Noname manuscript No.
(will be inserted by the editor)

A Middleware for Efficient Stream Processing in CUDA

Shinta Nakagawa · Fumihiko Ino · Kenichi Hagihara

Received: date / Accepted: date

Abstract This paper presents a middleware capable of out-of-order execution of kernels and data transfers for efficient stream processing in the compute unified device architecture (CUDA). Our middleware runs on the CUDA-compatible graphics processing unit (GPU). Using the middleware, application developers are allowed to easily overlap kernel computation with data transfer between the main memory and the video memory. To maximize the efficiency of this overlap, our middleware performs out-of-order execution of commands such as kernel invocations and data transfers. This run-time capability can be used by just replacing the original CUDA API calls with our API calls. We have applied the middleware to a practical application to understand the run-time overhead in performance. It reduces execution time by 19% and allows us to process large data that cannot be entirely stored in the video memory.

Keywords Stream processing · overlap · CUDA · GPU

1 Introduction

Stream processing [4] has become increasingly important with emergence of stream applications such as audio/video processing [9] and time-varying visualization [5]. In this stream programming model, applications are decomposed into stages of sequential computations, which are expressed as *kernels*. On the other hand, input/output data is organized as *streams*, namely sequences of similar data records. Input streams are then passed through the chain of kernels in a pipelined fashion, producing output streams. One advantage of stream processing is that it can exploit the parallelism inherent in the pipeline. For example, the execution of the stages can be overlapped with each other to exploit task parallelism. Furthermore, different stream elements can be simultaneously processed to exploit data parallelism.

One of the stream architecture that benefit from the advantages mentioned above is the graphics processing unit (GPU), originally designed for acceleration of graphics applications. However, it now can accelerate many scientific applications typically with a 10-fold speedup over CPU implementations [2]. Most of the applications are implemented using a flexible development framework, called compute unified device architecture (CUDA) [1]. This vendor-specific framework allows us to directly manage the hierarchy of memories, which is an advantage over graphics APIs such as OpenGL. Using this C-like language, developers are allowed to implement their GPU programs without knowledge of computer graphics.

In general, CUDA-based applications consist of host code and device code, each running on the CPU and on the GPU, respectively. The host code invokes the device code that implements kernels of stream applications. Thus, the GPU is treated as a coprocessor of the CPU. On the other hand, input/output streams must be stored in video memory, called device memory. Since device memory is separated from main memory, namely host memory, streams have to be transferred between them. Data transfer from host memory to device memory is called "download" and "readback" in the opposite direction. Data transfer can be overlapped with ker-

S. Nakagawa · F. Ino · K. Hagihara
Graduate School of Information Science and Technology,
Osaka University,
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Tel.: +81-6-6879-4353
Fax: +81-6-6879-4354
E-mail: {s-nakagw, ino}@ist.osaka-u.ac.jp

nel computation using the asynchronous CUDA API, which immediately returns from API calls before their completion.

Notice here that the term "stream" in CUDA has a definition different from that mentioned above. To avoid confusion, we denote the former as *"CUDA stream"* in this paper. A CUDA stream is a stream object, which denotes a sequence of commands that execute in order [1]. Such commands contain data download, kernel execution, and data readback. Using multiple CUDA streams, we can implement stream applications on the GPU. For example, a typical implementation will associate each of the stream elements with a CUDA stream to overlap kernel computation for a CUDA stream with data transfer for another CUDA stream.

One technical problem in CUDA is that data transfer commands from CUDA streams are processed in an in-order sequence. Similarly, kernels are also executed in an in-order sequence. These constraints will result in non-overlapping execution though there is no data dependence between different stream elements. For example, a readback command from a CUDA stream can prevent a download command of another CUDA stream from being overlapped with kernel computation. Therefore, developers have to call asynchronous API functions in the appropriate order to realize overlapping execution. Finding the appropriate order is a time-consuming task because it depends on run-time situations such as API call timings and application code.

In this paper, we propose a middleware capable of out-of-order execution of kernels and data transfers, aiming at achieving efficiently overlapped stream processing in CUDA. The proposed middleware is designed to reduce the development efforts needed for GPU accelerated stream processing. It automatically changes the execution order to maximize the performance by overlapping data transfer with kernel execution. This run-time optimization can easily be done by replacing the original CUDA API calls with our middleware API calls in host code. Our middleware currently assumes that (1) the application can be adapted to the stream programming model and (2) the kernel execution time and the data transfer time does not significantly vary between different stream elements. The middleware is implemented as a class of the C++ language.

