serial gcc compiler (version 4.4.1) and an NVidia nvcc compiler with optimization flags *use_fast_math* and *maxrregcount*. The performance has been measured by execution time, floating point operation per second (flops), cell updates per second (CUPS) and speed-up. The number of floating point operations remains almost same across all version of profile-profile alignment algorithm.
The profiles have been generated at random for various lengths ranging from 32 to 992 with a step size of 32. The reasons are that CUDA has a fixed warp size of 32 threads. Additionally, profiles of higher length up to 10,000 are considered to analyze the behaviour of the different versions of the profile-profile alignment. Profiles of length longer than 10,000 are not taken into account due to hardware limitations since then the tables become so huge that they may occupy the whole GPU memory. Moreover, only profiles of comparable length are considered. Execution times have been averaged over ten runs for each profile length. Four basic implementations of the alignment algorithm ProfProfAlignScalProd have been evaluated; in each case, the forward table is decomposed into blocks of size $k \times k$:

- Simple k: Each thread calculates one cell of the forward table (k^2 threads per block).
- Row k: Each thread computes one row of a block (k threads per block).
- Column k: Each thread yields one column of a block (k threads per block).
- Mix k: Each thread evaluates one row and one column of a block (k threads per block).

The intermediate matrices $\mathbf{H}$, $\mathbf{V}$, and $\mathbf{D}$ are processed by the GPU and the results are passed back to the CPU in order to calculate the $\mathbf{S}$ matrix. The results are illustrated in Figure 9 for three block sizes $k = 16$, 64, and 256 by considering kernel execution and transfer of results back to CPU. It appears that all four approaches yield similar execution times

for alignments of profiles longer than 500. Moreover, the block size seems to have no influence on the performance. The reason is that the four variants implementing the data independent part exploit parallelism quite similarly.

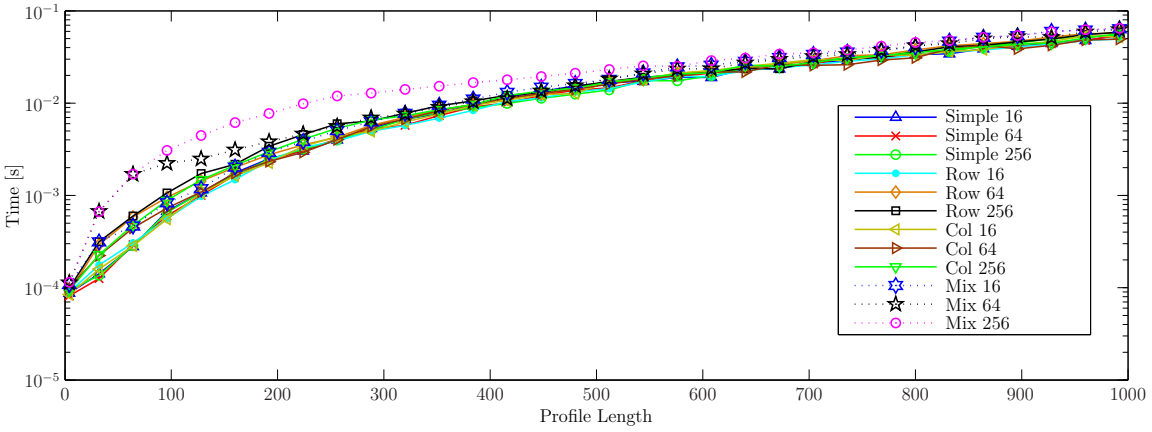

Figure 9. Runtime (in seconds) of profile-profile alignment algorithm ProfProfAlignScalProd on NVidia.

Next, the algorithm ProfProfAlignScalProd has been implemented by CUBLAS calculating the forward table using the library functions SAXPY (which takes the difference between two profiles) and SNRM2 (which calculates Euclidean distance). Figure 10 shows a comparison of this implementation with three implementations of Simple k (considering only kernel execution and transfer of results back to host memory) and an Intel CPU implementation via MKL using the library functions SAXPY and SNRM2 to establish the forward table. It appears that Simple k outperforms both the CUBLAS and the Intel CPU implementations by one order of magnitude. Moreover, the Intel CPU implementation is faster than the CUBLAS one up to profiles of length 700. The reason is that the CUBLAS function SNRM2 stores results back into the host memory for each cell.

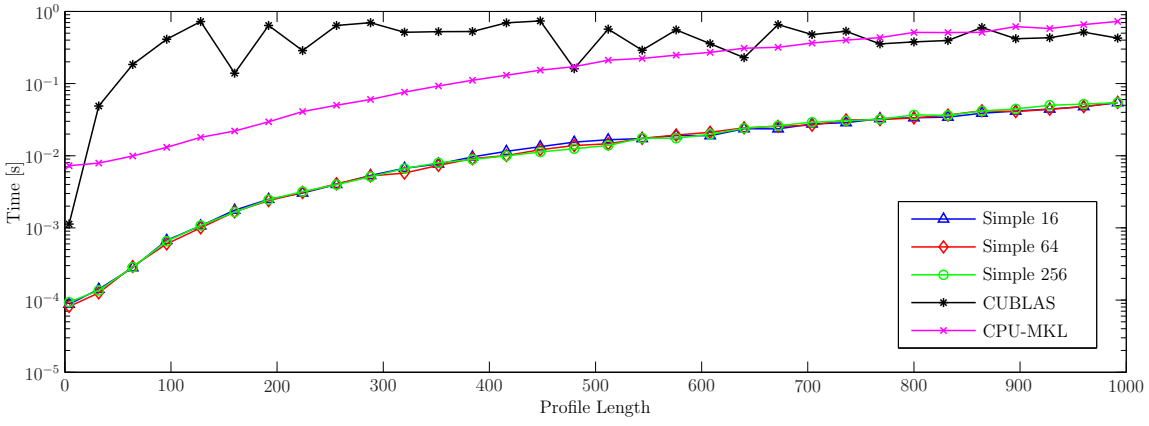

Figure 10. Runtime (in seconds) of profile-profile alignment algorithm ProfProfAlignScalProd on NVidia and Intel CPU using MKL.

The algorithm ProfProfAlignMatProd is difficult to implement for larger profiles. Indeed, the memory required to store the intermediate matrices on the device can become huge when compared with the size of the results, since only the data on the major diagonal are used. However, the algorithm ProfProfAlignMatProd can be realized using vector multiplication.
For this purpose, two variants have been considered; in each case, the forward table is decomposed into blocks of size $k \times k$:

- MatProd V1: use CUBLAS functions.

