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

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

6 Related Work

There are some existing research projects that exploit idle GPUs for acceleration of scientific applications. For example, the Folding@Home distributed computing system [1] accelerates protein-folding simulations by a volunteer computing approach. This system is developed using BOINC middleware [2], which deploys a screensaver to find idle resources. As mentioned in Section 1, this can cause significant host disruption due to resource conflicts on the GPU.

With respect to homology search, Singh *et al.* extended Liu's single-node implementation [3] to achieve further acceleration on a BOINC-based grid system. Their system exploits two types of parallelism, one for coarse-grained parallelization on computing nodes, and the other for fine-grained parallelization on the GPU. The system throughput reaches 6.4 GCUPS on ten dedicated nodes, each equipped with a G70 or G80 card [4]. A similar system proposed by Ino *et al.* [5] solves the problem of resource conflicts. Their system monitors the video memory usage to identify busy or idle states of the GPU [6]. Since usage is managed by the graphics driver, this monitoring strategy requires low overhead without disturbing host applications that run on the GPU. They showed that a non-dedicated GPU in their laboratory provides nearly the same throughput as two dedicated CPUs in a cluster system.

To avoid host disruption altogether, GPU-accelerated grid systems have relied on the use of screensavers. Further, the GPU architecture does not currently support pre-emption. Although NVIDIA plans to support pre-emption in future architectures, such as the Maxwell architecture to be released in 2013 [7], we need a mechanism to find the best tradeoff point between the throughput of guest applications and the undisturbed performance of host applications. Therefore, we emphasize that the contribution of our paper will be useful to design such future architectures and to discuss the impact of exploiting the power of future GPUs. For example, resource selection and fine-grained execution mechanisms that control the frame rate on resources can be applied to future pre-emptive systems.

With respect to accelerating the SW algorithm, many researchers have been attempting to implement the algorithm on accelerators. To the best of our knowledge, Liu *et al.* [8] were the first to implement the algorithm on the GPU. Their implementation uses the OpenGL graphics library [9]. On a GeForce 7900 GTX card, their achieved throughput reached 0.67 GCUPS, which is sixteen times higher than that of a CPU-based native implementation [10]. Manavski *et al.* [11] developed the first implementation that accelerates the SW algorithm using CUDA [12]. On two GeForce 8800 GTX cards, their implementation achieved a throughput of 3.6 GCUPS, which is 20% higher than that of a CPU-based implementation [13] optimized with single instruction multiple data (SIMD) instruction sets [14]. Additional optimized implementations [15], [16], [17], [18] have been proposed to utilize on-chip memory resources of the GPU. The throughput achieved by these implementations range from 5.6 to 9.6 GCUPS, depending on the graphics card.

Field-programmable gate array (FPGA) based solutions have achieved the highest throughput for homology search. Zhang *et al.* showed that an XD1000 FPGA board can achieve 25.6 GCUPS at peak performance. Similar results were reported by Li *et al.* [19], Storaasli *et al.* [20], and Meng *et al.* [21]. Compared to other accelerators, the FPGA board is a much more expensive and specialized piece of hardware. Since our objective is to exploit idle cycles in a typical office setting (i.e., without the need for high-end hardware), we focus our research on commercial off-the-shelf graphics cards.

7 Implementation Issue

Figure 10 shows an overview of our master-worker system. Since the master has to exchange a large number of messages between workers, we have implemented our system using an efficient threading model called Input/Output Completion Port (IOCP) [22]. This model is designed for performing multiple asynchronous I/O operations on a multiprocessor system, thus increasing overall message throughput.

- *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.*

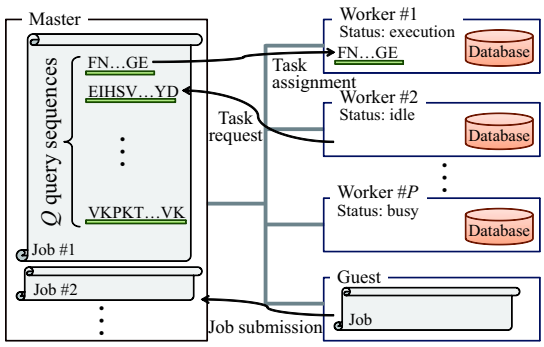


Fig. 10. Overview of our FGCS system architecture. A search job consists of Q independent queries. A task is associated with each query and is assigned to idle resources via the master-worker paradigm.

