Parallel Processing Letters
© World Scientific Publishing Company

HARNESSING THE POWER OF IDLE GPUS FOR ACCELERATION OF BIOLOGICAL SEQUENCE ALIGNMENT*

FUMIHIKO INO, YUKI KOTANI[†], YUMA MUNEKAWA and KENICHI HAGIHARA
Graduate School of Information Science and Technology, Osaka University
1-5 Yamada-oka, Suita, 565-0871 Osaka, Japan

Received July 31, 2009
Revised August 24, 2009
Communicated by Guest Editors

ABSTRACT

This paper presents a parallel system capable of accelerating biological sequence alignment on the graphics processing unit (GPU) grid. The GPU grid in this paper is a desktop grid system that utilizes idle GPUs and CPUs in the office and home. Our parallel implementation employs a master-worker paradigm to accelerate an OpenGL-based algorithm that runs on a single GPU. We integrate this implementation into a screensaver-based grid system that detects idle resources on which the alignment code can run. We also show some experimental results comparing our implementation with three different implementations running on a single GPU, a single CPU, or multiple CPUs. As a result, we find that a single non-dedicated GPU can provide us almost the same throughput as two dedicated CPUs in our laboratory environment, where GPU-equipped machines are ordinarily used to develop GPU applications. In a dedicated environment, the GPU-accelerated code achieves five times higher throughput than the CPU-based code. Furthermore, a linear speedup of 30.7X is observed on a 32-node cluster of dedicated GPUs. We also implement a compute unified device architecture (CUDA) based algorithm to demonstrate further acceleration.

Keywords: GPGPU, grid computing, sequence alignment, OpenGL, CUDA.

1. Introduction

With the increasing demand for producing realistic scenes in graphics applications, the graphics processing unit (GPU) is evolving as an accelerator for compute- and memory-intensive applications [1–3]. Although this chip is originally designed for special purposes, it now has a flexible graphics pipeline capable of executing scientific programs written in shading languages, such as C for graphics (Cg) [4] and OpenGL shading language (GLSL) [5].

*This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(20240002), Young Researchers (B)(19700061), and the Global COE Program "in silico medicine" at Osaka University.
[†]Currently with Central Japan Railway Company.

In addition to this graphics-oriented programmability, there are some non-graphics programming environments, such as compute unified device architecture (CUDA) [6], close-to-metal (CTM) [7], and OpenCL [8]. These environments reveal the GPU as pure data-parallel hardware that does not require computer graphics knowledge to write GPU programs. As compared with CPU-based implementations, CUDA-based implementations achieve higher performance for highly-threaded applications, typically with 10X–100X speedups depending on problem characteristics: available parallelism, data access pattern, and data size, for example.

Thus, the GPU is emerging as an attractive high-performance computing (HPC) platform for solving compute- and memory-intensive problems in various scientific fields. However, this chip is also useful for high-throughput computing (HTC) platforms, such as grid systems. That is, GPUs can be regarded as powerful resources in grid environments, where resources are shared among multiple organizations to construct a virtual supercomputer. We think that there is still huge computing power left in ordinary circumstances, because the GPU is mostly used for computer-aided design (CAD) systems in the company and for entertainment in the home. We also believe that the parallelism inherent in the display device will further increase in the future, leading to the advance in parallel architecture. Therefore, it is important for desktop grid systems [9] to support such commodity parallel hardware.

There are some grid projects that exploit the GPU for acceleration of scientific applications. To the best of our knowledge, the Folding@home project [10] achieves the highest performance of approximately 5 PFLOPS (July 2009) using 25,000 GPUs that typically run on dedicated systems 24 hours a day [10]. According to their statistics, this performance is equivalent to 85,000 Cell Broadband Engines (CBEs) or 5,200,000 CPUs. The main goal of their project is to understand the process of protein folding. A similar project [11] is running in the field of molecular dynamics. Though these systems demonstrate successful acceleration in dedicated environments, it is still not clear how much performance we could gain from non-dedicated GPUs, which are used for daily work in the office and laboratory. This motivates us to develop a grid system [12] that focuses on non-dedicated resources as well as dedicated resources.

In this paper, we demonstrate how well the GPU can accelerate HTC applications in non-dedicated grid environments. To achieve this, we extend our previous work [12] by integrating a bioinformatics application into our grid system: sequence alignment [13] for biological database. Our parallel implementation is an extension of Liu's GLSL-based algorithm [14] that accelerates the Smith-Waterman alignment [13] on a single GPU. The parallel implementation is capable of scanning biological database in a non-dedicated, distributed environment. Since the GPU grid is an instance of a distributed heterogeneous system, we parallelize the Smith-Waterman alignment using a master-worker paradigm, namely a well-known paradigm running successfully on existing grid systems. We also implement a CUDA-based algorithm [15] and integrate it into the system to demonstrate further acceleration.

The paper is structured as follows. Section 2 presents related work. Section 3 summarizes the OpenGL-based Smith-Waterman alignment on the single GPU, which is the basis of our parallel implementation. Section 4 describes our implementation with an overview of the grid system. Section 5 shows experimental results obtained using non-dedicated resources and dedicated resources. Section 6 presents discussion on some technical issues left in the system. Finally, Section 7 concludes the paper with future directions.

2. Related Work

The Folding@home project [10] develops several implementations to accelerate protein folding simulations on CPUs, CBEs, and nVIDIA/ATI GPUs. They employ a screensaver-based approach to detect idle resources, allowing us to perform large-scale volunteer computing based on a master-worker paradigm. In their system, grid applications suspend if resource owners (donators) execute their local application that requests exclusive DirectX mode [10]. A similar approach is employed in our system [12], but the main difference is that our system monitors the video memory consumption to detect idle GPUs. This prevents resource owners from being interfered with by grid applications even though local applications are not implemented with the exclusive mode. Thus, the problem of resource conflicts between resource owners and grid users should be addressed to share GPUs in non-dedicated environments.

Yamagiwa and Sousa [16] propose the Caravela system, which allows us to perform stream computing in a distributed environment. Their system employs a master-worker paradigm to efficiently stream data through a distributed, parallel pipeline. Similar to the Folding@home system, the Caravela system probably assumes dedicated environments.

To the best of our knowledge, Fan et al. [17] firstly demonstrates the impact of using multiple GPUs for acceleration of non-graphics applications. They construct a 32-node cluster of GPUs in order to accelerate a flow simulation based on the Lattice Boltzmann model. A 4.6X speedup is achieved by adding an nVIDIA GeForce 6800 Ultra card to each node of the cluster.

Strengert et al. [18] propose a programming framework for running non-graphics applications on multi-GPU environments. Their framework extends the CUDA specification [6] to make CUDA-like programs run on arbitrary multi-GPU environments. Using matrix multiplication, they achieve scalable performance on a single machine with multiple GPUs. Since the extended specification requires distributed shared memory to run on GPU clusters, it will be a challenging problem to scale the performance on such multi-node systems.

