PAPER *Special Section on Info-Plosion*

Accelerating Smith-Waterman Algorithm for Biological Database Search on CUDA-Compatible GPUs*

Yuma MUNEKAWA†, ***Nonmember***, **Fumihiko INO**†a), ***and*** **Kenichi HAGIHARA**†, ***Members***

SUMMARY This paper presents a fast method capable of accelerating the Smith-Waterman algorithm for biological database search on a cluster of graphics processing units (GPUs). Our method is implemented using compute unified device architecture (CUDA), which is available on the nVIDIA GPU. As compared with previous methods, our method has four major contributions. (1) The method efficiently uses on-chip shared memory to reduce the data amount being transferred between off-chip video memory and processing elements in the GPU. (2) It also reduces the number of data fetches by applying a data reuse technique to query and database sequences. (3) A pipelined method is also implemented to overlap GPU execution with database access. (4) Finally, a master/worker paradigm is employed to accelerate hundreds of database searches on a cluster system. In experiments, the peak performance on a GeForce GTX 280 card reaches 8.32 giga cell updates per second (GCUPS). We also find that our method reduces the amount of data fetches to 1/140, achieving approximately three times higher performance than a previous CUDA-based method. Our 32-node cluster version is approximately 28 times faster than a single GPU version. Furthermore, the effective performance reaches 75.6 giga instructions per second (GIPS) using 32 GeForce 8800 GTX cards.

key words: *Smith-Waterman algorithm, sequence alignment, acceleration, GPU, CUDA*

1. Introduction

In many bioinformatics organizations, biological databases are exponentially growing in size and number. For example, such a rapid growth can be seen in SWISS-PROT [2], which continuously updates a manually annotated protein (amino acid) sequence database over 20 years [3]. It now contains approximately 5×10^5 sequence entries, comprising about 1.7×10^8 amino acids. Using these databases, biologists are trying to identify structure, function and evolutionary relationships between sequences.

This identification process is usually assisted by sequence database search, which finds meaningful sequences in databases, according to the similarity between a query sequence and subject sequences in the databases. The similarity here can be computed by the Smith-Waterman (SW) algorithm [4], namely a well-known method for finding the optimal local alignment between two sequences. However, it requires a large amount of computation due to high computational complexity. Given a query sequence of length n and a subject sequence of length m, it takes $O(mn)$ time to perform a pairwise alignment between them. Thus, some acceleration methods are needed to use this algorithm for exponentially growing databases [2], [5].

Heuristic methods solve the alignment problem more quickly than exact methods. For example, BLAST [6] and FASTA [7] are widely used by researchers, because they are up to 40 times faster than a straightforward implementation of the SW algorithm [8]. However, heuristic methods have a problem of sensitivity. Accordingly, in order to develop a not only fast but also sensitive solution, many researchers are trying to accelerate the SW algorithm using various hardware platforms.

Manavski et al. [8] propose a fast method running on the graphics processing unit (GPU) [9], namely a commodity chip designed for acceleration of graphics applications. They implement the SW algorithm using compute unified device architecture (CUDA) [10], which is a development framework for accelerating general applications on the nVIDIA GPU. Although their implementation demonstrates higher performance than a CPU implementation [7], the acceleration is not enough to outperform an optimized implementation [11] that uses SSE instructions [12].

To the best of our knowledge, Munekawa et al. [1] present the first implementation that outperforms the optimized CPU implementation. They fully utilize memory resources available on the GPU. For example, their implementation exploits on-chip shared memory to save the bandwidth between the GPU and off-chip video memory. Similar approaches are reported in [1], [13], [14].

In this paper, we extend our preliminary work [1] with a pipeline technique and a cluster computing approach. The proposed method is designed to accelerate the SW algorithm for database search on a cluster of GPUs. Our method has four major contributions. Firstly, on-chip memory is used to reduce the data amount being transferred between off-chip memory and processing elements in the GPU. Secondly, a data reuse scheme further reduces the number of data fetches from off-chip memory. Thirdly, a pipeline technique minimizes the execution time by loading database files during kernel execution on the GPU. Finally, a master/worker paradigm is employed to process hundreds of search queries in parallel on a cluster system.

The rest of the paper is organized as follows. We begin in Sect. 2 by introducing related work. Section 3 features the CUDA framework and Sect. 4 gives an overview of the SW algorithm. Section 5 then describes our proposed method and Sect. 6 shows experimental results. Finally, Sect. 7 concludes the paper.

Manuscript received September 4, 2009.

†The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.

*A preliminary version of this paper appeared in [1].

a) E-mail: ino@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.E93.D.1479

2. Related Work

Liu et al. [15] develop the first implementation that runs on the GPU. Since their work is done before the dawning of CUDA, they employ the OpenGL library [16], 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 [7], a heuristic implementation of the SW algorithm running on the CPU. It provides a peak performance of 0.7 giga cell updates per second (GCUPS) at a query of length 4092. Their work is extended by Singh et al. [17] who show a further acceleration using several desktop machines equipped with GPUs.

Manavski et al. [8] present a CUDA-based implementation for the SW algorithm. Their performance reaches a peak of 1.8 GCUPS using a GeForce 8800 GTX card, but it is not clear whether this performance includes the data transfer time needed before/after GPU execution. The performance can be further increased by utilizing on-chip memory, which is in an order of magnitude faster than off-chip memory. A similar work but with a heuristic algorithm is presented by Schatz et al. [18]. Their CUDA-based implementation achieves a 3.5-fold speedup over a CPU implementation.

Although earlier GPU implementations show successful timing results, the acceleration is not enough to outperform an optimized CPU implementation proposed by Farrar [11]. This implementation is optimized using SSE instructions, which are originally designed to accelerate multimedia applications by 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.

Some recent methods [1], [13], [14], [19] exploit on-chip shared memory to achieve higher performance than the SSE-optimized implementation. The performance of these methods roughly ranges from 5.6 GCUPS to 9.6 GCUPS, depending on the employed graphics card. The main difference of this paper to these results is the further acceleration achieved by a pipeline technique. We also extend our single-GPU implementation with a master/worker paradigm, achieving a scalable performance with respect to the number of GPUs.