7.1 Idle Period Detection

Our system computes CPU usage by the GetSystemTimes function, which returns idle time, kernel time, and user time (see [23] for details). This function is provided as a part of the Windows API [24].

The kernel execution on the GPU can be identified by monitoring video memory usage, because the kernel consumes video memory. Video memory usage is obtained via the GetCaps function, which is part of the DirectDraw API [25]. Since this function obtains usage information from the graphics driver, it does not disturb host applications running on the GPU. Thus, the overhead is less than a few milliseconds.

The update of the frame buffer frequently occurs on recent Windows systems, which employ Windows Aero as a graphical user interface (GUI). Since this GUI is implemented by the DirectX graphics library [25], the update of the frame buffer consumes some GPU cycles, but does not affect video memory usage. Rather than using a screensaver-based approach, our system uses event handlers to monitor keyboard and mouse activities. Our system detects keyboard and mouse events by using the SetWindowsHookEx function, which is part of the Windows API.

Given that the overhead of activating a screensaver is too large to find short idle periods on the order of hundreds of milliseconds, we opt for an event-driven approach. In general, it takes approximately 200 milliseconds to activate a screensaver [6]; although this overhead is acceptable for coarse-grained cycle sharing systems, it is not small enough for FGCS systems.

7.2 Resource Selection

Figure 11 shows pseudocode of our resource selection algorithm. The master performs the task assignment using the following threads: (1) a main thread that processes the algorithm shown in Fig. 11, and (2) message handling threads that update the list $\mathcal{R}$ of idle workers in response to messages they receive. The message handling threads insert a resource entry to list $\mathcal{R}$ in a sorted order when they receive a request message from the resource indicating that the resource has become idle. Conversely, they remove the entry when they receive a failure

Input: A set $\mathcal{T}$ of tasks that compose a search job
Input: A list $\mathcal{R}$ of idle workers
Input: Property description D specified by guests
Output: A master-worker schedule with matchmaking

1: **while** $\mathcal{T} \neq \emptyset$ **do**
2: $r \leftarrow$ head of list $\mathcal{R}$ $\{r = NULL$ if $\mathcal{R} = \emptyset\}$
3: **while** $r \neq NULL$ **do**
4: **if** Worker r matches property description D **then**
5: Execute task $t \in \mathcal{T}$ on worker r in a non-blocking manner
6: $\mathcal{T} \leftarrow \mathcal{T} - \{t\}$
7: **end if**
8: $r \leftarrow r.next$ $\{r = NULL$ if the tail of $\mathcal{R}$ has been reached$\}$
9: **end while**
10: Wait for message handling threads to update $\mathcal{R}$
11: **end while**

Fig. 11. Resource selection algorithm with matchmaking [26]. This algorithm is executed by a main thread running as master. List $\mathcal{R}$ of idle workers is updated via message handling threads.

message indicating that the resource has become busy. Thus, selection is simply a matter of assigning a task to the resource located at the head of the list. Since list $\mathcal{R}$ is a shared resource between all threads, critical sections are used to implement the necessary locking mechanism.

This locking mechanism may cause some performance overhead. Since list $\mathcal{R}$ is shared between the main thread and the message handling threads, the overhead emerges as a performance bottleneck (1) when workers frequently change their resource status and (2) when the master lacks sufficient performance to manage a large number of workers. However, these multiple threads are required by the IOCP, which performs multiple simultaneous asynchronous operations. If we do not use the IOCP, messages between the master and workers are serially processed by non-overlapped operations. Such operations can be regarded as a locking mechanism that serializes incoming or outgoing messages. Our locking mechanism works for a list in main memory, which has a smaller overhead than the network device.

7.3 Modified Matrix Filling Code

Figure 12 shows pseudocode of the SW alignment algorithm to automatically control kernel execution times for the FGCS system. To simplify its description, we present a non-pipelined version of the code.

According to Eq. (4), our code loads the appropriate number L of subject sequences. In other words, the code dynamically changes the number of TBs processed by a kernel invocation. Recall here that a TB is responsible for processing a pair of a query sequence and a subject sequence. Thus, the kernel execution time is controlled by the number of TBs.

