Share-a-GPU: Providing Simple and Effective Time-Sharing on GPUs

Shaleen Garg, Kishore Kothapalli, and Suresh Purini
International Institute of Information Technology, Hyderabad, India
Email: {shaleen.garg@research., kkishore@, suresh.purini@}iiit.ac.in

***Abstract*—Time-sharing, which allows for multiple users to use a shared resource, is an important and fundamental aspect of modern computing systems. However, accelerators such as GPUs, that come without a native operating system do not support time sharing. The inability of accelerators to support time-sharing limits their applicability especially as they are getting deployed in Platform-as-a-Service and Resource-as-a-Service environments. In the former, elastic demands may require preemption where as in the latter, fine-grained economic models of service cost can be supported with time sharing.**

In this paper, we extend the concept of time sharing to the GPGPU computational space using a cooperative multitasking approach. Our technique is applicable to any GPGPU program written in Compute Unified Device Architecture (CUDA) API provided for C/C++ programming languages. With minimal support from the programmer, our framework incorporates process scheduling, light-weight memory management, and multi-GPU support. Our framework provides an abstraction where, in a round-robin manner, every workload can use a GPU(s) over a time quantum exclusively. We demonstrate the applicability of our scheduling framework by running many workloads concurrently in a time sharing manner.

I. Introduction

Accelerators such as GPUs, Intel Xeon Phi, and TPUs are being deployed in a variety of settings including personal electronic devices, laboratory servers, Platform-as-a-Service installations (PaaS), Resource-as-a-Service installations (RaaS), and supercomputers. For this reason, the last decade has seen a massive amount of research on accelerator-based computing resulting in robust support for libraries and tools (cf. [3], [2]), algorithm design and implementation (cf. [7], [8], [19]), and applications (cf. [14]) in a wide variety of domains.

However, accelerators such as GPUs do not expose any operating system and are attached to a host (typically a CPU) as an I/O device. Once a kernel is launched for execution on a GPU, the corresponding GPU resources cannot be timeshared for executing other kernels barring a few native approaches. Starting with the Fermi line of GPUs, NVIDIA supports concurrent kernel execution. In this model, NVIDIA supports a left-over scheduling policy where more than one kernel is scheduled for execution on a GPU if enough GPU resources are available. More recently, the Kepler line of GPUs support concurrent execution of kernels using the Hyper-Q technology [5], which uses multiple hardware queues to avoid false dependencies between kernels.

As kernels often make use of the entire GPU resources, native NVIDIA technologies such as concurrent execution and Hyper-Q are not helpful since resources are allocated using a left-over policy. Also, Hyper-Q and concurrent kernel execution assume that the GPU global memory is enough to cater to all the concurrent programs in the system. Therefore, using currently supported CUDA features, the total GPU memory and compute resource usage of all the kernels executing simultaneously cannot exceed the resources available on the GPU.

These limitations mean that effective time sharing of GPUs is not supported presently. As GPU based installations targeted at multi-user environments, such as laboratories, PaaS, and RaaS are becoming commonplace due to economic considerations, users sharing a GPU resource are forced to use the GPU in a First-Come-First-Serve (FCFS) basis. In an FCFS model, short duration workloads will have to wait for a potential long duration workload submitted earlier to finish.

Effective time-sharing requires a mechanism for preemption so that during a given time quantum a workload has full and exclusive access to complete system resources. Most operating systems including recent ones such as Android and iOS provide support for time-sharing on a variety of devices including mobile and handhelds. To do so on a GPU, one needs to design and implement mechanisms for preemptive execution along with the ability to save and restore the state of a kernel under execution on the GPU [22]. As GPU vendors do not expose their hardware details, supporting time-sharing on GPUs via even a seemingly simple round-robin (RR) scheduling is quite challenging.

There have been recent ventures into this problem but they fall short of the required goal. For example, Calhoun and Jiang [9] support preemptive round-robin scheduling of kernels on GPUs, but their context switches are very taxing because the state information they capture is very heavy. When only one program is run using their framework, the slowdown experienced is around 30x as compared to running the same program outside of their framework.

A more recent paper by Wu et al. [26] that also aims at enabling kernels share the GPU resource by preempting kernels works under the assumption that the GPU global memory is large enough to cater to the total memory required by all the workloads user execution. This limitation coupled with the fact that the available memory on GPU cards is only of the order of tens of GB impedes the usability of the framework presented in [26].

It is therefore essential to relocate the memory currently in-

use by a GPU kernel so that the next kernel can use the entire GPU memory. Doing so requires supporting a form of memory management across the memory systems of the GPU and the CPU. In the absence of access to the internal hardware and other aspects of the GPU, these become difficult to support with low overhead.

Simulation based studies on preemptive round-robin scheduling such as Xu et al. [27] are able to address specific performance issues as they can mine a wealth of data such as the number of cache misses provided by simulators such as GPU-Sim [28].

In this paper, we propose a cooperative multi-tasking appraoch for effective and efficient time-sharing of GPUs. Our approach builds a scheduler framework that orchestrates the execution of multiple kernels on a GPU based on three main ideas.

- We divide a kernel into multiple micro-kernels so that preemption can be managed on the GPU and the state of the kernel can be efficiently captured via the number of micro-kernels that finished execution.
- As multiple kernels are sharing the GPU resources, the available space on the GPU may no longer be sufficient. Hence, we provide for an automatic memory manager that keeps only the data required by the kernel under execution on the GPU and moves data corresponding to other kernels to the CPU memory. This data management is done while ensuring that the GPU computing resource is never idle and all data transfers are effectively hidden by compute.
- To address the needs of typical installations, we extend our framework to support multiple GPUs attached to a single CPU. By doing so, workloads operating in our framework can automatically utilize the resources of multiple GPUs in a time-sharing manner.

These three ideas ensure that kernels can share one or more GPU resources seamlessly. To make use of our framework, no changes to actual source programs are required. Our framework introduces simple modifications to the source programs that replace certain function calls with custom macros. Other details to be provided by the user that are program-specific are to be included as auxiliary source files (see Section II) at the compile stage. While our framework currently supports round-robin scheduling, other scheduling policies such as shortest-job-first can be suitably incorporated into our framework.