The rest of the paper is organized as follows. Section 2 introduces some related work. Section 3 presents preliminaries needed to understand our middleware. Section 4 then describes the details of the middleware. Section 5 shows some experimental results obtained using a practical application. Finally, Section 6 concludes the paper with future work.

2 Related Work

To the best of our knowledge, there is no work that tries to automate the overlap for CUDA-enabled stream applications. However, some development frameworks focus on GPU-accelerated stream processing. For example, Yamagiwa et al. [8] propose a stream-based distributed computing environment. Their environment is implemented using the DirectX graphics library. Our middleware differs from their environment in that the middleware focuses on the CUDA framework and optimizes the interaction between the GPU and the CPU.

Hou et al. [3] present bulk-synchronous GPU programming (BSGP), which is a programming language for stream processing on the CUDA-compatible GPU. Using their compiler, application developers can automatically translate sequential C programs into stream kernels with host code. Thus, the main focus of this language is to free application developers from the tedious chore of temporary stream management. Our middleware has almost the same purpose as their language but provides an out-of-order execution capability to maximize the effects of stream processing.

Some practical applications [6, 7] are implemented using the stream programming model with achieving the overlap. However, their purpose is to accelerate specific applications. In contrast, our middleware provides a general framework to application developers.

3 Preliminaries

This section explains how stream processing can be done in CUDA.

3.1 Stream Programming Model

We consider here a simple stream model that produces an output stream by applying a chain of commands, $f_1, f_2, \ldots, f_m$, to an input stream. Parameter m here represents the number of commands that compose the chain. Let I and O be the input stream and the output stream, respectively. The input stream then can be written as $I = \{e_1, e_2, \ldots, e_n\}$, where e_i $(1 \leq i \leq n)$ represents an element of the input stream and n represents the number of elements in the stream. Similarly, we have the output stream $O = \{g_1, g_2, \ldots, g_n\}$, where g_i $(1 \leq i \leq n)$ represents an element of the output stream. The computation then can be expressed as

$$g_i = f_m \circ f_{m-1} \circ \cdots \circ f_1(e_i), \tag{1}$$

where $1 \leq i \leq n$ and $\circ$ represents the composition operator. In other words, the computation consists of n

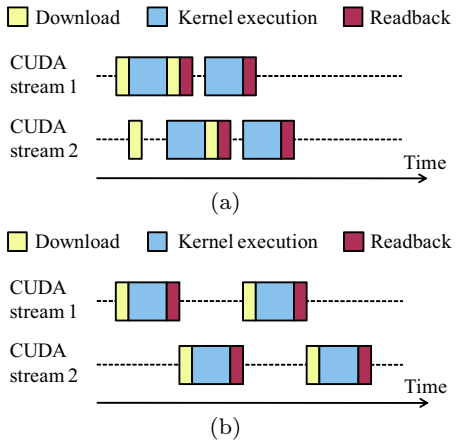

Fig. 1 Timeline view of naive methods running on a single GPU. (a) Overlapping execution continuously runs kernels but (b) non-overlapping execution causes waiting time due to data transfer commands. In both cases, four tasks are processed using two CUDA streams ($n = 4$ and $l = 2$). See also Fig. 2 for the details of the corresponding pseudocode.

```
Input: Input stream I = {e_1, e_2, ..., e_n}, size n, and
       number l of CUDA streams.
Output: Output stream O = {g_1, g_2, ..., g_n}.
 1: cudaStream_t str[l];
 2: for i = 1 to n do
 3:    Download stream element e_i using str[i mod l];
 4: end
 5: for i = 1 to n do
 6:    Launch kernel to compute g_i from e_i using str[i mod l];
 7: end
 8: for i = 1 to n do
 9:    Readback output stream g_i using str[i mod l];
10: end
11: cudaThreadSynchronize();
```

(a)

```
Input: Input stream I = {e_1, e_2, ..., e_n}, size n, and
       number l of CUDA streams.
Output: Output stream O = {g_1, g_2, ..., g_n}.
 1: cudaStream_t str[l];
 2: for i = 1 to n do
 3:    Download stream element e_i using str[i mod l];
 4:    Launch kernel to compute g_i from e_i using str[i mod l];
 5:    Readback output stream g_i using str[i mod l];
 6: end
 7: cudaThreadSynchronize();
```

(b)

Fig. 2 Pseudocode of naive methods. (a) Overlapping version [1] with pre-issued commands and (b) non-overlapping version without pre-issued commands.

tasks, where a *task* corresponds to the chain of commands to a stream element.