TABLE 1
Specification of experimental machines. GIPS stands for giga instructions per second, which determines the alignment throughput of our instruction-bound kernel.

Specification	Worker ID							
	#1	#2	#3	#4	#5	#6	#7	#8
OS	Windows XP	Windows XP	Windows XP	Windows XP	Windows XP	Windows XP	Windows XP	Windows Vista
CPU	Xeon E5440	Xeon E5450	Xeon X5450	Xeon E5450	Xeon E5450	Xeon E5450	Xeon E5450	Xeon X5472
	2.83 GHz	3.00 GHz	3.00 GHz	3.00 GHz	3.00 GHz	3.00 GHz	3.00 GHz	3.00 GHz
Main memory (GB)	32	8	32	8	8	16	8	8
GPU	GeForce	GeForce	GeForce	GeForce	GeForce	Quadro	GeForce	GeForce
	GTX 285	GTX 285	GTX 285	GTX 285	GTX 285	FX 5800	GTX 295	8800 GTX
VRAM capacity (MB)	1024	1024	2048	1024	1024	4096	896×2	768
Arithmetic (GIPS)	354	354	354	354	354	311	298×2	173

Input: Kernel execution time K specified by our system
Input: Maximum batch size L_{max}
Input: A query sequence
Input: Database that includes subject sequences
Output: High-score cells

```
Load a query sequence
n ← length of loaded query sequence
Transfer the query sequence to video memory
Set coefficients X and Y according to n and hardware
while not eof(database) do
    m̂ ← 0
    L ← 0
    while L ≤ L_max and K > Xnm̂ + Y⌈n/4⌉L do
        Load a subject sequence
        m ← length of loaded subject sequence
        m̂ ← m̂ + m
        L ← L + 1
        if eof(database) then
            break
        end if
    end while
    Transfer L subject sequences to video memory
    Invoke the matrix filling kernel to compute matrices
    Transfer results to main memory
end while
```

Fig. 12. Pseudocode of the Smith-Waterman alignment algorithm for an FGCS system. Our code dynamically determines the number L of subject sequences such that the kernel completes execution within specified time K. This code is not pipelined to simplify its description.

8 Further Experiments

For our experiments, we used query sequences of length n ranging from 63 to 511 amino acids. All queries were executed against the SWISS-PROT protein database [27], which is approximately 121 MB in size. This database contains 250,143 entries with a total of 90,588,910 amino acids. The database was transferred to worker machines before our experiments began.

Further, to have better load balancing between TBs, subject sequences were sorted by length m [3], [16]. Note that this sorting procedure does not always balance workloads, because TBs are not guaranteed to have the same length m after sorting. The effect of load balancing depends on the length distribution of subject sequences within a batch.

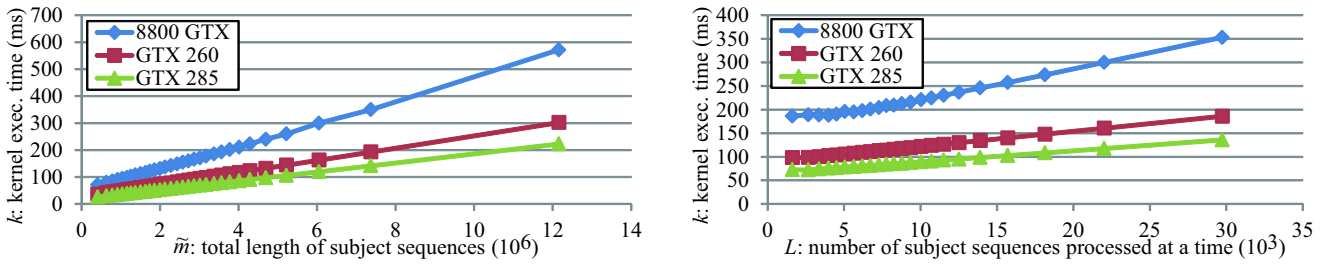
Table 1 summarizes the specifications of the eight worker machines used in case study. These worker machines are owned by graduate students who develop GPU and CPU applications for their research. The master process runs on a Windows XP machine with an Intel Core 2 Duo E6300 CPU clocked at 1.86 GHz. Experimental machines are interconnected by a Gigabit Ethernet switch.