In contrast to the GPU-based solutions mentioned above, field programmable gate arrays (FPGAs) also provide attractive, hardware-based solutions. Zhang et al. [20] implements the SW algorithm on an FPGA, which provides a peak performance of 25.6 GCUPS. This performance is 250 times faster than a CPU version running on a 2.2 GHz Opteron processor. One drawback of FPGA-based solutions is the cost of expensive FPGAs. FPGAs are not so widely used as compared with GPUs, which have a strong market in the entertainment area. Similar solutions are presented by Storaasli et al. [21] and by Li et al. [22].

3. Compute Unified Device Architecture (CUDA)

CUDA [10] is a programming framework for writing and running general-purpose applications on the nVIDIA GPU. This framework 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 illustrates an overview of the hardware model in CUDA. This model mainly consists of two parts: the GPU itself and off-chip video memory, which is called as device memory. The GPU has a set of multiprocessors (MPs), each including a set of stream processors (SPs) and shared memory. As we mentioned before, shared memory is useful to save the memory bandwidth between SPs and off-chip 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.

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 fast, on-chip shared 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 1 summarizes a hierarchy of memory resources in CUDA. Shared memory is as fast as registers while video memory takes 400 to 600 clock cycles to access non-cached data [10], [23]. However, the capacity of shared memory is currently limited by 16 KB per MP. In contrast, recent high-end GPUs have at least 1 GB of video memory, which can be used as constant, texture, global, and local memory.

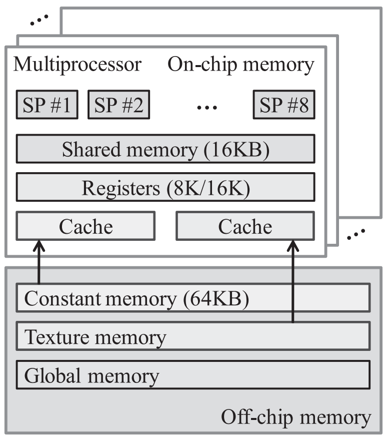


Fig. 1 Hardware model in CUDA. SP denotes stream processor in this figure. The number of registers and that of multiprocessors depend on GPU generation.

Table 1 Memory resources available in CUDA. Latency is presented in clock cycles. The cache working set is 8 KB per MP. The latency of constant memory depends on the locality of data access. Accesses to global memory can be coalesced if satisfying memory alignments [10].

Memory	Capacity	Cache	Latency	Access
Register	8 K/16 K per MP	N/A	1	R/W per thread
Shared	16 KB per MP		1	R/W per block
Constant	64 KB	Yes	30–600	R
Texture	1 + GB			
Global		No	400–600	R/W
Local			400–600	R/W per thread

Constant memory and texture memory are cached but not writable by SPs. They are writable by the CPU in advance of kernel launch. 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 [10] 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 important to avoid such implicit, inefficient access.

4. Smith-Waterman Algorithm

The SW algorithm [4] is based on a dynamic programming approach that obtains the optimal local alignment between two sequences. In general, this pairwise algorithm is iteratively applied to the query and each subject sequence in the database. Thus, we must process the algorithm MN times to deal with N query sequences for a database of M subject sequences. The algorithm finds similar subsequences in two steps as follows.

1. Matrix filling step, which computes a similarity matrix H by comparing the two sequences.
2. Backtracing step, which locates similar subsequences from matrix cells with higher scores.

The former step must be accelerated because it involves computation by more than an order of magnitude as compared with the latter step. In contrast, the latter step can be quickly processed on the CPU [15] because we are usually allowed to focus on only several subsequences in practical situations. For example, backtracing will start from only ten matrix cells if top ten similar subsequences are needed as the alignment result for the query.

Let A denote a query sequence $a_1a_2\dots a_n$ of length n. Let B denote a subject sequence $b_1b_2\dots b_m$ of length m to be compared with the query sequence A. The algorithm then computes an $m \times n$-cell matrix H to obtain the similarity for any pair of subsequences. 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

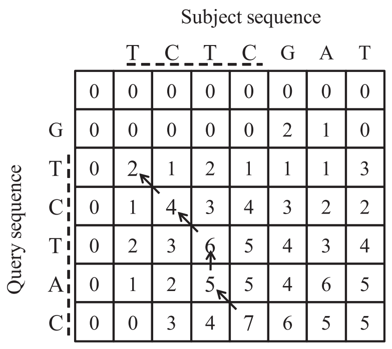


Fig. 2 Example of matrix H filled by Smith-Waterman algorithm. Underlined characters represent the most similar subsequences. A linear gap penalty of $G_{init} = G_{ext} = 1$ is used with a scoring matrix: $W(a_i, b_j) = 2$ if $a_i = b_j$; and $W(a_i, b_j) = -1$ otherwise.

$H_{i,j}$ of two subsequences ending in a_i and b_j, respectively, is then recursively defined as follows:

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

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

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

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 $j < 1$.

Figure 2 shows an example of matrix H filled by the SW algorithm. In this example, two sequences TCTCGAT and GTCTAC are aligned using a linear gap penalty. The most similar subsequences (TCTC and TCTAC) are determined by backtracing from cell $H_{4,6}$ with the highest score.

To compute Eq. (1), we have to fetch two characters and three integers if the scoring matrix W is encoded as constants, execute four arithmetic instructions if max() is implemented as a binary operator, and then write an integer. Similarly, Eqs. (2) and (3) fetch four integers if penalties G_{init} and G_{ext} are implemented as constants, execute six arithmetic instructions, and then write two integers. Since a matrix has mn cells, the SW algorithm accesses at least $42\,mn$ byte data and executes at least $10\,mn$ arithmetic instructions during a pairwise alignment.