With respect to the acceleration of sequence alignment, many projects tackle this problem by using various accelerators, such as the GPU [14, 15, 19], CBE [20], and FPGA [21]. FPGA-based solutions currently achieve the highest performance among these accelerators, but this hardware is not commonly used in our target environment. As compared with the CBE (Sony PlayStation 3), we think that the GPU has advantages in the memory capacity, the network connectivity, and the performance growth rate [2].

There are two methods for implementing the Smith-Waterman alignment on the GPU. Liu et al. [14] firstly develop a Smith-Waterman implementation using OpenGL [22] and GLSL [5]. Their implementation runs on a GeForce 7800 GTX card and achieves 3.5X–10X speedup over SSEARCH [23], namely a heuristic implementation of the Smith-Waterman algorithm running on the CPU. On the other hand, Manavski et al. [19] propose a CUDA-based solution, which is approximately 6X faster than the OpenGL-based solution. Finally, Munekawa et al. [15] presents another implementation that achieves further 3X acceleration through the use of fast but small on-chip shared memory [6].

3. Sequence Alignment on the Single GPU

The alignment of sequence $A = a_1a_2 \dots a_n$ to another sequence $B = b_1b_2 \dots b_m$ is to convert A to B by insertion or elimination of sequence elements, where n and m represent the length of A and that of B, respectively. In particular, local alignment is the process to identify the most similar part in the pair of sequences. The similarity here is defined according to the alignment cost being associated with the number of insertions and eliminations.

3.1. *Smith-Waterman Algorithm*

The Smith-Waterman algorithm [13] is a dynamic programming method that gives the exact solution to the problem of local alignment. Let $H_{i,j}$ be the highest similarity between $a_1a_2 \dots a_i$ and $b_1b_2 \dots b_j$, namely a part of sequences ending at element a_i and b_j, respectively, where $1 \le i \le n$ and $1 \le j \le m$. The similarity $H_{i,j}$ is then defined as follows:

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

where $s(a, b)$ represents the substitution cost needed for converting element a to another element b. $E_{i,j}$ and $F_{i,j}$ are given by

$$E_{i,j} = \max\{H_{i,j-1} - \alpha, E_{i,j-1} - \beta\}, \tag{2}$$

$$F_{i,j} = \max\{H_{i-1,j} - \alpha, F_{i-1,j} - \beta\}, \tag{3}$$

where α and β represent gap penalties for aligning sequence length. The initial values for $H_{i,j}$, $E_{i,j}$ and $F_{i,j}$ are 0 when $i < 1$ and $j < 1$. Note that we currently use a linear gap penalty in our implementation: $\alpha = \beta = 1$. Similarly, the substitution cost is defined as $s(a, b) = 2$ if $a = b$. Otherwise, $s(a, b) = -1$.

Applying Eq. (1) to all locations i and j, we obtain a matrix H that includes $H_{i,j}$ as a matrix cell. Figure 1 illustrates an example of matrix H computed for two sequences A and B. After this matrix computation, the most similar part can be obtained by performing traceback from the cell with the highest score in the matrix.

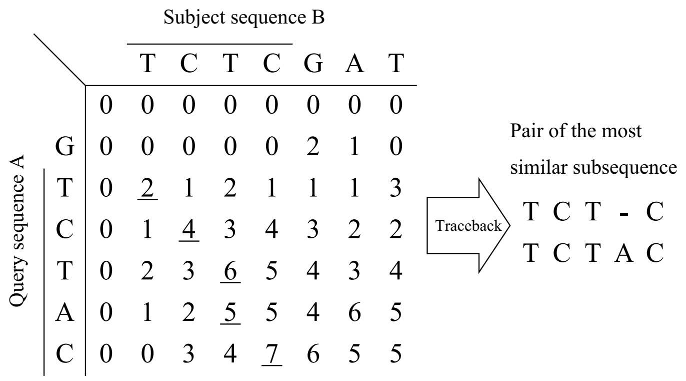


Fig. 1. Smith-Waterman algorithm. This example shows matrix H computed for two sequences: A = GTCTAC and B = TCTCGAT. The score on the matrix represents the similarity between subsequences. Traceback from the highest cell ($H_{4,6} = 7$, in this case) gives the alignment result: TCTC and TCTAC.

In practical situations, the Smith-Waterman algorithm is iteratively applied to all subject sequences in database. That is, alignments are usually done between N query sequences and M subject sequences in order to find pairs of subsequences with higher scores: top ten scores for each query, for example. In this situation, the most pairs can skip the traceback procedure to save the time. Therefore, the traceback cost is not so high compared with the matrix computation cost [14]. As Liu et al. did in their implementation, we also decided to implement the traceback procedure on the CPU. In addition, the parallelism in this procedure is not high enough to run efficiently on the GPU.

3.2. *Single GPU Implementation*

Figure 2 illustrates data dependencies in matrix computation. Equations (1)–(3) indicate that a 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, there is no dependence between any cells on the same antidiagonal, allowing us to process such cells in parallel. However, there are data dependencies between different antidiagonals of the same matrix. That is, the t-th antidiagonal depends on the $(t-1)$-th and the $(t-2)$-th antidiagonals. Thus, the available parallelism is limited to the diagonal length of the matrix, which is not sufficient against 2-D matrix data in terms of the efficiency.

To increase the parallelism, Liu et al. exploit both the data parallelism and the task parallelism. They simultaneously perform multiple alignments between a query sequence and M subject sequences in database. As shown in Fig. 2, they pack M matrices into a single 3-D matrix, exploiting the parallelism in the depth direction. Thus, the packed data allows the GPU to compute M antidiagonals at a time.

This parallelization scheme requires a $(\max(n, m) + 1) \times M$ texel texture [2] to store the t-th antidiagonals of M matrices. In the same manner, we can store the $(t-1)$-th and the $(t-2)$-th antidiagonals into 2-D textures with the same size.

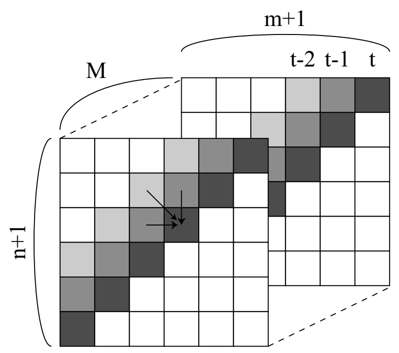


Fig. 2. Data dependencies in computation of M matrices, where M represents the number of subject sequences. Matrix cells on the same antidiagonals of M matrices can be computed in parallel.

Since similarity scores can be computed from the last two antidiagonals, we are allowed to have only three 2-D textures instead of a 3-D texture by selecting the appropriate input/output textures with incrementing t. Thus, as shown at line 6 in Fig. 3(a), $n + m - 1$ iterations of a loop compute the highest score between a query sequence and M subject sequences. Note that subject sequences must be sorted by length to maximize the parallelization efficiency. Otherwise, the matrix can be sparse, resulting in a lower efficiency due to load imbalance.

