Highly Efficient Mapping of the Smith-Waterman Algorithm on CUDA-compatible GPUs

Keisuke Dohi*, Khaled Benkrid†, Cheng Ling†, Tsuyoshi Hamada* and Yuichiro Shibata*
*Dept. of Computer and Information Sciences
Nagasaki University
Nagasaki, Japan
Email: {dohi, hamada, shibata}@pca.cis.nagasaki-u.ac.jp
†School of Engineering, King's Buildings.
The University of Edinburgh
Edinburgh, United Kingdom
Email: {k.benkrid, c.ling}@ed.ac.uk

***Abstract*—This paper describes a multi-threaded parallel design and implementation of the Smith-Waterman (SW) algorithm on graphic processing units (GPUs) with NVIDIA corporation's Compute Unified Device Architecture (CUDA). Central to this is a divide and conquer approach which divides the computation of a whole pairwise sequence alignment matrix into multiple sub-matrices (or parallelograms) each running efficiently on the available hardware resources of the GPU in hand, with temporary intermediate data stored in global memory. Moreover, we use thread warps and padding techniques in order to decrease the cost of thread synchronization, as well as loop unrolling in order to reduce the cost of conditional branches. While intermediate data is stored in global memory for large queries, the most inner loop in our implementation will only access shared memory and registers. As a result of these optimizations, our implementation of the SW algorithm achieves a throughput ranging between 9.09 GCUPS (Giga Cell Update per Second) and 12.71 GCUPS on a single-GPU version, and a throughput between 29.46 GCUPS and 43.05 GCUPS on a quad-GPU platform. Compared with the best GPU implementation of the SW algorithm reported to date, our implementation achieves up to 46% improvement in speed. The source code of our implementation is available in the public domain for Bioinformaticians to benefit from its performance.**

***Keywords*-CUDA; GPU; Smith-Waterman algorithm;**

I. Background

Biological sequence alignment is a widely used and crucial operation in bioinformatics and genetics research. The purpose of it is to find the best possible alignment of a set of sequences, with various real world applications including phylogenetic tree construction, disease diagnosis, and drug engineering. However, biological sequence alignment is also a computationally expensive application as its computing and memory requirements grow rapidly with the size of the databases and queries which leads to massive execution times on standard desktop computers.

Graphics Processing Units (GPUs) have been proposed recently as high performance and relatively low cost acceleration platforms for biological sequence alignment [1]. As modern GPUs have become increasingly powerful, inexpensive and relatively easier to program through high level API functions, they are increasingly being used for non-graphic or general purpose applications (the so-called GPGPU computing). In this paper, we will present how Compute Unified Device Architecture (CUDA) GPUs can be used as hardware platforms to accelerate pairwise sequence alignment using the Smith-Waterman (SW) algorithm [2] [3]. The SW algorithm adopts a dynamic programming mechanism which provides the best local alignment between two sequences using exhaustive search, at the expense of a high computational load compared to less accurate heuristics-based algorithms such as the BLAST algorithm [4]. However, with faster computing platforms, this trade-off need not take place as we can implement the exhaustive SW algorithm efficiently. The remainder of this paper is organized as follows. First, relevant background on the SW algorithm is presented. Then, the CUDA programming model is presented. After that, our multi-threaded parallel design and implementation of the SW algorithm and pseudo code will be presented. A comparative evaluation between our implementation and the state-of-the-art of GPU implementations as well as various micro-processor based implementations is then presented before conclusions and ideas for future work are laid out.

II. The Smith Waterman Algorithm

The SW algorithm is a dynamic programming algorithm which finds the best local fragment between two biological sequences. The search for the optimal local alignment consists of two stages. Firstly, an alignment matrix of two biological sequences (e.g. protein, DNA) is calculated and results in a maximum score. After that, the alignment traces back from the maximum score until a zero element is found.

978-1-4244-6967-3/10/$26.00 © 2010 IEEE

Note that the final trace back procedure will be done only for one or few subject sequences, the one(s) with the highest scores, out of the most subject sequences, and hence is often performed on the host CPU.

