Sequence Homology Search using Fine-Grained Cycle Sharing of Idle GPUs

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

Abstract—In this paper, we propose a fine-grained cycle sharing (FGCS) system capable of exploiting idle graphics processing units (GPUs) for accelerating sequence homology search in local area network environments. Our system exploits short idle periods on GPUs by running small parts of guest programs such that each part can be completed within hundreds of milliseconds. To detect such short idle periods from the pool of registered resources, our system continuously monitors keyboard and mouse activities via event handlers rather than waiting for a screensaver, as is typically deployed in existing systems. Our system also divides guest tasks into small parts according to a performance model that estimates execution times of the parts. This task division strategy minimizes any disruption to the owners of the GPU resources. Experimental results show that our FGCS system running on two non-dedicated GPUs achieves 111–116% of the throughput achieved by a single dedicated GPU. Furthermore, our system provides over two times the throughput of a screensaver-based system. We also show that the idle periods detected by our system constitute half of the system uptime. We believe that the GPUs hidden and often unused in office environments provide a powerful solution to sequence homology search.

Index Terms—Distributed systems, performance of systems, fine-grained cycle sharing, homology search, Smith-Waterman algorithm, GPGPU, CUDA.

1 Introduction

Homology search is one of the most fundamental tasks in bioinformatics. The objective of this search is to detect fragments of database sequences that are similar to a given sequence, namely a query sequence. Finding such similar sequences is useful for understanding complex biological phenomena. For example, the findings may lead us to understand functional and evolutionary relationships between biological sequences.

Homology search can be performed by iteratively processing a pairwise algorithm that determines similar fragments between two sequences. The Smith-Waterman (SW) algorithm [1] is widely used for this type of search. Because it generates precise results, the SW algorithm is more sensitive than other heuristic methods, such as BLAST [2] and FASTA [3], which can miss weak similarities; however, the SW algorithm is computationally intensive. Although the algorithm is optimized using dynamic programming [4], its execution time is up to 40 times more than that of the typical heuristic methods [5]. Further hindering the use of the SW algorithm, biological databases are rapidly increasing in size owing to the advance in sequencing technology. For example, a protein sequence database called UniProtKB/Swiss-Prot [6] doubles its number of entries every two years, even though it consists of manually annotated sequences.

- *F. Ino and K. Hagihara are with the Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan.*
  E-mail: ino@ist.osaka-u.ac.jp
- *Y. Munekawa is with the School of Buddhism, Bukkyo University, 96 Kitahananobo-cho, Murasakino, Kita-ku, Kyoto 603-8301, Japan.*

Manuscript received on 15 Jan., 2011; revised 11 Aug. 2011.

To address these problems, many researchers are trying to speed up the SW algorithm by using accelerators, including the graphics processing unit (GPU) [7], [8] and the Cell Broadband Engine (CBE) [9], [10]. These accelerators successfully achieve a tenfold speedup over CPU-based implementations such as SSEARCH [3]. As an example, Liu *et al.* [7] implemented the SW algorithm using compute unified device architecture (CUDA) [11], which is a development framework for the NVIDIA GPU [12]. Their implementation, running on two GeForce GTX 295 cards, achieves a throughput of 16.0 giga cell updates per second (GCUPS), which is slightly higher than that of a CBE-based implementation [9].

In addition to the single-node systems mentioned above, some researchers have developed multi-node systems to achieve further acceleration. Singh *et al.* [13] developed a volunteer computing system that accelerates the SW algorithm on a pool of GPU-equipped resources. In general, volunteer computing systems have two different types of users: *hosts*, who donate their resources to the system, and *guests*, who run computationally intensive applications on the donated resources. Their system uses a screensaver-based middleware called Berkeley Open Infrastructure for Network Computing (BOINC) [14] to find idle resources from the pool. Since the BOINC middleware was originally designed for CPUs, host and guest applications could simultaneously run on the same GPU, causing significant system slowdown. For example, the frame rate of the display drops around 1 frame per second (fps). Such slowdown results in disruption to hosts who interactively operate their donated resources. Kotani *et al.* [15] extended the screensaver-based approach to prevent such resource conflicts by monitoring video memory usage. Their system achieves five times higher throughput than a CPU-based system [16].

Thus, previous work has demonstrated that homology search can be successfully accelerated using idle GPU cycles; however, the screensaver-based approach cannot detect the minutes of idle time that occur before the screensaver is activated. Since using a GPU typically provides a tenfold speedup over using a CPU, we propose that the idle times missed in screensaver-based systems can be utilized in achieving higher throughput for the SW algorithm. Such GPU exploitation contributes to deal with not only the performance issue but also the power consumption issue. In 2010, a GPU-based system called TSUBAME2.0 [17] ranked second in the Green500 list [18], and GPU-based systems occupied two of the top three places in the TOP500 list [19]. Thus, exploiting the power of accelerators is necessary for next-generation high-performance computers. GPU-accelerated machines in office environments can be a solution to this green issue because such machines are ordinarily powered on for interactive office work.