We have to divide M subject sequences into batches, because there is a limitation on the maximum texture size, which depends on the GPU architecture. Our implementation currently assumes 4096 $\times$ 4096 texels per texture to make it possible to run the implementation on older GPUs. Thus, the GPU code in Fig. 3(b) computes matrices for pairs of a query sequence and 4096 subject sequences at a time. This code is iteratively invoked until reaching at the end of database entry, as shown at line 2 in Fig. 3(a).

4. Parallel Alignment on the GPU Grid

Figure 4 shows an overview of the GPU grid. The GPU grid has almost the same structure as existing grid systems, except that it explicitly utilizes GPUs as general-purpose computational resources. This system has the following three components: grid resources, a resource management server, and clients. Grid resources are a number of desktop PCs connected to the Internet. Any desktop PC can be registered as a grid resource regardless of having the GPU or not. These resources basically run local applications to serve their owners but they also execute grid applications if they are in the idle state. The resource management server takes the responsibility for monitoring registered resources and for selecting the appropriate resources for job execution. In addition, it accepts grid jobs from grid users. Finally, clients are frontend machines serving for grid users who submit grid jobs to the management server. These machines can also be registered as grid resources.

```
Input: a query sequence and $M$ subject sequences in database
Output: highest scores for every column of $M$ matrices
 1. Initialize texture $Query$ with the query sequence;
 2. **while** any unread subject sequences exist in the database **do**
 3.     Initialize texture $Subject$ with a batch of subject sequences;   /* batch size = 4096 */
 4.     Initialize texture $T_A$, $T_B$, and $T_C$;
 5.     Bind and attach $T_A$, $T_B$, and $T_C$ to color buffers $C_1$, $C_2$, and $C_3$, respectively;
 6.     **for** $t = 1$ **to** $n + m - 1$ **do**
 7.         Calculate texture coordinates $texCoord[4]$ and vertex coordinates $vertex[4]$;
 8.         **if** ($t$ mod $3 = 0$) **then**
 9.             Set $C_1$ as output texture and set $C_2$ and $C_3$ as input texture;
10.         **else if** ($t$ mod $3 = 1$) **then**
11.             Set $C_2$ as output texture and set $C_1$ and $C_3$ as input texture;
12.         **else**
13.             Set $C_3$ as output texture and set $C_1$ and $C_2$ as input texture;
14.         **end if**;
15.         Set $Query$ and $Subject$ as input texture;
16.         DrawQuad($texCoord$, $vertex$, $t$);       /* Kernel computation on the GPU */
17.     **end for**;
18.     Read back alignment scores to the main memory;
19.     Release the bindings of textures and buffers;
20. **end while**;
```

(a)

```
 1. float4 Linear_Alignment(float2 texCoord:TEXCOORD0,
                 uniform samplerRECT in_Matrix1:TEXUNIT0,
                 uniform samplerRECT in_Matrix2:TEXUNIT1,
                 uniform samplerRECT in_Query:TEXUNIT2,
                 uniform samplerRECT in_Subject:TEXUNIT3,
                 uniform float loopCounter):COLOR
 2. {
 3.     float a = texRECT(in_Query, float2(texCoord.y, 0.0f)).r;
 4.     float b = texRECT(in_Subject, float2(loopCounter−texCoord.y, texCoord.x)).r;
 5.     float left = texRECT(in_Matrix1, texCoord).r − 1.0f;
 6.     float up = texRECT(in_Matrix1, texCoord + float2(0.0f, −1.0f)).r − 1.0f;
 7.     float upleft = texRECT(in_Matrix2, texCoord + float2(0.0f, −1.0f)).r;
 8.     upleft = (a == b) ? upleft + 2.0f : upleft − 1.0f;
 9.     float score = max(max(max(0.0f, up), left), upleft);
10.     float highestScore = max(score, texRECT(in_Matrix1, texCoord).a);
11.     return float4(score, 0.0f, 0.0f, highestScore);
12. }
```

(b)

Fig. 3. Pseudocode of matrix computation on the single GPU. (a) The CPU code iteratively invokes the GPU code to scan M subject sequences for a query sequence. (b) The GPU code is applied to every texel of the output texture at line 16 in the CPU code.

4.1. *Resource Monitoring and Selection*

A screensaver-based system [12] runs on every registered resource to detect the idle state. We currently use 5 minutes as the waiting time for activating the screensaver, but this time can be changed by resource owners. After the screensaver activation,

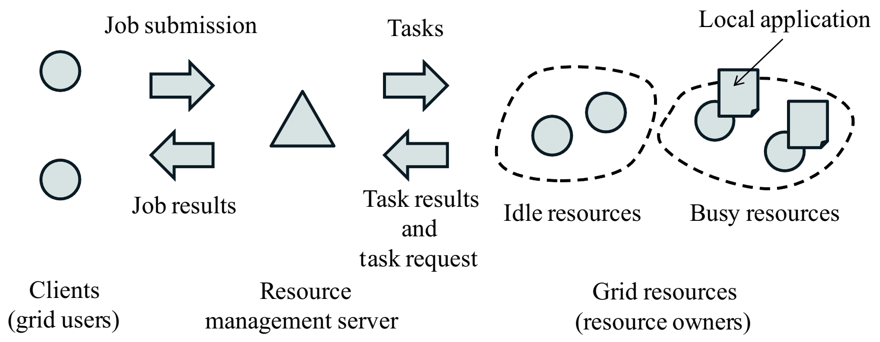


Fig. 4. Overview of the GPU grid.

the system checks the video memory usage and the CPU usage to confirm both the GPU and CPU are actually in the idle state. This information is available from the graphics driver and operating system (OS), so that the perturbation of resource monitoring is minimized in our system.

Idle resources are then selected for job execution, according to a Condor-like matchmaking framework [24]. The reason why we select resources is that the GPU is evolving with increasing heterogeneity of hardware and software. Actually, there are vendor specific extensions in the OpenGL library, and moreover, CUDA applications do not run on ATI cards nor even on nVIDIA cards of pre G80 architectures. To deal with this problem, we need a selection mechanism that allows grid users to specify the resources they want to use.

Similar to Condor's framework, a resource specification mechanism is provided on the basis of a formal language that supports various operators, such as arithmetic, boolean, and logical operators. For example, grid users must prepare a text file that describes their requirements in expressions like "vram>=512," "gpu==GeForce 7800 GTX,GeForce 8800 GTX," and "readback>=1000," in order to select resources that provide a readback rate of at least 1000 MB/s and have a GeForce 7800/8800 GTX card with at least 512 MB of video memory. The performance value here is measured by benchmark programs [25] at screensaver installation. To assist grid users in their resource specification, the system provides them resource statistics including histograms of registered resources.

