PAPER *Special Section on Parallel and Distributed Computing and Networking*

GPU-Chariot: A Programming Framework for Stream Applications Running on Multi-GPU Systems

Fumihiko INO†a), ***Member***, **Shinta NAKAGAWA**††, ***Nonmember***, ***and*** **Kenichi HAGIHARA**†, ***Member***

SUMMARY This paper presents a stream programming framework, named GPU-chariot, for accelerating stream applications running on graphics processing units (GPUs). The main contribution of our framework is that it realizes efficient software pipelines on multi-GPU systems by enabling out-of-order execution of CPU functions, kernels, and data transfers. To achieve this out-of-order execution, we apply a runtime scheduler that not only maximizes the utilization of system resources but also encapsulates the number of GPUs available in the system. In addition, we implement a load-balancing capability to flow data efficiently through multiple GPUs. Furthermore, a callback interface enables overlapping execution of functions in third-party libraries. By using kernels with different performance bottlenecks, we show that our out-of-order execution is up to 20% faster than in-order execution. Finally, we conduct several case studies on a 4-GPU system and demonstrate the advantages of GPU-chariot over a manually pipelined code. We conclude that GPU-chariot can be useful when developing stream applications with software pipelines on multiple GPUs and CPUs.

key words: *stream processing, GPGPU, CUDA, task scheduling*

1. Introduction

Stream processing is a programming paradigm that exploits data parallelism for accelerating compute-intensive applications [1]. Within this paradigm, input and output (I/O) data are organized into *data streams*, which are sequences of similar data elements. The input data stream flows through a series of processing stages in a pipelined manner. This pipelined execution produces the output data stream by enabling multiple data elements to be processed simultaneously. Because of its simplicity, stream processing is widely used in real-time media applications such as graphics rendering [2] and signal processing [3].

Numerous architectures have been developed to accelerate stream applications. An emerging architecture in this field is the GPU [4]. Although this hardware was originally designed for graphics applications, the emergence of the compute unified device architecture (CUDA) technology [5] made it possible to parallelize general applications using a C-like language. Using this programming framework, application hotspots can be implemented as *kernel functions* and parallely executed on hundreds of GPU cores to offload CPUs. Various scientific applications have been successfully accelerated, achieving a typical speedup of ten times over CPU implementations [6].

Several research results that can assist programmers in implementing stream applications running on a CUDA-compatible GPU are available. Udupa et al. [7] developed a compiling framework that maps StreamIt programs [8] to the GPU. Because StreamIt is a high-level language for stream applications, it relieves programmers from having to write and optimize their kernels to fully utilize the fast memory resources on the GPU chip. Similar frameworks [9]–[11] that can achieve significant acceleration over CPU-based implementations are available; however, the kernels generated by them are for single-GPU systems. Thus, an automated framework that addresses multi-GPU systems and scales application performance according to the number of available GPUs is required. In this paper, the term multi-GPU system is used to denote a single-node system equipped with multiple GPUs. Because multi-node systems are conventional systems that been have addressed by numerous studies [12]–[14], they are beyond the scope of this paper.

Huynh et al. [15] recently presented an automated framework that maps StreamIt programs to a multi-GPU system. Their framework employs multiple CUDA streams [5] for constructing software pipelines that overlap kernel execution with data transfer. Although the best speedup they obtained on a 4-GPU system was 2.97, the overlap efficiency can be further improved by taking advantage of data parallelism inherent in stream applications.

The key concept used to improve the overlap efficiency is the out-of-order execution of CUDA functions. However, this technique is currently not supported by CUDA streams. Moreover, to prevent pipeline stalls on all GPU architectures, except recently released GK110 [24], programmers are required to call CUDA functions in an appropriate order. Identifying an ordering that prevents pipeline stalls is time-consuming because it depends on runtime situations such as timing behaviors, data contents, and number of GPUs. Furthermore, programmers must create multiple CPU threads to process data elements on CPU cores. The execution of these multi-threaded programs requires mutual exclusion in order to enable resource sharing without violating correctness. In addition, to achieve linear speedups over single-GPU implementations, a load-balancing mechanism is needed to assign tasks to GPUs efficiently. Thus, programmers are burdened with nonessential but important work, such as task schedul-

Manuscript received December 18, 2012.
Manuscript revised May 7, 2013.

†The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
††The author is with the Storage Division, NEC Corporation, Fuchu-shi, 183–8501 Japan.
a) E-mail: ino@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.E96.D.2604

ing, resource allocation, and CPU thread management.

In this paper, we present a programming framework, termed GPU-chariot, for accelerating stream applications running on multi-GPU systems. GPU-chariot extends our previous study [9] in order to reduce development efforts for multi-GPU systems by automating the abovementioned time-consuming tasks. Our framework allows for out-of-order execution of CUDA functions, and realizes efficient software pipelines in four stages: (1) CPU execution, (2) *data download* from the CPU to the GPU, (3) GPU execution, and (4) *data readback* from the GPU to the CPU. Because this capability works automatically at runtime, our framework relieves programmers from determining an appropriate order for CUDA function calls. Using GPU-chariot, programmers are only required to write simple code statements that serially apply a sequence of operations to data stream elements.

The rest of the paper is organized as follows. Section 2 introduces related work in the area of stream processing on GPUs. Section 3 presents preliminaries, and identifies complications that occur when implementing stream applications with CUDA. In Sect. 4, we illustrate our GPU-chariot interface using a code example. A detailed description of the design and implementation of GPU-chariot is presented in Sect. 5. Section 6 presents the experimental results obtained from practical applications. Finally, in Sect. 7, we present conclusions and future work.

2. Related Work

NVIDIA has recently released their latest architecture called GK110 [24]. Unlike previous architectures, GK110 is equipped with the Hyper-Q technology, which provides 32 hardware work queues. These queues allows 32 CUDA streams to be processed concurrently on a device. Owing to multiple hardware queues, implicit synchronization is avoided between 32 CUDA streams (see Sect. 3.2). Consequently, overlapping execution can be easily realized if multiple CUDA streams are used for each device. The advantage of GPU-chariot over the Hyper-Q technology is a flexible, software-based scheduling framework. On the GK110 architecture, tasks are always processed in the first-come, first-served (FCFS) rule. In contrast, our task-scheduling scheme is capable of changing the order of task execution at runtime. Another advantage is an inter-device mechanism for avoiding excessive execution (see Sect. 5.3). This mechanism is useful to implement a sophisticated scheduling algorithm to prevent parallel queues from accessing intensively shared resources, such as the bus.

Another concern addressed by GPU-chariot is the gap of capabilities between different architectures. The latest GPUs can be classified into three groups, according to the number of direct memory access (DMA) engines and that of hardware work queues: (1) single DMA engine, single queue, (2) multiple DMA engines, single queue, and (3) multiple DMA engines, multiple queues. All architectures except GK110 have a single work queue. Consequently, implicit synchronization can prevent overlapping execution. In contrast, GK110 cards are free from such concerns, as we mentioned above. GeForce and Tesla cards have a different number of DMA engines, thus yielding different bandwidths for bidirectional data transfers. We believe that our runtime optimization can prove useful in hiding such architectural differences from the application code.

Various multi-GPU systems have been used to scale the performance of stream applications in a wide range of fields such as signal processing [16], volume visualization [17], and data imaging [18]. Although these studies demonstrated the effectiveness of stream processing on multi-GPU systems, they are specific to the application they implemented. In contrast, we concentrate on freeing application developers from the tedious chore of managing CUDA streams.

Huynh et al. [15] presented a unique framework that maps StreamIt programs [8] to multiple GPUs. Their main focus was to establish a program partition that efficiently utilizes memory resources on the GPU. Conversely, our study focuses on realizing an efficient overlapping mechanism, and clarifying how this mechanism can be implemented to increase application performance.

Several runtimes aiming to exploit the full resources in multi-GPU systems have been presented. Chen et al. [19] proposed a task scheduler capable of achieving dynamic load balancing at a finer granularity than what is supported in CUDA. They invoked a persistent kernel to process fine-grained tasks in the host's queues associated with the GPU. Diamos and Yalamanchili [20] proposed a dynamic parallelization technique, called kernel level speculation, which uses control speculation to expose parallelism. Although these studies successfully increase the efficiency of kernel execution, they have not attempted to increase the efficiency of pipelined execution.

There are research projects [10], [11] that assist programmers in developing stream applications running on single-GPU systems. These projects provide compiling frameworks that generate CUDA codes for StreamIt programs. The key concept for generating efficient kernels is to classify threads into two groups: dedicated threads for accessing data and threads for processing data. It might be interesting to integrate our runtime optimization technique with the static optimization techniques mentioned above.

