Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU

Yuma Munekawa, Fumihiko Ino, *Member, IEEE,* and Kenichi Hagihara

***Abstract*— This paper describes a design and implementation of the Smith-Waterman algorithm accelerated on the graphics processing unit (GPU). Our method is implemented using compute unified device architecture (CUDA), which is available on the nVIDIA GPU. The method efficiently uses on-chip shared memory to reduce the data amount being transferred between off-chip memory and processing elements in the GPU. Furthermore, it reduces the number of data fetches by applying a data reuse technique to query and database sequences. We show some experimental results comparing the proposed method with an OpenGL-based method. As a result, the speedup over the OpenGL-based method reaches a factor of 6.4 when using amino acid sequence database. We also find that shared memory reduces the amount of data fetches to 1/140, providing a peak performance of 5.65 giga cell updates per second (GCUPS). This performance is approximately three times faster than a prior CUDA-based implementation.**

I. Introduction

THE Smith-Waterman (SW) algorithm [1] is a well-known method for finding the optimal local alignment between two sequences. This algorithm is used by biologists to search meaningful sequences in biological databases. However, it requires a large amount of computation due to its high computational complexity, which is proportional to the product of the length of two sequences. Thus, some acceleration methods are needed to apply this algorithm to real databases, such as SWISS-PROT [2] and GenBank [3], which rapidly grow the data size.

Heuristic methods solve the alignment problem more quickly than exact methods. For example, BLAST [4] and FASTA [5] are now widely used in many research projects, because they are up to 40 times faster than a straightforward implementation of the SW algorithm [6]. However, heuristic methods have a problem of sensitivity.

Accordingly, many researchers are trying to accelerate the SW algorithm using various computing systems, aiming at providing a successful solution that is not only fast but also sensitive. For example, Manavski et al. [6] show a fast parallel implementation running on the graphics processing unit (GPU) [7], namely a commodity chip designed for acceleration of visualization applications. They implement the algorithm using compute unified device architecture (CUDA) [8], which is a development framework for performing general-purpose computation on the nVIDIA GPU. Their implementation demonstrates good acceleration results over the CPU implementation. However, it can be improved to fully utilize memory resources available on the GPU. For example, on-chip shared memory can be exploited to save the bandwidth between the GPU and off-chip memory.

In this paper, we present a design and implementation of the SW algorithm on the CUDA-compatible GPU, aiming at making it clear how memory resources should be used to achieve a further acceleration of this memory-intensive algorithm. Our method uses on-chip shared memory to reduce the data amount being transferred between off-chip memory and processing elements in the GPU. Furthermore, it also reduces the number of data fetches by applying a data reuse technique to query and database sequences.

The rest of the paper is organized as follows. We begin in Section II by introducing related work. Section III features the CUDA framework and Section IV gives an overview of the SW algorithm. Section V then describes our CUDA-based method and Section VI shows experimental results. Finally, Section VII concludes the paper.

II. Related Work

To the best of our knowledge, Liu et al. [9] develop the first implementation running on the GPU. Since their work is done before the dawning of CUDA, they employ the OpenGL library [10], namely a graphics library, to implement the SW algorithm on the GPU. They show how the algorithm can be mapped onto the graphics pipeline. Using an nVIDIA GeForce 7800 GTX card, their implementation achieves a 10-fold speedup over SSEARCH [5], a heuristic implementation of the SW algorithm running on the CPU. It provides a peak performance of 0.67 giga cell updates per second (GCUPS) at a query of length 4092.

Manavski et al. [6] present a CUDA-based implementation for the SW algorithm. Their performance reaches a peak of 3.6 GCUPS using two GeForce 8800 cards, but it is not clear whether this performance includes the data transfer time needed before/after GPU execution. We think that the design can be improved to utilize the full resources on the GPU, because they partly use local memory, which is the slowest memory resource on the GPU card. A similar work but with a heuristic algorithm is presented by Schatz et al. [11]. Their CUDA-based implementation achieves a 3.5-fold speedup over a CPU implementation.