We conduct experiments using our framework on a collection of workloads from the Rodinia benchmark [10] and CUDA SDK examples [17] on a platform of two NVIDIA K40c GPUs. Our experiments indicate that the overhead of running a single workload in our framework on a single GPU is under 4% on average whereas a speedup of 1.65x is observed when using two GPUs. When running multiple workloads simultaneously in our framework with a single GPU, the average completion time is proportional to the number of workloads whereas using two GPUs the average completion time is 40% faster.

A. Related Work

Given the rise in accelerator-based computing in general and GPU computing in particular, enabling multi-programming support for GPUs is witnessing a continued interest in recent years [9], [26], [29], [27]. The range of ideas explored in this context span from providing a software-based application level kernel preemption, hardware-assisted and simulation based studies for improving the GPU resource utilization, among others. In the following, we discuss works that are most related to the current paper by suitably categorizing them.

a) Software Based: Calhoun and Jiang [9] introduce an application level check pointing for CUDA kernels so that a kernel can be preempted. During the checkpointing, the state of the kernel is captured as the values of the variables used. The programmer has to indicate these variables to be included in the checkpoint by using pragma directives. A compiler parses these directives and supports application level checkpoint and restart that can eventually be used for kernel preemption. The result of Calhoun and Jiang however indicates that the overhead suffered by their scheme is quite high even for simple kernels such as adding two vectors. This is due to the large footprint of the checkpointing information.

Zhong and He [29] study a slicing of kernels into kernelets, a technique similar to what we propose in our current work. The kernelet is also identified as a continuous set of block indices. Their goal however is to pick an execution sequence of the kernelets of various kernels so as to minimize the total execution time. Their work assumes that all kernels can have their data reside in memory and no data transfers are needed. Their results are also limited to running a pair of kernels concurrently.

In a recent work, Wu et al. [26] consider a problem whose scope overlaps with our problem. Their approach also partitions a kernel into microkernels. Wu et al. [26] support two scheduling strategies: priority based and weighted round-robin. In their evaluation, however, results are shown for up to three kernels that are run concurrently. Another limitation of the work of Wu et al. [26] is the assumption that the data for all the kernels that run concurrently has to reside on the GPU at all times.

Providing software based application level checkpointing has been studied in e.g., Takizawa et al. [24] and Nukada et al. [18]. Checkpointing however aims at fault-tolerance and does not support preemption or time-sharing.

b) Simulator Based: There are several recent works that use simulators to study scheduling and resource sharing of GPUs. Pai and Govindarajan [20] implement a shortest-job-first based scheduling policy while picking one among multiple kernels that have to be executed on a single GPU. From well-known operating system as well as algorithmic literature it is clear that such a policy can minimize average latency [11], [22]. In this work, knowing the execution time of a GPU kernel is essential. For this, Pai and Govindarajan use profiling techniques and predict the time taken for a given kernel to execute on a GPU. Aguilera et al. [6] study how to support GPU resource sharing among kernels that

execute simultaneously on a GPU while balancing between the twin objectives of fairness and performance. Wang et al. [25] argue that not only resource sharing but also quality-of-service guarantees are important when sharing a single GPU among multiple simultaneously executing kernels. However, most of the above cited works rely on the ability of simulators such as GPUsim [28] to provide lots of runtime information about kernels executing on a GPU. Since such information is not available in practice, there is hindrance to the immediate applicability of these approaches.

A technique called warped-slicer by Xu et al. [27] studies how to allocate resources within an SM to more than one kernel assigned to the SM for maximizing utilization of SM resources such as ALUs and SFUs while simultaneously minimizing the performance loss suffered by individual kernels due to concurrent execution. To find such suitable co-habiting kernels in a single SM, their approach requires a prior profiling phase where extensive information about the resources required by a given kernel is collected.

Tanasic et al. [23] implement Dynamic Spatial Sharing (DSS), a hardware scheduling policy that dynamically partitions the resources (GPU cores) and assigns them to different processes according to the priorities assigned by the host OS. This is in addition to hardware preemption.

c) Memory Management: Noticing that limited availability of on-chip memory as one of the inherent drawbacks of GPUs for large, space-intensive applications, several researchers have worked on providing suitable abstractions that allow a GPU program to use the CPU memory as a secondary memory. Enabling such abstraction requires support of the form of virtual memory with the CPU memory and the GPU memory acting as two levels of a memory hierarchy. Examples of such schemes are presented in [15], [30]. In many of these works, the abstraction supported is for a single GPU application to make use of both the CPU and the GPU memory smoothly while the runtime system provides for automatic data transfer between the CPU and the GPU. A key concern therefore is to ensure that the application running on the GPU never encounters a situation where the required data is either not in GPU memory or is stale. In our setting, since the data transfers happen asynchronously while the concerned application is not currently executing on the GPU, our memory management mechanism is much simpler to work with.

d) Multi-GPU: Several researchers studied the possibility of executing a kernel written for a single GPU to be executed on multiple GPUs with minimal intervention by the programmer [21], [16], [13]. In many of these cases, a portion of the same kernel is run on multiple GPUs/devices and no resource sharing or scheduling is attempted. Ramashekar and Bondhugula [21] present a fully automatic scheme for data allocation and buffer management for programs involving affine loop nests. Lee et al. [16] start with a single data-parallel kernel written in OpenCL, partitions the workload across a collection of devices, generates kernels for executing the partitions, and merges the partial outputs together.

B. Organization of the Paper

The rest of the paper is organized as follows. Section II describes the architecture of the proposed framework including cooperative multitasking and memory management within the context of a single GPU. Experimental results of the framework are presented in Section III. Section IV then describes how our framework supports multiple GPUs along with results on multiple GPUs. The paper ends with concluding remarks in Section V.

II. Architecture of the Scheduler

Our aim in this project is to build a preemptive scheduler which orchestrates the execution of compute kernels on the available heterogeneous collection of GPUs. The scheduler runs as an user level process on the host (CPU) operating system.

