Cluster Computing
https://doi.org/10.1007/s10586-018-2421-7

Improving the performance of Smith–Waterman sequence algorithm on GPU using shared memory for biological protein sequences

D. Venkata Vara Prasad[1] · Suresh Jaganathan[1]

Received: 14 February 2018 / Revised: 2 March 2018 / Accepted: 6 March 2018

Abstract
In Bioinformatics, sequence alignment algorithm aims to find out whether biological sequences (e.g., DNA, RNA, or Protein sequences) are related or not. A variety of algorithms are developed, Smith–Waterman Algorithm (SW) is a well-known local alignment algorithm to find the similarity of two sequences and provides optimal result using dynamic programming. As the size of sequence database is doubling about every 6 months, the computational time also increases. Sequence alignment algorithms performance can have improved by using the parallel computing technology on the GPU. In this paper, we proposed a method to improve the performance of SW algorithm by using GPU's shared memory instead of global memory. By using shared memory, the data being transferred between the global memory and processing elements is reduced, which in turn improves the performance. The tabulated result showed positive sign of correctness in proposed method and tested using UniProt sequence database.

Keywords Bioinformatics · Sequence alignment · CUDA-GPU · Memory · Smith–Waterman algorithm

1 Introduction

In Bio-informatics, sequence alignment is the process of aligning the sequence of DNA, RNA, and protein to find the similarity region and helps to find the functional and structural relationship between the sequences. Using these relationships, the information about the unknown or new sequences can be identified. Sequence alignment is a widely used operation in the field of bioinformatics and computational biology. It has many applications. For example, if information about one of the sequences is known (e.g., cancer gene), when it matches with the unknown sequence then this information could be transferred. This helps in early disease diagnosis. Other applications like the study of evolutionary development, the history of species and their groupings. There are a wide variety of algorithms for sequence alignment, heuristic algorithms [1–3] or probabilistic method is fast but does not provide best matches whereas dynamic programming method, is slow but provides best matches. The sequence aligners are based on hash tables, and suffix/prefix tries.

Hashtable has short read hash or reference genome. In short read hash, aligners include RMAP [4], MAQ [5], SHRiMP [6] which causes overhead due to scanning of the genome when only fewer reads are aligned. In reference genome hash, aligners like SOAP [7], PASS [8] are parallelized using threads, but it requires a large amount of memory to formulate the index. But in suffix/prefix tries aligner, the advantage is only one search pass is required to find the multiple alignment copies in the reference genome. BLASTP is a heuristic algorithm, which is faster than any other approach. BLASTP takes a long time for scanning large protein sequences on sequential architecture. To reduce the processing time, the implementation of BLASTP is parallelized on GPU using CUDA. In GPU, two-level parallelism is used to gain efficiency-: coarse-grained and fine-grained. When compared to BLASTP [2], BLASTP implementation in GPU achieves speed up to 10% on an NVIDIA GeForce GTX 295.

The two basic alignments are local alignment (Smith–Waterman algorithm) and global alignment (Needleman

✉ D. Venkata Vara Prasad
dvvprasad@ssn.edu.in

Suresh Jaganathan
sureshj@ssn.edu.in

[1] Department of Computer Science and Engineering, SSN College of Engineering, Chennai, Tamilnadu, India

Published online: 13 March 2018

Wunsch algorithm), which follows dynamic programming method [9]. Global alignment forces the alignment to map the entire length of the query sequence, whereas local alignment identifies regions of similarity within long sequences as shown in Fig. 1. Smith–Waterman is a local sequence alignment algorithm which is used for finding regions of maximum similarity between two biological sequences and it also provides the optimal results. This algorithm has a quadratic time complexity which needs a long computation time for large-sized data. The Smith–Waterman algorithm provides optimal local alignment for the sequence pairs. It requires a large amount of processing time due to its high computational complexity, which is proportional to the product of the length of two sequences. To improve the performance of the Smith–Waterman algorithm, it is parallelized in GPU [10–13]. In CUDASW ++3.0 [14], the CPU and GPU SIMD instructions are combined to provide faster protein search algorithm. In this approach, the sequence alignment workloads are distributed over CPUs and GPUs based on their computing power. In CPUs, Streaming SIMD extensions (SSE)-based vector units and multi-threading are used. In GPUs, PTX SIMD instructions are used to achieve more parallelism than using SIMT. The current technology of parallel computing on the GPU developing very rapidly. In GPU computing, with thousands of threads in hundreds of cores on the GPU, massively parallel programming can be applied to obtain performance improvements. Graphics Processing Units (GPUs) provides parallel programming processors, large memory bandwidth, and large computational power compared to the computing capability of CPUs. GPU includes its own RAM separate from the system memory and supports thousands of concurrent resident hardware threads. Multiple threads are grouped together into blocks and blocks into a grid. A kernel is launched that spawns a grid of threads, each executing the kernel function. Threads of the same block run on one share memory and can share data through fast on-chip shared memory and can be synchronized by barriers. Such synchronization is not available between the block. Efficient, coherent access of the on-chip and off-chip memory resources can significantly impact the observed memory throughput.

```
Global  FTFTALILLAVAV
        F--TAL-LLA-AV

Local   FTFTALILL-AVAV
        --FTAL-LLAAV--
```

Fig. 1 Sequence alignment

Higher throughput can be critical to the performance of memory intensive applications like sequence alignment. The off-chip global memory is slower and larger in size. Global memory access on the GPU takes more than 100 × as many clock cycles access the memory and can be improved if threads within a SIMD warp access contiguous global memory addresses, resulting in fewer memory transactions. The shared memory is smaller but faster [15] and can be used to communicate between threads within the same block. Shared memory bandwidth can also be affected due to bank conflicts.