Finally, scheduling algorithms have been studied from a theoretical viewpoint. Our target problem is similar to a hybrid flow shop (HFS) problem [21], which is a common class of scheduling problems for manufacturing environments. The HFS problem assumes that a set of N tasks is processed in a series of K stages. However, our target problem differs from the HFS problem because it prohibits resources in different pipelines to pass data between them. In addition, our algorithm addresses an online scheduling problem where not all tasks are available from the beginning. Since obtaining the best schedule with the minimum make-span is a nondeterministic polynomial (NP) time hard problem, many heuristics were proposed in the past [22], [23]. Our algorithm can be regarded as a heuristic that uses a dispatching rule.

3. Preliminaries

3.1 Stream Programming Model

Let $\mathcal{I}$ and $\mathcal{O}$ be the input and output data streams, respectively. Suppose that a stream application has a set $\mathcal{F} = \{f_1, f_2, \dots, f_K\}$ of K pipeline stages, where f_k $(1 \leq k \leq K)$ represents the k-th pipeline stage. Then, the output data stream is given by $\mathcal{O} = f_K \circ f_{K-1} \circ \cdots \circ f_1(\mathcal{I})$, where $\circ$ represents the operator for function composition. Within the context of GPU computing, set $\mathcal{F}$ typically consists of CPU execution, data download, GPU execution, and data readback stages $(K = 4)$.

The stream programming model assumes that the computations of different data elements are not interdependent. Therefore, the input data stream $\mathcal{I}$ can be arbitrarily broken into chunks $e_1, e_2, \dots, e_N$ and processed in a pipelined manner; N represents the number of chunks and e_i $(1 \leq i \leq N)$ represents the i-th chunk. Moreover, because of the data independence property, the chunks can be processed in an out-of-order fashion. A *task* is defined as the processing of a chunk through a set $\mathcal{F}$ of pipeline stages. Furthermore, we assume that the data size $|e_i|$ of chunk e_i determines the granularity of a task.

Stream applications can be implemented using CUDA streams and asynchronous CUDA functions. A CUDA stream is a sequence of *commands* that are executed sequentially [5]. These commands are used for data download, kernel execution, and data readback. Because different CUDA streams may execute their commands concurrently, data transfers from one CUDA stream can overlap with the kernel execution from another CUDA stream. A code example illustrating this process can be found in Appendix A.

```
Scheduler sch(D, S); // an instance of our scheduler
float *h, *d_in, *d_out;
int x, y;  // x: data width, y: data height
int chunk_size = 512 * 8192;
int N = (x * y) / chunk_size;
dim3 gridDim, blockDim;

cudaMallocHost(&h, x * y);
sch.malloc(&d_in, chunk_size); // calls cudaMalloc()
sch.malloc(&d_out, chunk_size);

// column FFT phase
int width = x / N; // chunk width
for (int i=0; i<N; i++) { // for each task
    sch.download2D(i, width, d_in, width, &h[i * width
         ], x, width, y);
    sch.launchKernel(i, width, MyfftCol, gridDim,
         blockDim, 0, d_in, d_out, width, y);
    sch.readback2D(i, width, &h[i * width], x, d_out,
         width, width, y);
}
sch.synchronize();

// row FFT phase
int height = y / N; // chunk height
fftParam_t param(d_in, d_out, x, height); //
     parameters for cufftExecC2C
for (int i=0; i<N; i++) { // for each task
    sch.launchCallbackRoutine(i, height,
         MycufftPlan1dSetStream, &param, flagCPU);
    sch.download2D(i, height, d_in, x, &h[i * height *
          x], x, x, height);
    sch.launchCallbackRoutine(i, height, cufftExecC2C,
          &param, flagGPU);
    sch.readback2D(i, height, &h[i * height * x], x,
         d_out, x, x, height);
}
sch.synchronize();

sch.free(d_in); // calls cudaFree()
sch.free(d_out);
cudaMallocFree(h);
```

Fig. 1 Simplified example of host code for a SETI spectrometer system [16] using GPU-chariot. `N` tasks are processed on `D` GPUs, and each GPU uses `S` CUDA streams to achieve overlap. `MyfftCol` and `cufftExecC2C` are a user-defined kernel and a vendor-provided function, respectively.

3.2 Problem Description

When using CUDA streams, implicit synchronization [5] possesses several technical difficulties because it blocks commands from other CUDA streams. An example of blocking execution is presented in Fig. A· 1 of Appendix A. Because most GPU architectures, except recently released GK110 [24], have a single hardware work queue, commands are serially processed through this queue. Therefore, blocking behavior is caused by the in-order execution of data transfer commands. According to our preliminary experiments, the constraints of pre-GK110 architectures can be summarized as follows:

C1. Data transfer commands from CUDA streams are processed in an in-order sequence.
C2. Kernels are executed in an in-order sequence.

Resource allocation is another important issue for a multi-threaded host code. Typically, in multi-GPU systems, the GPUs share an I/O bus, such as the peripheral component interconnect express (PCIe) bus. Another shared resource is the CPU core, which must be fully exploited in order to accelerate the CPU execution stage. Thus, to achieve high performance, our resource allocation scheme must avoid resource starvation and flow more chunks through pipelines.

In summary, we consider the problem of mapping N tasks onto D software pipelines that lay on D GPUs and C CPU cores. Each task contains K serial commands, and each command is considered a pipeline stage. We assume that GPUs are connected via B bidirectional buses, and each unidirectional link can be used simultaneously by a download or readback command. Hence, we assume that each chip contains at least two DMA engines. Conversely, because GeForce GPUs have a single DMA engine, a download or readback command occupies a bidirectional link. In order to produce efficient schedules, we also assume that the time spent on each pipeline stage is proportional to the size of a chunk. Formally, for all $1 \leq i, j \leq N$ and $1 \leq k \leq K$, $|e_i| \geq |e_j| \Rightarrow t_{k,i} \geq t_{k,j}$, where $t_{k,i}$ represents the time spent for e_i on the k-th stage.

4. GPU-Chariot Interface

The GPU-chariot interface is designed to reduce efforts of modifying nonpipelined single-GPU implementations into pipelined multi-GPU implementations. Figure 1 presents a simplified host code example that is part of a SETI spectrometer system [16]. In this example, we sequentially compute the fast Fourier transform (FFT) of the columns and rows of a two-dimensional (2-D) matrix in two phases: the first phase is implemented by the user-written `MyfftCol` kernel, while the second phase is implemented by the vendor-provided `cufftExecC2C` function [25]. Note that to transpose the matrix data, the consecutive phases must be synchronized.

As shown in Fig. 1, the first phase, in addition to encapsulating CUDA functions within GPU-chariot functions, exhibits a simple loop structure for processing a sequence of chunks. Therefore, to apply commands to each chunk being processed, programmers are simply required to replace CUDA function calls with GPU-chariot function calls. The first of the two arguments of the replaced functions is used to provide the additional information of task ID and task granularity, which in our case are variables `i` and `width`, respectively. The task ID is required to associate commands with tasks because commands belonging to the same task should be assigned to the same GPU to minimize data transfer. Likewise, the task granularity is used by the runtime scheduler to schedule tasks according to the size of the chunks. Because GPU-chariot functions are processed asynchronously, similar to CUDA functions, the `synchronize` function is required to ensure that all issued commands are finalized before proceeding to the second phase.

In addition, GPU-chariot provides a callback interface capability for integrating CPU- and CUDA-based functions of third-party libraries into pipelines. Owing to this interface, GPU-chariot can be applied to any application that uses third-party libraries, such as CUFFT [5]. As shown in the second phase in Fig. 1, the function `launchCallbackRoutine` registers the user-written function `MycufftPlan1dSetStream` and vendor-provided function `cufftExecC2C` as callback functions. In addition, the flags `flagCPU` and `flagGPU` are provided to GPU-chariot for identifying the resources used by the registered function. During runtime, this information is used to select an overlappable command from the issued commands.

Because variable `D` appears only in line 1 during the declaration of the scheduler instance, the dependence of our host code on the number of GPUs, D, is limited. Furthermore, in our host code, there is no need to bind tasks with CUDA streams. Such bindings are automated by GPU-chariot.

5. GPU-Chariot Runtime

Given a CUDA-like program shown in Fig. 1, our GPU-chariot runtime performs several key steps during the program execution:

CPU thread management. Our runtime creates and joins CPU threads to handle software pipelines in the system.