More specifically, let $D = d_0 d_1 ... d_{m-1}$ denotes a database sequence with length m. Let $Q = q_0 q_1 ... q_{n-1}$ denotes a query sequence of length n. Let $W(a_i, b_j)$ denotes the substitution scoring matrix [5] , which gives a score describing the likelihood of substitution between characters a_i and b_j. Let G_{init} and G_{ext} denote penalties for opening a new gap and continuing an existing gap respectively.

With the above, the alignment matrix computation of the SW algorithm is described by the following equations:

$$\begin{aligned} E_{i,j} &= \max\begin{cases} H_{i,j-1} - G_{init} \\ E_{i,j-1} - G_{ext} \end{cases} \\ F_{i,j} &= \max\begin{cases} H_{i-1,j} - G_{init} \\ F_{i-1,j} - G_{ext} \end{cases} \\ H_{i,j} &= \max\begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + W(a_i, b_j) \end{cases} \end{aligned}$$

The values of $H_{i,j}$, $E_{i,j}$ and $F_{i,j}$ are defined as 0 if $i < 0$ or $j < 0$. The gap penalty is called linear if $G_{init} = G_{ext}$, otherwise, it is called affine. From these equations, we observe that the value of $H_{i,j}$ depends on the values of its upper neighbour $H_{i,j-1}$, left neighbour $H_{i-1,j}$ and left-upper neighbour $H_{i-1,j-1}$, as shown in Figure 1.

The above operations are massively parallelisable since the anti diagonal elements of the alignment matrix are independent of each other (as labelled with the dot pattern in in Figure 1), and hence can be computed in parallel. In addition, the computation of different alignment matrices between a query sequence and different subject sequences can also be done in parallel. Since GPU have the ability of allocate thousands of parallel threads to a particular task, it is a very appealing acceleration platform for the SW algorithm. The GPU parallelisation task is hence focused on the alignment matrices' calculation.

III. CUDA Programming Model

CUDA is a development environment for GPGPU offered by NVIDIA Corporation [6]. The language specification is an extension of C/C++ with a multithread programming model. There are a series of GPUs that are compliant with CUDA, and each CUDA-compliant GPU device has a version known as "CUDA compute capability". Additionally, the amount of hardware resources can be different from one CUDA-compliant GPU to another. In this paper, we use a GPU that has a GPU core called GT200b. The CUDA compute capability of GT200b is 1.3.

A the lowest level, the concept of CUDA is parallel processing with hierarchically grouped multiple threads. The group of threads is called "Thread block" and the group of thread blocks is called "Grid". Each thread executes the same code which is described as a sequential process, called "Kernel". In CUDA, a kernel execution is called "Kernel call". Each thread has a unique ID within a grid and is able to execute different instructions and/or operate on different data by using the ID, although they execute the same code. There is a synchronization statement that is essential in parallel computing as it synchronizes all threads within a thread block. There is no synchronization statement that synchronizes all of the threads within a grid. However, we can synchronize all threads within a grid out of the kernel call by using CUDA API. Additionally, as we discuss later, actual processing goes along in 32-thread unit, called "Warps" or 16-thread unit that are called "Half-warps".

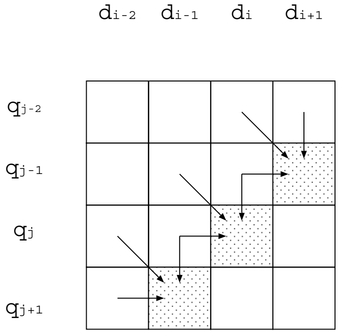

Figure 1. Data dependency of the SW dynamic programming algorithm. Arrows indicate dependency for the lower right element.

Memory in CUDA-compatible GPUs is hierarchical. It is important to understand the characteristics of these memories e.g. amount, bandwidth and region of sharing, to optimize CUDA code [7] [8]. The following presents the memories that we used in our implementation:

- Global memory: This memory has a large space (typically 1 GB) and is shared by all of the threads within a grid, but latency is high and bandwidth is lower than on-chip memory [7] . Additionally, it is not cached.
- Register: This is on-chip memory that is private to each thread. It is the fastest memory within CUDA and there are 16 K 128-bit registers within a thread block in GT200b GPU cores.
- Shared memory: This is on-chip memory that is shared by threads within a thread block. It is as fast as registers but its size is not very large (16 KB per thread block with GT200b GPU cores).
- Texture memory: This is read-only memory, shared by all threads within a grid and is cached. It looks like the cached global memory, hence its size depends on global memory.
- Constant memory: This is read-only memory like the texture memory. Its size is not large (64 KB per grid with GT200b GPU core) but it is as fast as registers if

all threads within a half-warp access the same address.

