Analysis of Resource Utilization on GPU

M.R. Pimple[1], S.R. Sathe[2]
Department of Computer Science, Visvesvaraya National Institute of Technology, Nagpur, India

***Abstract*—The problems arising due to massive data storage and data analysis can be handled by recent technologies, like cloud computing and parallel computing. MapReduce, MPI, CUDA, OpenMP, OpenCL are some of the widely available tools and techniques that use multithreading approach. However, it is a challenging task to use these technologies effectively to handle the compute intensive problems in the fields like life science, environment, fluid dynamics, image processing, etc. In this paper, we have used many core platforms with graphics processing units (GPU) to implement one of very important and fundamental problem of sequence alignment in the field of bioinformatics. Dynamic and concurrent kernel features offered by graphics card are used to speed up the performance. With these features, we achieved a speed up of around 120X and 55X. We have coupled well-known tiling technique with these features and observed a performance improvement up to 4X and 2X, as compared to non-tiling execution. The paper also analyses resource parameters, GPU occupancy and proposes their relationship with the design parameters for the chosen algorithm. These observations have been quantified and the relationship between the parameters is presented. The results of study can be extended further to study similar algorithms in this area.**

Keywords—Dynamic kernel; GPU; Multithreading; occupancy; parallel computing

I. Introduction

Graphics hardware along with multi-core system has emerged as a new combination for the applications that has computationally demanding tasks to be performed. The conventional graphic processors are now being used in various application domains including general purpose processing. Compute Unified Device Architecture (CUDA) provides tools to exploit resources on graphics processing units (GPU). With the help of this tool, it has become possible to handle compute intensive applications by invoking hundreds of parallel threads performing the task. However, in order to achieve performance improvement, it is essential to understand the architecture of the hardware, its limitations. Algorithms need to be restructured according to the underlying hardware in order to achieve speed up.

The main aim of this paper is to study and analyse the huge computational power offered by the graphics processors and utilize it to enhance the performance of a well-known problem of pair-wise sequence alignment. The paper discusses the parallelization of sequence alignment problem on many core platforms. The algorithm deals with finding the similarities between two or more biological sequences [DNA/protein]. The functional and structural relationships between two or more biological sequences can be found out by sequence alignment methods like local & global alignment. The similarity index can be used to explore the evolutionary relationship between the sequences. Needleman-Wunch [NW] [1] algorithm for global alignment and Smith Waterman [SW] [2] algorithm for local alignment are two widely used approaches based on dynamic programming [DP] method. The algorithm generates a "score matrix" to track the similarities between two sequences. It has three-fold data dependencies in north, west & northwest directions for every element of the matrix. As the size of the database increases, the searching time increases exponentially. Hence, the other approach is to use heuristic methods, such as FASTA and BLAST. Heuristic methods are faster than DP approach, but do not always guarantee the correctness of results. Dynamic programming method is preferred over heuristic approach for generating accurate results. With the availability of huge and ever increasing datasets, the serial CPU implementation by any method takes very large time to produce the results, even with the faster machines. Hence, over the past few years, the focus has been towards parallel implementation of the problem. With the availability of highly parallel programming platforms, like many and multi core machines, it has become possible to effectively use them to accelerate the performance of data parallel applications.

Due to the large volume of data and heavy data dependencies in the alignment problem, it is very difficult to apply it directly on the parallel platform. Hence, for parallel implementation, it is necessary to resolve these dependencies and then utilize the power of thousands of cores supported by the graphics card (GPUs).

In this paper, we have presented a method for generating score matrix for pair wise local sequence alignment problem using tilling technique. This method is coupled with the features like dynamic and concurrent kernel execution supported by the GPU card. The paper also presents the relationship of various design parameters with the resource parameters for improving the performance. The approach can easily be applied to the algorithms like global sequence alignment and multiple sequence alignment.

II. Related Work

Various strategies have been proposed in the literature to apply parallel computing methodology for sequence alignment problem. The basic biological information about any species is represented in the form of sequences like DNA, and protein. The sequence of unknown species or the sequence under investigation is compared with the known sequences from the standard sequence repository. The result of the comparison shows, the analogy or the differences between them. For pair wise sequence alignment method, two strategies are mainly used by the researchers.

- Algorithms that are based on dynamic programming methodology giving accurate results but taking exponential time to produce the output. For example, Needleman and Wunsch [NW] [1], Smith and Waterman [SW], [2] proposed the algorithm for global and sequence alignment, respectively.
- Heuristic approaches that are less accurate in finding the best possible alignment but are faster and widely used. For example, technique like FASTA & BLAST proposed by Wiber & Lipman [3], and later by Pearson & Lipman [4] is very popular.

Complexity of the alignment algorithm is directly proportional to the number of sequences and length of each sequence (e.g. O(nm) for 2 sequences of length n & m) With the availability of huge data for analysis, it is really challenging for the researchers to process the data and return the results within reasonable time period, so that biologists can infer the results quickly and carry out further analysis. With sequential algorithm, it takes many hours or even days to produce correct results especially for large number of longer sequences. Hence, researchers have used accelerators to speed up the compute intensive part of the algorithm. Because of the heavy data dependency, divergence code flow, and non-coalesced memory access it is very difficult to parallelize the sequence alignment algorithm and map it directly onto the processing platform. However, researchers have implemented the algorithm using various strategies and hardware accelerators.