4.2. *Master-Worker Scheme*

Figure 5 shows the processing flow of our parallel alignment. We parallelize the GPU implementation using a master-worker paradigm. Given a grid job consisting of N query sequences and M subject sequences, we decompose this job into N tasks, each corresponding to a single query sequence. Since there is no data dependence between tasks, each task is then independently assigned to an idle resource after matchmaking. Therefore, a grid resource is responsible for scanning M subject sequences for a query. Thus, this application can be regarded as a parameter-sweep

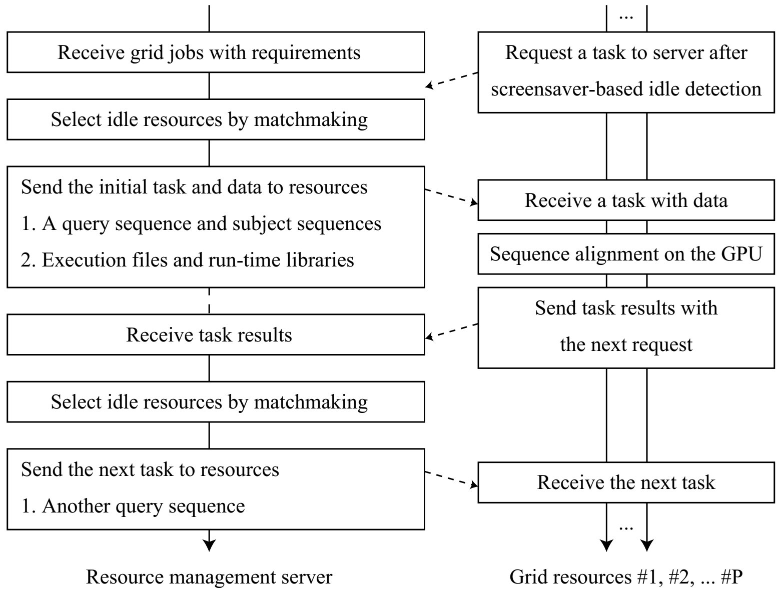

Receive grid jobs with requirements
Select idle resources by matchmaking
Send the initial task and data to resources
1. A query sequence and subject sequences
2. Execution files and run-time libraries
Receive task results
Select idle resources by matchmaking
Send the next task to resources
1. Another query sequence
Resource management server
...
Request a task to server after screensaver-based idle detection
Receive a task with data
Sequence alignment on the GPU
Send task results with the next request
Receive the next task
...
Grid resources #1, #2, ... #P

Fig. 5. Parallel alignment on the GPU grid. Our parallel implementation is based on a master-worker paradigm that decomposes a grid job into N tasks, where N represents the number of query sequences composing the job. Executable files, run-time libraries, and subject sequences are sent only once to each resource. Query sequences must be sent at every job execution.

application, which efficiently runs on existing grid systems.

The server has to send all necessary data to resources that process tasks. Such data contains executable files, run-time libraries, and query and subject sequences. However, once a resource receives a task from the server, the most part of data can be reused for the next task to save the communication time between the server and resources. Thus, only query sequences must be sent after the initial assignment, which is smaller than the remaining data by an order of magnitude.

5. Experimental Results

We now show experimental results to evaluate the system with respect to performance. In order to simplify the analysis, we first show results on dedicated resources and then on non-dedicated resources. Dedicated resources here correspond to cluster environments where there does not occur conflicts between grid users and resource owners.

Liu's algorithm is implemented using the C++ language, the OpenGL library [22], and the Cg toolkit [4]. We use Frame Buffer Object (FBO) [26] to store the matrix data in the video memory.

Table 1. Machine specification. Eight Windows XP machines are interconnected by 100 Mb/s Ethernet network. Machine #8 is a laptop PC while the remaining machines are desktop PCs. Machine #6 has a graphics card with dual GPUs. Arithmetic denotes the floating point performance of fragment processors in the GPU.

Resource ID	CPU	Clock (GHz)	RAM (GB)	GPU	VRAM Capacity (MB)	VRAM BW (GB/s)	Fill rate (Gpixel/s)	Arithmetic (GFLOPS)
#1	Pentium 4	3.4	2	8800 GTX	768	86.4	36.8	345.6
#2	Xeon	2.8	4	8800 GTX	768	86.4	36.8	345.6
#3	Pentium 4	2.8	2	8800 GTX	768	86.4	36.8	345.6
#4	Pentium 4	2.8	1	8800 GTX	768	86.4	36.8	345.6
#5	Xeon	3.8	4	8800 GTS	640	64.0	24.0	230.4
#6	Core 2 Duo	2.4	2	7950 GX2	512x2	38.4x2	12.0x2	64.0x2
#7	Pentium D	3.0	2	7900 GTX	512	51.2	15.6	124.8
#8	Core Duo	2.0	2	7900 GTX∗	512	38.4	12.0	96.0

∗: This card is GeForce Go 7900 GTX.

5.1. *Setup*

Alignments are carried out using the SWISS-PROT database [27] of release 51.0, which contains 241,242 ($= M$) entries in a file of 118 MB. This database has a total of 88,541,632 amino acids, so that the average length m per entry is 367. To maximize the performance, we have sorted subject sequences with respect to the length in advance. We use 64 ($= N$) query sequences of length $n = 367$. Each query is stored in a file of approximately 512 B.

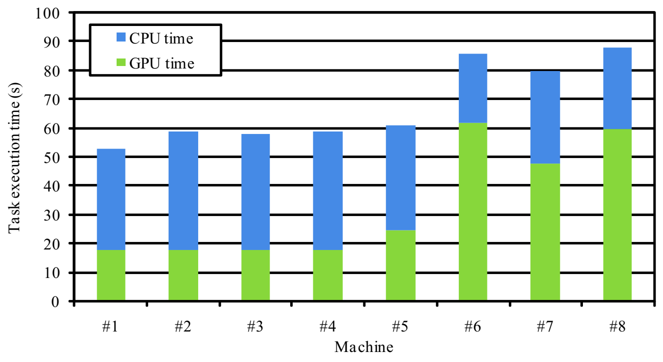
To understand how well alignments are accelerated on the GPU, we use SSEARCH [23] as a CPU implementation of the Smith-Waterman algorithm. Note that there is an SSE-optimized version [28] of SSEARCH, which is approximately three times faster than the original.

Table 1 summarizes the specification of eight Windows XP machines employed for experiments. The first five machines #1–#5 have a GPU of G80 architecture while the remaining machines have a GPU of G70 architecture. The last machine #8 is a laptop machine. All machines are placed in a local area network (LAN) environment with 100 Mb/s bandwidth. Using these heterogeneous machines, we compare our implementation with three implementations: multiple CPU, single CPU, and single GPU versions.

