The Journal of Systems and Software 117 (2016) 1–14

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

GPU-SAM: Leveraging multi-GPU split-and-merge execution for system-wide real-time support

Wookhyun Han, Hoon Sung Chwa, Hwidong Bae, Hyosu Kim, Insik Shin*

School of Computing,KAIST, Daejeon, South Korea

A R T I C L E I N F O

Article history:
Received 13 June 2015
Revised 20 January 2016
Accepted 8 February 2016
Available online 17 February 2016

Keywords:
Real-time systems
Multi-GPU
GPGPU

A B S T R A C T

Multi-GPUs appear as an attractive platform to speed up data-parallel GPGPU computation. The idea of split-and-merge execution has been introduced to accelerate the parallelism of multiple GPUs even further. However, it has not been explored before how to exploit such an idea for real-time multi-GPU systems properly. This paper presents an open-source real-time multi-GPU scheduling framework, called GPU-SAM, that transparently splits each GPGPU application into smaller computation units and executes them in parallel across multiple GPUs, aiming to satisfy real-time constraints. Multi-GPU split-and-merge execution offers the potential for reducing an overall execution time but at the same time brings various different influences on the schedulability of individual applications. Thereby, we analyze the benefit and cost of split-and-merge execution on multiple GPUs and derive schedulability analysis capturing seemingly conflicting influences. We also propose a GPU parallelism assignment policy that determines the multi-GPU mode of each application from the perspective of system-wide schedulability. Our experiment results show that GPU-SAM is able to improve schedulability in real-time multi-GPU systems by relaxing the restriction of launching a kernel on a single GPU only and choosing better multi-GPU execution modes.

1. Introduction

GPGPU (General-Purpose computation on Graphics Processing Units) offers an effective computing platform to accelerate a wide class of data-parallel computing, including the ones with real-time constraints. For instance, GPGPUs have been used to perform various features for automotive systems, including navigation (Homm et al., 2010), obstacle avoidance (Nordin, 2010), and image processing/filtering for object detection (Nagendra, 2011). In addition, many interactive augmented/virtual reality systems (Oculus, 2012; Sulon, 2014) employ GPGPUs to accelerate image processing (Madsen and Laursen, 2007), real-time stereo (Sizintsev et al., 2010), and ray tracing (Parker et al., 2010) to deliver real-time response. When many of such applications are integrated on a single platform, they may require higher GPU computing power beyond the capacity of a single GPU. The focus of this paper is to support multiple real-time GPGPU applications on multiple GPUs.

Recently, there is a growing interest in exploiting multiple GPUs (Noaje et al., 2010; Stuart et al., 2011; Garcia et al., 2013; Zhou and Fürlinger, 2015), since multi-GPU systems provide a potential for higher performance beyond single-GPU systems. However, the prevailing GPGPU programming models, including Khronos. OpenCL (2015) and CUDA (2015), do not provide a proper abstraction over multiple GPUs. OpenCL is a standard framework for parallel programming of heterogeneous platforms. Lately, many parallel computing platforms including not only GPUs (Du et al., 2012) but also CPUs (Karrenberg and Hack, 2012), DSPs (Li et al., 2012), and FPGAs (Shagrithaya et al., 2013; Czajkowski et al., 2012; Chen and Singh, 2012; Rodriguez-Donate et al., 2015) use OpenCL due to direct portability. In OpenCL programming models, programmers define functions, called *kernels*, that are executed on GPU in parallel by a number of threads, while each single kernel invocation should correspond to a single designated GPU. With such a *single-GPU-per-kernel* restriction, real-time applications may not fully utilize multiple GPUs even in the face of deadline misses. In general, it will impose a great burden on application developers if they are responsible for addressing the underutilization of available multiple GPUs.

To alleviate this problem, several previous studies have introduced the idea of *Split-and-Merge* (S & M) execution of a single kernel across multiple GPUs (Luk et al., 2009; Kim et al., 2011; Lee et al., 2013; Pandit and Govindarajan, 2014). Their proposed systems automatically partition the workload across multiple GPUs, execute the partial workloads in parallel, and merge the partial

* Corresponding author. Tel.: +82423503524.
E-mail address: insik.shin@cs.kaist.ac.kr, insik.shin@gmail.com (I. Shin).

http://dx.doi.org/10.1016/j.jss.2016.02.009
0164-1212/© 2016 Elsevier Inc. All rights reserved.

results into a complete one. Those systems differ from each other in terms of applicability (i.e., the type of kernels to support) and portability (i.e., whether code modification is required). Yet, they share the same goal of performance improvement from a single kernel's point of view. It is important to note that since the S & M execution of one application can influence the execution behavior of other applications in a various manner, simply achieving such a goal can harm the system-wide performance and/or schedulability. However, the issue of exploiting multi-GPU S & M execution properly for real-time systems has not been considered before.

Split-and-merge execution can be beneficial for real-time multi-GPU computing. It offers the potential for decreasing GPU response time by splitting a single kernel into several smaller units, called *sub-kernels*, and executing them in parallel across different GPUs. Moreover, the execution of such smaller sub-kernels helps to reduce the time to block higher-priority applications, which is inevitable from the non-preemptive nature of GPU processing. On the other hand, S & M execution can impose non-trivial overhead, since it requires extra memory transfer of input/output data between host and GPU memory and additional computation of merging the partial outputs. Such overhead can be translated to interfering (or blocking) other applications, thereby hurting their schedulability. Such a tradeoff between benefits and costs of S & M execution varies according to the type of GPGPU applications (i.e., compute- or memory-intensive) as well as the number of GPUs to launch parallel sub-kernels on. This entails a good strategy of determining how applications utilize multi-GPU S & M execution from the perspective of system-wide schedulability.

As such, split-and-merge execution raises many interesting research questions to explore for system-wide real-time support. For example, which scheduling policies are favorable for multi-GPU S & M execution? Unfortunately, the existing works (Luk et al., 2009; Kim et al., 2011; Lee et al., 2013; Pandit and Govindarajan, 2014) for multi-GPU S & M execution do not offer an adequate (open source and accessible) environment for further exploration and experimentation. Open source environments would be particularly useful for exploring such research issues with a low barrier-to-entry.

This motivates our work to develop an open-source real-time multi-GPU scheduling framework, called GPU-SAM[1] (Split And Merge), that orchestrates multi-GPU split-and-merge execution of multiple kernels transparently, aiming to provide real-time guarantees. GPU-SAM features two additional functionalities for system-wide real-time support: real-time scheduling and decision-making on multi-GPU execution. For real-time scheduling, it supports priority-based scheduling of memory transfer and kernel launch requests according to a predefined priority order.

In order to exploit split-and-merge execution more properly under real-time scheduling, we explore the issue of determining the multi-GPU mode of individual applications. We first build a benefit-cost analysis model of S & M execution across a different number of GPUs and construct a schedulability condition with the model. We then present an algorithm, called GPA (GPU Parallelism Assignment), that decides the number of parallel sub-kernels for each application based on the schedulability condition in order to improve system-wide schedulability. A key characteristic of GPA is its conceptual simplicity, while balancing the conflicting effects of S & M execution on system-wide schedulability. This is possible since its underlying schedulability condition captures such conflicting effects and translates them into schedulability terms.

GPA通过可调度性条件，确定子kernel数。

Our experimental and simulation results show that S & M execution can be advantageous for real-time multi-GPU computing and GPA can improve the system-wide schedulability substantially, compared to when decisions are made from an individual applications' viewpoint, by 25–50%.

Contributions. The main contribution of this paper can be summarized as follows.

- To the best of our knowledge, this work makes the first attempt to explore split-and-merge execution over multiple GPUs for real-time support.
- We introduce an algorithm, called GPA, that determines the multi-GPU execution mode of individual GPGPU applications to improve system-wide real-time schedulability. Our evaluation results show that GPA outperforms the existing approaches significantly, while performing comparable to an optimal solution obtained through exhaustive search.
- We present an open-source prototype implementation of GPU-SAM as an extension of OpenCL runtime that offers the state-of-the-art applicability and portability. It supports multi-GPU split-and-merge execution for real-time applications in a seamless manner regardless their data access patterns without any code modification required. We anticipate that with our open-source framework, many other S & M research issues could be explored more easily.

2. Background

This section introduces the basic GPU architecture and OpenCL, the standard GPGPU programming model.

GPU architecture. Modern GPUs consist of thousands of relatively simple processing cores optimized for parallel computing. They are specially tailored to SIMD (single-instruction multiple-data) processing; all threads running on a stream multiprocessor execute the same instruction. To deal with multiple data at once with the same code, GPU threads get access to different memory addresses based on their thread IDs, group IDs, and so forth.

通过TIDs和GroupIDs来获取不同内存地址的数据。

OpenCL programming model. OpenCL is one of the programming models for leveraging such massively parallel architectures. The OpenCL programming model consists of *kernels* and *host programs*. Kernels are a basic unit of non-preemptive executable code, each of which is assigned a single GPU and runs on it in original OpenCL. The host program executes on a CPU and invokes (or enqueues) kernel instances using command queues.

Once the kernel execution is requested by the host program, a N-dimensional workspace (or index space) is defined. Each element in the workspace is called *work-item* which is run by a single thread while executing the same kernel function but on different data. Work-items are partitioned into *work-group*, to be executed together on the same compute unit of a device. Within a work-group, work-items are enforced to share the same view of memory at each synchronization point such as *barrier*. On the other hand, there is no synchronization point for inter-work-groups, except the end of the kernel execution. Such independence among work-groups enables not only their out-of-order execution, but also the distributed execution of a single kernel even on multiple GPUs by splitting a single kernel into multiple sub-kernels. A sub-kernel is recognized as a single kernel in OpenCL runtime, so each sub-kernel must run on a single GPU but it can run on different GPU.

During the kernel execution, threads frequently access memory to get or store data. At this time, since GPUs provide their own device memory with much higher bandwidth than host memory, most applications try to run kernels using the device memory in the following steps: (i) *Upload Copy*: all input data is transferred from the host memory to the device memory through the PCI bus, (ii) *Kernel Launch*: with the copied data, a kernel is executed and stores intermediate output data in the device memory, and (iii)

[1] https://github.com/GPU-SAM

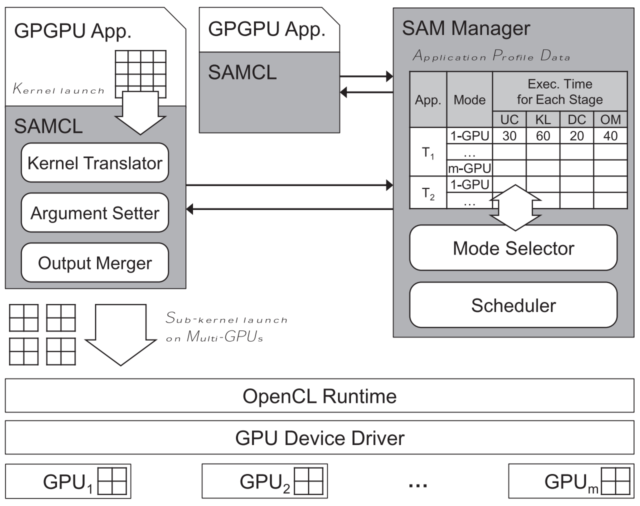
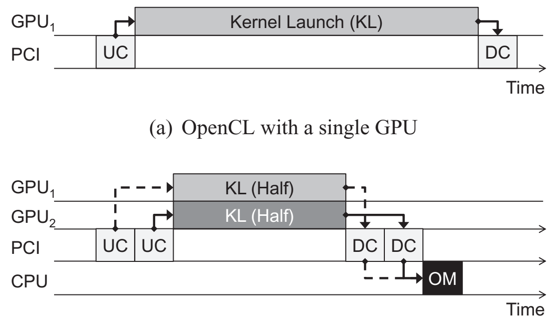


Fig. 1. Logical overview of GPU-SAM.



