Oversubscribed Command Queues in GPUs

Sooraj Puthoor
AMD Research
Sooraj.Puthoor@amd.com

Xulong Tang
Penn State
xzt102@cse.psu.edu

Joseph Gross
AMD Research
Joe.Gross@amd.com

Bradford M. Beckmann
AMD Research
Brad.Beckmann@amd.com

Abstract

As GPUs become larger and provide an increasing number of parallel execution units, a single kernel is no longer sufficient to utilize all available resources. As a result, GPU applications are beginning to use fine-grain asynchronous kernels, which are executed in parallel and expose more concurrency. Currently, the Heterogeneous System Architecture (HSA) and Compute Unified Device Architecture (CUDA) specifications support concurrent kernel launches with the help of multiple command queues (a.k.a. HSA queues and CUDA streams, respectively). In conjunction, GPU hardware has decreased launch overheads making fine-grain kernels more attractive.

Although increasing the number of command queues is good for kernel concurrency, the GPU hardware can only monitor a fixed number of queues at any given time. Therefore, if the number of command queues exceeds hardware's monitoring capability, the queues become oversubscribed and hardware has to service some of these queues sequentially. This mapping process periodically swaps between all allocated queues and limits the available concurrency to the ready kernels in the currently mapped queues. In this paper, we bring to attention the queue oversubscription challenge and demonstrate one solution, queue prioritization, which provides up to 45x speedup for NW benchmark against the baseline that swaps queues in a round-robin fashion.

CCS CONCEPTS

• Computer systems organization~Single instruction, multiple data • Software and its engineering~Scheduling

ACM Reference format:

S. Puthoor, X. Tang, J. Gross and B. Beckmann. In *GPGPU-11: In GPGPU-11: General Purpose GPUs, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 11 pages.
DOI: https://doi.org/10.1145/3180270.3180271

1. Introduction

GPUs are continually increasing their computational resources with the latest offerings from AMD and NVIDIA boasting impressive performance improvements over their previous generations [1][41]. With this increase in hardware resources, there is a demand to run a more diverse set of applications, such as machine learning, cloud computing, graph analytics, and high performance computing (HPC) [42][45][46][51]. While some of these workloads can launch a single kernel large enough to completely consume the entire GPU, many others rely on concurrent kernel launches to utilize all available resources.

For graphics workloads, the benefits of using concurrent kernels has been well documented [1][60]. By running multiple rendering tasks (a.k.a. kernels) concurrently, these tasks can share the available resources and increase utilization, leading to faster frame rate within the same power budget. Complementary, there have been previous works in the HPC domain that have also increased GPU utilization by running multiple kernels concurrently [52][56].

To run multiple kernels simultaneously, existing bulk-synchronous applications are often refactored to use asynchronous tasks. For these task-based implementations, their execution is typically represented as a Directed Acyclic Graph (DAG) with nodes of the DAG representing tasks and the edges representing dependencies between tasks. From the perspective of a GPU, a task is an instance of a GPU kernel with its associated arguments. As the task size decreases, GPU utilization usually increases because smaller tasks are ready to launch when a smaller fraction of data and resources become available versus larger tasks. The idea is similar to filling an hour glass with sand versus marbles. With sand, the hour glass empties faster because the fine-grain particles occupy the available free space at the narrow neck more quickly.

Of course, the benefit from running fine-grain tasks depends on amortizing the launch overhead. This can be done by both reducing the latency for an individual kernel launch and by allowing multiple kernel launches simultaneously. To reduce kernel launch latency, the Heterogeneous System Architecture (HSA) allows applications to directly enqueue work into user-mode queues and avoids heavy weight operating system (OS) or GPU driver involvement [28][47]. To allow simultaneous kernel launches, HSA allows applications to allocate multiple user-mode queues which hardware can service simultaneously. Compute Unified Device Architecture (CUDA) supports a similar feature by allowing applications to allocate multiple streams [17]. In the extreme case, the concurrency exposed to hardware is only limited by the virtual memory available to queue allocation.

While hypothetically more queues expose more concurrency, it is not feasible to build a hardware that can simultaneously monitor many queues. Instead, when the queues created by the application exceed the number of queues can be monitored by the hardware, the queues are oversubscribed and the GPU must employ a procedure to ensure all queues are eventually serviced. For instance, periodically swapping between all queues at regular intervals is one obvious solution.

Processing task graphs using this queue swapping mechanism can cause performance challenges. For example, when tasks in mapped queues are waiting for tasks in unmapped queues, GPU utilization can suffer significantly. While hardware will eventually map the queues with ready tasks ensuring forward progress, the delay caused by unintelligent queue mapping leads to significant performance degradation and the degradation becomes worse with more complex task graphs.

GPGPU-11, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5647-3/18/02…$15.00
https://doi.org/10.1145/3180270.3180271

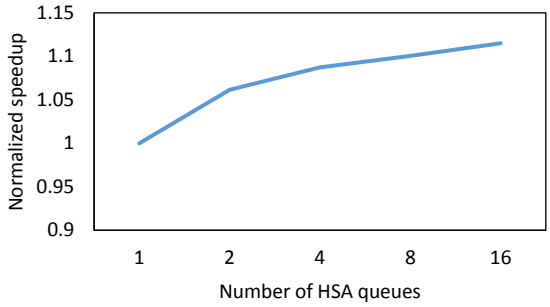

Figure 1. NW application speedup with multiple HSA queues relative to single HSA queue.

In this paper, we bring to attention this overlooked queue oversubscription challenge and show that a simple queue scheduling policy guided by prioritization can effectively mitigate oversubscription and improve the GPU performance.