Let $\rightarrow$ be the relation that represents data dependence. A sequential order of commands, $f_{s,i} \rightarrow f_{t,j}$, defines that the s-th command to the i-th element e_i have to be processed before the t-th command to the j-th element e_j, where $1 \leq s, t \leq m$ and $1 \leq i, j \leq n$. The data dependencies in the stream model can be expressed as

$$\forall s < t,\ f_{s,i} \rightarrow f_{t,i}, \tag{2}$$

where $1 \leq i \leq n$. Equation (2) indicates that commands to the same element have to be serially processed in a pipelined fashion. In contrast, different elements can be independently processed in the pipeline. Therefore,

$$\forall i \neq j,\ f_{s,i} \not\rightarrow f_{t,j}, \tag{3}$$

where $1 \leq s, t \leq m$ and $\not\rightarrow$ denotes the negation of $\rightarrow$.

3.2 Stream Processing in CUDA

Suppose that we have a stream application with a single kernel and have a single GPU for acceleration. In this case, the chain of commands consists of data download, kernel execution, and data readback ($m = 3$). Eq. (3) then indicates that kernel computation for a CUDA stream can be overlapped with data transfer for another CUDA stream, as shown in Fig. 1(a).

Let T_D, T_K, and T_R denote the data download time per task, the kernel execution time per task, and the data readback time per task, respectively. Suppose that the kernel execution time is much longer than the data transfer time: $T_K \gg T_D + T_R$. The optimal execution time T_{opt} then will be obtained if the GPU continuously runs kernels for all n tasks, as shown in Fig. 1(a). In this case, the optimal time can be given by

$$T_{opt} = T_D + nT_K + T_R. \tag{4}$$

This optimal time will be obtained if data transfers are fully overlapped with kernel computation using the asynchronous API with multiple CUDA streams, as shown in Fig. 2(a). In this code, n tasks are processed using l CUDA streams. All of download commands are first issued in an asynchronous mode. After this, all of kernels are invoked before any readback commands. Finally, cudaThreadSynchronize() is called to finalize all CUDA streams before proceeding further. Figure 1(a) illustrates the timing behavior of this code.

One problem in this naive code is that all of download commands are issued before kernel execution. Such a pre-issue strategy is not possible in stream applications that produce input data streams at regular intervals. To avoid this pre-issue, we can modify the loop structure as shown in Fig. 2(b). In this strategy, the chain of download, kernel execution, and readback commands is serially issued for every task, so that we can deal with input data streams that come one after another. Furthermore, it can reduce the video memory consumption from $O(n)$ to $O(1)$. However, our preliminary evaluation shows that such a strategy fails to overlap data transfer with kernel computation. Figure 1(b) illustrates how the code is executed on the GPU. The

problem here is that the readback command for the first task blocks the download command for the second task. We also carried out other preliminary experiments to find the constraints in CUDA streams. The constraints can be summarized as follows.

C1. Data transfer commands from CUDA streams are processed in an in-order sequence.
C2. Kernels are also executed in an in-order sequence. However, kernel and data transfer commands have no constraint except for Eq. (2).
C3. Graphics drivers currently do not support multi-threaded data transfer.
C4. Running kernels cannot be stopped until they complete their execution. Similarly, data transfer commands cannot be cancelled after their issue.

These constraints indicate that we need some mechanisms to perform out-of-order execution at run-time.

4 Proposed Middleware

The goal of our middleware is to realize overlapped stream processing without using the pre-issue strategy in CUDA. In other words, the middleware must realize overlapping execution using host code similar to that in Fig. 2(b). To achieve this, we dynamically reorder issued commands and schedule them with the appropriate order for the overlap. This section describes how such a scheduling mechanism is implemented in the middleware.

4.1 Architectural Design

The architectural design of our middleware mainly consists of three points as follows.

– Application-level mechanism. Since the details of graphics drivers are not opened to public, we have decided to implement the out-of-order capability at the application level. Therefore, application developers must modify their host code to replace CUDA API calls with our middleware API calls. Our API then not only invokes the appropriate CUDA functions but also performs reordering of issued commands before invocation.
– Buffering strategy. As we mentioned in Section 3.2, the optimal time usually will not be achieved if the GPU waits for kernel invocation. To prevent such waiting time, we have to buffer CUDA API calls as candidates for execution. Otherwise, we will fail to change the execution order at run-time due to constraint C4. Thus, our middleware has a *command buffer* that stores instances of CUDA API calls.

