Runtime Support for Adaptive Spatial Partitioning and Inter-Kernel Communication on GPUs

Yash Ukidave, Charu Kalra, David Kaeli
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA, U.S.A.
{yukidave, ckalra, kaeli}@ece.neu.edu

Perhaad Mistry, Dana Schaa
Advanced Micro Devices
{perhaad.mistry, dana.schaa}@amd.com

Abstract—GPUs have gained tremendous popularity in a broad range of application domains. These applications possess varying grains of parallelism and place high demands on compute resources – many times imposing real-time constraints, requiring flexible work schedules, and relying on concurrent execution of multiple kernels on the device. These requirements present a number of challenges when targeting current GPUs. To support this class of applications, and to take full advantage of the large number of compute cores present on the GPU, we need a new mechanism to support concurrent execution and provide flexible mapping of compute kernels to the GPU.

In this paper, we describe a new scheduling mechanism for dynamic spatial partitioning of the GPU, which adapts to the current execution state of compute workloads on the device. To enable this functionality, we extend the OpenCL runtime environment to map multiple command queues to a single device, and effectively partitioning the device. The result is that kernels that can benefit from concurrent execution on a partitioned device can effectively utilize the full compute resources on the GPU. To accelerate next-generation workloads, we also support an inter-kernel communication mechanism that enables concurrent kernels to interact in a producer-consumer relationship.

The proposed partitioning mechanism is evaluated using real world applications taken from signal and image processing, linear algebra, and data mining domains. For these performance-hungry applications we achieve a 3.1X performance speedup using a combination of the proposed scheduling scheme and inter-kernel communication, versus relying on the conventional GPU runtime.

I. INTRODUCTION

Modern GPU workloads possess varying degrees of parallelism and demand flexible techniques to schedule multiple kernels on the device. It has become increasingly important to support effective sharing of a single GPU for parallel applications. In order to maximize the benefits of the GPUs for a multicore platform, a mechanism which provides high efficiency and elastic resource allocation is required. High Performance Computing (HPC) clusters that include GPUs have even a stronger case for exploring dynamic allocation and deallocation of compute resources when applications need to share GPU resources [5]. In these scenarios, if we want to maximize application throughput, a kernel may need to adjust its own demands in order to leave room on the device to service the needs of other kernels executing concurrently.

In this paper, we propose a novel partitioning mechanism in which the compute resources shared among different concurrent applications can be controlled dynamically. We implement a scheduler that allows concurrent execution of different kernels using multiple command queues. The scheduler changes the resource partitioning according to predetermined characteristics of the applications. This functionality is communicated to the runtime environment and does not require kernel source code modifications, thereby minimizing programmer effort.

To enable efficient data communication between concurrently executing kernels, we implement a pipe-based communication channel and examine its potential benefits. Workloads belonging to signal/image processing, data mining, and big-data domains, which include large and small sized kernels in their execution are used in our evaluation. To study the benefits of our partitioning and communication support, we consider applications where multiple data streams are processed in parallel and possess a range of data sharing requirements. We demonstrate that our adaptive partitioning scheme, combined with an inter-kernel communication channel, outperforms the conventional GPU runtime by utilizing the device in a more efficient manner.

The paper is organized in the following sections: Section II describes the motivation for our work and describes the OpenCL programming model. We describe the implementation of our proposed adaptive partitioning mechanism and pipe-based communication channel in Section III. Sections IV and V present our evaluation methodology and discuss the performance of adaptive partitioning and pipe communication channel, respectively. Related work is described in Section VI. Section VII offers conclusions and discusses future work.

II. BACKGROUND AND MOTIVATION

A. OpenCL Terminology

OpenCL is a popular programming framework for heterogeneous computing, maintained by Khronos [15]. The programs which execute on a compute device are known as kernels. An instance of a kernel is referred to as a workitem, which executes on a processing core of a compute unit on the device. Workitems are expressed in terms of independent units of execution known as workgroups. Workgroups execute on the compute units of the device and are organized as an N-Dimensional index space known as the NDRange. A set of workitems within a single workgroup, which executes a single instruction in SIMD manner on the compute unit, is known as a wavefront. Software abstractions used for passing commands from the host to the device in an OpenCL computation are known as Command Queues.