Specifically, the contributions from this paper are:

- To the best of our knowledge, we are the first to highlight the largely underappreciated scheduling challenge with oversubscribed GPU command queues and the first to detail their scheduling mechanisms.
- We show that a simple priority-based scheduling policy can reduce the underutilization caused by command queue oversubscription and increase performance by 91x versus a naïve round-robin policy and 45x and 43x versus more optimized policies that do not map empty queues and immediately unmap queues when they become empty.
- We show that the proposed priority-based scheduling is particularly beneficial for fine-grain tasks and they achieved up to 11% speedup versus coarse-grain or medium-grain tasks.

The rest of the paper is organized as follows. Section 2 discusses the background on GPU command queue oversubscription and Section 3 introduces the command queue organization. Priority-based queue scheduling schemes are discussed in detail in Section 4. Section 5 discusses simulation methodology and Section 6 evaluates different queue scheduling policies. Finally, Section 7 reviews related work and Section 8 concludes the paper.

2. Background

Programmers define the parallel portions of applications as kernels and offload them to GPUs using command queues. Then GPUs monitor these command queues and execute the submitted kernels. As GPUs integrate more compute resources, concurrently executing multiple kernels can more efficiently utilize those available resources [1][17]. While CUDA requires multiple streams to achieve kernel concurrency, HSA can launch simultaneously executing kernels from a single queue [17][28]. HSA can further increase kernel concurrency with multiple queues by avoiding the sequential processing overhead of launching kernels from same queue [47]. Figure 1 shows the speedup of NW application with multiple HSA queues relative to a single HSA queue. The fine-grain version of NW has 64 tasks and even for this simple task graph, multiple HSA queues provide speedup.

While multiple command queues can increase the concurrency exposed to GPUs, traditional applications use bulk synchronization that lack multi-kernel concurrency. Meanwhile, structuring applications as tasks represented by a DAG has been proposed to increase concurrency [14]. In the DAG-based approach, the entire

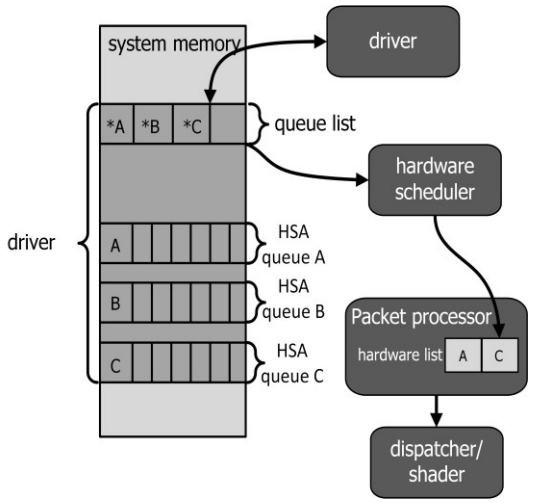

Figure 2. GPU queue and task scheduling hardware.

computation is represented by a task graph with nodes in the DAG representing tasks and edges representing the dependencies between these tasks. Once the application is represented in its DAG form, programming APIs allow it to be directly exposed to hardware.

HSA provides features to expose and process task graphs [27][28][29]. HSA-compatible systems, including commercially available AMD APUs [10][23], support hardware features, such as shared coherent virtual memory between different *agents* (CPU or GPU devices), and provide a low-level runtime API with extensive tasking capabilities. The tasking capabilities of HSA-compatible hardware have been evaluated in the past, highlighting its performance improvements [47]. HSA provides *kernel dispatch packets* that can be used to launch kernels on HSA agents and *barrier packets* that can be used to enforce dependencies between these kernels. Any task graph can be represented with these two packets by replacing tasks with kernel dispatch packets and edges with barrier packets. These packets are then enqueued to HSA queues. Once a kernel finishes execution, the corresponding barrier packets are processed and the dependent kernels can be launched. Additionally, since HSA queues are created in user space, there is no expensive OS involvement while processing packets, which significantly reduces task dispatch latency. As a result, fine-grain tasks, whose reduced resource requirement is easier to schedule, can be launched efficiently.

Historically, CPU task scheduling is managed by software, either the OS or user-level runtime, and benefits from a higher level, global view of the system. In general, the task scheduling problem is known to be NP-complete [55]. Thus, the goal of a task scheduler should be to find a good heuristic algorithm and make a best-effort schedule based on application and system knowledge. For instance, CPU software schedulers manage all active tasks in the system and schedulers optimized for high-throughput heterogeneous system can be enhanced with prioritization [15]. The application or user-level runtime may assign priorities to tasks to identify those with the earliest finishing time or those on the critical-path of a data-dependent execution [7], [15], [32].

In contrast to CPUs, GPU task scheduling is exclusively managed in hardware and current GPU hardware schedulers are handicapped by minimal knowledge of the kernels. With HSA user mode queuing, an application can create multiple user mode queues

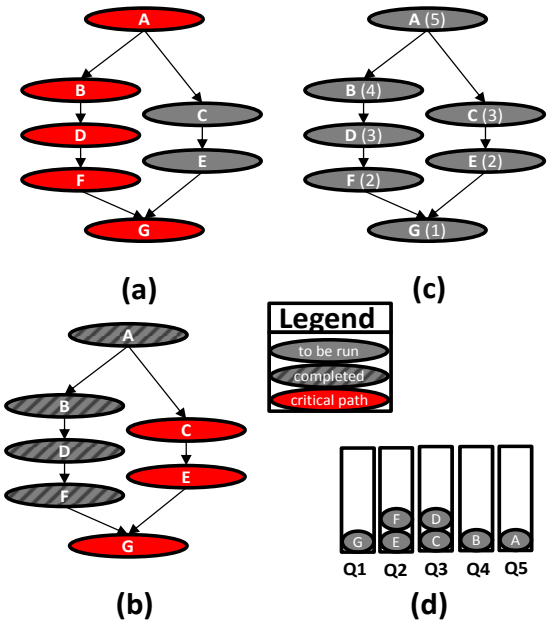

Figure 3. Task graph execution and priorities.