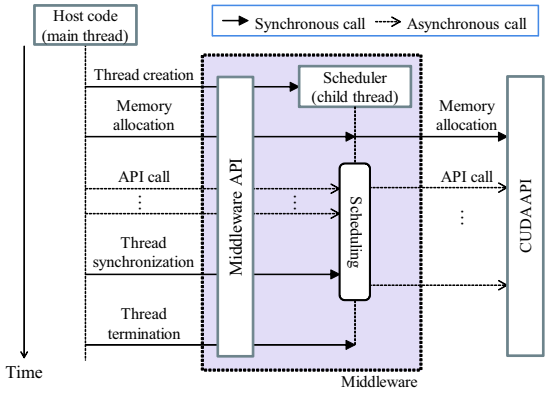

Fig. 3 Timing behavior of proposed middleware. Our middleware receives API calls from host code that implements the application. It creates a CPU thread to perform run-time scheduling for overlap. A thread is responsible for a single GPU that runs multiple CUDA streams.

– Scheduler thread creation. Since kernels can complete their execution asynchronously from other CUDA API calls, it is better to create another CPU thread in order to separate the scheduling capability from the original thread that executes the host code. This separation allows the created child thread to devote itself to continuously execute kernels on the GPU. Thus, we can execute a kernel without causing waiting time if the kernel is buffered for execution. One drawback of this strategy is that it consumes more CPU resources due to the additional thread.

Figure 3 illustrates how the middleware behaves between the host code and CUDA API calls. In this figure, synchronous and asynchronous function calls are represented by solid arrows and dashed arrows, respectively. As shown in this figure, CUDA API calls are encapsulated within our middleware API calls. The middleware first creates a child thread from the host code, namely a master thread. The child thread then runs as a scheduler who dispatches tasks to CUDA streams. The scheduler will terminate when all issued commands are completed after thread synchronization.

The scheduler also takes the responsibility of command reordering and load balancing between CUDA streams. As we mentioned before, our scheduler has a command buffer to enqueue and dequeue API call instances at run-time. Using this buffer, the scheduler optimizes the execution order during program execution. Since commands will be issued one after another, the order can vary with the progress of program execution. We present the details of the scheduling algorithm later in Section 4.3. A load balancing capability is also needed to continuously invoke kernels using multiple CUDA streams, as shown in Fig. 1(a).

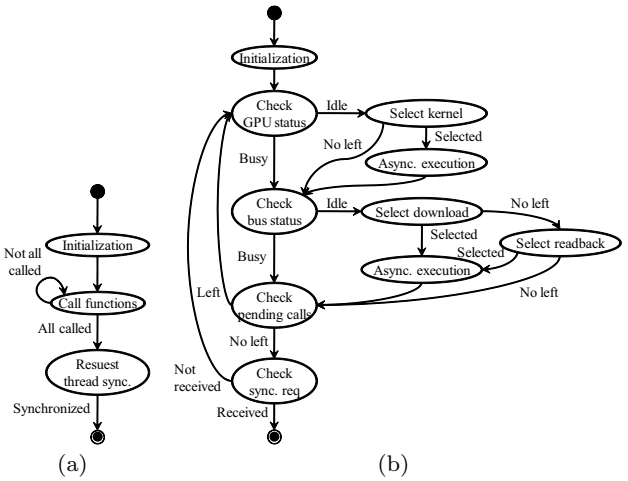

Fig. 4 State transition diagrams of run-time scheduler. (a) Diagram for the host code (main thread) and (b) that for the scheduler (child thread).

4.2 Run-time Scheduler

Figure 4 shows state transition diagrams for the host code and the scheduler. The host code runs as the main thread that creates a child thread for the scheduler. The code iteratively calls middleware API functions in an asynchronous mode, as shown in Fig. 3. On thread creation, the scheduler will be initialized with a default constructor, which sets the number l of CUDA streams and allocates memory for tasks. The middleware then enqueues the corresponding CUDA call instances into the command buffer for later execution. Once all API functions are called from the host code, it then requests synchronization to the child thread to finalize all of issued commands. This synchronization is necessary because threads are running in an asynchronous manner.