(a) OpenCL with a single GPU

(b) GPU-SAM

Fig. 2. GPGPU application execution flows: Single-GPU and GPU-SAM.

Download Copy: after the kernel execution is completed, their output data is copied back to the host memory.

3. GPU-SAM framework

The goal of GPU-SAM is to enable split-and-merge execution on multiple GPUs for supporting real-time applications without modification and enhance system-wide schedulability. Fig. 1 illustrates the GPU-SAM architecture with SAMCL and SAM Manager toward this goal. SAMCL is an extension of OpenCL that enables splitting and merging kernels for GPGPU applications. Since SAMCL hooks OpenCL APIs, legacy GPGPU applications are able to transparently work under SAMCL with no modification and run on all GPUs regardless of which GPUs are specified by programmers.

SAM Manager, on the other hand, works for real-time support of GPU-SAM. Whenever a task set in the system is changed, *Mode Selector* in SAM Manager determines how many GPUs each GPGPU application uses (i.e., 1-to-m GPU mode where *m* is the number of GPUs) based on offline profile data so that deadline misses are minimized from a system-wide viewpoint (more details are given in Section 4). In addition, *Scheduler* in SAM Manager enforces priority-based scheduling on both PCI and GPU where applications are scheduled in a FIFO manner by default. Every time a GPGPU application requests for a memory operation or a kernel launch operation, the request is instead sent to GPU-SAM scheduler and later granted based on its priority.

Scheduler决定了占用PCI和GPU的顺序

3.1. Overview of split-and-merge execution

First of all, what we mean by split is duplicating a kernel on multiple GPUs and assigning disjoint workspaces to each duplicated kernel. We call such kernels as *sub-kernels*, which share the same code but use different workspace, where the union of workspaces equals to the original kernel's workspace.

As previously described, a typical scenario of executing an OpenCL application can be viewed as a sequence of upload copy, kernel launch and download copy. When an application is executed with GPU-SAM on multiple GPUs, GPU-SAM supports split-and-merge execution of a kernel in the application as follows. (i) For upload copy (UC), SAMCL makes multi-GPUs ready for hosting sub-kernels by copying input data to each GPU so that all the sub-kernels can run with their own copy of input data in their own GPU memory. (ii) For kernel launch (KL), each sub-kernel is launched on its own GPU for concurrent execution. At this moment, each sub-kernel is subject to run on its own workspace to compute partial results. (iii) For download copy (DC), similar to the upload copy case, SAMCL copies the outputs of individual sub-kernels back to the host memory from their own GPU memory. (iv) SAMCL introduces an additional stage called *Output Merge* (OM), during which it performs merging the partial outputs of sub-kernels to produce a complete result that is the same as the result of single-GPU execution. Fig. 2 compares how a kernel program produces output on a single GPU with original OpenCL, and on two GPUs with SAMCL.

Issues. Implementing such a split-and-merge execution paradigm for real-time applications raises several issues to resolve. For example, sub-kernels should be generated correctly with disjoint workspaces. In addition, partial outputs should be merged maintaining coherence with respect to the single-GPU-per-kernel execution. For real-time support, it would be better to provide priority-based GPU and PCI scheduling despite their FIFO nature. The next subsection discusses those issues in details.

3.2. Implementation issues

Sub-kernels with disjoint workspaces. *Kernel Translator* and *Argument Setter* in SAMCL are two key components to split a kernel. The kernel translator works when kernel program objects are created. As previously mentioned, each work-item (or thread) shares the same kernel program code, but uses different memory addresses for read and write. The memory address for each thread access is then determined through various work-item built-in functions. Since such built-in functions can produce different computation results when a kernel is decomposed into sub-kernels, we need to modify them to maintain consistent computations. To this end, the kernel translator appends the following auxiliary arguments to all the functions in a kernel, including the kernel itself; `GPUSAM_num_groups`, `GPUSAM_group_offset`, and `GPUSAM_global_size`. We note that the return values of work-item built-in functions determine the memory addresses accessed by sub-kernels, and the return values may vary according to the number of sub-kernels determined by the mode selector. Thereby, we append such auxiliary arguments to deal with dynamic environments. Table 1 describes the translation rules for all the work-item built-in functions.

The argument setter then works when individual sub-kernels are launched. The argument setter makes the auxiliary arguments in a way that the workspace of each sub-kernel is exclusive to each other, and the union of those workspaces is equivalent to the workspace of the original kernel. For each sub-kernel, the argument setter assigns the values of `GPUSAM_num_groups` and `GPUSAM_global_size` such that those values are equal to the return values of `get_num_groups()` and `get_global_size()` invoked by the original kernel, respectively. On the other hand, the values of

Table 1
Translation rules for work-item built-in functions.

OpenCL	GPU-SAM
`get_work_dim()`	`get_work_dim()`
`get_global_size()`	`GPUSAM_global_size`
`get_global_id()`	`(get_global_id() + get_local_size() * GPUSAM_group_offset)`
`get_local_size()`	`get_local_size()`
`get_local_id()`	`get_local_id()`
`get_num_groups()`	`GPUSAM_num_groups`
`get_group_id()`	`(get_group_id() + GPUSAM_group_offset)`
`get_global_offset()`	`get_global_offset()`

assigned differently

GPUSAM_group_offset are assigned differently to guarantee disjoint workspaces.

Workload distribution. Though a kernel can be split into sub-kernels which have disjoint workspaces through the kernel translator and the argument setter, there exist problems of determining how much portion of its workload is assigned to each sub-kernel and which dimension of a workspace is divided. In the systems with homogeneous GPUs, SAMCL distributes workload to sub-kernels as equally as possible. In the systems with heterogeneous devices, there exists an additional challenge to balance workload to heterogeneous computing devices for maximizing performance, since the performance of a computing device varies by not only hardware specifications, such as the number of cores, clock frequency of cores, and memory bandwidth, but also application characteristics, such as compute-intensive and memory-intensive. We remain this problem as future work for extending our framework to heterogeneous systems.

Secondly, we need to decide how to divide workspace dimensions. A workspace size of a kernel is represented as up to 3 dimensions (x, y, z), and any of those can be divided to split the workspace into sub-workspaces. For example, a 2D workspace with dimension (4,2) can be split into two sub-workspaces of (2,2) or two sub-workspaces of (4,1). We design a workspace allocation scheme that maximizes the minimum length of dimensions. When running on 4 GPUs, a workspace with dimension (8,8) is split into four sub-workspaces of (4,4), and the argument setter sets four sub-kernels to have offsets of (0,0), (4,0), (0,4), and (4,4). This scheme is driven from the observation in Appendix A.

Equivalent output data. Since we distribute workloads to multiple GPUs, we need to merge partial outputs from each GPU to make a whole. Basically, *Output Merger* in SAMCL keeps the original data whenever duplicating the data across multiple GPUs. In the output merge process, for each element in the original data, the output merger overwrites it with a corresponding one in the result of a sub-kernel if they are different. It is important to note that there is at most one different element, because the kernel translator and the argument setter make disjoint workspaces across sub-kernels and thereby at most one sub-kernel updates the element. In this way, SAMCL is able to deal with even the case of irregular memory access patterns. More details about data coherency and correctness of SAMCL are described in Appendix B.

Inter-kernel dependencies. Applications may have multiple kernels under *inter-kernel dependency* in a way that one kernel takes as input the output data of a previous kernel. In order to maintain the coherence of the final output, GPU-SAM enforces buffer synchronization between host and device memory right before each kernel launch. To make it efficient, GPU-SAM does it on demand for the kernels under dependency. More details are provided in Appendix B.

Prioritized resource scheduling. While real-time applications make use of CPU, GPU, and PCI resources for GPGPU computing, GPU and PCI resources are allocated under FIFO scheduling by default. Thus, GPU-SAM enables priority-based scheduling on PCI and GPU for the sake of better real-time support. GPU-SAM scheduler in SAM Manager is another Linux process and creates individual priority queues for PCI and GPUs, respectively. When an application requests either a memory transfer or kernel launch operation to the OpenCL runtime, SAMCL hooks the request and sends it to GPU-SAM scheduler through POSIX message queues. The request is then enqueued to a corresponding resource queue if the resource is busy, or immediately granted otherwise. Upon receiving a signal from SAMCL indicating its application finishing the use of a resource, GPU-SAM scheduler dequeues the highest priority request from the corresponding queue. In a GPU-SAM prototype, fixed-priority scheduling is employed, and GPU-SAM scheduler is running on the highest priority of Linux scheduler.

Limitations. All existing multi-GPU split-and-merge systems (Luk et al., 2009; Kim et al., 2011; Lee et al., 2013; Pandit and Govindarajan, 2014), including GPU-SAM, share one restriction in applicability. They may not work in a case where applications use *global barriers* or *atomic operations*[2] to coordinate between global work-items. Since the current GPGPU programming models do not support synchronization primitives between multiple GPUs, it seems quite difficult to resolve this issue efficiently.

One limitation of GPU-SAM in supporting priority-based PCI scheduling comes into account when kernels make accesses to data in host memory directly. It is worth noting that since GPU device memory offers an order of magnitude higher bandwidth than host memory does (NVIDIA GTC, 2014), it is suggested to perform GPGPU computation over data in the device memory, rather than data in the host memory, for high performance. Thereby, such a limitation is not so problematic in practice.

4. Split-and-merge execution for system-wide real-time support

In the previous section, we introduced GPU-SAM framework that facilitates split-and-merge execution for real-time multi-GPU systems. This section explores the issues involved in leveraging GPU-SAM for system-wide schedulability improvement. When applications determine GPU execution modes from their individual points of view, as is done by many existing studies (Luk et al., 2009; Kim et al., 2011; Lee et al., 2013; Pandit and Govindarajan, 2014), it may not lead to system-wide optimal decision. This requires to consider a good policy of system-wide GPU execution mode assignment in order to improve the overall schedulability performance.

To this end, we first investigate the effects of split-and-merge execution (i) on each single application and (ii) on the interaction between applications from a schedulability viewpoint. We measure benefits and overheads of S & M execution through experiments with various OpenCL benchmarks and build an application-level benefit-cost analysis model based on the benchmark results. We then present end-to-end response time analysis that properly captures seemingly conflicting effects of S & M execution on system-wide schedulability. Finally, building upon the benefit-cost model and end-to-end response time analysis, we develop an efficient GPU execution mode assignment algorithm which decides

[2] Atomic operations (i.e., atomicAdd) can be used to coordinate between the work-items distributed across work-groups to avoid race conditions. However, the atomic operations are often subject to performance concerns due to their serialization nature, potentially leading to a substantial loss of parallelism and thus a great loss in performance. Thereby, for example, it is often suggested to use a *reduction* method rather than an atomicAdd to fully exploit the massive parallelism of GPU and thus reduce the complexity to $O(\ln n)$ from $O(n)$, where n is the number of work-items.

通过分析效益-成本和端到端响应时间分析，提出了高效的GPU执行模型分配算法。

the number of sub-kernels from the perspective of system-wide schedulability.