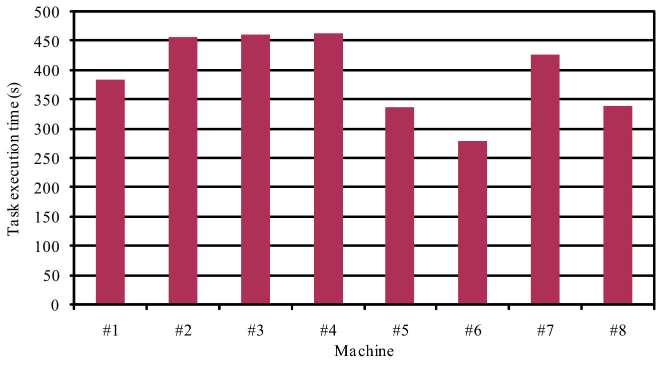
5.2. *Evaluation on Dedicated Resources*

We first measure the execution time on dedicated resources. The execution time here corresponds to the elapsed time from start to end: from when the server receives a grid job to when all task results are sent back to the server. Thus, it includes the time spent for data transfer between the server and resources. The single GPU and CPU performances are measured on resources #1 and #6, respectively, where we obtain the fastest result.

Originally, it takes five hours (17,920 seconds) to align sequences on a single CPU. This execution time is reduced to 11 minutes (673 seconds) using 8 GPUs. On



(a)



(b)

Fig. 6. Execution time for a task, which performs a single scan of database. (a) Results on the GPU and (b) those on the CPU. The GPU time and the CPU time correspond to the execution time of the GPU code and that of the CPU code, respectively.

the other hand, a single GPU version takes approximately an hour (3,392 seconds), which is close to the time (3,345 seconds) on 8 CPUs. Therefore, a 5X speedup is achieved using 8 machines regardless of having the GPU or not. We think that this nonlinear speedup is reasonable due to the heterogeneous machine configuration.

Figure 6 shows the task execution time spent for a single scan of database. Resource #6 demonstrates the fastest CPU time of 280 seconds in Fig. 6(b) but GPU-equipped laptop machine #8 completes the same scan within 88 seconds in Fig. 6(a). We also can see that resources #1–#4 spend only 18 seconds for the GPU code. Accordingly, 70% of the entire time is spent for other operations on the CPU, such as texture switching and data initialization. Therefore, it indicates that the CPU performance can determine the entire performance of OpenGL-based applications. Thus, the resource management server should gather both CPU and GPU information to allow users to select the appropriate resources from a pool of resources.

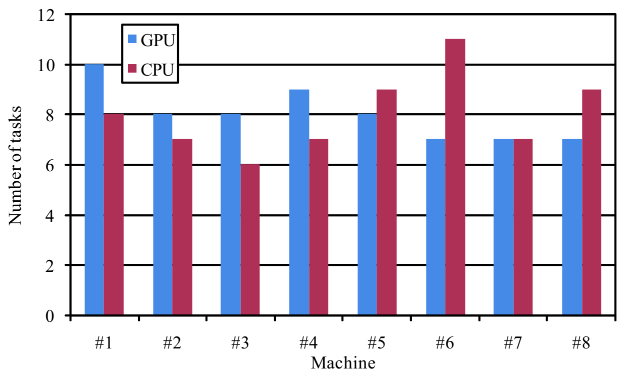


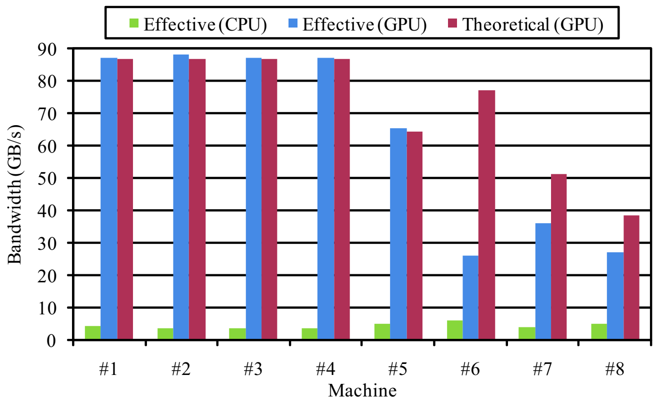
Fig. 7. Task distribution statistics.

We also investigate the communication time spent for data transfer between the server and resources. We find that the communication time ranges from 14.8 to 21.6 seconds at the first task execution, because each resource has to receive 118 MB of subject sequence data. However, this data transfer can be partly omitted after the initial execution, as we mentioned before. Thus, the communication time reduces to at most 20 milliseconds, which is much shorter than the task execution time shown in Fig. 6(a). This reduction is important to develop scalable systems capable of increasing application throughput with the number of resources.

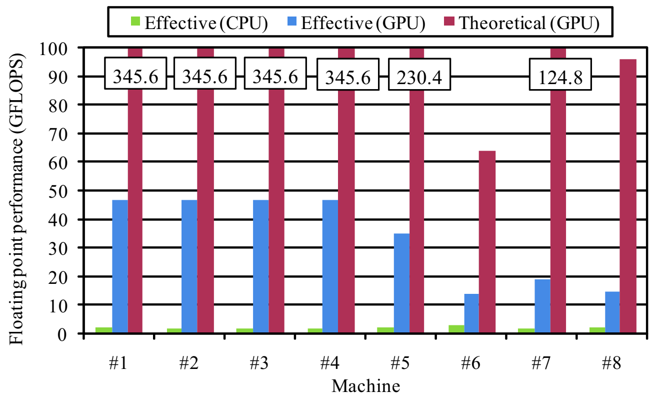
The effective network bandwidth in this experiment ranges from 45 to 63 Mb/s, which probably cannot be achieved in wide area network (WAN) environments. For example, the communication time will increase to at least 5 minutes if the effective bandwidth reduces to 3 Mb/s. On such slower network, we have to increase the task granularity, namely the number N of query sequences, to maximize the efficiency.

Figure 7 shows the number of task execution for each resource. Since a job contains 64 query sequences, the perfect load balancing will be achieved if each machine is responsible for 8 sequences. However, the number ranges from 6 to 11 mainly due to the heterogeneous architecture. Actually, resources #6–#8 is approximately 50% slower than the remaining resources, because they have relatively older GPUs. The GPU usually doubles the performance at every new architecture. Thus, there is a significant gap between the performance of the current generation and that of the next generation. In this sense, the GPU grid must deal with the problem of load balancing, which is addressed by the master-worker paradigm in our system.