On the other hand, the scheduler selects the appropriate command from the command buffer, which manages all of uncompleted instances at that time. After this selection, it calls the corresponding CUDA API functions in an asynchronous manner. Since the scheduler must access the command buffer, the buffer is shared between the master thread and the child thread. This also indicates that we need a lock/unlock mechanism to implement mutual exclusion for the buffer. We solve this problem by a critical section.

Though the details of the selection algorithm is described later in Section 4.3, we present here the selection priority between download, kernel execution, and readback commands. The key idea to achieve the overlap is that the middleware tries to have at least a kernel call instance in the command buffer, which prevents the GPU from waiting kernel invocation. Therefore, we give the highest priority to kernel call commands, as shown in Fig. 4. The second priority is given to download commands to make sure that at least a buffered kernel instance is ready for the next execution. Finally, readback commands are executed when the GPU is busy for kernel execution and there is no download instance in the command buffer.

Thus, the priority mentioned above implements the out-of-order capability of the middleware. Since the order is determined at run-time, the generated order might be different from the optimal order that can be determined after program execution.

4.3 Scheduling Algorithm

Suppose that the middleware has a set S of CUDA streams and a set Q of command buffers. For l CUDA streams, the former can be written as $S = \{s_0, s_1, \ldots, s_{l-1}\}$, where s_i $(0 \leq i \leq l-1)$ represents a stream object. Similarly, the latter can be written as $Q = \{q_0, q_1, \ldots, q_{l-1}\}$, where q_i $(0 \leq i \leq l-1)$ represents the buffer for the i-th CUDA stream s_i.

The basic idea of our scheduling algorithm is that it invokes kernels as soon as it can, and then tries to transfer data to overlap it with kernel computation. The algorithm mainly consists of two steps as follows.

1. Task assignment and buffering. On every middleware function call, the algorithm assigns the corresponding task to a CUDA stream s_i, where $1 \leq i \leq l-1$. The CUDA function instance that corresponds to the called middleware function is also enqueued to buffer q_i, which manages s_i.
2. Task selection and execution. The algorithm iteratively checks the status of running CUDA call instances. At every completion of an instance, it immediately selects and dequeues a CUDA call instance from the command buffer and then executes the dequeued instance. An instance A will be selected (1) if there is no running instance or (2) if there is a running instance that can be overlapped with the instance A.

Note here that both steps are asynchronously processed in the middleware. Step 1 is processed by the main thread while step 2 is iteratively processed by the child thread until the completion of all commands.

Figure 5 presents a pseudocode of the task assignment and buffering algorithm. This algorithm runs as a part of middleware functions replaced with the CUDA functions in host code. In addition to the arguments $args$ of the CUDA function f, our middleware function requires an additional argument, namely the task identifier t, to associate the instance of f with a task. In a case of Fig. 2(b), application developers can give

Algorithm 1: Task assignment and buffering
Input: CUDA function f, its arguments $args$, task identifier t, shared buffer set Q, and number l of CUDA streams.
Output: Updated buffer set Q.
1: $i := t \bmod l$;
2: Enter critical section;
3: Enqueue $f(args)$ into $q_i \in Q$;
4: Leave critical section;

Fig. 5 Pseudocode of task assignment and buffering algorithm. This algorithm runs as a part of middleware API calls replaced with the CUDA API calls. Using the task identifier, tasks are assigned to CUDA streams in a cyclic manner.

Algorithm 2: Task selection and execution
Input: CUDA stream set S, buffer set Q, and number l of CUDA streams.
1: $u := 0$; $v := 0$; // identifiers of active CUDA streams
2: **while** (command left in Q) **or** (no sync. request) **do**
3: **if** GPU is idle **then**
4: $\langle f(args), i\rangle :=$ select$(S, Q, l,$ "kernel"$, u)$;
5: **if** $f \neq NULL$ **then**
6: $u := i$; // update active identifier
7: Execute $f(args)$ using $s_i \in S$;
8: **end**
9: **end**
10: **if** Bus is idle **then**
11: $\langle f(args), i\rangle :=$ select$(S, Q, l,$ "download"$, v)$;
12: **if** $f = NULL$ **then**
13: $\langle f(args), i\rangle :=$ select$(S, Q, l,$ "readback"$, v)$;
14: **end**
15: **if** $f \neq NULL$ **then**
16: $v := i$; // update active identifier
17: Execute $f(args)$ using $s_i \in S$;
18: **end**
19: **end**
20: **end**