B. Concurrent Kernel Execution on GPUs

Workloads from a number of domains use multi-kernel processing on GPUs. The Madness framework utilizes multiple small-sized kernels for solving multi-dimensional integrals and differential equations on GPUs [18]. Similarly, multiple kernels of different sizes are used for Biometric Optical Pattern Matching [1] and Speeded-Up Robust Features(SURF) on GPUs [12]. This class of applications typically do not utilize the entire GPU for computation on each kernel launch. Researchers have also augmented Linux system services such
as AES encryption and garbage collection, employing the services of a GPU. The computations required by these services are launched on the GPU using multiple kernels of various sizes [10], [20]. Concurrent execution of such multi-sized kernels can improve the occupancy on the GPU. An advanced resource partitioning and scheduling mechanism is required to facilitate such concurrency on modern GPUs.

Concurrent execution of kernels on single GPU was first supported by Nvidia Fermi GPUs. A scheduling policy similar to a left-over policy, which schedules a complete kernel at a time [17], was supported on Fermi GPUs. The left-over policy only schedules the next kernel if the required number of compute units are available. Nvidia’s Kepler GPUs achieve concurrent execution of kernels using the Hyper-Q technology [16], which employs multiple hardware queues to avoid false dependencies between kernels. However, the number of concurrent kernels executing on the GPU is limited by number of available compute units and leads to a fixed partitioning of the GPU. The opportunity to schedule small kernels is lost in the presence of a dominant compute-hungry kernel. AMD GPUs also support multiple hardware queues using the Asynchronous Command Engine (ACE) [11]. This allows interleaving of kernel execution from different hardware queues, but does not allow concurrent execution of kernels. To address these issues, we introduce an adaptive spatial partitioning mechanism. Our mechanism dynamically changes the compute-unit allocation according to the resource requirements of the kernels scheduled on the device. This leads to better utilization of the GPU and avoids starvation of small-sized kernels. We design a workgroup scheduler and a partition handler mechanism to support the adaptive spatial partitioning.

Previous work has improved concurrent execution performance of two-kernel workloads on GPUs by providing fine-grain control on resource allocation [17]. But in this prior work, fine-grain control required modifications to the source code of the application kernel, which might be non-trivial for select applications. This has motivated us to develop OpenCL runtime-level support for concurrent execution using adaptive partitioning. We expose the options for adaptive and fixed partitioning to the user through the OpenCL runtime API. OpenCL 1.1 allows for the division of compute resources of the GPU to form a logical collection of compute units known as sub-devices. We use multiple OpenCL command queues and sub-device to submit workloads to the same GPU.

C. Inter-Kernel Communication on GPUs

Applications with stage-based computations are gaining popularity on GPUs. Such workloads present a high dependence on inter-kernel data communication [8], [12]. To allow real-time data communication between concurrent kernels, we implement the OpenCL pipe channel introduced in the OpenCL 2.0 specification. We evaluate the performance benefits of a OpenCL pipe channel using applications possessing a range of data sharing requirements. We implement the proposed adaptive partitioning and inter-kernel communication mechanism on a simulated AMD Southern Islands GPU using the Multi2Sim simulation framework [22].

III. SPATIAL PARTITIONING OF SOUTHERN ISLAND GPUS USING MULTIPLE COMMAND QUEUES ARCHITECTURE

We describe the proposed design of the scheduler for fixed and adaptive spatial partitioning of the GPU, which enables concurrent execution of NDRange.

A. Workgroup Scheduling Mechanism for Multiple Command Queue mapping on different Sub-Devices

To enable scheduling of workloads with multiple NDRange enqueued on multiple command queues, enhancements are made to the workgroup scheduling subsystem. These modifications are made to the scheduler of the simulated architectural model of the AMD Southern Islands (SI) GPU. The OpenCL sub-device API (clCreateSubdevices) is used to assign the requested number of compute units to the sub-device. Multiple command queues are mapped to different sub-devices to submit workloads for execution. The sub-device maintains a record of the command queues mapped to it. Figure 1 shows the described mapping scheme of multiple command queues to sub-devices. An NDRange which is enqueued for execution on a sub-device utilizes the compute units assigned to that specific sub-device. This leads to a fixed spatial partitioning of the GPU. Later, we will relax this constraint and describe a methodology for adaptive partitioning.

The workgroups of an NDRange can be scheduled on the compute units of a sub-device using our proposed workgroup scheduler, shown in Algorithm 1. The implementation uses lists to track availability of compute units from a sub-device and also to track the workgroups of the NDRange mapped to that sub-device.