Task and command buffering. To enable out-of-order execution of commands, GPU-chariot stores tasks in a *task buffer*. Tasks are registered with their issued commands, namely instances of CUDA and callback function calls that have to be executed.

Resource allocation. The objective of our GPU-chariot runtime is to allocate resources to CUDA streams and increase the number of concurrent overlapping commands, while avoiding excessive execution.

Task assignment and command dispatching. Our GPU-chariot runtime uses the abovementioned task buffer to assign tasks to CUDA streams, and to dispatch the commands required for overlapping execution. In other words, the dispatched commands are executed in pipelines while satisfying constraints C1 and C2. Thus, the execution order, which may vary according to the progress of the program execution, is optimized at runtime.

In the following sections, we describe in detail each step of GPU-chariot runtime, and explain how it addresses the abovementioned issues.

5.1 CPU Thread Management

GPU-chariot runs three types of CPU threads to process the host code, scheduler, and callback functions, as shown in Fig. 2. The host code is processed by the main thread, which creates D scheduling threads for handling D GPUs. The scheduling threads create and join child threads, called executioners, which asynchronously process callback functions. The lifetime of an executioner is defined from the beginning to the end of the registered callback function.

As shown in line 1 of Fig. 1, the main thread initializes a scheduler instance with the number of GPUs, D, and that of CUDA streams, S. Next, it creates D scheduling threads, each with S CUDA streams. Hence, there are DS CUDA streams in total. Each scheduling thread is responsible for assigning tasks and dispatching commands to a GPU (i.e., S CUDA streams). We employ this one-CPU-thread-per-GPU scheme to parallelize scheduling procedures and achieve high performance.

To bind each thread to a specific CPU core, we create a processor affinity mask. Such bindings in multi-threaded

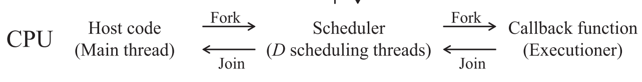


Fig. 2 CPU thread management in GPU-chariot. During runtime, scheduling threads are created to assign tasks to CUDA streams. Then, each scheduling thread creates CPU threads to process the execution stages registered by the callback interface.

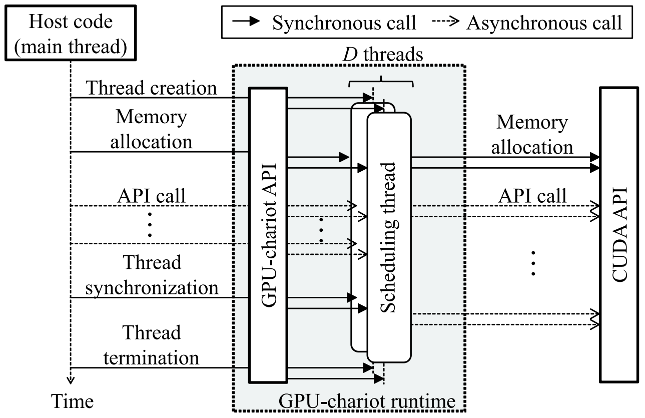


Fig. 3 Timing behavior of runtime of GPU-chariot. Our runtime receives GPU-chariot function calls from the host code, and then registers the corresponding CUDA function calls as command instances. Multi-threaded schedulers, which run concurrently with the main thread, dispatch the command instances.

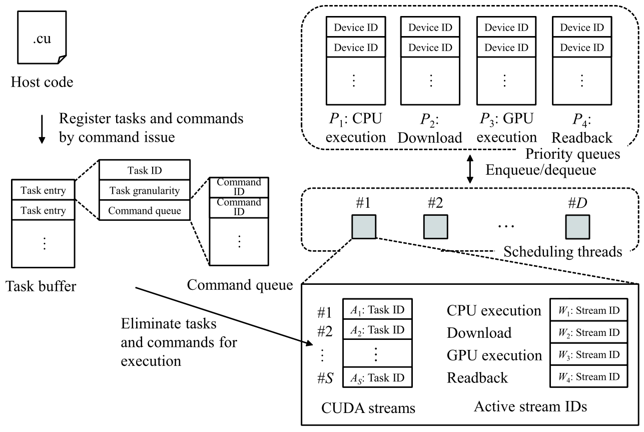


Fig. 4 Runtime architecture of GPU-chariot. Commands are associated with tasks to avoid changing the sequence of pipeline stages. First, scheduling threads request resource allocation by queuing their responsible device ID to the priority queue that manages the command they want to dispatch. The device in the head of the priority queue is allowed to dispatch its command to an idle CUDA stream.

applications are useful because they prevent cache misses due to thread migration.

5.2 Task and Command Buffering

Figure 3 illustrates the relationship between the host code and the application programming interface (API) of CUDA. In this figure, solid and dashed arrows represent synchronous and asynchronous calls, respectively. The host code asynchronously calls GPU-chariot functions in a sequential manner, as shown in the loop of the main body of the code in Fig. 1. At each call, a command is placed into a *command queue*, which is used later to execute corresponding CUDA function. Similarly, callback functions (i.e., CPU- and CUDA-based functions) are queued by the `launchCallbackRoutine` function.

The task buffer is organized such that it can associate tasks with commands. As shown in Fig. 4, an entry in the task buffer consists of the task ID, task granularity, and a command queue that stores a series of commands to be executed for the task. Because each function call of GPU-chariot places a command into a queue, the main thread first checks whether its corresponding task is already registered in the task buffer. If no corresponding task is found, a task entry is created with its ID, granularity, and command queue. As we mentioned in Sect. 4, this information is provided by the additional arguments of the replaced function. Otherwise, the main thread simply queues the command into the existing queue.

Programmers can specify at compile time which ordering rule should be used:

- The largest first (LF) rule. At each queuing operation, the buffered tasks are sorted in a descending order of task granularity.
- The FCFS rule. In this case, the order of the tasks remains unchanged.

The LF rule aims to increase the efficiency of overlapping execution by masking tasks of similar granularity. Overlaps occur between successive tasks. Therefore, to prevent large data transfers from being overlapped with the execution of small kernels, successive tasks should have similar granularity. Otherwise, small tasks prevent large tasks from being fully masked, thus causing pipeline stalls.

Note that the smallest first (SF) rule, which sorts tasks in an ascending order of task granularity, is not a good solution for the asynchronous API. The asynchronous API initiates task execution before receiving all tasks from the host code. Consequently, the smallest task cannot be always issued first. Conversely, the SF rule maximizes the last readback time, which cannot be masked with the kernel execution time. When times cannot be masked, the execution efficiency decreases.

Multiple threads can access the task buffer; the main thread registers tasks to the buffer while scheduling threads eliminate them for execution. Consequently, a lock/unlock mechanism is required to access the task buffer without violating correctness. We use critical sections to implement this mechanism, whose details are presented in Appendix B.

5.3 Resource Allocation

Because overloaded resources can significantly slowdown execution, commands must not be excessively processed at each pipeline stage. Thus, to realize efficient pipelines, our resource allocation scheme should (1) maximize the number of concurrent commands with overlap and (2) avoid excessive execution.

We can increase the number of concurrent commands by taking advantage of our parallel scheduler. As shown in Fig. 4, because the GPU-chariot runtime has a priority queue for each pipeline stage, commands at different stages can simultaneously allocate resources. In contrast, commands at the same stage are processed in an FCFS manner. Let P_k be the priority queue for the k-th stage, where $1 \leq k \leq K$. Entries of priority queues are IDs of devices that are available for resource allocation. The device in the head of queue P_k can allocate its resources for the k-th stage. After the command is executed, the head entry is dequeued to allow the next device to allocate its resources. Because each priority queue is shared among threads, we use critical sections to serialize the access to the queue. Owing to our queue organization, this serialization does not occur between commands at different stages, and thus performance degradation is minimized.

With respect to constraints (1) and (2), each command before allocating its resources must satisfy the following two conditions:

Condition for overlapping execution. The key to achieving overlap is to manage the information on running commands and prioritize queued commands that can be overlapped with running commands. As described in the following sections, counters are used to manage runtime information. In addition, GPU-chariot consists of a $K \times K$ matrix $\mathcal{M}$, which statically defines pairs of overlappable stages. For example, on GeForce cards, the download stage cannot overlap with the readback stage, because these cards have a single DMA engine. The matrix $\mathcal{M}$ is initialized during the declaration of the scheduler instance by inspecting the hardware through the CUDA API. Thus, the first condition for a queued command can be stated as follows: (a) there is no running command or (b) commands are running at other overlappable stages.