Farrar [12] proposes a CPU-based SW implementation optimized using SSE instructions. These instructions are originally designed to accelerate multimedia applications by

Manuscript received July 2, 2008. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(2)(20240002), Young Researchers (B)(19700061), and the Global COE Program "in silico medicine" at Osaka University.

Y. Munekawa, F. Ino, and K. Hagihara are with the Graduate School of Information Science and Technology, Osaka University, 560-8531 Osaka, Japan (e-mail: {y-munekw,ino}@ist.osaka-u.ac.jp).

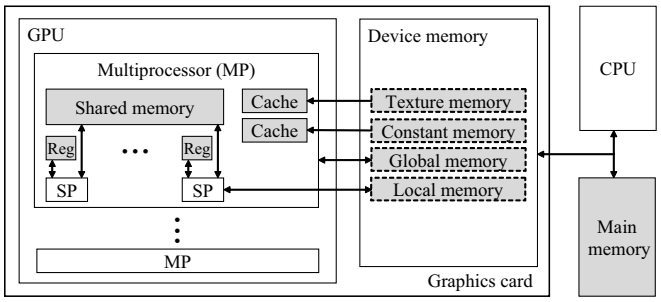

Fig. 1. Hardware model in CUDA. SP and reg denote stream processor and register, respectively.

TABLE I

MEMORY RESOURCES AVAILABLE IN CUDA. LATENCY IS PRESENTED IN CLOCK CYCLES. THE CACHE WORKING SET IS 8 KB PER MP.

Memory	Capacity	Cache	Latency	Access
Register	8192 per MP	N/A	1	R/W per thread
Shared	16 KB per MP		1*	R/W per block
Constant	64 KB	Yes	1–600**	R
Texture	512+ MB (shared)			
Global		No	400–600***	R/W
Local			400–600	R/W per thread

*: Accesses will be serialized if bank conflicts [8] happen
**: Latency depends on the locality of data accesses
***: Accesses can be coalesced if satisfying memory alignments [8]

performing single-instruction, multiple-data (SIMD) computation. The implementation delivers a peak performance of 3.0 GCUPS on a 2.0 GHz Core 2 Duo processor.

Zhang et al. [13] propose a field programmable gate array (FPGA) solution for the SW algorithm. Their solution provides a peak of 25.6 GCUPS, which is 250 times faster than a CPU version running on a 2.2 GHz Opteron processor. One drawback of FPGA solutions is the cost of expensive FPGAs because FPGAs are not so widely used as compared with GPUs, which have a strong market in the entertainment area. A similar work is presented by Li et al. [14].

III. COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

CUDA [8] is a programming environment for writing and running general-purpose applications on the nVIDIA GPU. This environment allows us to efficiently run highly-threaded applications on the GPU, regarding it as a massively parallel machine that computes threads on hundreds of processing elements. The kernel, namely the program running for every thread but each with a different thread ID, can be written in the C-like language.

Figure 1 summarizes the hardware model in CUDA. This model mainly consists of two parts: the GPU itself and off-chip device memory. The GPU consists of several multiprocessors (MPs), each including a set of stream processors (SPs) and shared memory, which is useful to save the memory bandwidth between SPs and device memory. During kernel execution, each thread is assigned to an SP in order to compute threads in a SIMD fashion. Thus, every SP within the same MP executes the same instruction but operates on a different thread at every clock cycle.

When writing CUDA programs, threads have to be structured into a hierarchy to batch them to MPs. In this hierarchy, a group of threads is called as a thread block. This hierarchical structure allows threads within the same thread block to share data in shared memory, namely fast on-chip memory. However, threads belonging to different thread blocks are not allowed to share data, because thread blocks are independently assigned to each of MPs. Therefore, developers must write their code such that there is no data dependence between different thread blocks.

Table I summarizes a hierarchy of memory resources provided by CUDA. Shared memory is as fast as registers while device memory takes 400 to 600 clock cycles to access non-cached data. However, the capacity of shared memory is limited by 16 KB per MP. In contrast, recent high-end GPUs have at least 512 MB of device memory, which can be used as constant, texture, global, and local memory. Constant memory and texture memory are cached but not writable by SPs. They are writable by the CPU in advance of kernel invocation. Texture memory has a larger space than constant memory.