4.1. *Benefit and cost analysis*

GPU-SAM has a great potential for real-time multi-GPU scheduling by reducing overall execution time through split-and-merge execution. However, it does not come for free. S & M execution could impose some non-trivial overheads, which mainly come from extra memory transfer in upload/download copy and additional computation for output merge. This subsection first investigates the benefits and overheads of S & M execution from an individual application perspective, and shows the results of running GPU-SAM on various types of GPGPU applications. For simplicity, all the multi-GPU mode executions are run on two GPUs.

Benefits/overheads of split-and-merge execution. We compare GPU-SAM on two GPUs (dual-GPU mode) and the original OpenCL on a single GPU (single-GPU mode).[3] The result is illustrated in Fig. 3, describing the benefits and overheads in each stage of an OpenCL application execution sequence.

Fig. 3 (a) shows that in dual-GPU mode, upload copy takes about twice time as much as that of single-GPU mode. This is because split-and-merge requires extra memory transfer, from the host memory to the device memory in another GPU. Since data copy time is proportionate to the data size, the upload copy overhead can be modeled through linear regression.

Fig. 3 (b) illustrates the benefit of split-and-merge execution by plotting the execution time of a kernel for matrix multiplication over different input data sizes. It shows that the execution time of a sub-kernel is a half of that of its original kernel. This also shows that the overhead of launching multiple sub-kernels is negligible.

As described in Fig. 3(c), the overhead for download copy is very similar to the one of upload copy, and it can also be modeled through linear regression. Since each sub-kernel generates a partial output, the outputs of sub-kernels should be merged after download copy. The time for output merge depends on the size of output data, increasing in proportion to the output data size. This output merge time can be easily reduced through parallelization APIs such as POSIX threads and OpenMP, as is the case with GPU-SAM implementation. Note that the extra data copy and output merge times are proportional to the number of GPUs each, while the execution time of a kernel is inversely proportional to the number of GPUs.

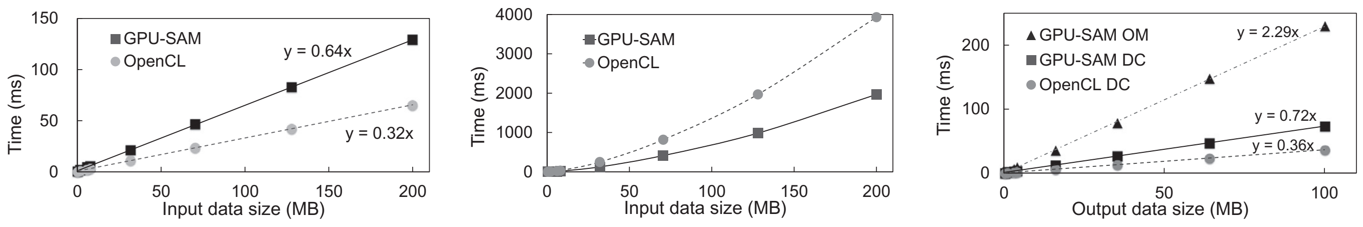
Benchmark results and correctness. By investigating how split-and-merge affects the execution time of each stage, we examined the potential benefits and overheads of GPU-SAM. It is then necessary to see if it is beneficial to run various GPGPU applications concurrently despite some overheads. To this end, we performed experiments with some representative benchmarks which can be used in many real-time applications (see Table 4). Table 2 describes the specifications for these benchmarks, including input/output data type, data size, buffer size, the number of workgroups, and whether the benchmark consists of inter-dependent kernels.

Fig. 4 illustrates the normalized execution times of benchmarks, comparing the original OpenCL in single-GPU mode and GPU-SAM in dual-GPU mode, and Table 3 shows the execution times of benchmarks in milliseconds. For presentational convenience, we define the execution time of a kernel as the time required to complete upload copy, kernel launch, download copy, and output merge. As described in the figure, most benchmarks (except Histogram and Pagerank) are GPU compute-intensive, spending

Table 2
Benchmark specifications.

Benchmark	Source	Input			Output			WorkGroups	Kernel Dependency	Correctness
		Type	# of data	Total Size	Type	# of data	Total Size			
AESEncrypt/Decrypt	AMDSDK	Bitmap image	512 × 512	256KB	Bitmap image	512 × 512	256KB	256	X	O
BinomialOption	AMDSDK	# of stock price	16384	256KB	# of FP numbers	16384	256KB	255	X	O
Filter	CUSTOM	Bitmap image	1024 × 768	768KB	Bitmap image	1024 × 768	768KB	768	X	O
Histogram	AMDSDK	# of 8-bits	16 millions	16MB	Integers	256	1KB	512	X	O
K-nearest	CUSTOM	# of FP numbers	1024 × 32	128KB	# of FP numbers	1024 × 320	1280KB	128	X	O
MatrixMultiplication	NVIDIA SDK	Matrix size	2048 × 2048	32MB	Matrix size	2048 × 2048	16MB	4096	X	O
Pagerank	CUSTOM	# of FP numbers	4 millions	16MB	# of FP numbers	4 millions	16MB	4096	O	O

[3] Both experiments of Figs. 3 and 4 are conducted on two NVIDIA GeForce GTX 560Ti GPUs and Intel Core i5-3550 CPU with NVIDIA proprietary driver Version 331.49 and OpenCL Version 1.1. More details are in Section 5.



(a) The overhead of upload copy (b) The execution time of a kernel for matrix multiplication (c) The overhead of download copy and output merge

Fig. 3. The benefits and overheads of multi-GPU mode execution.

Table 3
The execution time of seven benchmarks on OpenCL in single-GPU mode and GPU-SAM in dual-GPU mode.

Benchmark		UC	KL	DC	OM	Total
AESEncrypt	OpenCL	4.09	69.11	4.83	0	78.02
	GPU-SAM	9.96	42.91	9.04	10.88	78.79
AESDecrypt	OpenCL	4.09	209.50	4.61	0	218.20
	GPU-SAM	9.97	112.88	9.07	8.18	140.11
Binomial	OpenCL	0.72	73.40	0.83	0	74.65
Option	GPU-SAM	1.39	39.58	0.89	0.60	42.46
Filter	OpenCL	0.31	21.82	0.34	0	22.48
	GPU-SAM	0.74	13.21	0.99	0.51	15.45
Histogram	OpenCL	20.87	16.67	0.24	0	37.78
	GPU-SAM	54.23	10.00	0.95	0.40	65.59
K-nearest	OpenCL	0.80	111.97	0.49	0	113.25
	GPU-SAM	2.90	58.69	2.83	2.20	66.63
Matrix Mul-	OpenCL	10.97	249.74	6.10	0	266.81
tiplication	GPU-SAM	26.05	149.19	13.52	15.20	203.97
Pagerank	OpenCL	63.17	838.05	5.17	0	906.39
	GPU-SAM	297.63	419.66	60.36	65.80	843.46

Table 4
Explanation of benchmarks.

AES Encrypt/Decrypt	Encryption/decryption algorithm established by Advanced Encryption Standard (AES), applicable to secure real-time transport protocols.
Binomial Option	A numerical algorithm for valuation of options.
Filter	A data processing technique, applicable to real-time image processing applications such as augmented reality.
Histogram	A graphical representation technique for the distribution of data, applicable to various statistical purposes and real-time image processing such as histogram equalization.
K-nearest	K-Nearest Neighbors algorithm, applicable for a wide range of real-time machine learning applications, including face/voice/gesture recognition.
Matrix Mul.	Matrix multiplication, a typical GPU-favorable operation for a wide range of applications (i.e., real-time data processing techniques).
Pagerank	A link analysis algorithm used by Google's search engine, applicable for network/context analysis.

88.5–99.8% of the execution time on the kernel launch in single-GPU mode. These are common examples that gain a lot of benefits from largely decreased kernel launch time. We note that in some benchmarks (i.e., AES Encrypt/Decrypt, Filter, and Histogram), the upload buffer is used as an output buffer. In this case, it is necessary to duplicate the buffer on the host memory to update partial outputs from GPUs to the output buffer correctly. The cost of this additional duplication is about 0.5 times of original upload copy in our environment. Although S & M execution increases upload/download copy time by a factor of about 2.5 with additional buffer duplication and adds an extra output merge time, it is able to save 6.8–43.2% of the total execution time through parallel execution on two GPUs.

On the other hand, Histogram is one of the typical memory-intensive applications that can hardly benefit from split-and-merge execution. Histogram takes up 44.1% of the execution time on only upload copy in single-GPU mode. Such memory-intensive nature of Histogram makes GPU-SAM suffer from 73.6% increase of total execution time, even though kernel launch time is reduced by half.

Last but not least, Pagerank shows a different behavior due to its inter-kernel dependency. Since it launches multiple kernels which require an input from the output buffer of the former kernel, extra output merge and buffer synchronization occur. In our application, pagerank has five kernels and consecutive kernels have inter-kernel data dependency. This cause the upload copy and download copy to be increased by 4.7 times and 11.7 times respectively, while kernel launch time remains half. Although total execution time is reduced by 7.0% in this case, a little increase in the input data size may result in a huge loss in memory transfer time, making it inappropriate for GPU-SAM. Thereby, in order to apply GPU-SAM in practice, we need to carefully consider the characteristics of applications, such as memory-intensive nature and inter-kernel dependency.

In Appendix C, we show more benchmark specifications under various application characteristics including inter-kernel dependency. Furthermore, it is worth mentioning that all the benchmarks (except only two benchmarks using atomic operations) produced semantically equivalent results to single-GPU mode execution, which is shown in the last column of Table C.1.

Impact of problem size. In general, GPU achieves high throughput by exploiting a million of threads to hide latency. However, if the number of threads is small so that the threads in a kernel cannot fully utilize the GPU throughput decreases significantly. Therefore, when we split a kernel into two sub-kernels and each sub-kernel cannot fully utilize a GPU, the sub-kernel launch time would take more than a half of the kernel launch time of the original kernel. Similarly, if the size of upload copy of an application is too small to fully utilize the PCI bus, the upload copy time of dual-GPU mode would be less than twice of that of single-GPU mode. In addition, most GPGPU applications have the largest portion of kernel launch time. Therefore, as the problem size increases GPU-SAM can have larger benefits from S & M execution.

Fig. 5 shows the normalized execution times of square matrix multiplication with different problem sizes, comparing the original OpenCL in single-GPU mode and GPU-SAM in dual-GPU mode. As problem size increases the S & Mexecution can have larger benefits. For the case of the matrix size of 256 $\times$ 256, the kernel launch time is reduced by 43% in dual-GPU mode, and the upload/download copy time is increased by 77%. Thus, the total execution time is increased by 48%. For the case of matrix size 512 $\times$ 512, the kernel launch time is reduced by 47% in dual-GPU mode and the total execution time is increased by 4%. In the final case, 1024 $\times$ 1024 matrix, kernel launch time is reduced by 50%, and the total execution time is reduced by 25%. From the experiments, we observe that larger problem size can have larger benefit from S & M execution.

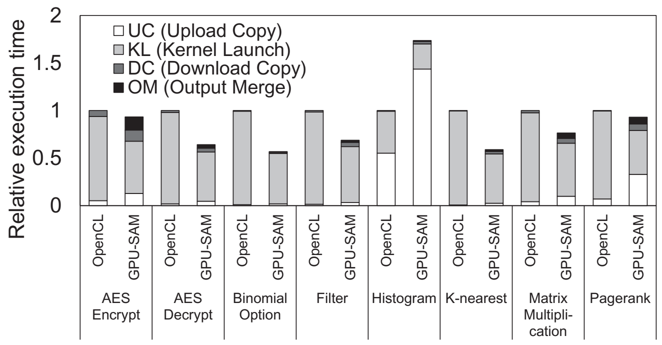


Fig. 4. The relative execution time of seven benchmarks on OpenCL in single-GPU mode and GPU-SAM in dual-GPU mode.