Field Programmable Gate Array (FPGA) and GPUs are the commonly used hardware accelerators for improving the execution time. Performance study of three applications on an FPGA & GPU is presented in [5]. Authors have studied Gaussian Elimination, Data Encryption, and Needleman-Wunch algorithm. The factors like, overall hardware features, application performance, programmability, overhead are considered for mapping applications onto various accelerators.

A space efficient global sequence alignment algorithm is presented by Scott Lloyd and Quinn O'Snell [6]. Authors presented the performance improvement in forward scan and trace back in hardware, without memory and I/o limitations. Parallel implementation of sequence alignment problem was also studied for clustering system [7] using message passing interface [MPI] technique. The authors have discussed major models like pipeline model and anti-diagonal model for parallel implementation of the dynamic programming algorithm. Gotoh [8] has proposed an improved version of SW algorithm with an affine penalty function. Algorithm proposed by Khajej-Saeed, Poole, and Perot [9] enhances the parallelism by reconstructing the recurrence relations for multiple GPUs. Implentation of SW algorithm on GPU is presneted by Lukas Ligowski, and Witold Rudnicki [10] on NVIDIA GPU platform. The paper presents the performance improvement by effiicient use of shared memory on graphics card. H.Khaled, R.EI Gohary, N.L. Badr, *et al* [11] have also presented GPU implementation of pairwise DNA sequence alignment problem. This implementation assigns differnet nucleotide weights and then merges the subsequences of match on GPU. The authors have obtained optimal local alignment according to predefined rules. Pair-wise sequence alignment for very long sequences was done in [12]. The authors have developed a single GPU implementation of the problem and have presented two algorithms, *BlockedAntidiagonal* and *StripedScore*. SW algorithm for protein database by using SIMD instruction of CPU and GPU is done in [13]. The paper presents CUDASW++ 3.0 algorithm that uses SSE-based vector execution units as accelerators. Yongchao Liu and Bertil Schmidt [14] have presented GSWABE algorithm for a pairwise sequence alignment problem for short DNA sequences. They have implemented general tile based approach for global, semi-global and local alignment algorithm on Kepler-based Tesla K40 GPU. The same problem is also implemented for long DNA sequences on Xeon Phi coprocessors by [15]. Authors have explored naive, tiled and distributed approaches on emerging platform.

Parallelization of similar problems like approximate string matching on GPU [16], finding edit distance for large sets of string pairs using MapReduce technique [17] and on GPUs [18] have been done for performance improvement. Problem of multiple sequence alignment [MSA] is one of the widely used and computationally complex problem in the deomain of computational biology. Algorithms for MSA must produce the highest score from the entire set of sequences and it is one of the complex optimization problems. Hence, heuristic methods are preferred over accurate methods. Jurate Daugelaite, Aisling O'Driscoll, and Roy D. Sleator [19] have summarized various MSA algorithms in distributed and cloud environment. High performance computing techniques have been used for MSA tools in [20]. Authors have developed MTA-TCofee tool. Optimal alignment of three sequences is presented by Junjie Li, Sanjay Ranka, & Sartaj Sahani [21]. The authors have also implemented a variant of global alignment, called *syntenic alignment* in their paper [22]. Paper [23] presents combination of G-MSA and T-Coffee algorithm for improving the performance of MSA on GPU. Comparison and analysis of various high performance computing archetectures in the field of bioinformatics, computational biology and systmes biology is presented in [24]. Global sequence alignment on multi-core platform using GPU is discussed by Siriwardena and RanaSinghe [25].

This paper presents a GPU implementation of pair wise sequence alignment algorithm (SW) as a case study to map the resource requirement of the algorithm to the available resources. The main features of our work are as follows:

- The pair-wise SW algorithm on CPU + single GPU platform is implemented. Multiple GPU implementations are presented in [9]. Allocation of strings, score matrix, deciding the block (tile) size, number of blocks, threads, launching concurrent kernels, is done on CPU side. The generation of score matrix, use of registers, invoking large number of threads, launching child kernel, is done on GPU side.
- The performance improvement using memory hierachies of the graphics card (like global memory, shared memory, constant memory, text memory) has been discussed by [10] [11]. However, the study of

GPU resources like cores, threads, warps, blocks, registers is done.

- The focus of our implementation is to effectively use GPU resources, to explore the features like multiple kernel execution supported by Kepler based NVIDIA CUDA cards (K5200, K6000). These features were not considered by previous studies [11-14]. The paper [15] has implemented the problem on Xeon-Phi coprocessor, and not on GPU.
- Our study mainly focuses on the use of resources like computing cores, registers per thread, shared memory per thread, thread block size. These parameters contribute towards GPU occupancy. Large number of cores available on graphics card can be very effectively utilized by exploring the features like dynamic kernels, concurrent kernel, thereby increasing the GPU occupancy.
- The paper mainly concentrates on parallelization of the score matrix generation part, which is the major compute intensive portion of the SW algorithm. The generation of aligned sequence (without gaps) is a backtracking process, carried out on CPU side.
- The implementation consists of splitting the score matrix into horizontal strips and then into the blocks or tiles. Tile size is decided by considering GPU resources. Every tile is then processed by anti-diagonal parallelization method using concurrent or dynamic kernel method. Whereas, the approach used in [12] is of vertical stripped SW algorithm considering the parameters of the global & shared memory of the GPU itself.
- The features like dynamic parallelism, use of multiple, concurrent kernels using streams supported by NVIDIA graphics cards have been explored.

The rest of the paper is organized as follows:

Section 3 describes the architecture of Graphics Card. Description of algorithm is presented in Section 4. Score matrix generation using various approaches is described in Section 5. Section 6 presents implementation of algorithm and comparative performance improvement. The conclusion is presented in Section 7.