The remaining local memory and global memory are not cached but writable by SPs. Global memory is the only space that can be used to send computational results back to the CPU. Note here that satisfying memory alignment requirements [8] is important to allow memory accesses to be coalesced into a single access. This memory coalescing technique increases the effective memory bandwidth by the order of magnitude. Local memory is implicitly used if the CUDA compiler consumes the register space. Since local memory cannot be accessed in a coalesced manner, it is better to explicitly use global memory instead of local memory to avoid such an implicit, inefficient use.

IV. SMITH-WATERMAN ALGORITHM

The SW algorithm [1] is a dynamic programming method for obtaining the optimal local alignment between two sequences. This algorithm finds similar segments in two steps: (1) computation of a similarity matrix H and (2) backtracing from the matrix cell with the highest score. As compared with the backtracing step, the first step involves computation by more than an order of magnitude. Therefore, we focus on this bottleneck step, which we parallelize on the GPU. Actually, backtracing can be quickly done on the CPU [9].

Let A denote a query sequence $a_1 a_2 \dots a_n$ of length n. Let B denote a database sequence $b_1 b_2 \dots b_m$ of length m to be compared with the query sequence A. The algorithm then computes an $n \times m$-cell matrix H to obtain the similarity for any pair of segments. Let $E_{i,j}$ and $F_{i,j}$ be the maximum similarity involving the first i-th symbols in A and the first j-th symbols in B, respectively. The maximum similarity $H_{i,j}$ of two segments ending in a_i and b_j, respectively, is then recursively defined as follows:

$$H_{i,j} = \max\{0, E_{i,j}, F_{i,j}, H_{i-1,j-1} + W(a_i, b_j)\}, \quad (1)$$

where $W(a_i, b_j)$ represents a scoring matrix. Similarities $E_{i,j}$ and $F_{i,j}$ are given by

$$E_{i,j} = \max\{H_{i,j-1} - G_{init}, E_{i,j-1} - G_{ext}\}, \quad (2)$$
$$F_{i,j} = \max\{H_{i-1,j} - G_{init}, F_{i-1,j} - G_{ext}\}, \quad (3)$$

where G_{init} and G_{ext} are penalties for opening a new gap and for extending an existing gap, respectively. The values for $H_{i,j}$, $E_{i,j}$, and $F_{i,j}$ are defined as zero if $i < 1$ or $j < 1$.

In general, the scoring matrix is experimentally determined as $W(a_i, b_j) > 0$ if $a_i = b_j$ and $W(a_i, b_j) < 0$ if $a_i \neq b_j$. The penalties are called as linear gap penalties if $G_{init} = G_{ext}$. Otherwise, they are affine gap penalties.

V. Proposed Method

The parallelization strategy we use for the proposed method is based on that employed in Liu's OpenGL-based method [9]. For better understanding of the proposed method, we first show the strategy then describe our memory assignment scheme and data reuse scheme.

A. Parallelization Strategy

Eqs. (1)–(3) indicate that matrix cell $H_{i,j}$ depends on its left neighbor $H_{i,j-1}$, upper neighbor $H_{i-1,j}$, and upper-left neighbor $H_{i-1,j-1}$. Therefore, cells on the k-th antidiagonal depend on the $(k-1)$-th and the $(k-2)$-th antidiagonals, where $1 \leq k \leq m+n-1$. In contrast, there is no dependence between any cells on the same antidiagonal. Thus, matrix H for a pairwise alignment must be serially computed from the first to the last antidiagonal but each antidiagonal can be computed in parallel with a maximum parallelism of n.

This dependency implies that we do not need the entire matrix H to find the cell with the highest score. Instead of the entire matrix, we allocate memory space only for three antidiagonals, and then reuse them for all $1 \leq k \leq m+n-1$, because the k-th (current) antidiagonal can be computed from the last two antidiagonals. Thus, as same as Liu's method [9], our method does not store the entire matrix H. Using the three antidiagonals at each iteration of the k loop, our method computes the highest score S_i for every column i in matrix H, where $1 \leq i \leq n$.