A. Compilation Framework

To use our scheduler, a programmer has to supplement his CUDA program with a metadata file containing annotated list of kernels to be launched on the GPU. A detailed description of the metadata file is provided later in this section. The source program and the metadata file is passed to a source-to-source translator. The source CUDA program is modified by adding certain macros and replacing the function calls cudaMalloc, cudaMemcpy and kernel launch calls with custom macros. The metadata file is used to generate helper functions which can assist the program execution in the scheduling framework. Figure 1 shows the scheduler compliant CUDA program automatically generated from the source vector addition program. Figure 2 shows the associated metadata file. Overall, the effort required to make a program compliant to our scheduler is very minimal and is a one time effort at the compilation time.

B. High-level Overview

Recent NVIDIA GPUs such as the K40c, P100, and V100 provide native technologies like HyperQ and concurrent kernel execution to natively enable sharing of GPU resources but that may not suffice in real world scenarios since they use a left-over policy where a new kernel is launched on the GPU only if there are required resources still available on the GPU. However, most GPU kernels are programmed in such a way that they can make use of the entire GPU resources by themselves and hence do not leave any free GPU resources. However, with a time-sharing approach, which we take in this work, a kernel can get all the available GPU resources.

The challenge is to let multiple active ready-to-execute kernels time-share the available GPUs which inherently supports no preemption. In the absence of such a preemption mechanism, we resort to cooperative multitasking, wherein a kernel yields the GPU after executing a predetermined number of thread blocks. Our source-to-source translator enforces kernel yielding by rewriting a kernel launch into a sequence of micro-kernel launches. The scheduler assigns microkernels to the available free GPUs and also manages their state as reflected in the global memory of the respective GPU they are currently

```
__global__ void vectorAdd(double *A, double *B, \
                          int N){
  int id = (BlockIdx(vectorAdd) * blockdim.x) +
    threadIdx.x;
  //Native call uses blockIdx.x instead BlockIdx()
  if(id < N)
      A[id] += B[id];
}

int main(){
  //begin pre-processing code
        ....
    //end pre-processing code

    //From cudaMalloc()
  customCudaMalloc(&d_A, size);
    //From cudaMemcpy()
  customCudaMemcpy(d_A, h_A, size, MemcpyH2D);
    //From cudaMalloc()
    customCudaMalloc(&d_B, size);
    //From cudaMemcpy()
  customCudaMemcpy(d_B, h_B, size, MemcpyH2D);
    /* Native Kernel call
    vectorAdd<<<grid, 1024, 0>>>(d_A, d_B, n); */
    KernelCall(vectorAdd, grid, \
     vectorAdd<<<ScBlocks, 1024, 0>>>(d_A, d_B, n));
    //begin post-processing code
        ....
}
```

Fig. 1: All the source program changes are done automatically.

Metadata about variables for naive VectorAdd program

```
//BEGIN .dat file
1
VECTORADD
int, d_A, h_A, size_int, yes, yes
Nbr, d_B, h_B, size_nbr, no, yes
#
//DataType, deviceVar, hostVar, size, Mutable?, InputVar?
//END of .dat file
```

Fig. 2: Metadata file for the example VectorAdd program.

assigned to for execution. In order to hide communication latency, we use a double buffering technique by overlapping the execution of a microkernel on the GPU with the state transfer of the next microkernel about to get access to the GPU.

C. Kernels and Microkernels

In the CUDA programming interface for GPUs, programs to be run on GPUs are described as kernels. Each such kernel is visualized as a computational three dimensional grid. Each grid cell is called a cooperative thread array (CTA) which is in turn a three dimensional block of threads. Threads are the smallest computational entity in a kernel. By definition and execution behavior, computations do not depend across CTAs and they can be executed in concurrent out-of-order batches without changing the intended output. Moreover, all the variables inside each CTA are rendered invaluable once it is complete. The state which needs to persist is stored in the global memory of the GPU. Hence each kernel launch can be transformed into a sequence of microkernel(a subset of CTAs) launches with no necessary ordering constraint between them.

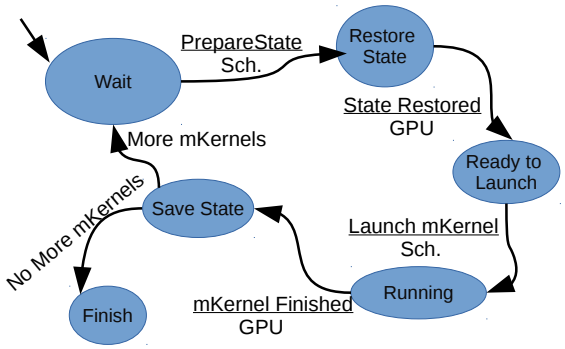


Fig. 3: Figure shows an illustration of the possible states of a user program along with events that trigger the state changes. On the arcs, the notation $\frac{A}{B}$ indicates that event A triggered by entity B makes the kernel change its state as indicated. (Sch. refers to the proposed scheduler and mKernel stands for micro-kernel).

D. Cooperative Multitasking

Our source-to-source translator replaces the kernel launch in a native source program with a `KernelCall` macro (refer Figure 1). The kernel call macro iterates over the three dimensional computational grid by tiling it into micro-grids. A microkernel is launched for every such microgrid. Further, the CUDA variables `blockIdx.x`, `blockIdx.y` and `blockIdx.z` are replaced with the macros `BlockIdx`, `BlockIdy` and `BlockIdz` respectively. These macros map the grid coordinates from the microgrid to their original coordinates in the macrogrid thus maintaining the semantics of the original kernel.