5. Proposed Method

We now describe our CUDA-based method accelerated on a cluster of GPUs. The problem parallelized by the proposed method is to process N query sequences with a database of M subject sequences. We first present our parallelization strategy and memory allocation scheme to explain how we adapt the algorithm to our target platform. We then describe the data reuse scheme and the pipeline technique.

5.1 Parallelization Strategy

Since the SW algorithm solves a pairwise alignment, there is no data dependency between different pairs of sequences. Therefore, the problem of MN alignments can be classified into an embarrassingly parallel problem. This motivates us to exploit a coarse-grained parallelism by a master/worker paradigm, which can efficiently parallelize this type of problem on cluster systems.

Figure 3 shows how our master/worker paradigm assigns alignment tasks to computing nodes in the cluster. A task here is associated with alignments between a query sequence and M subject sequences in the database. Note here that we assume that all nodes have the entire database. The database is not decomposed into smaller portions because such a decomposition involves a post-processing step to merge scores into a list, which must be then used to assign backtracing tasks to a part of nodes. In contrast, our per-query strategy allows backtracing tasks to be integrated into matrix filling tasks, simplifying task distribution process on the master node.

In our master/worker paradigm, worker nodes further exploit the parallelism in each task to accelerate alignments on the GPU. Since each task corresponds to M pairwise alignments, a naive strategy may parallelize the task using M threads. Although this strategy may increase the alignment throughput, it cannot accelerate each of pairwise alignments because each is processed in serial. Furthermore, the kernel cannot be launched until loading the last entry in the database. Therefore, we have decided to exploit not only the coarse-grained parallelism but also the fine-grained parallelism in a pairwise alignment. In addition, we overlap kernel execution with database access to minimize the entire execution time. For this reason, our kernel aligns the query sequence with L ($< M$) subject sequences at a time. Thus, the kernel is iteratively launched with different L subject sequences until reaching the last entry in the database.

The fine-grained parallelism we exploited is the same one used in Liu's OpenGL-based method [15]. According to Eqs. (1)–(3), their method computes matrices E, F and H in antidiagonal order. Since each matrix have $m + n - 1$ antidiagonals, this method involves $m+n-1$ iterations (i.e., serialization). However, each antidiagonal is computed in parallel with a maximum parallelism of n.

Figure 4 shows how we adapt Liu's method to the CUDA framework. Our kernel generates L thread blocks and assigns a pairwise alignment to each of them. On the other hand, each of n threads in a thread block is responsible for computing a row in matrices E, F and H. Thus, a kernel invocation computes L matrices using nL threads in order to process L pairwise alignments. After this kernel launch, the highest score $S_{i,l}$ is obtained for every row i of the l-th matrix H, where $1 \leq i \leq n$ and $1 \leq l \leq L$. These scores are then used to screen out sequences that should not be processed at the succeeding backtracing step.

Note here that it is important to sort subject sequences by their length m before the matrix filling step [15] because it balances the workload between thread blocks. Otherwise, every thread block is required to process a different number $m + n + 1$ of iterations to fill matrices.

5.2 Memory Allocation Scheme

The memory allocation scheme is the key factor that determines the kernel performance. In order to maximize the performance, our method computes matrices E, F and H mainly by fast on-chip memory. Since on-chip memory is not large enough to store the entire of matrices, our method avoids storing the entire of matrices during the matrix filling step. Instead of this, the method allocates the minimum amount of memory to compute the highest scores. For example, Eqs. (1)–(3) imply that cells on the k-th antidiagonal of matrix H can be computed from the last two antidiagonals, namely the $(k - 1)$-th and the $(k - 2)$-th antidiagonals of matrices E, F and H, where $k = i + j$. In this way, the memory usage can be reduced from $O(mn)$ to $O(n)$.

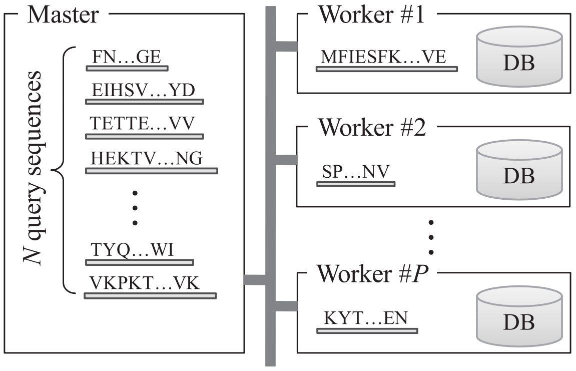


Fig. 3 Master/worker paradigm running on P nodes. The master node assigns alignment tasks to worker nodes, which have M subject sequences in the database. A task here is associated with alignments between a query sequence.

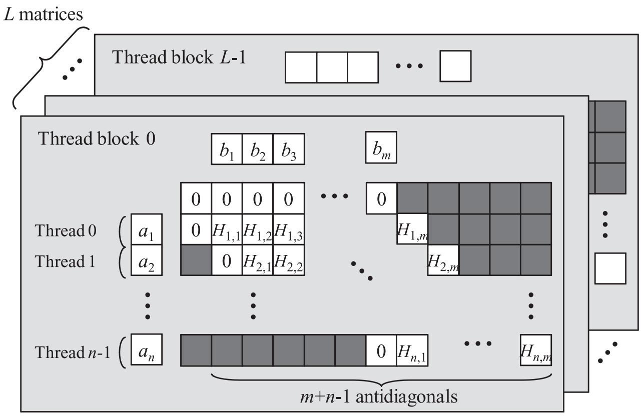


Fig. 4 Parallel matrix filling for L pairwise alignments. Matrix cells on the same antidiagonal are illustrated here in a column. Given a query sequence $a_1 a_2 \ldots a_n$ and L subject sequences, cells on the k-th antidiagonals of L matrices are computed in parallel, where $1 \leq k \leq m + n - 1$. Thus, matrix cells are computed by nL threads from left to right in this figure.