The parallelism mentioned above is not so high as compared with the length of sequences. To exploit more parallelism for higher efficiency, Liu et al. extend this strategy for N pairwise alignments. Since the query sequence can be independently aligned to different database sequences, they perform embarrassingly parallel computation to compute N matrices at the same time.

Figure 2 shows how we adapt the strategy to the CUDA framework. Our method assigns a pairwise alignment to each of thread blocks in order to perform embarrassingly parallel computation of N pairwise alignments. On the other hand, each of n threads in a thread block is responsible for computing a column in matrix H. Thus, a kernel invocation solves N pairwise alignments using nN threads. The kernel is iteratively launched with varying the database sequences until reaching the last entry in the database.

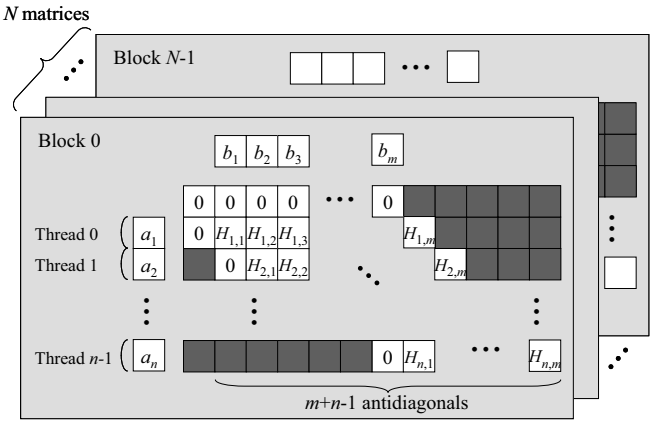

Fig. 2. Parallelization of matrix computation for N pairwise alignments. Matrix cells on the same antidiagonal are illustrated here in a row. Given a query sequence $a_1 a_2 \ldots a_n$ and N database queries, cells on the k-th antidiagonals of N matrices are computed in parallel, where $1 \leq k \leq m+n-1$. Thus, matrix cells are computed by nN threads from left to right in this figure.

Note here that it is important to sort database sequences by their length m before alignments [9]. Otherwise, it will cause a load imbalance problem because every thread block has a different number $m+n+1$ of iterations to fill the matrix H. The sorting procedure will balance the workload between thread blocks.

B. Memory Assignment Scheme

In our method, each of matrix cells on the $(k-1)$-th antidiagonal are accessed by multiple threads that compute cells on the k-th antidiagonal, where $1 \leq k \leq m+n-1$. For example, cell $H_{i-1,j}$ is needed to compute $H_{i,j}$ and $H_{i-1,j+1}$ on the $(i+j-1)$-th antidiagonal. Therefore, we can reduce the number of data fetches from device memory if such commonly accessed data is stored in shared memory. To reduce this, our method stores the $(k-1)$-th (last computed) antidiagonal in shared memory. On the other hand, we store the k-th (current) and the $(k-2)$-th (second last) antidiagonals on registers, because they are accessed only by the responsible thread. The highest scores $S_1, S_2, \ldots, S_n$ are stored in global memory to send them back to the CPU.

Since all threads refer the same query sequence A, we incorporate cache effects by storing the query A as character data in constant memory. On the other hand, we select texture memory for database sequences, because they consume more than 100 MB of memory space. Since N database sequences have different lengths, each of length m is stored in constant memory as an integer value. Thus, the kernel consumes $4N+n$ bytes of constant memory in total. We currently use $N = 8192$ according to the capacity of shared memory (Table I). Similarly, a thread block requires $4n$ bytes of shared memory and at least $2n$ registers for antidiagonals. Due to the capacity of shared memory and registers, the kernel requires $n < 4096$ to run.