A. Memory access issues with CUDA

Each type of memory within CUDA GPUs has its own performance characteristics. In this sub-section, we will discuss these issues for two different kinds of memories: global memory and shared memory.

One of the most important performance considerations is "Coalesced memory access" [8]. In CUDA, memory access instructions for global memory from threads within a half-warp can be packed into one or more wide memory access instructions. If they are packed into just one wide memory access instruction, it is called the coalesced memory access. If global memory accesses are not coalesced, then the performance of global memory access decreases considerably, at worst to one sixteenth [8]. Moreover, "Bank conflict" is an important consideration when we use shared memory as shared memory is divided into equally-sized 16 memory modules, called banks, which can be accessed simultaneously. A bank conflict happens if n threads within a half-warp attempt to access same bank at the same time. Here, the performance of shared memory access decreases to one n-th [7]. To avoid this problem, we have to be careful with the size of array elements that sit on shared memory and how to access them. We should also consider memory alignment when we pass data between different types of memory. We will follow up this issue in more detail in Section IV-B and IV-D.

B. SIMD elements in SPMD

There are 30 multi-processors (MPs) in a GT200b with GeForce GTX 275, GTX 285 and GTX 295 and each MP has 8 Stream Processors(SPs). In CUDA, a thread block means a group of threads that are executed on the same MP and thus can be synchronized by the "__synchthreads()" statement. A thread block consists of some smaller groups of threads called warps. Threads in the same warp share the same instruction stream and executed simultaneously. Therefore, the GPU architecture looks like an SPMD, but threads within the warp are always synchronized like in SIMD [9]. This is because there is usually no need to use the "__syncthreads()" statement to synchronize all threads in a thread block if each thread doesn't access the shared memory or global memory that other threads within other warps access.

IV. Implementation

There are many SW algorithm implementations on various parallel platforms [10] [11] [12] [13] [14] [15], and one of the fastest CUDA-compatible GPU implementations was presented by Liu et al. in [16]. This tried two types of implementations: "Inter-task parallelization" and "Intra-task parallelization". The "Inter-task parallelization" is a method whereby each thread processes one alignment matrix between a database sequence and a query sequence. By contrast, the "Intra-task parallelization" is a method whereby a thread block processes one alignment matrix between a database sequence and a query sequence. In our implementation, we used one warp to process one alignment matrix between a database sequence and a query sequence. To use this technique, we have reduced the resource usage per thread, the frequency of global memory access per matrix calculation, and the frequency of the synchronization statement call.

A. Calculation method

As already discussed, we used one warp to process one matrix. Since we set the number of threads per thread block to 256, a total of eight matrices are processed by one thread block. This number depends on the target GPU architecture. In general, if the number of threads per thread block is too small or too large, the performance decreases significantly. Moreover, we need to set the number of threads per thread block as a multiple of 32 [7].

Furthermore, the cost of conditional branch is quite high in GPU, especially when some threads within the same warp take different instruction flows as the threads within warp are synchronized like in a SIMD manner i.e. all threads within a warp have to take the same instruction flow in reality. As shown in Section II, the core of the SW algorithm itself is not very complicated, and consist of two nested loops with simple additions and a maximum function of 2 values. To reduce conditional branches, we used a traditional approach (other optimization methods have been proposed e.g. in [12] [14].). In our approach, each sequence is padded with null entries(NEs) to have the overall length of a multiple of 32, and the resulting alignment matrix is divided into parallelograms as shown by Figure 2. Regions where padded null entries exist lead to decreased performance because the cost of ineffective calculation increases.