CUDA introduced by Nvidia is a programming environment for writing and running parallel processing applications on the Nvidia GPU. CUDA allows programmers to develop GPGPU applications using the extended C programming language [16]. CUDA program launches the kernel which executes N times in parallel by concurrently executing threads in GPU. Concurrent threads are grouped together to form a thread block and set of thread blocks are called grid. Each thread has unique *(threadI dx, blockI dx)*. Kernel function call includes size of the grid *(dimGrid)* and a thread block *(dimBlock)* to be launched, i.e.,

$$Kernel<<<dimGrid, dimBlock>>>(listofparameters).$$

Thread of different block communicates through global memory in on-chip memory. Threads of same blocks communicate through the per-block shared memory. Each has executed the thread in a group called wraps using Single Instruction Multiple Thread (SIMT). Sequence Alignment algorithms performance can improve with the fast emergence of various CUDA-enabled GPU architectures. In bioinformatics, one of the standard algorithms to find the optimal similarity between sequences in the database is Smith–Waterman algorithm [17] which uses dynamic programming. Even though it provides an optimal result, it is much slower compared to other heuristic algorithms.

To address this issue, an efficient parallel computing is required for sequence database searches to reduce the execution time and to increase the performance. In this paper, we propose a method to enhance the Smith–Waterman algorithm to achieve reduced execution time and fully utilizing the shared memory instead of global memory. Paper is organized as Sect. 2 presents a review of related works, Sect. 3, deals with proposed architecture and explains the concept in detail, Sect. 4, does a detailed analysis of proposed work by experimenting and Sect. 5 concludes the work with possible feature enhancement.

2 Related works

The Smith–Waterman algorithm is modified to perform pairwise alignment faster than previous algorithms. Modified the overlap-based algorithm where it subdivides the iterative computation of elements of a matrix among blocks of the processor. Each block can simply recompute the data it needs instead of waiting for other processors to compute them. This technique is applied to parallelize a dynamic programming problem that allows the computation at different processors to overlap. This leads to an efficient algorithm for pairwise sequence alignment of two large sequences using a parallel version of the Smith–Waterman algorithm on GPU.

The Smith–Waterman algorithm is executed using different available parallel models like MPI, OpenMP, Hybrid and associative massive parallelism, an architecture consisting of clear speed co-processor and convey computer [18]. Parallel models provide an excellent platform on which to a run sequence alignment, and those parallel architectures will be able to cope far more easily with the vast data produced by the high throughput sequence. Smith–Waterman algorithm formulation is modified based on the anti-diagonals structure of data. This representation focuses on parallelism of parts in algorithm without changing the initial formulation of the algorithm. The formulation used is a rotation of score matrix. This transformation explicit the parallelism contained in the algorithm and facilitated its exploitation across different platforms of parallelization. Amadou and Oumarou [19] proposed mathematical modeling for calculating the time for matrix's scoring on the OpenMP and GP-GPU. This equation sets allows making a wise choice in the number of thread (OpenMP) and the size of the grid computing (GP-GPUs).

Smith–Waterman is known to be a highly compute-intensive task, and it is based on dynamic programming. But dynamic programming has its own drawbacks such as heavy memory consumption and a significant amount of computations. Issues in parallelizing using anti-diagonal approach are, (i) high amount of cache-miss and data fetch operations for both row-major and column-major data organization and (ii) complex computations for data reordering in anti-diagonal order. Harsh Shukla and Monika Shah [11] introduced a version of the parallel scan Smith–Waterman algorithm to improve performance, which reduces it up-shift and down-shift to a single stage for referring left-dependencies using maximum parallel scan. Observed remarkable performance gain by comparing the work with other approaches like anti-diagonals and blocked anti-diagonal for both constant gap model and affine gap model.

Algorithms like Levenshtein edit distance, Longest Common Subsequence, and Smith–Waterman used dynamic programming approach to find similarities between two sequences. Shankar et al. [20] tried to find an appropriate method to compute similarity in gene/protein sequence, both within the families and across the families. Authors did a comparative analysis and time complexity for their proposed method and revealed that Smith-waterman approach is appropriate method when gene/protein sequence belongs to the same family, and Longest Common Subsequence is best suited when sequence belong to two different families.

Zeiad et al. [21] did a comparative study for a parallelized version of the Smith–Waterman algorithm using different models like MPI, OpenMP, and hybrid MPI/OpenMP and found in OpenMP, shared memory opens the possibility to have immediate access to all data from all processors without explicit communication, which improves the Smith–Waterman algorithm to some extent. Liang et al. [22] focused on improving the SW algorithm in a different way, i.e., by improving the mapping, especially for short query sequences by better usage of shared memory. When users input a query sequence, the extended package determines how to run the query based on the length of the query sequence as well as the space of shared memory per streaming multiprocessor, but the data set used for verification is very short, approximately it is 20. Bustamam et al. [10] did the same job by implementing three parallelization models, which includes Inter-Task Parallelization, Intra-task Parallelization, and a combination of both models. In Inter-task parallelization model, each task is assigned to exactly one thread running separately and in parallel, whereas in Intra-task parallelization model, each task is assigned to exactly one thread and executed separately in parallel. In the third model, the computation processes are separated using the threshold value. If the subject sequence length is less than the threshold, then the calculation run by Inter-task parallelization, whereas if the subject sequence length greater than or equal to the threshold run it by Intra-task parallelization.