Function select$(S, Q, l, type, j)$
Input: CUDA stream set S, buffer set Q, number l of CUDA streams, command type $type$, and CUDA stream identifier j.
Output: Pair $\langle f(args), i\rangle$ of executable function $f(args)$ and CUDA stream identifier i.
1: **for** $k = j$ **to** $j + l - 1$ **do**
2: $i := k \bmod l$;
3: **if** ($s_i \in S$ is idle) **and** ($q_i \in Q$ is not empty) **then**
4: Set f as the first element of q_i; // f: buffered command
5: **if** f is $type$ command **then**
6: Enter critical section;
7: Dequeue $f(args)$ from q_i; // $args$: arguments of f
8: Leave critical section;
9: **return** $\langle f(args), i\rangle$;
10: **end**
11: **end**
12: **end**
13: **return** $\langle NULL, 0\rangle$; // no left

Fig. 6 Pseudocode of task selection and execution algorithm. According to the state transition diagram in Fig. 4(b), command buffers are scanned in a cyclic manner.

variable i as the task identifier to the middleware. Using this identifier, the algorithm assigns tasks to CUDA streams in a cyclic manner. The master thread has to enter a critical section to access the shared command buffer.

In contrast to the algorithm that runs on the master thread, the child thread executes the task selection and execution algorithm presented in Fig. 6. This algorithm takes sets S and Q, and their size l as inputs, then executes commands using the state transition diagram in Fig. 4(b). Variables u and v in line 1, where $0 \leq u, v \leq l - 1$, represent the identifiers of active stream objects that currently occupy the GPU and the data bus, respectively. These identifiers are used to switch CUDA streams in a cyclic manner, which contributes to achieve load balancing between CUDA streams.

The algorithm iteratively calls select() function to choose the command that should be executed from buffers. This function requires command type *type* and active stream identifier j as inputs. The type here is one of "kernel," "download," and "readback." The function then checks command buffers and CUDA stream status from the active CUDA stream s_j in a cyclic manner. Finally, it returns a pair $\langle f(args), i\rangle$ of a CUDA API instance and a stream object identifier. Given a pair $\langle f(args), i\rangle$ from select() function, the middleware executes $f(args)$ using CUDA stream s_i.

5 Experimental Results

We now show evaluation results to demonstrate the effectiveness of our middleware. Firstly, the run-time overhead is investigated with overlapping effects. We then show case study to understand how the middleware can be used in practical applications.

In experiments, we used a PC having an Intel Core i7 940 CPU and an NVIDIA GeForce GTX 280 GPU. The system is equipped with 12 GB main memory and 1 GB device memory. We have installed CUDA 2.3 and CUDA driver 190.38 with Windows XP 64-bit Edition.

5.1 Overhead Analysis

We applied the middleware to two dummy programs: a compute-intensive kernel and a memory-intensive kernel. The reason why we use dummy programs here is that we need to arbitrarily change the kernel execution time T_K and the data transfer time $T_D + T_R$ for investigation. In the following discussion, let r be the time ratio given by $r = (T_D + T_R)/T_K$. Both dummy kernels are iteratively invoked to process 16 stream elements ($n = 16$). Each element consists of an array of integers with size d ranging from 2M to 20M. The time ratio r can be controlled by changing the array size.

The compute-intensive kernel loads integers from global memory, and then repeats shift operations on them. After this, it stores results back to global memory. All of the data accesses are efficiently carried out in a coalesced manner [1]. In contrast, the memory-intensive kernel measures the memory bandwidth by iteratively performing data load/store to global memory. Since data transfer between device memory and main memory consumes memory bandwidth, we use