8.1 Coefficient Estimation for Performance Model

Our system estimates coefficients X and Y using linear regression. To do so, we run the original kernel with different configurations and measure kernel execution time. Figures 13(a) and 13(b) show the kernel execution time with different total lengths $\widehat{m}$ of subject sequences and that with different batch sizes L of subject sequences, respectively. Results indicate that kernel execution time k is proportional to both $\widehat{m}$ and L. Since Eq. (4) describes this linear behavior, we can estimate coefficients X and Y from the gradients of the lines in both figures.

Table 2 shows the values of coefficients X and Y measured on GTX 285 and 8800 GTX cards. Results indicate that the coefficients depend on the length n of the query sequence. As we increase n, TBs consume more register files and shared memory, reducing the number Z of active TBs [12] that can simultaneously run on each multiprocessor. The number Z determines the performance behavior of the kernel.

As shown in the table, from the point of view of query length and the number of active TBs, the coefficients can be classified into distinct groups. For example, the GTX 285 card has six groups: $\{63\}$, $\{127\}$, $\{255\}$, $\{383, 511\}$, $\{640, 768\}$, and $\{896, 1022\}$. Note that we separate $n = 63$ from $n = 127$ though both have the same value $Z = 7$. This exception is due to the TB size, which is less than 32 when $n \leq 124$. In this case, the GPU reduces the efficiency, because the TB size is lower than the warp size [12]. Thus, we decided to use the mean values of X and Y for each group. For example, we



(a) (b)

Fig. 13. Kernel execution time k measured with length $n = 255$ of query sequence. (a) Results using fixed batch size $L = 8,000$ with varying total length $\widehat{m}$ of subject sequences. (b) Results using fixed total length $\widehat{m} = 4,000,000$ with varying batch size L. The gradients of each line in (a) and (b) represent coefficients X and Y, respectively.

TABLE 2
Estimated coefficients used for performance model. Coefficients basically depend on length n of query sequence, which determines number Z of active TBs per multiprocessor. Some results on 8800 GTX card cannot be obtained due to lack of resources.

Query length n	GTX 285 X (10^{-5})	GTX 285 Y (10^{-4})	GTX 285 Active TBs: Z	8800 GTX X (10^{-5})	8800 GTX Y (10^{-4})	8800 GTX Active TBs: Z
63	1.114	0.267	7	4.106	1.015	3
127	0.728	0.360	7	2.220	1.089	3
255	0.648	0.374	6	1.580	1.019	3
383	0.653	0.540	3	2.361	3.010	1
511	0.628	0.541	3	2.025	2.930	1
640	0.620	0.760	2	1.867	2.853	1
768	0.607	0.769	2	1.766	2.846	1
896	0.721	1.465	1	—	—	—
1022	0.665	1.518	1	—	—	—

used $X = 0.693 \times 10^{-5}$ and $Y = 1.491 \times 10^{-4}$ for the GTX 285 card when $n \geq 896$.

8.2 Evaluation Results on 8800 GTX

Figure 14 shows the distribution of kernel execution time with different lengths n of the query sequence. As compared with the results on the GTX 285 card, the accuracy drops for the 8800 GTX card. The reason for this relatively low accuracy can be explained by Fig. 14, where kernel execution times reach 128 milliseconds on the 8800 GTX. This implies that the 8800 GTX reduces the kernel efficiency if tasks are divided into small subtasks.

Figure 15 shows the frame rate measured on the 8800 GTX card. Similar to the results 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 10.8 fps, whereas the original kernel drops the rate to 0.6 fps.