- MatProd V2 k: perform all calculations on the GPU (k threads per block).

Note that MatProd V1 k can be implemented with CUBLAS functions gemm for subtraction of matrices and dot for componentwise multiplication of vectors which corresponds to the multiplication of the diagonals of the involved matrices. Indeed, gemm performs the operation

$$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}. \tag{14}$$

Thus the subtraction of two matrices can be carried out by setting $\alpha = 1$, $\beta = -1$, and taking the identity matrix for $\boldsymbol{B}$. But the identity matrix has the same size as the matrix $\boldsymbol{A}$ so that unnecessary computations are performed. Hence, the idea to implement ProfProfAlignMatProd by MatProd V1 has been discarded.
On the other hand, we have specified the wavefront approach in Subsection 4.2. Its implementation can be based on two variants depending on the storage of data: In SMwavefront k, the data produced by the blocks are stored in global memory. These data will be transferred to shared memory when a new block is being launched that requires access to these data. In GMwavefront k, the data produced by the blocks are fully kept in global memory. In both cases, the blocks have size of $k \times k$. The results exhibit that both approaches have almost the same performance (Fig. 11 and 12). However, for larger length of profiles, GMwavefront k will become superior to SMwavefront k, since size of shared memory is limited. This figure also depicts the effect of communication-to-computation ratio which decreases with the increase of the block size as illustrated by the block sizes k=16, 64, and 256.

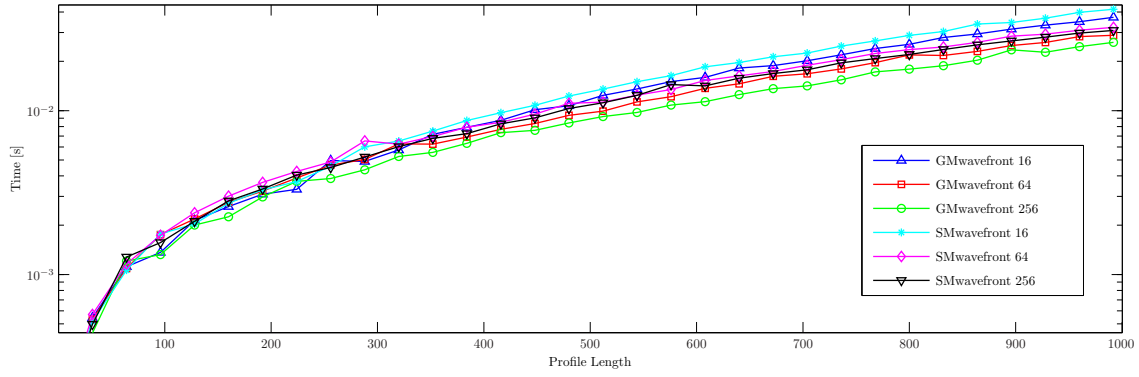

Figure 11. Runtime (in seconds) of profile-profile alignment algorithm using wavefront approach (shared vs. global memory) with profiles of length < 1000.

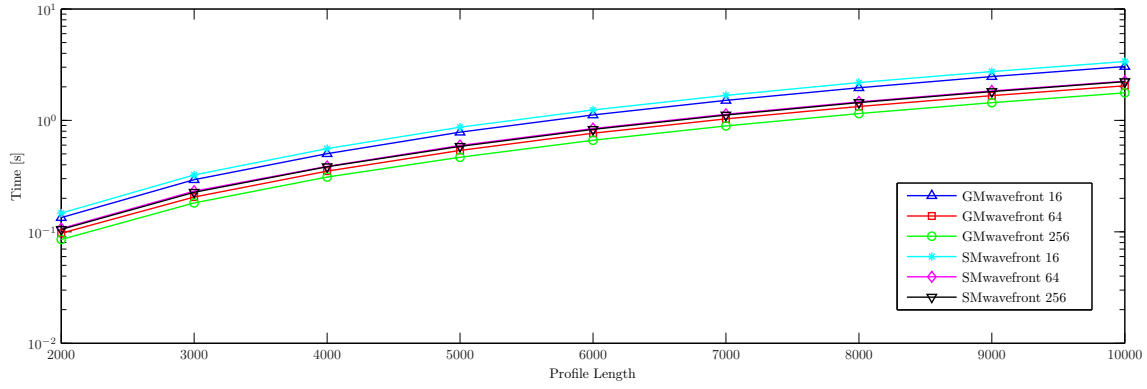

Figure 12. Runtime (in seconds) of profile-profile alignment algorithm using wavefront approach (shared vs. global memory) with profiles of length > 1000.

The anti-diagonal parallelism among blocks is exploited by CPU. This will incur some delay due to switching between GPU and CPU (Fig. 13). The results exhibit that both GMwavefront k and SMwavefront k have almost the same performance. The switching delay for profiles of length < 1000 is negligible.

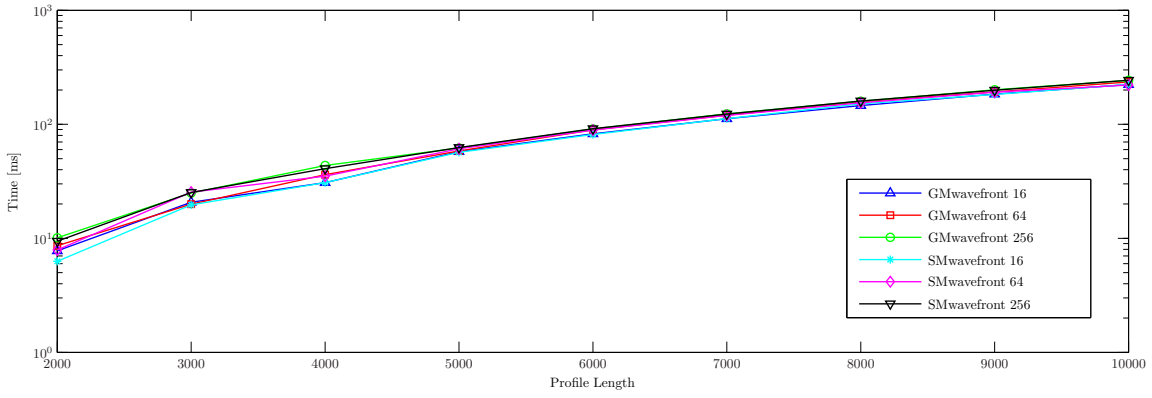

Figure 13. Switching delay (in milliseconds) between GPU and CPU using wavefront approach (shared vs. global memory) with profiles of length > 1000.

