Fine-grained GPU parallelization of Pairwise Local Sequence Alignment

Chirag Jain [1], Subodh Kumar [2]

Department of Computer Science and Engineering
IIT Delhi
[1] cs1100215@cse.iitd.ac.in
[2] subodh@cse.iitd.ac.in

***Abstract*—The Smith-Waterman algorithm is used in Bio-informatics to perform pairwise local alignment between a query sequence and a subject sequence. We present a GPU based parallel version of this algorithm that is able to perform pair-wise alignment faster than previous algorithms. In particular, it parallelizes each alignment, rather than relying on parallelism across multiple pair alignments, which many other proposed GPU algorithms do. As a result it scales better. We further extend our algorithm to work efficiently on a cluster of GPUs. At a high level, our approach subdivides the iterative computation of elements of a matrix among blocks of processors such that each block can simply recompute the data it needs instead of waiting for other processors to compute them. Sometimes this may lead to excessive recomputation, however. We evaluate these cases and employ a hybrid approach, recomputing only limited data and communicating the rest. Our algorithm is also extended to produce not only the best but all 'best K' alignments. Our results on SSCA#1 benchmark show that our method is upto 5-24 times faster than previous method.**

I. Introduction

Comparing a pair of nucleotide or protein sequences and identifying regions of similarity allows biologists to discover functional, structural and evolutionary characteristics. This problem is commonly referred to as *sequence alignment* in Bio-informatics. Further, *global* alignment matches the complete sequence of characters. *Local* alignment on the other hand computes similarity of all sub-sequences, which may be well separated. In local alignment, non-identical characters and gaps are placed so as to align the identical characters of a *subject* and a *query* sequence (Fig. 1). Local alignment is computationally more challenging and often heuristics and approximation algorithms ([1], [2]) are applied. However, dynamic programming approaches like the Smith-Waterman algorithm (SW, for short) [3] are able to compute the best alignment based on a provided scoring scheme. If m is the length of the subject sequence and n is the length of the query, SW takes $O(mn)$ time. This quadratic time complexity is still prohibitive for large sequences and parallelization, including on GPUs, has been a popular area of research. Ours is an algorithm in this genre. First we describe the basic approach of Smith and Waterman.

A. Smith-Waterman algorithm

The local alignment algorithm of Smith and Waterman [3] uses dynamic programming to find the maximally similar segments from a given pair of sequences. It compares segments of all possible lengths and evaluates the similarity measure exhaustively for all options, before picking the best. This algorithm comprises three main steps:

1) Set up the dynamic programming matrix (i.e., *SW matrix*).
2) Iteratively compute the scores of the cells of the matrix.
3) Identify the optimal alignment through a final traceback.

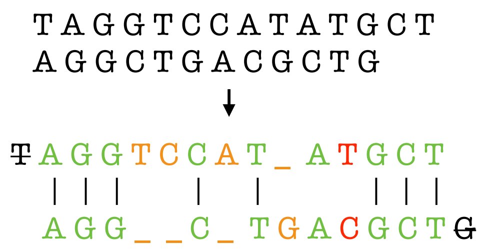

T A G G T C C A T A T G C T
A G G C T G A C G C T G
~~T~~ A G G T C C A T _ A T G C T
A G G _ _ C _ T G A C G C T ~~G~~

Fig. 1. A possible local alignment of two given sequences

We place the two given sequences respectively as the top row and the first column in the SW matrix. Let the score of cell (i, j) be denoted by $H(i, j)$. Let G_s, G_e and S_{ij} denote the gap opening-penalty, the gap extension-penalty and the similarity-score, respectively. Two buffers $E(i, j)$ and $F(i, j)$ are used to preserve the values from the previous iterations that are needed in the current one. The SW algorithm is defined by the following recurrences:

$$H(i,j) = \max \begin{cases} H(i-1, j-1) + S_{ij} \\ E(i,j) \\ F(i,j) \\ 0 \end{cases}$$

$$E(i,j) = \max \begin{cases} E(i-1, j) - G_e \\ H(i-1, j) - G_s - G_e \end{cases}$$

$$F(i,j) = \max \begin{cases} F(i, j-1) - G_e \\ H(i, j-1) - G_s - G_e \end{cases}$$

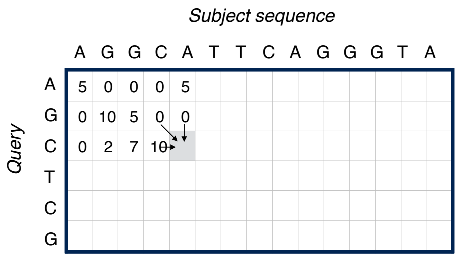


Fig. 2. SW Matrix Example

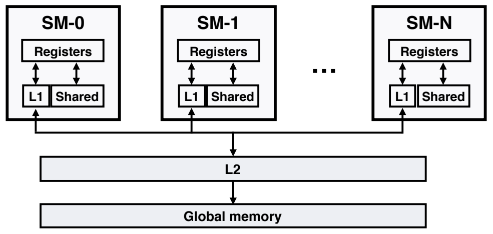


Fig. 3. Memory hierarchy on GPU with N+1 multiprocessors

The cells of the first row and the first column are initialized to zero. The similarity scores S_{ij}s are application dependent. In our experiments we have used the scores given by [4]: match score of 5, mismatch score of -3, gap opening-penalty of 8 and gap extension-penalty of 1. Fig. 2 shows an example of two sequences being aligned using this algorithm. Note that the computation corresponding to each cell (i, j) depends on the values in the cells above it in its column, to its left in its row and the top-left diagonals. After iteratively filling the matrix, a final traceback yields the resulting alignments. This traceback is a small fraction of the total alignment time, as also observed by Wirawan et al. [5]. Sandes and de Melo [6] also report 97.86% of time being spent in finding the optimal score and the end point of the optimal alignment while performing chromosome alignment on GPU GTX 285. Therefore, work in this paper focusses on optimizing the runtime for computing the SW matrix.