Condition for non-excessive execution. To avoid overloaded situations, the GPU-chariot runtime uses counters and thresholds for each pipeline stage. A counter maintains the number of commands currently running at a stage, while a threshold defines the maximum permitted number of commands running at the stage. Excessive execution then can be avoided (c) if the counter values are below the corresponding thresholds.

Because maximum thresholds are restricted both locally and globally, they are classified into two groups. Consequently, GPU-chariot utilizes two threshold vectors: local vector $\mathcal{L}$ and global vector $\mathcal{G}$. Local vector $\mathcal{L}$ is given by $\mathcal{L} = (L_1, L_2, L_3, L_4)$, where L_k $(1 \leq k \leq 4)$ represents the maximum number of concurrent commands a scheduling thread can dispatch at the k-th stage. Conversely, global vector $\mathcal{G} = (G_1, G_2, G_3, G_4)$ considers the total number for all D scheduling threads in the system. GPU-chariot currently uses $\mathcal{L} = (\min(C, S), 1, 1, 1)$ and $\mathcal{G} = (\min(C, DS), B, D, B)$. The first elements of these vectors indicate that the number of CPU execution commands is limited by the amount of physical resources (i.e., C CPU cores) and logical resources (i.e., S CUDA streams for each GPU). The third elements represent that a GPU can execute at most one kernel at a time. The remaining elements are determined on the basis of the assumption made about the I/O bus mentioned in Sect. 3.2.

5.4 Task Assignment and Command Dispatching

Scheduling threads are responsible for assigning tasks to CUDA streams, selecting commands for execution, and dispatching commands to CUDA streams. To construct efficient software pipelines, two key requirements must be satisfied. First, we should minimize the time required to dispatch commands to an idle stage by employing multiple CUDA streams. From this viewpoint, if all pipeline stages are kept busy, idle CUDA streams are not critical. Second, we should minimize the interactions between scheduling threads so that they can run as independent as possible.

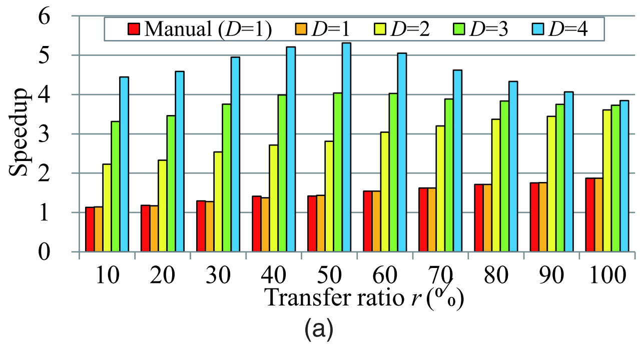
To satisfy these requirements, our scheduling threads circulate pipeline stages independently. In particular, for the k-th stage of its responsible GPU $(1 \leq k \leq 4)$, each thread processes the following four steps:

1. Resource allocation step. The scheduling thread requests resource allocation for the k-th stage, according to the two conditions mentioned in Sect. 5.3. If the responsible device ID of the thread is in the head entry of priority queue P_k, it allocates the resource. Otherwise, this step is skipped.
2. CUDA stream selection step. After resources are allocated, the scheduling thread searches for a CUDA stream j $(1 \leq j \leq S)$ that has an overlappable command f ready to be dispatched at the k-th stage. Such a CUDA stream is found in a cyclic manner to achieve load balancing between CUDA streams. In particular, GPU-chariot uses a vector $\mathcal{W} = (W_1, W_2, W_3, W_4)$, where W_k $(1 \leq k \leq 4)$ represents the ID of the active CUDA stream, that currently runs a command at the k-th stage (Fig. 4). If an idle CUDA stream exists, the scheduling thread pops a task from the task buffer and assigns it to the idle CUDA stream.
3. Command dispatching step. The function that corresponds to command f is called using CUDA stream j. In addition, the active stream ID W_k, local counter, and global counter of the k-th stage are updated accordingly.
4. Idle detection step. The scheduling thread checks the status of the k-th stage. If CUDA stream W_k is in the idle state (i.e., the command execution at the k-th stage is completed), the thread dequeues the head entry from the priority queue P_k and decrements the local and global counters of the k-th stage.

A more detailed description of the scheduling algorithm can be found in Appendix B.

6. Experimental Results

In this section, we present the evaluation results of GPU-chariot in terms of the overhead of the runtime scheduler, and discuss its applicability to practical applications [16], [26]. We compare GPU-chariot code with not only pure CUDA code but also hybrid code of CUDA and



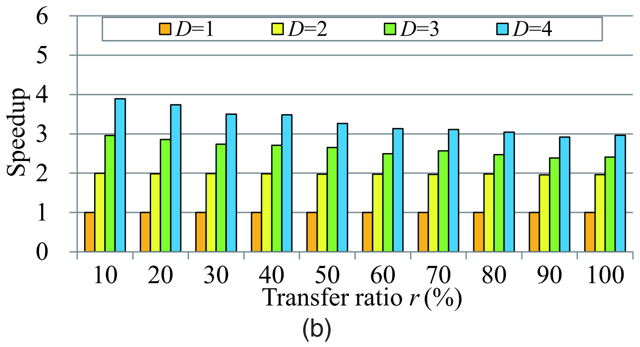


Fig. 5 Speedups of the dummy kernel over a nonpipelined version with fixed chunk sizes. (a) Results for GPU-chariot. (b) Results for OpenMP. GPU-chariot using a single GPU ($D = 1$) achieves almost the same speedup as a manually pipelined version.

OpenMP [27]. Similar to the GPU Computing SDK sample (cudaOpenMP) code [28], this hybrid code calls the `cudaSetDevice` function to utilize all available devices with a single CUDA stream per device.

Experiments were conducted on a Linux PC equipped with four NVIDIA GeForce GTX 580 cards, two Intel Xeon X5670 2.93 GHz CPUs, and an Intel 5520 chipset. The system had in total $C = 12$ physical cores and 48 GB main memory. Each graphics card had 1.5 GB device memory and a single DMA engine. The graphics cards were connected to a PCIe expansion box, namely ELSA Vridge X200 Tri. This box had two PCIe lanes, enabling two graphics cards to share a PCIe lane (i.e., $B = 2$ and $D = 4$). The experimental machine had a Western Digital WD3200AAKS 320 GB hard drive connected to a Serial Advanced Technology Attachment (SATA) 2.0 interface.

6.1 Overhead Analysis

We performed detailed analysis of the overhead using a dummy program that could arbitrarily change the kernel execution and data transfer times. The dummy kernel loaded integers from global memory, repeatedly applied shift operations on them, and stored results back to global memory. We iteratively invoked the kernel to process $N = 48$ chunks. Each chunk e_i $(1 \leq i \leq N)$ consisted of an array of integers, and the chunk size $|e_i|$ ranged from 3 to 30 MB.

Figure 5 shows speedups over a nonpipelined implementation, confirming that how GPU-chariot reduces the execution time by software pipelining on multiple GPUs. The transfer ratio r in Fig. 5 is given by $r = (T_D + T_R)/T_K$, where T_D, T_R, and T_K represent the data download time, data readback time, and kernel execution time, respectively. As we increase r, the performance bottleneck moves from kernel execution to data transfer. In general, the least upper bound on the speedup reaches a factor of two when $r = 100$ and $D = 1$. We controlled the transfer ratio r by changing the element size $|e_i|$ while fixing T_K. We used the FCFS rule because the chunk size was fixed during program execution.

As shown in Fig. 5 (a), when $r = 100$ and $D = 1$, GPU-chariot achieves a speedup of 1.9. This speedup is close to the least upper bound, and is also nearly equal to the one achieved by a manually pipelined version, which first issues all downloads, then invokes all kernels, and finally calls the readback functions. In this case, we cannot observe the runtime overhead of GPU-chariot.

The speedups in Fig. 5 (a) increase linearly with the number of GPUs, D, when $r \leq 50$. Because these speedups are larger than D, we demonstrate an efficient use of overlap. In contrast, when $r = 100$ and $D \geq 3$, a similar linear increase in speedups is not observed. In particular, when we increase r above 60%, speedups for $D \geq 3$ decrease. We can attribute this decreasing behavior to the reduced efficiency of overlap, i.e., shared I/O buses frequently increase the waiting time on GPUs, whereas this overhead does not occur if GPUs occupy their own I/O bus ($D \leq 2$). Considering this waiting time for $D = 4$ and $r = 100$, the execution time of the kernel is not long enough to fully mask the data transfer and waiting times. In contrast, the execution time of the kernel for $D = 4$ and $r = 50$ is long enough to cover them, thus yielding a superlinear speedup. Note here that decreasing speedups do not always mean performance degradation. In fact, by adding two GPUs, GPU-chariot reduces the execution time for $r = 100$ from 1,503 ms ($D = 2$) to 1,411 ms ($D = 4$).