In this paper, we propose a fine-grained cycle sharing (FGCS) system capable of exploiting idle GPUs for accelerating sequence homology search. Contrary to screensaver-based approaches, our system is designed to identify and use idle periods spanning a few seconds. To realize this idea, our system detects short idle periods via event handlers monitoring keyboard and mouse inputs. Once detected, idle periods are used to run subtasks, namely small parts of a guest task. Subtask granularity is determined at runtime according to a performance model such that each part can be completed within hundreds of milliseconds. Such small subtasks allow us to minimize host disruption during guest task execution. Our system also intelligently selects from the pool of registered resources by utilizing the idle period length distribution, which approximately follows a power law distribution. Since our objective is to exploit idle resources in office settings, our system currently runs on Windows, which is the standard operating system for most office environments.

The remainder of the paper is structured as follows: Section 2 presents preliminaries including an overview of the SW algorithm and the CUDA-based implementation [8] employed in our system; Section 3 describes our system with a focus on the FGCS capability; Section 4 shows our experimental results; and Section 5 concludes our paper and describes future work. Details on related works, implementation issues, and additional evaluation results are presented in the supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.xxxx.xx.

2 GPU-Accelerated Smith-Waterman (SW) Alignment

2.1 Smith-Waterman (SW) Algorithm

The SW algorithm [1] gives an exact solution to the problem of pairwise local alignment. The algorithm finds the most similar part of two sequences according to the distance between them. The distance here is the minimum operational cost needed to transform one sequence into the other, with the insertion/deletion of a gap or the substitution of a symbol in the sequence.

Let $A = a_1a_2 \dots a_n$ and $B = b_1b_2 \dots b_m$ be a query sequence of length n and a subject sequence of length m, respectively. a_i represents the i-th symbol in A and b_j represents the j-th symbol in B, where $1 \leq i \leq n$ and $1 \leq j \leq m$. For all $1 \leq i \leq n$ and $1 \leq j \leq m$, the SW algorithm computes the similarity score $H_{i,j}$ of the optimal alignment ending at positions i and j in A and B, respectively. Let $E_{i,j}$ and $F_{i,j}$ be the similarity scores of the optimal alignment ending at the same position but with a gap in A and B, respectively. $H_{i,j}$ is then recursively given by

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

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

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

where α is the gap penalty for the first gap, β is the gap penalty for subsequent gaps, and $s(a_i, b_j)$ represents the cost to substitute symbol a_i with symbol b_j. Note that matrices H, E, and F are initialized with zeros.

According to Eqs. (1)–(3), the SW algorithm uses dynamic programming to compute $n \times m$ cells in similarity matrix H. The throughput in cell updates per second (CUPS) can be given by nm/T, where T represents the execution time for computing the matrix. After the matrix is filled, the algorithm performs backtracing from the cell with the maximum score in order to identify similar fragments. Thus, the SW algorithm consists of two processing phases: (1) matrix filling and (2) backtracing.

Since there are many subject sequences in the database, the SW algorithm must be iteratively processed with different subject sequences. Let S be the number of subject sequences in the database and Q be that of query sequences that compose a search *job*. We assume that a search job consists of Q *tasks* (i.e., Q search queries), each corresponding to a problem of local alignment between a query sequence and S subject sequences. In homology search, the matrix filling phase will be processed QS times to obtain QS matrices. In contrast, the backtracing phase can be skipped in most cases because we are usually interested only in high-scored cells. Thus, the time complexity of the backtracing phase can be approximated by $\mathcal{O}(Q)$ in practical situations. Accordingly, the CPU can quickly complete the backtracing phase after the GPU identifies high-scored cells at $\mathcal{O}(QS)$.

2.2 CUDA-Based Implementation

The CUDA-based implementation [8] employed in our system accelerates the matrix filling phase on the NVIDIA GPU. The implementation is based on Liu's parallelization scheme [20], which uses the OpenGL graphics library [21]. Both implementations compute similarity scores between a query sequence and S subject sequences. Two key aspects of our implementation are summarized as follows:

- Pipelined execution. Centralized CPU code iteratively loads a batch of L ($\leq S$) subject sequences from the database and invokes a kernel [11] to process the batch. A batch here corresponds to a *subtask*. Therefore, the kernel will be invoked $\lceil S/L \rceil$ times to complete a search task.

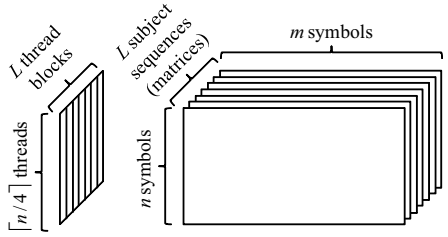

Fig. 1. Parallel matrix filling on a single GPU. Each thread block containing $\lceil n/4 \rceil$ threads is responsible for filling one of L matrices. Matrix cells are computed by $\lceil n/4 \rceil L$ threads from left-top to right-bottom.

This step-by-step alignment allows the CPU to pre-load the next batch while the GPU processes the current batch. Such overlapping execution is repeated until reaching the last entry of the database. Our original implementation uses the maximum batch size L_{max} of 32,768, which is restricted by the capacity of constant memory [8].

- Parallelization. As shown in Fig. 1, the kernel solves L pairwise problems simultaneously. Given no data dependence between different pairs of sequences, the implementation assigns a pairwise problem to a thread block (TB) [11], which is a group of threads that are not allowed to have data dependence between one another. Thus, the kernel generates L TBs for L matrices. Each TB contains $\lceil n/4 \rceil$ threads, and a thread is responsible for filling four successive rows of a matrix.