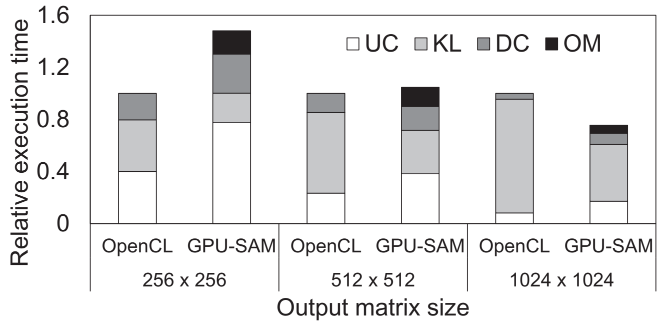


Fig. 5. The relative execution times of square matrix multiplication with different problem sizes on OpenCL in single-GPU mode and GPU-SAM in dual-GPU mode.

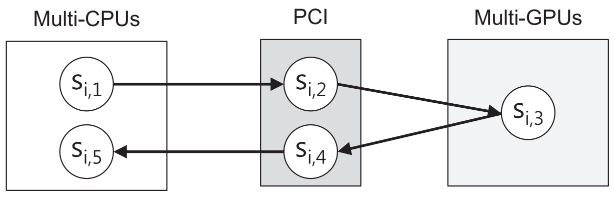


Fig. 6. Pipeline task model.

4.2. Schedulability analysis

In this subsection, we examine the influence of split-and-merge execution on the response time of other applications and present end-to-end response time analysis that properly captures such an influence across various resources, which serves as the basis of our GPU parallelism assignment algorithm.

System model. GPGPU applications utilize CPU and GPU for computation and the PCI bus for data transfer between the host and device memory. Capturing this, we model each GPGPU application as a pipeline task, as shown in Fig. 6, as follows. A task τ_i invokes a series of jobs sporadically with a minimum separation delay of T_i, and it has an end-to-end deadline of D_i ($D_i \le T_i$). Every task τ_i consists of n_i stages, $s_{i,1}, \ldots, s_{i,n_i}$, in each of which τ_i has one or more subtasks to run on a specific resource $w_{i,j} \in$ {CPU, PCI, GPU}. Each stage $s_{i,j}$ is sequentially activated upon the completion of a preceding stage $s_{i,j-1}$. Thus, no stages within a single task can overlap each other. Depending on the resource type $w_{i,j}$ of stage $s_{i,j}$, the subtasks of τ_i can be characterized:

(i) if $w_{i,j}$ = CPU, one preemptive subtask runs in $s_{i,j}$,
(ii) if $w_{i,j}$ = PCI, one non-preemptive subtask runs, and
(iii) if $w_{i,j}$ = GPU, m_i non-preemptive subtasks execute in parallel with each other, where $1 \le m_i \le M_G$. We note that m_i indicates the number of sub-kernels which can be concurrently executed on GPUs, and m_i will be chosen between 1 and M_G, where M_G denotes the number of available GPUs, by our proposed GPU parallelism assignment algorithm.

We note that each subtask has a set of different execution time requirements $\{C_{i,j}^{(1)}, \ldots, C_{i,j}^{(M_G)}\}$ depending on how many GPUs τ_i utilizes. Then, $C_{i,j}^{(m_i)}$ indicates the execution time requirement of each subtask in stage $s_{i,j}$ when τ_i is subject to m_i-GPU execution mode. We assume that the execution time requirement for each GPU execution mode is given by offline profile data as shown in Fig. 1.

We consider GPU-SAM employs a fixed-priority scheduling algorithm. A given task τ_i has the same priority across all resources, and it must complete execution of all subtasks prior to its end-to-end deadline D_i.

It is important to note that we model both PCI data transfer and GPU execution as single-resource activities while they actually involve another resource (CPU) for initiation. Data transfer between the host and device memory is operated by DMA (Direct Memory Access) engine, while initiated by some CPU-side operations (i.e., OpenCL upload/download copy APIs) including memory mapping between userspace and OS kernel space. GPU-SAM assigns a highest priority to such CPU operations in order to introduce no additional delay, though it takes relatively short time to complete the CPU operations. For example, in our measurements, it takes less than 0.2ms for 30MB DMA transfer, translating to 2% of the entire data transfer time of about 10ms. Such a CPU time can be added into PCI memory transfer time. We note that CPU-side operation for DMA transfer can preempt and delay other CPU subtasks. Such delay can be added to the execution times of other CPU tasks efficiently. For instance, for 30MB DMA transfer, the execution times of subtasks can increase by 0.2ms * α, where α indicates the maximum number of DMA transfers to overlap (i.e., $\alpha = \lceil T_i / 10\text{ms} \rceil$). A similar reasoning can be applied to the case of GPU execution, while the time to execute some CPU-side operations for GPU kernel launch is typically much shorter (i.e., 15µs).

End-to-end response time analysis. We apply holistic analysis to upper-bound the end-to-end delay of each task. The holistic analysis was first introduced in Tindell and Clark (1994), and many extensions were proposed in Palencia and Harbour (1998, 1999, 2003); Pellizzoni and Lipari (2005). The analysis shows that the worst-case end-to-end response time R_i of a pipeline task τ_i can be derived by summing up the maximum delays that individual stages $s_{i,j}$ experience, that is,

$$R_i = \sum_{\forall j} r_{i,j}.$$

The analysis basically employs the concept of release jitter to deal with the precedence constraints between stages. The release jitter $J_{i,j}$ of each stage $s_{i,j}$ is defined as the difference between the worst-case and best-case release times and is presented as

$$J_{i,j} = \sum_{q=1}^{j-1} r_{i,q} - \sum_{k=1}^{j-1} C_{i,k}^{(m_i)}$$

where $J_{i,1}$ is equal to 0. The use of release jitter allows to calculate the local response time at each stage independently of other stages in the same task, but by involving interference with other task's subtasks running on the same resource only.

To compute local response time ($r_{i,j}$), we use a popular response time analysis (RTA) (Bertogna and Cirinei, 2007; Lee and Shin, 2014) and incorporate release jitter into interference calculation. The local response time $r_{i,j}$ can be calculated iteratively in the following expression:

$$r_{i,j}^{a+1} \leftarrow C_{i,j}^{(m_i)} + I_{i,j}^{a} + B_{i,j}, \tag{1}$$

where $I_{i,j}^{a}$ is the maximum interference of higher priority subtasks on a subtask in $s_{i,\,j}$, and $B_{i,\,j}$ is the maximum blocking time from lower priority subtasks. The iteration starts with $r_{i,j}^{0} = C_{i,j}^{(m_i)}$ and continues until it converges or reaches a predefined value. The interference $I_{i,j}^{a}$ is upper-bounded as follows, depending on the type of resources,

$$I_{i,j}^{a} = \begin{cases} \left\lceil \frac{1}{N(w_{i,j})} \cdot X_{i,j}^{a} \right\rceil & \text{if } w_{i,j} = \text{ CPU or PCI }, \\ \left\lceil \frac{1}{N(w_{i,j})} \cdot (X_{i,j}^{a} + Y_{i,j}) \right\rceil & \text{if } w_{i,j} = \text{ GPU }, \end{cases}$$

where

$$X_{i,j}^{a} = \sum_{\tau_k \in hp(\tau_i)} \sum_{p: w_{k,p} = w_{i,j}} \left\lceil \frac{J_{k,p} + r_{i,j}^{a}}{T_k} \right\rceil \cdot C_{k,p}^{(m_k)} \cdot |\tau_{k,p}|,$$

$$Y_{i,j} = C_{i,j}^{(m_i)} \cdot (|\tau_{i,j}| - 1),$$

while $N(w_{i,\,j})$ is the number of processing elements in resource $w_{i,\,j}$, $hp(\tau_i)$ is the set of higher priority tasks than τ_i, and $|\tau_{k,\,p}|$ is the number of subtasks in stage $s_{k,\,p}$. In the cases of CPU and PCI resources, the interference of higher priority subtasks on a subtask in $s_{i,\,j}$ is upper-bounded by using $X_{i,j}^{a}$ as similarly shown in Tindell and Clark (1994), where $X_{i,j}^{a}$ describes the sum of the maximum workload of all subtasks having a higher priority than τ_i running on resource $w_{i,\,j}$ in any interval of length $r_{i,j}^{a}$. In the case of GPU resource, it is worth noting that m_i subtasks run concurrently under global non-preemptive scheduling of M_G GPUs, where $1 \le m_i \le M_G$. This requires to consider a case where m_i subtasks interfere with each other running on the same GPUs, in particular, in the presence of other higher-priority tasks running on other GPUs. The term $Y_{i,\,j}$ captures such intra-task interference (Chwa et al., 2013) by including the workload of other $(m_i - 1)$ subtasks.

The blocking time $B_{i,\,j}$ is calculated as follows:

$$B_{i,j} = \begin{cases} 0 & \text{if } w_{i,j} = \text{ CPU }, \\ \left\lceil \frac{1}{N(w_{i,j})} \cdot Z_{i,j} \right\rceil & \text{if } w_{i,j} = \text{ PCI or GPU }, \end{cases}$$

where

$$Z_{i,j} = \sum_{\tau_k \in lp(\tau_i)} \sum_{p: w_{k,p} = w_{i,j}} \sum_{N(w_{i,j}) \text{ largest subtasks } \in \tau_{k,p}} (C_{k,p}^{(m_k)} - 1),$$

while $lp(\tau_i)$ is the set of lower priority tasks than τ_i. Since a subtask running on CPU resource preempts lower priority subtasks at any time, $B_{i,\,j}$ is zero. On the other hands, the execution of a subtask running on PCI or GPU can be blocked by currently running lower priority subtasks. The maximum blocking time $B_{i,\,j}$ from lower priority subtasks is upper-bounded by choosing $N(w_{i,\,j})$ subtasks which have the largest execution time requirements (Lee and Shin, 2014).[4]

Discussion. We discuss how the analysis captures the conflicting influences of split-and-merge execution. Such influences on an individual task can broadly fall into three categories. (i) For a task τ_i that does S & M execution, the effect is directly reflected on its execution time $C_{i,j}^{(m_i)}$ on each resource in a way that it decreases GPU execution time but increases CPU and PCI execution times. (ii) For higher priority tasks than τ_i, if τ_i is split into more sub-kernels having a smaller non-preemptive region, the blocking time from τ_i decreases. The blocking time from τ_i is captured by the third term (B) in Eq. (1) when calculating the response time of a higher priority task. Thus, S & M execution of τ_i can decrease the response time of higher priority tasks. (iii) For lower priority tasks than τ_i, splitting into more sub-kernels of τ_i imposes more interference on lower priority tasks due to additional overheads (i.e., memory copy and merge operations) on the PCI and CPU resources. The interference from τ_i is captured by the second term (I) in Eq. (1) when calculating the response time of a lower priority task. Unlike the higher priority task case, S & M execution of τ_i can increase the response time of lower priority tasks.

