GPU-SPARC: Accelerating Parallelism in Multi-GPU Real-Time Systems

Wookhyun Han, Hwidong Bae, Hyosu Kim, Jiyoen Lee, Insik Shin
Dept. of Computer Science
KAIST, South Korea
insik.shin@cs.kaist.ac.kr

***Abstract*—GPU (General-Purpose computation on Graphics Processing Units) offers an effective computing platform to accelerate a wide class of data-parallel computing. Multi-GPU's appear as an attractive platform to speed up the computation of data-parallel GPU. This paper aims to explore the feasibility of relaxing the task-level restriction of single GPU use in multi-GPU real-time systems. We develop a multi-GPU runtime support system, called GPU-SPARC, where GPU applications can be automatically split and run concurrently over multi-GPU's. We present the prototype of GPU-SPARC on OpenCL runtime that can provide the service to existing OpenCL applications without any modification to them unless global synchronization is employed. The multi-GPU parallel computing offers the potential for performance improvement but at the same time incurs additional resource consumption. Thereby, we analysis the benefit and cost of executing a GPU application on multiple GPU's and propose a GPU execution mode assignment policy from the perspective of system-wide schedulability. Our experiment results show that GPU-SPARC is able to improve schedulability in real-time multi-GPU systems by relaxing the single-GPU-per-task restriction and choosing better GPU execution modes.**

I. Introduction

With the proliferation of GPGPU (General-Purpose computation on Graphics Processing Units), there is an increasing demand for leveraging GPU to support various real-time applications. One of the major barriers to real-time GPU computing is the non-preemptive nature of GPU, in particular, occurring in kernel launch and data transfer. To overcome such a non-preemptive nature, a couple of studies [1], [2] share the principle of making non-preemptible regions smaller to allow higher-priority application to experience shorter blocking times for kernel launch [1] and for data transfer between host and device memory [2].

As a natural progression from a single GPU, there is an emerging trend towards multi-GPU environments. Multi-GPU environments offer an opportunity to improve performance through load balancing [3], [4], yet raise challenges for real-time scheduling. One of the important lessons from the literature is that the cause of difficulty in global real-time multiprocessor scheduling is the so-called *single-CPU-per-task* restriction that a task can use only a single processor even when multiple other processors are idle at the time [5]. Such a restriction could lead to a scheduling anomaly, called Dhall's effect [6], in which some task sets with significantly low processor utilizations close to 1 may be unschedulable no matter how many processors are available under global RM (rate-monotonic) or EDF (earliest deadline first) scheduling. It is worthy noting that a set of optimal dynamic-priority scheduling algorithms[1], such as pFair [7], LLREF [8], DP-Fair [9], and RUN [10], can successfully resolve such an anomaly. However, the insights behind such optimal multiprocessor scheduling are not directly applicable to non-preemptive GPU environments, since they are inherently based on the preemptive behavior of dynamic-priority scheduling. Hence, relaxing the single-GPU-per-task restriction would help significantly to reduce the difficulty of real-time multi-GPU scheduling.

This motivates our work to develop a real-time multi-GPU support system, called GPU-SPARC (GPU-SPlit And Run Concurrently), that offers the flexibility to relax the single-GPU-per-task restriction by choice towards optimizing the system schedulability. To this end, we first design the run-time mechanism of GPU-SPARC that enables the parallel execution of a single application over multi-GPU's in a transparent manner. GPU-SPARC achieves this through a split-and-merge procedure that decomposes a single kernel into multiple smaller sub-kernels, executes each individual sub-kernel on a different GPU, and merges the partial outputs of sub-kernels to a complete one automatically. GPU-SPARC preserves correctness for a wide range of GPU applications regardless of memory access pattern.

The multi-GPU execution of a single kernel offers the potential for computation speedup at the expense of incurring some overheads due to extra data transfer. This leads to the problem of deciding which applications run in single-GPU mode and which others in multi-GPU mode. There are a couple of studies [11], [12] on the support of the so-called *single-kernel multi-GPU* (SKMG) execution in general-purpose computing, and they approached the decision problem from an individual application perspective; each GPU application is determined independently to run in multi-GPU mode if it is expected to attain a positive speedup, compared to single-GPU execution. However, such individual decisions may harm the system-wide schedulability. In this paper, we present an algorithm, called GEMA (GPU Execution Mode Assignment), that makes decisions from the system-level perspective. Our simulation results show that relaxing the single-GPU-per-kernel restriction allows to improve substantially the schedulability of multi-GPU systems, and GEMA is able to find solutions close to optimal ones (1.4% loss of optimality), significantly outperforming individual decisions up to 56.6%.

We present an open-source prototype implementation of GPU-SPARC[2] as an extension to OpenCL runtime, without requiring any code modification. To validate the correctness of

[1]Their optimality holds for scheduling of a set of periodic/sporadic tasks with deadlines equal to periods on multiprocessors.

[2]`https://github.com/GPUSparc`

GPU-SPARC, we performed experiments with applications from NVIDIA and OpenCL SDK code samples. Our experiment results show that all the samples except those using atomic instructions produce the same results when they run in the multi-GPU mode with GPU-SPARC and in the single-GPU mode with the regular OpenCL runtime. With some more other applications, our experiment results show that GPU compute-intensive real-time applications can reduce response times significantly via multi-GPU-per-kernel executions, leading to substantial schedulability improvement with the proposed GPU execution mode assignment scheme.

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 the issues of relaxing the single-GPU-per-kernel restriction in real-time multi-GPU systems.
- We design the GPU-SPARC system that enables the single-kernel multi-GPU execution in a transparent manner for applications written for a single GPU.
- We introduce the GEMA algorithm that determines in which modes (either single-GPU or multi-GPU mode) individual applications run to improve system-wide schedulability. Our simulation results show that the performance of GEMA is very close to the optimal solution obtained through exhaustive search.
- We present an open-source prototype implementation of GPU-SPARC as an extension of OpenCL runtime that requires no code modifications, demonstrating that GPU-SPARC with GEMA can improve the schedulability of real-time multi-GPU systems significantly.

II. Background

This section first describes the background on GPU architecture briefly. It then introduces OpenCL, the standard GPU programming model, and presents an example to motivate the proposed framework, GPU-SPARC.

A. GPU Architecture

Modern GPU's consist of thousands of relatively simple processing cores optimized for parallel computing. For example, the current NVIDIA GPU, GTX TITAN Black, has 90 streaming multiprocessors consisting of 2880 cores.. GPU's are specially tailored to SIMD (single-instruction multiple-data) processing; the threads running on a multiprocessor are partitioned into groups in which all threads execute the same instruction simultaneously. 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, number of threads in a group, and so forth.