Algorithms 1 and 2 show the pseudocode for the `KernelCall` macro and the main kernel scheduler respectively. The KernelCall of the user processes and the scheduler communicate using FIFO queues provided by the Linux kernel. In order to launch a microkernel, a user process requests for a timeslice by sending `RequestTimeSlice` message to the scheduler. At this time, the user process will be in the `Wait` state as per the state diagram in Figure 3. We recommend the reader to refer the communication timeline diagram in Figure 4 to follow the ensuing discussion. The scheduler enqueues the requesting process and once it reaches the head of the queue, it sends a `PrepareState` message to the user process. The user process moves from the `Wait` state to the `RestoreState` state. In this state, the process acquires GPU memory through the usual `cudaMalloc` function call and transfers the state from the CPU main memory to the GPU global memory. While this state transfer is going on, another kernel could be executing on the GPU. Thus we overlap communication and computation. Then the user process moves to `ReadyToLaunch` state by sending `ReadyToLaunch` message to the scheduler. Once the current

Algorithm 1 KernelCall Procedure.

```
1: while microkernels left do
2:     SendMessage(RequestTimeSlice) /* to scheduler */
3:     WaitForMessage(PrepareState) /* from scheduler */
4:     Request GPU memory.
5:     Restore state on GPU global memory.
6:     SendMessage(ReadyToLaunch) /* to scheduler */
7:     WaitForMessage(LaunchKernel) /* from scheduler */
8:     Launch micro-kernel
9:     SendMessage(YieldedGPU) /* to scheduler */
10:    Save state to CPU main memory
11:    SendMessage(YieldedGPUMemory) /* to scheduler */
```

Algorithm 2 Asynchronous event driven pseudocode for scheduler.

```
1: while TRUE do
2:     upon event RequestTimeSlice do
3:         Enqueue process request
4:     upon event ReadyToLaunch do
5:         if GPU free then
6:             Dequeue process at the head of the queue.
7:             SendMessage(LaunchMicrokernel)
8:     upon event YieldedGPU do
9:         Mark the GPU as free.
10:    upon event YieldedGPUMemory do
11:        SendMessage(PrepareState)
12:    /* to the process at the head of the queue*/
```

process occupying the GPU yields, the scheduler dequeues the process at the head of the queue and sends `LaunchKernel` message if it is already in `ReadyToLaunch` state. Then the corresponding process moves to `Running` state and launches the microkernel. Once the microkernel execution is over, the process sends a `YieldedGPU` message to the scheduler and moves to the state `SaveState`. This enables the scheduler to launch a microkernel from some other process at the head of the queue. The process in `SaveState` transfers the state from GPU to CPU memory and then sends `YieldedGPUMemory` to the scheduler. Note that again there is an overlap between communication and computation maximizing GPU utlization. The scheduler on receiving this message permits the next process at the head of the queue to enter `RestoreState` by sending `PrepareState` message to it. Once all microkernels corresponding to an user process gets over, the process enters `Finish` state. It has to be noted that all the necessary code for communication and bookkeeping is automatically generated without burdening the end programmer.

Currently, the framework supports round-robin scheduling algorithm to schedule kernels. One can alternatively use a different scheduling algorithm like priority based round-robin, in a plug and play fashion to suit their needs.

Kernel Control Block (KCB) Our framework maintains a per kernel data structure called Kernel Control Block (KCB). The KCB contains information such as the state of the kernel from Figure 3, the progress of a kernel in terms of the microgrids already launched, GPU time taken by the last microkernel and the requested grid size. As we add more functionality to our framework, like multi GPU support, we extend the KCB to hold more information about a kernel.

E. Memory Management

If we have to retain the global memory state of all the kernels in the system within the GPU memory, then it puts a restriction on the number of active kernels that can be multiplexed on the GPU. We eliminate this constraint by saving and restoring the GPU global state of a kernel in the CPU memory. As mentioned earlier in the Section II-A, a programmer has to supplement the CUDA source program with a metadata file (refer Figure 2) wherein for each kernel in the program the following information has to be provided: the device variable, its associated host variable, the dynamic size, does the variable get dirty in the kernel and if the variable is an input variable or not. This meta-data is used by a source-to-source translator to generate suitable source code in the form of macros provided as a helper file to be included in and used by the program. Note that all the usages of the macros is automatically incorporated in the program at the preliminary parsing and augmentation step, so the programmer need not know the details about the underlying scheduler. These generated macros help transfer the state of dirty variables between GPU and CPU memories as a kernel occupies and yields a GPU. While a kernel is executing on the GPU, the kernel at the head of the queue prepares itself for subsequent execution on GPU by transferring its state to the GPU global memory in the background. Thus by overlapping communication and computation we maximize GPU utilization, thereby improving the turnaround time for all the kernels in the system. Further, before a microkernel already occupying the GPU transfers its state back to the CPU memory, it checks if there are any other processes waiting in the queue. If there are no other processes waiting, then it launches the next microkernel from its microgrid by eliminating a state transfer from GPU to CPU and back.

F. Timeslice Length

The length of the timeslice is indirectly controlled by the dimensionality of the microgrid as no preemption is possible while a microkernel is executing. The dimensionality of the microgrid is a runtime configurable parameter which is supplied by the scheduler to the user process. Although, in the current work, this parameter is set to a fixed value for all the processes, it is possible to vary this parameter on a per-process basis thereby controlling the timeslice allocated to the corresponding process. For example, we can dynamically profile the time taken by a CTA of a kernel and based on that decide the number CTAs per microkernel launch for fair sharing of GPU computing time. If the timeslice length is shorter, then the overheads due to kernel launch will be high and moreover we may not be able to hide the communication latency by overlapping it with computation. The timeslice

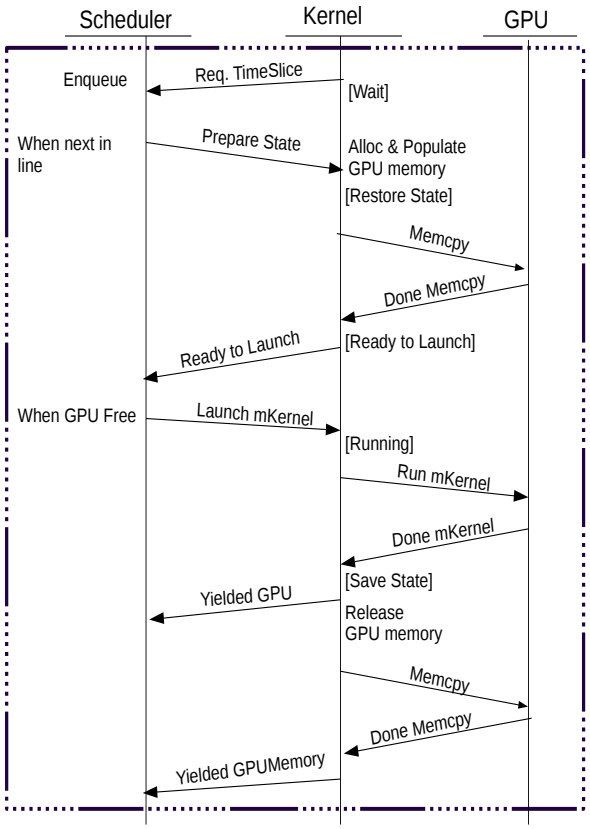


Fig. 4: A picture showing the timeline of communication across the scheduler, kernel, and a GPU. For the kernel, text written inside square brackets refer to the state of the kernel while text written without brackets correspond to actions taken by the kernel. Memcpy is used to indicate transfer of memory across the GPU and the CPU and the direction of the transfer is not indicated explicitly. The events enclosed in the box repeat until there are no more micro-kernels to execute.