B. GPGPU memory architecture

Graphics Processing Units (GPUs) provide a general purpose processing platform as they offer compelling performance to cost ratio. GPU is well suited for many parallel compute-bound applications. It usually includes its own random access memory (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 SM (multi-processor) and can share data through fast on-chip *shared memory* and can be synchronized by barriers. Such synchronization is not available between the *blocks* [7], however.

Efficient coherent access of the on-chip and off-chip memory resources (Fig. 3) can significantly impact the observed memory throughput. Higher throughput can be critical to 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 100x as many clock cycles as the faster memory access and the memory bandwidth can be dramatically improved if threads within a SIMD *warp* access contiguous global memory addresses, resulting in fewer memory transactions. The *shared memory* is smaller but faster and can be used to communicate between threads within the same block. Shared memory bandwidth can also be affected due to bank conflicts. We demonstrate in Section II how our algorithm makes an efficient use of this memory hierarchy.

C. Previous Work

Many parallel versions of SW algorithm have been designed for clusters ([8], [9], [10], [11]), FPGAs ([12], [13], [14]), Grids ([15], [16]) and CellBEs ([17], [18], [19]). However, recent advent of GPUs has lead to a significant spurt in research in this area yielding impressive speedups ([20], [21], [22], [4], [23], [24], [25]). Many of these algorithms solve a variant of pairwise sequence alignment, where they align a query against multiple subject sequences in a database and therefore solve many independent SW problems using inter-task parallelization. These techniques do not present an optimal solution for alignments involving very large sequences, however. For example, it has been shown that CUDASW++ 2.0 cannot handle strings longer than 70,000 characters on NVIDIA Tesla C2050 [24].

Efficient alignment of large sequences is important in Bioinformatics because cross-species chromosome alignments can reveal ancestral relationships and may be used to identify the peculiarities of the species [6]. In this paper, we focus on improving the execution times of a single but very large alignment problem. The sizes of the subject sequence and query sequence in the typical biological applications are in the range $10^7 - 10^{10}$ and $10^4 - 10^5$, respectively [4]. Much relevant literature exists in this context as well ([4], [23], [24], [26], [6], [27]). Khajeh-Saeed et al. [4] reformulate the SW algorithm to use sequential memory accesses. This is achieved using a parallel scan approach to account for the horizontal dependency of each cell's score in the algorithm though it induces a significant computational and synchronization overhead. Siriwardena and Ranasinghe [23] propose two implementations for the global pairwise alignment with different memory access strategies but their runtime has been reported to be slower than the one in [4]. More recently, Li et al. [24] divide the SW matrix into many strips and compute them on individual SMs. CUDAlign 1 tool [26] is able to compare DNA sequences with more than 10^6 base pairs using a GPU version of the SW algorithm. CUDAlign 2 [6] extends

CUDAlign 1 to produce both the score and the full optimal alignment in linear space by using Myers-Miller algorithm. Their work is further optimized by Sandes et al. [27] by pruning the cells in the SW matrix that have no impact on the final optimal score. More recently, [28] reports a multi-GPU version.

Each of these algorithms require synchronization between blocks and have high I/O traffic between global memory and the SMs. In this paper, we propose a new technique to solve this problem, which not only obviates the issue of inter-block synchronization but also brings a significant reduction in the I/O traffic. Requisite attention to the detail that efficient utilization of the memory hierarchy plays an important role in the performance of memory-intensive applications on GPUs has helped us achieve better execution times for this problem on the target architecture.

We next describe two of the per-pair parallel alignment algorithms in more detail. Our approach is partly inspired by these algorithms.

1) ParallelScan method: The *ParallelScan* method of Khajeh-Saeed et al. [4] reformulates the SW algorithm to allow sequential memory accesses of SW matrix rows. This is achieved using parallel scan to account for the horizontal dependency of each cell in the algorithm. The parallel scan adds a significant computational overhead on the execution times (approximately 70% of the total time). There are two reasons for this behavior. First, the tree based reductions is a non-ideal match for GPU's architecture. Second, this approach requires inter-block synchronization. This can only be achieved by employing multiple sequential kernels computing each row of the score matrix.

If m is the size of the subject sequence, n that of the query sequence (with $m > n$) and p the number of processors, the algorithm fills the alignment matrix cells horizontally (parallel to the subject sequence) and works on p cells simultaneously. First step involves computing scores assuming only the vertical dependency of each cell and the second step involves resolving horizontal dependency (using Blelloch's scan algorithm). In the SW algorithm, the score at each cell depends on the $O(n)$ scores to its left because the sizes of aligned subsequences are limited due to the negative mismatch and gap penalties. These n elements can be scanned by n processors in $O(\log n)$ time. So for each row, time

$$r(m,n,p) = O\left(\frac{m}{p}\log(n)\right) \tag{1}$$

and therefore the total time

$$t(m,n,p) = O\left(\frac{mn\log(n)}{p}\right) \tag{2}$$

Note that it is the time required to complete $O(mn)$ work using p processors. Equation 2 shows the additional $log(n)$ factor introduced due to parallel scan computation. Executing multiple sequential GPU kernels on the GPU is expensive: they require $12n$ kernel calls. The implementation uses $O(m)$ global memory and an expensive $O(mn)$ I/O traffic between global memory and SMs. Their implementation assumed the size of optimal alignments to be shorter than 1024 as they resolve the horizontal dependency of a cell with 1024 cells only. This implementation returns incorrect results when size of alignments goes beyond 1024. *ParallelScan* method produces the best-K alignments.