The scheduler in a multiprocessor selects a group to be executed when the currently running group is completed or stalled due to register dependency, memory access, or synchronization. This scheduling is done by hardware with negligible context-switch overhead. Therefore, a larger number of threads is generally beneficial in hiding the memory access latency of a group with the execution of others, and this yields better GPU throughput.

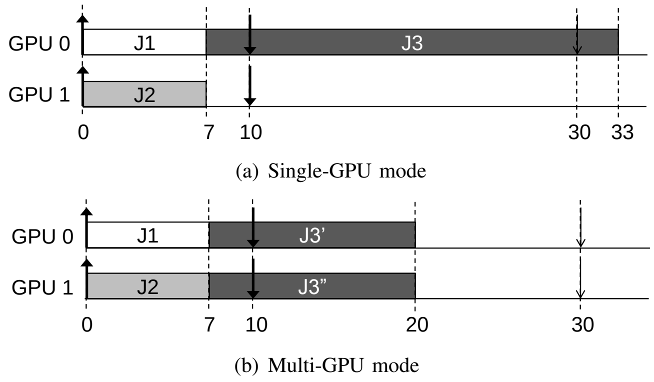


Fig. 1: An example that illustrates the impact of single-resource restriction on real-time scheduling

B. OpenCL Programming Model

Open Computing Language, OpenCL, is one of the programming models for leveraging massively parallel architectures. The OpenCL execution model consists of *kernels* and *host programs*. Kernels are the basic unit of executable code that runs on devices. The host program executes on a CPU, and invokes (or enqueues) kernel execution instances using command queues.

When a kernel is enqueued for execution by the host program, an N-dimensional workspace (or index space) is defined. Each independent element of execution in this workspace is called *work-item*. A work-item is run by a single thread, each of which executes the same kernel function but on different data. Work-items are grouped into *work-group*, to be executed together on the same compute unit of a device. GPU's support synchronization between the work-items only within a work-group and it is the programmers' responsibility to ensure synchronization between global work-items across work-groups.

One limitation in distributing work-items across different GPU's comes into account when applications use *global barriers* or *atomic operations* [3] to coordinate between global work-items. Since the current GPU programming models do not support synchronization primitives between multiple GPU's, in this paper, we do not consider such applications due to the significant overhead of supporting their SKMG execution, as is the case with most, if not all, previous work [11], [12].

C. Motivational Example

In the prevailing GPU programming models and runtime supports such as OpenCL and CUDA, a single kernel is assigned a single GPU only, even though several GPU's are available. Such a single-resource restriction (single-GPU-per-kernel) is regarded as the "*root of all evil*" in global real-time multiprocessor scheduling [5]. As an example, Figure 1 shows

[3]Atomic operations (i.e., atomicAdd) can be used 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.

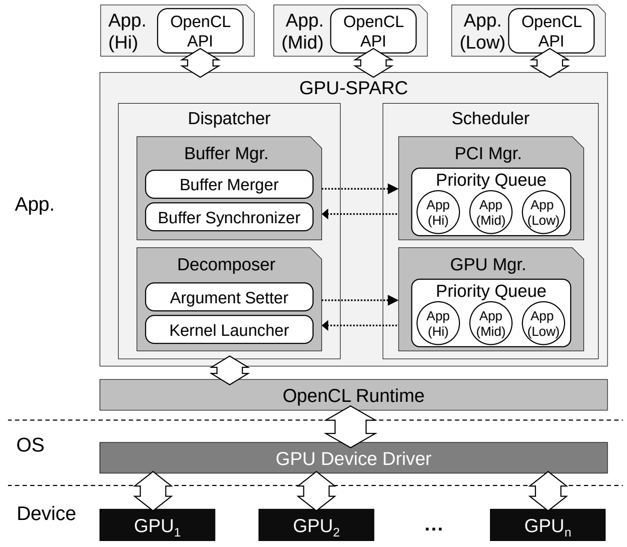
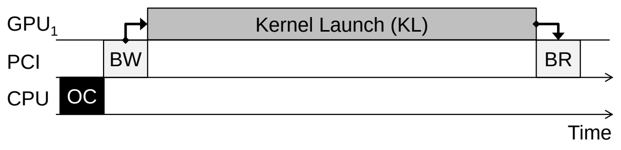
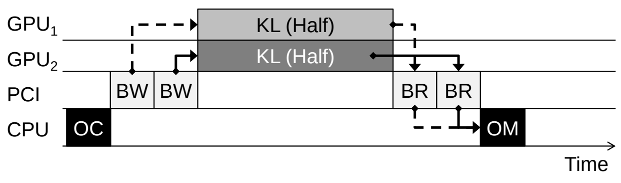


Fig. 2: Logical overview of GPU-SPARC



(a) OpenCL with a single GPU



(b) GPU-SPARC

Fig. 3: GPU application execution flows: Single-GPU and GPU-SPARC

three jobs released simultaneously at time 0. Two jobs, $J1$ and $J2$, are supposed to run for 7 time units to meet the deadline of 10, respectively. The other job, $J3$, needs to execute 26 time units before the deadline of 30. Under the single-GPU-per-kernel restriction, there is no job priority assignment that can successfully schedule those three jobs without any deadline miss. Figure 1(a) illustrates one failure, where $J1$ and $J2$ are assigned higher priorities. On the other hand, suppose $J3$ is able to run concurrently on two GPU's. Figure 1(b) then shows a successful schedule with $J3$ executing on both GPU0 and GPU1, even though jobs are assigned the same priorities as in Figure 1(a). This shows the importance of relaxing the single-GPU-per-kernel restriction in real-time scheduling.

III. GPU-SPARC

In this section, we introduce GPU-SPARC, our real-time multi-GPU runtime support system that provides an illusion of a single GPU over multiple GPU's, enabling individual OpenCL applications written for a single GPU to run on multi-GPU's without any code modification. This section describes the design and implementation of GPU-SPARC, and examines its performance benefit and cost with its correctness.

A. *Design of* GPU-SPARC

The key goal of the GPU-SPARC system is to provide a single GPU illusion over multi-GPU's for a wide range of real-time GPU applications so as to improve system-wide schedulability. Figure 2 illustrates the GPU-SPARC architecture that achieves this goal. GPU-SPARC has three core components, *Mode Selector*, *Scheduler*, and *Dispatcher*. The mode selector decides in which GPU mode (either single-GPU or multi-GPU) each individual GPU application runs when the application joins the system from a system-wide viewpoint (which will be explained in a later section). The scheduler determines on which GPU's and in which order GPU applications perform memory and kernel operations via non-preemptive, priority-based scheduling. The dispatcher performs core functions to support multi-GPU-per-kernel execution. The dispatcher is implemented as an extension of OpenCL runtime library, wrapping existing OpenCL API calls to avoid any code modification.