With respect to the scoring matrix W, it can be stored in registers, constant memory, or texture memory according to

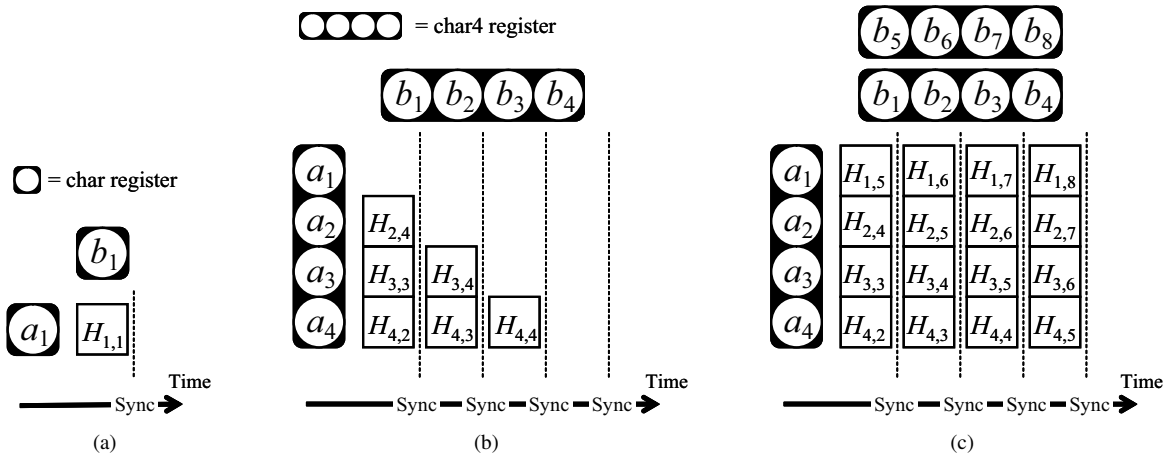

Fig. 4. Data reuse scheme with vectorization and lookahead techniques. (a) the naive scheme, (b) the vector scheme, and (c) that with lookahead, each showing matrix cell(s) computed by a single thread at a single iteration of the k loop. The naive scheme can be vectorized to reduce the number of texture fetches and to achieve a higher parallelism. Lookahead of the database sequence, $b_5b_6b_7b_8$ in this example, allows us to compute 4×4 matrix cells per fetch. Without this lookahead, the computation is restricted to only six cells located in the lower triangular area.

```
Input: query sequence __constant__ char query[n],
       N database sequences DBtex[m][N]
Output: highest scores S[nN] for every column of N matrices
 1.  __shared__ int Ds[n] // (k − 1)-th antidiagonal
 2.  Ds[tid] := −1; // tid: thread ID
 3.  dia := 0; // (k − 2)-th antidiagonal
 4.  que := query[tid];
 5.  for k := 0 to m + n − 2 do // for all antidiagonals
 6.    idx := k − tid;
 7.    if (idx < 0) or (idx > n) then ; // out of cells
 8.    else
 9.      sub := DBtex[idx][bid]; // bid: block ID
10.      if (que == sub) then W := 2; // compute W(a_i, b_j)
11.      else W := −1;
12.      end if
13.      H := max(max(max(0, dia + W),
               Ds[tid]), Ds[tid + 1]); // compute Eq. (1)
14.      score := max(score, H); // update the highest score
         // update the diagonals for the next iteration
15.      dia := Ds[tid + 1] + 1;
16.      Ds[tid + 1] := H − 1;
17.    end if
18.    __syncthreads(); // synchronization
19.  end for
20.  S[bid ∗ size + tid] := score; // size = 16⌈n/16⌉
```

Fig. 3. Pseudocode of naive version of proposed kernel. This naive kernel uses shared memory but does not reuse data for the sake of simplicity. It assumes a linear gap penalty: $G_{init} = G_{ext} = 1$. The thread $\langle tid, bid \rangle$ is responsible for the $(tid + 1)$-th column of matrix H computed for the $(bid + 1)$-th database sequence.

the matrix size. We currently encode the scoring matrix W in the kernel code.