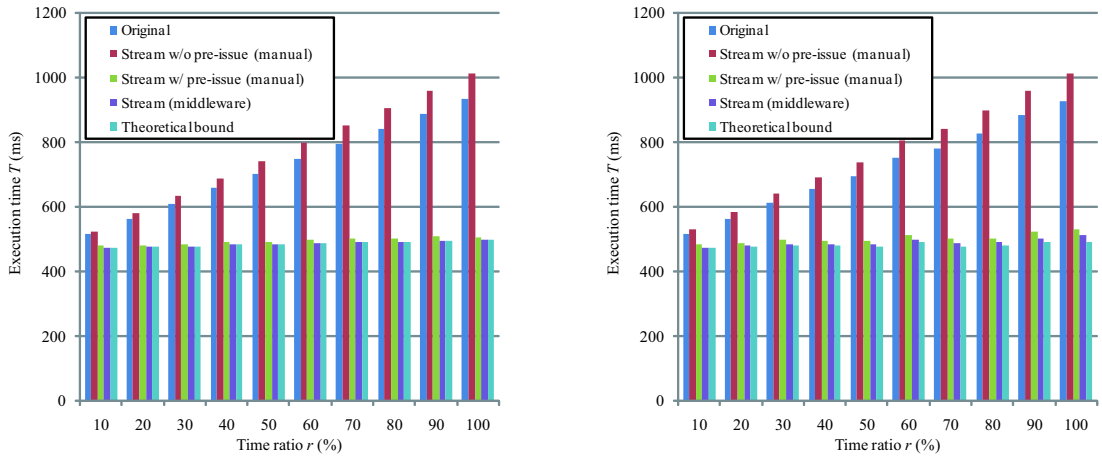

Fig. 7 Measured time for (a) compute-intensive kernel and for (b) memory-intensive kernel. Results contain execution time of original version, that of manually written stream version with/without pre-issue, that of stream version generated by middleware, and theoretical bound of stream version. Theoretical bound is given by Eq. (4).

this kernel to understand the overlapping effects under a bandwidth exhausted situation.

Figure 7 explains how the middleware reduces the execution time by the overlap. It shows five results, containing the measured time of the original program, that of the manually written stream program with/without the pre-issue strategy, that of the stream program generated by our middleware, and the theoretical bound of the stream program. The theoretical bound here is given by Eq. (4).

By comparing middleware results with manual results, we can see that the middleware efficiently realizes the overlap. On the other hand, the manually written code fails to overcome the original code if the pre-issue strategy is not employed in the code. The reason why our middleware outperforms the original code is that it hides the overheads inherent in CUDA streams. Such overheads include (1) the initialization of CUDA streams and (2) the scheduling cost of the middleware. The former can be represented as $O(dl)$. This initialization time ranges from 2 ms to 20 ms in this experiment. The latter also increases with the number l of CUDA streams, but the scheduling time is at most 20 μs per API instance. Therefore, such a scheduler can generate better execution order of commands with small overhead, which achieves higher performance than manually written stream versions.

The middleware achieves almost no overhead for the compute-intensive kernel. For the memory-intensive kernel, on the other hand, the overhead increases with the ratio r and reaches 4.6% when data transfers take the same time as kernel execution ($r = 0.5$). In such bandwidth exhausted situations, the kernel performance can be decreased due to data transfers. Thus, overlapping execution runs efficiently if the kernel is compute-intensive rather than memory-intensive.

5.2 Case Study

We applied the middleware to an optical propagation simulator based on the Fourier transform. The simulator downloads six input streams, invokes two different kernels ($m = 2$), and readbacks two output streams. The data size of input stream elements and that of output stream elements are 1224 bytes and 32 KB, respectively. In general, this simulator is iteratively executed as a parameter-sweep application, which processes a large number of data elements in practical situations. We measured the performance using at most 50,000 tasks ($400 \leq n \leq 50,000$), which require 1.6 GB of video memory to run.

We first modified the host code to replace CUDA function calls with middleware function calls, as shown in Fig. 8. In this code, "CudaStreamOptimizer" is the middleware class and a variable "obj" represents its instance. In addition to this replacement, we also added some API calls for initialization, synchronization, and finalization. We next measured the breakdown of execution time using the original code. It takes 0.06 ms and 0.04 ms to execute download and readback commands for a single task, respectively. On the other hand, the first kernel and the second kernel take 0.19 ms and 0.08 ms for a task, respectively.

Figure 9 shows the execution time with different numbers of tasks. The original and manual stream versions cannot process more than 29,600 tasks due to the

```
Input: Pointer h_in to input stream, task size n, number l of
       CUDA streams, memory size memSize, kernel function
       kernel, parameters gridDim and blockDim.
Output: Pointer h_out to output stream.
 1: CudaStreamOptimizer obj; // An instance of middleware
 2: obj.initialize();
 3: obj.setStream(l); // Set the number of CUDA streams
 4: float *d_in, *d_out; // Pointers to device memory
 5: obj.malloc(&d_in, memSize); // Replaces cudaMalloc()
 6: obj.malloc(&d_out, memSize);
 7: for (int i = 0; i < n; i++) { // Use i as task identifier
 8:    obj.download(i, d_in, h_in + memSize * i, memSize);
 9:    obj.launchKernel(i, kernel, gridDim, blockDim,
          d_in, d_out, ..., NULL);
10:    obj.readback(i, h_out + memSize * i, d_out, memSize);
11: }
12: obj.synchronize();
13: obj.finalize();
```

Fig. 8 Pseudocode of modified host code. n tasks are processed using l CUDA streams. Each CUDA stream allocates $memSize$ bytes of video memory for computation. Arguments $kernel$, $gridDim$, $blockDim$ are the function and parameters used in the original code.