Furthermore, we have compared five implementations of the profile-profile alignment algorithm: Simple k, MatProd V2 k, SMwavefront k (for block sizes k=16, 64, and 256) and the Intel CPU implementation with and without the MKL. First, kernel execution and transfer of results back to host have been considered. The results in Fig. 14 exhibit that the Simple k approach performs best for profile length up to 500 while Simple k and SMwavefront k have almost similar execution times with a slight edge in performance to SMwavefront k for lengths longer than 500. The performance degradation of SMwavefront k for smaller length is due to the computation-to-communication cost since a smaller number of blocks is executed concurrently. As the profile lengths increase, multiple anti-diagonal block executions for SMwavefront k result in superior performance when compared with Simple k and MatProd V2 k (Fig. 15).

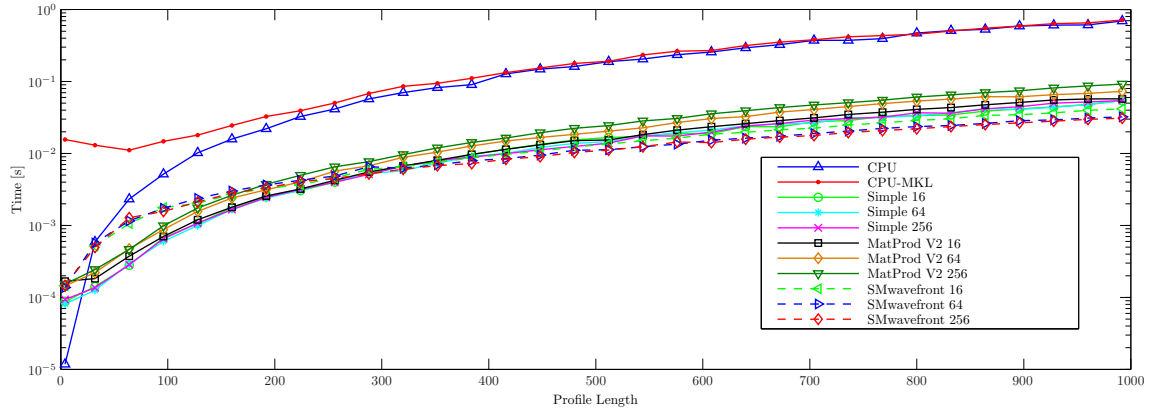

Figure 14. Runtime (in seconds) of profile-profile alignment algorithms on NVidia and Intel CPU (with and without MKL) by considering kernel execution and transfer of results back to CPU for profiles of length < 1000.

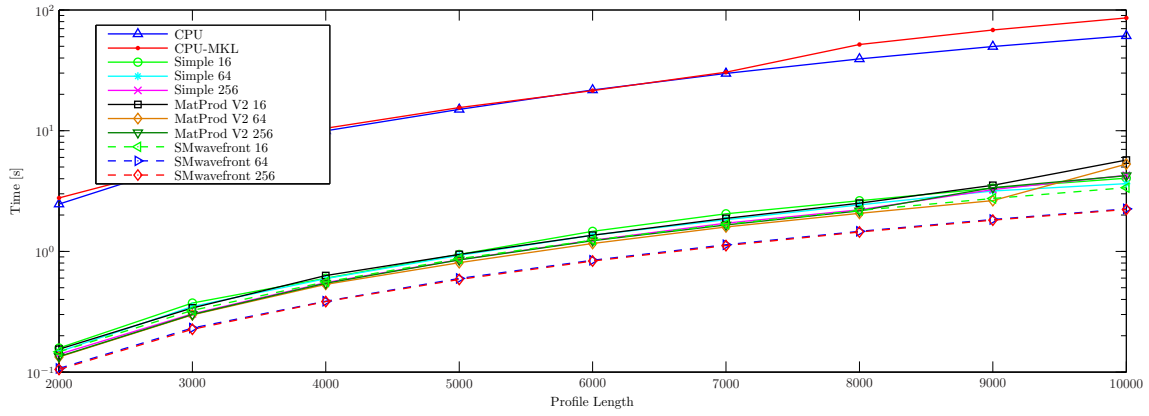

Figure 15. Runtime (in seconds) of profile-profile alignment algorithms on NVidia and Intel CPU (with and without MKL) by considering kernel execution and transfer of results back to CPU for profiles of length > 1000.

Second, kernel execution, memory allocation, and transfer of results back to host have been taken into account. Figure 16 depicts that MatProd V2 k has almost the same performance as the Intel CPU implementation (with and without MKL) because duplication of profiles for the purpose of vector subtraction is time consuming. When using the MKL BLAS1 routine SAXPY to calculate the difference between profile vectors, the new data will overwrite the old ones. To avoid this, the profiles should be pre-stored causing a degradation in performance. The SMwavefront k execution time is calculated by excluding switching delay between CPU and GPU. However, the impact of this delay on the performance of SMwavefront k is not significant for profiles of length < 1000, but the switching delay for longer profile lengths becomes a significant factor for the superior performance of SMwavefront k when compared with Simple k. For profiles of length > 1000, the runtime performance of MatProd V2 k is almost similar to that of Simple k and SMwavefront k (Fig. 17).