3 Homology Search System

In this section, we explain how resources are selected for job execution and how kernel execution time is controlled to minimize host disruption. The key to such minimization is reducing kernel execution time at the ideal tradeoff point between the throughput of guest applications and the delay of host applications. This reduction allows the GPU to periodically switch the active kernel, which occupies the resources in the GPU chip. If we do not reduce kernel execution time, hosts will suffer frequent system slowdown because the GPU cannot terminate kernel execution until its completion. In other words, the GPU denies interruption of kernel execution, which can make host applications wait for the completion of guest applications.

This wait time results in a delay in updating the frame buffer, namely the display, so that the length of the delay can be equivalent to the execution time of guest kernels. In this sense, it is unavoidable to perfectly eliminate the interference to hosts. Thus, we have decided to reduce the execution time of guest kernels to allow not only host applications to quickly occupy the resources, but also guest applications to run with nearly maximum throughput. This decision prevents the interference from being visible, because the wait time is fixed at certain value. Furthermore, we avoid running guest applications when the GPU is busy with host applications.

3.1 System Overview

According to the assumption in Section 2.1, there is no data dependence between different tasks. To deal with such independent tasks, our system employs a master-worker paradigm, which consists of a master and multiple worker machines. A simple illustration of our system can be found in Fig. 10 of the supplementary material. A worker corresponds to a machine registered in the system. Our system assumes that each worker has a single GPU. The master is the frontend machine that manages workers and jobs as follows:

- Resource management. The master is responsible for managing all registered resources. It has detailed resource information, such as hardware specifications, arithmetic performance, driver version, and video memory usage. Furthermore, busy or idle state information is gathered from the resource. Details of the interactions between the master and workers are presented below in Section 3.2.
- Job management. The master accepts grid jobs from guests, which are then queued to a job scheduler. The job scheduler decomposes accepted jobs into independent tasks (i.e., search queries) and assigns the tasks to idle resources in a first-in-first-out manner. The appropriate resources are selected according to resource information mentioned above. Guests are allowed to specify the resources to be used for their applications by using a matchmaking mechanism [22]. This mechanism requires guests to present a property description text file that describes the requested resources using attributes and comparison operators [15]. Details of job management are presented in Section 3.3.

Workers are responsible for monitoring themselves and for executing tasks as follows:

- Resource monitoring. Workers monitor their own resources and send status information back to the master. Network latency between the master and a worker may cause the master to assign tasks to a worker that has changed its status from idle to busy. In such a case, the worker cancels the assigned task and notifies the master of the failure. The failed task is then queued again to the scheduler for reassignment. Given the inherently short idle periods available, it is difficult for FGCS systems to eliminate such scenarios. The method for determining resource status is presented in Section 3.2.
- Task execution. The workers execute tasks assigned by the master and return their computation results. An assigned task is divided into small subtasks (i.e., batches), which are then processed in a pipelined, FGCS manner. Workers also terminate a task whenever the responsible resources turn out to be busy. Section 3.4 presents the performance model used for task division. Section 3.5 explains the algorithm that fills the matrix in an FGCS manner.

3.2 Idle Period Detection

Our system finds idle workers according to the following definition: a worker is idle if both its CPU and GPU are

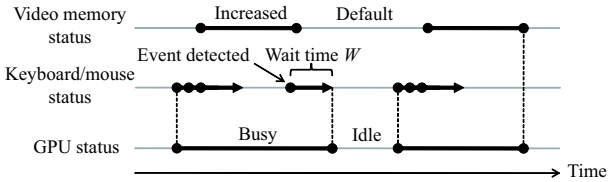

Fig. 2. Idle GPU detection. We assume a GPU to be idle if no keyboard or mouse activities are detected during the last W time units, and video memory usage does not change from a default value.

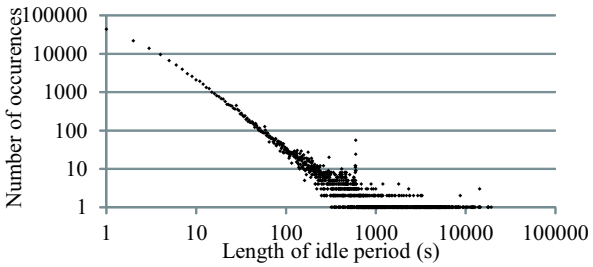

Fig. 3. Distribution of idle period lengths obtained from 14 desktop machines running for 20 work days. Each point (x, y) indicates that idle periods of x seconds are observed y times. Both axes are in logarithmic scale. See Fig. 18 of the supplementary material for typical distributions obtained from each machine.

idle [15]. We take CPU status into consideration, because our pipelined code overlaps file operations with kernel execution. If CPU status is ignored, guest tasks may be assigned to heavily loaded CPUs, which can significantly slow down the matrix filling phase because of slow file operations [8].

Our system determines CPU status according to CPU usage as follows: the CPU is idle if CPU usage is smaller than $\sigma\%$, where $0 \leq \sigma \leq 100$ represents the threshold for determining CPU status. This parameter can be specified by hosts to control the maximum host disruption that can be accepted during guest execution. The default value of σ is 10.