Note here that lacking the entire matrix H is not a critical problem in practical situations because the backtracing step is usually done for several sequences and not for all MN sequences. That is, once the most of sequences are screened out by their scores, necessary matrices can be quickly computed by performing only several alignments.

Since on-chip memory consists of registers and shared memory, we have to determine which antidiagonals should be stored in shared memory. As shown in Fig. 4, our parallelization strategy assigns a row to each thread. In this strategy, the $(k-1)$-th antidiagonal of matrix H and that of matrix F are accessed by multiple threads that compute cells on the k-th antidiagonal, where $1 \leq k \leq m+n-1$. Therefore, we have decided to store them in shared memory. On the other hand, the remaining antidiagonals are accessed only by the responsible thread. Therefore, such antidiagonals should be stored in registers.

With respect to the query sequence, it is accessed only once by threads. Therefore, we store them in global memory and load them in a coalesced manner. On the other hand, we incorporate cache effects by storing M subject sequences in texture memory, because threads do not load them in a coalesced manner. Since subject sequences have different lengths, each of length m is stored in constant memory as an integer value. Thus, the kernel consumes $4L + n$ bytes of constant memory in total. We currently use $L = 8192$ according to the capacity of constant memory (see Table 1). The highest scores $S_{i,l}$ are stored in global memory to send them back to the CPU. Finally, the scoring matrix W can be stored in shared memory or texture memory. We currently encode the scoring matrix W in the kernel code.

Figure 5 shows a pseudocode of the proposed kernel. Let *tid* and *bid* denote the thread ID and the block ID, respectively. This kernel is launched for every thread $\langle tid, bid\rangle$, where $0 \leq tid < n$ and $0 \leq bid < L$. In this kernel, the antidiagonals of matrices are swept by the k loop at line 10. Note here that every thread does not always update its responsible row at each iteration. For example, only the first threads $\langle tid, bid\rangle = \langle 0, *\rangle$ are allowed to compute the antidiagonals of L matrices when $k = 1$. Such flow controls are realized by the branch instruction at line 12. Threads that are allowed to compute a cell $H_{i,j}$ fetch the symbol b_j from texture memory at line 14, and then compute the cell $H_{i,j}$ at line 18. After this, they update the responsible antidiagonals at lines 22–24 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 $S_{tid+1,bid+1}$ of its responsible row $tid + 1$ to global memory at line 28. The highest scores are written in a coalesced manner. 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 28.

```
Input. (1) $query[n]$: a query sequence of length $n$
       (2) $DBtex[m][L]$: $L$ subject sequences of length $m$
Output. $S[nL]$: highest scores for every column of $L$ matrices
 1.  __shared__ int $H_1[n]$    // $(k-1)$-th antidiagonal of $H$
 2.  __shared__ int $F_1[n]$    // $(k-1)$-th antidiagonal of $F$
 3.  $H_1[tid] := -G_{init}$;   // $tid$: thread ID
 4.  $F_1[tid] := 0$;
 5.  $H_2 := 0$;   // $(k-2)$-th antidiagonal of $H$
 6.  $H_0 := 0$;   // $k$-th antidiagonal of $H$
 7.  $E_0 := 0$;   // $k$-th antidiagonal of $E$
 8.  $F_0 := 0$;   // $k$-th antidiagonal of $F$
 9.  $que := query[tid]$;
10.  for $k := 0$ to $m+n-2$ do    // for all antidiagonals
11.    $idx := k - tid$;
12.    if $(idx < 0)$ or $(idx > n)$ then ;    // out of cells
13.    else
14.      $sub := DBtex[idx][bid]$;    // $bid$: block ID
15.      $W := (que == sub)\ ?\ 2 : -1$;    // compute $W(a_i, b_j)$
16.      $E_0 := \max(H_1[tid] - G_{init}), E_0 - G_{ext})$;
17.      $F_0 := \max(H_1[tid-1] - G_{init}), F_1[tid] - G_{ext})$;
18.      $H_0 := \max(\max(\max(0, H_2 + W), E_0), F_0)$;
19.      // update the highest score
20.      $score := \max(score, H_0)$;
21.      // update antidiagonals for the next iteration
22.      $H_2 := H_1[tid-1] + G_{init}$;
23.      $H_1[tid] := H_0 - G_{init}$;
24.      $F_1[idx-1] := F_0$;
25.    end if
26.    __syncthreads();    // synchronization
27.  end for
28.  $S[bid * size + tid] := score$;    // $size = 16\lceil n/16\rceil$
```

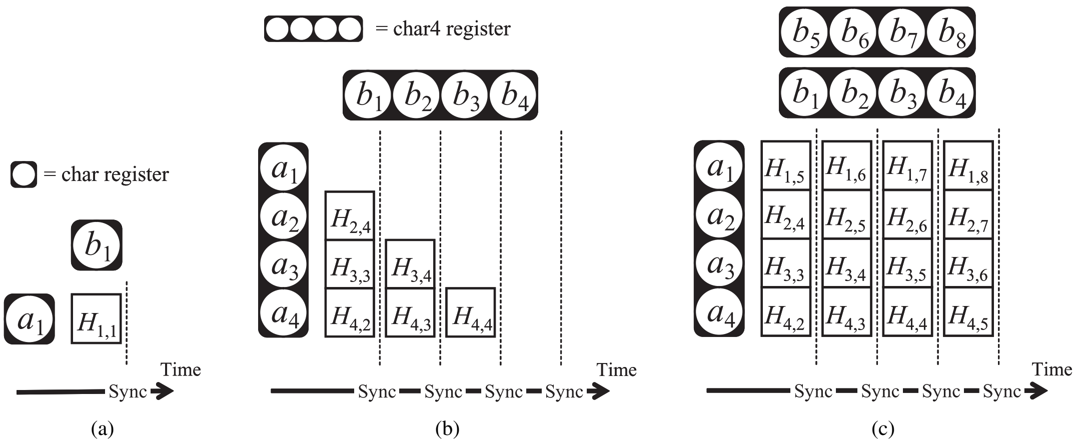
Fig. 5 Pseudocode of naive version of proposed kernel. This naive kernel uses shared memory but does not reuse data for the sake of simplicity. The thread $\langle tid, bid\rangle$ is responsible for the $(tid+1)$-th row of matrix H computed for the $(bid + 1)$-th subject sequence.