2) StripedAlignment method: The *StripedAlignment* approach of Li et al. [24] is based on dividing the SW matrix into $\lceil m/s \rceil$ strips of constant width s and height n. SM i works on every strip j s.t. $j \mod sm = i$, where sm is the count of SMs on the GPU. Hence each strip needs some boundary scores from the previous one to commence its own work. The time taken by this strategy not only depends on the matrix size, but also the workload of each SM per strip because the small workload forces the SMs to wait for the data needed to start working on the next strip. Our experiments in Section III indicate that this factor leads to poor performance of this algorithm unless the query size is very large.

Further, this method performs inter-SM communication using global memory and each strip shares $O(n)$ data with its neighbor strip. So the total global I/O traffic is $O(\frac{m}{s} \times n)$. The results of their experiments prove that the I/O traffic introduces a significant overhead. If we exclude the high global memory access time however, this algorithm has optimal complexity of $O(mn/p)$ when both m and n are sufficiently large. Unlike *ParallelScan*, the SW matrix is computed through a single kernel invocation. This algorithm is designed to return only the single best alignment of the given pair of sequences.

The main contribution of this paper include:

- An efficient GPU based parallel algorithm to compute the SW matrix, which uses efficient memory operations, mostly in shared memory, and eliminates most synchronization
- An adaptatable algorithm that works efficiently on both small and large query sizes
- An extension to the algorithm that returns the best K scores

In the rest of this paper, we describe the details of our algorithm in Section II and evaluate it using experiments in Section III. Section IV concludes the paper and presents future work.

II. New Overlap-based approach

Both implementations discussed in the previous section require inter-block or inter-thread communication using shared or global memory because of dependency between cells. We have designed an *Overlap-based* approach, which attempts to eliminate the inter-block dependencies.

A. Parallelization strategy

The *Overlap-based* approach exploits two key observations of the SW algorithm to overcome the need for communication and synchronization. First, the scores of the cells along the diagonal of the score matrix can be computed in parallel (see Fig. 4). So at each time step, threads within a GPU block work on the cells that are diagonally placed in the matrix.

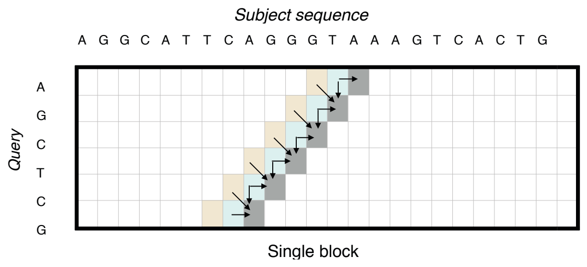


Fig. 4. The gray cells, computed by threads within a block, are only dependent on the scores of previous two diagonals. The successive diagonals are computed iteratively by the same block.

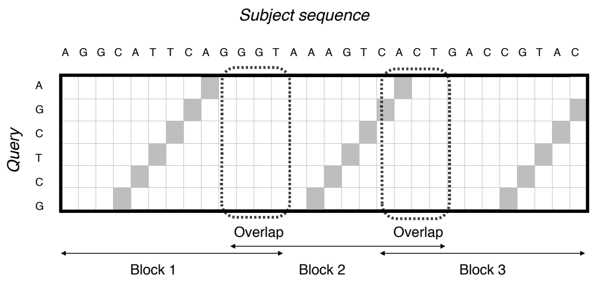


Fig. 5. Illustration of our parallelization strategy for query length $n < 1024$

Each diagonal depends upon the score of the previous two diagonal scores, and that is all we need to store in the shared memory. The threads of the block iteratively compute the scores of successive diagonals. Second, due to the negative gap penalty values, the lengths of the alignments are restricted to $O(\min(m, n))$, i.e., $O(n)$ in our case. We decompose the matrix into overlapping chunks of columns, with an overlap of $O(n)$ between successive chunks. Introducing the overlaps of size $O(n)$ allows us to assign each chunk to a separate GPU block. Threads of the block adopt the diagonal strategy and work without any synchronization or communication with other blocks. Any alignment spanning the cells in the overlapped region would fall at least in one of the overlapping chunks. Fig. 5 illustrates this, where gray cells represent the cells being computed in parallel. Each thread maintains its locally best score and its corresponding position for final aggregation later.