With respect to the GPU status, we extend the approach used in our previous screensaver-based system [15] to determine the GPU status for FGCS systems. Figure 2 illustrates how the system detects idle GPUs. The system assumes that a GPU is busy if one of the following two situations occurs on the worker: (1) the GPU executes a kernel; or (2) the GPU updates frame buffer, namely the display, because of the keyboard or mouse activity. The former can be identified by monitoring video memory usage, because the kernel consumes video memory. The latter can be identified by detecting keyboard or mouse events. A more detailed description of these identifications can be found in Section 7.1 of the supplementary material.

Since event detection does not directly capture the update of the frame buffer, it does not immediately imply the consumption of GPU cycles. To deal with this gap, the system assumes that the GPU is busy for a certain period after detection, as shown in Fig. 2. In the figure, W is the timeout delay needed to resume the idle state after event detection. The system experimentally uses a wait time (W) of one second.

3.3 Resource Selection

After workers detect idle periods, the master will be requested to assign tasks to them. Since our system exploits short idle periods, it is important to assign tasks to resources that are most likely to stay in the idle state for the longest amount of time. In this section, we describe how such resources are selected from a list $\mathcal{R}$ of idle resources.

According to preliminary experiments, we found that the length of idle periods approximates a power law distribution. Figure 3 plots the overall distribution of idle period lengths. These results were obtained from 14 desktop machines running for 20 work days in our university. Plots in Fig. 3 can be approximated by a straight line that implies a power law relation. Suppose here that we have two idle resources, A and B, which remain idle for one second and three seconds, respectively. Figure 3 indicates that the conditional probabilities of remaining idle in the next five seconds are 17% and 40% for resources A and B, respectively. Thus, the longer a resource remains idle, the higher is the conditional probability of remaining idle. That is, we can expect that the resources that remain idle for a long time will probably keep their idle state in the future. Therefore, our system gives higher priority to such long-idle-period resources. To do this, the system maintains the list of idle resources in descending order of length of the current idle period. Resources are then selected from the head of this resource list $\mathcal{R}$ in order to assign tasks to them. Section 7.2 of the supplementary material describes our resource selection algorithm in detail.

3.4 Performance Model

As mentioned above, host disruption will likely occur when the SW code is executed as a guest application. Such disruptions occur in the following two scenarios: (1) host applications experience slow performance because the SW code consumes CPU cycles, and (2) the update of the frame buffer can be delayed or even skipped because of the guest kernel running on the GPU. The former can be minimized by running the SW code with low priority. Such an execution configuration allows CPU resources to be assigned to host applications. The latter can be minimized by reducing required kernel execution times. Below, we describe how this reduction can be accomplished according to our performance model, which estimates the execution time of the matrix filling kernel.

Let k be the execution time of the matrix filling kernel. Let $\widehat{m}$ also be the total length of subject sequences processed by a single kernel invocation. The length $\widehat{m}$ can be given by $\widehat{m} = \sum_{l=1}^{L} m_l$, where m_l $(1 \leq l \leq L)$ represents the l-th subject sequence processed by the kernel invocation.

Our model must capture the performance bottleneck of the matrix filling kernel in order to control kernel execution time k. According to our previous profiling analysis [8], we found that the kernel consists of instruction-bound code rather than memory-bound code. Therefore, we decide to take the time complexity of the matrix filling into consideration. The model also captures the overhead of switching threads because

the switching overhead can be regarded as a parallelization overhead, which does not occur during serial execution. Thus, kernel execution time k can be modeled by adding this additional overhead to the time complexity as follows:

$$k = X \cdot n\widehat{m} + Y \cdot \lceil n/4 \rceil L, \quad (4)$$

where X and Y represent the coefficient for the time complexity and that for the thread-oriented overhead, respectively. These coefficients are experimentally determined for hardware and input configurations. See Section 8.1 of the supplementary material for details.

The first term of Eq. (4) represents the time complexity of the matrix filling phase, which can be given by the number $n\widehat{m}$ of matrix cells to be filled by threads. The second term represents the number $\lceil n/4 \rceil L$ of threads generated by a kernel invocation. We assume a simple linear model that increases the switching overhead with the number of threads, because the GPU adopts a highly threaded architecture that overlaps memory transactions with data-independent computation.

3.5 Matrix Filling for Fine-Grained Resource Sharing

The SW code must be modified to ensure kernel completion within a short timeframe. To achieve this, our approach is to dynamically select the appropriate value of L before kernel invocation. In this section, we describe how we modify the code to achieve this dynamic behavior.

Our modified code, which can be found in Fig. 12 of the supplementary material, requires additional inputs as compared to the original code: (1) maximum values L_{max} of the batch size and (2) K of the kernel execution time be specified by the system. After loading the query sequence, the code initializes coefficients X and Y according to the length n of the query sequence and the hardware of the graphics card. It then iteratively loads a subject sequence from the database. The number L of loaded subject sequences is determined by Eq. (4), which estimates kernel execution time k such that k can be approximated with specified time K. After this input/output (I/O) operation, the code invokes the matrix filling kernel to compute matrices and sends results back to main memory. This invocation is further iterated until the last entry of the database is processed.

With respect to specified time K, we use a default value of 100 milliseconds. This value has typically been identified as the maximum delay in a GUI because it is regarded as the limit of human perception for changes in a GUI [23]. As mentioned above in Section 3, specified time K may be equivalent to the delay in updating the frame buffer. Thus, we expect that the maximum delay will be approximately 100 milliseconds.