1. Available CU List: List of compute units(CU) where workgroups of active NDRanges can be dispatched.
2. Pending Workgroup(WG) List: List of workgroups of active NDRanges which are yet to be assigned a CU for execution.
3. Usable CU List: List of CUs assigned to each sub-device. It is a subset of the available CU list. NDRanges mapped to a particular sub-device use CUs described in the usable list.

Our scheduler implementation can be described as a round robin scheduling scheme. Once a workgroup is scheduled to a compute unit, it’s local memory requirements and wavefront resource requirements do not change, which facilitates the use of a round robin algorithm. Algorithm 1 assumes a fixed workgroup-level granularity for populating the wavefront pools of compute units. We provide resource checking to verify if a workgroup can be mapped to a compute unit. We check the available local memory, wavefront pool entries, vector registers and scalar registers before scheduling a workgroup on a compute unit. This scheduling step differs from the
Algorithm 1 Workgroup scheduling mechanism to map workgroups to usable compute units.

Input: Usable Compute Unit (CU) Pool
Input: Active NDRanges
Output: Mapping of Workgroups to Compute Units

for all Active NDRanges do
  Get NDRange.Usable_CUs
  WG = Get NDRange.Pending_Workgroups
  while NDRange.Usable_CUs != Empty do
    k = ListHead(NDRange.Usable_CUs)
    Schedulable = Check Resources(k, WG)
    if Schedulable == True then
      Map Workgroup (k, WG)
    else
      Add WG to NDRange.Pending_Workgroups
    end if
  end while
end for

scheduling mechanism used on older GPU systems, where the resource requirements of all workgroups could be calculated prior to kernel dispatch (occupancy).

B. Adaptive Spatial Partitioning of GPU using multiple Command Queues

The command queue mapping and workgroup scheduling scheme for fixed partitioning of the GPU is described in Section III-A. This scheme can lead to resource starvation for small-sized NDRanges in applications possessing multiple NDRanges which can execute concurrently. We extend the fixed spatial partitioning mechanism to support adaptive partitioning of resources (i.e., compute units) across different sub-devices. The OpenCL sub-device API (clCreateSubDevices) is modified to support two new flags that specify: (1) Adaptive or (2) Fixed partitioned sub-device. The compute units, and the NDRanges that belong to an adaptive/fixed sub-device, are referred to as adaptive/fixed compute units and adaptive/fixed NDRanges, respectively. This added support for adaptive partitioning allows the runtime to allocate compute units to sub-devices based on the size of NDRange.

Figure 2 presents our handler for adaptive partitioning on a GPU. The adaptive partitioning handler is invoked only whenever a new NDRange is scheduled for execution or an active NDRange completes its execution on the GPU. The adaptive handler comprises of three modules: (1) a Dispatcher, (2) a NDRange Scheduler, and (3) a Load Balancer.

- **Dispatcher:** The dispatcher maintains a Pending List of NDRanges queued for execution on the device. If the pending list is not empty, the NDRange which is at the head of this list is scheduled for execution. If the pending list is empty, the Load Balancer is invoked to reassign resources to the active NDRanges.

- **NDR Scheduler:** The NDR scheduler is run by the dispatcher whenever an NDRange has to start execution on the device. For a sub-device with the fixed property type, the scheduler checks for available compute units from the free list, as well as from the adaptive compute units. If the number of available compute units is less than the number requested, the associated NDRange is added to the pending

---

Fig. 2. Flowchart describing the mechanism of handling adaptive partitioning of a GPU in the OpenCL runtime.

Load Balancer

- Load-Balancer: The Load-Balancer manages resource allocation for all of the adaptive NDRanges on the device. It forms sets of compute units proportional to the size of each adaptive NDRange. After mapping compute units to the adaptive NDRanges, the load balancer executes the workgroup scheduler as described in Algorithm 1. The load-balancer prevents pre-emption of an active NDRange by maintaining at least 1 compute unit for each active adaptive NDRange. An incoming adaptive NDRange is added to the pending list to avoid pre-emption of an active adaptive NDRange. Our mechanism ensures that there is at least 1 compute unit available for each adaptive NDRange executing on the device. A long running adaptive NDRange can be terminated to prevent blocking of resources. Such termination is done using a timer, which can be configured by the user. The default value of timer is set to 10 billion GPU cycles.