should be at least long enough so that it covers the time taken by the previous microkernel occupying the GPU to transfer its state to CPU and for the next mircokernel to-be-launched to transfer its state to GPU. If S_{prev} and S_{next} denote the size of the state for the previous and next microkernels, then a lower bound on timeslice for effectively hiding the communication latency is

$$T_{slice} > \frac{S_{prev} + S_{next}}{pcie\_bus\_speed}. \quad (1)$$

However, if the timeslice length is too long, it would adversely effect the turnaround time for kernels with shorter execution times.

III. Results and Analysis

Experimental Platform All our experiments were performed using two NVIDIA K40c GPUs. The two GPUs are connected to an Intel Core i7 X980 CPU using PCIx links. The Intel i7 X980 CPU has 6 physical cores and 2 threads per core, it has a combined clock speed of 3.33 GHz, a RAM of 24 GB, and runs on Linux Version 3.10.0.

The NVIDIA K40c GPU has 15 SMXs each with 192 cores for a total of 2880 cores. Each core is clocked at 745 MHz and offers a single precision throughput of 4.3 TFlops. The throughput in case of double precision arithmetic is 1.43 TFlops. The K40c GPU has a device memory of 12 GB while each SMX has a 64 KB memory shared across its 192 cores. To program the K40c GPU, we use CUDA Version 8.0.61 [4]. The CUDA programming model consists of a kernel that is logically viewed as a collection of blocks indexed by a three tuple. Each block has multiple threads that are further grouped into warps. The number of threads in a warp and the number of warps in a block are configurable by the programmer based on the application.

A. Dataset

In this section, we study our framework on various workloads with different resource requirements and characteristics from the Rodinia benchmark [10], CUDA SDK examples [17], and string sorting from [12]. Key properties of the workloads are listed in Table I. For each workload, we identify the nature of the workload as memory intensive or compute intensive with respect to GPU computing. This property impacts parameters of the framework such as the timeslice.

The input for the BFS workload is generated as an Erdős-Rényi random graph, $\mathcal{G}(n, p)$ with $n = 32$ M and $p = 10^{-7}$. The input(s) to the matrix programs, Gaussian, matTrans, and matMul are dense matrices of floating point numbers chosen uniformly at random. For the kmeans workload, the input is a collection of 2-D points with integer point coordinates. For the dxtc workload, the input is an image. For the StringSort workload, the input is a list of random strings of various sizes.

In Table I, the column labeled "Memory Footprint" indicates the GPU global memory used by the workload (as reported by `nvidia-smi`) on an input of size as specified in the column labeled "Size". The column "Native Runtime" shows the time taken by the corresponding workload on that input of the specified input size to run on a single GPU (K40c) without using the proposed scheduler framework.

B. Results

a) Running Each Workload on a Single GPU Using our Framework: To understand the overheads introduced by the scheduler framework we start by studying how the framework handles individual workloads launched using the framework compared to a native launch. The scheduler framework introduces overheads that depend on the number of native kernel launches in the workload, the timeslice used by the scheduler (to determine the size and number of the micro kernel(s)), and the interaction between the micro kernel and the framework.

We experiment with three different values for the timeslice: 10 ms, 100 ms, and 1000 ms. For each workload, we record the time taken for the workload to finish its execution using the framework and the number of micro kernels launched

TABLE I: Test Programs

Name	Description	Source	Characteristic	Input	Size	Memory Footprint	Native Runtime (sec)
BFS	Breadth First Search	Rodinia [10]	Memory	Graph	$G(n,p), n = 32$ M $p = 10^{-7}$	1321 MiB	0.395
Gaussian	Gaussian Elimination		Compute	Coefficient matrix	$10.24k \times 10.24k$	883 MiB	327.45
Kmeans	Clustering Algorithm		Memory	Points as coordinates	3.5×10^6	1006 MiB	0.064
matMul	Matrix Multiplication	cudaSDK [17]	Compute	Matrices	$25k \times 20k$	5424 MiB	54.198
matTrans	Matrix Transpose		Memory	Matrix	26480×26480	5433 MiB	0.179
dxtc	Image Compression		Compute	Image	$40k \times 12.78k$ image	2280 MiB	20.75
StringSort	String Sorting	[12]	Memory	List of strings	1 GiB List	3072 MiB	0.203

through the framework across all the native kernel calls of the workload. Since the overheads can be charged to each micro kernel, for each workload, we measure the overhead per micro kernel launched. The result of this experiment is shown in Figure 5 where the primary Y-axis shows the ratio of the time spent by the workload using the framework compared to a native launch. The secondary Y-axis of Figure 5 shows the overhead per micro-kernel launched in milliseconds.

From Figure 5, we observe the following. Firstly, the average overhead introduced by the framework is small and is under 6% even at a timeslice of 10 ms. With increase in timeslice, the number of micro kernels launched per workload decreases in general. As a result, the overhead also decreases as can be observed. Further, the overhead per micro kernel is nearly uniform across workloads indicating that the framework is agnostic to the type of workload under execution. Note that for workloads whose native runtime is less than the timeslice, the framework does not introduce any additional overheads. For this reason, workloads such as BFS (for the input in Table I) do not experience change in the overheads as the timeslice is increased from 100 ms to 1000 ms.