(HSA queues) to launch kernels. However, it is challenging for the GPU to locate all outstanding works on these queues when it can only monitor a fixed number of queues at a time. Therefore, the GPU periodically swaps different queues into the available hardware queue slots (hereafter referred as hardware queues) and processes ready tasks on the monitored queues. To ensure all queues are eventually serviced, the hardware will swap in and out queues at regular intervals, called the *scheduling quantum*.

When the number of allocated HSA queues is greater than the number of hardware queues, the GPU wastes significant time rotating between all allocated queues in search of ready tasks. Furthermore, barrier packets that handle task dependencies can block queues while waiting for prior tasks to complete. In this situation, it is important for the GPU to choose a scheduling policy that prioritizes queues having ready tasks. In this work, combining round-robin and priority-based scheduling with various queue mapping and unmapping criteria, we evaluate five different scheduling policies (three round-robin and two priority-based).

3. GPU Command Queue Organization

Figure 2 shows the GPU queue scheduling and task dispatching mechanism used in this paper. The GPU device driver is responsible for creating queues in the system memory. The driver also creates a *queue list* that contains the queue descriptors of all the queues created by the application. Queue descriptors have all the necessary information to access HSA queues, such as the pointer to the base address of a queue. This queue list is passed to the GPU hardware scheduler during initialization phase. Although the queue list is read by the hardware scheduler, it is modified only by the device driver. The hardware scheduler maps and unmaps queues from the queue list to the hardware list stored inside the *packet processor* at each scheduling quantum. The number of entries in the hardware list is limited and each entry in the hardware list corresponds to the HSA queue mapped to a hardware queue. The packet processor monitors these hardware queues, processes both kernel dispatch and barrier HSA packets, resolves dependencies expressed by barrier packets and forwards ready

```
hsa_queue_create(gpu_agent, queue_size,
        HSA_QUEUE_TYPE_SINGLE, NULL, NULL,
        UINT32_MAX, UINT32_MAX, q_ptr, priority);
```

Figure 4. HSA API modifications. ***priority*** is added to HSA queue create API.

tasks to the dispatcher. The dispatcher then launches these tasks on the shaders in work-group granularity.

When the HSA queues are oversubscribed, the hardware scheduler unmaps a queue from the hardware list and maps a new queue from the queue list. Our baseline hardware scheduler selects a new queue to be mapped at each scheduling quantum based on a round-robin policy. However, the newly selected queue may not have ready tasks, resulting in idling GPU resources. Priority-based queue scheduling techniques tries to reduce idling by intelligently mapping a queue with ready tasks. The next section discusses this priority-based queue scheduling in detail.

4. Priority-Based Queue Scheduling

While HSA can expose more task level concurrency to the GPU, it shifts the burden of task scheduling to the hardware. In a complex task graph with hundreds of interdependent tasks, many task queues can become unblocked at any given time. From a data dependency standpoint, the task scheduler is free to launch any ready task. However, a naïve policy may not be able to achieve efficient use of resources and may leave tasks blocked for long periods of time.

Figure 3 explains the deficiency of the naïve scheduler. Tasks on the initial critical path of execution are marked in red in Figure 3 (a). After task A is executed, both tasks B and C are ready to execute. A naïve scheduler could pick either of these two tasks, but executing task B first is the better scheduling decision because task B is on the critical path. Task C can be delayed, but Figure 3 (b) shows delaying the execution of task C until tasks B, D, and F complete, puts task C on the critical path. A more informed scheduler would have executed task C before task F.

To make better decisions, a task priority-aware scheduler is needed. Many techniques have been proposed in literature to prioritize tasks [15][54]. One such technique is the Heterogeneous Earliest-Finish-Time (HEFT) algorithm proposed by Topçuoğlu et al. [54] that computes upward ranks of the tasks and selects the task with highest upward rank for scheduling. A task's upward rank is its distance from an exit node. Figure 3 (c) shows the same task graph from Figure 3 (a) with each task annotated with their upward rank. The ranking uses HEFT algorithm but assumes equal computation time for all tasks. Since the tasks are now annotated with priorities, a scheduler that is aware of the priorities can make informed scheduling decisions. For example, after task A is completed, priority-based scheduler will choose task B to be scheduled because the rank of task B (rank 4) is higher than task C (rank 3). In this paper, we use this HEFT algorithm to determine the priorities of each task.

4.1 Priority-based Hardware Scheduling in HSA

There are three challenges to implement a priority-based task scheduler, (a) the tasks need to be annotated by a ranking algorithm, (b) these annotations should be exposed to the scheduler, and (c) the scheduler should be able to use these rankings when making scheduling decisions.

Since HSA relies on queues to expose concurrency and schedule tasks, we modified the HSA queue creation API [Figure 4], to include a priority field. A queue priority (or rank) is a positive integer value that can be specified at the time of queue creation and

Table 1. Scheduling policies.