Figure 3 demonstrates the adaptive partitioning policy for two adaptive NDRanges (NDR#0 and NDR#1) mapped to
scheduling. In scheduling, the scheduler iterates over the compute units of
adaptive and closely-coupled applications, which demand a heterogeneous platform to effectively run a wider range of different sub-devices. This opens up the possibility of utilizing a flexible mapping scheme allows us to control compute resources allocated to sub-devices for mapping different NDRanges of an application. The hybrid partitioning policy uses a combination of adaptive and fixed resources. This mechanism assures maximum occupancy of the compute device and can help preserve resources. In latency-based scheduling, the scheduler iterates over the compute units in a round-robin fashion and assigns one pending workgroup to each usable compute unit. The scheduler does not consider whether the compute units are fully occupied. The mapping process continues until all pending workgroups are assigned. This mechanism ensures that each usable compute unit does useful work on each GPU cycle and minimizes compute latency.

We also introduce three different partitioning policies: 1) full-fixed, 2) full-adaptive and 3) hybrid partitioning. The full-fixed partitioning policy maps the NDRanges of each application to a single compute unit, and the maximum allowable number of workgroups per compute unit are computed, or when the compute unit spends all of its resources. This mechanism assures maximum occupancy of a compute device and can help preserve resources. The full-fixed partitioning policy maps the NDRanges of each application to a single compute unit, and the maximum allowable number of workgroups per compute unit are computed, or when the compute unit spends all of its resources. This mechanism assures maximum occupancy of a compute device and can help preserve resources. The hybrid partitioning policy uses a combination of adaptive and fixed sub-devices for mapping different NDRanges of an application.

D. Continuous Processing Using OpenCL Pipes

The proposed spatial partitioning and workgroup scheduling scheme allows us to control compute resources allocated to different sub-devices. This opens up the possibility of utilizing heterogeneous platforms to effectively run a wider range of adaptive and closely-coupled applications, which demand a high degree of communication [8, 23]. In such workloads, even though the enqueued kernels utilize the same memory space, they will still need to be stopped and restarted often to support synchronization and exchange of data.

Next, we describe the implementation of pipe-based communication channel between different sub-devices. Pipes allow communication between different NDRanges executing simultaneously on the same GPU. A pipe is a typed memory object which maintains data in a FIFO manner. The pipe object is created and initialized by the host. Pipe object stores data in the form of packets. The size of a pipe is calculated based on the type of pipe memory and number of packets requested during creation. Access to the pipe is restricted to the kernels executing on the device (i.e., GPU) and cannot be updated by the host. Memory transactions on the pipe object are carried out using OpenCL built-in functions such as read_pipe and write_pipe. A pipe object is passed to the kernel as a read-only memory object. Multiple pipes having different access permissions can be accessed in the same kernel. The pipe object is implemented using a buffer which is modeled as a queue. Due to the unavailability of compiler frameworks for OpenCL 2.0, we implement built-in functions of the pipe operations as user functions in the kernel.

The implementation model of the pipe is shown in Figure 4. The producer kernel and consumer kernel can be mapped to the same or different sub-devices. The state of the data in the pipe object is maintained until the pipe object is released by the host. Changes to the state of the pipe are visible to all kernels accessing the pipe in the same OpenCL context.

The communication procedure for applications utilizing pipes is also shown in Figure 4. Such applications perform computation on contiguous chunks of data known as tiles. The producer kernel processes the input data and stores the output in an intermediate buffer. The data/tile ID of the processed data is written to the pipe by the producer kernel. The consumer kernel extracts the data/tile ID from the pipe and accesses data from the intermediate buffer using the extracted ID as an offset into the buffer. Both of the kernels can execute on separate command queues mapped to different sub-devices on the GPU and maintain a real-time communication channel.