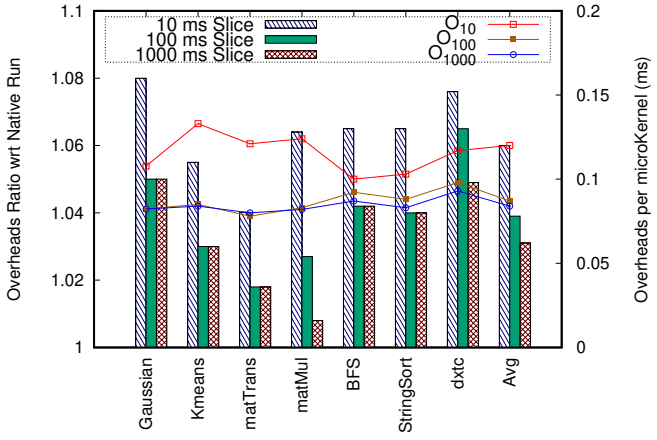
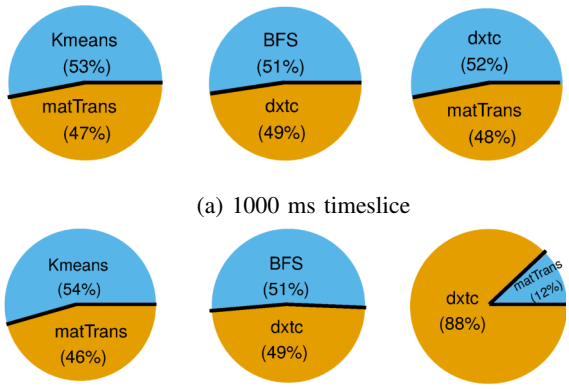


Fig. 5: Figure showing the overheads when workloads are run alone using our framework. Line labeled O_t anchored to the secondary Y-axis shows the overhead per micro-kernel launched when using a timeslice of t millisecond.

b) Running Two Workloads on a Single GPU Using our Framework: We now consider how two workloads share a GPU using our framework. We choose the two workloads



(a) 1000 ms timeslice

(b) 500 ms timeslice

Fig. 6: Figure showing the ratio of timeslices consumed by workloads in a pair of memory intensive, compute intensive, and memory intensive-compute intensive workloads running simultaneously in our framework. The timeslice used in part (a) is 1000 ms and the timeslice used in part (b) is 500 ms.

in three ways: both workloads being compute intensive, both workloads being memory intensive, and one workload being compute intensive and the other memory intensive. Note that memory intensive workloads generally have a huge memory footprint compared to compute intensive workloads. For the chosen combinations involving a memory intensive workload Equation 1 gives us a lower bound close to 1000 ms for the timeslice (Biggest Memory allowed for each program is 50% of the GPU ie. S_{prev} & S_{next} is approximately 5 GB respectively for NVIDIA K40c. Recorded average PCI bus speed for the experimental setup is 10 GB/s). This timeslice is chosen for one set of experiments. A timeslice of 500 ms is used in order to show the change in slice ratio when a timeslice lower than the prescribed lower bound is used. We study the ratio of timeslices distributed to the two programs as they run simultaneously.

The result of this experiment is shown in Figure 6. From Figure 6, we observe the following. When using 1000 ms timeslice (Figure 6a), the slice ratios for all the pairs are nearly 50% for each workload. When using 500 ms timeslice (Figure 6b), the slice ratios for all except dxtc-matTrans pair are nearly 50 %. This suggests that Equation 1 is a good guide to choose the timeslice.

For the dxtc-matTrans pair, note that matTrans is a mem-

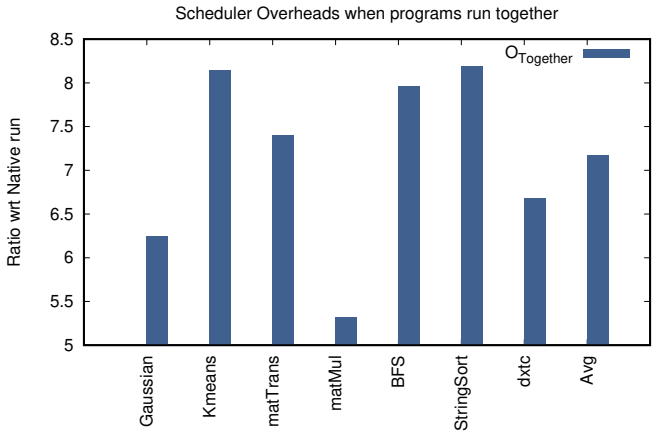


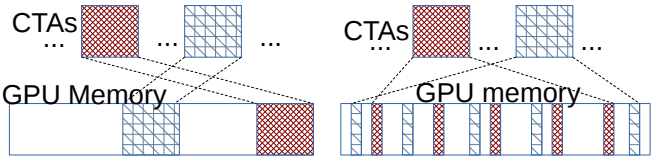
Fig. 7: Figure showing the overheads when all the seven workloads from Table I are run together in our framework.

ory intensive workload whereas dxtc is a compute intensive workload. With a timeslice of 500 ms, the memory transfer for the matTrans workload will not finish. Our framework therefore offers an additional timeslice to the dxtc workload. This allows our framework to hide memory transfers with compute. Hence, the dxtc workload consumes 88% of the slices while the two programs are running together. For the pair of memory intensive workloads with a 500 ms timeslice, each workload gets to use an effective timeslice of 1000 ms.

c) Running All Workloads on a Single GPU Using our Framework: To understand the overheads experienced by each program with respect to their native runtime, we run all the workloads together using the scheduler framework. These workloads now share the GPU in a cooperative multi-tasking manner and are scheduled in a round robin fashion. In this experiment, we use a timeslice of 1000 ms. The result of this experiment is shown in Figure 7.

From Figure 7 we observe that the overheads vary across workloads for the following reasons. The BFS workload from [10] issues 28 kernel calls in the native setting (for the input used in Table I). The average time spent by each kernel call in the native setting is close 1 ms. This means that every kernel call can utilize only 1 ms of the timeslice and relinquish the GPU resource. However, as the scheduler uses round-robin policy to assign timeslices to workloads, the BFS workload will get another timeslice at the end of six slices. For this reason, the BFS workload suffers a higher overhead. Moreover, as each native kernel call undergoes a registration process in our framework, the overheads tend to be higher. A similar phenomenon is observed in the case of workloads such as Gaussian, Stringsort, kmeans, and dxtc.