Each thread of a CUDA block computes the score of a single cell in the current diagonal being computed by this block. During the complete kernel execution, this thread is responsible for computing the complete row of the chunk that is assigned to the block. At the same time, there is an upper bound on the number of threads per block: 1024 on Kepler and Fermi architectures. Thus only upto 1024 cells-long diagonal may be computed simultaneously by threads of a block. For each chunk then, we handle a maximum of 1024 rows at each horizontal sweep and store the horizontal boundary values in the global memory (Fig. 6). These values are used while computing the next 1024 rows of the chunk. Please note that a single buffer of the size equal to the chunk's width is sufficient for this purpose. The interleaved thread decomposition ensures

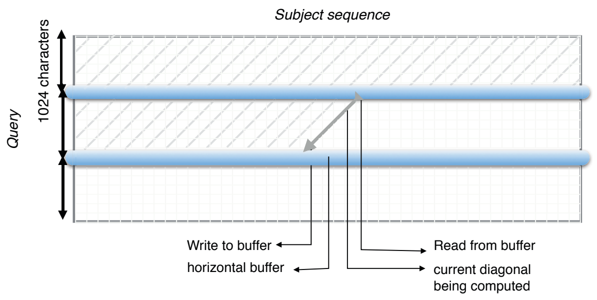


Fig. 6. Handling query length $n > 1024$ using a horizontal buffer in each block

that different threads of a warp access different shared memory banks, maximizing memory throughput. The overall benefits of adopting the *Overlap-based* scheme are:

- There is no need of synchronization or communication between different GPU blocks. The entire SW matrix can be filled using a single kernel invocation.
- This method makes better utilization of GPU memory resources. In each GPU block, the access to global memory for reading the subject sequence is maximally coalesced into few memory transactions. The intermediate score (H, E, F) buffers of size $(\min(n, 1024))$ per block fit in the fast shared memory.
- The I/O traffic between global memory and SMs is now reduced to $O(m + m \times \lfloor n/1024 \rfloor)$.

B. Complexity analysis

We start with a PRAM like analysis, which is useful to compute the optimal number of chunks into which the SW matrix should be decomposed. It also helps us evaluate the additional computational overhead incurred due to the overlaps. Please recall the notation introduced in Section I-C for the analysis below.

Let the length of the diagonal be $n' = min(n, 1024)$. Here we assume p to be a multiple of n' for simplicity, say $p = kn'$. The length of each overlap is known to be $O(n)$. If k' is the number of chunks into which we decompose the SW matrix, the number of cells computed per row of the matrix is $O(m + k'n)$, after double-counting the overlap. To have balanced load distribution among k' GPU blocks, the width of each chunk ought to be

$$w = O\left(\frac{m + k'n}{k'}\right) \tag{3}$$

If n' processors are assigned to each chunk, the number of time steps t' required for scoring one chunk diagonal-wise would be $O((w + n') \times \lceil n/1024 \rceil)$, i.e.,

$$t' = O\left(\frac{m + k'n}{k'} \times \left\lceil \frac{n}{1024} \right\rceil\right) \tag{4}$$

Since there are only $p = kn'$ processors, by work scheduling principle the actual time is:

$$\begin{aligned} t &= O\left(t' \times \frac{k'}{k}\right) \\ &= O\left(\left(\frac{m+k'n}{k'} \times \left\lceil \frac{n}{1024} \right\rceil\right) \times \frac{k'}{k}\right) \\ &= O\left(\frac{(m+k'n)(n)}{p}\right) \end{aligned} \tag{5}$$

The main point of equation 5 is to show that the total time t is positively correlated with the number of chunks k', suggesting a small k'. At the same time, we have n' processors working on each chunk, and hence $k' = p/n'$ in order to utilize all the processors. If we plug this value back into equation 5 and simplify,

$$t = O\left(\frac{\left(m + p\left\lceil \frac{n}{1024} \right\rceil\right)(n)}{p}\right) \tag{6}$$

Now, this algorithm for computing the best alignment is well-defined on a PRAM model:

> With p processors, divide the whole SW matrix into $k' = p/n'$ chunks and let n' processors work on each chunk.

GPUs, however, hide the actual number of processors behind many more logical threads. Using the actual number of hardware GPU cores as p in the above analysis will leave a GPU with a very low occupancy. We instead empirically determine the effective number of useful threads for different input sizes. By experimenting with different count of threads on multiple input sizes using NVIDIA Tesla K20M, which has 13 SMs, we find that having 13×2048 threads yields the best or close to the best timings by ensuring 100% GPU occupancy. Considering that the length of the query sequence, n, is much smaller than that of the subject sequence, m, even if $m > 10^2 \times n$, Equation 6 reduces to $t = O(mn/p)$ implying that the additional computational overhead due to the overlaps is not significant. The experimental evaluation of this algorithm against the existing methods is done in Section III.

C. Best-K alignments

Biologists are often interested in discovering the best-K alignments for a more comprehensive analysis of the match possibilities. That is why the BLAST tool hosted by NCBI [29] displays the 100 best aligned sequences by default. To support such queries, we use a parallel priority queue data structure per block that maintains the best K scores and their locations. The queue is implemented using a sorted array. Updates to the queue are made after each time step of the single diagonal computation of the block. Note that the score of a cell needs to be registered in the priority queue only when

- the score is greater than the minimum value in the priority queue.
- this cell involves a match: an optimal local alignment cannot have a gap or mismatch at the ends.

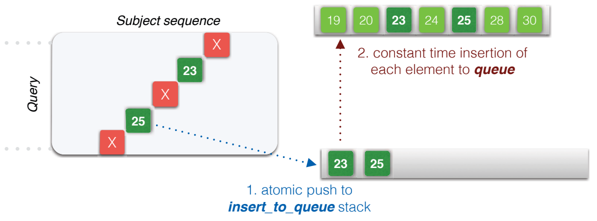

Subject sequence
Query
X
23
X
25
X
19
20
23
24
25
28
30
2. constant time insertion of each element to queue
23
25
1. atomic push to insert_to_queue stack

Fig. 7. Insertion of two scores 23 and 25 to priority queue at a particular iteration

The rest of the scores are ignored. The steps to insert the new "high" scores to the priority queue of size at most K after each diagonal computation are:

1) Each thread of the block which has a score higher than the current minimum score in the priority queue makes an atomic push to *insert_to_queue* stack (Fig. 7).
2) Barrier-synchronize the threads of the block.
3) Iterating for each element in *insert_to_queue* stack:

   > K threads parallely determine its index in *PriorityQueue* and if in best-K, insert the element in in $O(1)$ time.