IV. EVALUATION METHODOLOGY

A. Platform for Evaluation

Our evaluation requires fine-grained control over workgroup scheduling, NDRange creation and support for OpenCL sub-devices on the targeted platform. We have implemented

1 synchronization between kernels only happens at kernel boundaries
multiple command queue mapping, a new workgroup scheduler, an adaptive partitioning handler and an OpenCL pipe mechanism in Multi2Sim, a cycle-level heterogeneous platform simulator [22]. Multi2Sim is a publicly available architectural simulation framework targeting x86, AMD GPUs and NVIDIA GPUs. Multi2Sim provides software layers defined as “runtime” and “driver”, to execute OpenCL applications on different simulated GPUs. The runtime layer provided by Multi2Sim includes our library that extends the OpenCL API. Support for multiple command queues and sub-devices are added within the OpenCL runtime layer. The resource usage tracking within the compute units and information regarding the runtime requirements of each workgroup is added to the driver layer. To execute applications on Multi2Sim, we link our program with the Multi2Sim OpenCL runtime library. We simulate the AMD Southern Islands (SI) Radeon 7970 GPU model in the results presented here. The details of the configuration of this simulated GPU is shown in Table I.

### Table I. The device configuration of the Southern Islands GPU.

<table>
<thead>
<tr>
<th>Compute Unit Config</th>
<th>Memory Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td># of CU’s</td>
<td>32</td>
</tr>
<tr>
<td># of Wavefront Pools / CU</td>
<td>4</td>
</tr>
<tr>
<td># of SIMD Units / CU</td>
<td>4</td>
</tr>
<tr>
<td># of lanes / SIMD</td>
<td>16</td>
</tr>
<tr>
<td># of scalar reg / CU</td>
<td>64K</td>
</tr>
<tr>
<td># of shared L2</td>
<td>2K</td>
</tr>
<tr>
<td>Frequency</td>
<td>1GHz</td>
</tr>
<tr>
<td>L1 (1 CU)</td>
<td>16KB</td>
</tr>
<tr>
<td>L2</td>
<td>128KB</td>
</tr>
<tr>
<td>Global Memory</td>
<td>1GB</td>
</tr>
<tr>
<td>Local Memory / CU</td>
<td>64K</td>
</tr>
</tbody>
</table>

We also study the behavior of multiple command queue-based applications on devices with Nvidia HyperQ technology. The evaluation for this study are performed on a real Nvidia Tesla K20c GPU.

### B. Evaluated Benchmarks

We begin by describing the two sets of benchmarks used in this study. The first set is used to evaluate the different partitioning policies. The second is used for evaluating pipe-based communications. Existing suites such as Rodinia and Parboil are not appropriate for our study since they only utilize single command queue mapping [3], [19].

#### Set 1: Multiple command queue mapping

1. **Matrix Equation Solver(MES):** A Linear solver implemented on GPUs [18]. The benchmark evaluates the equation $C = (A^{-1}B) * \beta((A + B) * B)$, where $A$, $B$, and $C$ are square matrices. The computation is done in three parts:

   **Kernel 0:** $C_0 = (A^{-1}B) !$
   **Kernel 1:** $C_1 = \beta((A + B) * B) !$
   **Kernel 2:** $C = C_0 * C_1$.

   Kernel#0 and Kernel#1 execute in parallel using different sub-devices. Kernel#2 is computed using the entire GPU.

2. **Communication Channel Analyzer(COM):** Emulates a communication receiver with 4 channels. Each channel is assigned a separate command queue mapped to different compute units. Each channel performs a unique signal scrambling computation on the same input data.

3. **Big Data Clustering(BDC):** A big-data analysis application uses three computational kernels. Each kernel is mapped to a different command queue.

**Kernel 0:** Performs clustering of input, involving all input data points.

**Kernel 1:** Reduction kernel, to reduce each of 20 clusters to 100 representative points.

**Kernel 2:** Sorting computation to sort the representative points. All three kernels have different NDRange sizes, Kernel#0 being a large sized persistent kernel.

#### Set 2: Pipe-based Communication

1. **Audio Signal Processing(AUD):** This is a two-channel audio signal processing application, providing three different stages of compute. The three stages are: (1) a FFT (Fast Fourier Transform) (2) a FIR (Finite Impulse Response) filter and (3) a Gaussian Noise Elimination [14]. Execution is pipelined, with each stage providing data to the next for processing. Each stage of compute runs in a separate command queue mapped to a different compute unit. The data communication between stages is achieved using pipe objects.

2. **Search-Bin Application(SBN):** The search benchmark, described in Set 1, is enhanced to perform bin allocation with the search benchmark [13]. This application uses two kernels to search through an input data set. The third kernel assigns the searched data to different bins. The search kernel provides data to the bin kernel using a pipe object. All kernels are mapped to different command queues for concurrent execution.