4 Experiments

We compare our FGCS system with a screensaver-based system [15], [16] in terms of alignment throughput. Before describing this comparison, we evaluate the accuracy of the performance model to investigate whether kernel execution time is actually controlled by the performance model. We also present a case study to better understand the impact of our FGCS system. See Section 8 of the supplementary material for a detailed explanation of experimental setup.

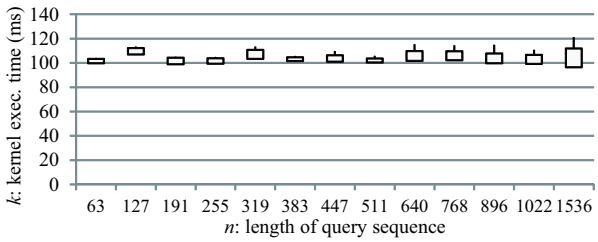

Fig. 4. Distribution of kernel execution times versus query sequence lengths n. The line segments display the range between the minimum and maximum times and vertical bars represent the 95% confidence intervals. Results are measured with specified time K of 100 milliseconds.

4.1 Accuracy of Performance Model

By using the coefficients presented in Section 8.1 of the supplementary material, we investigated kernel execution times to evaluate the accuracy of the performance model. We executed our modified code with different lengths n of the query sequence. Figure 4 shows the distribution of kernel execution time obtained on the GTX 285 card. For each n, the bar shows the minimum value, the maximum value, and the 95% confidence interval. Since the code processes a series of batches to complete a task, these values are computed from the first batch to the second-to-last batch. The last batch is not included because it does not have sufficiently long subject sequences to keep the execution time close to specified time $K = 100$.

In Fig. 4, 95% of kernel execution times range from 96 to 111 milliseconds; the mean time is 104 milliseconds. These results are acceptable for our FGCS system, because its purpose is to complete kernel executions within a relatively short amount of time rather than to exactly obtain kernel execution times of K. We obtained similar results on the 8800 GTX card, which can be found in Section 8.2 of the supplementary material.

To investigate the overhead of task division, we measured the total execution time spent for a query sequence. In contrast to kernel execution time, the total execution time depends on length n. Our modified kernel takes approximately 5.4 and 19.6 seconds to process a query of length $n = 511$ on the GTX 285 and 8800 GTX cards, respectively. Since the original kernel [8] takes 5.3 and 17.6 seconds to process the same query on each card, the overhead is 2% and 11% on the GTX 285 and 8800 GTX cards, respectively. Thus, our kernel successfully controls kernel execution time around 100 milliseconds despite hardware differences, but reduces the efficiency on a slow card.

An important behavioral aspect of the original kernel is that it increases kernel execution times as the code invokes more kernels. This occurs because of subject sequence order. The total length $\widehat{m}$ of subject sequences increases with the number of kernel invocations, because the original kernel processes a fixed number L_{max} of sorted subject sequences. For example, $\widehat{m}$ ranges from 277,475 to 8,533,652 amino acids in the original kernel. In contrast, our modified code dynamically decreases the number L of subject sequences to keep kernel execution times at approximately $K = 100$ milliseconds.

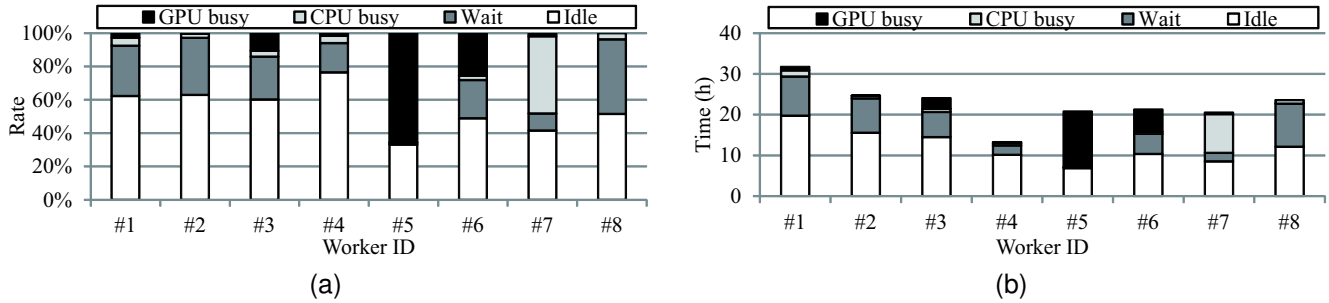

Fig. 5. Worker machine resource statistics. Resource status obtained during owner use over four days presented as (a) percentages and (b) times.

4.2 Case study

The following case study was performed in our laboratory over four days, each day from 10:00 to 18:00. Thus, we do not consider offline time during which hosts can be continuously exploited for guest applications. Such offline scenarios are appropriately handled by existing systems. In contrast, our system handles the non-dedicated situation. In this case study, our system iteratively processed an alignment job containing eight different query sequences. The length n of the query sequences ranged from 63 to 511 amino acids. See Table 1 of the supplementary material for the specifications of worker machines.