Policy name	Scheduling policy	Mapping criteria	Unmapping criteria
RR-blind	Round robin	Blind	Timeout
RR-tout	Round robin	Queue state aware	Timeout
RR-imm	Round robin	Queue state aware	Immediately on empty
QP-tout	Priority based	Queue state aware	Timeout
QP-imm	Priority based	Queue state aware	Immediately on empty

an application can create queues with any priority levels as needed. An application can then use any ranking algorithm to determine a task's priority and enqueue it into the appropriately ranked HSA queue. Figure 3 (d) shows the mapping of prioritized tasks from Figure 3 (c) to HSA queues. Q1 corresponds to the HSA queue with priority 1, Q2 with priority 2 and so on. With the tasks enqueued to the HSA queue with matching priority, a priority-based queue scheduling effectively becomes a priority-based task scheduler.

By enqueuing a prioritized task into a related prioritized HSA queue, the queue scheduler achieves the desired order of task graph execution. At the beginning of every scheduling quantum, the priority-based scheduler maps queues with the highest priority to the hardware queues. Since the application has enqueued tasks to the queues based on task priority, the scheduler giving preference to a high priority queue over a low priority queue directly translates to a high priority task being scheduled before a low priority task. We describe the round-robin and priority-based scheduling policies in detail and identify key optimizations for each policy in the next subsection.

4.2 Scheduling Policies

While round-robin and priority-based scheduling schemes determine the queue to be mapped, they do not decide when to map or unmap a queue. Based on when a queue is unmapped, the scheduling policies are further classified as *timeout* and *immediate*. The default timeout policy waits for the completion of a scheduling quantum to swap out a queue. This policy will lead to hardware queue idling if a mapped queue becomes empty in the middle of a scheduling quantum. To avoid this, we introduced the immediate scheduling policy that swaps a queue immediately after that queue runs out of work. This policy allows empty queues to be swapped out immediately and do not occupy a hardware queue slot.

Mapping an empty queue to a hardware queue can result in its underutilization. Such a situation can happen if the scheduler is not aware of the state of the queues and we classify it as a *blind* scheduling policy. The blind policy maps a queue even if it is empty. To determine whether a queue is empty, the hardware scheduler has to read the queue descriptor before mapping it. However, this extra memory read is not a big overhead as compared to a hardware queue idling due to an empty mapped queue. Combining blind, timeout, and immediate policies with round-robin and priority-based schemes, we explored three variants of round-robin scheduling and two variants of priority-based scheduling.

Round-robin blind (RR-blind): The default round-robin hardware task scheduler picks the next HSA queue to be mapped to hardware in simple round-robin fashion. To ensure fair resource allocation, the GPU task scheduler unmaps one HSA queue and maps a new one at every scheduling quantum. The selection of the victim queue to be unmapped is also done using the same round-robin policy. Since this scheduling policy does not have any notion about the state of a queue, it is called RR-blind and can end up mapping an empty HSA queue with no tasks to hardware, resulting in idle GPU resources. Note that a queue is mapped/unmapped at the end of a scheduling quantum.

Round-robin timeout (RR-tout)*:* This optimized round-robin scheduler is similar to the RR-blind but does not map an HSA queue to a hardware slot if that queue is empty. This scheduling policy is aware of the state of the queue (empty or non-empty queue) and uses that information to make a better scheduling decision than RR-blind. RR-tout also maps/unmaps a queue at the end of a scheduling quantum, when the scheduling timer timeouts and hence the name round-robin timeout.

Round-robin immediate (RR-imm): This scheduling policy further enhances the RR-tout scheduling policy by unmapping hardware queues as soon as they become empty instead of waiting for the end of the scheduling quantum. By immediately interrupting the scheduler in the middle of a scheduling quantum, hardware ensures its finite hardware queue slots are monitoring non-empty queues.

Queue priority timeout (QP-tout): This scheduling policy uses user-provided HSA queue priorities to more effectively schedule software queues to hardware. At the beginning of every scheduling quantum, the scheduler maps the highest priority, non-empty HSA queues. Once mapped to hardware, a queue will never be unmapped in favor of a lower priority queue before it is emptied.

Queue priority immediate (QP-imm): This is the same as RR-imm except that the new queue to be mapped is chosen based on its priority.

Table 1 summarizes these scheduling policies. The blind suffix in the policy name means that it maps a queue without looking at its state. The rest of the policies are aware of the state of queue before mapping and only maps them to hardware when they have work. The -blind and -tout policies wait for the scheduling quantum timeout before unmapping a queue, whereas the -imm variant unmaps a queue immediately after it becomes empty and the scheduling quantum timeout.

Table 2. Simulation parameters.

CPU Clock	3GHz
CPU Cores	2
CPU L1 Data Cache	64KB
CPU L1 Instruction Cache	32KB
CPU Shared L2 Cache	2MB
GPU Clock	800MHz
GPU CUs	8
GPU Exec Units	5 (4 VALU and 1 AGU)
GPU Exec Unit width	16
GPU L1 Data Cache	64KB
GPU L1 Instruction Cache	16KB
GPU Shared L2 Cache	512KB
Hardware queues	20
Scheduling quantum	100us

5. Methodology

5.1 Simulator

We used the gem5 [8] simulator for modeling our baseline APU system. The APU system modeled consists of a detailed x86 out-of-order CPU model combined with an AMD GCN GPU model [3]. Each compute unit (CU) of the GPU has four, 16-wide SIMD ALUs

Table 3. Benchmark tile size, task size, number of tasks and number of priority levels for a given input matrix size and different task granularities.