As shown in Fig. 5 (b), the OpenMP version successfully utilizes all four GPUs when $r = 10$. However, speedups for $D \geq 3$ decrease as we increase r, because this version uses a single CUDA stream per device, which cannot overlap kernel execution with data transfer. Thus, OpenMP code with `cudaSetDevice` is simple but fails to achieve pipelined execution on pre-GK110 architectures. Due to similar reason, speedups for $D \leq 2$ are at most D for all r. A full overlap of kernel execution and data transfer requires (1) multiple CUDA streams per device, and for all GPU architectures except GK110, (2) out-of-order execution. Because OpenMP is designed for writing multithreaded CPU code, it cannot assist programmers in achieving these two points. Consequently, we think that OpenMP is useful to develop multi-GPU applications rather than stream applications. In contrast, GPU-chariot provides a simple programming interface that hides not only CUDA streams but also an appropriate order for CUDA function calls from the application code.

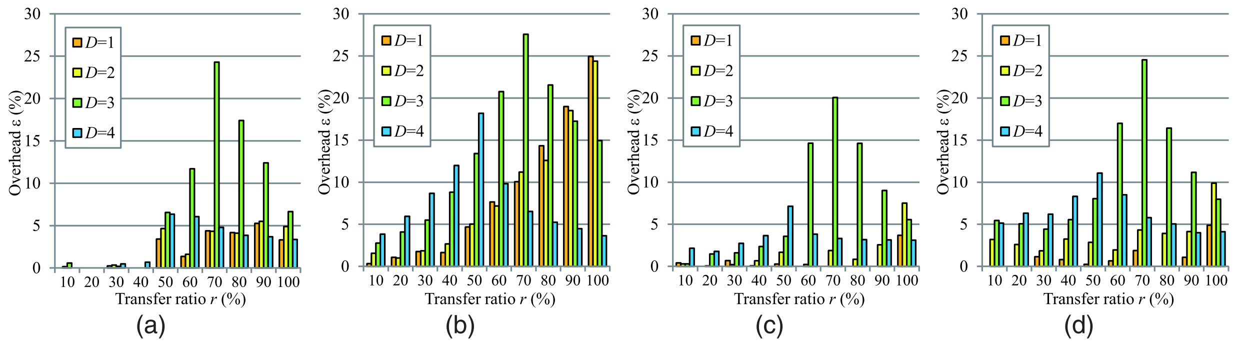


Fig. 6 Overheads over a lower bound. (a) Results for fixed chunk sizes when using the FCFS rule. Results for random chunk sizes when using (b) the FCFS rule, (c) the LF rule, and (d) the SF rule.

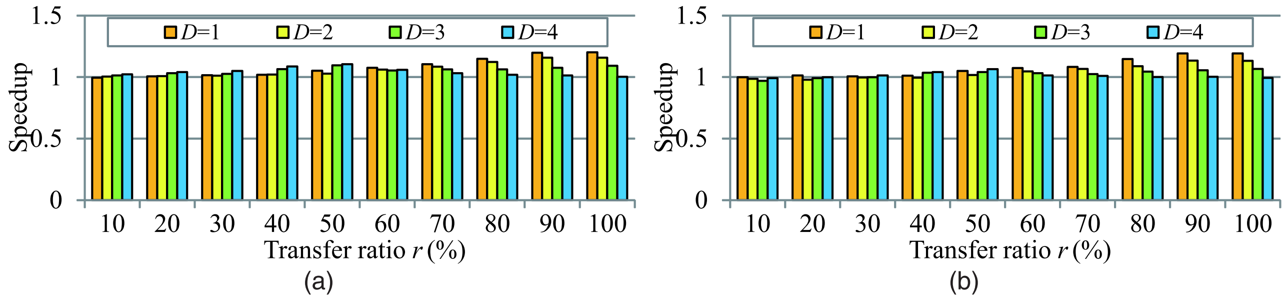


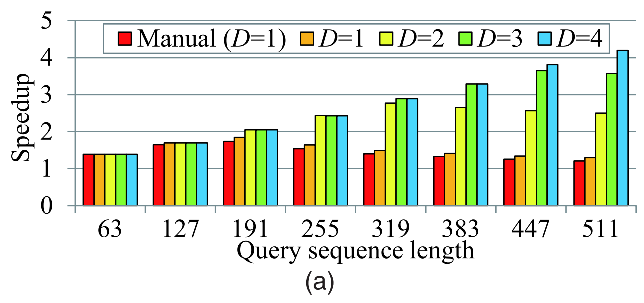
Fig. 7 Speedups over the FCFS rule with random chunk sizes. (a) Results for the LF rule. (b) Results for the SF rule.

Figure 6 shows the overheads of GPU-chariot, which illustrate the efficiency of the produced schedules. Here overhead ϵ is given by $\epsilon = T/T_{LB} - 1$, where T is the execution time and T_{LB} represents the lower bound (described in Appendix C). This lower bound assumes that if kernel execution dominates the performance, tasks are assigned equally to GPUs. Conversely, if data transfer dominates the performance, tasks are flown equally on buses. The overhead can be minimized to $\epsilon = 0$ if both the first download time and last readback time are minimized and the bottleneck stage is kept busy continuously (i.e., full overlap is achieved). As shown in Fig. 6 (a), in most cases, GPU-chariot achieves $\epsilon \leq 5$, and almost no overhead when $r \leq 40$.

However, overhead ϵ reaches 20% when $D = 3$. This increase occurs because the assumption mentioned above does not fully apply to the 3-GPU configuration. In particular, we connected two GPUs to a single bus, and the third GPU to the remaining bus. In this case, as mentioned above, the two GPUs connected to the same bus cannot use kernel execution to fully hide data transfer when $r \geq 60$. In contrast, the remaining GPU has a dedicated bus, and thus achieves full overlap by processing more tasks than the two GPUs. Consequently, the third GPU processes 18 tasks during execution and 16 tasks in lower bound analysis ($r = 70$). Thus, software pipelines in 3-GPU systems can have different performance bottlenecks depending on the transfer ratio r. By taking into account the number of actually assigned tasks, the overhead ϵ reduces from 24% to 11% when $r = 70$. Although pipelines have different bottlenecks when $r = 60$, as we increase r to 100%, they all have the same bottleneck of data transfer. This explains why overhead ϵ decreases as we increase r above 70%.

We next measured the overhead ϵ for random chunk sizes (Fig. 6 (b)–(d)). We modified the kernel such that it can change chunk sizes at runtime. We also executed the kernel using the LF and SF rules, in addition to the FCFS rule. Figure 6 (a) and (b) indicate that for random chunk sizes, the use of the FCFS rule causes an increase in the overhead. For nonuniform chunks, the use of the FCFS rule results in a drop in the overlap efficiency because small kernel execution is masked with large data transfers. Therefore, kernels cannot be executed continuously on the GPU, thus causing the bottleneck stage to be idle. In contrast, the LF and SF rules have smaller overheads than the FCFS rule, as shown in Fig. 6 (c) and (d). Consequently, both rules are faster than FCFS by at most 20% (Fig. 7). Thus, our sorting strategy contributes in increasing the efficiency of pipelined execution for random chunk sizes. These results also imply the importance of changing the order of task execution at runtime, which is not supported by the Hyper-Q technology.

Figure 6 (c) and (d) indicate that the LF rule is faster than the SF rule. The former rule allows the smallest task to be processed at the end of execution. Thus, the LF rule minimizes the unmaskable time (i.e., the last readback time) that appears at the end of execution. In contrast, as we mentioned in Sect. 5.2, the SF rule maximizes the last readback time. Moreover, the unmaskable time appears at the beginning of execution (i.e., the first download time). Minimizing the first download time requires a blocking mechanism



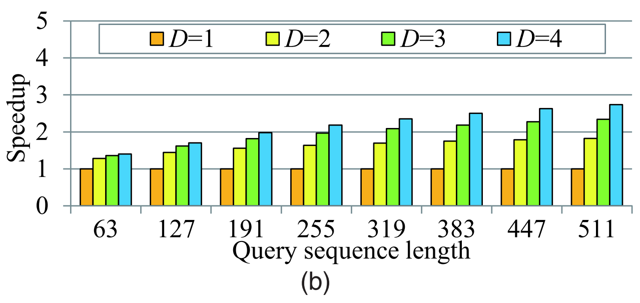


Fig. 8 Speedups of sequence alignment over a nonpipelined version. The LF rule was used for measurement. (a) Results for GPU-chariot. (b) Results for OpenMP.

to buffer all tasks before starting the scheduling procedure. Since this blocking approach results in delaying execution, we do not adopt it.