   The insertion of v elements in the queue at a particular iteration takes $O(v)$ time.

The storage buffers required above are allocated in the shared memory. But still this is not a work-efficient algorithm. In the worst case, this method can increase the runtime by $O(n)$ times compared to the best-1 version if many cells report successively better scores. However, in practice we find that the actual number of updates to the priority queue are quite low after a few initial iterations. The average count of elements inserted per iteration in the first 100 iterations is close to 1 and it further reduces to about 0.06 if we consider the first 10^4 iterations (See table I). This happens because the minimum value in the priority queue continues to increase during the execution and therefore the count of updates to the queue continues to decline. After all blocks report their best-K scores as sorted arrays, the final best-K scores are aggregated on the CPU instead of a GPU because even sequential merge takes an insignificant fraction of the time (less than a millisecond in our experiments). We compare both the best-1 and best-K implementations of *Overlap-based* scheme in Section III.

TABLE I
TOTAL COUNT OF HIGH SCORES IN FIRST BLOCK DURING FIRST I ITERATIONS

I	Count of high scores		
	$m = 10^6$ $n = 10^3$	$m = 10^8$ $n = 10^2$	$m = 10^8$ $n = 10^3$
10^2	112	122	122
10^3	422	299	410
10^4	605	440	604

TABLE II
PERFORMANCE EVALUATION OF OVERLAP-BASED METHOD ON TESLA K20M

Subject sequence (m)	Query sequence (n)	Timings (ms)			Speedups	
		Overlap-based (t_1)	ParallelScan(t_2)	StripedAlignment(t_3)	t_2/t_1	t_3/t_1
10^5	128	3.4	50.9	1189.0	14.97	349.71
10^6		15.1	165.3	11788.1	10.95	780.67
10^7		133.0	1217.2	117519.5	9.15	883.61
10^8		1018.2	*	1172986.2	-	1152.02
10^5	1024	12.7	301.7	1323.4	23.76	104.20
10^6		87.1	1039.8	13084.1	11.94	149.58
10^7		754.0	8295.1	129855.5	11.00	172.22
10^8		7359.1	*	**	-	-
10^5	10240	411.4	2617.1	2348.6	6.36	5.71
10^6		1411.7	9599.1	22343.3	6.80	15.83
10^7		11488.7	80962.2	**	7.05	-
10^8		112303.7	*	**	-	-
$2 * 10^5$	100352	29358.1	32745.8	42488.0	1.12	1.45
10^6		38102.0	93799.9	**	2.46	-
10^7		136037.5	789906.0	**	5.81	-

* Out of memory
** CUDA launch error

TABLE III
EVALUATION OF OVERHEAD OF OVERLAPS IN OVERLAP-BASED METHOD FOR LONG QUERY LENGTHS

Subject sequence (m)	Query sequence (n)	Timings (ms)		Slowdown factor due to overlaps
		Overlap-based (t_1)	Without overlaps(t_2)	t_1/t_2
10^5	10240	411.4	133.5	3.08
10^6		1411.7	1150.2	1.23
10^7		11488.7	11305.1	1.02
10^8		112303.7	112922.5	0.99
$2 * 10^5$	50176	8101.4	1194.1	6.78
10^6		12506.8	5617.6	2.23
10^7		62124.2	55330.0	1.12
$2 * 10^5$	100352	29358.1	2400.0	12.23
10^6		38102.0	11214.9	3.40
10^7		136037.5	110278.6	1.23

TABLE IV
REDUCTION IN OVERHEAD OF OVERLAPS USING MODIFIED OVERLAP-BASED METHOD FOR LONG QUERY LENGTHS

Subject sequence (m)	Query sequence (n)	Timings (ms)	Slowdown factor due to overlaps
		Modified Overlap-based (t_3)	$t_3/t(Without\, overlaps)$
10^5	10240	178.6	1.34
10^6		1534.4	1.33
10^7		12365.0	1.09
10^8		120690.2	1.07
$2 * 10^5$	50176	1597.2	1.34
10^6		7467.3	1.33
10^7		66379.8	1.20
$2 * 10^5$	100352	3179.5	1.32
10^6		14925.9	1.33
10^7		147291.4	1.34

D. Large query length

If the length of the query sequence is comparable to that of the subject sequence ($m \approx n$), the *Overlap-based* method discussed above fails to achieve optimal time. The overhead of overlaps in this case dominates the overall execution time of the complete algorithm. In the worst case, the optimal sequence alignment can span across most of the SW matrix. Therefore, we propose a modification to our *Overlap-based* approach to perform well on average cases without significant impact on the worst-case performance.