3 System architecture

In existing parallel versions of SW algorithm, the computed matrix is stored in the global memory of GPU. Whenever the data is required from the matrix, the value will be fetched from the matrix in global memory then processed and stored back in global memory. Many cycles are required to fetch data from global memory. To improve the performance of SW algorithm, shared memory of GPU (Fig. 2) is used to store the matrix during computation. In GPU, data fetch from shared memory takes less than a

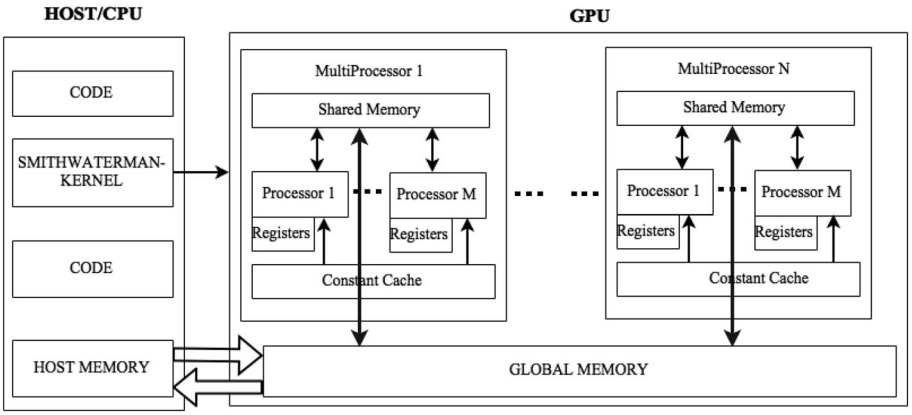

Fig. 2 Proposed system

cycle. Since the memory size of shared memory is less, the matrix needs to be split in such a way that it fits in memory and provide the same result as when global memory is used. Tiling concept in CUDA is used to process the matrix. Smith–Waterman algorithm (refer Algorithm 1) is implemented in three different methods for comparison.

The first method is sequential, comprises of three main steps, (i) setting up the dynamic programming matrix, (ii) iteratively compute the scores of the cells and (iii) identify the optimal alignment through a final traceback. *The first step*, let us consider, *CGTGAATTCAT* (*sequence A*) and *GACTTAC* (*sequence B*) be the sequence for local alignment. A matrix with (A + 1, B + 1) size is formed to store the sequences. The values in the first row and first column are set to zero as shown in Fig. 3. The *second step* of the algorithm is filling the entire matrix, so it is more important to know the neighbor values (i.e., diagonal, upper and left) of the current cell to fill the cell as shown in Fig. 4.

The *final step* for the alignment is trace backing. Find out the maximum score obtained in the entire matrix for the local alignment of the sequences. The traceback begins from the position which has the highest value and then pointing back with the pointers. To find out the possible previous value, then move to next possible previous value and continue until reaching the score zero as shown in Fig. 5. It is possible to find two different pointers pointing out from the same cell, where both alignments can be considered. The best one is found by scoring each alignment separately and finding maximum score among them (Fig. 6).

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0											
A	0											
C	0											
T	0											
T	0											
A	0											
C	0											

Fig. 3 Initialization of matrix

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	5	1	5	1	0	0	0	0	0	0
A	0	0	1	2	1	10	6	2	0	0	5	1
C	0	5	1	0	0	6	7	3	0	5	1	2
T	0	1	2	6	2	2	3	12				
T	0											
A	0											
C	0											

Fig. 4 Matrix filling with scores

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	5	1	5	1	0	0	0	0	0	0
A	0	0	1	2	1	10	6	2	0	0	5	1
C	0	5	1	0	0	6	7	3	0	5	1	2
T	0	1	2	6	2	2	3	12	8	4	2	6
T	0	0	0	7	3	0	0	8	17	13	9	7
A	0	0	0	3	4	8	5	4	13	14	18	14
C	0	5	1	0	0	4	5	2	9	18	14	15

Fig. 5 Traceback of the alignment

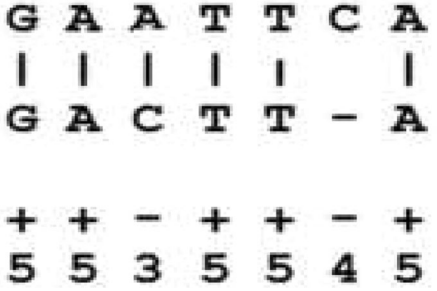

Fig. 6 Scoring of the best alignment

Algorithm 1: Smith-Waterman Algorithm

Input : Two sequences $A = a_1, a_2 \ldots, a_n$ and $B = b_1, b_2 \ldots, b_m$
Output: Optimal local alignment and score α
Initialization: Set $H_{k,0} = 0$ for $0 \le k \le n$ and $H_{k,1} = 0$ for $0 \le l \le m$
for $i = 1, 2, \ldots, n$ do:
  for $j = 1, 2, \ldots, m$ do:

$$H_{i,j} = max \begin{cases} 0 \\ H_{i-1,j-1} + s(a_i, b_j) \\ H_{i-k,j} - (G_s + kG_e) \\ H_{i,j-1} - (G_s + lG_e) \end{cases}$$

set backtrace $T(i,j)$ to the maximizing pair (i,j)
set $(i,j) := \arg\max\{H_{i,j} | i = 1,2, \ldots n, j = 1,2, \ldots, m\}$
the best score is $\alpha := H_{i,j}$ repeat
if $T(i,j)(i-1, j-1)$
  print (x_{i-1}, y_{j-1})
else if $T(i,j) = (i-1, j)$
  print $(x_{i-1}, -)$
else
  print $(-, y_{i-1})$
set $(i,j) := T(i,j)$
until $H_{i,j} = 0$