6.2 Case 1: Amino Acid Sequence Alignment

As a case study, we used a biological application that iteratively performs local alignment of amino acid sequences [26]. This application requires two inputs, namely a query sequence and a database of amino acids, and returns similarity scores between the query sequence and subject sequences in the database. In order to overlap disk access with kernel execution, the original code divides the database into chunks [26]. These chunks are processed in a pipeline of five stages: file read, data download, kernel execution, data readback, and file write. Moreover, we used our callback interface to integrate the file read and write stages into the pipeline.

The database [29] contains 536,029 subject sequences, while each chunk contains 8,192 subject sequences. Consequently, a query sequence involves $N = 66$ tasks that require alignment. The length of a subject sequence ranges from 2 to 35,213 amino acids, with an average of 755 amino acids per sequence. Thus, chunks have different total lengths. Because task granularity is nonuniform, we selected to apply the LF rule during measurements. For experiments, we used 8 query sequences of different lengths, ranging from 63 to 511 amino acids. As shown in Table 1, for short queries, the file read stage dominates the performance, whereas for long queries, the kernel execution stage dominates the performance.

Figure 8 shows speedups of our pipelined code over a nonpipelined version. For the shortest query of 63 amino acids, GPU-chariot achieves a speedup of 1.4 times (Fig. 8 (a)). This speedup is the same as that achieved by a manually pipelined version, which performs in-order execution. However, because for short queries, the file read and write stages are the performance bottleneck, the speedup does not increase with the number of GPUs, D, as shown in Table 1. Thus, multi-GPU solutions are not effective for situations with intensive I/O operations. A similar behavior is observed with the OpenMP version (Fig. 8 (b)).

In contrast, speedups for long queries increase with the number of GPUs, because the bottleneck stage is effectively accelerated with multiple GPUs. The speedup reaches a factor of 4.1 when using $D = 4$ GPUs for the longest query of 511 amino acids. Similarly, the speedups of the OpenMP version increase as we increase the length of query sequence, but the highest speedup is below a factor of 3. Figure 8 (a) also demonstrates the effectiveness of our out-of-order execution scheme for long queries. For the query of 511 amino acids, our pipelined code is 7% faster than the manual version when $D = 1$.

Finally, this biological application implies that both the pure CUDA approach and the hybrid approach of OpenMP and CUDA require a complicated code to overlap the file write stage with other stages. Because the file write stage is a CPU execution stage, it cannot be associated with CUDA streams directly. In this case, we have to ensure that, for each task, the file write stage begins after the completion of the data readback stage. A naive method for achieving this is to call the `cudaMemcpy` function, but this synchronous function prevents pipelined execution. GPU-chariot internally solve this issue by the callback interface. Our GPU-chariot runtime calls the `cudaMemcpyAsync` function and monitors the execution progress by the `cudaStreamQuery` function to process the CPU execution stage immediately after completing the data readback stage. Although this mechanism is complicated, our GPU-chariot runtime hides the mechanism from the application code.

Table 1 Breakdown of execution time for sequence alignment.

Stage	Short query (length: 63)		Long query (length: 511)	
	Execution time (s)	Ratio (%)	Execution time (s)	Ratio (%)
File read	1.90	63.4	1.90	17.0
Download	0.05	1.6	0.05	0.4
Kernel	0.80	26.6	8.18	73.3
Readback	0.03	1.0	0.23	2.1
File write	0.22	7.4	0.80	7.2
Total	2.99	100.0	11.17	100.0

6.3 Case 2: SETI Spectrometer System

We next applied GPU-chariot to a SETI spectrometer system [16], which iteratively applies the 2-D FFT to matrices of 8192×8192 complex numbers. Since each matrix is too

Table 2 Breakdown of execution time for a spectrometer system.

Stage	Column FFT phase		Row FFT phase		Total	
	Execution time (ms)	Ratio (%)	Execution time (ms)	Ratio (%)	Execution time (ms)	Ratio (%)
Download	131.6	49.3	131.6	50.6	263.2	49.9
Kernel	18.3	6.9	11.5	4.4	29.8	5.7
Readback	117.0	43.8	117.0	45.0	234.0	44.4
Total	266.9	100.0	260.1	100.0	527.0	100.0

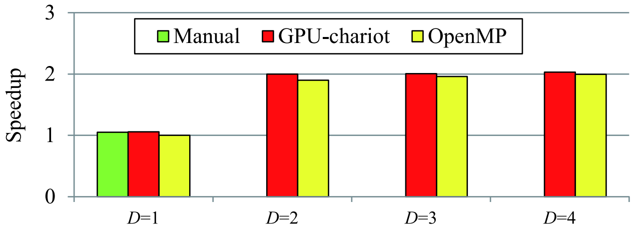


Fig. 9 Speedups of a spectrometer system over a nonpipelined version. The FCFS rule is used for measurement. Manual results for $D \geq 2$ are not available.

large to store in device memory, it is divided into $N = 16$ submatrices. During the first phase, each matrix is divided into 16 submatrices of 512×8192 elements, and then a 1-D FFT is performed on each column (Fig. 1). During the second phase, the intermediate matrix is divided into 16 submatrices of 8192×512 elements, and a 1-D FFT is performed on each row using the CUFFT library [25]. Because chunks have the same data size, we used the FCFS rule for evaluation. Table 2 shows the breakdown of the execution time. Because the download and readback stages consume 94% of the execution time, the performance bottleneck of this system is the data transfer.

Figure 9 shows speedups over a nonpipelined version. Both GPU-chariot and manual versions increase the performance on a single device ($D = 1$) by approximately 5%. These results are reasonable because the kernels consume only 6% of the execution time (Table 2). In contrast, the OpenMP version fails to achieve this overlap. Even when we increased the number of GPUs, the speedups did not increase significantly beyond a factor of two. Since the two PCIe buses in our experimental machine are shared, the I/O bus limits the performance of this data-intensive application. Thus, if data transfer dominates the execution time, multi-GPU solutions cannot achieve a linear speedup beyond B.

7. Conclusion

We have presented a stream programming framework called GPU-chariot, which is capable of out-of-order execution of CPU functions, CUDA kernels, and data transfers on multi-GPU systems. GPU-chariot reduces development efforts needed for task scheduling, resource allocation, and CPU thread management. Using GPU-chariot, programmers can easily develop a host code that constructs software pipelines that lay over an arbitrary number of GPUs and CPUs. Our GPU-chariot runtime maximizes the efficiency of pipelined execution by sorting tasks in a descending order of granularity. With respect to the programmability, GPU-chariot provides a simple programming interface that hides not only CUDA streams but also an appropriate order for CUDA function calls from the application code.

Through experimentation, we showed that our automated code achieves high performance, close to the least upper bound of the speedup. For random size data, our out-of-order code runs up to 20% faster than the manual code that performs in-order execution. Two case studies demonstrate that GPU-chariot can be applied to stream applications that have various pipeline stages, such as CPU functions and CUDA-based functions of third-party libraries.

Future work would include evaluation of GPU-chariot on the latest GK110 architecture.

Acknowledgments

This study was partly supported by the JSPS KAKENHI Grant Number 23300007, 23700057, and the JST CREST program "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems." We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media processing with streams," IEEE Micro, vol.21, no.2, pp.35–46, March 2001.
[2] D. Nagayasu, F. Ino, and K. Hagihara, "A decompression pipeline for accelerating out-of-core volume rendering of time-varying data," Comput. Graph., vol.32, no.3, pp.350–362, June 2008.
[3] J.W. Romein, P.C. Broekema, E. van Meijeren, K. van der Schaaf, and W.H. Zwart, "Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer," Proc. 18th ACM Symp. Parallelism in Algorithms and Architectures (SPAA'06), pp.59–66, June 2006.
[4] 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, March 2008.
[5] NVIDIA Corporation, "CUDA Programming Guide Version 4.2," April 2012.
[6] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU computing," Proc. IEEE, vol.96, no.5, pp.879–899, May 2008.
[7] A. Udupa, R. Govindarajan, and M.J. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," Proc. 7th Int'l Symp. Code Generation and Optimization (CGO'09), pp.200–209, March 2009.
[8] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," Proc. 11th Int'l Conf. Compiler Construction (CC'02), pp.179–196, April 2002.
[9] S. Nakagawa, F. Ino, and K. Hagihara, "A middleware for efficient stream processing in CUDA," Computer Science - Research and Development, vol.25, no.1/2, pp.41–49, May 2010.
[10] A. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: Portable stream programming on graphics engines," Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), pp.381–392, March 2011.
[11] A. Hagiescu, H.P. Huynh, W.F. Wong, and R.S.M. Goh, "Automated

architecture-aware mapping of streaming applications onto GPUs," Proc. 25th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'11), pp.467–478, May 2011.