### V. Performance Results

#### A. Performance Enhancements due to Multiple Command Queue Mapping

We evaluate the execution performance of the Set 1 benchmark applications when applying multiple command queue mapping for a fixed-partition simulated AMD device and run on an actual Nvidia K20 GPU device with HyperQ. The evaluation with single queue mapping is the baseline for evaluating speedup due to multiple command queues on both platforms. The execution performance is shown in Figure 5.

COM enqueues 4 kernels to the same device by mapping each kernel to 8 compute units on the SI GPU. The MES application also enqueues 2 kernels by mapping each of them to 16 compute units on the AMD Southern Islands device. Both applications experience a performance gain of 2.9x over the single-queue implementation. Applications using multiple command queues on the fixed-partition SI device exhibit an average performance speedup of 3.1x over the single command queue. Concurrent execution of kernels tries to overlap computation of each kernel and reduces execution time of the application.
The applications using multiple command queues on the K20 GPU show an average performance gain of 2.2x over a single queue implementation. The multiple command queues on the Kepler GPU get mapped to different hardware queues using the HyperQ feature and are executed concurrently. We can see significant benefits for applications consisting of kernels which do not saturate GPU resources. The applications with all kernels optimized to saturate the GPU would result in marginal performance improvements.

B. Effective Utilization of Cache Memory

We explore the performance of the L2 cache on the simulated SI GPU using multiple command queues. Our L2 cache study is motivated by the fact that the L2 is the first shared memory resource (L1s are private on a GPU). All compute units are configured to have one L1 cache and share six L2 caches. Applications using multiple command queues execute on different sets of compute units. Kernels which utilize the same working set, benefit from the L2 caches shared across compute units.

The L2 cache efficiency is evaluated in Figure 6. Applications such as MES, COM and BDC show a 22% improvement in the L2 cache hit rate when using multiple command queues. As the global writes are visible to the L1 and L2 caches on SI GPUs, the single command queue implementation flushes the L2 cache after completion of the kernel [7]. Such L2 cache flushes do not occur when using multiple command queues. This improves the cache hit-rate of concurrent kernels operating on the same input data. SER and TEX enqueue kernels which operate on different input data. Hence, the kernels do not access the same data and exhibit a cache hit-rate similar to their single command queue implementation. BDC schedules three different kernels which use the same data buffers, and hence shows an improvement in cache hit rate.

C. Effects of Different Workgroup Scheduling Mechanisms and Partitioning Policies

Next, we evaluate the performance of the workgroup scheduling and partitioning policies described in Section III-C.
and TEX show an improvement in execution performance of 35.7% and 23%, respectively (as compared to full-adaptive partitioning). Hybrid partitioning produces a compute unit assignment similar to full-fixed partitioning for applications which use similar-sized NDRanges for computations. Thus, MES and SER (which have similar-sized NDRanges) do not receive any significant advantage in performance when using hybrid partitioning. In contrast, COM shows a performance improvement of 16.6% over adaptive partitioning when using hybrid partitioning. This improvement is attributed to the improved load-balancing of the adaptive NDRanges in the COM application. Assigning large NDRanges to adaptive sub-devices in hybrid partitioning may lead to over-subscription of that sub-device and thereby degrading the performance of the large NDRange.

Latency-based scheduling yields the best performance when using full-fixed and hybrid partitioning policies, as seen in Figure 7b. The compute units assigned to the fixed sub-device are not re-assigned during execution. This enables latency-based scheduling to use all compute units as effectively as possible in each fixed sub-device when applying hybrid and full-fixed partitioning. Latency-based scheduling experiences some degradation in performance for full-adaptive partitioning. The first NDRange scheduled for execution occupies the entire GPU and the subsequent NDRanges do not receive any free compute units for scheduling the computation. This leads to a delay due to re-assignment of compute units from the first NDRange in the case of full-adaptive partitioning. The first NDRange of the BDC application is a large persistent kernel and experiences a 96% increase in execution time (cycles) versus using full-fixed partitioning and latency-based scheduling. All other applications observed an average increase of 45.2% in execution time (cycles) for full-adaptive partitioning using latency-based scheduling.