The second method is OpenMP, which uses shared memory and computation are performed to decompose into multiple tasks. Tasks are performed by the available computational units and the treatment to be performed, and data variables are stored in a location accessible to all processing units. The algorithm comprises three main steps, (i) initialization of a matrix, (ii) matrix filling with the appropriate scores using parallelism and (iii) trace back the sequences for a suitable alignment. Steps 1 and 3 are similar to the Sequential method. Computation of matrix in the Smith–Waterman algorithm is parallelized using striped alignment method in OpenMP. Matrix is split in row-wise according to the available number of threads and threads are executed in parallel as shown in Fig. 7.

The third method is a parallel implementation of Smith–Waterman, which uses the global memory of GPU. Initialization and Traceback steps execute as CPU code, and the matrix is computed after invoking the GPU. This method consists of three separate steps, (i) calculation of alignment scores, (ii) determination of tracebacks and (iii) producing the alignments as output. Sequences are formed in a matrix, row representing the number of sequences (X), with length (N). These sequences are padded with the special character when shorter than N. All sequences are copied in a single string x of length $X \times N$. On the column,

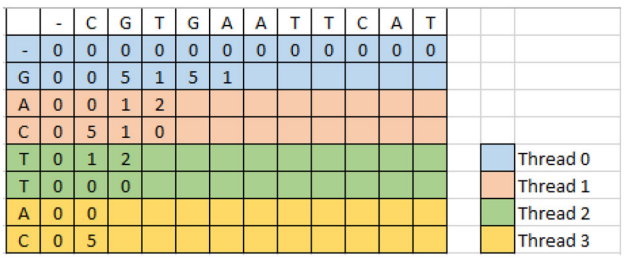

Fig. 7 Matrix filling with scores

there are Y target sequences, each of length M are placed and are padded when shorter than M. The target sequences are copied in a single string y of length $Y \times M$.

In the *first step*, the alignment score for each sequence alignment is calculated. The strings x and y are copied to the global memory of the GPU. Each sequence alignment is calculated in parallel. During the alignment, there will be $X \times Y$ threads active, and $X \times Y \times min(N, M)$ threads will be in active during the peak performance. In the entire phase, the maximum value is tracked, which is necessary to decide which tracebacks need to be calculated.

In the *second step*, using the maximum value tracked in step 1, the traceback of each alignment is determined. Based on user-specified value, such as the maximum value in the matrix required, the tracebacks are calculated in parallel. The process in this step is reverse of the first, which starts at the bottom-right. The profiles are stored in main memory.

The *third step* is executed on the CPU, which produces the alignment profiles. Since the algorithm is executed in parallel, the output profiles are not in order. So, the order of input and the order of output may vary. The output contains information about each profile, including sequence information, target sequence information, score and the start and end of the alignments.

3.1 Calculation of alignment scores

Global memory is the main memory which has a size from 100 MB to 12 GB. It is used to store the sequence strings, the maximum value of matrix which is used for traceback, and the scores. This memory is relatively slow compared to shared memory. But the shared memory per block is limited to 48 KB. Because of its faster access the intermediate results while processing the alignments are stored in the shared memory.

Each sequence alignment matrix is divided into small matrices of size 8×8. Each cell in the 8×8 matrices is mapped to blocks of 64 threads. The scores and the maximum value are stored in shared memory. There will be several data transfers from host to main memory. During alignment, maximum values from the neighboring blocks are retrieved from global memory. The current cell value is computed using the scores from a neighboring cell. For example, if it is the first block, scores are initialized to zero. The cell which has value for all neighboring cell are calculated in parallel. Calculations start at the top-left and pass through the matrix via the diagonal to the bottom-right. The maximum number of threads are active while processing diagonal cells. The idle threads in a block are used to find the maximum value. Once the smaller matrix is computed, the results are copied to global memory. Similar to the cells within the matrix, each block depends on the

three surrounding blocks as shown in Fig. 8. During the start, the total number of launched blocks are $X \times Y$ with 64 threads. The maximum number of thread blocks launched is $(X \times N \times Y \times M)/64$.

3.2 Determination of trace-backs

Once the alignment matrix is calculated, traceback is done to generate the aligned sequence profiles based on the computed value in the matrix. This step is the reverse process of the previous step. The application process starts at the bottom-right block which has the highest value of all the cell block. When a cell is identified as the starting point of an alignment profile, the information regarding the starting call is copied to the host memory of the computer. The query sequence character in x, the target sequence character in y and score are copied, the direction or pointer from where the scores have come: the cell located at the left, upper-left or upper is also recorded, and these processes are executed in parallel. In the host (CPU) an array is used to store the starting cells, and the GPU maintains an integer index which points to the first free position. When a thread detects a starting cell, the index value is increased, and the host also manages a matrix to store the direction or pointers. The thread now finds the location where the score comes from a negative value in the matrix. The Smith–Waterman algorithm requires that all values be larger than or equal to zero. To avoid the allocation of additional memory, a negative sign is used to mark the traceback on the GPU. The process of each thread involves checking for the starting points, and also for negative values. When a negative value is found, the direction is copied to the host and the score will be marked as negative. At the end of processing each block, all negative values are copied to main memory for the neighbouring blocks to continue the traceback until the block in the upper-left corner is reached.