We first analyzed the breakdown of resource status and the results are shown in Fig. 5. Idle time occupies, on an average, 54% of the system uptime. In addition, 25% of system uptime is spent waiting, as described in Section 3.2. This wait time corresponds to short idle periods of less than $W = 1$ second, which are not exploited by our system. Workers #1, #2, #4, and #8 have short GPU/CPU busy times, such that 90% of the system uptime consists of idle time and wait time. Thus, the owners of these machines perform their research using little of their CPU and GPU resources. Such work typically includes document editing, PDF file reading, and so on. Conversely, workers #3, #5, and #6 have relatively long GPU busy times that utilize over 10% of system uptime. These machines are primarily used for developing GPU applications. Worker #7 has the longest CPU busy time. The owner of this machine primarily develops CPU applications.

With respect to the time scale shown in Fig. 5(b), worker #1 has the longest uptime of 32 hours, while worker #4 has the shortest uptime of 13 hours. In total, the system uptime is 180 hours, including 98 hours of idle time. Clearly, the eight worker machine owners have different usage styles resulting in different resource status. More details on task execution statistics are presented in Section 8.3 of the supplementary material.

Since the database is sent to workers before experiments, communication time does not limit the alignment throughput in the study. The execution time is 67 hours in total, which includes kernel execution time of 64 hours and communication time of 3 hours. Scalability and distribution issues should be considered in future work.

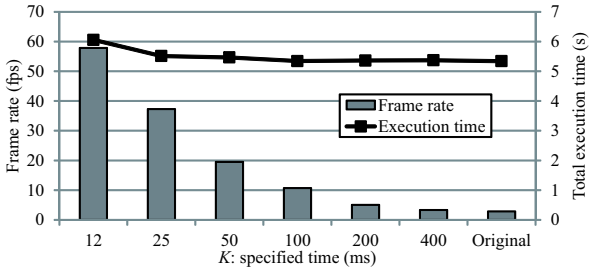

Fig. 6. Frame rates and total execution times measured using length $n = 511$ of query sequence.

4.3 Degree of Interference

Our approach may cause greater interference to hosts than previous screensaver-based approaches do. This additional interference occurs (1) when the keyboard or mouse is operated at intervals shorter than the screensaver timeout period while having a CPU load of less than 10% and no consumption of video memory, and (2) when an idle period is incorrectly detected during a busy state (i.e., when a false negative occurs).

The first case causes the delay of kernel termination when moving from idle state to busy state. According to our investigation mentioned below, we found that document editing and web browsing apply in this case. Thus, the interference manifests as a short delay of around $K = 100$ milliseconds, which occurs at the end of a rest period of more than $W = 1$ second.

The second case decreases frame rates due to guest task execution. However, we could not observe a false negative except the instant delay mentioned above. Thus, a significant system slowdown is prevented during the case study. To assess the worst case of interference, we measured the frame rate while executing guest tasks. Figure 6 shows the frame rate measured using Fraps [24] on the GTX 285 card. The frame rate linearly increases as we decrease specified time K. In particular, our modified kernel keeps the frame rate of 10.7 fps, whereas the original kernel drops the rate to 2.9 fps. A similar behavior was observed on the 8800 GTX card (see Fig. 15 of the supplementary material). We therefore consider that our rate is acceptable for office workers who edit text files at 10.7 characters per second at most, even though idle periods are continuously detected on busy workers.

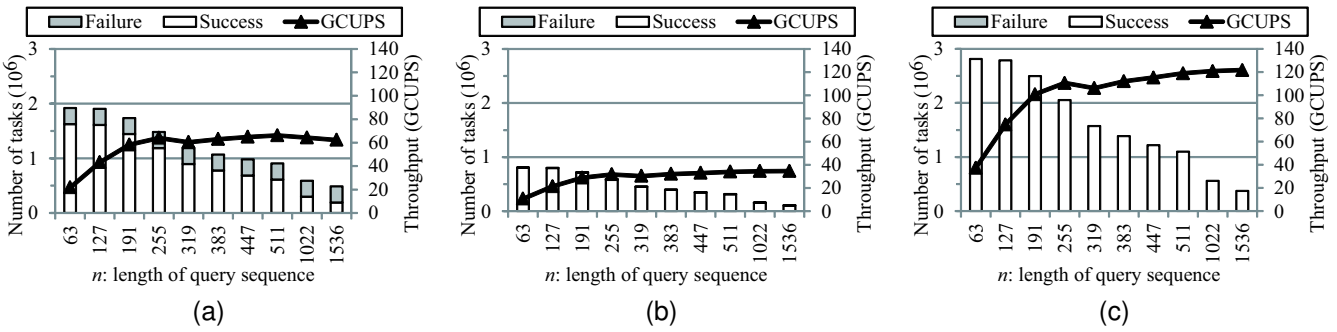

Fig. 7. Comparison to previous systems in terms of alignment throughput. Results on (a) our FGCS system, (b) screensaver-based system, and (c) cluster system. Presented results are sum of values estimated on 14 workers.

The interference also depends on the accuracy of idle period detection, which is determined by the following three factors: (1) the accuracy of event handlers, (2) the wait time W needed to resume the idle state after event detection, and (3) the accuracy of the idle status definition. Since event handlers are originally designed for interactive GUI operations, we conclude that they are accurate enough to deal with idle periods measured in seconds. The wait time W of one second can result in a false positive, because our system cannot detect the first second of idle periods, as illustrated in Fig. 2. With respect to the correctness of the idle status definition, we investigated the resource status using various host scenarios such as graphics rendering, video viewing, music playing, virus scanning, I/O loading, and other types of CPU/GPU computing. Our system detects the idle state during document editing and web browsing. The system does not execute guest kernels under the remaining scenarios, so that we could not observe significant system slowdown.