[4] We note that our response time analysis can be tightened by applying the state-of-the-art analysis techniques, such as min techniques (Bertogna and Cirinei, 2007), *limited carry-in* techniques (Guan et al., 2009), and *effective problem windows* for non-preemptive tasks (Lee and Shin, 2014). Due to space limit, however, we leave it to an extended version.

4.3. GPU parallelism assignment

We consider the *GPU parallelism assignment* problem that, given a task set τ, determines the number of sub-kernels for every task $\tau_i \in \tau$ such that the task set is deemed schedulable according to the end-to-end response time analysis presented in Section 4.2.

From a system-wide schedulability viewpoint, if there exists a task τ_i to miss a deadline (i.e., $R_i > D_i$) with a certain configuration, the GPU parallel mode of each individual task should be reconsidered in a direction to make the task schedulable. As discussed in the previous subsection, there are three options to reduce the response time of a task: (i) changing its number of sub-kernels, (ii) decreasing the number of sub-kernels of higher priority tasks, or (iii) increasing the number of sub-kernels of lower priority tasks. However, each option can cause a domino effect on the other tasks, while may be harming the others' schedulability. Thereby, this makes it difficult to find an optimal configuration on the number of sub-kernels for all tasks in a reasonable time. Fortunately, our proposed analysis easily identifies the system-wide schedulability and closely summarizes how the conflicting effects of split-and-merge execution work on the response time of a task. Our schedulability analysis is then able to provide a guideline to solve the GPU parallelism assignment problem.

We present a heuristic algorithm, called GPA *(GPU Parallelism Assignment)*, that determines the number of sub-kernels of all individual tasks based on our schedulability analysis in polynomial time. As shown in Algorithm 1, GPA divides the task set τ into two disjoint subsets: $\mathcal{S}$ and $\mathcal{R}$, where $\mathcal{S}$ is a subset of tasks whose m_i has been assigned, and $\mathcal{R}$ is a subset of remaining tasks whose m_i must be assigned.

The general idea is that GPA gradually increases, through a finite number of iterative steps, the potential to make the system deemed schedulable (i.e., $R_i \le D_i$ for $\forall \tau_i$). In the beginning, the number of sub-kernels for each task is initialized to the one that shows the best performance when each task runs alone. We denote by $\widehat{m_i}$ the initial configuration of task τ_i (line 3). Then, in each iteration (lines 4–24), GPA seeks to minimize the maximum value of R_i/D_i among all tasks τ_i by changing the number of sub-kernels of a single task (τ_i^*) from 1 to M_G. GPA repeats this process until the system becomes deemed schedulable or all tasks are determined. Thereby, GPA invokes the end-to-end response time analysis $O(M_G \cdot n^3)$ times, where n is the number of tasks.

5. Experimental andsimulation results

We performed experimental and simulation studies to see how much split-and-merge execution is beneficial for real-time multi-GPU systems and how well the proposed algorithm, GPA, can improve the benefit further.

For performance evaluation, we considered four schemes for multi-GPU split-and-merge execution: Single-GPU, INDIVIDUAL, OPTIMAL, and GPA. Single-GPU does not exploit S & M execution but allows each single kernel to run on a single GPU only. On the other hand, the other three schemes employ S & Mexecution, while differing from each other in terms of the policies for determining the number of GPUs to assign to each task. INDIVIDUAL

Algorithm 1 GPA - GPU Parallelism Assignment.

```
1: $\mathcal{S} \leftarrow \phi$
2: $\mathcal{R} \leftarrow \{\tau_1 \ldots \tau_n\}$
3: Initialize the number of sub-kernels $m_i$ of each task in $\mathcal{R}$ to $\widehat{m_i}$
4: while $\mathcal{R}$ is not empty do
5:   if $\mathcal{R} \cup \mathcal{S}$ is schedulable then
6:     return success
7:   end if
8:   for each candidate $\tau_i \in \mathcal{R}$ do
9:     for each possible number of sub-kernels $k$ from 1 to $M_G$ do
10:      change $m_i$ to $k$
11:      for each candidate $\tau_j \in \mathcal{R} \cup \mathcal{S}$ do
12:        $R_j$ = End-to-End Response Time Analysis ($\tau_j$)
13:      end for
14:      $Z_{ik} \leftarrow \max_{\tau_j \in \mathcal{R} \cup \mathcal{S}} R_j / D_j$
15:    end for
16:    change $m_i$ to $\widehat{m_i}$
17:  end for
18:  $Z^* \leftarrow \min_{\tau_i \in \mathcal{R}} Z_{ik}$
19:  $\tau_i^* \leftarrow$ the task that has $Z^*$ value
20:  $k^* \leftarrow$ the number of sub-kernels when $\tau_i^*$ has $Z^*$ value
21:  change the number of sub-kernels of $\tau_i^*$ to $k^*$
22:  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\tau_i^*\}$
23:  $\mathcal{R} \leftarrow \mathcal{R} \backslash \{\tau_i^*\}$
24: end while
25: if $\mathcal{S}$ is schedulable then
26:   return success
27: end if
28: return fail
```

makes locally optimal decisions from an individual task perspective in order to minimize the WCET (worst-case execution time) of each individual task. OPTIMAL finds a globally optimal assignment through exhaustive search, exploring an exponential search space of $O(M_G^n)$, where n is the number of tasks. GPA follows the algorithm described in Algorithm 1.

5.1. Experimental results

We conducted experiments with the GPU-SAM prototype system to show the need of GPA and to measure scheduling overhead.

Experimental Setup. Our experiment was conducted on a machine that has two NVIDIA GeForce GTX 560Ti GPUs with PCIe 2.0 x 16 and Intel Core i5-3550 CPU. The Linux kernel Version 3.5.0-27, NVIDIA proprietary driver Version 331.49, and OpenCL Version 1.1 were used. The GPU has 8 compute units and the CPU has 4 cores. GPU-SAM uses 4 CPU threads to merge output buffers. We note that although our experiment was taken a number of times to obtain WCET, measuring the exact WCET is not in the scope of this paper.

Preemption size and priority. For better system-wide schedulability, it is necessary to reduce non-preemptive regions on important resources (i.e., PCI and GPU) despite some overheads. Chunking the input/output data and transfer them in a finer-grained way is a well-known strategy to reduce PCI non-preemptive regions (Kim et al., 2011). We decide data chunk size as 1MB, which introduces a negligible PCI overhead of less than 1%.

Applications are scheduled as real-time processes by Linux kernel, using the SCHED_FIFO scheduling policy. SCHED_FIFO uses a fixed priority between 1 (lowest) and 99 (highest). We use the`sched_setscheduler` function to set priority and scheduling policy of applications.

Table 5
Application specifications.

Index	Application	Priority	Period	Exec. Time	GPU Exec. Time
P_1	Matrix Mul.	High	400 ms	340 ms	320 ms
P_2	Image Filter	Medium	80 ms	50 ms	48 ms
P_3	K-nearest	Low	500 ms	80 ms	77 ms

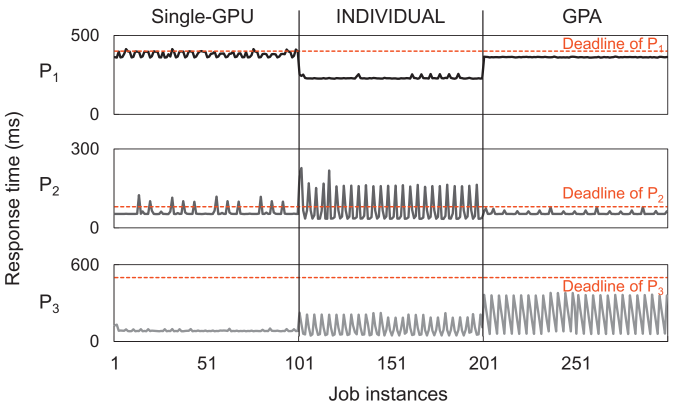
System-wide schedulability. Table 5 describes the application set used in this experiment. In practice, matrix multiplications have been widely applied to real-time signal processing (e.g., Fast Fourier Transformation using matrix multiplications), and image filtering is one of the popular methods for real-time object detection. A k-nearest neighbors algorithm is often employed for real-time object recognition. We consider an implicit-deadline application set such that the deadline of each application is equal to a period. The periods of $P1$ and $P2$ are arbitrarily set, and the period of $P3$ is determined not to exceed the response time of P3 according to end-to-end response time analysis. Jobs continue to execute regardless of their deadline misses. When a job which misses deadline is finished, a new period of the next job starts immediately. We measured the execution time of each application, including upload/download copy and kernel launch, when it runs on a vanilla OpenCL runtime.

Fig. 7 (a) plots the response time of each job instance of the applications under Single-GPU, INDIVIDUAL, and GPA. Response times are shown to fluctuate due to non-preemptive regions and different periods of the applications. Fig. 7(b) breaks down the execution times of job instances, into three resources (CPU, GPU, and PCI) when running alone in single-GPU mode and dual-GPU mode. The total execution times are normalized to the vanilla OpenCL case. In single-GPU mode, GPU-SAM does upload copy to both GPUs to assign a job to an available GPU immediately for GPU load balancing purpose.

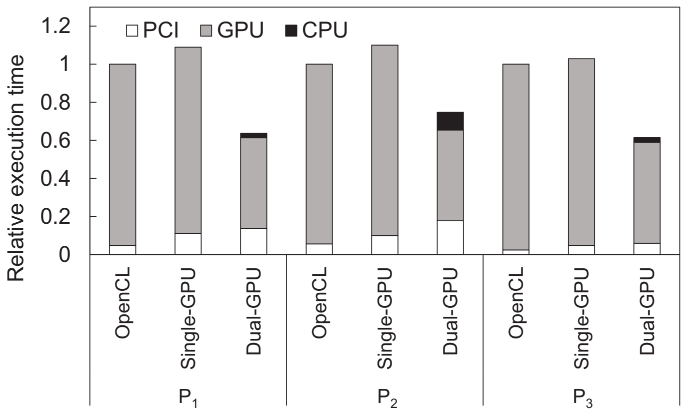
(i) Single-GPU. When all three applications run in Single-GPU mode on two GPUs, the two highest priority applications, P_1 and P_2, are free from GPU interference but are subject to only the blocking effect of their lower-priority application, P_3. Due to the non-preemptive nature of GPU processing, such a blocking effect can delay the executions of P_1 and P_2 by up to the GPU execution time of P_3, which is 77ms. In such cases, P_1 and P_2 can miss deadlines as shown in Fig. 7(a).

(ii) INDIVIDUAL. In order to investigate the influence of split-and-merge execution, we perform another experiment with all three applications running in dual-GPU mode. Here, Fig. 7(b) shows that each application has a great potential to reduce response time through parallel GPU execution. As shown in Fig. 7(a), P_1 benefits much from S & Mexecution mainly from reducing the GPU execution time of its sub-kernel by half. It thereby no longer misses a deadline. On the other hand, P_2 still suffers from deadline misses although it runs via S & Mexecution on two GPUs. This is largely because the S & Mexecution of P_1 can impose interference on P_2 as much as the GPU execution time of P_1's sub-kernel, which is 160ms. This illustrates that the S & Mexecution of one application can help to satisfy its own timing constraint but threatens the schedulability of its lower-priority application.

(iii) GPA. In this example, GPA determines that P_1 and P_2 run in single-GPU mode, but P_3 does in dual-GPU mode. In this case, as explained before, P_1 and P_2 are free from GPU interference on two GPUs, and the amount of blocking time from P_3 reduces by half through the split-and-merge execution of P_3. Fig. 7(a) shows that P_1 and P_2 are then able to meet



(a) Response time



(b) Job specification

Fig. 7. The response times of job instances and job specifications.