The transfer of data, H and F, between each parallelogram is done using global memory and the transfer of data within the parallelogram is done by using shared memory. To use shared memory, we divided each parallelogram to 32-column sub-parallelograms as shown in Figure 3. The pseudo code for the calculation of the data from a sub-parallelograms (the most inner loop calculation) is shown in Figure 4. The prefixes "r", "s", "t" and "c" relate to register, shared memory, texture memory and constant memory, respectively. r_Max is the maximum r_H score that each thread processes. Data that have to be passed to the downside large parallelogram, H and F are the first 32 elements of s_H and s_F, stored in global memory. Data that have to be passed to the next sub-parallelogram are the second 32 elements of s_H and s_F stored in shared memory. r_H_diag, r_Max and r_E are also passed to the next sub-parallelogram. tid is the unique ID of each thread and r_score is the score of the substitution matrix. Note that we do not use global memory within the most inner loop as it is slower than shared memory and

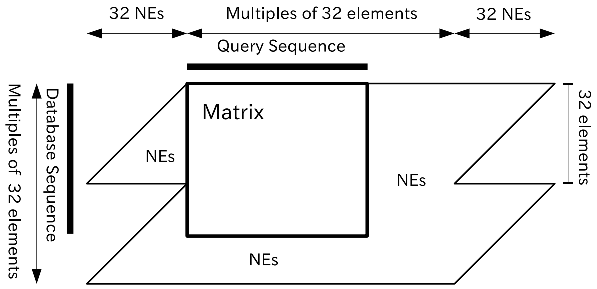

Figure 2. Region of calculation of the SW kernel and padding entries of query and database sequences. The square that is enclosed by a heavy line shows alignment matrix.

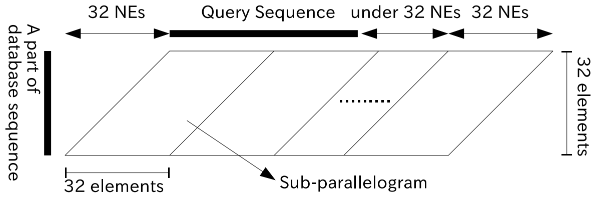

Figure 3. Dividing parallelogram to sub-parallelogram. The most inner loops in the SW kernel calculate each sub-parallelogram.

registers. After calculating all sub-parallelograms, we can get the alignment score by returning the maximum r_Max of each thread within the warp to the host.

The data that is passed to the next sub-parallelogram (r_E and r_H_diag) propagate in the lateral direction. Thus, it can be held by registers (since each thread process data laterally and this data is local to each thread).

B. Shared memory usage and occupancy

The GPU has 16 KB shared memory and 16 K registers per thread block. We can use all of these resources, but we have to consider the "Occupancy" [7] [8]. Occupancy is the ratio of the number of active warps to the maximum number of warps that each multi-processor can process. The maximum number of warps is determined by GPU hardware. In the case of GT200b, it is 32 warps. The number of active warps is determied by 3 factors, number of threads per MP, register usage and shared memory usage. The register and shared memory usages can be obtained by compiling results. A high occupancy means that are making the hardware busy. So this metric is very important for performance measurement. To get high occupancy, we have to reduce the amount of shared memory and register usage. In our implementation, there are 256 threads per thread block and we use 6216 bytes of shared memory per thread block and 24 registers per thread. Therefore the occupancy is 0.5, the number of active thread blocks in each MP is 2. As a result of this, we determined the number of thread blocks per kernel call to be 60 to make all multiprocessors busy.

```
for $i = 0$ to 31 do
  // Load score
  $r\_query\_idx =$
  $s\_query[tid + i + r\_query\_offset]$
  $r\_score =$
  $t\_score[r\_query\_idx][r\_subject\_idx]$
  // Calculate F
  $r\_F = max($
  $s\_F[tid + i] - c\_Gap\_extend,$
  $\_H[tid + i] - c\_Gap\_init)$
  // Calculate H and Max
  $r\_H =$
  $max(0, r\_E, r\_F, r\_H\_diag + score)$
  $r\_H\_diag = s\_H[tid + i]$
  $r\_Max = max(r\_H, r\_Max)$
  // Store H and F
  $s\_H[tid + i] = r\_H$
  $s\_F[tid + i] = r\_F$
  //Calculate next E
  $r\_E =$
  $max(r\_E - c\_Gap\_extend,$
  $r\_H - c\_Gap\_init)$
end for
```

Figure 4. The pseudo code for the most inner loop of SW kernel, when there are 32 threads per warp. The r_E, r_H_diag, s_H and s_F are initialized to 0 at the start of kernel execution.