We next analyze the performance of the GPU code. Figure 8 shows the effective performance in terms of floating point arithmetic and memory bandwidth. The performance here is computed from the GPU time in Fig. 6(a), which does not include the CPU time. In Fig. 8, we can see that resources #1–#4 achieve the highest performance of 47 GFLOPS. However, this is not close to the theoretical peak performance of 345.6 GFLOPS. The reason for this is that the Smith-Waterman alignment is a memory-intensive application, where the memory bandwidth limits



(a)



(b)

Fig. 8. Effective performance of the alignment code in terms of (a) floating point arithmetic and (b) memory bandwidth. The GPU performance does not take into account the transfer time between the main memory and video memory. Effective bandwidth slightly exceeds the theoretical bandwidth due to texture cache.

the entire performance. Actually, the effective bandwidth on these resources reaches the theoretical bandwidth of 86.4 GB/s. We also find that the effective bandwidth slightly exceeds the theoretical bandwidth in some cases, showing a better utilization of texture cache, which saves the bandwidth between the video memory and the GPU. A detailed cache analysis can be seen in [29]. For further acceleration, we have to increase the arithmetic intensity [3] in the GPU code. This can be resolved by CUDA [6], which exposes on-chip memory to application developers. For example, shared memory in CUDA is useful to save the bandwidth of video memory, which we present later in Section 5.4.

Finally, we measure the performance on a 32-node cluster system. This homogeneous cluster has resource #2 as a computing node. As shown in Fig. 9, a linear

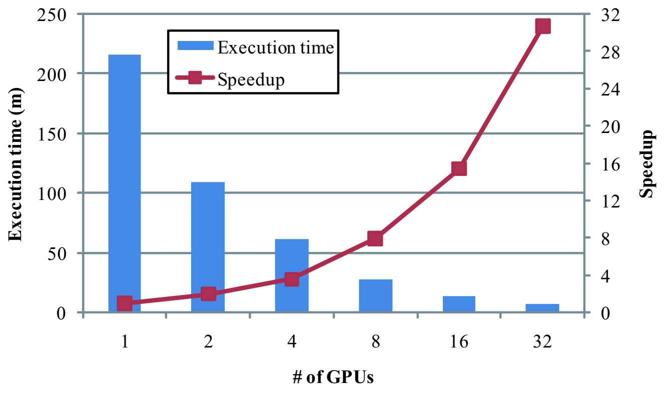


Fig. 9. Execution time and speedup of OpenGL-based implementation measured on a 32-node dedicated cluster.

speedup of 30.7X is observed when using 32 nodes for 256 (= N) query sequences. Thus, the performance can be scaled if there is enough tasks for resources. It also indicates that we need more query sequences to scale the system performance with the number of resources.

5.3. *Evaluation on Non-dedicated Resources*

We next analyze the performance on non-dedicated resources. In the following experiments, we use four resources #1, #5, #6, and #7 for five successive days. These resources are used by graduate students in our laboratory who perform research on general-purpose computation on the GPU. Each resource is powered on for 8 hours per day to develop GPU applications. Dedicated situations such as nights and holidays are not included in the following analysis.

Using four GPU-equipped machines, we observe an average of 202 scans per day (i.e., 8 hours): 62, 30, 71, and 39 scans on resources #1, #5, #6 and #7, respectively. Thus, by multiplying these results by the execution time in Fig. 6(a), the cumulative execution time reach 54, 30, 104, and 52 minutes on resources #1, #5, #6 and #7, respectively. The same total number of scans takes approximately 16 hours on the single CPU of resource #6. Thus, a single non-dedicated GPU can achieve almost the same throughput as two dedicated CPUs in the present case.

Figures 10 and 11 show idle period statistics observed from four different owners over two different months. In Fig. 10, 40% of idle periods are less than 5 minutes. Such short idle periods cannot be detected by our screensaver-based system, because the system currently requires at least 5 minutes of keyboard/mouse inactivity before idle detection. Thus, the system detects only 60% of owner inactivity. However, these detections cover at least 85% of idle time, as shown in the cumulative analysis in Fig. 11. It should be noted here that the ratio of detected time over the entire time is in the small range of 85% to 88% though we monitor four different owners over

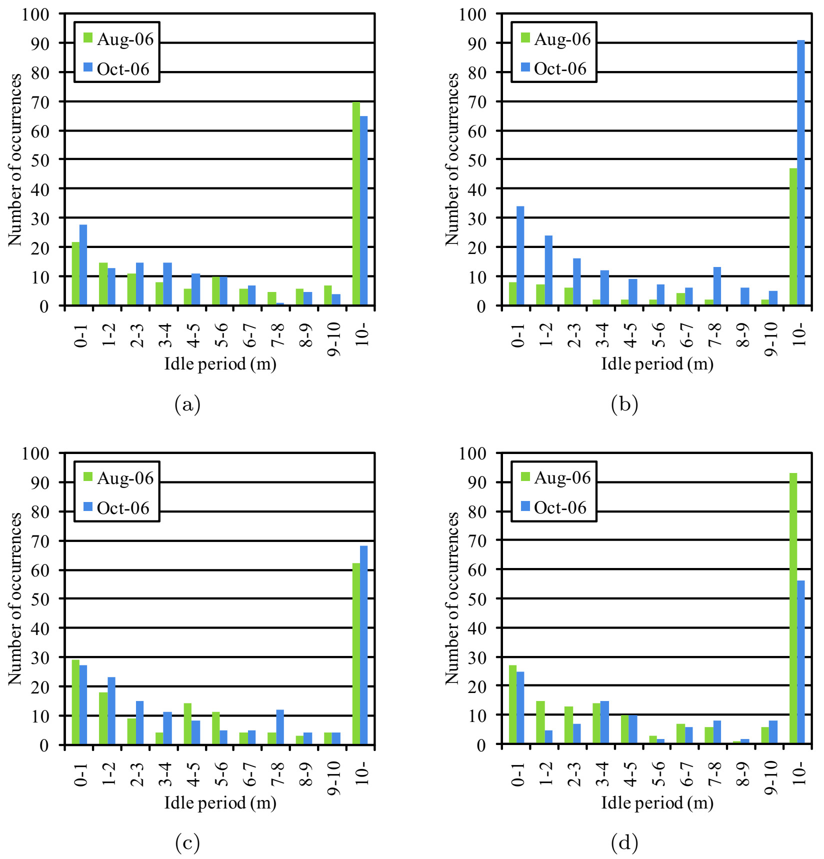


Fig. 10. Idle period statistics of two different months. (a)–(d) Breakdown of idle period length observed from four different owners. Our screensaver-based system detects idle periods of at least 5 minutes.

different periods of time. For example, Fig. 11(b) indicates that an owner shows a different behavior in each month, but there is no significant difference with respect to the distribution of idle period length. In this sense, we think that 5 minutes of waiting time seems a reasonable length for our 1-minute tasks.