Each GPU application then comes with its own dispatcher that communicates with the GPU-SPARC scheduler. Upon receiving an OpenCL API call from an application for memory operation or kernel execution, its dispatcher sends a request to the scheduler and waits for an acceptance response. The scheduler maintains prioritized queues of such requests and responds to the requests according to the requesting application's priority, informing which GPU's to use. When a request is granted, its dispatcher adjusts the API call before forwarding it to the OpenCL runtime. When the application was decided to run in single-GPU mode, the dispatcher simply specifies which GPU to use for dynamic GPU allocation. In multi-GPU mode, the dispatcher arranges a series of OpenCL API calls to enable a single kernel to run over multi-GPU's. The detailed procedure of dispatching is described in the next subsection.

The key techniques of GPU-SPARC for enabling multi-GPU-per-kernel execution are based on a split-and-merge strategy. GPU-SPARC splits a kernel K with N work-groups into multiple smaller kernels, called *sub-kernels*, such that each sub-kernel S_i runs the same code of K with N_i work-groups on a GPU G_i, respectively, where $\sum N_i = N$. To this end, GPU-SPARC allows each sub-kernel S_i to have its own copy of input and output data in the memory of its own GPU G_i and maintains data coherence among the multiple copies between kernel executions.

Since sub-kernels produce partial output results from the original kernel's perspective, GPU-SPARC merges such partial output results into a complete one. The complete result will be distributed to each GPU, when accessed in a later kernel execution, to ensure data coherence. Figure 3 compares execution sequences of an OpenCL application running on two systems; an original OpenCL with a single GPU, and GPU-SPARC with two GPU's. The detailed explanation of each step is described in the next subsection.

B. *Implementation of* GPU-SPARC

A key consideration in implementing GPU-SPARC is how to make multi-GPU mode execution produce semantically equivalent results to single-GPU mode execution. To this end, GPU-SPARC supports the following features:

- Coherent memory - All the GPU's should have coherent data in their device memory before kernel execution. In GPU-SPARC, *buffer synchronizer* is responsible for synchronizing each input buffer on GPU's.
- Equivalent computation - Every thread in each sub-kernel should provide computation results equivalent to those a corresponding thread in the original kernel provides. In GPU-SPARC, *kernel code translator* makes certain OpenCL built-in functions adjustable by adding some assistant arguments. *Argument setter* then dynamically adjusts those arguments so that all the threads in sub-kernels are able to access proper memory addresses when *kernel launcher* executes sub-kernels.
- Correct data collection - The merged output of sub-kernels should be equivalent to the output of the original kernel. For this purpose, *buffer merger* combines partial outputs of sub-kernels to generate a whole.

In GPU-SPARC, a typical scenario of executing an OpenCL application in multi-GPU mode can be viewed as a sequence of object creation, buffer write, kernel launch, buffer read and output merge (See Figure 3). Without loss of generality, we assume that GPU-SPARC uses two GPU's for running the application.

Object creation. Originally, an OpenCL application is programmed to run on a single GPU and thereby allocates all the objects in one GPU. Upon each object allocation by the application, the dispatcher of GPU-SPARC duplicates the object allocation on another GPU. Among objects that applications create, buffers and kernel programs are two important types of objects to be considered.

Buffers can be allocated as read-only or writable. 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 both GPU's 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, dispatcher 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 buffer merger merges B_1 and B_2 into B_S, and the buffer synchronizer copies the data in B_S into B_1 and B_2 when data coherence needs to be enforced for B_1 and B_2 (right before launching a kernel that accesses B_1 and B_2).

Kernel program objects are created by an OpenCL API, called `clCreateProgramWithSource(str)`. The parameter, `str`, is the source code of a kernel to run on GPU's. As previously mentioned, each thread (or work-item) shares the same kernel 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, our kernel code translator appends the following auxiliary arguments to all the functions in the kernel, including the kernel itself; `GPUSparc_num_groups`, `GPUSparc_group_offset`, and `GPUSparc_global_size`. It then translates built-in function calls appropriately based on our translation rules. Table I describes the translation rules for all the work-item built-in functions.

TABLE I: Translation rules for work-item built-in functions

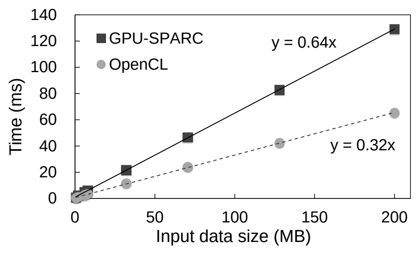
OpenCL	GPU-SPARC
`get_work_dim()`	`get_work_dim()`
`get_global_size()`	`GPUSparc_global_size`
`get_global_id()`	`(get_global_id()` `+ get_local_size()` `* GPUSparc_group_offset)`
`get_local_size()`	`get_local_size()`
`get_local_id()`	`get_local_id()`
`get_num_groups()`	`GPUSparc_num_groups`
`get_group_id()`	`(get_group_id()` `+ GPUSparc_group_offset)`
`get_global_offset()`	`get_global_offset()`

Buffer write. At this step, the application requests to transfer some input data from the host memory to the buffer (B_1) created at the previous object creation step. In addition to transferring data to B_1, dispatcher also transfers the data to the buffer of another GPU, B_2, so that an individual sub-kernel can run with its own copy of input data. Last, dispatcher transfers the data to the corresponding shadow buffer B_S for coherence purpose and merging.

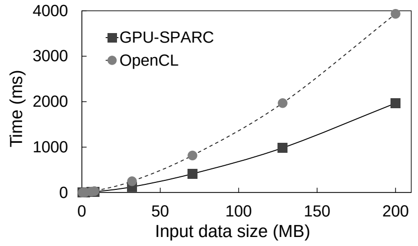
Kernel launch. Dispatcher generally performs two operations when a kernel K launches. First, dispatcher enforces coherence for all the buffers of sub-kernels of K across GPU's before every kernel launch. There can be some uninitialized data in the buffer due to lack of support on some proper memory operations (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. Second, dispatcher ensures auxiliary arguments of each sub-kernel to have appropriate values so that the merged output of sub-kernels is equivalent to that of the original kernel. Thereby, the argument setter makes the arguments in a way that each workspace of sub-kernels is exclusive to each other, and the union of workspace is the same as the workspace of the original kernel. Now the kernel launcher can execute the kernel. Yet, there exists another case we need to handle for inter-kernel dependency, which will be explained later.

Buffer read and output merge. After all the kernels complete the execution, dispatcher 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. Buffer 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 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.