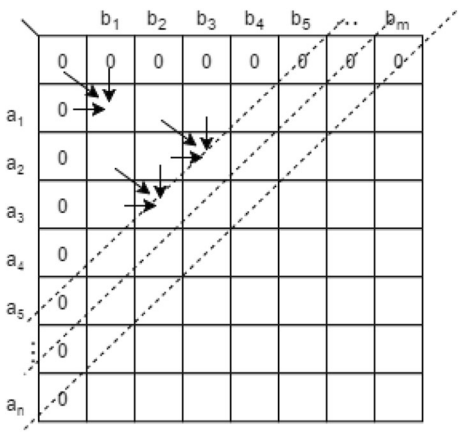

Fig. 8 Data dependency in matrix

The output produced in step two contains the locations in the direction matrix where to start the traceback. Using the directions, the matrix is processed and using the information the application goes through the matrix and counts the number of gaps, mismatches, and matches and creates the alignment profile with following information: the query sequence, the target sequence, score and the alignment profile.

3.2.1 Proposed method

Design of Smith–Waterman algorithm in GPU (shared memory), if a matrix can be fully loaded into the GPU shared memory, load the matrix and if not possible, split it into tiles, so that it fits in the shared memory, then calculate the tiles in parallel. This method does, (i) use segments/tile width to divide the full dynamic programming matrix into small tiles, (ii) copy the tile size subject sequence and query sequence to shared memory, (iii) calculate the matrix with the SW algorithm, (iv) traceback the matrix and (v) plot the aligned sequence.

The proposed method consists of three separate steps, (i) calculation of alignment scores, (ii) determination of tracebacks and (iii) producing the alignments as output. The GPU consist of global memory, shared memory, constant memory and Texture memory. Global memory is the main memory which is usually in few hundred MBs to few GBs. Shared memory is distributed across the GPU and located within the shared memory and is allocated 16, 32, or 48 KB. It is relatively small in size but can access

Table 1 Calculate score—analysis report

Duration	200.5 µs
Grid size	[2,36,1]
Block size	[32,32,1]
Registers/thread	22
Shared memory/block	8.316 KiB
Shared memory requested	48 KiB
Shared memory executed	48 KiB
Shared memory bank size	4 B

Table 2 Traceback—analysis report

Duration	98.105 µs
Grid size	[2,36,1]
Block size	[32,32,1]
Registers/thread	46
Shared memory/block	4.262 KiB
Shared memory requested	48 KiB
Shared memory executed	48 KiB
Shared memory bank size	4 B

Fig. 9 GPU utilization

faster than global memory. For this reason, intermediate results and scores matrix are stored in shared memory.

3.3 Step 1: calculate score

Each sequence alignment is subdivided into 32×32 matrices cells. These 32×32 matrices are mapped to thread blocks of 1024 threads. The characters of the two sequences, the matrix scores, and maximum values are stored in shared memory. When there are null scores from the neighbouring block, and if it is the first block, scores are initialized to zero and it's all calculated in parallel. Each block of matrix calculation starts at the top-left, and it passes through the matrix via the diagonal to the bottom-right. Similar to the matrix, each block depends on the surrounding blocks. Initially, $X \times Y$ blocks of 1024 threads are launched. Table 1 shows sample score report.

3.4 Step 2: traceback

This is opposite to the previous step. The process starts at the bottom-right block and the bottom right cell of the block. Using the maximum value set by the user, the starting point is identified. The GPU holds an index which will point to the first position. When a thread detects a starting point this index is increased by one by using an atomic function and guarantees that only this thread alone is accessing the value and creates a matrix to store the direction, this continues until the block in the upper-left corner is reached. Table 2 shows sample traceback report.