5.3 Data Reuse Scheme

Our data reuse scheme aims at reducing the number of texture fetches needed for matrix computation. To realize this, we pack query and subject sequences into vector data formatted in type char4 [10], as shown in Fig. 6 (b). Accordingly, the optimized kernel now assigns four succeeding rows to each thread, performing per-vector computation. Therefore, a thread block consists of $\lceil n/4\rceil$ threads in this kernel. Since thread blocks are currently allowed to have a maximum of 512 threads per block [10], the kernel requires $n \leq 2048$ to run.

The optimized kernel fetches vector data instead of scalar data. An 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. 6 (c)). Otherwise, as shown in Fig. 6 (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 rows 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 fetched from off-chip memory. At each iteration, the naive kernel fetches a symbol b_j of the subject 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 tex-



= char4 register
= char register
b_1
a_1
$H_{1,1}$
Time
Sync
(a)
$b_1 b_2 b_3 b_4$
a_1 a_2 a_3 a_4
$H_{2,4}$ $H_{3,3}$ $H_{4,2}$ $H_{3,4}$ $H_{4,3}$ $H_{4,4}$
Time
Sync Sync Sync Sync
(b)
$b_5 b_6 b_7 b_8$
$b_1 b_2 b_3 b_4$
a_1 a_2 a_3 a_4
$H_{1,5}$ $H_{1,6}$ $H_{1,7}$ $H_{1,8}$
$H_{2,4}$ $H_{2,5}$ $H_{2,6}$ $H_{2,7}$
$H_{3,3}$ $H_{3,4}$ $H_{3,5}$ $H_{3,6}$
$H_{4,2}$ $H_{4,3}$ $H_{4,4}$ $H_{4,5}$
Time
Sync Sync Sync Sync
(c)

Fig. 6 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.

```
1. while (not end of DB)
2.   load $L$ subject sequences from DB;
3.   download loaded sequences to video memory;
4.   launch the kernel;
5.   readback results from video memory;
6. end while
```

(a)

```
1. while (not end of DB)
2.   load $L$ subject sequences from DB;
3.   if (not first iteration)
4.     readback results from video memory;
5.   end if
6.   download loaded sequences to video memory;
7.   launch the kernel;
8. end while
9. readback results from video memory;
```

(b)

Fig. 7 Pseudocode of CPU program. (a) Non-pipelined code and (b) pipelined code.

ture fetches by 75% as compared with the naive kernel. Furthermore, it also reduces the number of branch instructions at line 12 in Fig. 5, due to the less number of iterations.

5.4 Pipelined Alignment

Although our kernel is optimized to CUDA-compatible GPUs for higher alignment throughput, it is not sufficient to minimize the execution time. This motivates us to focus on the CPU code, which loads many sequences from the database and then launches the kernel. For example, we find that 50% of execution time is spent for database access when processing a query sequence of length $n = 255$. This ratio will increase in the future because the GPU doubles the performance at every release [9].

Figure 7 shows how we modify the CPU code to realize pipelined alignments. The naive code launches the kernel at line 4 in Fig. 7 (a). After this launch, control will be immediately returned to the CPU because the launch is processed in asynchronous way. However, the readback operation at line 5 makes the CPU idle until completing the kernel execution. To solve this idle problem, we change the execution order as shown in Fig. 7 (b). In this pipelined code, subject sequences for the next L alignments are loaded just after the kernel launch for the current alignments. The same pipeline behavior can be realized by using streams [10]. However, we currently avoid using streams because it is not supported in GeForce 8800 series cards.

Let t_1, t_2 and t_3 be the execution time for a launched kernel, and that for data download and readback, and that for database access of L subject sequences. Let T be the execution time needed for the entire of MN pairwise alignments. The execution time T of the non-pipelined method is then given by $\lceil M/L \rceil (t_1 + t_2 + t_3)$. In contrast, the pipelined method reduces this to $(\lceil M/L \rceil - 1)(\max(t_1, t_3) + t_2) + t_1 + t_2 + t_3$. The last three terms $t_1 + t_2 + t_3$ here represent the cost of the first and last iterations, which cannot be entirely overlapped with kernel execution.

6. Experimental Results

In order to evaluate the performance of our method, we have implemented the method using Visual Studio 2005 and CUDA [10]. The implementation is benchmarked on a single machine and on a cluster of 32 GPUs. We first show timing results on a single machine, with a breakdown analysis of execution time and a comparison to a previous method [15], [24] implemented using the OpenGL library [16] and C for graphics (Cg) [25]. We then demonstrate how the performance scales on a cluster system.

6.1 Setup

Experiments are conducted using query sequences of length n ranging from 63 to 511 amino acids. All queries are run

Table 2 Machine specification. The master node and 32 worker nodes are interconnected by a 1 Gb/s Ethernet network. Arithmetic performance is presented by integer performance in giga instructions per second (GIPS). Bandwidth represents the theoretical bound of off-chip memory.

Item	Standalone system	Cluster system	
		Master node	Worker node
CPU	Core i7 940 2.93 GHz	Xeon X5450 3 GHz	Xeon (Nocona) 2.8 GHz
RAM	3 GB	8 GB	2 GB
GPU	GTX 280	—	8800 GTX
VRAM	1024 MB	—	768 MB
Arithmetic	311 GIPS	—	173 GIPS
Bandwidth	141.7 GB/s	—	86.4 GB/s
Driver	178.28	—	169.09
CUDA	2.0	—	1.1
OS	Windows XP	Windows XP (x64)	Windows XP

Table 3 Performance comparison with OpenGL-based method [15]. Execution time and alignment throughput are measured using a query sequence ($N = 1$).