Recall that in the *Overlap-based* method, to remove the inter-block synchronization overhead, we used the overlap size of $O(n)$. If $m \approx n$, we reduce the size of the overlap to a small value r and restore inter-block synchronization, although at lower frequency, to ensure correctness. The overlap size r is kept to be $min(n, m/100)$ so that the overhead of overlaps doesn't dominate (argued in section II-B). The steps we follow are as follows:

TABLE V
PERFORMANCE EVALUATION OF MODIFIED OVERLAP-BASED METHOD ON TESLA K20M

Subject sequence (m)	**Query sequence (n)**	**Timings (ms)**			**Speedups**	
		Overlap-based (t_1)	ParallelScan(t_2)	StripedAlignment(t_3)	t_2/t_1	t_3/t_1
10^5	128	3.4	50.9	1189.0	14.97	349.71
10^6		15.1	165.3	11788.1	10.95	780.67
10^7		133.0	1217.2	117519.5	9.15	883.61
10^8		1018.2	*	1172986.2	-	1152.02
10^5	1024	12.7	301.7	1323.4	23.76	104.20
10^6		87.1	1039.8	13084.1	11.94	149.58
10^7		754.0	8295.1	129855.5	11.00	172.22
10^8		7359.1	*	**	-	-
10^5	10240	178.6	2617.1	2348.6	14.65	15.59
10^6		1534.4	9599.1	22343.3	6.26	18.12
10^7		12365.0	80962.2	**	6.55	-
10^8		120690.2	*	**	-	-
$2 * 10^5$	100352	3183.0	32745.8	42488.0	10.29	16.37
10^6		14959.3	93799.9	**	6.27	-
10^7		146096.4	789906.0	**	5.41	-

[*] Out of memory
[**] CUDA launch error

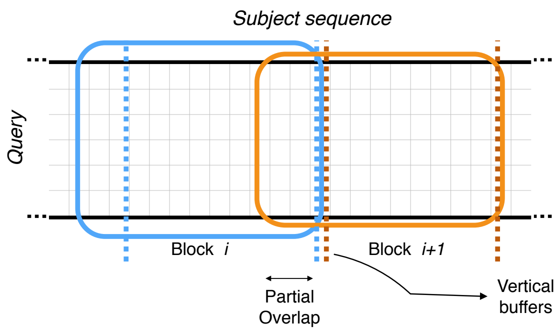


Fig. 9. Modified Overlap-based approach when $m \approx n$

1) Invoke the kernel with a partial overlap size r and let each block save its intermediate scores (E and H scores) corresponding to the rth and the last column of its chunk to the global memory (Fig. 9) using a coalesced write access.
2) After the kernel execution, the scores of the rth column of each block are compared with the scores of last column of the previous block on the CPU. An exact match of scores for any block with its neighbor implies that its precomputed scores are correct. (In other words, the match does not extend beyond the overlap.) Otherwise it recomputes its scores after receiving the intermediate scores from the previous block, again through a coalesced read from the global memory.
3) Algorithm stops when no blocks require recomputation.

We have evaluated the improvement of this *Modified Overlap-based* method over the earlier *Overlap-based* method with large query lengths in section III.

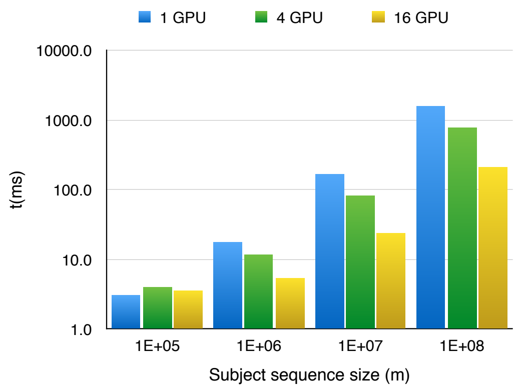
E. Scaling to multiple GPUs

As the blocks working on different chunks hardly need inter-block synchronization during an iteration, scaling the *Modified Overlap-based* method beyond a single GPU is straightforward by having overlapping chunks distributed over multiple GPUs on multiple nodes. Now the communication of the boundary values is done over MPI. Also, the optimal scores and locations produced by each device can be aggregated on the master node. We evaluate the performance of this method on up to 16 GPU devices in section III.

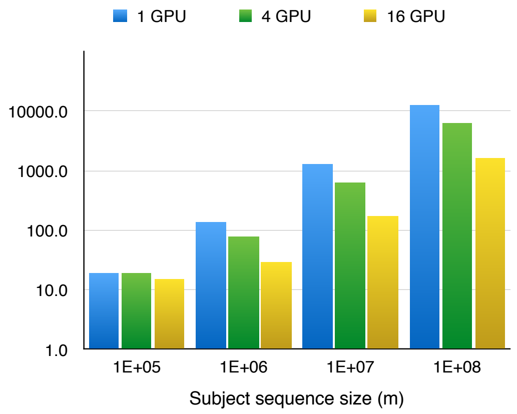
III. RESULTS AND DISCUSSION

In this paper, the timings being reported cover the complete execution time of application from reading the sequences until alignments are saved in the host memory. SSCA#1 benchmark [30] represents a Bio-informatics problem that involves performing a local pairwise alignment of two synthetic long codon sequences, finding the end-points of subsequences that are good matches according to the specified criteria. This is followed by identifying actual codon sequences already located. We perform this final traceback using a naive serial implementation on CPU, although efficient backtracking GPU algorithms also exist ([6], [31]). The time spent in traceback is inconsequential to the overall execution time of this problem. Moreover, this benchmark tends to produce relatively small alignments (less than 64 units long). The tests on a single GPU are carried out on a Tesla K20M graphics card with 13 SMs, 192 CUDA cores per SM and 5GB RAM. It is installed on a PC with Intel Sandybridge 2.60 GHz processor running the linux OS.