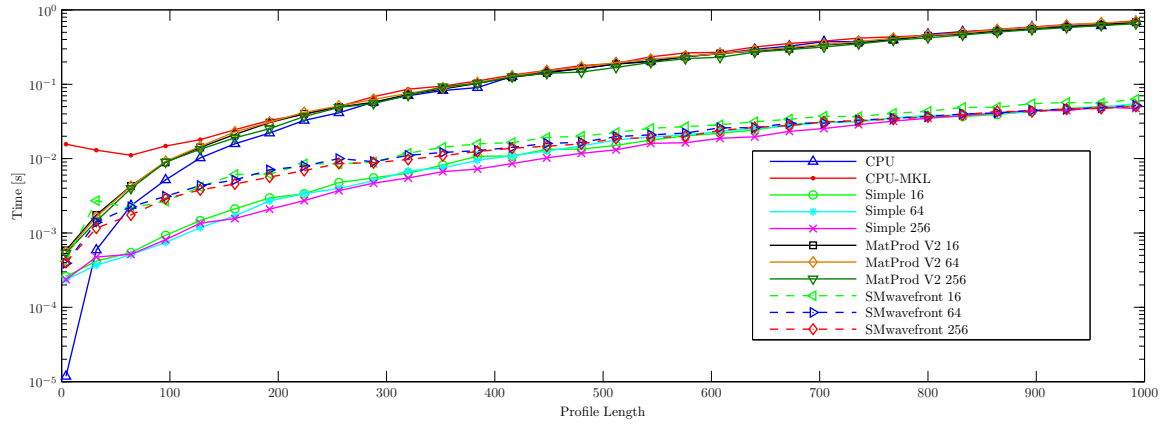

Figure 16. Runtime (in seconds) of profile-profile alignment algorithms on NVidia and Intel CPU (with and without MKL) by considering memory allocation, data transfer and kernel execution for profiles of length < 1000.

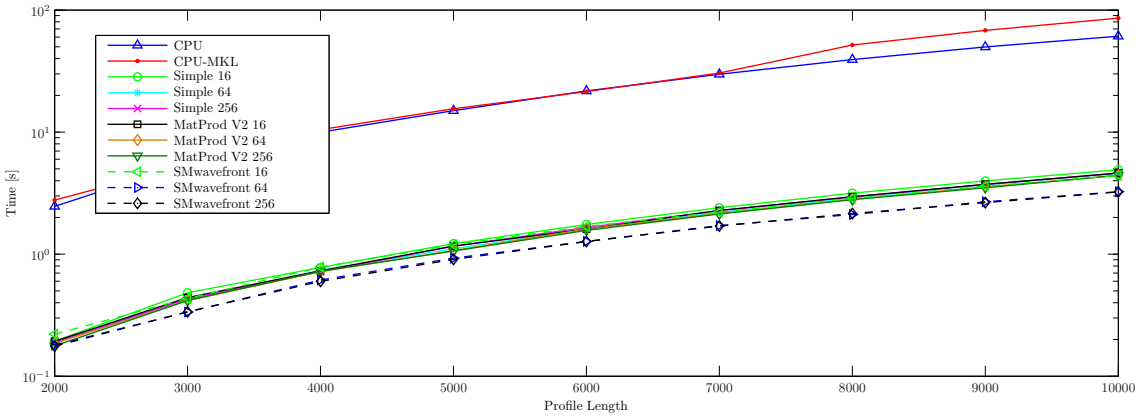

Figure 17. Runtime (in seconds) of profile-profile alignment algorithms on NVidia and Intel CPU (with and without MKL) by considering memory allocation, data transfer and kernel execution for profiles of length > 1000.

Next, the speed-ups attained with the NVidia implementations when compared with the Intel CPU implementations using MKL have been calculated. First, kernel execution and transfer of results back to the host have been considered. The results in Fig. 18 illustrate that NVidia implementations of Simple k and SMwavefront k achieve speed-up factors of one order of magnitude when compared with the Intel CPU implementations using MKL. For the profile lengths $>$ 1000, SMwavefront k exhibits maximum speed-up factor of about 38.5 (mean 30) while Simple k and MatProd V2 k achieve average speed-up factor of about 20 (Fig. 19). Second, this result remains valid when overheads for memory allocation are taken into account (Fig. 20 and 21). Note that MatProd V2 k does not have a significant speed-up due to duplication of profiles for subtraction purposes. Tables 1 and 2 provide more details which depicts that SMwavefront k performs much better than other implementations for longer profiles.

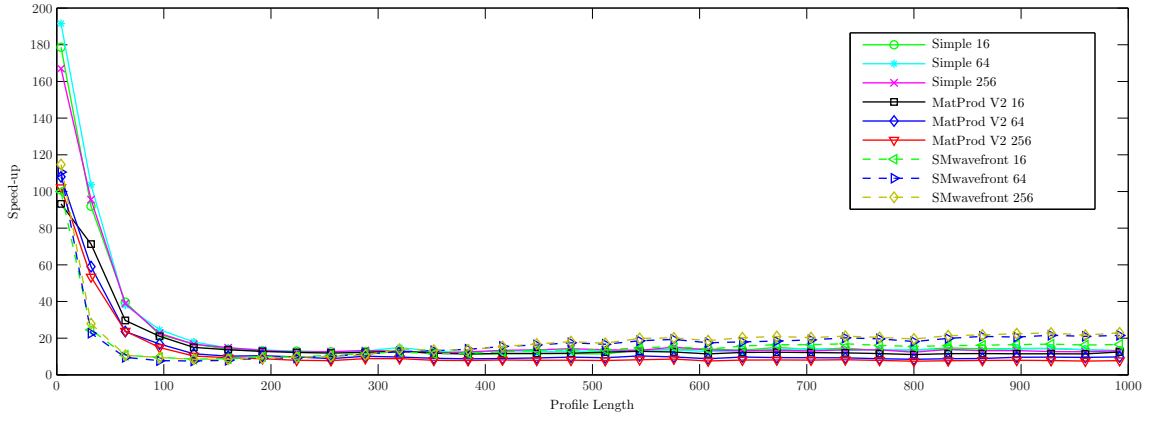

Figure 18. Speed-ups of Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution and transfer of results back to CPU memory for profiles of length < 1000.

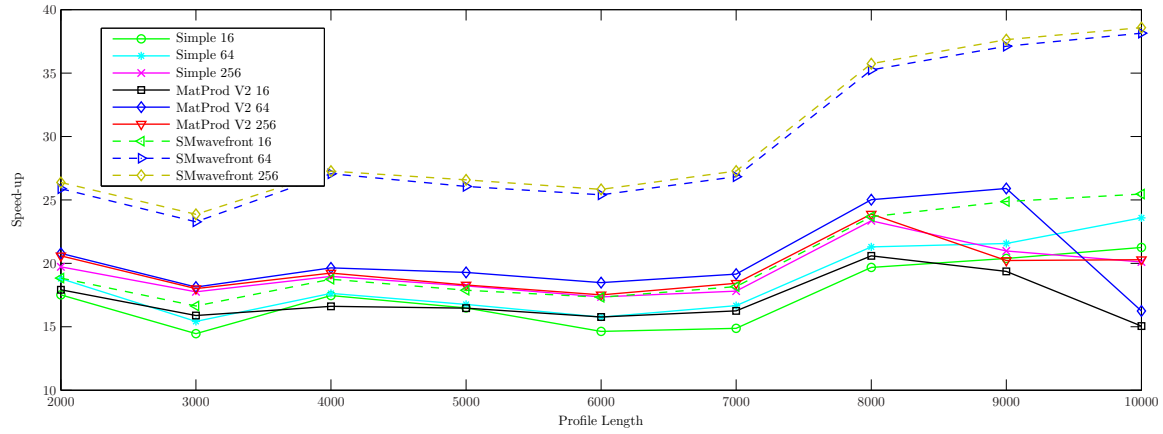

Figure 19. Speed-ups of Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution and transfer of results back to CPU memory for profiles of length > 1000.