[12] U. Ramachandran, R.S. Nikhil, J.M. Rehg, Y. Angelov, A. Paul, S. Adhikari, K.M. Mackenzie, N. Harel, and K. Knobe, "Stampede: A cluster programming middleware for interactive stream-oriented applications," IEEE Trans. Parallel Distrib. Syst., vol.14, no.11, pp.1140–1154, Nov. 2003.

[13] V. Bhat, M. Parashar, H. Liu, N. Kandasamy, M. Khandekar, S. Klasky, and S. Abdelwahed, "A self-managing wide-area data streaming service," Cluster Computing, vol.10, no.4, pp.365–383, Dec. 2007.

[14] F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," IEEE Trans. Parallel Distrib. Syst., vol.23, no.4, pp.751–759, April 2012.

[15] H.P. Huynh, A. Hagiescu, W.F. Wong, and R.S.M. Goh, "Scalable framework for mapping streaming applications onto multi-GPU systems," Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'12), pp.1–10, Feb. 2012.

[16] H. Kondo, E. Heien, M. Okita, D. Werthimer, and K. Hagihara, "A multi-GPU spectrometer system for real-time wide bandwidth radio signal analysis," Proc. 2nd Int'l Symp. Parallel and Distributed Processing with Applications (ISPA'10), pp.594–504, Sept. 2010.

[17] J.A. Stuart, C.K. Chen, K.L. Ma, and J.D. Owens, "Multi-GPU volume rendering using MapReduce," Proc. 19th ACM Int'l Symp. High Performance Distributed Computing (HPDC'10), pp.841–848, June 2010.

[18] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using CUDA compatible GPUs," Parallel Computing, vol.36, no.2/3, pp.129–141, Feb. 2010.

[19] L. Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao, "Dynamic load balancing on single- and multi-GPU systems," Proc. 24th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'10), May 2010. 12 pages (CD-ROM).

[20] G. Diamos and S. Yalamanchili, "Speculative execution on multi-GPU systems," Proc. 24th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'10), May 2010. 11 pages (CD-ROM).

[21] R. Ruiz and J.A. Vázquez-Rodríguez, "The hybrid flow shop scheduling problem," European J. Operational Research, vol.205, no.1, pp.1–18, Aug. 2010.

[22] S.A. Brah and G.E. Wheeler, "Comparison of scheduling rules in a flow shop with multiple processors: A simulation," Simulation, vol.71, no.5, pp.302–311, Nov. 1998.

[23] T. Kis and E. Pesch, "A review of exact solution methods for the non-preemptive multiprocessor flowshop problem," European J. Operational Research, vol.164, no.3, pp.592–608, Aug. 2005.

[24] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," May 2012.

[25] NVIDIA Corporation, "CUDA Toolkit 4.2 CUFFT Library," April 2012.

[26] Y. Munekawa, F. Ino, and K. Hagihara, "Accelerating Smith-Waterman algorithm for biological database search on CUDA-compatible GPUs," IEICE Trans. Inf. & Syst., vol.E93-D, no.6, pp.1479–1488, June 2010.

[27] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, Parallel Programming in OpenMP, Morgan Kaufmann, San Mateo, CA, 2000.

[28] NVIDIA Corporation, "GPU Computing SDK," April 2012. http://developer.nvidia.com/gpu-computing-sdk/

[29] 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.

Appendix A: Stream Processing with CUDA

The CUDA programming model assumes that the host and device (i.e., the CPU and GPU) have their own separate memory spaces, called host memory and device memory, respectively. Therefore, CUDA applications usually consist of two types of source codes, namely the host code and device code. The host code runs on the CPU to invoke the device code (i.e., the kernel function) on the GPU and to transfer I/O data streams between host memory and device memory.

A CUDA stream can be defined by creating a stream object, as shown in line 9 of Fig. A·1. In this example, a task contains download, kernel execution, and readback commands. Moreover, N tasks are processed using S CUDA streams in a round-robin fashion. As shown in Fig. A·2 (a), we can use multiple CUDA streams to overlap data transfer with kernel execution. However, because of implicit synchronization, mentioned in Sect. 3.2, on most GPUs, the

```
 1  cudaStream_t stream[S]; // S: number of CUDA streams
 2  float *h, *d_in, *d_out;
 3
 4  cudaMallocHost(&h, N * size); // N: number of tasks
 5  cudaMalloc(&d_in, N * size); // size: size of a chunk
 6  cudaMalloc(&d_out, N * size);
 7
 8  for (int i=0; i<S; i++) { // for each CUDA stream
 9      cudaStreamCreate(&stream[i]);
10  }
11  for (int i=0; i<N; i++) { // for each task
12      cudaMemcpyAsync(d_in + i * size, h + i * size,
          size, cudaMemcpyHostToDevice, stream[i % S]);
13      myKernel<<<gridDim, blockDim, 0, stream[i % S]>>>
14          (d_out + i * size, d_in + i * size, size);
15      cudaMemcpyAsync(h + i * size, d_out + i * size,
          size, cudaMemcpyDeviceToHost, stream[i % S]);
16  }
17  cudaThreadSynchronize();
18
19  cudaFree(d_in);
20  cudaFree(d_out);
21  cudaFreeHost(h);
```

Fig. A· 1 Naive host code of stream processing. This example uses S CUDA streams to process N tasks on a single-GPU system.

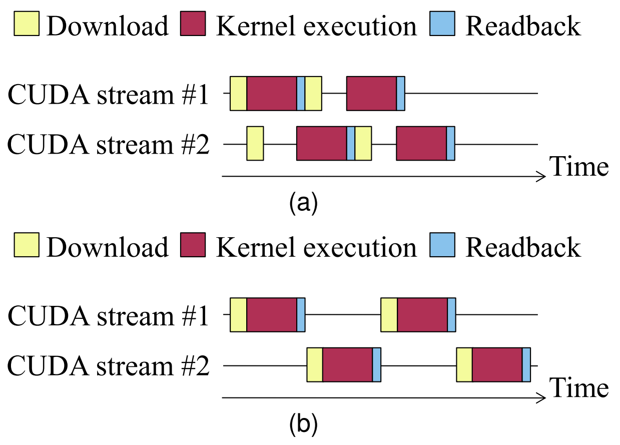


Fig. A· 2 Timeline view of (a) overlapping execution and (b) nonoverlapping execution of commands on a single GPU with a single DMA engine. In overlapping execution, the GPU is kept busy, whereas in nonoverlapping execution, the GPU waits for the initiating data to be downloaded. Both cases use two CUDA streams to process four tasks.

code in Fig. A· 1 results in nonoverlapping execution, as shown in Fig. A· 2 (b). In this example, the first readback command from CUDA stream #1 is issued before the first download command from CUDA stream #2. Thus, the former command prevents the latter from being overlapped with kernel execution.

The host code in Fig. A· 1 becomes more complicated for multi-GPU systems, because it must be multi-threaded to process multiple tasks on CPU cores. Note that CUDA 4.0 and later allow CPU threads to access all GPUs in the system, hence removing the limitation of one-CPU-thread-per-GPU. However, multi-threaded host code is essential in constructing multiple software pipelines, which parallelize CPU stages to run stream applications efficiently on multi-GPU systems. For details, see the cudaOpenMP code in the GPU Computing SDK [28].

Appendix B: Scheduling Algorithm

Our parallel scheduling threads process the online algorithm presented in Algorithm 1. The algorithm produces an online schedule from five inputs: device ID d ($0 \leq d \leq D - 1$), the number of CUDA streams S, the number of pipeline stages K, the task buffer $\mathcal{Q}$, and a set $V = \{\mathcal{L}, \mathcal{G}\}$ of local and global vectors.

During thread creation, each thread independently initializes data structures for the k-th stage, where $1 \leq k \leq K$. Each data structure includes a shared priority queue P_k, active CUDA stream ID W_k, local counter $C_k.local$, and global counter $C_k.global$. In line 8, created threads also initialize the active task ID A_j. This variable stores the ID of the active task that is currently assigned to CUDA stream j, where $1 \leq j \leq S$. Since all variables except priority queues and global counters contain per-device information, each scheduling thread stores them as local, private variables.