Figure 3 shows a pseudocode of the proposed kernel. Let tid and bid denote the thread ID and the block ID, respectively. This kernel is invoked for every thread $\langle tid, bid \rangle$, where $0 \leq tid < n$ and $0 \leq bid < N$. In this kernel, the antidiagonals of matrix H are swept by the k loop at line 5. Note here that every thread does not always update its responsible column at each iteration. For example, only the first threads $\langle tid, bid \rangle = \langle 0, * \rangle$ are allowed to compute the antidiagonals of N matrices when $k = 1$. Such flow control is realized by the branch instruction at line 7. Threads that are allowed to compute a matrix cell $H_{i,j}$ fetch the symbol b_j from the database texture at line 9, and then compute the cell $H_{i,j}$ at line 13. After this, they update the responsible antidiagonals at lines 15 and 16 for the next iteration. Since this update is done using shared memory, synchronization is needed before proceeding to the next iteration. Finally, every thread copies the highest score S_{tid+1} of its responsible column $tid + 1$ to global memory at line 20. The highest scores $S_1, S_2, \ldots, S_n$ are written in a coalesced manner in order to use the full memory bandwidth. This memory coalescing can be realized if all of the first threads $\langle 0, * \rangle$ write the responsible score to an offset address of a multiple of 16. Therefore, we use $size = 16\lceil n/16 \rceil$ at line 20.

C. Data Reuse Scheme

Our data reuse scheme aims at reducing the number of texture fetches needed for matrix computation. To realize this, we pack the sequences into vector data formatted in type char4 [8], as shown in Fig. 4(b). Accordingly, the modified kernel assigns four succeeding columns to each thread, performing per-vector computation. Therefore, a thread block contains $\lceil n/4 \rceil$ threads after vectorization. Since thread blocks are currently allowed to have a maximum of 512 threads per block [8], the vector kernel requires $n \leq 2048$ to run.

The vector kernel fetches vector data instead of scalar data. One important point here is that it is better to incorporate a lookahead of database symbols to compute 4×4 cells at each iteration (see Fig. 4(c)). Otherwise, as shown in Fig. 4(b), the computation is restricted to only six cells because the responsible thread has not yet loaded the database symbols needed for that computation. Since four columns are computed at each iteration, the number of iterations reduces from $m + n - 1$ to $\lceil (m + n - 1)/4 \rceil$ if using the lookahead.

The data reuse scheme also contributes to reduce the amount of data being transferred from device memory. At each iteration, the naive kernel fetches a symbol b_j of the

TABLE II

PERFORMANCE COMPARISON WITH OPENGL-BASED METHOD [9]. EXECUTION TIME CONTAINS GPU TIME, CPU TIME, AND CPU–GPU TRANSFER TIME.

Query length	Execution time T (s)		Throughput P (GCUPS)	
n	Proposed	OpenGL	Proposed	OpenGL
63	2.96	10.25	1.93	0.56
127	3.38	15.93	3.40	0.72
191	3.98	21.74	4.34	0.80
255	4.66	27.24	4.95	0.85
319	6.04	33.30	4.78	0.87
383	6.67	39.68	5.20	0.88
447	7.57	46.21	5.35	0.88
511	8.19	52.78	5.65	0.88

database sequence to compute a matrix cell $H_{i,j}$. In contrast, the vector kernel with the lookahead technique fetches four succeeding symbols of the database sequence and computes 16 matrix cells per iteration. Therefore, the vector kernel reduces texture fetches by 75% as compared with the naive kernel. Furthermore, it also reduces the number of branch instructions at line 7 in Fig. 3, due to the less number of iterations.

VI. EXPERIMENTAL RESULT

We now show some experimental results to evaluate the performance of the proposed method. We compare it with the OpenGL-based method [9]. For experiments, we use a desktop PC equipped with a 3.0 GHz Core 2 Duo E6850 CPU, 4 GB main memory, and an nVIDIA GeForce 8800 GTX GPU with 768 MB device memory. All implementations run on Windows XP with driver version 169.21. The methods are implemented using Visual Studio 2005 and CUDA 1.1 [8]. We also use nVIDIA SDK 9.5 and C for graphics (Cg) 1.5 [15] for graphics libraries.

The experiments are conducted using query sequences of length n ranging from 63 to 511 amino acids. All queries are run against the SWISS-PROT database [2]. This database is approximately 121 MB in file size, containing $250,143$ ($= |B|$) entries with a total of 90,588,910 amino acids. Thus, the average length m_{avg} of database sequences is $m_{avg} = 362$. The database sequences are sorted according to length m in advance.