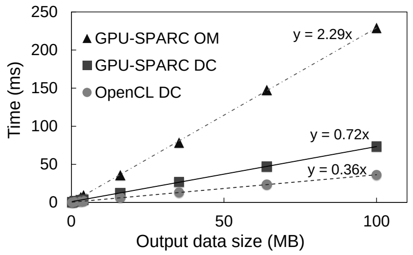
Output merge can happen without explicit buffer read requests when applications consist of multiple inter-dependent kernels. GPU-SPARC handles such cases in the following way. Consider a kernel K with its two sub-kernels K_1 and K_2 executing on two GPU's, and writing their own partial outputs



(a) The overhead of buffer write



(b) The execution time of a kernel for matrix multiplication



(c) The overhead of buffer read and output merge

Fig. 4: The benefits and overheads of multi-GPU mode execution

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 GPU's, 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, buffer synchronizer copies B_S back to B_1 and B_2, making K'_1 and K'_2 be ready to launch.

Last, it is noteworthy that the buffer merger and the buffer synchronizer run on-demand. Both merge and synchronization need data transfer between host memory and GPU memory. Because it is costly, GPU-SPARC tries to reduce the number of transfers as much as possible for better performance. For an instance, GPU-SPARC does not create the shadow buffer for read-only buffer in the object creation step, to prevent unnecessary synchronization.

C. Feasibility Analysis

GPU-SPARC has a great potential for real-time multi-GPU scheduling by reducing overall execution time through kernel splitting. However, it does not come for free. Kernel splitting could impose some non-trivial overheads, which mainly come from extra memory copies in buffer write, buffer read, and output merge. This subsection first investigates the overall benefits and overheads of kernel splitting, and shows the results of running GPU-SPARC on various types of GPU applications. For simplicity, all the multi-GPU mode executions are run on two GPU's.

Kernel splitting benefits/overheads. To observe the effect of splitting kernels, we compare a simplified version of GPU-SPARC and the original OpenCL. The result is illustrated in Figure 4, describing the benefits and overheads in each step of an OpenCL application execution sequence.

Figure 4(a) shows that in the multi-GPU mode, buffer write takes about twice time as much as that of the single-GPU mode. This is because kernel splitting requires an extra memory transfer, from the host memory to the input buffer in another GPU. Since data copy time is proportionate to the data size, the buffer write overhead can be modeled through linear regression.

Figure 4(b) illustrates the benefit of kernel splitting, 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.

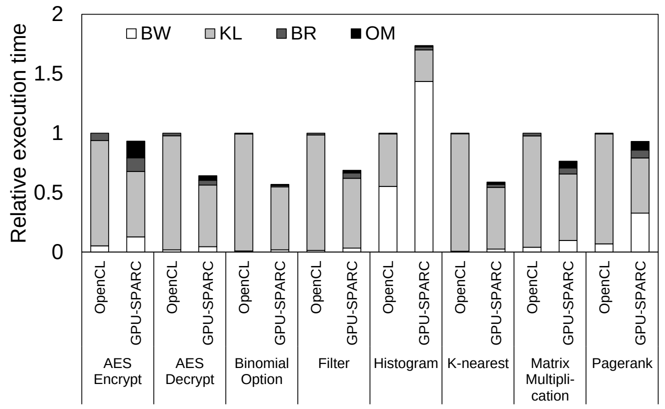


Fig. 5: The execution time of eight benchmarks on OpenCL in single-GPU mode and GPU-SPARC in dual-GPU mode

As described in Figure 4(c), the overhead for buffer read is very similar to the one of buffer write, 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 buffer read. The time for output merge depends on the size of output data, increasing in proportion to the output data size. Note that the output merge time can be easily reduced through parallelization, as is the case with GPU-SPARC implementation.

Benchmark results and correctness. By investigating how kernel splitting affects the execution time of each step in the execution sequence, we examined the potential benefits and overheads of GPU-SPARC. It is then necessary to see if it is beneficial to run various GPU 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 III). Table II describes the specifications of these benchmarks, including input/output data type, data size, buffer size, the number of work-groups, and whether the benchmark consists of inter-dependent kernels. It is worth mentioning that all the benchmarks produced semantically equivalent results to single-GPU mode execution, which is shown in the last column of the table.

Figure 5 illustrates the normalized execution times of benchmarks, comparing the original OpenCL in single-GPU mode and GPU-SPARC in dual-GPU mode. For presentational convenience, we define the execution time of a kernel as the time required to complete buffer write(BW), kernel launch(KL), buffer read(BR), and output merge(OM). As described in the figure, most benchmarks (except Histogram

TABLE II: Benchmark specification

Benchmark	Source	Input			Output			Work-Groups	Kernel Depen-dency	Correct-ness
		Type	Size	Buf Size	Type	Size	Buf 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

TABLE III: Explanation of benchmarks

AES Encrypt/Decrypt	Encryption/decryption algorithm established by Advanced Encryption Standard (AES)
Binomial Option	A numerical algorithm for valuation of options
Filter	A imaging processing technique
Histogram	A graphical representation technique for the distribution of data, applicable to various statistical purposes
K-nearest	K-Nearest Neighbors algorithm, applicable for a wide range of machine learning applications, including face/voice/gesture recognition
Matrix Mul.	Matrix multiplication, a typical GPU-favorable operation for a wide range of applications
Page Rank	A link analysis algorithm used by Google's search engine, applicable for network/context analysis

and Pagerank) are GPU compute-intensive, spending 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. Although buffer write/read time is increased by about 2.5 times, and output merge time is newly added (which is similar to that of buffer read), they save 6.8%-43.2% of the total execution time through concurrent kernel execution on two GPU's. Note that some amount of time spent in buffer write/read is caused by synchronizing uninitialized portions of GPU buffers, as described in the previous subsection. Since such synchronizations are not needed in the latest version of OpenCL (i.e., 1.2), it is possible to further reduce the overall execution time.

On the other hand, Histogram is one of the typical memory-intensive applications that can hardly benefit from kernel splitting. Histogram takes up 44.1% of the execution time on only buffer write in single-GPU mode. Such memory-intensive nature of Histogram makes GPU-SPARC 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 buffer synchronization and merge occur. This causes the buffer write and read time 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-SPARC. Thereby, in order to apply GPU-SPARC in practice, we need to carefully consider the characteristics of applications, such as memory-intensive nature and inter-kernel dependency.