Benchmarks		NW	LUD	Cholesky
Input Size		1536 X 1536	1024 X 1024	2048 X 2048
Tile Size	Fine	32 X 32	32 X 32	128 X 128
	Medium	32 X 32	32 X 32	256 X 256
	Coarse	32 X 32	32 X 32	384 X 384
Task size	Fine	10 tiles	1 tile, 6 tiles, 36 tiles	1 tile
	Medium	20 tiles	1 tile, 9 tiles, 81 tiles	1 tile
	Coarse	30 tiles	1 tile, 12 tiles, 144 tiles	1 tile
Number of wavefronts in a task	Fine	10	1, 6, 576	3, 4, 4
	Medium	20	1, 9, 1296	8, 10, 16
	Coarse	30	1, 12, 2304	12, 21, 36
Number of GPU tasks in task graph	Fine	275	590	800
	Medium	165	362	112
	Coarse	130	269	50
Number of priority levels in task graph	Fine	95	94	46
	Medium	95	94	22
	Coarse	95	94	16

and one 16-wide SIMD address generation unit that generates addresses for memory instructions. Each wavefront has 64 work items and takes 4 cycles to execute on a 16-wide SIMD execution unit. To keep execution resources busy during long latency memory operations, a CU stores up to 40 wavefront contexts, overlapping execution of different wavefronts with long latency memory operations. Each CU has a private L1 data cache. The instruction cache is shared by 4 CUs. All L1 data caches and instruction caches are backed by a common shared L2. Both CPU and GPU share the same virtual address space and maintain coherent caches.

The hardware scheduler models the queue swap logic discussed in Section 3. Different scheduling policies are modeled by the hardware scheduler and queue swapping is guided by the scheduling policy. This queue swapping logic can support static task graph execution, dynamic task graph execution and fork-join dynamic parallelism supported by CUDA [40]. This model supports the stock HSA runtime and we used version 1.1 for our experiments [27]. We run gem5 in system call emulation mode (SE mode) so that the emulated device driver can communicate with the HSA runtime via emulated Linux ioctl calls [30]. The emulated device driver is also responsible for managing the queue structures in the system memory and communicating this information to the hardware scheduler using the mechanism described in Section 3. Since the system does not stall task submission when unmapping queues and an individual queue is mapped for a long time before getting unmapped, queue unmapping has minimal impact on performance, thus we only modeled it functionally.

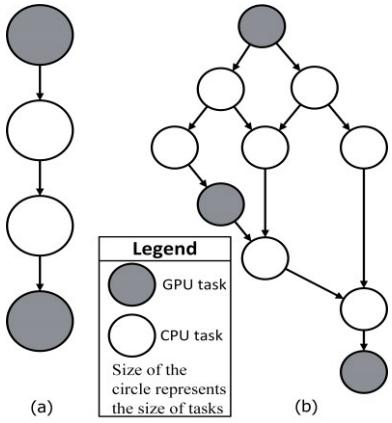

Figure 5. Cholesky task graph (a) 2x2 tiled (b) 3x3 tiled. The shaded nodes represent CPU tasks. The tasks in 2x2 tiled task graph are 1.5x the tasks in 3x3 tiled task graph.

Table 2 lists the simulation parameters used for the evaluation of different scheduling policies. The packet processor modeled has 20 hardware queues and all the applications create up to 128 HSA queues to launch tasks. For queue priority based schedulers, a prioritized task is enqueued on the HSA queue with the same priority. But, for round-robin schedulers, the tasks are enqueued randomly to avoid pathological scenarios. The scheduling quantum was fixed at 100us. We modeled a GPU core that is similar to an AMD FX-8800P APU [23] with 8 CUs. These CUs run at 800MHz frequency whereas CPUs run at a much faster clock speed of 3GHz. Since our benchmarks are multi-threaded, we instantiated 2 CPU cores in our system. So, for some benchmarks, our simulated system will be running both CPU and GPU concurrently with these heterogeneous cores communicating over a fully coherent cache subsystem modeled in ruby [38]. Next, we discuss these benchmarks in detail.

5.2 Benchmarks

We used three task-based benchmarks, Needleman-Wunsch, LU decomposition and Cholesky factorization [47], to evaluate different scheduling policies. Since our objective is to use fine-grain tasks to fill the GPU, we used the asynchronous task-based versions of these benchmarks instead of their bulk-synchronous versions. These task-based versions use Asynchronous Task and Memory Interface (ATMI), a runtime that allows programmers to specify tasks and their dependencies with simple data structures [4].

Needleman-Wunsch (NW): NW uses the Needleman-Wunsch algorithm to align protein sequences. It has two type of GPU tasks operating on a tiled 2-D input matrix. Each task operates on a predetermined number of tiles. For a given input matrix size, as number of tiles in a task decreases, the task becomes more fine-grain and the number of task increases making the task graph more complex. However, since NW has only GPU tasks, the GPU does the same amount of work irrespective of the granularity of tasking for a given input matrix size.

LU Decomposition (LUD)*:* LUD decomposes a tiled 2-D matrix into an upper and lower triangular matrix. It has four different types of GPU task that operate on this tiled input matrix. Similar to NW, the task sizes vary with the number of tiles in a task and fine-grain tasks operate on fewer tiles, making the task graph larger and more complex. However, out of the four GPU tasks, one task always operates on a single tile irrespective of task granularity. Thus, for tasks of this type, the amount of work done on the GPU remains same irrespective of the tasking granularity.