In previous SW implementation studies, it was revealed that the maximum bit width of H, F and E is under 16 bits in most real world applications [15]. In that case, we can use short type variables (i.e. 16 bits) to store these values. This would decrease memory I/O traffic and hence improves performance. However, this also brings about memory bank conflicts if we use shared memory in a straightforward way as shown in Figure 5 [7]. We can easily pack H and F to 32-bit variables as there are some vector types like a "short2" in CUDA API.

C. Using multiple GPUs

In GTX295, there are two GPU cores in one module. Additionally, some motherboards have multiple PCI-Express x16 bus interfaces, hence allowing for multiple GPUs in one node. To use multiple GPUs in one process, we used OpenMP. A straightforward gain is achieved by dividing the database into multiple chunks and allocating each chunk to one of the GPU cores to process them in parallel.

Since the execution time depends on the length of the query sequence and database sequence, it is very important to balance the execution time between parallel GPUs to achieve high efficiency. For this, we divided the database into equal-length subsets. An easy way to do this dynamically is to assign the longest sequence to the GPU that has the shortest total length of the already assigned sequences. This

```
// (a)
__shared__ short s_H[8][64];
__shared__ short s_F[8][64];
// (b)
__shared__ short2 s_HF[8][64];
// (c)
__shared__ short s_HF[8][128];
// (d)
__shared__ int s_H[8][64];
__shared__ int s_F[8][64];
```

Figure 5. Examples of shared memory definition. (a): A bank conflict occurs because two adjacent elements e.g. s_H[0] and s_H[1] sit in the same bank with different threads attempting to access them at the same time. (b): There is no bank conflict here when some threads access each element of the array. (c): Interleaving H and F within a array, there is no bank conflict here and there is no need for packing. (d): There is no bank conflict here, and there is no need for packing. However, a larger memory is needed compared to the other definitions.

does not guarantee the optimal distribution but resulting imbalance in subset sizes is negligible for large databases (490,000 sequences in our implementation).

D. Other optimization issues

In addition to the aforementioned optimization techniques, this section presents further optimizations we used. First, we used loop unrolling, which is a very well-known optimization technique that is effective in reducing conditional branching overheads [17].

The second optimization technique used is optimized reduction used to find the maximum element of the alignment matrix to be returned to the host. The optimized reduction method that we used is described in [9]. In it, reduction is achieved with no synchronized statement which improves its execution time considerably.

The third optimization technique used is concurrent memory copy and execution. To use this technique, we separated the calculation sequence to three steps. First, the kernel call step which is the core calculation task executed on the GPU and is the most time consuming step. The results of kernel call are stored in global memory. Second, data is transferred from global memory to the host memory. In order to parallelize data copy and kernel execution, we used the Stream asynchronous memory copy statement and management organization [7]. For this, a double buffer is used in the global memory whereby read/write from/to the buffer is toggled at each iteration.

Third, data loaded from global memory to the host is sorted and tied up with the sequence names in the database file. Here again, we use a double buffer on the host to allow for concurrent execution of data sorting and data loading from global memory.

V. Results and Evaluation

The cell updates per second (CUPS) is a commonly used measure for SW execution performance as it normalises in terms of the database and query sequence sizes. Given a query sequence Q and a sequence database D, the Giga CUPS (GCUPS) is defined as below:

$$GCUPS = \frac{|Q| \times |D|}{t \times 10^9}$$

where $|Q|$ represents the length of the query sequence, $|D|$ the total length of the database sequences, and t means the elapsed time in seconds. In this paper, the elapsed time t includes the transfer time of the query sequence from the host to the GPU global memory, the calculation time of the SW algorithm scores on the GPU, result data transfer time from global memory to host, and finally score sorting on the host. The performance of our implementation is evaluated by 12 query sequences with lengths ranging from 143 to 567 (see Table I). These sequences include query sequences that are widely used in the literature to test sequence alignment algorithms, including Striped Smith-Waterman [10] and other Smith-Waterman implementations [11] [16]. All queries were run against Swiss-Prot 57.6 which comprises 174,780,353 amino acids in 495,880 sequence entries. Blosum50 scoring matrix was used with a gap penalty of 10-2k. GPU implementations were carried out on a GTX 295 graphics card, which has 30 SPs and 896 MB RAM per GPU, installed on a PC with a Core 2 Duo E7200 2.53 GHz processor, 2 GB DDR2 800 MHz, and running Cent OS 5.0. This graphics card has a core frequency of 576 MHz and a memory clock of 999 MHz.