D. Timeline Describing Load-Balancing Mechanisms for the Adaptive Partitioning Policy

The timeline shown Figure 8 describes the allocation and deallocation of compute units to different NDRanges for the applications, when using full-adaptive partitioning policy. NDR0 (i.e NDRange 0) is the first to be executed on the GPU for all of the applications. The sampling is done at intervals of 5000 GPU cycles. NDR0 arrives when the GPU is unoccupied, and hence, receives a large number of compute units (20 CUs for COM, 22 CUs for SER, and 22 CUs for TEX). As new NDRange arrive on the device, the load-balancer re-assigns the compute units. An equal number of compute units are assigned to same-sized NDRanges in COM and SER. As observed in Figure 8b, the NDRange of different sizes in TEX are allocated with different number of compute units when the application reaches a steady-state. The load-balancer re-assigns the compute units whenever an NDRange completes execution.

As seen in Figure 8a, NDR2 and NDR3 in COM are assigned additional compute units as NDR0 and NDR1 complete their execution. The same effect is observed for NDR1 of the SER application and for NDR1 and NDR2 of the TEX application. The average latency observed to re-assign one compute unit between two NDRange for all the applications is 9300 cycles.

E. Performance Evaluation of using Pipe Memory Objects

We evaluate the execution speedup and L2 cache behavior of the Set 2 benchmarks, which utilize the pipe-object for inter-kernel communication. The baseline in these experiments assumes a single command queue for the OpenCL kernels. Figure 9 shows the execution performance. Both applications transfer shared data using the pipe object between stages of computation. A 3.7x speedup is observed for multiple command queues over the single command queue implementation. Due to data dependencies present between stages, the single command queue implementation completes all updates to the dependent data before enqueueing the next kernel. The pipe object allows such kernels to overlap computation and communication effectively. Figure 10 shows the L2 cache performance for Set 2 applications using a pipe object. We observe a 35% increase in the L2 cache hit rate for applications using pipe objects. The improved hit rate is attributed to the number of accesses that use the data buffers shared across concurrently executing kernels.
VI. RELATED WORK

Next, we discuss related work on multiple command queues, relevant workloads and programming frameworks.

Gregg et al. examine concurrency on GPUs [6] by combining kernels from multiple CUDA streams without architectural support. They show that resource allocation of concatenated kernels limits performance. We extend their evaluation by independently scheduling the workgroups of each kernel separately. Tanasic et al. introduce flexible spatial partitioning of GPU resources to support preemption on GPUs [21]. Their partitioning mechanism requires explicit assignment of CUs to kernels, which is then modified in presence of additional workloads. Our approach provides the choice of using explicit assignment of CUs, or runtime controlled assignment with superior load-balancing. We allow the use of a hybrid policy which utilizes both explicit and runtime assignments at the same time for improving execution performance. Lustig et al. examine host-device interaction and propose memory improvements to allow for greater overlap between computation and data movement [9]. Our work compliments Lustig et al. by adding additional command queues, allowing for overlapped processing. Boyer et al. demonstrate dynamic load balancing of computation shared between heterogeneous devices [2]. Real-time and adaptive applications can benefit from mapping multiple queues to a device [8], [23]. Persistent kernel programming [20], which is an upcoming style of programming heterogeneous devices, allows different paths of a branch to be executed by each workgroup. By providing spatial partitioning of the GPU and implementing OpenCL pipes, we improve programmability of such applications.

VII. CONCLUSIONS

In this paper, we demonstrate the benefits of mapping multiple command queues on a GPU. We have studied a range of scheduling policy trade-offs related to partitioning of the GPU for concurrent execution of multiple NDRange. We have also evaluated the effects of different workgroup scheduling mechanisms. Tighter control over the workgroup-to-compute unit mapping results in significant speedups (up to 3x over a single command queue implementation). Different partitioning mechanisms can enable the user to select an appropriate policy to suit their application. A developer with no inherent knowledge of the device architecture can rely on the adaptive partitioning for their application. Developers with a better understanding of the device architecture can select the hybrid or fixed partitioning policy to tune their application for maximum performance. We have also evaluated the performance and programmability benefits of a communication pipe between OpenCL kernels executing on different compute units on a device.

In future work, we plan to consider spatial affinity between compute units and command queues in our scheduler and will incorporate this into a microarchitectural model for a workgroup scheduler. We plan to extend our study of pipe objects to shared memory APU systems, providing shared virtual memory between CPU and GPU devices.

VIII. ACKNOWLEDGEMENT

We would like to thank AMD for their support and Nvidia for providing hardware to conduct this work. This work was supported in part by a NSF CISE award CSR-1319501.

REFERENCES