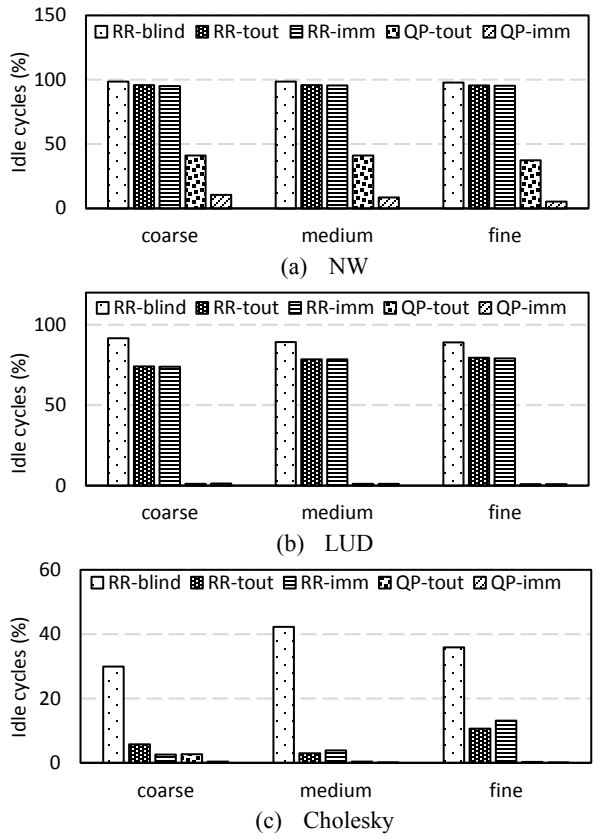

Figure 6. Percentage of *idle cycles* of each benchmark for different task granularities.

Cholesky factorization: Cholesky factorization involves the factorization of a symmetric positive-definite matrix. The task-based algorithm of Cholesky factorization employs four different types of tasks, three GPU tasks and one CPU task, for factorizing a 2-D matrix. Unlike NW or LUD where tasking granularity is changed by the number of tiles in a task, the tasking granularity of Cholesky factorization is changed by changing the tile size itself. Additionally, since there are GPU and CPU tasks in the task graph, the ratio of work done by CPU and GPU changes depending on the tasking granularity.

Figure 5 compares the task graph of a 2x2 tiled and 3x3 tiled Cholesky factorization algorithm. The shaded tasks represent CPU tasks and the unshaded tasks are GPU tasks. The tile sizes and hence the task size of a 2x2 task graph is 1.5x that of a 3x3 task graph. Comparing Figure 5 (a) and Figure 5 (b), the ratio of GPU to CPU tasks with 2x2 tiling is 1:1 but that of a 3x3 tiling is 3:7. That means, for Cholesky factorization, unlike NW and LUD, the amount of work done by GPU is different for different task sizes.

We compared the effect of task granularity on different scheduling policies by running fine-grain, medium-grain and coarse-grain tasks. Table 3 lists the task sizes, tile sizes, number of tasks and number of priority levels in a task graph for different task granularities and for a given input size. The prioritization policy uses the HEFT algorithm. As expected, Table 3 shows with finer task granularity, the number of tasks and task graph complexity increases. It can also be seen from that table that unlike NW and Cholesky, different tasks in LUD operate on different number of tiles. For example, fine-grain LUD tasks operate on 1, 6 and 36 tiles with these tasks launching 1, 6 and 567 wavefronts respectively. Thus, even the fine-grain LUD tasks are capable of occupying the entire GPU. In contrast, even coarse-grain NW or Cholesky do not launch enough tasks to fill the entire GPU with a capacity of 320 concurrent wavefronts (8 CUs X 40 wavefronts per CU). It should be noted in fine-grain LUD that 366 out of the 590 tasks are still large tasks. In this case GPU utilization can still improve, since asynchronous task scheduling allows small tasks to execute simultaneously with larger tasks.

Table 3 also shows that the number of priority levels for both LUD and NW. Since LUD and NW keep the same tile size but only change the number of tasks for different granularities, the prioritization levels did not change. Cholesky, on the other hand, has different tile sizes for different task granularities resulting in fewer priority levels for coarse-grain tasks.

6. Results

The main objective of priority-based scheduling is to map oversubscribed queues in the best order to maximize utilization. Therefore, efficiency of these scheduling policies can be quantified by measuring idle cycles and the number of active wavefronts. Idle cycles are the number of cycles the GPU has no work to execute during task graph execution and are counted starting when the GPU begins executing the first task until the last GPU task completes. Fewer idle cycles indicates that the scheduling policy was more effective in filling available hardware queues. Active wavefronts are the number of wavefronts that are concurrently running on a GPU and is a measurement of the resources utilization. Active wavefronts are sampled each time a new workgroup is launched. Since active wavefronts measures GPU utilization when it is active and idle cycles measure the time the GPU is inactive, a combination of these two gives insight into application performance.

6.1 GPU Idle Cycles

Figure 6 (a) shows the percentage of GPU idle cycles of NW benchmark for different task granularities using a 2048x2048 matrix. Across all evaluated task granularities, one can see that the round-robin based scheduling policies leave the GPU idle for much of application's execution. As expected, the RR-blind scheduling policy that maps empty queues performs the worst with more than 97% idle cycles for all evaluated task granularities. Although optimized RR schedulers (RR-tout and RR-imm) reduce the idle cycles, it is QP-based scheduling policies that are able to greatly reduce idle cycles during execution. While QP-tout brings the idle cycles down to 40%, QP-imm further reduces the idle cycles to fewer than 10% for all evaluated task granularities. Similar to NW, LUD also shows significant idle cycle reduction with QP-based scheduling schemes as shown in Figure 6 (b). Idle cycles are reduced to just 2% for all experiments with QP-based scheduling.