III. GPU Architecture

GPUs have large number of processing elements called as streaming multiprocessors (SMs) to host thousands of threads and blocks of threads. Higher throughput is achieved by concurrently executing these large number of threads. This is thread level parallelism (TLP). The implementation has been done on multi-core machines with NVIDIA graphics cards Quadro K5200, K6000. CUDA C is the programming language supported for accessing GPU cards. These are professional class GPU cards for integrating high performance computing applications. The cards connect to the host processor via a PCIe 3.0 bus. It is a programming challenge to effectively manage the data traffic between the host (CPU) and the device (GPU). If this data traffic is handled properly, it would lead to performance improvement by proper utilization of memory bandwidth. The other issue in executing algorithm is to judiciously manage the memory traffic between the streaming multi-processors and various memory components on the card. Both the cards have Kepler micro architecture that supports dynamic parallelism. With this feature, CUDA kernel can create a child kernel (as shown in Fig. 1) that can perform new independent, parallel task, create and use new streams, events, without CPU involvement. The Kepler architecture supports L1 cache per SM with a unified memory request path for loads and stores. Memory model is shown in Fig. 2. The detail technical specification of cards used is shown in Table 1. The multi-core system with 16 cores, Intel Xeon E5-2698 processor with 2.3 GHz clock frequency with GPU card, was used for implementation.

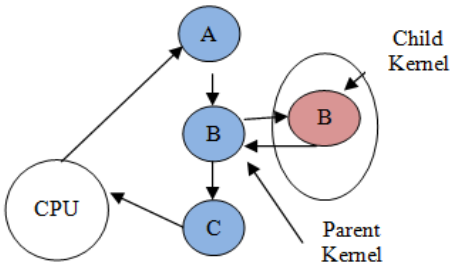

Fig. 1. Dynamic Parallelism in CUDA.

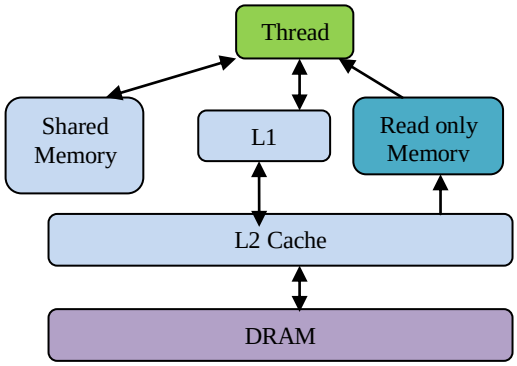

Fig. 2. Kepler Memory Hierarchy.

TABLE I. SPECIFICATIONS OF GPU

Specification	**Quadro K5200**	**Quadro K6000**
GPU Memory	8 GB GDDR5	12 GB GDDR5
Memory Interface	256 -bit	384-bit
Memory Bandwidth	192.0 GB/s	288 GB/s
CUDA Cores	2304	2880
System Interface	PCI-E3.0x16	PCI-E3.0x16
Shared Memory per Block	49152 bytes	49152 bytes
Maximum Threads per Block	1024	1024
Number of Multiprocessors (SM)	12	15
Number of CUDA cores per SM	192	192

IV. Algorithm Description

In the biological literature, global alignment is often known as NW alignment and local alignment as a SW alignment [1][2]. Global alignment method is used to catch the regions of high similarity between two sequences. But, it may not be possible to find out the regions of high local similarity, during overall optimal global alignment. Hence, local alignment is used to effectively tap the regions of high local similarity. There are certain issues to be considered while aligning two sequences for similarity quotient.

- Length of sequences may not be equal.
- There may be small matching regions in the sequences.
- Whether to allow partial matches or not. (i.e. some amino acid pairs can replace the other one)
- There may be the cases of insertions, deletions, or substitutions from the common ancestral sequence. This may lead to variable length regions, mutations, or gaps in the new alignment.

Consider strings S_1 & S_2, (over the alphabet {A,C,G,T}) of lengths n & m respectively. Then dynamic programming approach solves local alignment problem in $O(nm)$ time. The score matrix S is created, which is used to generate similarity index between two strings. The recurrence relation establishes a recursive relationship between the element $S(i,j)$ and other elements of the score matrix. The base conditions are: $S(i,0) = 0$, and $S(0,j) = 0$. The recurrence relation for $S(i,j)$, when both i and j are strictly positive is given in Fig. 3, where α, β denote gap penalty. Fig. 4 shows data dependency.

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + w(\alpha_i -) \\ S_{i,j-1} + w(-, \beta_j) \\ S_{i-1,j-1} + w(\alpha_i, \beta_j) \\ 0 \end{cases}$$

Fig. 3. Recurrence Relation in Score Matrix.

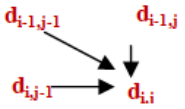

Fig. 4. Data Dependency of SW Algorithm.

V. Score Matrix Generation

This section describes parallel approach for alignment problem, CUDA kernels for generating score matrix, and algorithm parameters.