Query length	Execution time T (s)		Throughput P (GCUPS)	
n	Proposed	OpenGL	Proposed	OpenGL
63	2.03	7.48	2.81	0.76
127	2.07	10.88	5.55	1.06
191	2.30	14.54	7.52	1.19
255	2.79	18.25	8.32	1.27
319	3.76	22.10	7.68	1.31
383	4.36	25.88	7.96	1.34
447	5.06	29.82	8.00	1.36
511	5.70	34.14	8.12	1.36

against the SWISS-PROT database [2], which is approximately 121 MB in file size, containing 250,143 (= M) entries with a total of 90,588,910 amino acids. Thus, the length m of subject sequences is 362 amino acids in average. The subject sequences are sorted according to length m in advance. Alignments are carried out with an affine gap penalty $G_{init} = 10, G_{ext} = 2$ 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.

Table 2 summarizes the machine specification used for experiments. For a single-GPU environment, we use a Windows PC equipped with a Core i7 940 CPU and an nVIDIA GeForce GTX 280 card. On the other hand, our GPU-equipped cluster consists of a master node and 32 worker nodes, each with a GeForce 8800 GTX card. All nodes are interconnected by a 1 Gb/s Ethernet network. Since pairwise alignments can be classified into an integer application, the arithmetic performance in Table 2 is presented in giga instructions per second (GIPS).

6.2 Single GPU Performance

Table 3 shows the execution time T and the throughput $P = Mmn/T$ for query sequences with different lengths n. The execution time T includes the GPU time T_1 for kernel execution, the transfer time T_2 for data download and readback, and the CPU time T_3 for database access and other CPU-related overheads. The performance is measured using the proposed method and the previous OpenGL-based method [15], [24].

Table 4 Effective kernel performance of proposed method and that of OpenGL-based method [15].

Query length	Arithmetic (GIPS)		Bandwidth (GB/s)	
n	Proposed	OpenGL	Proposed	OpenGL
63	65.7	22.7	256.9	88.9
127	86.5	27.7	338.2	108.4
191	93.2	29.2	364.6	114.3
255	94.7	30.0	370.5	117.3
319	83.0	30.2	324.7	118.3
383	85.4	30.4	334.2	119.0
447	85.8	29.8	335.7	116.5
511	88.1	29.3	344.7	114.6

The proposed method is up to 6.5 times faster than the OpenGL-based method. The performance reaches 8.32 GCUPS when processing a query sequence of length $n = 255$. On the other hand, a previous CUDA-based method achieves 1.8 GCUPS on a GeForce 8800 GTX card [8]. We also measured the performance on the same card and found that the performance reaches 4.42 GCUPS when $n = 255$. Although this is not a fair comparison due to different CUDA/driver versions, our on-chip method achieves a 2.5X speedup over the off-chip method.

We next analyze the efficiency of the kernel with respect to arithmetic performance and memory bandwidth. In this analysis, the arithmetic performance is given by $10mn \cdot M/T_1$ (see Sect. 4). Similarly, the effective bandwidth is given by $42mn \cdot M/T_1$. As shown in Table 4, the proposed method achieves at least 2.8 times higher effective performance than the OpenGL-based method. This speedup is close to that of the CUDA-based method [8] mentioned above. Since on-chip memory cannot be explicitly used in the OpenGL framework, the OpenGL-based kernel has to store matrices in textures. Therefore, the OpenGL-based kernel shows almost the same speedup as the CUDA-based kernel that mainly uses off-chip memory. The kernel speedup of 2.8X also indicates that there is a gap between the entire speedup of 6.5X. This can be explained by the difference of the CPU overhead needed for kernel execution.

With respect to the effective bandwidth, our kernel exceeds the theoretical bandwidth of off-chip memory. In contrast, the performance of the OpenGL-based method is limited by the theoretical bandwidth of 141.7 GB/s. In our method, the amount of data fetched from off-chip memory is first reduced to 1/35 by the shared memory scheme and further is reduced to 1/4 by the data reuse scheme. Thus, the explicit use of registers and shared memory reduces the amount of data fetches to 1/140 in total. In this way, CUDA allows us to increase the ratio of computation to memory access in order to overcome the bandwidth issue typically appeared in various OpenGL-based applications [15], [26], [27].

The results presented above indicate that the proposed method eliminates the performance bottleneck by on-chip memory but it now suffers from computation (i.e., the instruction issue rate). To confirm this, we measure the performance for $n = 255$ on a GeForce 9800 GTX card and on a GeForce 8800 GTX card. The former has 17% higher

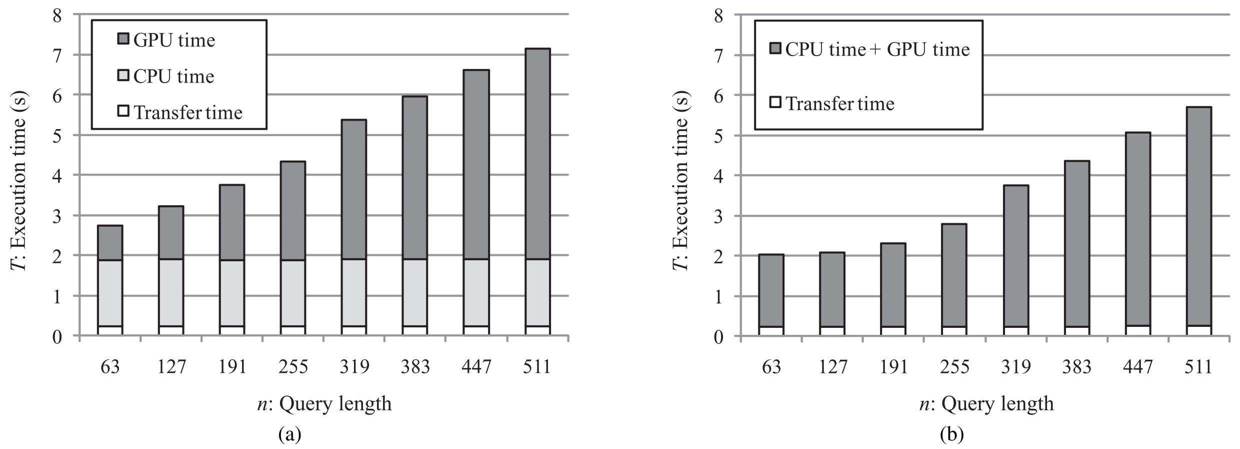


Fig. 8 Comparison of execution time between (a) non-pipelined method and (b) pipelined method.