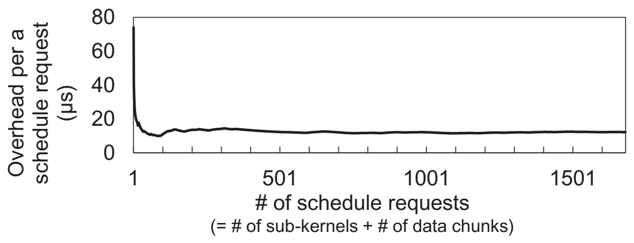


Fig. 8. GPU-SAM scheduler overhead.

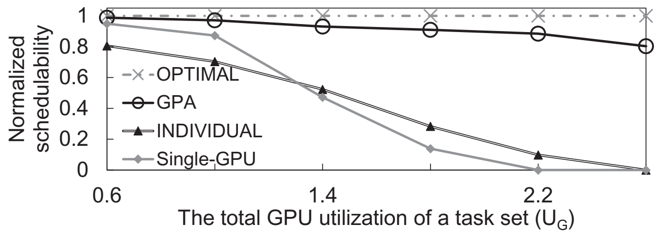


Fig. 9. Schedulability, normalized to global optimum algorithm, under different total GPU utilizations of a task set.

deadlines, while P_3 experiences longer response times, compared to the Single-GPU mode, but still meets its deadline. This example shows that the S & Mexecution of one application can improve the schedulability of higher-priority applications by reducing its blocking time. It also indicates that all the applications can satisfy their deadlines when their GPU parallelism modes are properly determined.

GPU-SAM **scheduler overhead.** We have measured the overhead of GPU-SAM scheduler for scheduling and communicating with SAMCL. The scheduling overhead of the scheduler is measured by the time taken for priority queue management. Fig. 8 shows the average scheduling overhead of GPU-SAM scheduler per a schedule request. The x-axis represents a different number of total schedule requests, and the y-axis represents the average CPU time used by the scheduler in handing a request. The figure indicates that the overhead is almost linear to the number of schedule requests; the scheduling overhead per a request of GPU-SAM is about 12μs.

The communication overhead is measured by the sum of elapsed time, from sending a request to successfully receiving it. The average communication overhead for each request is kept stable at about 45μs. In total, the overall overhead for a single request to be about 57μs for both communication and scheduling, which is quite a low overhead. For an instance, Matrix Multiplication in Table 5 sends 98 requests with split-and-merge execution over 340 ms of the total execution time. Since 98 requests produce about 5ms overhead, we can see that it is only 1.6% of the total execution time, which is acceptable.

5.2. Simulation results

We have carried out simulations to evaluate the performance of GPA under various task characteristics in comparison to OPTIMAL as well as Single-GPU and INDIVIDUAL. During the simulations, we also measured the execution time of GPA.

Simulation environment. Each task τ_i was generated according to the following parameters: (i) Period and deadline ($T_i = D_i$) are uniformly chosen in [100, 1000]. (ii) The number of sub-tasks (n_i) is set to 1 + 4k, where k is randomly selected in [1, 5], and the sub-tasks are mapped to the sequence of {CPU-PCI-GPU-PCI-}^{k}CPU (see Fig. 6). (iii) The total worst-case execution time (WCET) of τ_i is uniformly chosen in [5, T_i] and arbitrarily distributed to sub-tasks. (iv) The task is chosen to have an inter-kernel data dependency with a probability of P_D. Then, the overheads of upload/download copy and kernel execution time in multi-GPU modes were determined accordingly depending on whether it has the dependency or not, based on our benefit and cost analysis shown in Section 4.1.

We generated 100,000 task sets with a total GPU utilization U_G ranging from 0.1 to M_G. GPU-bounded tasks consist of 70% of all tasks, PCI-bounded tasks consist of 20%, and the rest are CPU-bounded. According to the parameters determined as described above, we first generated a set of 2 tasks and then keep creating an additional task set by adding a new task into the old set until the total GPU utilization U_G becomes greater than a pre-specified value. During the simulation, the system is assumed to have 4 CPUs, 1 PCI, and 4 GPUs.

Simulation results. Fig. 9 plots the schedulability performance of four schemes (normalized to the OPTIMAL case) when $P_D = 0$. The figure shows a widening performance gap between Single-GPU and the globally optimal one (OPTIMAL) with an increasing total GPU workload (U_G). It also shows that the performance can become worse, compared to OPTIMAL, when decisions are made from an individual application's viewpoint (INDIVIDUAL). This is because some applications can benefit from multi-GPU executions with lower response times at the expense of consuming more resources (i.e., PCI, CPU) and, hence, imposing a larger amount of interference on other CPU- and/or PCI-intensive applications. On the other hand, the figure shows that the performance of GPA stays

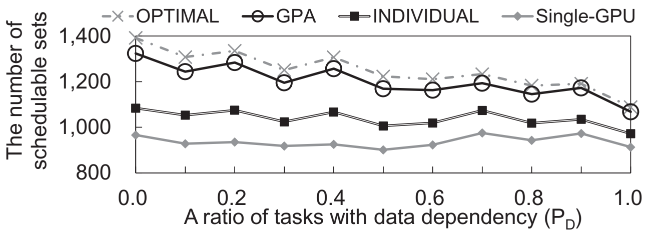


Fig. 10. Schedulability under different data dependency task ratios.

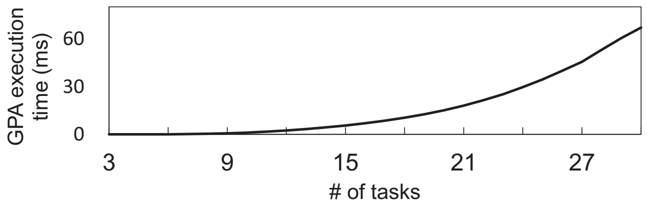


Fig. 11. GPA running time.

very close to OPTIMAL with the loss of up to 19.8% during the simulation.

Fig. 10 compares four schemes (Single-GPU, INDIVIDUAL, GPA, and OPTIMAL) in terms of the number of schedulable task sets when the total GPU utilization U_G is fixed to 1.4. The *x*-axis represents the probability (P_D) with which a task is selected to have an inter-kernel data dependency. The figure shows that Single-GPU stays insensitive to the ratio of tasks with data dependency, while all the other schemes have lower performances when the ratio increases. This is because inter-kernel dependency inherently requires additional data transfer in multi-GPU modes to maintain data coherence across GPUs. In spite of such performance degradation, GPA yields nearly the same results as OPTIMAL does.

GPA running time. We have measured the execution time of GPA. Fig. 11 shows the average execution time of GPA with respect to the number of tasks. We randomly generate 1000 task sets whose utilizations range from 0.1 to 4.0. We note that GPU execution mode assignment is required only when a new task comes in. When the number of tasks is 30, assigning GPU execution mode for all tasks takes less than 100 ms, which seems to be acceptable as an occasional event handling overhead.

6. Related work

Multi-GPU Split-and-Merge support. Several works (Luk et al., 2009; Kim et al., 2011; Lee et al., 2013; Pandit and Govindarajan, 2014) employed multi-GPU split-and-merge execution for performance improvement from the viewpoint of a single kernel. They differ in terms of portability and applicability. In portability, an earlier work (Luk et al., 2009) requires programmers to write programs through the new API to use split-and-merge execution, while the others require no code modification. The applicability of works (Luk et al., 2009; Kim et al., 2011) is limited to the case of regular memory access, while the others support irregular access patterns as well. They analyze the memory access pattern of a kernel and transfer partial memory region each GPU reads and writes. On the other hand, our approach transfers all candidate memory region. Though our approach is not efficient and needs to merge outputs, it is able to transparently cope with irregular memory access patterns without any code modification. Our work can be differentiated from all the above works (Luk et al., 2009; Kim et al., 2011; Lee et al., 2013; Pandit and Govindarajan, 2014) as follows: (i) none of them considers the issues involved supporting real-time constraints, (ii) none of them offers open source implementation, and (iii) our work proposes an analysis-based policy for determining how many GPUs to be used by each application to improve the schedulability of an entire system.

Real-time GPU support. The non-preemptive nature of GPU and DMA architecture could introduce priority inversion. The requests of higher-priority GPGPU applications for memory copy and/or kernel launch may be blocked when lower-priority ones are involved in memory operations and/or kernel execution. A couple of approaches (Kato et al., 2011; Basaran and Kang, 2012) were introduced to reduce the maximum possible blocking time, sharing the principle of making non-preemptible regions smaller. RGEM (Kato et al., 2011) presents a user-space runtime solution that splits a memory-copy transaction into multiple smaller ones with preemption available between smaller transactions. Both RGEM and our work employ fixed priority scheduling, so they maintain global scheduling queues and control the access to resources (GPU and PCI) according to priority. In RGEM, global scheduling queues reside in a POSIX shared memory, and applications access to the queues through POSIX shared memory APIs. In GPU-SAM, SAM Manager maintain global scheduling queues, and applications access to the queues through POSIX message queue. Access to the resources is controlled by POSIX semaphore in RGEM, while GPU-SAM controls through POSIX message queue. Basaran and Kang (2012) propose to decompose a kernel into smaller sub-kernels to allow preemption between sub-kernel boundaries. Berezovskyi et al. (2012) introduce a method to calculate the worst-case makespan of a number of threads within a kernel, which is useful for an estimate of worst-case execution time. Elliott and Anderson (2012) present a robust interrupt handling method for multi-CPU and multi-GPU environments on top of closed-source GPU drivers. GDev (Kato et al., 2012) integrates GPU resource management into OS to facilitate the OS-managed sharing of GPU resource, for instance, allowing different CPU processes to share GPU memory. GPUSync (Elliott et al., 2013) introduces a lock-based multi-GPU management for real-time systems. They allow kernel migration using peer-to-peer memory copy, and the migration cost predictor is introduced to decide migration. Various configurations of multi-CPU and multi-GPU systems are explored in Elliott and Anderson (2014). However, none of them in this category considers split-and-merge execution across multiple GPUs.

7. Conclusion

This paper presents GPU-SAM to accelerate the massive parallelism in real-time multi-GPU systems through split-and-merge execution. We analyze the impact of split-and-merge execution of one application on the application itself and the other applications, and reflect it into schedulability analysis. Building upon the analysis, we develop an algorithm, called GPA, that determines the multi-GPU parallel execution mode of each application. Our experimental and simulation results show that S & M execution can improve the system-wide schedulability substantially and can bring even much more improvement when the multi-GPU parallelism mode is properly determined by GPA.