A. Many Core Implementation on GPU

CUDA enabled GPU card with compute capability greater than 3.5 supports the features like dynamic parallelism, concurrent kernels. Dynamic parallelism is expressed by invoking nested kernels. Fig. 5 shows the algorithm for dynamic parallelism. Here, "*gpuBC*" is parent kernel that creates and calls child kernel "*fillmatrix*". Parent kernel creates a grid of size (T *X* T) of blocks (where T is number of threads per block). Total number of blocks in each direction is $(N+1)/(T+1)$, where "N" is length of query string. The child kernel "*fillmatrix*" generates the entries in the score matrix(C), in the diagonal parallelization manner. There is an implicit synchronization between a child & parent grid. Main program on the host allocates and initializes the score matrix C on the host, copies it on the device and calls the parent kernel. The parent kernel calls the child kernel on the device. Concurrent kernel execution can be invoked by using independent "stream" for every host thread. Fig. 6 shows the algorithm for this approach. For example, generation of score matrix can be split into four parts. Due to diagonal dependency, these four parts can be wrapped into three independent streams as shown in Fig. 7. These streams can be executed concurrently in the following order. Stream1 executes kernel1, stream2 executes kernel2 & kernel3, and stream3 executes kernel 4. The execution sequence is shown in Fig. 8. *CudaStreamCreate(&stream(i))* creates three streams for kernel 1, kernels 2 & 3, and kernel 4, respectively. Streams are synchronized using *CudaStreamSynchronize()*. The grid pattern (number of blocks, number of threads per block) is specified as an argument to each kernel.

B. Tiling Approach

For the strings of very large sizes (especially string lengths, that generate the score matrix of size more than the size of global memory of the card), score matrix on host side is divided into suitable chunks (or tiles). It is essential to calculate proper tile size and the effective address calculations of all subsequent threads, using Block ID and Thread ID model of CUDA environment. For example, if tile size is $t \; x \; t$, element size is 'e', size of memory is 'm', then, in order to accommodate the entire tile in the global memory of GPU card, equation 1 should be satisfied.

$$t \times t \times e \leq m \quad (1)$$

```
// Dynamic Kernels
//  Parent Kernel
__global__ void gpuBC(int *c_d, int *b_d)
{  // create grid for child kernel, with block size TxT
dim3 thrperblk(T,T);
dim3 numblks ((int)((N+1)/T+1), (int)((N+1)/T+1));
maxsum=N+N;
for (sum = 0; sum <= maxsum; sum++)
{     // calling Child Kernel
fillmatrixC<<<numblks, thrperblk>>>(c_d, sum);
cudaThreadSynchronize();     } }
// Parent Kernel ends here
Main( )
{  // allocate score matrix (c), strings s1, s2 on host
// initialize the c, s1 & s2 on host
//  copy s1, s2, matrix C on device using CudaMalloc()
//  Match= +m, mismatch= -t,gap = -g
//    Call to Parent Kernel
gpuBC<<<1,1>>>(c_d);
//  copy  matrix c back to host CudaMalloc()
// thread synchronization
cudaThreadSynchronize();
// timing calculations &  cleanup...
   free(c_h);   cudaFree(c_d); cudaFree(b_d);
   return(0);
}
```

Fig. 5. Dynamic Kernels in CUDA.

Score matrix is split into horizontal strips. Each strip is then broken into blocks or tiles. Within every strip, each tile is executed one by one as shown in Fig. 9. The algorithm is presented in Fig. 10.

```
//Concurrent Kernel execution using "Streams"
Main( )
{ // allocate score matrix (c), strings S1, S2 of size N  on host
// initialize the c[N+1][N+1], s1 & s2 on host,  sum=N+N
//  rowmin, rowmax, colmin, colmax are data boundaries for kernel execution, copy s1, s2, matrix C on device using CudaMalloc()
//  Match= +m, mismatch= -t,gap = -g  //  create grid with block size T
    dim3 thrperblk(T,T);
    dim3 numblks ((int)((N+1)/T+1), (int)((N+1)/T+1));
 //  create streams
    for (i=0; i<3; i++)
    cudaStreamCreate (&stream(i));
 //  Kernel calls using streams,  kernel1, then kernel2 & kernel3 concurrently , and then kernel4
     for (sum = (rowmin+colmin); sum <= (rowmax+colmax); sum1++)
    { kernel1<<<numblks, thrperblk, 0, stream0>>>(c_d, sum1);
        cudaThreadSynchronize();
        cudaStreamSynchronize(stream);  }
    for  (sum  =  (rowmin+colmin);  sum  <=  (rowmax+colmax); sum1++)
    {   kernel2<<<numblks, thrperblk, 0, stream1>>>(c_d, sum1);
       cudaThreadSynchronize();  }
    for  (sum  =  (rowmin+colmin);  sum  <=  (rowmax+colmax); sum1++)
    {   kernel3<<<numblks, thrperblk, 0, stream2>>>(c_d, sum1);
       cudaThreadSynchronize();  }
    cudaStreamSynchronize(stream0);
    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2);  //synchronizing  previous streams
    for  (sum  =  (rowmin+colmin);  sum  <=  (rowmax+colmax); sum1++)
    {   kernel4<<<numblks, thrperblk, 0, stream1>>>(c_d, sum1);
       cudaThreadSynchronize();  }
   cudaStreamSynchronize(stream1);  }    // synchronizing ALL streams
 //  copy  matrix C back to host,  destroy streams
```

Fig. 6. Concurrent Kernels in CUDA.

Fig. 7. Four Kernels to Fill Score Matrix.

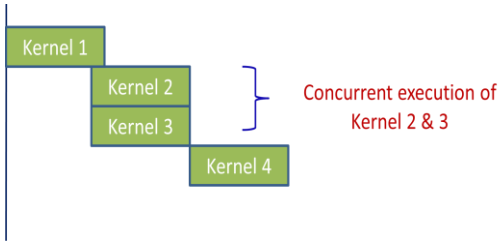

Fig. 8. Concurrent Execution of Kernels.