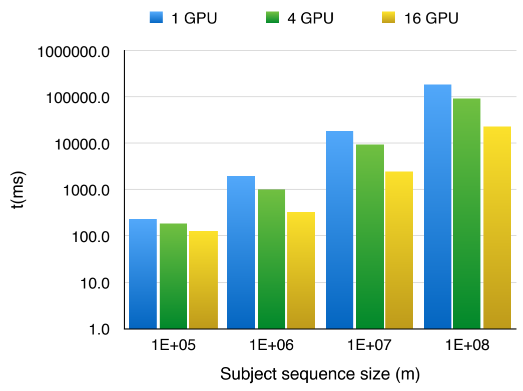
The timings of our *Overlap-based* method are compared to both *ParallelScan* and *StripedAlignment* methods on different values of m and n when the single best alignment is expected (Table II). The *Overlap-based* method achieves a speedup of at least 5 times if the query size is less than 10^4. However, the computational overhead due to the overlaps starts dominating when we choose the query size to be 10^5 or longer. The *StripedAlignment* is executed with an optimal width $s = 400$. As argued before, it exhibits good performance only when



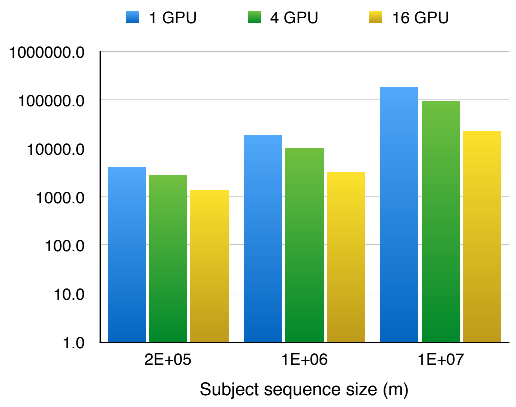
(a) Query size: 128



(b) Query size: 1024



(c) Query size: 10240



(d) Query size: 100352

Fig. 8. Time taken by *Modified Overlap-based* method on different number of GPUs.

query lengths are much longer than s. We also find that this method runs out of memory for very large input sizes as it makes use of global memory for inter-SM communication.

In order to find the empirical value of the computational overhead due to overlaps on large query lengths, we compare the timings of the standard *Overlap-based* method to that obtained by setting the overlap length to 0. Of course, this change yields incorrect results as no synchronization is actually performed but it has an idealized best-case performance. It helps us understand the slowdown factor solely due to the presence of overlaps for different values of m and n (Table III). Results corroborate our argument for the poor performance of the *Overlap-based* method when query length is large and close to m. However, the slowdown factor improves to just over 1.3 if we use the *Modified Overlap-based* method for handling long queries (Table IV).

The experiments shown in Table II are re-evaluated using the *Modified Overlap-based* method and the new speedup values are shown in Table V. Due to the absence of a significant overlap overhead, this method maintains a speedup factor of at least 5 times in every row. Next, we compare the *best-1* and *best-K* implementation of *Overlap-based* method which return the best 1 and the best K alignments, respectively (Table VI). We observe a slowdown factor of about two on different input sizes because of the priority queue operations. These timings when compared against the *best-k* implementation of *ParallelScan* are still faster by atleast 3 times.

TABLE VI
ESTIMATING OVERHEAD OF MAINTAING PRIORITY QUEUE FOR RETURNING BEST K ALIGNMENTS (K=100)

m	**n**	**Timings (ms)**		**Slowdown**
		Best 1(t_1)	Best K(t_2)	t_2/t_1
10^6	1024	80.5	166.4	2.1
10^7		737.4	1255.1	1.7
10^8		7406.0	12368.5	1.7
10^6	10240	1427.4	2812.7	2.0
10^7		11592.0	21495.5	1.9
10^8		113305.4	208235.9	1.8

We have also performed our tests on a cluster of nodes connected via Ethernet, each containing an Intel Xeon CPU clocked at 2.67GHz and two NVIDIA Fermi GPUs M2070 to find the multi-GPU performance of the *Modified Overlap-based* algorithm. The results in Figure 8 indicate that this

TABLE VII
RESULTS OF MODIFIED OVERLAP-BASED METHOD ON DROSOPHILA MELANOG CHROMOSOME

m	n	Iterations	Timings (ms)
10^5		2	69.3
10^6	2048	1	267.5
10^7		1	2400.4
10^5		7	1127.4
10^6	10240	2	3007.0
10^7		1	12488.9

algorithm can gain 2x and 8x speedup with 4 GPUs and 16 GPUs, respectively.

The performance of the *Modified Overlap-based* algorithm also depends on the similarity of the two sequences being considered: the higher the similarity (size of optimal match), the more the recomputation. Alignment size on SSCA benchmark is not large enough to require recomputations. So, we also perform single GPU experiments involving real nucleotide sequences. For this, we picked subsequences of *Drosophila melanog* chromosome 3L (accession no. NT_037436.3) and *Drosophila melanog* chromosome 2L (accession no. NT_033779.4) as our subject and query sequence respectively. The timings and the count of times recomputation happens has been shown in Table VII. Traceback on CPU is ignored in this case. We see that the count of iterations decreases with increasing value of m/n due to the increasing overlap size.

IV. CONCLUSION AND FUTURE WORK

We have presented a technique to parallelize a dynamic-programming problem that allows the computation at different processors to overlap. In particular, 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. It achieves a speedup of at least five times on subject sequence of sizes up to 100 million. Our method also either reduces or entirely eliminates the inter-block synchronization on the GPU and makes efficient use of its memory resources. The use of a parallel priority queue while computing scores also allows us to obtain the best K alignments. We have presented results for $K = 100$ and see a slowdown factor of only about 2. This method can handle large sequence sizes comparable to the size of genomes and easily scales to multiple GPUs using MPI due to reduced communication among blocks. This work also opens interesting perspectives as similar strategies of acceleration could be applied to more general dynamic programming problems. This remains an area of future research.