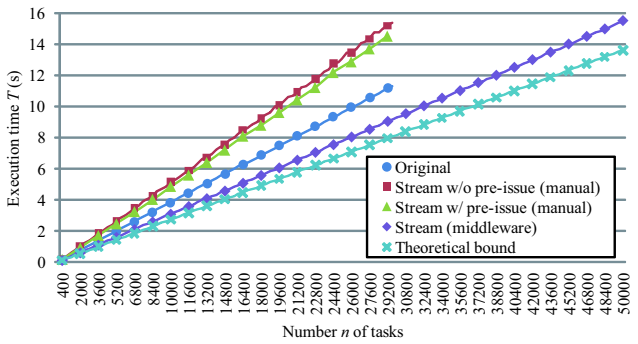

Fig. 9 Measured time for an optical simulator with different numbers of tasks. Our middleware allows us to deal with large-scale simulations that exhaust the video memory.

lack of video memory. In contrast, our middleware saves the memory consumption so that successfully obtains simulation results for 50,000 tasks.

With respect to performance, the main difference to dummy programs is that both manually written code fails to overcome the original code though we employ the pre-issue strategy. This is due to the initialization time mentioned before, because this overhead increases with the number l of CUDA streams. While our middleware always uses two CUDA streams ($l = 2$) with buffer reuse, the manual code requires n CUDA streams for n tasks. Therefore, the overhead in the manual code increases as we increase the number n of tasks. In contrast, our middleware can minimize this overhead, improving the performance by 19%.

6 Conclusion

We have presented a middleware capable of out-of-order execution of CUDA kernels and data transfers for overlapped stream processing on the GPU. The middleware reduces the development efforts needed for the overlap. It has a run-time mechanism that maximizes overlapping effects by finding the appropriate execution order from issued API calls.

In experiments, our middleware successfully overlaps data transfer with kernel computation. We also find that the overlapping effects are increased by out-of-order execution and these effects are close to the theoretical bound for two dummy programs. We also have shown case study, where the middleware improves the performance of an optical simulator by 19%. It allows us to deal with large data that cannot be entirely stored in the video memory.

Future work includes the support of load balancing to deal with unbalanced tasks.

Acknowledgements This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(20240002) and the Global COE Program "in silico medicine" at Osaka University. We would like to thank the anonymous reviewers for their valuable comments.

References

1. NVIDIA Corporation. CUDA Programming Guide Version 2.3, July 2009. `http://developer.nvidia.com/cuda/`.
2. M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27, July 2008.
3. Q. Hou, K. Zhou, and B. Guo. BSGP: Bulk-synchronous GPU programming. *ACM Trans. Graphics*, 27(3), Article 19, Aug. 2008.
4. 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*, 21(2):35–46, Mar. 2001.
5. D. Nagayasu, F. Ino, and K. Hagihara. A decompression pipeline for accelerating out-of-core volume rendering of time-varying data. *Computers and Graphics*, 32(3):350–362, June 2008.
6. J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC'08)*, Nov. 2008. 9 pages (CD-ROM).
7. C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W.-M. W. Hwu. GPU acceleration of cutoff pair potentials for molecular modeling applications. In *Proc. 5th Int'l Conf. Computing Frontiers (CF'08)*, pages 273–282, May 2008.
8. S. Yamagiwa and L. Sousa. Design and implementation of a stream-based distributed computing platform using graphics processing units. In *Proc. 4th Int'l Conf. Computing Frontiers (CF'07)*, pages 197–204, May 2007.
9. X. Yang, X. Yan, Z. Xing, Y. Deng, J. Jiang, J. Du, and Y. Zhang. Fei teng 64 stream processing system: Architecture, compiler, and programming. *IEEE Trans. Parallel and Distributed Systems*, 20(8):1142–1157, Aug. 2009.