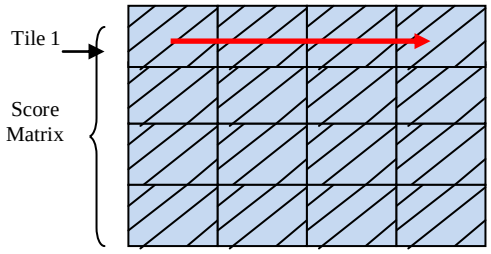

Fig. 9. Horizontal Strips and Tiles.

```
//t x t tiles of (NxN)matrix, total t^2 tiles to be processed
Main( )
{  //  allocate & initialize score matrix (c_h), strings S1, S2 on host
  // allocate tile (c_h1)of size t*t on host , copy s1, s2 on device (s1_d, s2_d)
  // For each horizontal strip & tile of size txt
   // copy tile from host to device(c_d), execute & copy back to host
   for (i = 0; i<N; i=i+N/t)
     for (j=0; j<N; j=j+N/t)
  {     rowmin = i; colmin =j; rowmax = i+N/t; colmax= j+N/t;
     create_c_h1(c_h1, c_h, rowmin, rowmax, colmin, colmax);
cudaMemcpy(c_d,c_h1,(N/t+1)*(N/t+1)*sizeof(int),cudaMemcpyHostToDevice);
     K-Scoremat<<<1,1>>>(c_d, s1_d, s2_d, rowmin, colmin, rowmax, colmax);
       cudaThreadSynchronize();
cudaMemcpy(c_h1,c_d,(N/t+1)*(N/t+1)*sizeof(int),cudaMemcpyDeviceToHost);
     create_c_hback(c_h1, c_h, rowmin, rowmax, colmin, colmax);
     cudaFree(c_d);  cudaThreadSynchronize();
     cudaMalloc((void **)&c_d,(N/t+1)*(N/t+1)*sizeof(int));  } }
 // kernel execution
__global__ void K-Scoremat(int *c_d, char *s1_d, char *s2_d, int rmin, int cmin, int rmax, int cmax)
{   int sum,maxsum, rowmin, colmin;
    int T=32;  // Block size
    rowmin = rmin;  colmin = cmin;  rowmax = rmax, colmax = cmax;
    maxsum = N;
    dim3 thrperblk(T,T);
    dim3 numblks ((int)(((N/t)+1)/T+1), (int)(((N/t)+1)/T+1));
    for (sum = 0; sum <= maxsum; sum++)
    {    // calling Child kernel FiilmatrixC
       fillmatrixC<<<numblks, thrperblk>>>(c_d, s1_d, s2_d, sum, rowmin, colmin);  }  }
// kernel ends here
```

Fig. 10. Tiling Algorithm.

C. Resource Requirement & GPU Occupancy

Occupancy is a function of GPU card parameters and resource requirement of the algorithm. Hence, potential limitations for occupancy are the resources like registers, memory and number of streaming multi-processors (SM) required by the algorithm. Resources would be fully utilized, only when

$$Number\ of\ concurrent\ threads\ required\ by\ algorithm \geq Number\ of\ parallel\ threads\ on\ device$$

For pair wise sequence alignment problem, maximum occupancy would be experienced, if

$$\sqrt{(N+1)^2+(N+1)^2} = C_g \quad (2)$$

Where, N is length of string, and C_g is total number of GPU cores on device.

$$Occupancy = \frac{Active\ Threads\ per\ block}{Threads\ per\ SM} \quad (3)$$

Occupancy can be determined by considering device parameters as well as certain design parameters. These parameters are shown in Table 2.

- *Register usage*-The number of registers needed per thread limits the register usage. Occupancy can be decided by thread ratio.

$$\text{Active Threads per Block, } T_a = R_g/R_a$$
$$\therefore Occupancy = O_1 = T_a/T_g \quad (4)$$

- *Shared Memory usage*-Occupancy can also be decided by considering the shared memory usage. $No.\ of\ threads\ supported,$

$$T_a = S_g/S_a$$
$$\therefore Occupancy = O_2 = \frac{Active\ Threads\ per\ block}{Threads\ per\ SM}$$
$$\therefore Occupancy = O_2 = T_a/T_g \quad (5)$$

- *Thread Block Size*-Block size is a design criteria, which decides how many SMs can be utilized depending upon the number of active blocks used by each kernel. One warp consists of 32 threads.

$$\therefore No.\ of\ warps\ per\ block\ \ W_a = T_a/32$$
$$\therefore No.\ of\ Active\ threads\ per\ block = T_a = B_a \times Z_a$$
$$Occupancy = O_3 = \frac{No\ of\ Active\ Threads\ per\ block}{No\ of\ threads\ per\ SM}$$
$$\therefore Occupancy = O_3 = T_a/T_g \quad (6)$$
$$If\ O_1, O_2, O_3 \geq 1,\ Occupancy = 1\ (100\%)$$

Every resource parameter contributes to the GPU occupancy. Occupancy may not be the measure of the performance, but low occupancy codes reflect underutilization of the enormous resources offered by the execution platform.