clock rate but 19% lower memory bandwidth than the latter. We then find that our method increases the throughput by 16% (from 4.42 to 5.51 GCUPS) but the OpenGL-based method decreases the throughput by 10% (from 1.05 to 0.95 GCUPS). Thus, an interesting point here is that the performance bottleneck can vary according to the underlying programming model though the same parallelization strategy is implemented on the same hardware.

According to these results, we think that our kernel may be further improved by reducing the number of instructions. For example, the kernel executes many instructions for control flows and index computation, in addition to the essential arithmetic instructions analyzed in Sect. 4. To confirm this, we investigated the assembly code and found that 66% of GPU cycles are spent on control flow instructions, data access instructions, and synchronization instructions. It also should be noted here that the instruction throughput can be decreased by 33% if shared memory is used in the kernel [23]. Thus, shared memory is useful to accelerate memory-intensive applications but the instruction issue rate can limit the kernel performance.

Finally, we investigate the impact of our pipelined method in terms of execution time. Figure 8 shows the execution time of the non-pipelined method and that of the pipelined method. Since the pipelined method overlaps kernel execution with database access, the execution time T in Fig. 8 (a) is reduced by 21–39%, as shown in Fig. 8 (b). In more detail, we observe $T_1 > T_3$ when $n \geq 191$ while $T_1 < T_3$ when $n \leq 127$. In both cases, the execution time T is roughly reduced by $\min(T_1, T_3)$. To confirm this, we compare the measured time with the estimated time presented in Sect. 5.4. We then find that the timing gap between them is at most 175 ms, which corresponds to at most 7% of the execution time T. This pipeline overhead becomes relatively small as we increase n, because time T increases with length n.

6.3 Cluster Performance

We performed hundreds of database searches using the GPU-equipped cluster system presented in Table 2. A search job here consists of N query sequences, which are then decomposed into N independent tasks. Each task is assigned to computing nodes in the cluster using the master/worker paradigm. We assume that the database is distributed to every node in advance of job execution. The master/worker framework is implemented using Windows Sockets API.

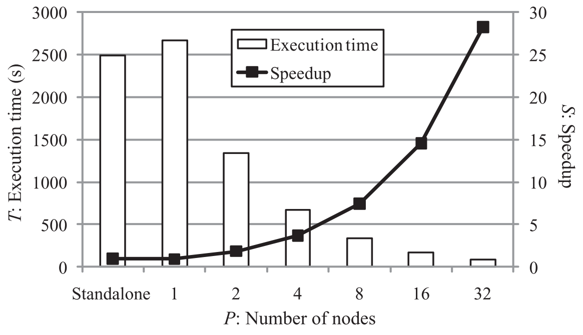


Fig. 9 Execution time and speedup on cluster system. Alignments are done for 256 query sequences of length 63–511.

Figure 9 shows the execution time T and the speedup S measured using P worker nodes using 256 (=N) query sequences. The execution time here corresponds to the elapsed time from start to end: from when the master node receives a job to when all alignment results are sent back to the master. Thus, it includes the communication time needed for task distribution but not for database distribution. The speedup S represents the acceleration ratio over the standalone system, which does not use the master/worker paradigm. The length n of query sequences ranges from 63 to 511, as we used in Sect. 6.2. Thus, tasks have eight different granularities depending on n.

As shown in Fig. 9, our 32-node cluster system reduces the execution time from 41 minutes to 88 seconds, achieving a 28X speedup when $P = 32$. The efficiency S/P ranges from 88% to 92%. Since we have more overheads as we increase P (as T decreases), the efficiency S/P decreases with the increase of P. For example, such overheads include the communication between the master and workers, which accounts for 13% of execution time when $P = 32$.

By comparing the standalone performance with the single-worker performance ($P = 1$), we can understand the impact of overheads incurred on the master node. The timing gap between them is approximately three minutes mainly spent by task distribution and result collection. Although we show that the performance scales well in our 32-node system, some optimization techniques should be applied to the master node in order to achieve higher efficiency on large systems.

Figure 10 shows the alignment performance measured on the cluster system. Our system increases the performance from 2.68 GCUPS to 75.6 GCUPS when using 32 nodes ($P = 32$). This performance can be further increased by overlapping communication with computation on worker nodes. This overlap will achieve 10% higher performance because communication time is at most 8 seconds while the entire time T is 88 seconds when $P = 32$.

Figure 11 shows the speedup S measured with different numbers of query sequences. We obtain relatively high speedups when $N \geq 2P$. On the other hand, lower speedups are observed when $N \leq P$, because alignment tasks are constructed according to a per-query strategy. In such cases, we need another strategy to distribute alignment workload equally to P workers. For example, the database should be decomposed into smaller P portions, which are then distributed to workers. The appropriate strategy should be selected by comparing N with P when the master node receives alignment jobs from users.

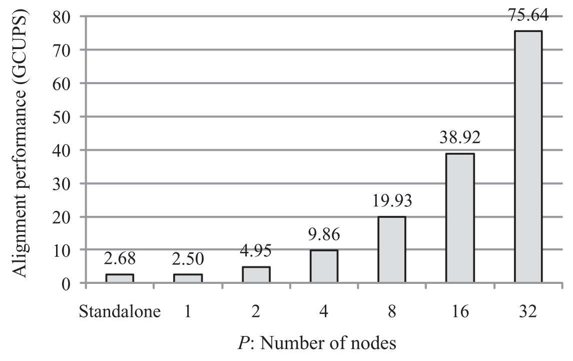


Fig. 10 Alignment performance with different numbers of worker nodes.