The matMul workload has only one kernel call in the native setting and a large runtime. When this workload is run together with all the workloads listed in Table I, the matMul workload get to utilize all the timeslices once the other workloads finish. Therefore, the matMul workload has a small overhead.



(a) An example of block coalesced memory access by a kernel.

(b) An example of non-block coalesced memory access by a kernel.

Fig. 8: Illustrative example of Memory footprint by a kernel

IV. Multi-GPU Support

When there are multiple GPUs in a system, our scheduling framework seamlessly uses them to execute microkernels from different processes in parallel. The Kernel Control Block(KCB) is suitably modified to keep track of which kernels are mapped to which devices. We thus achieve inter-kernel parallelism where the kernels are from different processes. In the next section, we present how we achieve intra-kernel parallelism when a kernel has nice memory access patterns.

A. Intra-kernel Parallelism

If there are N GPUs and M processes where $M < N$, then some GPUs would remain idle. In order to increase GPU utilization, we extend our basic scheduling framework such that multiple microkernels from within a process can be launched in parallel on multiple GPUs. We call this as intra-kernel parallelism. This way even when there is only a single kernel in the system, the available multiple GPUs can be put to use by executing multiple microkernels in parallel. For example, Figure 9 shows that speedup of different kernels when run alone on system with two GPUs. We provide more details on this experiment in Section IV-B.

Theoretically, a CTA can read and write arbitrary GPU memory location, which means that all the memory must be available to each CTA and hence each GPU. This would lead to two things; firstly, there would be double allocation of data elements in each GPU, and secondly, partial output elements from each GPU need to be merged into a single output element at the end, since now each one has their own version of the output. This would increase the effective kernel memory footprint. Generally, any GPU kernel's memory accesses by CTAs can be classified under one of the following patterns:

- Block Coalesced: Reads and writes here are strictly block aligned which means memory location(s) to be read/written by a thread is computed using its unique global ID which is a linear function based on its blockId and threadId. There is just one group of contiguous reads/writes by a blockId.
- Non-Block Coalesced: This kind of a transaction footprint is characterized by no apparent pattern to reads/writes by a CTA. Any thread in the CTA can read/write anywhere without any sense of grouping based on their blockId.

Figure 8 illustrates these two different memory access patterns. For block coalesced memory accesses, we introduce a one-

to-one map function which, given a global BlockId, returns the start address of the data blocks it accesses and the size of the contiguous space used by the CTA. Although this mapping function can be automatically extracted through program analysis, at this point we expect the programmer to provide this information as a metadata during the compilation process. Using this mapping function, the block of memory that is required by a microkernel is determined and only that block is moved to the respective GPU global memory.

B. Experiments with Multiple GPUs using our Framework

We now experiment with using multiple GPUs in our framework. We start with running each workload on a platform that has two K40c GPUs attached to a single CPU. Our framework assigns microkernels of the workload to both the GPUs and does the necessary book-keeping for this purpose. To understand the overhead of this bookkeeping, we first experiment with individual workloads running on our multi-GPU platform using the framework. The timeslice used is 500 ms. We compare the time taken by the workload to finish execution in our framework with the corresponding native runtime as listed in Table I. The results of this experiment are shown in Figure 9.

Figure 9 plots the speedup experienced by each workload running on two GPUs as compared to its native runtime on a single gpu. As can be observed from Figure 9, the time taken on two GPUs is on average 1.65x faster than the native runtime on a single gpu compared to an ideal value of 2x. The sources of overheads includes a registration overhead per native kernel call and cost of book keeping back end variables for each GPU. One big advantage of our framework is the ability of the framework to automatically map GPU programs written for a single GPU setting to execute on more than one GPU. Such support for automatic multi-GPU execution for a single program is studied by Ramashekar and Bondhugula [21]. However, Ramashekar and Bondhugula did not consider extending their framework to multiple workloads running in a cooperative multi-tasking manner.

We now experiment with running all the workloads from Table I on our platform consisting of two GPUs. We use a timeslice of 1000 ms. This experiment shows how multiple workloads can use our framework and share more than one GPU attached to a single CPU system. In this setting, we study how using two GPUs reduces the time taken by each workload to finish its execution. In Figure 10 we show the speedup experienced by each workload using our framework with two GPUs as compared to that of using our framework with one GPU. As can be noted, most programs experience a speedup of 1.55x on the two GPU setting while the ideal speed up should be a factor of two. The difference can be attributed to several reasons. Firstly, each native kernel call inside a program warrants a new registration phase with the scheduler. It also puts the program at the end of the queue. Secondly, memory transfers happen for all the GPUs simultaneously over the same PCI lane, therefore, there is a decrease in the transfer speed per GPU. Further, workloads such as Gaussian have a lot of kernel calls (20K for our input size in Table I) which offsets the speedup with repeated registration overhead.

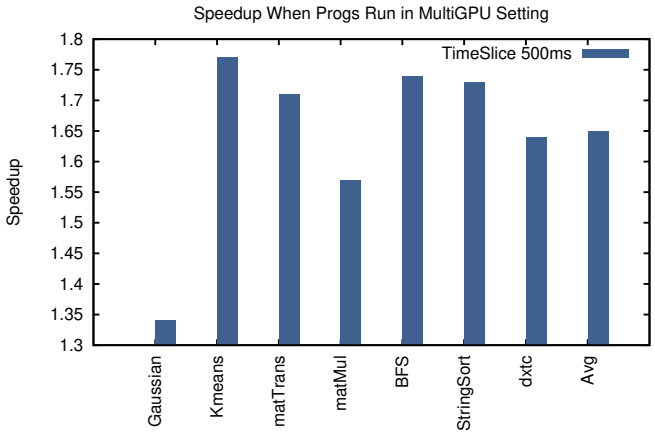


Fig. 9: Speedups in workload runtime when run individually on two GPUs using our scheduler framework.