8.3 Task Execution Statistics

Figure 16 shows how the detected idle time is exploited for alignment tasks. In total, the system assigns 80,347 tasks to workers and successfully completes 63,676 tasks during the

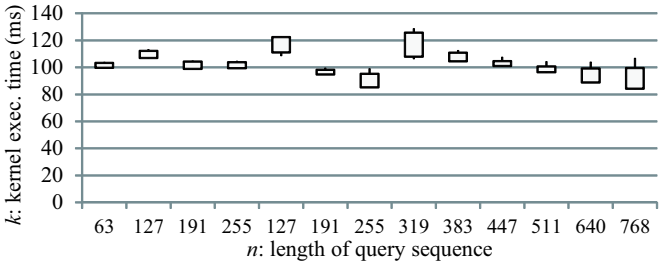


Fig. 14. Distribution of kernel execution times versus query sequence lengths n on 8800 GTX. The line segments display the range between the minimum and maximum times and vertical bars represent the 95% confidence intervals. Results are measured with specified time K of 100 milliseconds.

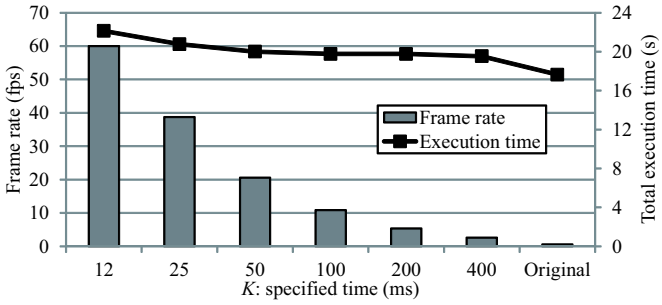


Fig. 15. Frame rates and total execution times measured using length $n = 511$ of query sequence on 8800 GTX. When $K = 12$, an overhead of 25% is observed on 8800 GTX due to excessive calls of kernels.

four days. Thus, we obtain a success rate of 79.7%. Worker #1 processes the largest number (15,165) of tasks. As shown in Fig. 5(b), this worker has the highest system uptime and the highest idle time, so that it indeed achieves the highest throughput among the eight workers. In contrast, worker #8 results in the smallest number (2,799) of tasks and the lowest success rate (46.5%). This worker has idle time totaling 12 hours, which is the mean value of the eight workers. Despite this relatively higher idle time, the success rate suffers because worker #8 has the previous generation card. More than 3,000 tasks are cancelled on this worker because the 8800 GTX card

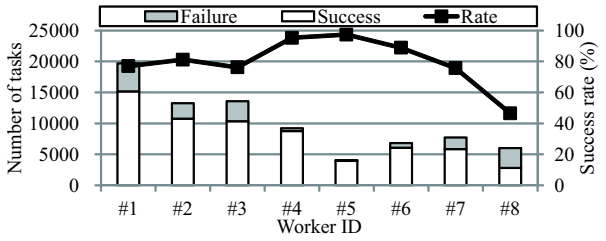


Fig. 16. Task statistics. Each bar represents the number of tasks processed over four days, whereas each line represents the corresponding success rate of task execution.

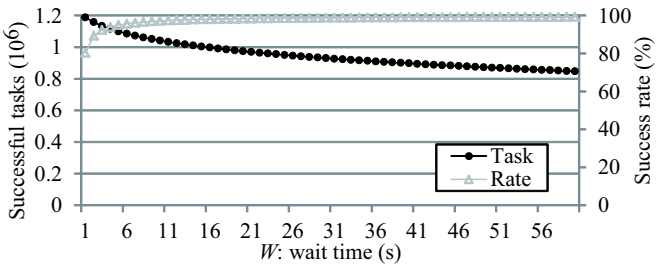


Fig. 17. Number of successful tasks and success rates with different lengths (W) of wait time ($n = 255$).

takes three times longer than the GTX 285 card. As mentioned in Section 4.1, a query of $n = 511$ takes 19.6 seconds to process a series of kernel invocations although each invocation completes around 100 milliseconds. Worker #5 achieves the highest success rate of 96%. Figure 5(a) implies that the owner of this worker intensively used the GPU for his or her research. Further, the wait time on this machine is only 12 minutes in total. Thus, the owner rarely operates the machine in an interactive manner, which strongly contributes to an efficient use of idle periods.

8.4 Scheduling Tradeoff

The proper granularity can be determined experimentally using two strategies: (1) scalability analysis based on communication and computation time, and (2) statistical analysis based on idle period distribution.