The GPU exploitation contributes to minimize the relative overhead of task distribution in master-worker systems, because it allows us to increase the granularity of tasks. For example, the CPU-based implementation will cause many tasks that cannot be completed within a single idle period, because it requires at least 10 minutes of keyboard/mouse inactivity to process a 5-minute task (see Fig. 6(b)). In this case, we should decompose each task into smaller subtasks to complete each within a minute. This will decrease the number of suspended tasks but will increase the interaction between the server and resources. Since the server performance usually determines the scalability of master-worker systems, the GPU-accelerated code

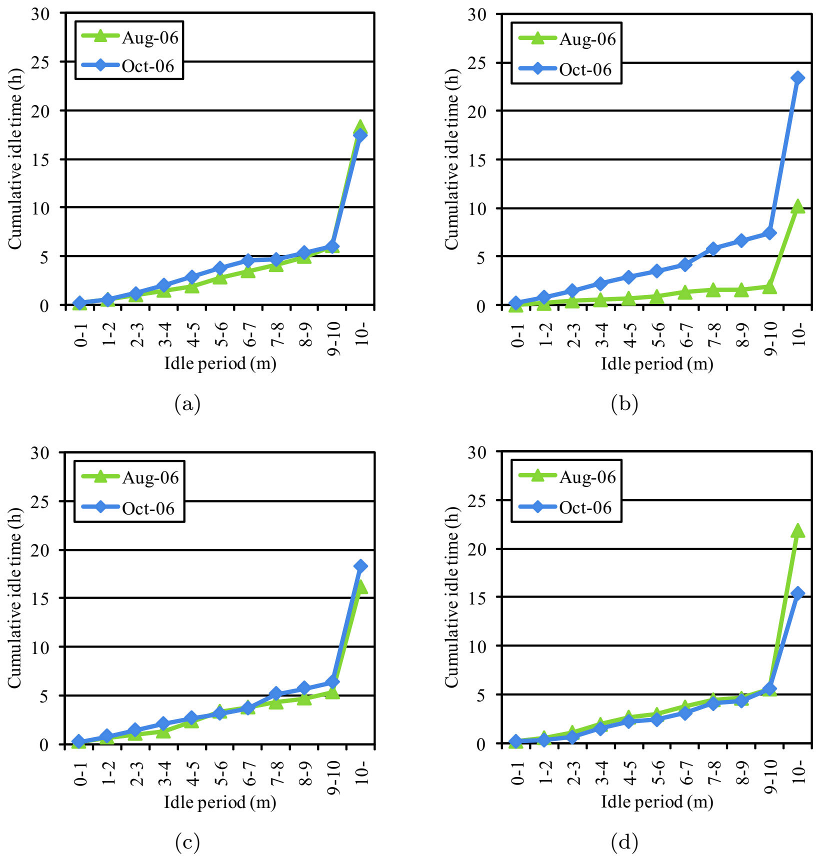


Fig. 11. Breakdown of idle period statistics. (a)–(d) Cumulative idle time for each owner.

contributes to improve not only the performance but also the scalability of grid systems.

With respect to energy consumption, we find that the alignment code consumes additional power of around 150W when running on a GeForce 8800 GTX card. From the owner's point of view, this increases the energy consumption of resource #2 by 75%. From the user's point of view, on the other hand, the energy efficiency computed from the additional part reaches 313 MFLOPS/W, which is close to 371 MFLOPS/W achieved by Blue Gene/P [30]. Though this is not a fair comparison, we feel that the GPU grid might become an attractive solution to save the energy for parameter-sweep applications.

5.4. *Evaluation Using CUDA-based Implementation*

We finally show how the performance can be increased using a CUDA-based implementation [15]. For the results in this section, we use a desktop PC based on a Core

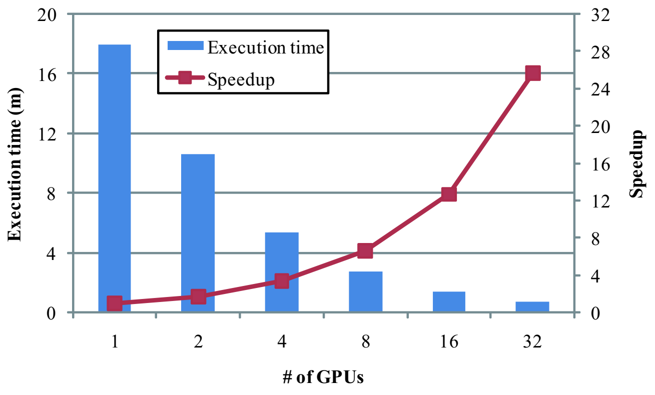


Fig. 12. Execution time and speedup of CUDA-based implementation measured on a 32-node dedicated cluster.

2 Duo 3-GHz CPU equipped with a GeForce 8800 GTX card. As we mentioned in Section 5.2, CUDA allows us to use on-chip memory to save the bandwidth of off-chip memory. This optimization is useful to increase the performance of memory-intensive applications, such as the Smith-Waterman alignment.

Our CUDA-based implementation completes an alignment task within 4.3 seconds, which is approximately 12 times faster than the OpenGL-based implementation. This performance improvement is mainly achieved by a data reuse technique [15] that significantly reduces the amount of data being fetched from off-chip video memory.

Similar to the OpenGL-based implementation, we measure the performance on non-dedicated resources. A graduate student operates the machine for 8 hours to create presentation slides. We find that the total idle time reaches approximately 1.5 hours. During this idle time, the system completes 597 scans in total. Since a single scan takes 4.3 seconds, the cumulative execution time reaches 42 minutes. Note here that the gap between 1.5 hours and 42 minutes is due to the 5 minutes of the waiting time needed before screensaver activation. That is, given an idle period of T minutes, only $T-5$ minutes can be used for task execution. For example, 50% of idle time will be wasted if $T = 10$.

Fig. 12 shows the performance on the 32-node cluster system mentioned in Section 5.2. As compared with OpenGL results in Fig. 9, the CUDA-based implementation significantly reduces the execution time with a 12X acceleration. However, the speedup in Fig. 12 is lower than that in Fig. 9, because CUDA-based tasks can be processed in a relatively shorter time. In other words, we have to increase the granularity of tasks (or jobs) to obtain the same speedup as that observed from OpenGL results. In summary, we think that CUDA-based implementations are essential to maximize the performance of grid systems.

6. Discussion

We have presented the effective performance of idle GPUs in non-dedicated environments. However, there is still some technical issues left in the system. For example, load balancing will be a critical problem in the GPU grid, because the GPU doubles the performance at every release as shown in Table 1. Actually, the best GPU time in Fig. 6(a) is at least 3.4 times shorter than others. With respect to the CPU, this ratio is reduced to a factor of 1.7. Furthermore, there is a significant performance gap between CUDA-compatible GPUs and others, as we presented in Section 5.4. Therefore, older GPUs possibly limit the entire performance of the grid system if the last task of a job is assigned to one of them. Thus, the GPU grid should be flexible enough to deal with such higher heterogeneity. Though our system solves this problem by the master-worker paradigm, it might be needed to screen out slow GPUs in the future. In particular, a task duplication mechanism [31] will be useful to minimize the turnaround time of jobs because the last task can be rapidly completed by duplication.