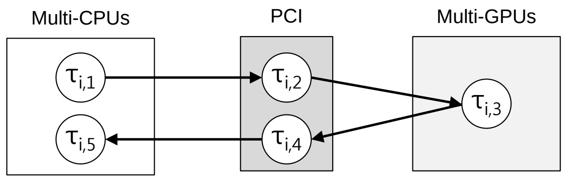


Fig. 6: Pipeline task model

IV. GPU Execution Mode Assignment

As explained in the previous section, GPU applications have a different degree of benefit and cost with the concurrent execution on multi-GPU's by GPU-SPARC, depending on their own memory-intensive or compute-intensive natures. In general, GPU compute-bound applications benefit much from GPU-SPARC, while memory-bound applications would not benefit. Then, this raises an issue of determining which applications run in single-GPU mode and which others run in multi-GPU mode. When applications determine their GPU execution modes from their individual points of view, it may not lead to system-wide optimal decision [4]

In this section, we first describe our system model, recapitulate existing schedulability analysis, present an algorithm to make a system-wide decision, and evaluate the performance the proposed algorithm through simulation.

A. System Model

GPU 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 GPU application as a pipeline task as seen in Figure 6. A task τ_i consists of n_i subtasks, $\tau_{i,1}, \ldots, \tau_{i,n_i}$, under precedence constraints and each subtask utilizes either CPU, GPU, or the PCI bus. It is further characterized by $(T_i, D_i, M_i, \{C^S_{i,j}, C^M_{i,j}, H_{i,j}\})$, where $1 \le j \le n_i$. The task τ_i invokes a series of jobs sporadically with a minimum separation delay of T_i. It has an end-to-end deadline of D_i ($D_i \le T_i$), and M_i indicates its GPU execution mode which is either single-GPU or multi-GPU mode. For each subtask $\tau_{i,j}$, $H_{i,j}$ indicates which type of resource (either CPU, GPU, or the PCI bus) $\tau_{i,j}$ utilizes, and $C^S_{i,j}$ and $C^M_{i,j}$ represent how long $\tau_{i,j}$ holds the resource $H_{i,j}$ in single-GPU and multi-GPU modes, respectively. We consider GPU-SPARC employs non-preemptive and fixed-priority scheduling according to Deadline-Monotonic (DM) algorithm [13].

[4]Due to space limit, we shows an example in Appedix.

Algorithm 1 GEMA - GPU Execution Mode Assignment

```
1: $\mathcal{M} \leftarrow \phi$
2: $\mathcal{S} \leftarrow \{\tau_1 ... \tau_n\}$
3: Initialize the execution mode of each task in $\mathcal{S}$ to single-GPU mode
4: while $\mathcal{S}$ is not empty do
5:     if $\mathcal{S} \cup \mathcal{M}$ is schedulable then
6:         return $\mathcal{M}$
7:     end if
8:     for each candidate $\tau_i \in \mathcal{S}$ do
9:         change the execution mode of $\tau_i$ to multi-GPU mode
10:        for each candidate $\tau_j \in \mathcal{S} \cup \mathcal{M}$ do
11:            $R_j$ = End-to-End Response Time Analysis ($\tau_j$)
12:        end for
13:        $Z_i \leftarrow \max_{\tau_j \in \mathcal{S} \cup \mathcal{M}} R_j / D_j$
14:        change the execution mode of $\tau_i$ back to single-GPU mode
15:    end for
16:    $Z^* \leftarrow \min_{\tau_i \in \mathcal{S}} Z_i$
17:    $\tau_i^* \leftarrow$ the sub-task that has $Z^*$ value
18:    change the execution mode of $\tau_i^*$ to multi-GPU mode
19:    $\mathcal{M} \leftarrow \tau_i^*$
20:    $\mathcal{S} \leftarrow \mathcal{S} \backslash \tau_i^*$
21: end while
22: return $\mathcal{M}$
```

B. End-to-End Response Time Analysis

We consider holistic schedulability analysis for fixed-priority scheduling. The holistic analysis was first introduced in [14], and many extensions were proposed in [15], [16], [17], [18]. 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 subtasks τ_{i_j} experience, that is,

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

The maximum delay $w_{i,j}$ can be calculated iteratively as follows:

$$w_{i,j}^{(a+1)} = C_{i,j}^{M_i} + B_{i,j} + \left\lfloor \frac{\sum_{\tau_k \in hp(i)} \sum_{p: H_{i,j} = H_{k,p}} \left\lceil \frac{J_{k,p} + w_{i,j}^{(a)}}{T_k} \right\rceil \alpha}{m_{H_{i,j}}} \right\rfloor,$$

where $\alpha = C_{k,p}^{S}$ if $H_{i,j}$ = GPU, or $\alpha = C_{k,p}^{M_k}$ otherwise. $B_{i,j}$ is the maximum blocking time from lower priority tasks when $H_{i,j}$ is a non-preemptive resource (i.e., GPU and PCI), hp is the set of higher-priority tasks of τ_i. The iteration starts with $w_{i,j} = 0$ and continues until it converges or reaches a predefined value.

The analysis basically employs the concept of release jitter to deal with the precedence constraints between subtasks. The release jitter $J_{i,j}$ of each subtask $\tau_{i,j}$ is defined as

$$J_{i,j} = \sum_{k=1}^{j-1} w_{i,k} - \sum_{k=1}^{j-1} C_{i,k}^{M_i}.$$

where $J_{i,1}$ is equal to 0.

C. GPU Execution Mode Assignment Algorithm

We present a heuristic algorithm, called GEMA *(GPU Execution Mode Assignment)*, that determines the GPU execution modes of all individual tasks. As shown in Algorithm 1, GEMA divides the task set τ into two disjoint subsets: $\mathcal{S}$ and $\mathcal{M}$, where $\mathcal{S}$ is a subset of tasks in single-GPU mode and $\mathcal{M}$ is a subset of tasks in multi-GPU mode. The general idea is that GEMA gradually increases, through a finite number of iterative steps, the potential to make the system deemed schedulable (i.e., $R_i \leq D_i$ for $\forall \tau_i$). To this end, in each iteration, GEMA seeks to minimize the maximum value of R_i/D_i among all tasks τ_i by changing the GPU execution mode of a single task (τ_i^*) to multi-GPU mode from single-GPU mode. GEMA repeats this process until the system becomes deemed schedulable or all tasks are determined for multi-GPU mode. Thereby, GEMA invokes the end-to-end response time analysis $O(n^3)$ times, where n is the number of tasks.

D. Simulation Evaluation

We present simulation results to evaluate the performance of GEMA.

Simulation Environment. Each task τ_i is 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 Figure 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 mode are determined accordingly depending on whether it has the dependency or not, based on our feasibility analysis shown in Section III-C.