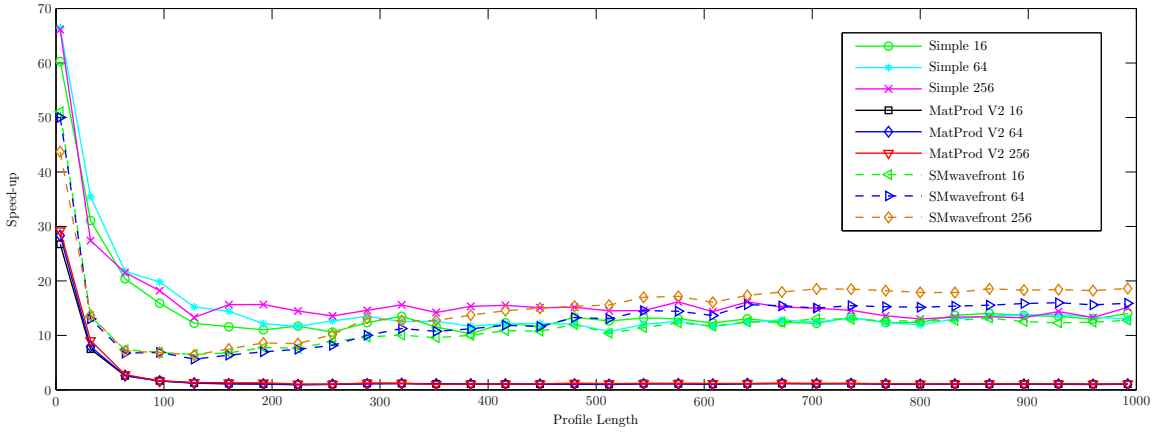

Figure 20. Speed-ups of Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution, memory allocation, and data transfer between CPU and GPU memory for profiles of length < 1000.

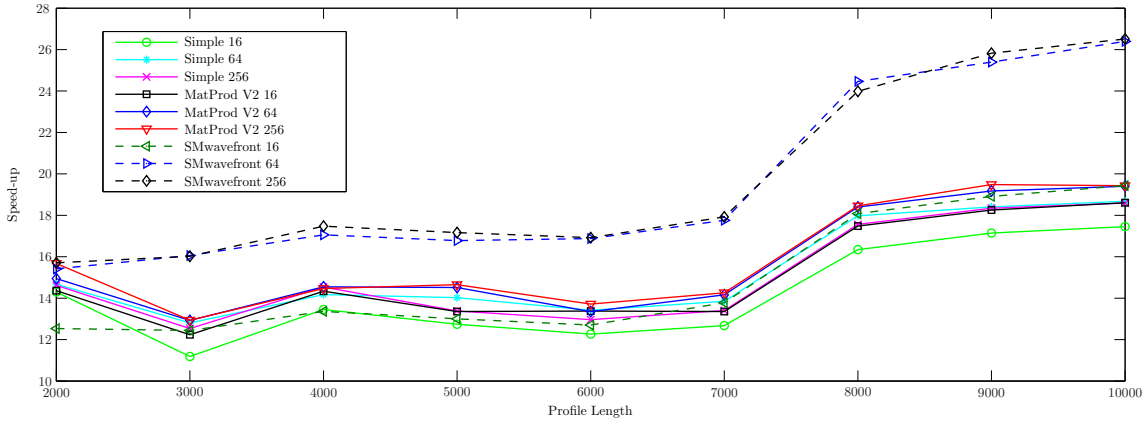

Figure 21. Speed-ups of Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution, memory allocation, and data transfer between CPU and GPU memory for profiles of length > 1000.

Table 1. Minimum, maximum, and average speed-ups for Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution and transfer of results back to CPU memory.

Speed-up	profile length < 1000			profile length > 1000		
	min	max	mean	min	max	mean
Simple 16	11.4237	178.667	22.0731	14.4551	21.2546	17.4177
Simple 64	12.1434	191.6012	23.0617	15.4259	23.5844	18.6107
Simple 256	11.8536	167.0107	21.8563	17.3517	23.3564	19.3647
MatProd V2 16	11.0705	93.2269	17.3302	15.0601	20.5957	17.1029
MatProd V2 64	8.5052	107.991	14.7831	16.2426	25.8997	20.2941
MatProd V2 256	7.4691	102.1288	13.1962	17.5116	23.8790	19.6056
SMwavefront 16	8.5569	99.9712	16.6403	16.6426	25.4623	20.1806
SMwavefront 64	7.5743	110.5914	18.9352	23.2700	38.1476	29.4485
SMwavefront 256	8.4524	114.7355	20.0702	23.8636	38.5810	29.9065

Table 2. Minimum, maximum, and average speed-ups for Simple k, MatProd V2 k, and SMwavefront over CPU-MKL by considering kernel execution, memory allocation, and data transfer between CPU and GPU memory.