A. Optimization techniques

In this section, we evaluate further optimization techniques as shown in Section IV-B and IV-D. Note that the shared memory definition used is the method (d) in Figure 5.

First, the effects of the loop-unrolling and concurrent memory copy techniques are shown by Table I. We unrolled 16 iterations resulting in a loop of 2 iterations, and register usage increased from 20 to 22. Both optimization techniques worked well. By unrolling the most inner loop of the SW algorithm, we got 17 % speed up in execution time. Additionally, by using concurrent copy and execution, we achieved an extra 9 % speed-up of the SW execution. Overall, we can get over 26 % speed up by using both of techniques. Because of the use of loop-unrolling, the amount of register usage increased, but this did not affect the occupancy in this instance.

Table II shows the performance of the four definitions of shared memory data shown in Figure 5. This shows method (d) to be the fastest. Method (a) which has the bank conflict problem is the slowest, whereas methods (b) and (c) are in the same range. There is no bank conflict in methods (b) and

Table I

Performance Evaluation of loop unrolling and concurrent memory copy using Blosum50 scoring matrix with a penalty of 10-2k.The performance is in GCUPS.

Queries	Length	Default	Unrolling	Memory copy	Both	Total Speed-up[%]
P02232	144	7.17	8.30	7.71	9.04	26.18
P01111	189	8.38	9.72	8.94	10.48	25.13
P05013	189	8.38	9.72	8.94	10.48	25.14
P09488	218	8.64	10.03	9.15	10.73	24.23
P14942	222	8.79	10.22	9.31	10.92	24.25
P00762	246	8.83	10.27	9.30	10.91	23.58
P10318	362	9.45	11.02	9.81	11.52	21.90
P07327	374	9.77	11.39	10.14	11.90	21.90
P01008	464	10.05	11.73	10.36	12.17	21.14
P10635	497	10.18	11.90	10.49	12.31	20.92
P25705	553	10.24	11.97	10.51	12.35	20.60
P03435	567	10.50	12.27	10.78	12.66	20.61

Table II

Performance Evaluation of shared memory using Blosum50 scoring matrix with a penalty of 10-2k.The performance is in GCUPS.

Queries	(a)	(b)	(c)	(d)
P02232	6.85	8.53	8.54	9.04
P01111	7.94	9.89	9.90	10.48
P05013	7.94	9.89	9.89	10.48
P09488	8.13	10.13	10.13	10.73
P14942	8.28	10.31	10.31	10.92
P00762	8.27	10.30	10.30	10.91
P10318	8.72	10.88	10.87	11.52
P07327	9.01	11.23	11.23	11.90
P01008	9.21	11.49	11.49	12.17
P10635	9.32	11.62	11.62	12.31
P25705	9.34	11.65	11.64	12.35
P03435	9.58	11.94	11.94	12.66

Table III

Performance Evaluation using swiss-prot release 57.6 and Blosum50 scoring matrix with a penalty of 10-2k.The performance is in GCUPS.

Queries	Length	1 GPU	2 GPUs	4 GPUs
P02232	144	9.09	17.26	29.46
P01111	189	10.53	19.98	34.44
P05013	189	10.53	19.98	34.42
P09488	218	10.78	20.45	35.44
P14942	222	10.97	20.83	36.09
P00762	246	10.96	20.84	36.23
P10318	362	11.57	22.01	38.78
P07327	374	11.96	22.76	40.07
P01008	464	12.21	23.25	41.19
P10635	497	12.36	23.54	41.19
P25705	553	12.39	23.61	41.97
P03435	567	12.71	24.22	43.05

(c), so it would appear that reason behind the performance drop is not memory bandwidth, but rather the additional computing cost due to data packing or index calculations. Although methods (a), (b) and (c) are slower than method (d), resource usage is lower. In this instance, there is no need to reduce the amount of shared memory usage but methods (b) and (c) are better than method (d) when this is needed.