- Resource requirement of the algorithm

Number of GPU Cores-Let the tile size be $t \times t$, length of diagonal be x. For diagonal parallelization method, number of threads required per block is maximum at diagonal. For 100% occupancy, all the cores should be utilized. Then for maximum utilization of GPU cores,

$$x \geq G \quad \text{but,} \quad x = \sqrt{2} \times t$$
$$\sqrt{2} \times t \geq G$$
$$\therefore Tile\ Size,\ t \geq G/\sqrt{2} \quad (7)$$

Memory Size-It is required that, tile should be accommodated into the memory completely. $Tile\ Size = t \times t \times s$,

where 's' is the size of element

$$Tile\ Size \leq S_m$$
$$\therefore Tile\ Size,\ t \leq \sqrt{S_m/s} \quad (8)$$

Combining equations (7) (8), we get

$$\frac{G}{\sqrt{2}} \leq t \leq \sqrt{S_m/s} \quad (9)$$

Table 3 shows the corresponding values for GPU card K5200 & K6000

D. Date Transfer Issues

Time required to transfer the data from host memory to device memory depends upon the bandwidth of PCI bus. On device side, memory may be allocated as pinned memory or non-pinned (pageable) memory. It is observed that, the peak bandwidth between various device memories is much higher than the peak bandwidth between the host and device memory. Thus, data transfer time between host and device, is the major contributor towards the overall performance. Higher bandwidth is possible between the host and the device when transfer overheads are minimal, and data transfer is overlapped with kernel execution and other data transfers.

TABLE II. PARAMETERS FOR OCCUPANCY

Device Parameters		**Design Parameters**	
Registers per SM	R_g	Registers used by the kernel	R_a
Threads per SM	T_g	Threads per block	T_a
Shared memory per SM	S_g	Shared memory required per thread by kernel	S_a
Warps per SM	W_g	Active warps per block	W_a
Number of GPU cores	G	No. of active blocks per kernel	B_a
		No. of Active Threads per block	Z_a

TABLE III. TILE SIZE LIMITS FOR GPU CARDS

GPU Card	Tile size limits
K5200	$1629 \leq t \leq 46340$
K6000	$2036 \leq t \leq 56755$

VI. RESULTS AND DISCUSSION

A. Many Core Implementation

Experiments were carried out for parallel implementation of SW algorithm on many core systems. Parallelization was done using following approaches:

1) Using only dynamic kernel.
2) Using only concurrent kernel.
3) Using tiling technique, coupled with above two methods.

For approach 'a', dynamic parallelism was tested. Parent kernel on device launches the child kernel. For 'b', multiple kernels, wrapped in different streams were launched from the host. However, for approach 'c', tiling method was used. Entire score matrix was split into horizontal strips and then into tiles of size that could be accommodated into the global memory of the device. Processing of each tile was carried out using anti-diagonal method of parallelization. In this method, both the features (a & b above) were tested. The implementation was compared against serial CPU based implementation on the same platform. Speed up was calculated with respect to time taken to execute the serial version of the algorithm on CPU.

$$Speed\ up = \frac{Time\ required\ to\ execute\ serial\ CPU\ version}{Time\ required\ to\ execute\ GPU\ vrsion} \quad (10)$$

Speed up of about 120X and 55X was observed using dynamic kernel and concurrent kernel features respectively. Initially, the speed up achieved by both the approaches is comparable. As the string size increases, the size of score matrix and searching time also increases. The speed up saturates for higher string sizes, when bandwidth is fully utilized. Tiling technique outperforms above two approaches, for larger string sizes. Speed up of about 240X is observed with the use of combined (tiling + dynamic & concurrent kernel) technique. Fig. 11 shows the results. Nearly same speed up is observed when tiling method is used with either concurrent or dynamic kernel approach. The comparative speed up with and without using tiling technique with both the approaches (dynamic & concurrent kernel) was carried out.

$$Speed\ up = \frac{Execution\ tim\ of\ only\ dynamic\ or\ concurrent\ kernel}{Execution\ time\ of\ Tiling+corresponding\ kernel} \quad (11)$$

Fig. 12 shows the speed up when tiling technique is coupled with concurrent & dynamic kernel features. With this method, speed up of 4.2X (for tiling + concurrent kernel over only concurrent kernel) and 2X (for tiling+dynamic kernel over only dynamic kernel) is achieved.

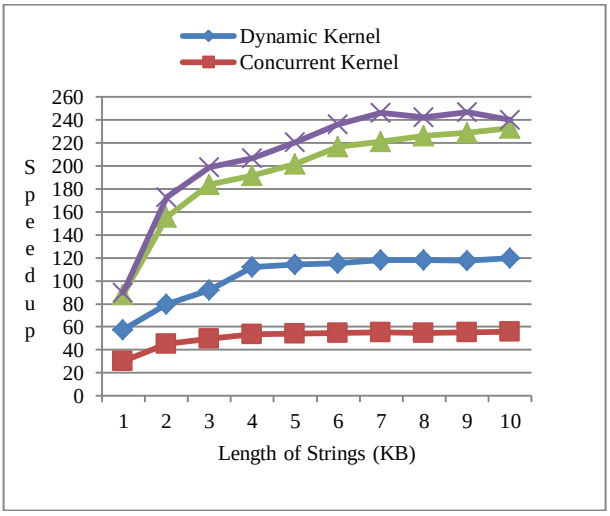

Fig. 11. Speed up for Tiling and Non-Tiling Approaches.

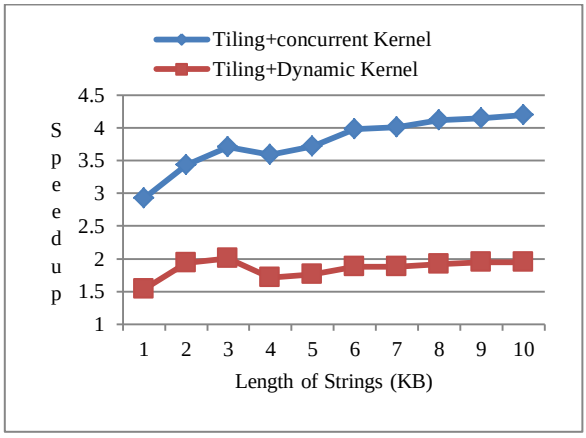

Fig. 12. Speed up When Tiling is used with Respect to Non-Tiling Technique.