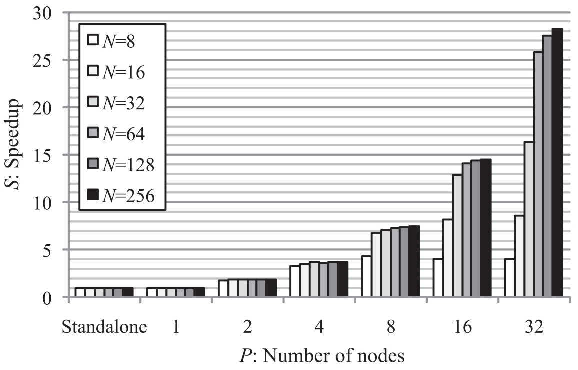


Fig. 11 Speedup measured using different numbers of query sequences. The number N is altered from 8 to 256 query sequences.

7. Conclusion

We have presented a fast method capable of accelerating the SW algorithm for biological database search on a cluster of CUDA-compatible GPUs. Our method is based on a preliminary method [1] that first uses on-chip memory to save the bandwidth between the GPU and off-chip memory. The number of data fetches is also reduced by applying a data reuse technique to query and subject sequences. We further minimize the execution time by a pipelined method that overlaps kernel execution with database access. Finally, the method is integrated into a master/worker framework to scale the performance with the number of GPUs.

The experimental results show that the proposed method achieves a peak performance of 8.12 GCUPS using a GeForce GTX 280 card. This performance is 6.5 times higher than an OpenGL-based method [15], [24]. With respect to the kernel performance, using on-chip memory contributes to a 2.5X speedup over an off-chip kernel [8]. We also show that the performance bottleneck varies depending on the underlying programming model. In particular, the performance of the CUDA-based kernel can be limited by the instruction issue rate if shared memory is used for memory-intensive applications. The master/worker framework also demonstrates a scalable performance on a 32-node cluster system. The performance reaches 75.6 GCUPS when using 32 GeForce 8800 GTX cards.

One future work is to extend our method to deal with longer queries with a length of more than 2048 amino acids. We are also planning to enhance the cluster system to select the appropriate parallelization strategy according to the number of queries and that of available computing nodes.

Acknowledgements

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.

References

[1] Y. Munekawa, F. Ino, and K. Hagihara, "Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU," Proc. 8th IEEE Int'l Conf. Bioinformatics and Bioengineering (BIBE'08), (CD-ROM), Oct. 2008.
[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] "Uniprotkb/swiss-prot release 57.6 statistics," July 2009. http://au.expasy.org/sprot/relnotes/relstat.html
[4] T.F. Smith and M.S. Waterman, "Identification of common molecular subsequences," J. Molecular Biology, vol.147, pp.195–197, 1981.

[5] 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.

[6] 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.

[7] 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.

[8] 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, March 2008,

[9] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell, "A survey of general-purpose computation on graphics hardware," Comput. Graph. Forum, vol.26, no.1, pp.80–113, March 2007.

[10] nVIDIA Corporation, "CUDA programming guide version 2.0," June 2008. http://developer.nvidia.com/cuda/

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

[12] A. Klimovitski, "Using SSE and SSE2: Misconceptions and reality," Intel Developer Update Magazine, March 2001.

[13] Y. Liu, D.L. Maskell, and B. Schmidt, "CUDASW++: Optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," BMC Research Notes, vol.2, no.73, May 2009.

[14] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using cuda, for massively parallel scanning of sequence databases," Proc. 8th Int'l Workshop High Performance Computational Biology (HiCOMB'09), (CD-ROM), May 2009.

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

[16] D. Shreiner, M. Woo, J. Neider, and T. Davis, OpenGL Programming Guide, fifth ed., Addison-Wesley, Reading, MA, Aug. 2005.

[17] A. Singh, C. Chen, W. Liu, W. Mitchell, and B. Schmidt, "A hybrid computational grid architecture for comparative genomics," IEEE Trans. Inf. Technol. Biomed., vol.12, no.2, pp.218–225, March 2008.

[18] 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.

[19] G.M. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," Proc. 23rd IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'06), 10 pages (CD-ROM), May 2009.

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

[21] O. Storaasli, W. Yu, D. Strenski, and J. Maltby, "Performance evaluation of FPGA-based biological applications," Proc. Cray User Group (CUG'07), May 2007.

[22] 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, June 2007.

[23] V. Volkov and J.W. Demmel, "Benchmarking GPUs to tune dense linear algebra," Proc. Int'l Conf. High Performance Computing, Networking and Storage (SC'08), (CD-ROM), Nov. 2008.

[24] F. Ino, Y. Kotani, and K. Hagihara, "Harnessing the power of idle GPUs for acceleration of biological sequence alignment," Proc. 2nd Workshop Large-Scale Parallel Processing (LSPP'09), (CD-ROM), May 2009.

[25] 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, July 2003.

[26] N. Galoppo, N.K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'05), (CD-ROM), Nov. 2005.

[27] F. Ino, S. Yoshida, and K. Hagihara, "RGBA packing for fast cone beam reconstruction on the GPU," Proc. SPIE Medical Imaging (MI 2009), (CD-ROM), Feb. 2009.

Yuma Munekawa received the B.E. degree in information and computer sciences from Osaka University, Osaka, Japan, in 2008. He is currently working toward the M.E. degree at the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. His current research interests include high performance computing, grid computing, and systems architecture and design.

Fumihiko Ino received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently an Associate Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.

Kenichi Hagihara received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1974, 1976, and 1979, respectively. From 1994 to 2002, he was a Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. Since 2002, he has been a Professor in the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. From 1992 to 1993, he was a Visiting Researcher at the University of Maryland. His research interests include the fundamentals and practical application of parallel processing.