We generate 100,000 task sets with a total GPU utilization U_G ranging from 0 to m_G. According to the parameters determined as described above, we first generate 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. In order to evaluate the performance of GEMA in comparison with some reference points, we add three other GPU mode assignment schemes: Single-GPU, LOCAL and GLOBAL. Single-GPU simply assigns all the tasks into single-GPU mode. LOCAL makes locally optimal decisions from an individual task perspective by assigning tasks into multi-GPU mode if their WCETs become smaller compared to single-GPU mode, or into single-GPU mode otherwise. GLOBAL finds a globally optimal assignment through exhaustive search, exploring 2^n cases, where n is the number of tasks. During the simulation, the system is assumed to have 4 CPUs, 1 PCI, and 4 GPU's.

Simulation Results. Figure 7 plots the schedulability performance of four schemes (normalized to the GLOBAL case) when $P_D = 0$. The figure shows a widening performance gap between Single-GPU and the globally optimal one (GLOBAL) with an increasing total GPU workload (U_G). It also shows that the performance can get worse, compared to GLOBAL, when decisions are made from an individual application's

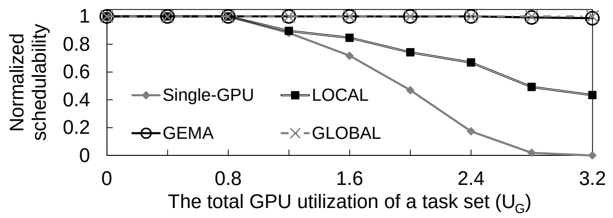


Fig. 7: Schedulability, normalized to global optimum algorithm, under different total GPU utilizations of a task set.

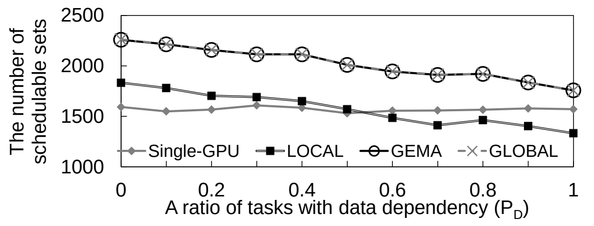


Fig. 8: Schedulability under different data dependency task ratios

viewpoint (LOCAL). 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 GEMA stays very close to GLOBAL with the loss of up to 1.4% during the simulation.

Figure 8 compares the four schemes in terms of the number of schedulable task sets when the total GPU utilization U_G is 1.2. 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 scheme have lower performances when the ratio increases. This is because inter-kernel dependency inherently requires additional data transfer in multi-GPU mode to maintain data coherence across GPU's. It is interesting to observe that the performance of LOCAL can become poor or even worse than Single-GPU when a greater number of tasks are subject to the data dependency. On the other hand, GEMA yields nearly the same results as GLOBAL does.

V. Prototype implementation and Experiment

We have implemented GPU-SPARC prototype as an extension to OpenCL runtime with two additional features: (i) it supports the concurrent execution of an individual kernel on multiple GPU's, and (ii) it offers preemptive, priority-based scheduling for real-time support. Since the key characteristics of GPU-SPARC is its capability of supporting the first feature, our experiment is mainly focused on system-wide schedulability in different mode assignment schemes. To this end, we measure how well GEMA improves the system-wide schedulability, and also how much the scheduling itself affects the overall execution time. Note that although our experiment is taken a number of times to obtain WCET (worst-case execution time), measuring the exact WCET is not in the scope of this paper.

Experimental Setup. Our experiment is conducted on a machine that has two NVIDIA GeForce GTX 560Ti GPU's 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 are used. The GPU has 8 compute units and the CPU has 4 cores. GPU-SPARC uses 4 CPU threads to merge output buffers.

TABLE IV: An application set

Index	Application	Priority	Period	Execution Time
P_1	K-nearest	High	200ms	115ms
P_2	Matrix Multiplication	Medium	500ms	265ms
P_3	Pagerank	Low	900ms	610ms

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 [11]. We decide data chunk size as 2MB, which introduces a negligible PCI overhead of less than 1%. This overhead can be readily measured by comparing copy cost on GPU-SPARC to on a vanilla OpenCL runtime.

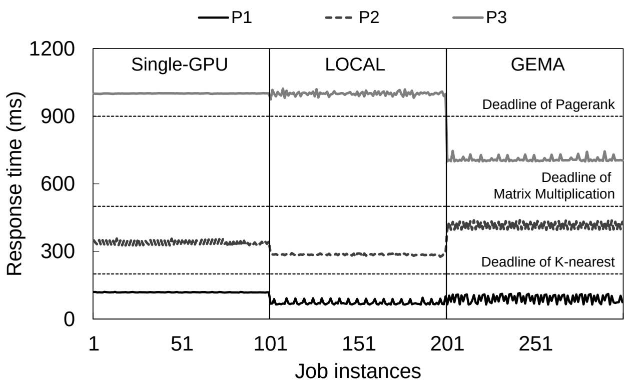
On the other hand, splitting a kernel into smaller-size sub-kernels is known to be effective for reducing GPU non-preemptive regions [2]. However, it is not easy to decide the best sub-kernel size for each application that produces a negligible GPU overhead, because of the need for considering application characteristics. Thus, we just set the number of work-groups in a sub-kernel to 16 times of the number of compute units in a GPU, which produces less than 10% of GPU overhead on our applications. The applications are scheduled as real-time processes by Linux kernel, using SCHED_FIFO scheduling policy. We use the `sched_setscheduler` function to set priority and scheduling policy of applications.

A. System-wide Schedulability

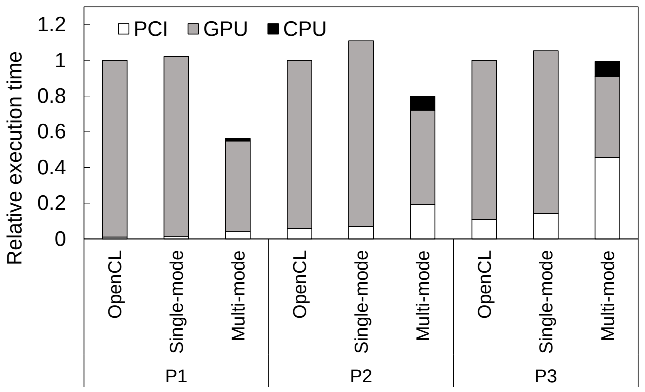
Table IV describes the application set used in this experiment. We consider an implicit-deadline application set such that the deadline of each application is equal to a period. Upon a deadline miss, the application releases a new job while starting a new period at the moment. We measured the execution time of each application, including buffer write/read and kernel launch, when it runs on a vanilla OpenCL runtime.