The kernel uses 46 registers for each thread (47,104 registers for each block), these registers prevent the kernel from fully utilizing the GPU. Device *GeForce GTX TITAN*

```
>sp|P12883|MYH7_HUMAN Myosin-7 OS=Homo sapiens GN=MYH7 PE=1 SV=5
MGDSEMAVFGAAAPYLRKSEKERLEAQTRPFDLKKDVFVPDDKQEFVKAKIVSREGGKVT
AETEYGKTVTVKEDQVMQQNPPKFDKIEDMAMLTFLHEPAVLYNLKDRYGSWMIYTYSGL
FCVTVNPYKWLPVYTPEVVAAYRGKKRSEAPPHIFSISDNAYQYMLTDRENQSILITGES
GAGKTVNTKRVIQYFAVIAAIGDRSKKDQSPGKGTLEDQIIQANPALEAFGNAKTVRNDN
SSRFGKFIRIHFGATGKLASADIETYLLEKSRVIFQLKAERDYHIFYQILSNKKPELLDM
LLITNNPYDYAFISQGETTVASIDDAEELMATDNAFDVLGFTSEEKNSMYKLTGAIMHFG
NMKFKLKQREEQAEPDGTEEADKSAYLMGLNSADLLKGLCHPRVKVGNEYVTKGQNVQQV
IYATGALAKAVYERMFNWMVTRINATLETKQPRQYFIGVLDIAGFEIFDFNSFEQLCINF
TNEKLQQFFNHHMFVLEQEEYKKEGIEWTFIDFGMDLQACIDLIEKPMGIMSILEEECMF
PKATDMTFKAKLFDNHLGKSANFQKPRNIKGKPEAHFSLIHYAGIVDYNIIGWLQKNKDP
LNETVVGLYQKSSLKLLSTLFANYAGADAPIEKGKGKAKKGSSFQTVSALHRENLNKLMT
NLRSTHPHFVRCIIPNETKSPGVMDNPLVMHQLRCNGVLEGIRICRKGFPNRILYGDFRQ
RYRILNPAAIPEGQFIDSRKGAEKLLSSLDIDHNQYKFGHTKVFFKAGLLGLLEEMRDER
LSRIITRIQAQSRGVLARMEYKKLLERRDSLLVIQWNIRAFMGVKNWPWMKLYFKIKPLL
KSAEREKEMASMKEEFTRLKEALEKSEARRKELEEKMVSLLQEKNDLQLQVQAEQDNLAD
AEERCDQLIKNKIQLEAKVKEMNERLEDEEEMNAELTAKKRKLEDECSELKRDIDDLELT
LAKVEKEKHATENKVKNLTEEMAGLDEIIAKLTKEKKALQEAHQQALDDLQAEEDKVNTL
TKAKVKLEQQVDDLEGSLEQEKKVRMDLERAKRKLEGDLKLTQESIMDLENDKQQLDERL
KKKDFELNALNARIEDEQALGSQLQKKLKELQARIEELEEELEAERTARAKVEKLRSDLS
RELEEISERLEEAGGATSVQIEMNKKREAEFQKMRRDLEEATLQHEATAAALRKKHADSV
AELGEQIDNLQRVKQKLEKEKSEFKLELDDVTSNMEQIIKAKANLEKMCRTLEDQMNEHR
SKAEETQRSVNDLTSQRAKLQTENGELSRQLDEKEALISQLTRGKLTYTQQLEDLKRQLE
EEVKAKNALAHALQSARHDCDLLREQYEEETEAKAELQRVLSKANSEVAQWRTKYETDAI
QRTEELEEAKKKLAQRLQEAEEAVEAVNAKCSSLEKTKHRLQNEIEDLMVDVERSNAAAA
ALDKKQRNFDKILAEWKQKYEESQSELESSQKEARSLSTELFKLKNAYEESLEHLETFKR
ENKNLQEEISDLTEQLGSSGKTIHELEKVRKQLEAEKMELQSALEEAEASLEHEEGKILR
AQLEFNQIKAEIERKLAEKDEEMEQAKRNHLRVVDSLQTSLDAETRSRNEALRVKKKMEG
DLNEMEIQLSHANRMAAEAQKQVKSLQSLLKDTQIQLDDAVRANDDLKENIAIVERRNNL
LQAELEELRAVVEQTERSRKLAEQELIETSERVQLLHSQNTSLINQKKKMDADLSQLQTE
VEEAVQECRNAEEKAKKAITDAAMMAEELKKEQDTSAHLERMKKNMEQTIKDLQHRLDEA
EQIALKGGKKQLQKLEARVRELENELEAEQKRNAESVKGMRKSERRIKELTYQTEEDRKN
LLRLQDLVDKLQLKVKAYKRQAEEAEEQANTNLSKFRKVQHELDEAEERADIAESQVNKL
RAKSRDIGTKGLNEE
```

Fig. 10 UniProt dataset

provides up to 65,536 registers for each block. Because the kernel uses 47,104 registers for each block, each shared memory is limited to one block (32 warps) which is simultaneously executing. Figure 9 shows sample GPU Utilization report.

Table 3 Sequential—alignment results

Number of sequence in DB	Max. length of subject seq.	Query length	Execution time (s)
50	9905	60	82.55
104	9744	240	1870.42
104	9744	420	4592.94

Table 4 OpenMP—alignment results

Number of sequence in DB	Max. length of subject seq.	Query length	Execution time (s)
50	9905	60	0.49
104	9744	240	4.26
104	9744	420	6.54
1472	9905	1801	192.20
2304	9905	1801	266.22
4982	9905	1441	366.05

Table 5 Performance evaluation of OpenMP, GPU using global and GPU using shared memory for varying query length

DB count	Subject seq. length	Query length	Open MP (s)	GPU (global) (s)	GPU (shared) (s)
2927	2998	300	172.8	61.557	22.133
		500	261.73	76.105	27.517
		1000	486.51	114.731	43.648
		2000	926.67	199.197	71.480
		2950	1367.96	279.987	99.517
2375	3998	300	177.58	64.828	23.917
		500	285.16	80.025	29.653
		1000	531.15	119.658	47.121
		2000	979.08	212.179	77.230
		3851		378.609	130.518
2376	4901	300	178.31	80.007	28.760
		500		99.246	36.279
		1000		145.754	57.047
		2000	991.81	257.314	94.792
		3851		449.588	159.164
2376	5532	300	191.75	88.042	32.541
		500	296.52	108.167	40.348
		1000	522.31	161.364	64.179
		2000	977.88	284.322	105.038
		3851		498.860	185.129

4 Results and analysis

4.1 Datasets used-protein sequence

The dataset that has been used for sequence alignment is *UniProt* dataset, which is a comprehensive resource for protein sequence and annotation data. It consists of 550,000 protein sequences, length used for alignment is 13,721 and query sequence length is 10,000. The number of sequences in the database is 19,837. Figure 10 shows the sample *UniProt* dataset. The Smith–Waterman algorithm is tested with different subject sequence length and query length. The results are tabulated in Table 3. As the length of the subject sequence and the query sequence increases, the execution time for the alignment also increases. Table 4 shows the alignment results when executed using OpenMP.