From the point of view of quality of service, we consider that the specified time K can be used to estimate the minimum frame rate on workers. As mentioned in Section 3.5, time K can be equivalent to the delay in updating the frame buffer. Thus, the frame rate will be around $1/K$. Although this does not strictly guarantee the frame rate, we can roughly control the minimum rate by specifying the value of K, as shown in Fig. 6.

4.4 Comparison with Previous Systems

In this section, we compare our FGCS system with a screensaver-based system [16] and a cluster system; our comparison focuses on alignment throughput. The screensaver-based system requires five minutes of wait time to determine that the resource actually is idle and has no interaction with hosts. We regard the screensaver-based system as a coarse-grained cycle sharing system, while we regard the cluster system as a dedicated system. To compare these systems fairly, we used logs obtained on 14 machines in our university. These logs contain the start and end times of idle states detected during 20 work days. Across all machines, the system uptime is 1668 hours in total. The overall distribution of idle periods is shown in Fig. 3.

Using these logs, we simulated the behavior of the three systems to evaluate the throughput with varying lengths n of query sequence. Our simulation assumes that (1) all workers have a GTX 285 card, (2) it takes 0.17 seconds to send a query sequence and receive computational results, (3) and the screensaver-based and cluster systems execute the original version [8] of the SW code.

To compute the throughput for each system, we counted the number of successfully completed tasks. Figure 7 shows simulation results for the three systems. Our FGCS system detects 1011 hours of time, which is 2.1 times more than that the 479 hours detected by the screensaver-based system. This improved detection leads to two times more throughput than the screensaver-based system. As an example, the FGCS system achieves an alignment throughput of 64.0 GCUPS when $n = 255$ (Fig. 7(a)), while the screensaver-based system achieves that of 31.7 GCUPS (Fig. 7(b)). Again with $n = 255$, the FGCS system throughput is 58% of the throughput achieved by the cluster system (Fig. 7(c)). Results indicate that adding two graphics cards into a desktop machine in an office environment is equivalent to adding a graphics card into a computing node in a dedicated cluster. With respect to SW alignment, we believe that a cluster of P GPUs can be built by adding $2P$ GPUs into desktop machines ordinarily used for office work.

Figure 7 also shows the number of successful and failed tasks. As we increase length n from 64 to 1536, execution time per task increases from 2.1 to 16.0 seconds. Because of this increase in execution time, both the FGCS and screensaver-based systems reduce the number of successful tasks; however, throughput remains steady approximately 65 GCUPS in our system and 32 GCUPS in the screensaver-based system. The cluster system achieves the maximum throughput for all lengths n because it uses fully idle resources.

Note that the cluster system shows decreased throughput when $n \leq 191$. This lower throughput is consistent with decreases observed in the FGCS and screensaver-based systems. As such, the throughput of such short query sequences is determined by kernel performance rather than cycle sharing overhead.

Thus, we conclude that our FGCS system efficiently exploits idle periods of less than five minutes, periods of time that cannot be exploited in the screensaver-based system. In contrast, the FGCS system fails to complete 155 times more tasks than that of the screensaver-based system. This is due

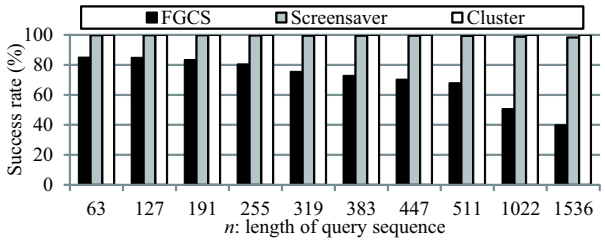

Fig. 8. Success rate of task execution with different lengths (n) of query sequence.

to the power law distribution of idle period lengths, which we mentioned in Section 3.3.

Figure 8 shows the success rate of task execution with different lengths n of the query sequence. In the FGCS system, the success rate decreases as we increase n, because the execution time per task increases with n. We observe a success rate of at least 75% if a task completes within three seconds. Therefore, the efficiency of the system can be increased by future GPUs, which will complete the same task within shorter time. Since our system cancels task execution when workers turn out to be busy, we can further increase the alignment throughput by supporting a checkpoint restart capability that resumes task execution from the last kernel invocation.

4.5 Scheduling Tradeoff

There is a tradeoff between the task granularity and the task success rate. As we increase the task granularity, the master increases the efficiency of master-worker execution but produces more failed tasks. As we decrease the task granularity, we can use more idle periods with a higher success rate but the master can limit the entire performance. Thus, it is important to find the best tradeoff point between the guest throughput and the success rate. Since a lower success rate involves more cancelled tasks, which in turn slow down the master, finding the tradeoff point will play an important role in increasing the efficiency of larger systems using our model. Section 8.4 of the supplementary material gives further analysis on this tradeoff issue.

4.6 Flexibility for Other Applications

Our FGCS approach requires changes to the guest application to enable control of fine-grained tasks. In our approach, the following three requirements must be satisfied in order to adapt an application: (1) the application must run with the master-worker execution model, (2) a performance model must be constructed to estimate the kernel execution time k, and (3) the application code must be rewritten such that the kernel can efficiently complete within a short timeframe K. More details regarding this flexibility issue are presented in Section 8.5 of the supplementary material.