The scalability analysis roughly estimates the minimum granularity that fully utilizes idle workers. Let p represent the number of idle workers, and T_1 and T_2 represent the execution time and the communication time of a task, respectively. Assuming non-overlapping messages, the master must then satisfy $T_1 \geq T_2(p-1)$ in order to minimize the amount of time it takes the workers to receive a task. In our environment, where $0 \leq p \leq 14$, a task must run at least $0.17 \times 13 = 2.21$ seconds to keep workers busy during master-worker execution.

The statistical analysis gives us the expected length T_{exp} of idle periods. Due to the power law distribution of idle periods, the expected length can theoretically extend to infinity. However, this does not happen in practice, because idle periods of office machines are limited to a specific time. Therefore, the task granularity should be determined such that it satisfies $T_1 \leq T_{exp}$. In our logs, the expected length T_{exp} ranges from 4.7 to 28.8 seconds with a mean of 11.4 and a standard deviation of 5.7. We should not maximize T_1 because the penalty of failed executions increases with T_1.

Similar to the task granularity, the wait time W can determine guest throughput. We estimated the throughput with W varying from 1 to 60 seconds, as shown in Fig. 17. The throughput decreases as we increase W. Thus, it is better to minimize W to maximize the throughput. In contrast, the success rate of task execution decreases as we increase W. The idle period distribution in Fig. 3 also implies that increasing W decreases the number of idle period detections, which prevents frequent changes of resource status and reduces the number of messages exchanged between the master and its workers.

Finally, we investigated the variation of idle period distributions. Figure 18 shows some typical variations obtained from each machine. Figure 19 presents the conditional probability of remaining idle in the next five seconds, given an idle period of X seconds. This figure indicates that selecting long-idle-period workers is not always the best solution. For example, some workers that are idle for $X = 5$ seconds have a higher probability than others that are idle for $X = 7$ seconds. A straightforward solution to this problem is to select high-probability workers. However, this solution requires workers to construct a histogram of idle period lengths. In contrast, our approach requires workers to report just the length of the current idle period. Another advantage of our approach is that it gives a lower priority to interactively operated workers, which usually have shorter idle periods.

Note that the interference occurs only at the end of idle periods. Assuming that the master always has search jobs, the degree of interference is represented by the total number of idle periods. Thus, high-usage machines have sparse distributions with lower interference, as shown in Fig. 18(b).

8.5 Flexibility for Other Applications

As we mentioned in Section 4.6, three requirements must be satisfied in order to adapt an application. The first requirement characterizes applications that can run efficiently on network-accessible machines. For example, parameter sweep applications are an important class of killer applications for grid and volunteer computing systems.

The second requirement indicates the limitations of dynamic applications that vary the kernel execution time at run-time. For example, iterative applications may cause large interference to hosts if the GPU determines the number of iterations according to computational results. If such decision is done by the CPU, our approach has the opportunity to predict the kernel execution time. The performance model can be replaced by a dynamic adjustment mechanism that iteratively processes a series of tiny subtasks until reaching $K = 100$ milliseconds. However, we do not select this solution because such tiny subtasks require more kernel calls than our approach, causing a large overhead of 25% as shown in Fig. 15.

Our solution to the last requirement is to reduce the number of TBs per kernel invocation. This reduction is easily done for arbitrary kernels, because TBs in CUDA have neither constraints on their execution order nor capabilities to perform global synchronization. However, some types of kernels