After this initialization phase, parallel threads start to independently circulate pipeline stages until they process all tasks buffered before a synchronization request. First, the threads call the task_selection() function to choose an overlappable command f at the k-th stage. If such a command exists, the threads execute the four steps presented in Sect. 5.4.

Algorithm 2 shows the algorithm implemented in the task_selection() function. In order to select an overlappable command f at the k-th stage, this function first searches idle CUDA streams in a cyclic manner. If the idle CUDA stream does not have a task, it then assigns a task from the task buffer $\mathcal{Q}$. Otherwise, it checks if the head command f in the queue can be overlapped with a running command. This investigation is performed by the canOverlap() function, according to conditions (a), (b), and (c) in Sect. 5.3. If the task_selection() function is called before resources are allocated (i.e., if *pop* is set to false), this function simply returns the overlappable command f, which is then used in the resource allocation request. Otherwise, the thread is allowed to dequeue command f for execution. In this case, the function task_selection() returns a pair $\langle f(args), j\rangle$, where f is the selected command, $args$ is its arguments, and j is the CUDA stream to be used for execution.

Algorithm 1 Scheduling algorithm($d, S, K, \mathcal{Q}, V$)

Input: Responsible device ID d, number S of CUDA streams, number K of pipeline stages, task buffer $\mathcal{Q}$, and set $V = \{\mathcal{L}, \mathcal{G}\}$ of local and global vectors.
Output: An online schedule.

```
 1: for $k \leftarrow 1$ to $K$ do                                  ▹ for each pipeline stage
 2:     Set $P_k$ be an empty queue                               ▹ shared priority queue
 3:     $W_k \leftarrow 0$                                        ▹ active CUDA stream ID
 4:     $C_k.local \leftarrow 0$                                  ▹ local counter
 5:     $C_k.global \leftarrow 0$                                 ▹ shared global counter
 6: end for
 7: for $j \leftarrow 1$ to $S$ do
 8:     $A_j \leftarrow NULL$                                     ▹ active task on CUDA stream $j$
 9: end for
10: $\mathcal{A} \leftarrow \{A_1, A_2, \ldots, A_S\}$
11: Initialize $K \times K$ matrix $\mathcal{M}$ by hardware inspection   ▹ defines
    overlappable stages
12: Synchronize all scheduling threads
13: while $\mathcal{Q}$ is not empty or synchronization not requested do
14:     for $k \leftarrow 1$ to $K$ do                              ▹ for each pipeline stage
15:         $\langle f(args), j\rangle \leftarrow$ task_selection($S, \mathcal{Q}, \mathcal{A}, \mathcal{M}, k, W_k, C_k, V$, false)
16:         if $d \notin P_k$ and $f \neq NULL$ then
17:             Enter critical section
18:             Enqueue $d$ to $P_k$                              ▹ request resources
19:             Leave critical section
20:         end if
21:         if $d$ = the head of $P_k$ then
22:             $\langle f(args), j\rangle \leftarrow$ task_selection($S, \mathcal{Q}, \mathcal{A}, \mathcal{M}, k, W_k, C_k, V$,
    true)
23:             if $f \neq NULL$ then
24:                 $C_k.local \leftarrow C_k.local + 1$
25:                 Enter critical section
26:                 $C_k.global \leftarrow C_k.global + 1$
27:                 Leave critical section
28:                 $W_k \leftarrow j$                            ▹ update active stream ID
29:                 Dispatch $f(args)$ to CUDA stream $j$
30:             end if
31:         end if
32:         if CUDA stream $W_k$ is idle then                      ▹ command completed
33:             $C_k.local \leftarrow C_k.local - 1$
34:             Enter critical section
35:             $C_k.global \leftarrow C_k.global - 1$
36:             Dequeue $d$ from $P_k$                            ▹ release resources
37:             Leave critical section
38:         end if
39:     end for
40: end while
```

Appendix C: Lower Bound Analysis

Without loss of generality, we assume that chunks are sorted in ascending order of data size: for all $1 \leq i, j \leq N$, $i \geq j \Rightarrow |e_i| \geq |e_j|$. Let d_i, k_i, and r_i be the data download time, kernel execution time, and data readback time, respectively, for the i-th chunk ($1 \leq i \leq N$).

Our lower bound on the execution time, T_{LB}, is given by

$$T_{LB} = \min(T_1, T_2), \tag{A· 1}$$

where T_1 represents a lower bound on execution time as-

Algorithm 2 Task selection$(S, Q, \mathcal{A}, \mathcal{M}, k, W_k, C_k, V, pop)$

Input: Number S of CUDA streams, task buffer Q, set $\mathcal{A} = \{A_1, A_2, \ldots, A_S\}$ of active tasks, matrix $\mathcal{M}$, stage ID k, active CUDA stream ID W_k, local and global counters C_k, local and global vectors V, and boolean value pop.

Output: Pair $\langle f(args), j\rangle$ of executable function $f(args)$ and CUDA stream ID j.

```
1:  for $i \leftarrow W_k$ to $W_k + S - 1$ do
2:      $j \leftarrow i \bmod S$
3:      if CUDA stream $j$ is idle then
4:          Enter critical section
5:          if $A_j = NULL$ and $Q$ is not empty then
6:              $A_j \leftarrow Q$.pop_front()                         ▷ task assignment
7:          end if
8:          Leave critical section
9:          if $A_j \neq NULL$ and $A_j.queue$ is not empty then
10:             $f \leftarrow$ the head of $A_j.queue$
11:             if canOverlap($f, \mathcal{M}, C_k, V$) and Command $f$ processes the $k$-th stage then
                    ▷ canOverlap() returns true if $f$ satisfies conditions (a), (b). and (c) in Sect. 5.3.
12:                 if $pop$ then
13:                     $f(args) \leftarrow A_j.queue$.pop_front()
14:                     if $A_j.queue$ is empty then
15:                         $A_j \leftarrow NULL$                      ▷ task finished
16:                     end if
17:                 end if
18:                 return $\langle f(args), j\rangle$
19:             end if
20:         end if
21:     end if
22: end for
23: return $\langle NULL, 0\rangle$                                  ▷ no command left
```

suming that data transfer dominates the performance, and T_2 represents a lower bound on execution time assuming that kernel execution dominates the performance. In both cases, the execution time is minimized when the bottleneck stage is kept busy continuously, and both the first download time and the last readback time are minimized.

In cases where data transfer dominates the performance, we assume that N tasks are equally flown on B buses. Then, T_1 is given by

$$T_1 = \sum_{i=1}^{N} (d_i + r_i)/B. \tag{A$\cdot$2}$$

Conversely, for kernel dominant cases, T_2 is given by

$$T_2 = T_{head} + T_{body} + T_{tail}, \tag{A$\cdot$3}$$

where T_{body} represents the sum of kernel execution times, namely the fully masked part, as shown in Fig. A· 3. In addition, T_{head} and T_{tail} represent the unmaskable parts that appear at the beginning and end of execution, respectively. With respect to T_{body}, we assume that N tasks are equally assigned to D GPUs. Therefore, T_{body} is given by

$$T_{body} = \sum_{i=1}^{N} k_i/D. \tag{A$\cdot$4}$$

T_{head} and T_{tail} are given by

$$T_{head} = \sum_{i=1}^{\lceil D/B \rceil} d_i, \tag{A$\cdot$5}$$

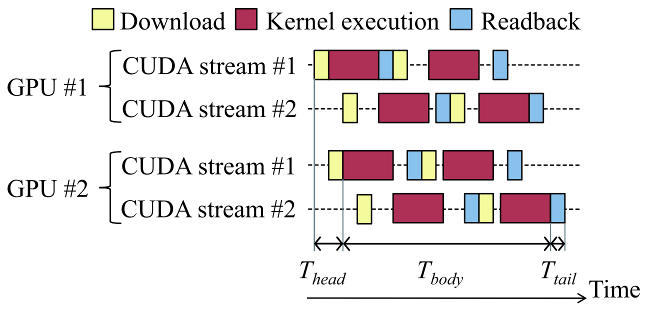


Fig. A· 3 Timeline view of multi-GPU execution. In this example, two GPUs share a single bus. GPU #2 delays the execution because it is blocked until GPU #1 completes the first data download.

$$T_{tail} = r_1. \tag{A$\cdot$6}$$

Note that Eq. (A· 5) considers the delay of kernel execution. This delay is due to the shared bus, which causes waiting time for GPUs, as shown in Fig. A· 3.

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

Shinta Nakagawa received the B.E. and M.E. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 2009 and 2011, respectively. He is currently with NEC Corporation. His current research interests include high performance computing, grid computing, and systems architecture and design.

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