5 Conclusion

In this paper, we presented an FGCS system capable of accelerating homology search using idle GPUs in an office environment. Our system exploits short idle periods on the order of seconds, which are not captured via existing screensaver-based systems. To achieve this, our system monitors keyboard and mouse activities using event handlers and executes small parts of tasks such that each part can be completed within hundreds of milliseconds. Such an approach prevents hosts from experiencing frequent system slowdown, allowing guest applications to run with minimal host disruption.

We also presented a performance model that estimates kernel execution time of the SW algorithm. This model is useful for running guest tasks at the best tradeoff point between throughput of guest applications and the delay of host applications. The scheduling algorithm in our system takes advantage of statistical analysis indicating that the idle period length distribution follows a power law. According to this analysis, our system assigns tasks to resources that have been idle for long periods of time.

We performed experiments in our laboratory and have found that half of the system uptime can be utilized in achieving higher throughput for the SW algorithm. The simulation results show that the FGCS system running on 14 GTX 285 cards achieves a throughput of 64.0 GCUPS, which is equivalent to 58% of the throughput achieved by the cluster system. We believe that the GPUs hidden (and often unused) in office environments provide a powerful solution to the problem of homology search. We also believe that FGCS systems will become a strong driving force to enhance the GPU architecture.

Future work would include exploitation of short idle periods of less than one second. We found that such idle periods occupy approximately 25% of system uptime. We also plan to develop a resume capability to increase the efficiency of task execution when faced with the issue of workers cancelling a task. We think that such a capability is useful to scale the performance with the number of workers.

Acknowledgments

The authors would like to thank the anonymous reviewers for helpful comments to improve their paper. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (B)(23300007), Young Researchers (B)(23700057), and the Okawa Foundation for Information and Telecommunications.

References

[1] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Molecular Biology*, vol. 147, pp. 195–197, 1981.
[2] 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.
[3] 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.
[4] R. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.
[5] T. Rognes and E. Seeberg, "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, Aug. 2000.
[6] 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.

[7] 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, 10 pages.
[8] Y. Munekawa, F. Ino, and K. Hagihara, "Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs," *IEICE Trans. Information and Systems*, vol. E93-D, no. 6, pp. 1479–1488, Jun. 2010.
[9] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz, "SWPS3 — fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2," *BMC Research Notes*, vol. 1, no. 107, Oct. 2008, 4 pages.
[10] M. Farrar, "Optimizing Smith-Waterman for the Cell Broadband Engine," 2008, 5 pages. [Online]. Available: http://sites.google.com/site/farrarmichael/SW-CellBE.pdf
[11] NVIDIA Corporation, "CUDA Programming Guide Version 2.3," Jul. 2009. [Online]. Available: http://developer.nvidia.com/cuda/
[12] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.
[13] A. Singh, C. Chen, W. Liu, W. Mitchell, and B. Schmidt, "A hybrid computational grid architecture for comparative genomics," *IEEE Trans. Information Technology in Biomedicine*, vol. 12, no. 2, pp. 218–225, Mar. 2008.
[14] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proc. 5th IEEE/ACM Int'l Workshop Grid Computing (GRID'04)*, Nov. 2004, pp. 4–10.
[15] 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, Dec. 2008.
[16] F. Ino, Y. Kotani, Y. Munekawa, and K. Hagihara, "Harnessing the power of idle GPUs for acceleration of biological sequence alignment," *Parallel Processing Letters*, vol. 19, no. 4, pp. 513–533, Dec. 2009.
[17] Tokyo Institute of Technology, "TSUBAME2," 2010, http://www.gsic.titech.ac.jp/en/tsubame2/.
[18] W. chun Feng and K. Cameron, "The Green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, Dec. 2007. [Online]. Available: http://www.green500.org/
[19] H. Meuer, E. Strohmaier, H. D. Simon, and J. Dongarra, "TOP500 supercomputing sites," Nov. 2010. [Online]. Available: http://www.top500.org/
[20] 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.
[21] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, 5th ed. Reading, MA: Addison-Wesley, Aug. 2005.
[22] R. Raman, M. Livny, and M. Solomon, "Policy driven heterogeneous resource co-allocation with gangmatching," in *Proc. 12th IEEE Int'l Symp. High Performance Distributed Computing (HPDC'03)*, Jun. 2003, pp. 80–89.
[23] J. R. Dabrowski and E. V. Munson, "Is 100 milliseconds too fast?" in *CHI'01 extended abstracts on Human factors in computing systems*, Mar. 2001, pp. 317–318.
[24] Beepa Pty Ltd., "Fraps: real-time video capture & benchmarking," 2011, http://www.fraps.com/.

PLACE PHOTO HERE

Fumihiko Ino (S'01–A'03–M'04) 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.

PLACE PHOTO HERE

Yuma Munekawa received the B.E. and M.E. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 2008 and 2010, respectively. He is currently working toward the B.A. degree at the School of Buddhism, Bukkyo University, Kyoto, Japan. His current research interests include high performance computing, grid computing, and systems architecture and design.

PLACE PHOTO HERE

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. His research interests include the fundamentals and practical application of parallel processing.