Table II shows the execution time T and the throughput P in GCUPS for query sequences with different length n, where $P = m_{avg}n|B|/T$. The execution time here contains the time T_1 spent on the GPU, that T_2 on the CPU, and that T_3 needed for data transfer between main memory and device memory. Alignments are carried out with a linear gap penalty $G_{init} = G_{ext} = 1$ and a scoring matrix W such that $W(a_i, b_j) = 2$ if $a_i = b_j$ and $W(a_i, b_j) = -1$ otherwise.

The proposed method is up to 6.4 times faster than the OpenGL method. This performance is equivalent to 5.65 GCUPS while the prior CUDA-based implementation [6] provides 1.85 GCUPS on the same single GPU card. Thus, we obtain a 3-fold speedup using the appropriate memory resource.

TABLE III

BREAKDOWN OF EXECUTION TIME FOR QUERY LENGTH $n = 255$.

Breakdown	Proposed (s)	OpenGL (s)
GPU time T_1	2.76	13.42
CPU time T_2	1.44	11.99
CPU–GPU transfer time T_3	0.46	1.83
Total T	4.66	27.24

TABLE IV

EFFECTIVE PERFORMANCE OF CUDA-BASED KERNEL AND OPENGL-BASED KERNEL [9].

Query length	Floating point (GFLOPS)		Effective BW (GB/s)	
n	Proposed	OpenGL	Proposed	OpenGL
63	42.45	11.14	1.28	47.46
127	61.38	12.75	1.86	54.34
191	65.67	13.37	1.99	56.97
255	66.12	13.60	2.00	57.98
319	55.21	13.44	1.67	57.27
383	57.49	13.29	1.74	56.66
447	56.50	13.04	1.71	55.59
511	58.29	12.68	1.77	54.02

Table III shows a breakdown of execution time T for $n = 255$, where the kernel achieves the highest performance (i.e. the best performance with respect to the GPU time T_1). Our method reduces both the GPU time and the CPU time. By comparing the GPU time, we can find that the CUDA-based kernel achieves a 4.9-fold speedup over the OpenGL-based kernel. This is mainly due to on-chip memory available in the CUDA framework. Since this memory cannot be explicitly used in the OpenGL framework, the OpenGL-based kernel has to store matrix H in textures. In contrast, CUDA allows us to explicitly use shared memory and registers in the kernel. Thus, the explicit use of on-chip memory directly reduces the amount of data transfer between device memory and SPs.

Another interesting point in Table III is that the CPU time and the CPU–GPU transfer time also contribute to reduce execution time T. Since the OpenGL-based method is implemented on the graphics pipeline, non-graphics GPGPU applications can suffer from graphics-related overheads, such as texture bindings and geometry computation. Such additional overheads increase the CPU time, as compared with the CUDA-based method. Due to the same reason, CUDA provides a faster data transfer than OpenGL. It should be noted here that CUDA applications does not always outperform OpenGL applications, because the OpenGL framework enables us to use graphics-oriented hardware components that are not available in CUDA.

We next analyze the efficiency of the kernel with respect to floating point performance and memory bandwidth. In this analysis, the number of floating point instructions is given by $8m_{avg}n|B|$, which is counted from the kernel. Similarly, the effective bandwidth is computed from the number of memory instructions needed for device memory access.

As shown in Table IV, the proposed method achieves at least 3.8 times higher floating point performance than the OpenGL-based method. However, the effective bandwidth of our method results in 1/37 of that of the OpenGL-

TABLE V

IMPACT OF DATA REUSE SCHEME IN TERMS OF KERNEL PERFORMANCE. THE PROPOSED METHOD DIFFERS FROM THE NAIVE METHOD IN USING THE DATA REUSE TECHNIQUE. PERFORMANCE IS MEASURED USING QUERY LENGTH $n = 255$.