B. Evaluation of SW implementation

The implementation results are shown in Table III. For a single GPU implementation, we achieved the highest performance of 12.71 GCUPS, and the lowest performance of 9.09 GCUPS. For a quad-GPU implementation, we achieved the highest performance of 43.05 GCUPS, and the lowest performance of 29.46 GCUPS. The gap penalty has no effect on performance. Next, we compare the performance of our implementation with CUDASW++-2.0b2 which is one of the fastest SW implementations on CUDA-compatible GPUs [16]. The evaluation is performed with the Swiss-Prot database release 57.6, using Blosum50 scoring matrix with a gap penalty of 10-2k. The results are shown in Figure 6. Apart from the case of query P02232, our implementation is clearly faster than the CUDASW++-2.0b2 (with the case of query P02232 achieving a similar performance). In the multiple GPU implementation of CUDASW++, the database sequences are transferd for each query sequence. By contrast, our implementation transfers the database sequences just one time. It is main the reason for the relatively large performance difference shown on multiple GPU implementation.

Finally, we compare the performance of our implementation with several recent CPU-based implementation by Farrar [10] [12]. Farrar reported the performance of a SW implementation on a number of processors including Xeon and Cell B.E. We evaluated the performance of Farrar's implementation on two further processors, the first using a PC with a Core 2 Duo E7200 2.53 GHz processor, 2 GB DDR2 800 MHz SDRAM running CentOS 5.0, and the second using a PC with a Core i7 CPU 920 2.67 GHz, 4 GB DDR3 1066 MHz running the CentOS 5.0. Comparison results are shown in Figure 7. The data of Xeon and Cell B.E. implementations were taken from Farrar's paper [12]. In the case of a single GPU implementation, our

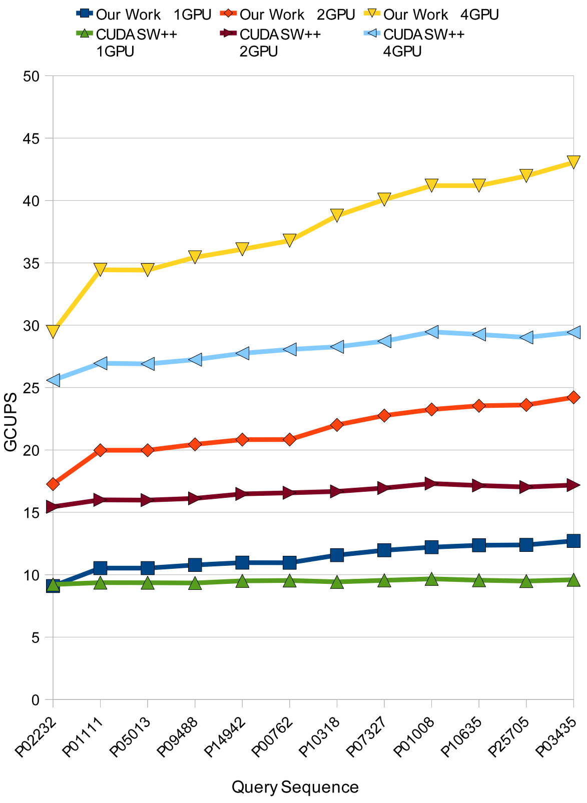

Figure 6. Performance comparison between our implementation and CUDASW++-2.0b2 using swiss-prot 57.6 and Blosum50 scoring matrix using a penalty of 10-2k.