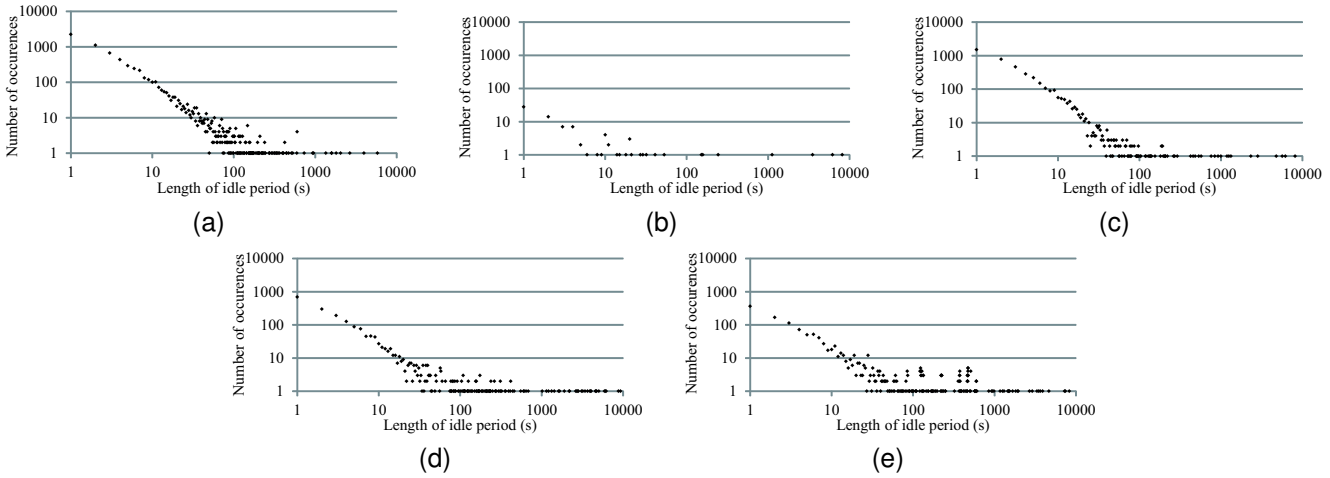


Fig. 18. Overall distribution in Fig. 3 can be classified into five groups: (a)/(b) a dense/sparse distribution due to frequent/rare idle periods or long/short system uptime, (c)/(d) a steep/gentle sloped distribution due to a small/large mean of idle period lengths, and (e) an irregular distribution due to lower machine utilization.

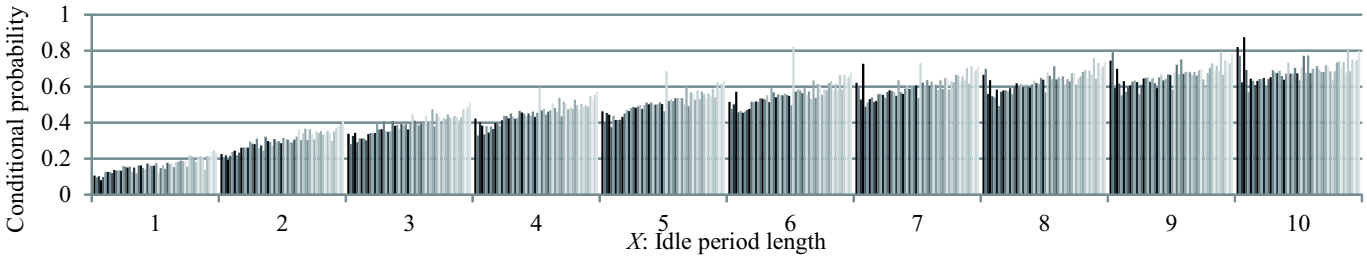


Fig. 19. Conditional probabilities of remaining idle in the next five seconds, given an idle period of X seconds. There are 56 vertical bars for each length X and bars with the same color corresponds to a worker running for five work days. Workers are sorted by the mean length of idle periods. A worker with a lower mean length tends to have a lower conditional probability.

create a large overhead due to task division, failing to reduce the kernel execution time. For example, sorting kernels and reduction kernels have to write intermediate results to global memory in order to restart their computation with the next subtask. In contrast, some applications do not have to write intermediate results between kernel calls. Parameter sweep applications can be such applications, because they contain many independent tasks which do not need to exchange computational results. Actually, our kernel restarts computation with different pairwise problems, so that intermediate results are not produced. Furthermore, such independent tasks can be executed in parallel by using the stream programming model on the GPU.

Finally, we discuss on the applicability of our approach to different application domains. Optimization is a promising class of parameter sweep applications that can be solved by our approach. For example, an optical propagation simulator [28] can be accelerated to design the mask layout for integrated circuits. To find the best layout, this simulator processes a large number of independent tasks, each with a different layout. Each task applies a series of convolution operations to a pair of 64×64-pixel images, and these operations are implemented by multiply-add and reduction kernels. The former simply exploits data parallelism by partitioning the computational domain. Therefore, the kernel execution time can be controlled by the number of TBs. In contrast, the latter can create a large overhead as mentioned above. However, this reduction kernel completes within 5 milliseconds due to the small data size. In this case, we are allowed to run the reduction kernel without task division. Thus, this simulator can be implemented using the master-worker paradigm, and each kernel can be completed within a short timeframe.