The result of the Smith–Waterman algorithm in OpenMP, GPU (global memory) and GPU (shared memory) for varying query length are shown in Table 5. It shows that the proposed method has higher performance gain than OpenMP and GPU using global memory. GPU using shared memory has sped up to 2.7% than GPU using global memory and 13.7% speed up to OpenMP. Figure 11

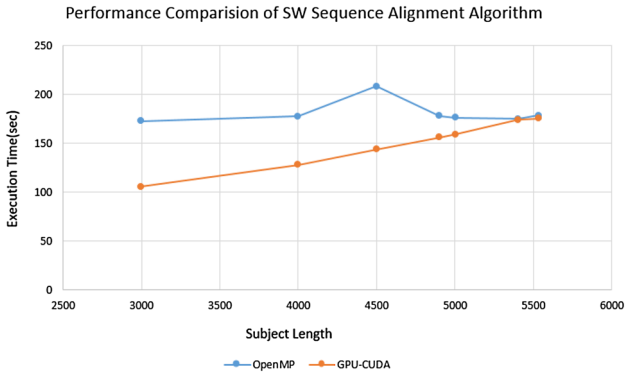

Fig. 11 Performance evaluation for subject length

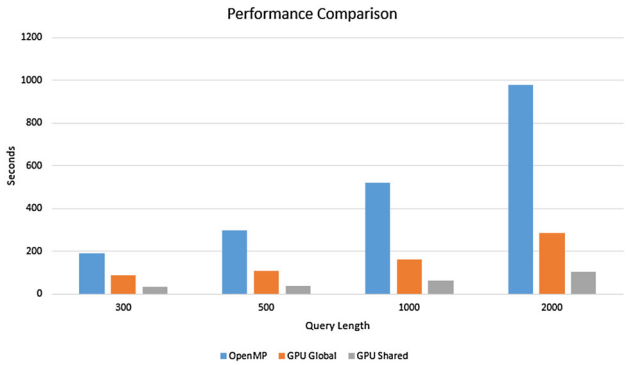

Fig. 12 Performance evaluation of CUDA with OpenMP, GPU using global and GPU using shared memory for varying query length

Table 6 Performance evaluation of CUDA for varying tile size

DB count	Sub. seq. length	Query length	GPU (global)	Shared-tile 32 × 32 (s)	Shared-tile 32 × 16 (s)
13,721	19,837	300	1748.76	643.42	558.80
		6758	16608.40	6252.25	5599.44
		10,000	23973.65	8595.39	7794.09

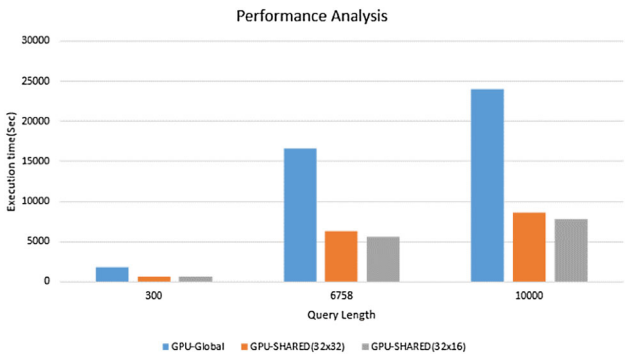

Fig. 13 Performance evaluation of GPU (global) and GPU (shared) for varying tile size

shows the results in the plot for subject length versus execution time for two methods, i.e., OpenMP and GPU (global).

The Alignment execution time of OpenMP, GPU (global memory) and GPU (shared memory) for query length of 300, 500, 1000 and 2000 are plotted in Fig. 12.

The proposed method is tested using varying tile size, i.e., 32 × 32 and 32 × 16 and the result is shown in Table 6. The result shows the Smith–Waterman implementation in GPU using shared memory with 32 × 16 has less execution time compared to OpenMP and GPU (global memory) method. The execution time of GPU (global memory) and GPU (shared memory) with different tile size are plotted in Fig. 13.

5 Conclusion

In this paper, we focused on studying the Smith–Waterman algorithm by implementing it in various methods like, Sequential, OpenMP, GPU using global memory and also proposed a method which used shared memory GPU. The proposed method achieves an improved performance gain when compared to other methods. It is observed a steep rise in performance of 2.7–13.0% with query length varying from 300 to 10,000. The maximum length of the subject sequence used is 13,700 and query length is 10,000 with database sequence count of 13,728. When adopting tiles of size 32 × 32 and 32 × 16 and Smith–Waterman performance displays much higher performance, especially for the tile-sized 32 × 16. The tabulated results clearly show that the performance of Smith–Waterman algorithms improves in proposed method and this indicates the importance of shared memory GPU. For future work, we plan to extend our work to find alignment for multiple query sequences instead of one query sequence.