As a preliminary study, this paper focuses on multi-GPU execution mode assignment in supporting split-and-merge execution for real-time systems. However, there are many other issues worth further investigation. In this paper, we assume the priority of each application is predefined, while appropriate priority ordering can improve schedulability further. This is an interesting topic for future research. S & M execution inherently imposes overheads by introducing additional data transfer. Such overheads can be substantially reduced by utilizing peer-to-peer memory transfer, such as (NVIDIA GPUDirect, 2012), to facilitate data transfer between multiple GPUs. In addition, those overheads can be dramatically reduced in on-chip GPU systems (e.g., AMD Llano Branover et al., 2012) where GPU shares main memory with CPUs. We plan to extend GPU-SAM towards heterogeneous multiple CPUs/GPUs, including on-chip GPU and other accelerators such as FPGAs and DSPs. Our work is portable to such heterogeneous systems, but there exist difficulties to balance workload in the systems. Since the performance difference of heterogenous devices depends not only on hardware specifications but also on application characteristics, deep performance analysis techniques capturing those effect are required to balance workload in such heterogeneous systems. As an open source framework, we hope GPU-SAM could facilitate future research on leveraging S & M execution for real-time multi-GPU systems. Another direction of future work is to extend our system taking into consideration architectural characteristics (Ding and Zhang, 2012; Ding et al., 2013; Ding and Zhang, 2013; Zhang and Ding, 2014; Liu and Zhang, 2015; Liu and Zhang, 2014), parallelism (Wang et al., 2015), or energy (Anne and Muthukumar, 2013).

Acknowledgment

This work was supported in part by BSRP (NRF-2015R1D1A1A01058713, NRF-2010-0006650, NRF-2012R1A1A1014930), IITP (2011-10041313, B0101-15-0557), ICT/SW Creative research program (R2215-15-1019), NCRC (2010-0028680), and NRF(2015M3A9A7067220) funded by the Korea Government (MEST/MSIP/MOTIE). This work was also conducted at High-Speed Vehicle Research Center of KAIST with the support of Defense Acquisition Program Administration (DAPA) and Agency for Defense Development (ADD).

Appendix A. Workspace allocation scheme

In Section 3, we explained our workspace allocation scheme to divide a workspace into several sub-workspaces. Two factors are considered to design the scheme. Firstly, applications utilize memory bandwidth of a GPU with coalesced memory access. Therefore, dividing a certain dimension can cause performance loss when the application accesses memory along the dimension. Secondly, we do not know dimensions which applications access along. If an application accesses memory along the x-dimension but the length of the x-dimension is too short, dividing the x-dimension can cause performance loss since memory bandwidth can be underutilized. This is the worst-case situation that we must avoid. On the other hand, if the length of the x-dimension is long enough, dividing the x-dimension would not cause performance loss since memory bandwidth can be still utilized. From the observations, we design workload allocation scheme which maximizes the minimum length of dimensions as much as possible to avoid the worst-case.

Fig. A.1 shows normalized kernel execution times of workspaces to validate our scheme. we prepare two matrix multiplication kernels with the same size of workspaces, (64, 8192) and (8192, 64), respectively, and divide them along x-dimensions and y-dimensions in half. When dividing the first kernel with workspace (64,8192) into two sub-kernels with workspace (32,8192), the execution time of each sub-kernel is 93% of that of the original kernel. This is the worst-case situation where memory bandwidth is underutilized. In the case, split-and-merge execution with dual-GPU has a benefit of only 7%. A sub-kernel with workspace (64,4096), which is result of our scheme, can have a beneit of 50% through S & M execution. When dividing the second kernel with workspace (8192,64) under the scheme, we divide the x-dimension, but length of x-dimension, 4096, is long enough to utilize memory bandwidth. Finally, a sub-kernel of workspace (8192,32) utilizes memory bandwidth enough and has a benefit of 50%, since dividing the y-dimension does not affect memory bandwidth utilization. In conclusion, our scheme is designed to avoid the worst-case and has 50% benefits always in the case.

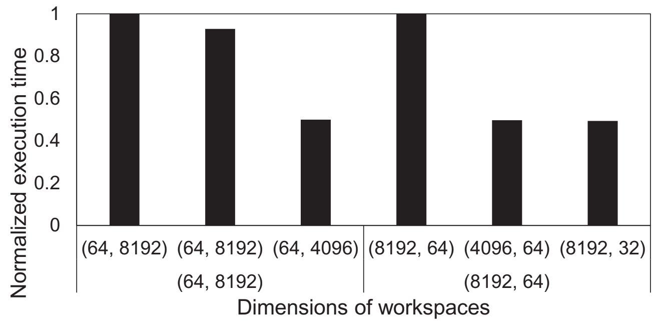


Fig. A.1. Normalized kernel execution times of workspaces.

Appendix B. Implementation Details

In Section 3, we briefly explained how GPU-SAM enables split-and-merge execution of kernels. This appendix describes missing, but important implementation details for supporting (i) *data coherency and correctness* and (ii) *inter-kernel dependency*.

Data coherency and correctness. Originally, an OpenCL application is programmed to run on a single GPU so that the application allocates all the objects in one GPU. Upon each object allocation by the application, GPU-SAM duplicates the object allocation on another GPU.

As one of the objects, buffers can be allocated as read-only or writable on GPUs. Read-only buffers are used only for the input data, and writable buffers can be also used as output buffers that store computation results. For memory coherency, the buffers created in multiple GPUs should have the same data before kernel launch. Since writable buffers are subject to change, when writable buffer B_1 is created on GPU$_1$, GPU-SAM creates B_2 on GPU$_2$, and allocates a *shadow buffer* B_S of the same size on the host memory. The shadow buffer is then used for synchronization and merge; the output merger merges B_1 and B_2 into B_S, and an additional component of SAMCL, *Buffer Synchronizer*, copies the data in B_S into B_1 and B_2 when data coherence is enforced for B_1 and B_2 (right before launching a kernel that accesses B_1 and B_2).

Once buffers are created, the application requests to transfer some input data from the host memory to all the buffers so that an individual sub-kernel can run with its own copy of input data. Last, the data is transferred to the corresponding shadow buffer B_S for coherence purpose and merging.

Before every kernel launch, the buffer synchronizer enforces coherence for all the buffers of sub-kernels of K across GPUs before every kernel launch. There can be some uninitialized data in the buffer due to lack of support on some proper memory operations (i.e., `memset()`) in some OpenCL version (i.e., 1.1). In order for all the buffers to contain consistent data, the buffer synchronizer needs to duplicate the data of B_S to B_1 and B_2, clearing dirty bits.

After all the kernels complete the execution, the output merger merges all the output buffers of sub-kernels to produce the final result. For output merging, let $B[i]$ indicate the element of B of index i. The output merger duplicates B_1 and B_2 to temporary buffers B_1^T and B_2^T on the host memory. It then compares each element of the shadow buffer $B_S[i]$ to $B_1^T[i]$ and $B_2^T[i]$, and overwrites $B_S[i]$ with $B_1^T[i]$ or $B_2^T[i]$ if $B_S[i]$ differs from $B_1^T[i]$ or $B_2^T[i]$. Since we assume that there is no global barrier, and the buffer synchronizer ensures that B_1 and B_2 contain the same data before kernel launch, there must be no conflict; if $B_1^T[i]$ is not equal to $B_S[i]$, $B_2^T[i]$ must be equal to $B_S[i]$, and vice versa.

Table C.1
A full list of benchmark specifications.

		Input			Output					
Benchmark	Source	Type	# of data	Total Size	Type	# of data	Total Size	WorkGroups	Kernel Dependency	Correctness
AESEncrypt/Decrypt	AMD	Bitmap image	2048 × 2048	12MB	Bitmap image	2048 × 2048	12MB	12288	X	O
AtomicCounters	AMD	# of integers	1M	4MB	32-bit integer	1	4Byte	8192	X	X
BinarySearch	AMD	# of integers	1M	4MB	32-bit integers	4	16Byte	16	X	O
BinomialOption	AMD	# of Stock Price	16384	256KB	FP numbers	16384	256KB	255	X	O
BitonicSort	AMD	# of integers	1048576	4MB	32-bit integers	1048576	4MB	524288	O	O
BlackScholes	NVIDIA	# of Stock Price	1048576	4MB	FP numbers	2097152	8MB	1024	X	O
BoxFilter	AMD	Bitmap image	1024 × 1024	4MB	Bitmap image	1024 × 1024	4MB	4096	O	O
DCT8x8x	NVIDIA	# of FP numbers	166777216	16MB	FP numbers	166777216	16MB	8192	O	O
DotProduct	NVIDIA	# of FP numbers	5111808	39MB	FP numbers	1277952	4875KB	4992	X	O
DwtHarr1D	AMD	# of FP numbers	1M	4MB	FP numbers	1048576	4MB	512	X	O
DXTCompression	NVIDIA	Image	512 × 512	4MB	Array	32768	128KB	16384	O	O
FastWalshTransform	AMD	# of FP numbers	1M	4MB	FP numbers	1048576	4MB	2048	O	O
FDTD3d	NVIDIA	Volume Size	376^3	216MB	Volume	376^3	216MB	564	O	O
Filter	CUSTOM	Bitmap image	1024 × 768	768KB	Bitmap image	1024 × 768	768KB	768	X	O
FloydWarshall	AMD	# of Nodes	256	256KB	Nodes	256	256KB	256	O	O
HiddenMarkovModel	NVIDIA	# of state	4096	64MB	Sequences	100	400Byte	4096	O	O
Histogram	AMD	# of 8-bits	16 millions	16MB	Integers	256	1KB	512	X	O
HistogramAtomics	AMD	# of 8-bits	16 millions	16MB	Integers	256	1KB	512	X	X
K-nearest	CUSTOM	# of FP numbers	1024 × 32	128KB	# of FP numbers	1024 × 320	1280KB	128	X	O
MatrixMultiplication	NVIDIA	Matrix Size	2048 × 2048	32MB	Matrix	2048 × 2048	16MB	4096	X	O
MatrixTranspose	AMD	Matrix Size	2048 × 2048	16MB	Matrix	2048 × 2048	16MB	16384	X	O
MatVecMul	NVIDIA	Matrix Size	1100 × 60981	255MB	Vector	1100	255KB	239	X	O
MersenneTwister	NVIDIA	# of matrices	4096	64KB	FP numbers	228M	91MB	32	O	O
Pagerank	CUSTOM	# of FP numbers	4 millions	16MB	# of FP numbers	4 millions	16MB	4096	O	O
PrefixSum	AMD	# of integers	2K	8KB	Integers	2K	8KB	2	X	O
QuasiRandomSequence	AMD	# of vectors	1024	1KB	FP numbers	8192	8KB	8	X	O
RadixSort	NVIDIA	# of FP numbers	1M	4MB	FP numbers	1M	4MB	4096	O	O
RecursiveGaussian	AMD	Matrix Size	512 × 512	1MB	Matrix	512 × 512	1MB	2	O	O
Reduction	NVIDIA	Array Size	1M	4MB	Integer	1	4Byte	512	O	O
ScanLargeArrays	AMD	Array Size	1K	4KB	Array	1K	4KB	4	O	O
SimpleConvolution	AMD	Matrix Size	64 × 64	16KB	Matrix	64 × 64	16KB	16	X	O
SobelFilter	AMD	Bitmap image	512 × 512	1MB	Bitmap image	512 × 512	1MB	1024	X	O
SortingNetworks	NVIDIA	# of Arrays	64 × 16384	4MB	Arrays	64 × 16384	4MB	2048	O	O
Tridiagonal	NVIDIA	# of Systems	128 × 16384	8MB	Systems	128 × 16384	8MB	16384	O	O
VectorAdd	NVIDIA	# of FP numbers	114447872	436MB	FP numbers	114447872	436MB	447062	X	O

Both merge and synchronization need data transfer between host memory and GPU memory. Because it is costly, GPU-SAM tries to reduce the number of transfers as much as possible by running the output merger and the buffer synchronizer on-demand. For an instance, prevent unnecessary synchronization, GPU-SAM does not create the shadow buffer for read-only buffer.