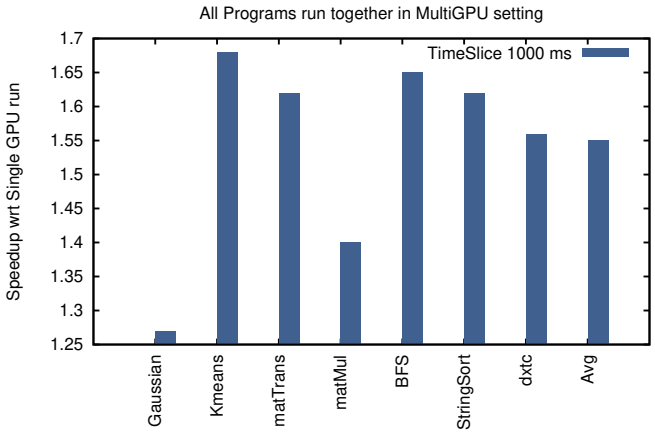


Fig. 10: Speedup for each workload when all the workloads from Table I are run together on a system with two K40c GPUs compared to running on a single K40c GPU. Both the runs are in a cooperative multi-tasking manner using our framework.

V. Conclusion and Future Work

In this work, we present an enforced cooperative multitasking based approach to support a multi-program abstraction on GPUs including coarse-grained memory management. Our framework consists of a scheduler and a source-to-source translator which automatically augments a source program to make it scheduler compliant. When there is more than one GPU available, our scheduler uses them all. Further, microkernels within a kernel can also be launched in parallel if they exhibit suitable memory access patterns. In future, we wish to extend our framework to support heterogeneous and multi-node installations. We also wish to study how the concept of microkernels can be used in other resource-

constrained settings such as mobile devices.

REFERENCES

[1] The Top500 Supercomputer Sites, https://www.top500.org/
[2] Nvidia Thrust, Available at https://docs.nvidia.com/cuda/thrust/index.html
[3] cusparse library, https://docs.nvidia.com/cuda/cusparse/index.html
[4] Nvidia Corporation. http://docs.nvidia.com/cuda
[5] Nvidia. NVIDIAs Next Generation CUDA Computer Architecture Kepler GK110. 2012.
[6] P. Aguilera, K. Morrow, N. S Kim. Fair share: Allocation of GPU resources for both performance and fairness, 32nd IEEE International Conference on Computer Design (ICCD), pp: 440–447, 2014.
[7] D. I. Arkhipov, D. Wu, K. Li, and A. C. Regan. Sorting with GPUs: A Survey, in arXiv:1709.02520.
[8] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC) , Article 12, 10 pages.
[9] J. Calhoun and H. Jiang. Preemption of a CUDA kernel function. In 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2012.
[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, K Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. in Proc. of IEEE Intl. Symp. on Workload Characterization (IISWC), pp: 44–54, 2009.
[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, 3rd Edition, MIT Press, 2009.
[12] A. Deshpande and P. J. Narayanan. Can GPUs Sort Strings Efficiently? in Proc. of Intl. Conf. on High Performance Computing(HiPC), pp: 305–313, 2013.
[13] G. Kim, M. Lee, J. Jeong, J. Kim. Multi-GPU System Design with Memory Networks. in Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp:484–495, 2014.
[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In Proc. NIPS, pp. 1106–1114, 2012.
[15] J. Lee, M. Samadi, S. Mahlke. VAST: The Illusion of a Large Memory Space for GPUs. Proceedings of the 23rd International Conference on Parallel Architectures and Compilation(PACT) pp: 443–454, 2014.
[16] J. Lee, M. Samadi, Y. Park, S. Mahlke. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. in Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), article:9, 2013.
[17] J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable Parallel Programming with CUDA.
[18] A. Nukada, H. Takizawa, S. Matsuoka. NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA. in Proc. of IPDPS Workshops, pp. 104–113, 2011.
[19] C. Pachorkar, M. Chaitanya, K. Kothapalli, and D. Bera. Efficient Parallel Ear Decomposition of Graphs with Application to Betweenness-Centrality. HiPC 2016: 301-310
[20] S. Pai, R. Govindarajan, M. J. Thazhuthaveetil. Preemptive thread block scheduling with online structural runtime prediction for concurrent GPGPU kernels. in Proc. of Intl. Conf. on Parallel Architectures and Compilation (PACT), pp: 483–484, 2014.
[21] T. Ramashekar, U. Bondhugula. Automatic Data Allocation and Buffer Management for Multi-GPU Machines. in Proc. of ACM Transactions on Architecture and Code Optimization (TACO), article: 60, 2013.
[22] A. Silberschatz, P. Galvin, G. Gagne. Operating System Concepts, 8 Edition, Wiley, 2009.
[23] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, M. Valero. Enabling Preemptive Multiprogramming on GPUs, in Proc. International Symposium on Computer Architecture (ISCA), pp. 193–204, 2014.
[24] H. Takizawa, K. Sato, K. Komatsu, H. Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. in Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp:408–413, 2009.
[25] Z. Wang, J. Yang, R. G. Melhem, B. R. Childers, Y. Zhang, M. Guo. Quality of Service Support for Fine-Grained Sharing on GPUs. in Proc. of Intl. Symp. on Computer Architecture (ISCA), pp: 269–281, 2017.
[26] B. Wu, X. Liu, X. Zhou, C. Jiang. FLEP: Enabling Flexible and Efficient Preemption on GPUs. in Proc. of Intl. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys. (ASPLOS), pp: 483-496, 2017.
[27] Q. Xu, H. Jeon, K. Kim, W. W. Ro and M. Annavaram. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming, in Proc. of. International Symposium on Computer Architecture (ISCA), pp. 483–496, 2016.
[28] GPGPU-Sim, http://www.gpgpu-sim.org/
[29] J. Zhong and B. He. Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling. IEEE Transactions on Parallel and Distributed Systems archive Volume 25 Issue 6, pp: 1522–1532, 2014.
[30] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, S. W. Keckler. Towards High Performance Paged Memory for GPUs. in Proc. of ACM Intl. Conf. on High Performance Computer Architecture (HPCA), pp. 345–357, 2016.