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. **215**(3), 403–410 (1990)
2. Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res. **25**(17), 3389–3402 (1997)
3. Pearson, W.R., Lipman, D.J.: Improved tools for biological sequence comparison. Proc. Natl. Acad. Sci. **85**(8), 2444–2448 (1988)
4. Smith, A.D., Xuan, Z., Zhang, M.Q.: Using quality scores and longer reads improves accuracy of Solexa read mapping. BMC Bioinform. **9**(128), 1–8 (2008). https://doi.org/10.1186/1471-2105-9-128

5. Li, H., Ruan, J., Durbin, R.: Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res. **18**(11), 1851–1858 (2008)
6. Rumble, S.M., Lacroute, P., Dalca, A.V., Fiume, M., Sidow, A., Brudno, M.: SHRiMP: accurate mapping of short color-space reads. PLoS Comput. Biol. (2009). https://doi.org/10.1371/journal.pcbi.1000386
7. Li, R., Li, Y., Kristiansen, K., Wang, J.: SOAP: short oligonucleotide alignment program. Bioinformatics. **24**(5), 713–714 (2008)
8. Campagna, D., Albiero, A., Bilardi, A., Caniato, E., Forcato, C., Manavski, S., Vitulo, N., Valle, G.: PASS: a program to align short sequences. Bioinformatics. **25**(7), 967–968 (2009)
9. Zhou, Z.-M., Chen, Z.-W.: Dynamic programming for protein sequence alignment. Int. J. Bio-Sci. Bio-Technol. **5**(2), 141–150 (2013)
10. Bustamam, A., Ardaneswari, G., Lestari, D.: Implementation of Cuda GPU-based parallel computing on Smith–Waterman algorithm to sequence database searches. In: IEEE International Conference on Advanced Computer Science and Information Systems (ICACSIS), pp. 137–142 (2013)
11. Shukla, H., Shah, M.: Optimizing parallel scan Smith–Waterman algorithm on GPU. Int. J. Adv. Comput. Eng. Netw. **2**, 86–89 (2014)
12. Huang, L.-T., Wu, C.-C., Lai, L.-F., Li, Y.-J.: Improving the mapping of Smith–Waterman sequence database searches onto CUDA-enabled GPUs. BioMed Res. Int. (2015). https://doi.org/10.1155/2015/185179
13. Jain, C., Kumar, S.: Fine-grained GPU parallelization of pairwise local sequence alignment. In: IEEE International Conference on High Performance Computing (HiPC), pp. 1–10 (2014)
14. Liu, Y., Wirawan, A., Schmidt, B.: CUDASW ++ 3.0: accelerating Smith–Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinform. **14**(117), 1–10 (2013)
15. Khoirudin, Shun-Liang, J.: GPU application in CUDA memory. Int. J. Adv. Comput. **6**(2), 1–10 (2015). https://doi.org/10.5121/acij.2015.6201
16. Ghorpade, Jayashree, Parande, Jitendra, Kulkarni, Madhura, Bawaskar, Amit: GPGPU processing in CUDA architecture. Int. J. Adv. Comput. **3**(1), 105–120 (2012). https://doi.org/10.5121/acij.2012.3109
17. Ligowski, L., Rudnicki, W.: An efficient implementation of Smith–Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–8 (2009), ISSN:1530-2075
18. Pandey, J., Khare, N., Pandey, R.: A survey of parallel models for sequence alignment using Smith-Waterman algorithm. IOSR J. Comput. Eng. **17**, 48–52 (2015)
19. Chaibou, A., Sie, O.: Comparative study of the parallelization of the Smith-Waterman algorithm on OpenMP and Cuda C. J. Comput. Commun. **3**, 107–117 (2015)
20. Biradar, S., Desai, V., Madagouda, B., Patil, M.: Comparative analysis of dynamic programming algorithms to find similarity in gene sequences. Int. J. Res. Eng. Technol. **2**(8), 312–316 (2013)
21. El-Saghir, Z., Kelash, H., Elnazly, S., Faheem, H.: Parallel implementation of Smith–Waterman algorithm using MPI, OpenMP and hybrid model. Int. J. Innov. Technol. Explor. Eng. vol. 4, pp. 1–5 (2014). ISSN: 2278-3075
22. Huang, L.-T., Wu, C.-C., Lai, L.-F., Li, Y.-J.: Improving the mapping of Smith–Waterman sequence database searches onto CUDA-enabled GPUs. BioMed Res. Int. **2015**, 1–10 (2015)

D. Venkata Vara Prasad is a Professor in the Department of Computer Science and Engineering and has 20 years of teaching and research experience. He received his B.E. degree from University of Mysore, M.E. from the Andhra University, Visakhapatnam and Ph.D. from the Jawaharlal Nehru Technological University Anantapur. His Ph.D. work is on "Chip area minimization using interconnect length optimization". His area of research is computer Architecture, GPU Computing. He is a member of IEEE and also a Life member of CSI and ISTE. He has published a number of research papers in International and National Journals and conferences. He has published two text books: Computer Architecture, and Microprocessors and Microcontrollers. He is a Principle Investigator for SSN-nVIDIA GPU Education Center.

Suresh Jaganathan Associate Professor in the Department of Computer Science and Engineering has more than 20 years of teaching experience. He received his Ph.D. in Computer Science from Jawaharlal Nehru Technological University, Hyderabad, M.E. Software Engineering from Anna University and B.E. Computer Science and Engineering, from Mepco Schlenk Engineering College, Sivakasi, Madurai Kamarajar University, Madurai. He has more than 25 publications in International Journals and Conferences of which four are Refereed SCI Indexed journals. Apart from this, to his credit he filed 4 Patents and written a book on "Cloud Computing: A Practical Approach for Learning and Implementation", published by Pearson Publications. He is reviewer for refereed Elsevier Journal of Network and Computer Applications. His areas of interest are Distributed Computing, Multimedia Streaming, Big Data Analytics and Machine learning.