Inter-kernel dependency. Output merge can also happen without explicit buffer read requests when applications consist of multiple inter-dependent kernels. GPU-SAM handles such cases in the following way. Consider a kernel K with its two sub-kernels K_1 and K_2 executing on two GPUs, and writing their own partial outputs on B_1 and B_2, respectively. Then, suppose another kernel K' has two sub-kernels K'_1 and K'_2 that want to execute individually on two GPUs, taking as input the output data of its preceding kernel K. Since the output data of K is split in B_1 and B_2, the partial output results should be merged into their shadow buffer as previously described. When output merging ends, the buffer synchronizer copies B_S back to B_1 and B_2, making K'_1 and K'_2 be ready to launch.

Appendix C. Benchmark results

Table C.1 shows a full list of benchmark specifications from AMD and NVIDIA SDK used for the benefit and cost analysis (see Section 4.1). It contains input/output data type, the number of data, total data size, the number of work-groups, inter-kernel dependencies, and correctness on multi-GPU execution. We exclude some AMD applications that can be executed only on AMD GPUs, and applications for testing purpose (i.e., bandwidth, rendering quality, and simple functionality). Note that GPU-SAM cannot produce correct output for AtomicCounters and HistogramAtomics because they use atomic instructions for global synchronization. Except those applications, all the others introduce correct results regardless of existence of kernel dependencies and irregular memory access patterns.

References

Anne, N., Muthukumar, V., 2013. Energy aware scheduling of aperiodic real-time tasks on multiprocessor systems. J. Comput. Sci. Eng. 7 (1), 30–43.

Basaran, C., Kang, K.-D., 2012. Supporting preemptive task executions and memory copies in GPGPUs. In: ECRTS.

Berezovskyi, K., Bletsas, K., Andersson, B., 2012. Makespan computation for GPU threads running on a single streaming multiprocessor. In: ECRTS.

Bertogna, M., Cirinei, M., 2007. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: RTSS, pp. 149–160.

Branover, A., Foley, D., Steinman, M., 2012. AMD fusion APU: Llano. IEEE M. 32 (2), 28–37.

Chen, D., Singh, D., 2012. Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering. In: Proceedings of 2012 22nd International Conference on Field Programmable Logic and Applications (FPL). IEEE.

Chwa, H.S., Lee, J., Phan, K.-M., Easwaran, A., Shin, I., 2013. Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms. In: ECRTS.

CUDA, https://developer.nvidia.com/cuda-zone (Accessed 2015).

Czajkowski, T.S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D.P., 2012. From opencl to high-performance hardware on FPGAS. In: Proceedings of 2012 22nd International Conference on Field Programmable Logic and Applications (FPL). IEEE.

Ding, Y., Wu, L., Zhang, W., 2013. Bounding worst-case DRAM performance on multicore processors. J. Comput. Sci. Eng. 7 (1), 53–66.

Ding, Y., Zhang, W., 2012. Multicore-aware code co-positioning to reduce WCET on dual-core processors with shared instruction caches. J. Comput. Sci. Eng. 6 (1), 12–25.

Ding, Y., Zhang, W., 2013. Multicore real-time scheduling to reduce inter-thread cache interferences. J. Comput. Sci. Eng. 7 (1), 67–80.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J., 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform gpu programming. Parallel Comput. 38 (8), 391–407.

Elliott, G., Anderson, J., 2012. Robust real-time multiprocessor interrupt handling motivated by GPUS. In: ECRTS.

Elliott, G., Ward, B., Anderson, J., 2013. GPUSync: a framework for real-time GPU management. In: RTSS.

Elliott, G.A., Anderson, J.H., 2014. Exploring the multitude of real-time multi-GPU configurations. In: RTSS.

Garcia, C., Botella, G., Ayuso, F., Prieto, M., Tirado, F., 2013. Multi-GPU based on multicriteria optimization for motion estimation system. EURASIP J. Adv. Signal Process. 2013 (1), 1–12.

Guan, N., Stigge, M., Yi, W., Yu, G., 2009. New response time bounds of fixed priority multiprocessor scheduling. In: RTSS.

Homm, F., Kaempchen, N., Ota, J., Burschka, D., 2010. Efficient occupancy grid computation on the GPU with lidar and radar for road boundary detection. In: IEEE Intelligent Vehicles Symposium (IV).
Karrenberg, R., Hack, S., 2012. Improving performance of OpenCL on cpus. In: Compiler Construction. Springer.
Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., Rajkumar, R., 2011. Rgem: A responsive GPGGU execution model for runtime engines. In: RTSS.
Kato, S., McThrow, M., Maltzahn, C., Brandt, S., 2012. Gdev: first-class GPU resource management in the operating system. In: USENIX ATC.
Khronos, OpenCL, https://www.khronos.org/opencl/ (Accessed 2015).
Kim, J., Kim, H., Lee, J.H., Lee, J., 2011. Achieving a single compute device image in opencl for multiple gpus. In: PPoPP.
Lee, J., Samadi, M., Park, Y., Mahlke, S., 2013. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In: PACT.
Lee, J., Shin, K.G., 2014. Imporvement of real-time multi-core schedulability with forced non-preemption. IEEE Tran. Parallel Distrib. Syst. 25 (5), 1233–1243.
Li, J.-J., Kuan, C.-B., Wu, T.-Y., Lee, J.K., 2012. Enabling an opencl compiler for embedded multicore dsp systems. In: Proceedings of 2012 41st International Conference on Parallel Processing Workshops (ICPPW),. IEEE.
Liu, Y., Zhang, W., 2014. Two-level scratchpad memory architectures to achieve time predictability and high performance. J. Comput. Sci. Eng. 8 (4), 215–227.
Liu, Y., Zhang, W., 2015. Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors. J. Comput. Sci. Eng. 9 (2), 51–72.
Luk, C.-K., Hong, S., Kim, H., 2009. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: MICRO-42.
Madsen, C., Laursen, R., 2007. A scalable gpu-based approach to shading and shadowing for photo-realistic real-time augmented reality. In: Proceedings of International Conference on Computer Graphics Theory and Applications, pp. 252–261.
Nagendra, P., 2011. Performance characterization of automotive computer vision systems using graphics processing units (GPUS). In: Proceedings of IEEE Image Information Processing (ICIIP).
Noaje, G., Krajecki, M., Jaillet, C., 2010. Multigpu computing using MPI or OpenMP. In: Proceedings of 2010 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP). IEEE.
Nordin, P. Obstacle-Avoidance Control-Candidate Evaluation Using a GPU. http://www.computer-graphics.se/gpu-computing/Presentations/gpgpuPeterNordin.pdf
NVIDIA, GPUDirect, http://developer.nvidia.com/gpudirect (Accessed 2014).
NVIDIA. GTC, 2014. http://www.gputechconf.com/highlights/2014-replays. Keynote (Accessed 2014).
Oculus VR, 2012. https://www.oculus.com/rift/ (Accessed 2012).
Palencia, J., Harbour, H., 1998. Schedulability analysis for tasks with static and dynamic offsets. In: RTSS.
Palencia, J., Harbour, H., 1999. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In: RTSS.
Palencia, J., Harbour, H., 2003. Offset-based response time analysis of distributed systems scheduled under EDF. In: ECRTS.
Pandit, P., Govindarajan, R., 2014. Fluidic kernels: cooperative execution of openCl programs on multiple heterogeneous devices. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization.
Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., et al., 2010. Optix: a general purpose ray tracing engine. In: ACM Transactions on Graphics (TOG), Vol. 29. ACM, p. 66.
Pellizzoni, R., Lipari, G., 2005. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In: RTAS.
Rodriguez-Donate, C., Botella, G., Garcia, C., Cabal-Yepez, E., Prieto-Matias, M., 2015. Early experiences with opencl on fpgas: Convolution case study. In: Proceedings of 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE.
Shagrithaya, K., Kepa, K., Athanas, P., 2013. Enabling development of openCL applications on FPGA platforms. In: Proceedings of 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP). IEEE.
Sizintsev, M., Kuthirummal, S., Samarasekera, S., Kumar, R., Sawhney, H.S., Chaudhry, A., 2010. GPU accelerated realtime stereo for augmented reality. In: Proceedings of International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT).
Stuart, J., Owens, J.D., et al., 2011. Multi-GPU mapreduce on GPU clusters. In: Proceedings of 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS),. IEEE.
Sulon, 2014. http://sulontechnologies.com (Accessed 2014).
Tindell, K., Clark, J., 1994. Holistic schedulability analysis for distributed hard real-time systems. Elsevier Microprocess. Microprogr. 40(2–3), 117–134.
Wang, Y., An, H., Liu, Z., Li, L., Yu, L., Zhen, Y., 2015. Speculative parallelism characterization profiling in general purpose computing applications. J. Comput. Sci. Eng. 9 (1), 20–28.
Zhang, W., Ding, Y., 2014. Exploiting standard deviation of CPI to evaluate architectural time-predictability. J. Comput. Sci. Eng. 8 (1), 34–42.
Zhou, L., Fürlinger, K., 2015. Dart-cuda: A PGAS runtime system for multi-GPu systems. In: Proceedings of 2015 14th International Symposium on Parallel and Distributed Computing (ISPDC). IEEE.

Wookhyun Han received B.S. and M.S. degree in Computer Science in 2011, and 2013, respectively, from KAIST (Korea Advanced Institute of Science and Technology), South Korea. He is currently working towards the Ph.D. degree in Computer Science from KAIST. His research interests include resource management in real-time embedded systems and cyber-physical systems.

Hoon Sung Chwa received B.S. and M.S. degrees in Computer Science in 2009 and 2011, respectively, from KAIST (Korea Advanced Institute of Science and Technology), South Korea. He is currently working toward the Ph.D. degree in Computer Science from KAIST. His research interests include system design and analysis with timing guarantees and resource management in real-time embedded systems and cyber-physical systems. He won two best paper awards from the 33rd IEEE Real-Time Systems Symposium (RTSS) in 2012 and from the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA) in 2014.

Hwidong Bae received B.S. and M.S. degrees in Computer Science in 2010 and 2012, respectively, from KAIST (Korea Advanced Institute of Science and Technology), South Korea. His research interests include real-time systems and cyber-physical systems.

Hyosu Kim received B.S degree in Computer Science from Sungkyunkwan University (SKKU), South Korea in 2010 and M.S. degree in Computer Science from KAIST (Korea Advanced Institute of Science and Technology), South Korea in 2012. He is currently working toward the Ph.D. degree in Computer Science from KAIST. His research interests include mobile resource management in real-time mobile systems and cyber-physical systems.

Insik Shin is currently an associate professor in Department of Computer Science at KAIST, South Korea, where he joined in 2008. He received a B.S. from Korea University, an M.S. from Stanford University, and a Ph.D. from University of Pennsylvania all in Computer Science in 1994, 1998, and 2006, respectively. He has been a post-doctoral research fellow at Malardalen University, Sweden, and a visiting scholar at University of Illinois, Urbana-Champaign until 2008. His research interests lie in cyber-physical systems and real-time embedded systems. He is currently a member of Editorial Boards of Journal of Computing Science and Engineering. He has been co-chairs of various workshops including satellite workshops of RTSS, CPSWeek and RTCSA and has served various program committees in real-time embedded systems, including RTSS, RTAS, ECRTS, and EMSOFT. He received best paper awards, including Best Paper Awards from RTSS in 2003 and 2012, Best Student Paper Award from RTAS in 2011, and Best Paper runner-ups at ECRTS and RTSS in 2008.