Another remaining issue is the reliability problem. Most graphics cards currently do not have error correction code (ECC) memory, because this capability is not essential to run graphics applications. In this situation, errors typically appear as wrong pixels on a screen. Such errors are not critical because it is hard for us to recognize errors hidden on the screen. However, this does not apply to scientific applications, which require correct results. For example, Maruyama et al. [32] find that 8 out of 60 GPUs cause approximately 10 bit-flips during 3-day run of a memory test program. Although the recent memory device such as GDDR5 partly supports ECC function [32], we think that the reliability can be realized by performing redundant computation on a large number of GPUs in the office and home.

Finally, it is important to identify the killer application of the GPU grid. Similar to existing grid systems, we think that parameter-sweep applications will be the killer application, because the GPU grid is an instance of grid systems. For example, the entire performance of grid systems is usually determined by the network bandwidth at the server, regardless of GPU acceleration. On the other hand, GPU acceleration is useful to increase the granularity of tasks in the master-worker system. In particular, the CUDA-compatible GPU helps us to maximize the system performance with relatively coarse-grained tasks, as we presented in Section 5.4.

We also think that parameter-sweep applications can be accelerated well in a single GPU environment. Such applications usually consist of a large number of independent tasks, making it easier to optimize the code to the CUDA-compatible GPU. For example, this type of applications have many parameters to be swept with the same code. Therefore, if there is common data accessed for different parameters, such data can be stored in on-chip shared memory to save the bandwidth of off-chip video memory [15, 34]. A memory coalescing technique [6] can be further exploited to maximize the effective memory bandwidth. Thus, using GPUs in grid systems will further accelerate parameter-sweep applications.

7. Conclusion

We have presented a parallel and distributed implementation of biological sequence alignment, aiming at demonstrating the effectiveness of utilizing idle GPUs in the office and home. Our implementation extends Liu's algorithm [14] to further accelerate database scanning in a non-dedicated, distributed environment.

Experimental results show that a single non-dedicated GPU can provide us almost the same throughput as two dedicated CPUs in our laboratory environment. Similar to this non-dedicated environment, the GPU-accelerated code also provides us higher alignment throughput in a dedicated environment, allowing us to use a task with five times more computation than the CPU-based code. The CUDA-based implementation demonstrates further acceleration over the OpenGL-based implementation. Thus, it contributes to increase application throughput using a less number of grid resources. We think that the GPU grid is useful to collect powerful resources hidden in current grid systems, leading to further acceleration of parameter-sweep applications in non-dedicated environments.

Future work includes the evaluation of the system in WAN environments. A hierarchal framework should be integrated into the system in order to manage thousands of resources.

Acknowledgment

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

References

[1] 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.
[2] 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," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.
[3] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient computation of sum-products on GPUs through software-managed cache," in *Proc. 22nd ACM Int'l Conf. Supercomputing (ICS'08)*, Jun. 2008, pp. 309–318.
[4] 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.
[5] R. J. Rost, *OpenGL Shading Language*, 2nd ed. Reading, MA: Addison-Wesley, Jan. 2006.
[6] nVIDIA Corporation, "CUDA Programming Guide Version 1.1," Nov. 2007, http://developer.nvidia.com/cuda/.
[7] AMD, "ATI CTM Guide," 2006, http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.
[8] Khronos OpenCL Working Group, "The OpenCL specification," 2009, http://www.khronos.org/registry/cl/.
[9] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: architecture and performance of an enterprise desktop grid system," *J. Parallel and Distributed Computing*, vol. 63, no. 5, pp. 597–610, May 2003.

[10] The Folding@Home Project, "Folding@home distributed computing," 2008, http://folding.stanford.edu/.

[11] GPUGRID.net, 2008, http://www.gpugrid.net/.

[12] Y. Kotani, F. Ino, and K. Hagihara, "A resource selection system for cycle stealing in GPU grids," *J. Grid Computing*, vol. 6, no. 4, pp. 399–416, Oct. 2008.

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

[14] 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.

[15] Y. Munekawa, F. Ino, and K. Hagihara, "Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU," in *Proc. 8th IEEE Int'l Conf. Bioinformatics and Bioengineering (BIBE'08)*, Oct. 2008, 6 pages (CD-ROM).

[16] S. Yamagiwa and L. Sousa, "Design and implementation of a stream-based distributed computing platform using graphics processing units," in *Proc. 4th Int'l Conf. Computing Frontiers (CF'07)*, May 2007, pp. 197–204.

[17] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU cluster for high performance computing," in *Proc. Int'l Conf. High Performance Computing, Networking and Storage (SC'04)*, Nov. 2004, 12 pages (CD-ROM).

[18] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, "CUDASA: Compute unified device and systems architecture," in *Proc. 7th Eurographics Symp. Parallel Graphics and Visualization (EGPGV'08)*, Apr. 2008, pp. 49–56.

[19] 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.

[20] M. Farrar, "Optimizing Smith-Waterman for the cell broadband engine," 2008, http://farrar.michael.googlepages.com/.

[21] 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.

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

[23] 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.

[24] R. Raman, M. Livny, and M. Solomon, "Resource management through multilateral matchmaking," in *Proc. 9th IEEE Int'l Symp. High Performance Distributed Computing (HPDC'00)*, Aug. 2000, pp. 290–291.

[25] I. Buck, K. Fatahalian, and P. Hanrahan, "GPUBench: Evaluating GPU performance for numerical and scientific applications," in *Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04)*, Aug. 2004, p. C-20.

[26] OpenGL Extension Registry, "Gl_ext_framebuffer_object," 2006, http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt.

[27] 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.

[28] X. Meng and V. Chaudhary, "Exploiting multi-level parallelism for homology search using general purpose processors," in *Proc. 11th Int'l Conf. Parallel and Distributed Systems (ICPADS'05), Volume II Workshops*, Jul. 2005, pp. 331–335.

[29] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra,"

in *Proc. Int'l Conf. High Performance Computing, Networking and Storage (SC'08)*, Nov. 2008, 11 pages (CD-ROM).

[30] W. chun Feng and K. Cameron, "The green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, Dec. 2007, http://www.green500.org/.

[31] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating System Design and Implementation (OSDI'04)*, Dec. 2004, pp. 137–150.

[32] N. Maruyama, A. Nukada, and S. Matsuoka, "Software-based ECC for GPUs," in *Proc. Symp. Application Accelerators in High Performance Computing (SAAHPC'09)*, Jul. 2009, http://saahpc.ncsa.illinois.edu/sessions/day2/session2/Maruyama_presentation.pdf.

[33] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl, "Large volume visualization of compressed time-dependent datasets on GPU clusters," *Parallel Computing*, vol. 31, no. 2, pp. 205–219, Feb. 2005.

[34] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for computing the costs of all-pairs shortest paths on the CUDA-compatible GPU," in *Proc. 6th Int'l Symp. Parallel and Distributed Processing and Applications (ISPA'08)*, Dec. 2008, pp. 284–291.