The main focus was on score matrix generation part of the algorithm, since, it is the major contributor towards the execution time. The serial execution of trace-back part of the algorithm was not considered. Therefore, it would be inappropriate to compare the results directly, with the results of any previous outcomes.

B. Resource Utilization and GPU Occupancy

GPU occupancy defines how efficiently the algorithm utilizes the resources provided by the underlying hardware. Occupancy will be less, if more registers, more shared memory per thread are needed by the kernel and the thread block size is small. For large data sets, occupancy is more than 100%. The tile size limits given in Table 3 has been verified and the results are shown in Table 4. It is observed that there is about 50% reduction in execution time, when tile size limits are followed. For all experiments, thread block size is minimum 256 and maximum 1024 threads per block. If the block size is less, number of blocks required for the given data size would be much more, and occupancy would be less.

TABLE IV. TESTING TILE SIZE LIMITS

GPU Card	String Length	Tile Size $t < t_{min}$		Tile Size $t > t_{min}$	
	(KB)	*t*	*Execution time (sec)*	*t*	*Execution time (sec)*
K5200 t_{min}= 1629	8	1024	34.433426	2048	**15.896714**
	10	1280	41.521371	2560	**27.354436**
	12	1536	62.6178184	3072	**45.267902**
K6000 t_{min}= 2036	8	1024	30.952559	2048	**14.506293**
	10	1280	50.319086	2560	**22.117568**
	12	1536	69.889266	3072	**34.735937**

C. Issues in Data Transfer

The aspects like, allocating memory on GPU using cudamalloc() or cudaHostAlloc(), use of pinned or non-pinned memory allocation, use of constant memory for read only data were explored. Memory allocation on GPU can be done using non-pinned (pageable) or pinned allocation method. The pinned transfers are faster than non-pinned transfers for smaller data sizes (for string sizes from 16KB upto 44KB), as shown in Table 5. But too much allocation of pinned memory degrades the performance. Hence, for large string sizes, pageable, i.e. non-pinned memory allocation is preferred. Constant memory of the GPU card can be used to store all read only data of the algorithm. A request for constant memory for the entire warp is split into two parts. When all the threads in a warp access the same memory location, two requests for each half warp are generated. Reading from constant memory location is thus as fast as reading from the registers. There is a serialized access to the addresses by the threads in a half warp, leading to performance improvement. Table 6 shows the improvement in execution time while using constant memory for non-pinned allocation. Use of pinned memory and constant memory contribute towards the performance improvement only for limited data sizes. But, due to limited size of constant memory (64KB), dynamic memory allocation is required even for storing constant data.

TABLE V. PINNED AND NON-PINNED MEMORY

String Size (KB)	Execution time Non-pinned memory (sec)	Execution time Pinned memory (sec)
16	159.347031	**155.331391**
20	309.529562	**302.237375**
24	538.443125	**515.289562**
28	845.583813	**815.48725**
32	1204.949	**1209.99325**
36	1725.6545	**1721.26288**
40	2449.70975	**2357.2095**
44	3284.45625	**3005.6027**

TABLE VI. CONSTANT MEMORY

String Size (KB)	Execution time for using Constant memory (sec)	Execution time for No use of Constant memory (sec)
1	**0.051225**	0.060482
4	**2.608883**	2.682937
8	**19.793088**	19.999418
12	**64.76441**	65.431309
16	**151.555297**	159.347031
20	**298.78875**	309.529562
24	**513.011656**	538.443125
28	**812.932937**	845.583813
32	Not Working	Not Working

VII. CONCLUSION

The main focus of our study was to explore the features of the graphics cards and map the resource requirement of the algorithm under consideration with the available resources. Experiments with compute intensive part of pair-wise SW algorithm, i.e. score matrix generation were performed. Hence, our results are not directly comparable to the previous results. Heavy data dependent applications can be parallelized on GPU platform by coupling traditional tiling technique with the features like concurrent and dynamic kernel execution. Speedup up to 120X and 55X was observed, while using dynamic and concurrent kernel features respectively. Further performance improvement of about 240X was possible by using tiling method. Tile size was decided by considering the relationship between various device and algorithm parameters. This led to achieving a speed up of about 2X relative to using only dynamic kernel and about 4.2X relative to using only concurrent kernel approach. The utilization of GPU resources was tested with respect to register usage, shared memory usage and thread block size. It is observed that, for higher occupancy, it is necessary to do more work per thread, use more registers per thread in order to access slower shared memory. The relationship between the tile size and available resources on the device for better resource utilization and performance improvement is presented. We plan to extend our work on incorporating memory and compiler optimization issues on parallelizing the dynamic programming based algorithms on GPU. The proposed strategy can also be extended for global sequence alignment, multiple sequence alignment problems as well.

REFERENCES

[1] S. Needleman, C. Wunch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," Journal of Molecular Biology, 48, (3), pp. 443-453, 1970.

[2] T. Smith, T., M. Waterman, "Identification of Common Molecular Subsequences," Journal of Molecular Biology, 147, (1), pp. 195-197, 1981.

[3] W. Wilber, D. Lipman, "Rapid Similarity Searches of Nucleic Acid and Protein Data Banks," Proc. Natl. Academy Sci. USA, 80, pp. 726-730, 1983.