Speed-up	profile length < 1000			profile length > 1000		
	min	max	mean	min	max	mean
Simple 16	10.2763	60.2915	14.9457	11.1831	17.4553	14.1728
Simple 64	10.6931	66.4489	15.5892	12.8044	18.6781	15.3341
Simple 256	13.0005	66.1113	16.9095	12.5326	18.5901	15.1040
MatProd V2 16	0.9829	26.7251	2.1586	12.2318	18.6078	15.0413
MatProd V2 64	0.9406	28.3095	2.1860	12.9247	19.4008	15.7134
MatProd V2 256	1.0225	29.2206	2.3378	12.9391	19.4863	15.8976
SMwavefront 16	6.5005	51.0477	12.0965	12.4434	19.4381	14.9164
SMwavefront 64	5.6300	50.0176	13.5337	12.4110	26.3981	19.5776
SMwavefront 256	6.3944	43.6919	15.4248	12.7083	26.5118	19.7270

To sum up, the theoretical floating point operations per second (FLOPS) and cell updates per second attained with Nvidia implementations and CPU-MKL have been calculated (Fig. 22 and 23). Theoretical FLOPS are used since actual FLOPS are depending on the hardware architecture. The average GFLOPS are given by three times 21 multiplications, 21 additions, and 21 subtractions (20 amino acids plus blank) for each cell to calculate the squared Euclidean distance. This large number of floating point operations is the reason for smaller values of cell updates per second (CUPS) and floating point operations per second (FLOPS) (Fig. 24 and 25). Another factor that contributes to small values of CUPS for Simple k and MatProd V2 k is due to the processing of the data dependent part on the host CPU. Tables 3 and 4 provide more details about FLOPS and CUPS, respectively. SMwavefront k achieves about 8 GFLOPS and 42 MCUPS on average and clearly outperforms Simple k and MatProd V2 k implementations.

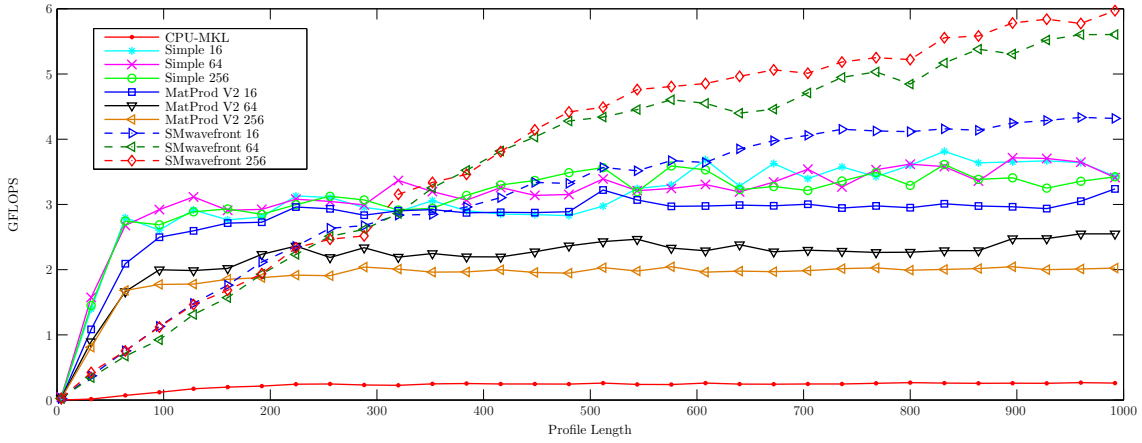

Figure 22. Performance (GFLOPS) of CPU-MKL, Simple k, MatProd V2 k, and SMwavefront k by considering kernel execution and transfer of results back to CPU memory for profiles of length < 1000.

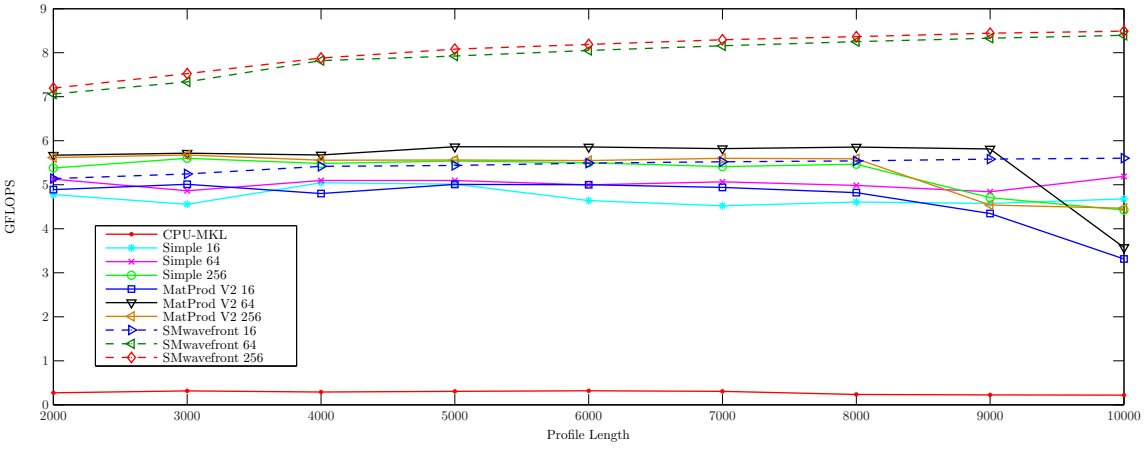

Figure 23. Performance (GFLOPS) of CPU-MKL, Simple k, MatProd V2 k, and SMwavefront k by considering kernel execution and transfer of results back to CPU memory for profiles of length > 1000.

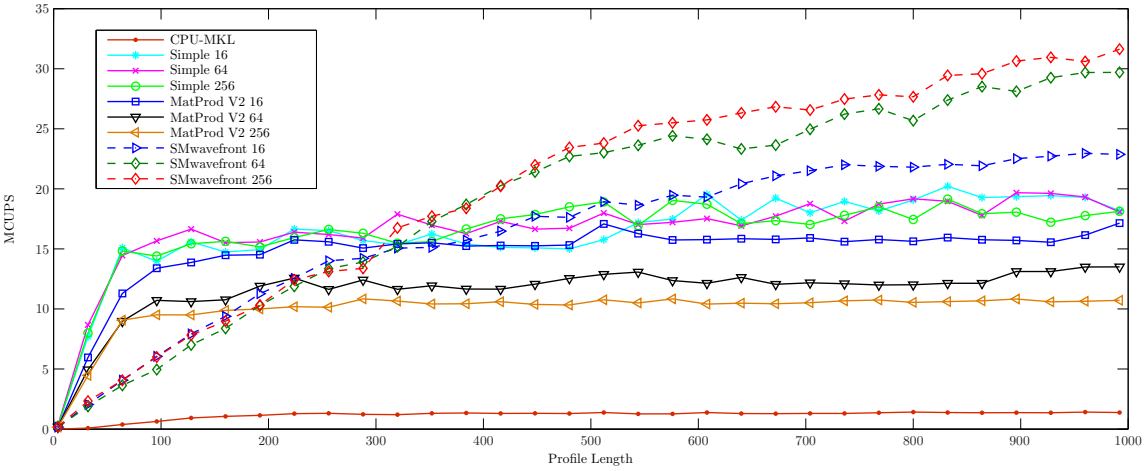

Figure 24. Performance (MCUPS) of CPU-MKL, Simple k, MatProd V2 k, and SMwavefront k by considering kernel execution and transfer of results back to CPU memory for profiles of length < 1000.