Figure 9(a) plots the response time of each job of the applications under Single-GPU, LOCAL, and GEMA. Figure 9(b) breaks down the execution times, normalized to the vanilla OpenCL, into three resources CPU, GPU, and PCI when running alone in single-GPU mode and multi-GPU mode. Response times are shown to fluctuate due to non-preemptive regions and different periods of the applications.

(i) Single-GPU. When applications run under Single-GPU, P_2 and P_3 run on the same GPU and P_1 runs on the other. Notice that lower priority applications can be interfered by higher priority ones when the resource is not fully available. Also, all the applications are affected by non-preemptive execution of other applications, resulting in fluctuating response times. The jobs of P_1 and P_2 are successfully finished within



(a) Response time



(b) Job specification

Fig. 9: The response time of job instances and job specification

their dealines. However, the lowest-priority application, P_3, is shown to miss all deadlines, due to GPU interference from P_2 and PCI interference from both P_1 and P_2.

(ii) LOCAL. In order to evaluate the advantage of multi-GPU execution, we perform another experiment with all three applications running in multi-GPU mode. Note that P_1 and P_2 are GPU compute-intensive applications which can gain a lot of benefit from multi-GPU execution, but P_3 cannot since it has inter-kernel dependency. The response times of P_1 under LOCAL become smaller than those under Single-GPU, since it runs with the highest priority and it is beneficial from multi-GPU execution as shown in Figure 9(b). In addition, the response times of P_2 under LOCAL become smaller than those under Single-GPU, because a benefit from multi-GPU execution is larger than the amount of interference from P_1. However, the response times of P_3 stay similar to those under LOCAL and still exceed its deadlines, since P_3 cannot take enough advantage from multi-GPU execution.

(iii) GEMA. From the result of GEMA, GPU-SPARC determines P_1 to run in multi-GPU mode, and the others in single-GPU mode. The response times of P_1 stay similar to those under LOCAL, since it has the highest priority and gets interference from only non-preemptive regions. Although the response times of P_2 become larger than those under Single-GPU due to additional GPU interference from P_1, it still meets the deadline. Unlike LOCAL, the response times of P_3 become smaller enough to meet its deadline, since there is no GPU interference from P_2 and PCI interference from P_2 is reduced. Because P_3 is now properly scheduled, non-preemptive regions of P_3 cause P_1 to be fluctuated, which in turn affects P_2 to also be fluctuated. In conclusion, all applications can be scheduled without any deadline miss under GEMA despite some fluctuations.

B. Scheduler Overhead

We have measured the overhead of GPU-SPARC scheduler for scheduling and communicating with the dispatcher. The scheduling overhead of the scheduler is measured by the time taken for priority queue management. Figure 10 shows the average scheduling overhead of GPU-SPARC scheduler per a schedule request. The x-axis represents a different number of total schedule requests, and the y-axis represents the average

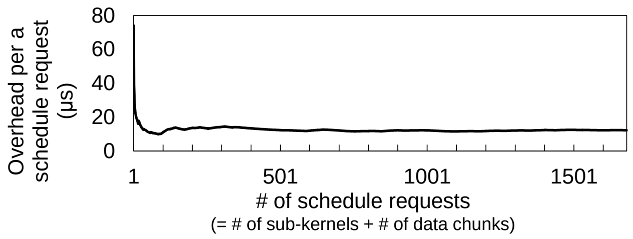


Fig. 10: GPU-SPARC scheduler overhead

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-SPARC 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 IV sends 84 requests with over 200ms of the total execution time. Since 84 requests produce about 5ms overhead, we can see that it is only 2.5% of the total execution time, which is acceptable.

VI. Related Work

Multi-GPU support.

There is an emerging trend towards multi-GPU environments that offer an opportunity to scale performance up with another level of parallelism. A couple of approaches have been introduced to provide a single GPU illusion over multi-GPU's. With the approaches, GPU applications written for a single GPU can utilize multi-GPU's for high performance without the manual process of re-engineering the applications. One is to increase inter-application parallelism by migrating applications between GPU's at some preemptable points for load balancing purposes. This is supported by allowing the kernels of an individual application to be allocated to different GPU's with its data migrated between GPU's transparently [3], [4]. PTask [3] provides a channel abstraction between kernels with a data-flow programming model. With the specification of programmers on data flow between kernels, the proposed scheduler performs automatic GPU allocation optimizing data transfer between host and device memory. This allows two consecutive GPU kernels to re-use computation results without an extra copy to the host memory and back. GPUSync [4] 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. Another approach is to accelerate intra-application parallelism by enabling the parallel execution of a single application over multi-GPU's. This is achieved by distributing the workloads of a single kernel across multiple GPU's [11], [12].

Real-time GPU support. The non-preemptive nature of GPU and DMA architecture could introduce priority inversion. The requests of higher-priority GPU applications for memory copy operations and/or kernel launch may be blocked when lower-priority ones are involved in memory operations and/or kernel execution. A couple of approaches [1], [2] are introduced to reduce the maximum possible blocking time, sharing the principle of making non-preemptible regions smaller. RGEM [1] presents a user-space runtime solution that splits a memory-copy transaction into multiple smaller ones with preemption available between smaller transactions. Basaran and Kang [2] proposes to decompose a kernel into smaller sub-kernels to allow preemption between sub-kernel boundaries. Berezovskyi, *et al.* [19] 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 [20] present a robust interrupt handling method for multi-CPU and multi-GPU environments on top of closed-source GPU drivers. GDev [21] 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.

Our work can be differentiated from all the above studies as follows: (i) GPU-SPARC provides a mechanism for the multi-GPU execution of each single individual kernel (similar to [11] only), and (ii) this paper proposes a policy of determining the execution mode of real-time GPU application, either single-GPU or multi-GPU, to improve the schedulability of an entire system.

VII. Conclusion

This paper presents GPU-SPARC to accelerate the massive parallelism in real-time multi-GPU systems. GPU-SPARC splits a single kernel into multiple sub-kernels and executes the sub-kernels concurrently over multiple GPU's. We analyze the benefit and cost of executing a kernel on multiple GPU's and develop GEMA, an execution mode assignment algorithm that determines the single-GPU/multi-GPU mode of individual tasks. Our simulation and experiment results show that GEMA effectively assigns execution mode to improve the system-wide schedulability.

In this papaer, GPU-SPARC supports two modes, single-GPU and multi-GPU. In the current multi-GPU mode, one is forced to use all the GPU's. We plan to extend this by providing finer-grained modes between single-GPU and multi-GPU. In addition, since GEMA is currently limited to homogeneous multi-GPU's, we would like to extend it towards heterogenous multi-GPU systems. Last, we also plan to reduce data transfer overhead by utilizing [22] to facilitate data transfer between multiple GPUs.