[4] W.R. Pearson, D. Lipman, "Improved Tools for Biological Sequence Comparison," Proc. Natl. Academy Sci. USA, 85, pp. 2444-2448, 1988.

[5] C. Shuai, L. Jie, J. Sheaffer, K. Skadron, J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," proceedings of Symposium on Application Specific Processors, SASP'08, California, USA, pp. 101-107, June 2008.

[6] S. Lloyd, Q. Snell, "Hardware Accelerated Sequence Alignment with Traceback,", International Journal of Reconfigurable Computing, Article ID 762362, 10 pages , 2009.

[7] Y.Chen, S. Yu, M. Leng, "Parallel Sequence Alignment Algorithm for Clustering System," International Federation for Information Processing (IFIP),207, pp. 311-321, 2006.

[8] O. Gotoh, "An Improved Algorithm for Matching Biological Sequences," Journal of Molecular Biology, 162, pp. 705-708, 1982

[9] A. Khajeh-Saeed, S. Poole, J. Perot, "Acceleration of the Smith-Waterman algorithm using single & multiple graphics processors," Int. Journal of Computational Physics, 229, (11), pp. 4247-4258, 2010.

[10] L. Ligowski, W. Rudnicki, "An Efficient Implementation of Smith Waterman Algorithm on GPU using CUDA, for Massively Parallel Scanning of Sequence Databases," IEEE International Symposium on Parallel & Distributed Processing (ISPDP), Rome, Italy, pp. 1-8, May 23-29, 2009.

[11] H. Khaled, R. Gohary, N. Badr, H.M. Fahneem, "Accelerating Pairwise DNA Sequence Alignment using the CUDA Compatible GPU," International Journal of Computer Applications (IJCA), 14, (1), 2013.

[12] J. Li, S. Ranka, S. Sahni, "Pairwise Sequence Alignment for Very Long Sequence on GPUs," 2nd International IEEE Conference on Computational Advances in Bio and Medical Sciences (ICCABS), LasVegas, NV, USA, pp. 1-6, 2012.

[13] Y. Liu, A. Wirawan, B. Schmidt, "CUDASW++3.0: Accelerating Smith-Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions," Journal of BMC Bioinformatics, 14, (117), 2013.

[14] Y. Liu, B. Schmidt, "GSWABE: Faster GPU-Accelerated Sequence Alignment with Optimal Alignment Retrieval for Short DNA Sequences," Int. Journal of Concurrency And Computation: Practice And Experience, 27, (4), pp. 958-972, 2015.

[15] Y. Liu, T. Tran, F. Lauenroth, B. Schmidt, "SWAPHI-LS: Smith-Waterman Algorithm on Xeon Phi Coprocessors for Long DNA Sequences'" IEEE International Conference on Cluster Computing (CLUSTER), Madrid, Spain, pp. 257-265, Sept, 2014.

[16] K. Nakano, "Efficient Implementation of the Approximate String Matching on the Memory Machine Models," 3rd IEEE International Conference on Networking & Computing (ICNC), Okinawa, Japan, pp. 223-229, 2012.

[17] S. Jhaver, L. Khan, B. Thuraisingham, "Calculating Edit Distance for Large Sets of String Pairs using MapReduce," ASE International Conference on BigData / SocialComuting / CyberSecurity, Stanford University, USA, 2014

[18] R. Farivar, H. Kharbanda, S. Venkataraman, R.H. Campbell, "An Algorithm for Fast Edit Distance Computation on GPUs," IEEE Conference on Innovative Parallel Computing (InPar), SanJose, CA, USA, pp. 1-9, 2012.

[19] J. Daugelaite, A. Driscoll, R. Sleator, "An Overview of Multiple Sequence Alignments and Cloud Computing in Bioinformatics," Hindawi Publishing Corporation, International Scholarly Research Notices (ISRN) Biomathematics, Article ID 615630, 14 pages, 2013.

[20] M. Orobitg, F. Guirado, F. Cores, F. Cores, J. Llados, C. Notredame, "High Performance Computing Improvements on Bioinformatics Consistency-based Multiple Sequence Alignment Tools," International Journal of Parallel Computing, 42, pp. 18-34, 2015.

[21] J. Li, S. Ranka, S. Sahni, "Optimal Alignment of Three Sequences on A GPU," proceedings of 6th International Conference on Bioinformatics and Computational Biology (BICoB'14), Las Vegas, Nevada, USA, pp. 177-182, 2014.

[22] J. Li, S. Ranka, S. Sahni, "Parallel Syntenic Alignment on GPUs," proceedings of ACM Conference on Bioinformatics, Computational Biology, Biomedicine (ACM-BCB), Orlando, Florida, USA, pp. 266-273, 2012.

[23] S. Fazeli, S. Rahimi, "Investigation and Parallel Implementation of Multiple Sequence Alignment using Graphics Processing Units (GPU)," Int. Journal of Advanced Biotechnology and Research (IJBR), pp. 1201-1208, 2016.

[24] M. Nobile, P. Cazzaniga, A. Tangherloni, D. Besozzi, "Graphics Processing Units in Bioinformatics, Computational Biology and Systems Biology," Briefings in Bioinformatics, 18(5), pp. 870-885, 2017.

[25] T. Siriwardena, D. RanaSinghe, "Global Sequence Alignment using CUDA compatible multi-core GPU," 5th IEEE International Conference on Information and Automation for Sustainability (ICIAFS), Colombo, Srilanka, pp. 201-206, 2010.