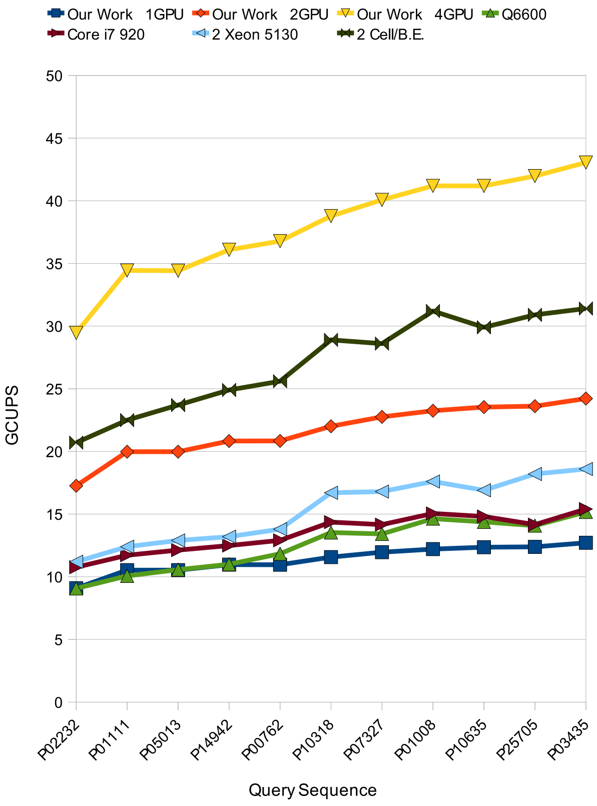

Figure 7. Performance comparison between our implementation and the striped Smith-Waterman implementation using Blosum50 scoring matrix with a penalty of 10-2k. We used all CPU cores.

implementation is slower than CPU-based implementations for all query sequences, whereas the multiple GPU-based implementations are faster than the CPU-based implementations for all query sequences.

VI. Conclusion

In this paper we have presented a novel technique for the implementation of the Smith-Waterman algorithm on CUDA-compatible GPUs.

Compared with previous GPU implementations that calculate the alignment matrix by thread blocks or threads, our implementation focused on warp level synchronization to decrease the cost of synchronization. Central to this technique is a divide and conquer approach to alignment matrix calculation in which a whole pairwise alignment matrix is subdivided into parallelogram regions. This leads to the efficient calculation of the alignment matrix by 32 threads, and the reduction in the number of loads/stores to/from global memory. Shared memory data definition was very important in order to avoid bank conflicts as we access shared memory more frequently (instead of global memory). In addition, we showed that some other optimization techniques, namely loop-unrolling and double-buffering to use concurrent memory copy and execution, which are used widely for CPU code optimization, worked very well for GPUs. We also presented the performance of our SW implementation using multiple GPUs. When we use 4 GPUs, for instance, the performance is much higher than the Farrar's implementation that used two Cell B.E. processors (50 % more), at a much lower cost (75 % less). We could have used more GPUs to scale up the implementation even further if there were more PCI-Express slots on the motherboard.

Future work includes a comparison with other implementation platforms e.g. FPGAs, in addition to the optimization of sequence database sub-division for multiple GPUs for optimal load balancing. Finally, we note that the source code of our SW GPU implementation is available online for the wider public to download [18].

References

[1] T. M. Charalambous, P and A. Stamatakis, "Intial experiences porting a bioinformatics application to a graphics processor." *In Processings of 10th Panhellenic Conference on Informatics.*, 2005.

[2] T. Smith and M. Waterman, "Identification of common molecular subsequences," *J Molecular Biology*, vol. 147, pp. 195–197, 1981.

[3] O. Gotoh, "An improved algorithm for matching biological sequences," *J Mol Biol*, vol. 162, pp. 707–708, 1982.

[4] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basical local alignment search tool," *J Mol Biol*, vol. 215, no. 3, pp. 403–410, 1990.

[5] S. Henikoff and J. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, p. 10915, 1992.

[6] "CUDA ZONE," http://www.nvidia.com/object/cuda_home.html.

[7] "NVIDIA CUDA programming guide 2.3." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[8] "NVIDIA CUDA Best Practices Guide 2.3." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf

[9] "NVIDIA Optimizing Parallel Reduction in CUDA." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[10] M. Farrar, "Striped smith-waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[11] A. Szalkowski, C. Ledergerber, P. Krahenbuhl, and C. Dessimoz, "SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2," *BMC Research Notes*, vol. 1, p. 107, 2008.

[12] M. Farrar, "Optimizing smith-waterman for the cell broadband engine." [Online]. Available: http://farrar.michael.googlepages.com/SW-CellBE.pdf

[13] S. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[14] T. Rognes and E. Seeberg, "Six-fold speedup of smith-waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.

[15] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 4, pp. 561–570, 2009. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2008.2005314

[16] Y. Liu, D. 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. [Online]. Available: http://www.biomedcentral.com/1756-0500/2/73

[17] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[18] "SourceForge project page," http://en.sourceforge.jp/projects/nesw/.