As future work, we intend to further optimise the recomputation phase of the *Modified Overlap-based* method. In this version of the algorithm, we recompute the complete blocks where boundaries mismatch. We can reduce the magnitude of recomputation by concentrating on the regions which can actually affect the final result. Secondly, we also wish to rebalance the load among GPU SMs as the count of blocks active at a particular iteration can decrease to very few depending on the input sequences.

REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[2] W. R. Pearson, "[5] rapid and sensitive sequence comparison with FASTP and FASTA," *Methods in enzymology*, vol. 183, pp. 63–98, 1990.

[3] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[4] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the smith-waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, vol. 229, no. 11, pp. 4247–4258, 2010.

[5] A. Wirawan, C. K. Kwoh, and B. Schmidt, "Multi-threaded vectorized distance matrix computation on the CELL/BE and x86/SSE2 architectures," *Bioinformatics*, vol. 26, no. 10, pp. 1368–1369, 2010.

[6] E. F. de O Sandes and A. C. M. A. de Melo, "Smith-waterman alignment of huge sequences with GPU in linear space," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 1199–1211, IEEE, 2011.

[7] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, March 2014.

[8] S. Aluru, N. Futamura, and K. Mehrotra, "Parallel biological sequence comparison using prefix computations," *Journal of Parallel and Distributed Computing*, vol. 63, no. 3, pp. 264–272, 2003.

[9] C. Cehn and B. Schmidt, "Computing large-scale alignments on a multi-cluster," in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pp. 38–45, IEEE, 2003.

[10] A. M. Aji and W.-c. Feng, "Optimizing performance, cost, and sensitivity in pairwise sequence search on a cluster of playstations," in *BioInformatics and BioEngineering, 2008. BIBE 2008. 8th IEEE International Conference on*, pp. 1–6, IEEE, 2008.

[11] R. B. Batista, A. Boukerche, and A. C. M. A. de Melo, "A parallel strategy for biological sequence alignment in restricted memory space," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 548–561, 2008.

[12] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform," in *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, pp. 39–48, ACM, 2007.

[13] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun, "A reconfigurable accelerator for smith–waterman algorithm," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 54, no. 12, pp. 1077–1081, 2007.

[14] I. T. Li, W. Shum, and K. Truong, "160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (FPGA)," *BMC bioinformatics*, vol. 8, no. 1, p. 185, 2007.

[15] C. Chen and B. Schmidt, "An adaptive grid implementation of DNA sequence alignment," *Future Generation Computer Systems*, vol. 21, no. 7, pp. 988–1003, 2005.

[16] A. YarKhan and J. J. Dongarra, "Biological sequence alignment on the computational grid using the grads framework," *Future Generation Computer Systems*, vol. 21, no. 6, pp. 980–986, 2005.

[17] F. Sánchez, F. Cabarcas, A. Ramirez, and M. Valero, "Long DNA sequence comparison on multicore architectures," in *Euro-Par 2010-Parallel Processing*, pp. 247–259, Springer, 2010.

[18] A. Sarje and S. Aluru, "Parallel genomic alignments on the cell broadband engine," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 11, pp. 1600–1610, 2009.

[19] W. R. Rudnicki, A. Jankowski, A. Modzelewski, A. Piotrowski, and A. Zadrożny, "The new SIMD implementation of the smith-waterman algorithm on cell microprocessor," *Fundamenta Informaticae*, vol. 96, no. 1, pp. 181–194, 2009.

[20] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC bioinformatics*, vol. 9, p. S10, 2008.

[21] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: Optimizing smith-waterman sequence database searches for CUDA-enabled graphics processing units," *BMC research notes*, vol. 2, no. 1, p. 73, 2009.

[22] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++2.0: enhanced smith-waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualised SIMD abstractions," *BMC research notes*, vol. 3, no. 1, p. 93, 2010.

[23] T. Siriwardena and D. Ranasinghe, "Accelerating global sequence alignment using CUDA compatible multi-core GPU," *In Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on*, pp. 201–206, 2010.

[24] J. Li, S. Ranka, and S. Sahni, "Pairwise sequence alignment for very long sequences on GPUs," *IEEE International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, 2012.

[25] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++3.0: accelerating smith-waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC bioinformatics*, vol. 14, no. 1, p. 117, 2013.

[26] E. F. O. Sandes and A. C. de Melo, "Cudalign: using GPU to accelerate the comparison of megabase genomic sequences," in *ACM Sigplan Notices*, vol. 45, pp. 137–146, ACM, 2010.

[27] S. de O, E. Flavius, and A. C. de Melo, "Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 5, pp. 1009–1021, 2013.

[28] E. F. de O. Sandes, G. Miranda, A. C. Melo, X. Martorell, and E. Ayguade, "Fine-grain parallel megabase sequence comparison with multiple heterogeneous GPUs," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, (New York, NY, USA), pp. 383–384, ACM, 2014.

[29] T. Madden, "Chapter 16. the blast sequence analysis tool. the NCBI handbook; 2002," 2012.

[30] D. A. Bader *et al.*, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *CTWatch Quarterly*, vol. 2, no. 4B, 2006.

[31] J. Blazewicz, W. Frohmberg, M. Kierzynka, E. Pesch, and P. Wojciechowski, "Protein alignment algorithms with an efficient backtracking routine on multiple GPUs," *BMC bioinformatics*, vol. 12, no. 1, p. 181, 2011.