Figure 6 (c) shows idle cycles for Cholesky benchmark. Except for the RR-blind scheduling policy, all other scheduling policies achieve fewer than 15% idle cycles. The task graph for Cholesky has relatively high concurrency, as indicated by fewer priority levels in its task graph. For example, the 2048x2048 matrix with fine-grain tasking has 800 GPU tasks but with just 46 priority levels (refer Table 3). In contrast, the fine-grain task graph of NW benchmark with 1536x1536 input matrix has 275 tasks but with 96 priority levels. A higher number of tasks per priority level indicates that a relatively large number of tasks are ready to execute at any point during the task graph execution. Because of this, a scheduling algorithm that maps only non-empty queues is likely to have at least one ready task among the 20 mapped queues. Hence, even optimized RR schedulers see few idle cycles for Cholesky. Despite this, it can be seen from Figure 6 (c) that QP-based schedulers are able to further reduce idle cycles.

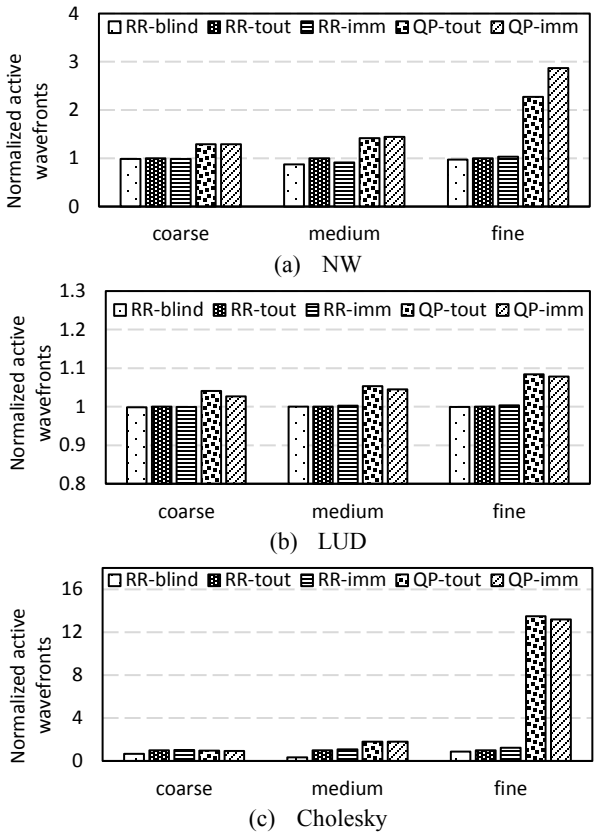

Figure 7. Active wavefronts for different task granularity normalized to their respective RR-tout.

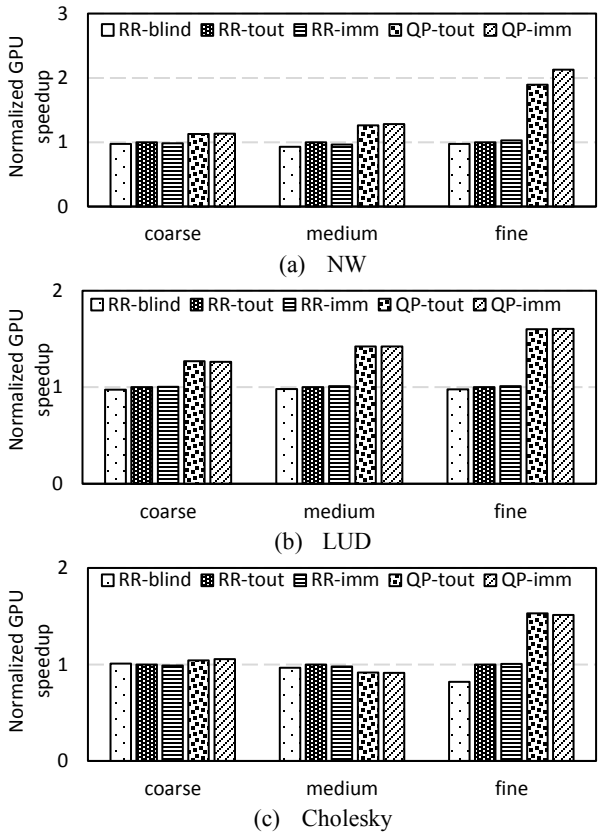

Figure 8. Execution speedup on GPU normalized to RR-tout of each task granularity.

6.2 GPU Resource Utilization

GPU utilization is measured by the number of wavefronts concurrently running on GPU called active wavefronts. Figure 7 shows the active wavefronts normalized to RR-tout scheduling policy. A large number of active wavefronts potentially indicates high utilization of GPU resources. For NW, QP scheduling schemes achieve better GPU utilization, with fine-grain tasks getting the maximum benefit. Fine-grain tasks expose higher concurrency and the priority scheduling policies are able to take advantage of this as indicated by their increase in active wavefronts. Figure 7 (b) shows active wavefronts improves for LUD as well with QP-based scheduling and fine-grain tasks. However, the improvement is only modest (8% for fine-grain tasks) as compared to NW. Since LUD has large tasks that can fully occupy a GPU, the active wavefronts are high even when running a single task. Thus fine-grain tasking achieves only modest additional improvement by running small tasks with relatively less resource requirement concurrently with the larger tasks.

Figure 7 (c) shows active wavefronts for Cholesky. Following the trend of NW and LUD, Cholesky also achieves higher resource utilization with fine-grain tasks and QP-based scheduling polices. However, the medium and coarse granular tasks show only modest utilization improvements with QP-based scheduling. QP-based scheduling increases utilization of medium-granular tasking by 80% whereas no improvement is visible with coarse-granular tasking. For coarse-granular tasking, there are just 50 GPU tasks in the task graph. Since there are 20 hardware queues exposing 20 out of the 50 available tasks at all times, many tasks are dispatched simultaneously even with optimized RR scheduling policies. Because of this, QP cannot do much better than RR variants as shown by the similar number of active wavefronts across all scheduling policies for coarse-grain tasks.