Similar to homology search, search-based applications may satisfy the three requirements mentioned above. As such an example, a spectrometer system [29] can be accelerated by our system. This system is developed for the search for extraterrestrial intelligence (SETI) project [30], which has demonstrated the impact of volunteer computing. The system applies the Fast Fourier Transform (FFT) operation to stream data. The data is organized into a series of chunks and there is no data dependence between different chunks. Furthermore, the chunk size can be arbitrarily changed to control the kernel execution time.

REFERENCES

[1] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons from eight years of volunteer distributed computing," in *Proc. 26th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'09)*, Apr. 2009, 8 pages (CD-ROM).

[2] 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.
[3] 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.
[4] NVIDIA Corporation, "NVIDIA GeForce 8800 GPU architecture overview," NVIDIA Corporation, Tech. Rep. TB-02787-001_v01, Nov. 2006. [Online]. Available: http://www.nvidia.com/object/IO_37100.html
[5] 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.
[6] 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.
[7] J.-H. Huang, "Opening keynote in GPU Technology Conference 2010," Sep. 2010. [Online]. Available: http://livesmooth.istreamplanet.com/nvidia100921/
[8] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig, "GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment," in *Proc. 13th Int'l Conf. High Performance Computing (HiPC'06)*, Dec. 2006, pp. 363–374.
[9] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide*, 5th ed. Reading, MA: Addison-Wesley, Aug. 2005.
[10] 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.
[11] 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.
[12] NVIDIA Corporation, "CUDA Programming Guide Version 2.3," Jul. 2009. [Online]. Available: http://developer.nvidia.com/cuda/
[13] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, Jan. 2007.
[14] A. Klimovitski, "Using SSE and SSE2: Misconceptions and reality," in *Intel Developer Update Magazine*, Mar. 2001.
[15] 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.
[16] 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.
[17] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases," in *Proc. 8th Int'l Workshop High Performance Computational Biology (HiCOMB'09)*, May 2009, 8 pages (CD-ROM).
[18] G. M. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," in *Proc. 23rd IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'06)*, May 2009, 10 pages (CD-ROM).
[19] I. T. Li, W. Shum, and K. Truong, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC Bioinformatics*, vol. 8, no. 185, Jun. 2007, 7 pages.
[20] O. Storaasli, W. Yu, D. Strenski, and J. Maltby, "Performance evaluation of FPGA-based biological applications," in *Proc. Cray User Group (CUG'07)*, May 2007, 14 pages.
[21] X. Meng and V. Chaudhary, "A high-performance heterogeneous computing platform for biological sequence analysis," *IEEE Trans. Parallel and Distributed Systems*, vol. 21, no. 9, pp. 1267–1280, Sep. 2010.
[22] Microsoft Corporation, "I/O completion ports (windows)," 2010. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx
[23] ejor, "Get CPU usage with GetSystemTimes," Dec. 2004. [Online]. Available: http://www.codeproject.com/KB/threads/Get_CPU_Usage.aspx
[24] Microsoft Corporation, "Windows API," 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx
[25] ——, "DirectX," 2007. [Online]. Available: http://www.microsoft.com/directx/
[26] R. Raman, M. Livny, and M. Solomon, "Matchmaking: An extensible framework for distributed resource management," *Cluster Computing*, vol. 2, no. 2, pp. 129–138, 1999.
[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] M. Motokubota, F. Ino, and K. Hagihara, "Accelerating parameter sweep applications using CUDA," in *Proc. 19th Euromicro Int'l Conf. Parallel, Distributed and Network-Based Computing (PDP'11)*, Feb. 2011, pp. 111–118.
[29] H. Kondo, E. Heien, M. Okita, D. Werthimer, and K. Hagihara, "A multi-GPU spectrometer system for real-time wide bandwidth radio signal analysis," in *Proc. 2nd Int'l Symp. Parallel and Distributed Processing with Applications (ISPA'10)*, Sep. 2010, pp. 594–504.
[30] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, Nov. 2002.