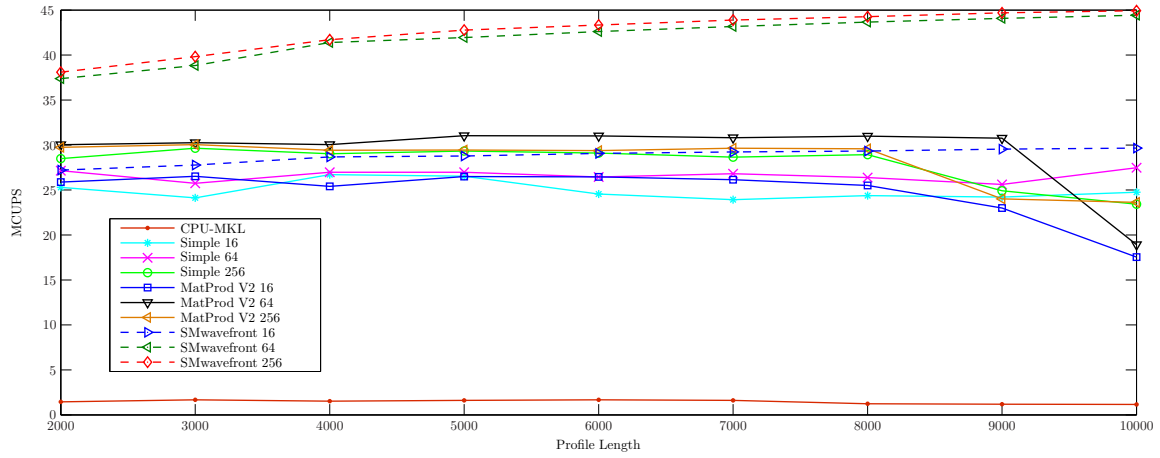

Figure 25. Performance (MCUPS) of CPU-MKL, Simple k, MatProd V2 k, and SMwavefront k by considering kernel execution and transfer of results back to CPU memory for profiles of length > 1000.

Table 3. Minimum, maximum, and average for GFLOPS for CPU-MKL, Simple k, MatProd V2 k, and SMwavefront k by considering kernel execution and transfer of results back to CPU memory.

GFLOPS	profile length < 1000			profile length > 1000		
	min	max	mean		max	mean
CPU-MKL	0.0002	0.2670	0.2197	0.2201	0.3170	0.2756
Simple 16	0.0404	3.8139	3.0569	4.5218	5.0463	4.7130
Simple 64	0.0433	3.7140	3.1094	4.8398	5.1905	5.0291
Simple 256	0.0377	3.6114	3.0647	4.4252	5.5998	5.2797
MatProd V2 16	0.0211	3.2372	2.7508	3.3144	5.0093	4.6805
MatProd V2 64	0.0244	2.5486	2.1586	3.5747	5.8635	5.5391
MatProd V2 256	0.0231	2.0433	1.8611	4.4646	5.6768	5.3510
SMwavefront 16	0.0226	4.3320	3.0583	5.1396	5.6038	5.4428
SMwavefront 64	0.0250	5.6054	3.5882	7.0603	8.3956	7.9255
SMwavefront 256	0.0259	5.9696	3.7859	7.1951	8.4910	8.0516

Table 4. Minimum, maximum, and average for MCUPS for CPU-MKL, Simple k, MatProd V2 k, and SMwavefront k by considering kernel execution and transfer of results back to CPU memory.

MCUPS	profile length < 1000			profile length > 1000		
	min	max	mean	min	max	mean
CPU-MKL	0.0016	1.4146	1.1668	1.1646	1.6275	1.4588
Simple 16	0.2860	20.2117	16.2485	23.9297	26.7088	24.9435
Simple 64	0.3067	19.6801	16.5991	25.6114	27.4666	26.6164
Simple 256	0.2674	19.1384	16.2904	23.4169	29.6419	27.9431
MatProd V2 16	0.1493	17.1513	14.6193	17.5391	26.5160	24.7716
MatProd V2 64	0.1729	13.5027	11.4729	18.9163	31.0323	29.3160
MatProd V2 256	0.1635	10.8376	9.8933	23.6256	30.0492	28.3202
SMwavefront 16	0.1601	22.9589	16.2339	27.1116	29.6536	28.8059
SMwavefront 64	0.1771	29.6980	19.0420	37.3809	44.4270	41.9456
SMwavefront 256	0.1837	31.6278	20.0917	38.0948	44.9318	42.6131

6. Conclusion

The results exhibit that modern graphics cards can be utilized as efficient hardware accelerators for profile-profile alignment. This involves not only profile-profile alignment but also profile-sequence alignment that are both part of progressive alignment. As we have already proposed an efficient solution of profile-sequence alignment on NVidia, the basic progressive alignment method could be efficiently implemented on NVidia, too. This solution would allow to calculate large scale multiple alignments significantly faster on relatively low-cost GPU than on the CPU. The wavefront approach is a very good candidate for the implementation of profile-profile alignment on a GPU because it has better hardware utilization and speed-up compared to Simple k and MatProd V2 k.

Acknowledgement

We would like to thank Cem Bassoy for helpful comments. The work of the first author was sponsored by DAAD and Higher Education Commission of Pakistan.