Performance measure	CUDA		OpenGL
	Proposed	Naive	
GPU time T_1 (s)	2.76	4.15	13.42
Floating point (GFLOPS)	66.12	44.00	13.60
Effective BW (GB/s)	2.00	5.33	57.98

based method. These results indicate that the CUDA-based method moves the performance bottleneck from the memory bandwidth to the instruction issue rate. That is, it eliminates the bottleneck by on-chip memory, but it now suffers from computation. To confirm this, we measure the performance on a GeForce 9800 GTX card, which has 17% higher clock rate but 19% lower memory bandwidth than the 8800 GTX card. We then find that the CUDA-based method increases the throughput by 16% (a peak throughput of 6.58 GCUPS) but the OpenGL-based method decreases the throughput by 23% (a peak throughput of 0.68 GCUPS). Thus, the performance bottleneck can vary according to the underlying programming model though the same parallel algorithm is implemented on the same hardware.

In our method, the amount of data transfer between device memory and SPs is first reduced to 1/35 by the shared memory scheme and further reduced to 1/4 by the data reuse scheme. Thus, the proposed method reduces the amount of data fetches to 1/140 in total. CUDA allows us to increase the ratio of computation to memory access in order to overcome the bandwidth issue appeared in traditional OpenGL-based GPGPU applications.

Finally, we investigate the effectiveness of our data reuse scheme. We develop a naive scalar kernel that does not use the data reuse scheme. Table V shows the performance for query length $n = 255$. Our data reuse scheme reduces the GPU time by 33% and improves the floating point performance by 50%. In contrast, the scheme decreases the effective bandwidth by 62%. However, this is not a critical problem because the performance bottleneck has already moved from the memory bandwidth to the instruction issue rate, as we mentioned earlier. In fact, the naive kernel is three times faster than the OpenGL-based method though its effective bandwidth is merely 10% of the OpenGL-based method. Due to the computation demanding nature, we will obtain higher performance by applying optimization techniques to computation.

VII. CONCLUSION

We have presented a design and implementation of the SW algorithm on the CUDA-compatible GPU. Our method differs from the prior CUDA-based method [6] in terms of utilizing the full memory resources, such as shared memory and constant memory. It also reduces the number of instructions by applying a data reuse technique to query and database sequences.

The experimental results show that the proposed method achieves a peak performance of 5.65 GCUPS using a GeForce 8800 GTX card. This performance is equivalent to a 3.1-fold speedup over the prior CUDA-based method and a 6.4-fold speedup over an OpenGL-based method [9]. We also show that the performance of the CUDA-based method is limited by the instruction issue rate while that of the OpenGL-based method is limited by the memory bandwidth. Thus, the performance bottleneck varies depending on the underlying programming model. CUDA is useful to overcome the bandwidth problem appeared in traditional OpenGL-based GPGPU applications if we fully use the memory resources provided by CUDA.

One future work is to extend our implementation to deal with longer queries with a length of more than 2048. We are also planning to enhance the implementation by supporting affine gap penalties.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

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

[2] A. Bairoch and R. Apweiler, "The SWISS-PROT protein sequence data bank and its supplement TrEMBL," *Nucleic Acids Research*, vol. 25, no. 1, pp. 31–36, Jan. 1997.

[3] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler, "GenBank," *Nucleic Acids Research*, vol. 28, no. 1, pp. 15–18, Jan. 2000.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *J. Molecular Biology*, vol. 215, no. 3, pp. 403–410, Oct. 1990.

[5] W. R. Pearson, "Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms," *Genomics*, vol. 11, no. 3, pp. 635–650, Nov. 1991.

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

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[8] nVIDIA Corporation, "CUDA Programming Guide Version 1.1," Nov. 2007. [Online]. Available: http://developer.nvidia.com/cuda/

[9] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 9, pp. 1270–1281, Sep. 2007.

[10] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, 5th ed. Reading, MA: Addison-Wesley, Aug. 2005.

[11] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, no. 474, Dec. 2007, 10 pages.

[12] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, Jan. 2007.

[13] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform," in *Proc. 1st Workshop High-performance reconfigurable computing technology and applications (HPRCTA'06)*, Nov. 2007, pp. 39–48.

[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. 185, Jun. 2007, 7 pages.

[15] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 896–897, Jul. 2003.