References

[1] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "Rgem: A responsive gpgpu execution model for runtime engines," in *RTSS*, 2011.

[2] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *ECRTS*, 2012.

[3] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: operating system abstractions to manage gpus as compute devices," in *SOSP*, 2011.

[4] G. Elliott, B. Ward, and J. Anderson, "GPUSync: a framework for real-time gpu management," in *RTSS*, 2013.

[5] C. L. Liu, "Scheduling algorithms for multiprocessors in a hard real-time environment," *JPS Space Programs Summary 37-60*, vol. II, pp. 28–31, 1969.

[6] S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[7] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[8] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *RTSS*, 2006.

[9] G. Levin, F. Shelby, S. Caitlin, P. Ian, and B. Scott, "Dp-fair: A simple model for understanding optimal multiprocessor scheduling," in *ECRTS*, 2010.

[10] P. Regnier, G. Lima, E. Massa, G. Levin, and S. A. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *RTSS*, 2011.

[11] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," in *PPoPP*, 2011.

[12] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems," in *PACT*, 2013.

[13] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237 – 250, 1982.

[14] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, vol. 40, pp. 117–134, 1994.

[15] J. Palencia and M. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *RTSS*, 1998.

[16] J. Palencia and H. Harbour, "Exploiting precedence relations in the schedulability analysis of distributed real-time systems," in *RTSS*, 1999.

[17] J. C. Palencia and M. G. Harbour, "Response time analysis of EDF distributed real-time systems," in *Journal of Embedded Computing*, December 2003.

[18] R. Pellizzoni and G. Lipari, "Improved schedulability analysis of real-time transactions with earliest deadline scheduling," in *RTAS*, 2005.

[19] K. Berezovskyi, K. Bletsas, and B. Andersson, "Makespan computation for gpu threads running on a single streaming multiprocessor," in *ECRTS*, 2012.

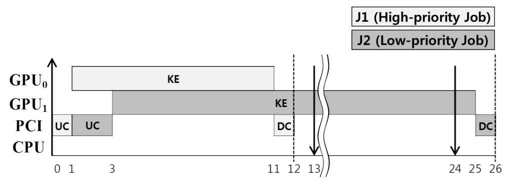
[20] G. Elliott and J. Anderson, "Robust real-time multiprocessor interrupt handling motivated by gpus," in *ECRTS*, 2012.

[21] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: first-class gpu resource management in the operating system," in *USENIX ATC*, 2012.

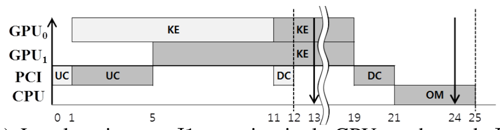
[22] *NVIDIA. GPUDirect*, http://developer.nvidia.com/gpudirect.

TABLE V: Full lists of benchmark specifications

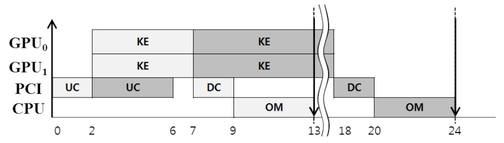
Benchmark	Source	Input			Output			Work-Groups	Kernel Depen-dency	Correct-ness
		Type	Size	Buf Size	Type	Size	Buf Size			
AESEncrypt/Decrypt	AMD	Bitmap image	2048×2048	12MB	Bitmap image	2048×2048	12MB	12288	X	O
AtomicCounters	AMD	# of integers	1M	4MB	32-bit integer	1	4Byte	8192	X	X
BinarySearch	AMD	# 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	376×376×376	216MB	Volume	376×376×376	216MB	564	O	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
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
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	F 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	An 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



(a) Baseline: Both $J1$ and $J2$ run in single-GPU mode



(b) Local optimum: $J1$ runs in single-GPU mode, and $J2$ runs in multi-GPU mode



(c) Global optimum: Both $J1$ and $J2$ run in multi-GPU mode

Fig. 11: An example to illustrate a need for a global decision-making process on execution mode

Appendix - A Motivational Example for System-wide Decision Making for GPU Mode

As explained previously in Section III-C, GPU applications have a different degree of benefit and cost with the concurrent execution on multi-GPU's by GPU-SPARC, depending on their own memory-intensive or compute-intensive natures. In general, GPU compute-bound applications benefit much from GPU-SPARC, while memory-bound applications would not benefit. Then, this raises an issue of determining which applications run in single-GPU mode and which others run in multi-GPU mode.

Figure 11 illustrates an example in which individual, local decision-making on execution mode could lead to a situation, which is not globally optimized. In the figure, two jobs, $J1$ and $J2$, are released at time 0 with deadline of 13 and 24, respectively. Here, we consider $J1$ and $J2$ make use of four resources, CPU, PCI express and two GPU's. $J1$ is the higher-priority job, demanding 1 time unit for upload copy, 10 time units for kernel execution, and 1 time unit for download copy. $J2$, with the lower priority, requires to execute as long as 2, 22 and 1 time units for upload copy, kernel execution, and download copy, respectively.

Figure 11(a) shows that $J2$ misses a deadline when both jobs run in single-GPU mode. Upon a deadline miss, $J2$ might want to run in multi-GPU mode, since it is compute-intensive; it could reduce its response time and thereby increases a chance to meet the deadline. Meanwhile, $J1$ might want to keep in single-GPU mode because it already met the deadline. Moreover, it turns out that its execution in multi-GPU mode is rather harmful to the response time of $J1$ due to its doubled upload and download copy times as well as additional 4 time units for output merge. Figure 11(b), however, presents that $J2$ still misses the deadline when $J1$ runs in single-GPU mode and $J2$ runs in multi-GPU mode. Although individual jobs locally decide in which mode they run in their own optimal ways, $J2$ cannot meet the deadline.

Figure 11(c) shows a situation, where $J1$ and $J2$ both meet their deadlines when they run in multi-GPU mode. This motivates a need for decision-making from the system's perspective, rather than from an individual application's viewpoint. This entails the schedulability analysis of the entire system to make a better system-wide decision.

APPENDIX - BENCHMARK DATA

Table V shows the full lists of benchmark specifications from AMD and NVIDIA SDK. Simlar to Table II, it contains input/output data type, data size, buffer size, the number of work-groups, inter-kernel dependencies, and correctness on the multi-GPU executions. We exclude some AMD applications that can be executed only on AMD GPU's, and applications for testing purpose (i.e., bandwidth, rendering quaility, and simple functionality). Note that GPU-SPARC cannot produce correct output for AtomicCounters and HistogramAtomics because they use atomic instructions for global synchronizations. Except those applications, all the others introduce correct results regardless of existence of kernel dependencies and irregular memory access patterns.