References

[1] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide 4.0, 2011
[2] Altschul S., Gish W., Miller W., Myers E., Lipman D., Basic local alignment search tool, J. Mol. Biol., 215, 403-410, 1990
[3] Bassoy C., Torgasin S., Yang M., Zimmermann K.H., Accelerating scalar-product based sequence alignment using graphics processor units, Signal Process. Syst., 61, 117-125, 2010
[4] Che S., Li J., Sheaffer J., Skadron K., Lach J., Accelerating compute-intensive applications with gpus and fpgas, Application Specific Processors, SASP, 101-107, 2008
[5] Chenna R. et al., Multiple sequence alignment with the clustal series of programs, Nucleic Acids Res., 31, 3497-3500, 2003
[6] Durban R., Eddy, S., Krogh, A., Mitchison, G., Biological sequence analysis: Probabilistic models of proteins and nucleic acids (Cambridge Univ. Press, Cambridge, 1998)
[7] Dzivi P., Sequence alignment using graphics processor units, Master's thesis, The University of Western Australia, 2008
[8] Edgar R.C., Sjölander K., A comparison of scoring functions for protein sequence profile alignment, Bioinf., 20, 1301-1308, 2004
[9] Stacey K.A., Recombination, In: Kendrew John, Lawrence Eleanor (eds.), The Encyclopedia of Molecular Biology, Blackwell Science, Oxford, 945-950, 1994
[10] Gusfield D., Algorithms on strings, trees, and sequences (Cambridge Univ Press, Cambridge, 1997)
[11] Kun S., VLSI array processors (Prentice Hall, Englewood Cliffs, NJ, 1988)
[12] Liu Y., Schmidt B., Maskell D.L., MSAProbs: multiple sequence alignment based on pair hidden Markov models and partition function posterior probabilities, Bioinf., 26, 1958-1964, 2010
[13] Manavski S.A., Valle G., Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment, BMC Bioinf., 9 (Suppl. 2), 2008
[14] Munekawa Y., Ino F., Hagihara K., Design and implementation of the smith-waterman algorithm on the cuda-compatible gpu, In: 8th IEEE International Conference on BioInformatics and BioEngineering (October 2008), BIBE, 1-6, 2008
[15] Needleman S.B., Wunsch C.D., A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol., 48, 443-453, 1970
[16] Notredame C., Higgins D., Heringa J., T-coffee: a novel method for fast and accurate multiple sequence alignment, J. Mol. Biol., 302, 205-217, 2000
[17] Pearson W., Lipman D., Improved tools for biological sequence comparison, In: Proceedings of the National Academy of Sciences of the United States of America, 85, 2444-2448, 1988

[18] Schatz M., Trapnell C., Delcher A., Varshney A., High-throughput sequence alignment using graphics processing units, BMC Bioinf., 8, 2007
[19] Shaffer, C., Next-generation sequencing outpaces expectations, Nat. Biotech., 25, 2007
[20] Waterman M., Introduction to Computational Biology (Chapman and Hall, London, 1995)
[21] Waterman M., Smith T., Identification of common molecular subsequences, J. Mol. Biol., 147, 195-197, 1981
[22] Zimmermann K.H., Introduction to protein informatics (Kluwer Academic Publishers, Norwell, MA, 2003)

Appendix A: Proof of identities

Theorem A.1.
The matrix identity in (11) holds.

Proof. Take the $l \times n$ matrix

$$H_1 = Q_- - Q = \left(-p - \boldsymbol{q}_1, \ldots, -p - \boldsymbol{q}_n\right),$$

where Q_- is the $l \times n$ matrix defined in (7).
Expanding the matrix H in (11) gives

$$\begin{aligned}
H &= \operatorname{diag}[H_1^T \cdot H_1] \\
&= \operatorname{diag}\left[\begin{pmatrix} -p_1 - q_{1,1} & \cdots & -p_l - q_{1,l} \\ \vdots & \ddots & \vdots \\ -p_1 - q_{n,1} & \cdots & -p_l - q_{n,l} \end{pmatrix} \cdot \begin{pmatrix} -p_1 - q_{1,1} & \cdots & -p_1 - q_{n,1} \\ \vdots & \ddots & \vdots \\ -p_l - q_{1,l} & \cdots & -p_l - q_{n,l} \end{pmatrix}\right] \\
&= \operatorname{diag}\begin{bmatrix} (-p_1 - q_{1,1})^2 + \ldots + (-p_l - q_{1,l})^2 & \cdots \\ \vdots & \ddots \\ & \cdots & (-p_1 - q_{n,1})^2 + \ldots + (-p_l - q_{n,l})^2 \end{bmatrix} \\
&= \operatorname{diag}\begin{bmatrix} d(-p - \boldsymbol{q}_1)^2 & \cdots \\ \vdots & \ddots & \vdots \\ & \cdots & d(-p - \boldsymbol{q}_n)^2 \end{bmatrix}.
\end{aligned}$$

Taking diagonal entries, we obtain the squared Euclidean distances

$$H = (d(-p - \boldsymbol{q}_j)^2)_j.$$

□

Theorem A.2.
The matrix identity in (12) is valid.

The proof is similar to that of Theorem A.1.

Theorem A.3.
The matrix identity in (13) holds.

Proof. Pick the $l \times (m \times n)$ matrix

$$D_1 = \left(\boldsymbol{p}_1 - \boldsymbol{q}_1, \ldots, \boldsymbol{p}_1 - \boldsymbol{q}_n, \ldots, \boldsymbol{p}_m - \boldsymbol{q}_1, \ldots, \boldsymbol{p}_m - \boldsymbol{q}_n\right).$$

By (13), we obtain

$$\begin{aligned}
D &= \operatorname{diag}[D_1^T \cdot D_1] \\
&= \operatorname{diag}\left[\begin{pmatrix} p_{1,1} - q_{1,1} & \cdots & p_{1,l} - q_{1,l} \\ & \vdots & \\ p_{m,1} - q_{n,1} & \cdots & p_{m,l} - q_{n,l} \end{pmatrix} \cdot \begin{pmatrix} p_{1,1} - q_{1,1} & \cdots & p_{m,1} - q_{n,1} \\ & \vdots & \\ p_{1,l} - q_{1,l} & \cdots & p_{m,l} - q_{n,l} \end{pmatrix}\right] \\
&= \operatorname{diag}\begin{bmatrix} d(\boldsymbol{p}_1 - \boldsymbol{q}_1)^2 & \cdots \\ \vdots & \ddots & \vdots \\ & \vdots & d(\boldsymbol{p}_m - \boldsymbol{q}_n)^2 \end{bmatrix}.
\end{aligned}$$

By taking diagonal entries, we obtain the squared Euclidean distances given by

$$\boldsymbol{D} \;=\; (d(\boldsymbol{p}_i - \boldsymbol{q}_j)^2)_{i,j}.$$

□