6.3 Application Performance

The GPU resource utilization improvement can directly translate to GPU execution speedup. Figure 8 shows the execution speedup on GPU for different task granularities normalized against their RR-tout. For NW, one can see that all task granularities benefit from QP scheduling with fine-grain tasks improving the most. By comparing Figure 7 and Figure 8, it can be observed that the performance benefits directly come from increasing active wavefronts. Although LUD achieves only a modest increase of active wavefronts, the GPU shows tangible speedup emphasizing the ability of parallel tasks to fill more available GPU resources.

However, for Cholesky, the increase of active wavefronts translates to a significant performance gain only for fine-grain tasks. The medium granularity tasking shows a slight performance degradation, while coarse granularity shows a slight improvement. The performance degradation for medium granularity Cholesky is due to a decrease in cache hit rate, as shown in Figure 9. One can observe that both the L1 hit rate [Figure 9 (a)] and L2 hit rate [Figure 9 (b)] are lower for every scheduling policy compared against the RR-blind policy.

Increasing active wavefronts can negatively affect the execution time of concurrently running wavefronts due to interference. As a

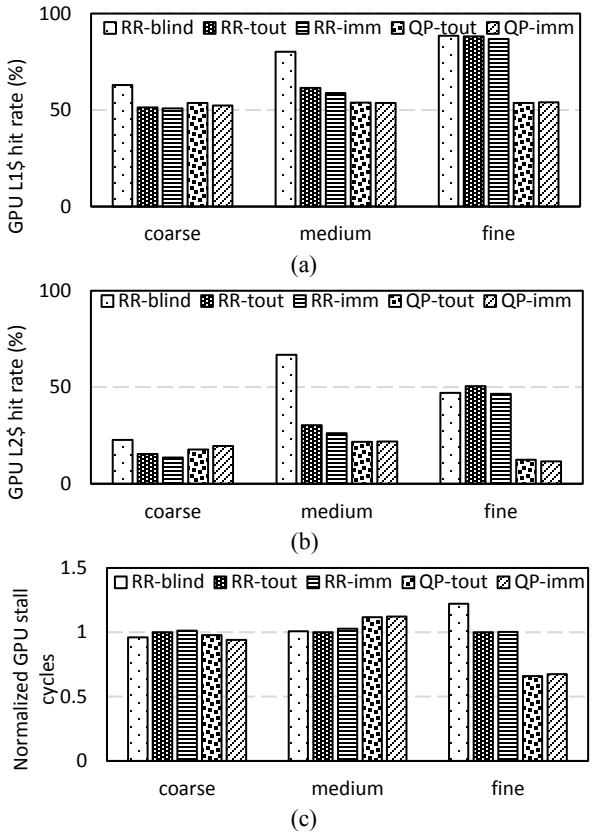

Figure 9. Cholesky benchmark (a) GPU L1 cache hit rate (b) GPU L2 cache hit rate and (c) GPU stall cycles normalized to RR-tout.

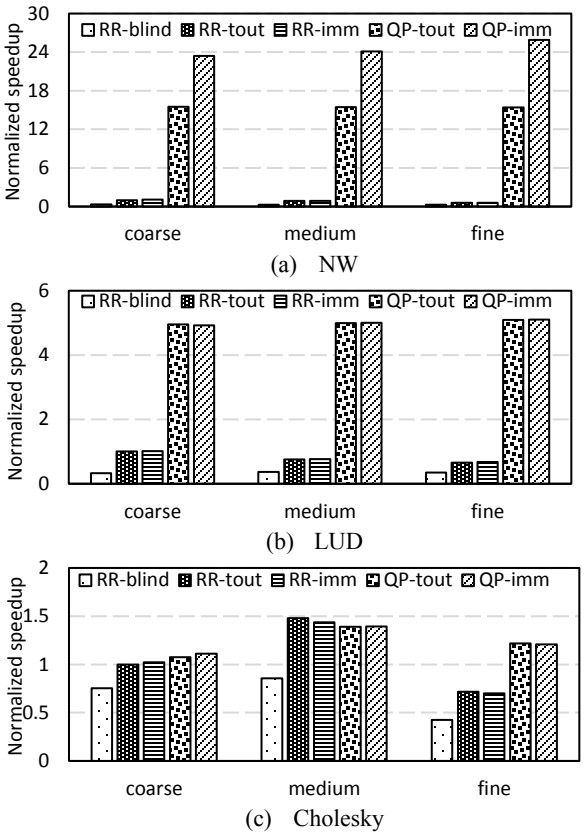

Figure 10. Application speedup for different task granularities normalized to the RR-tout of coarse-grain task.

result, individual wavefronts, and consequently individual kernels, take longer to complete. Because of this interference, the throughput benefits from running large number of active wavefronts can be eclipsed by the longer kernel execution time.

Comparing the cache hit rate [Figure 9] and active wavefronts of Cholesky [Figure 7 (c)], one can see that with a larger number of active wavefronts, the L1 and L2 cache hit rates are reduced. Since all concurrently running wavefronts compete for cache resources, more active wavefronts result in less cache capacity per wavefront. This competition for cache resources leads to cache thrashing. This effect is directly reflected by the increase in GPU stall cycles as shown in Figure 9 (c). Stall cycles indicate that a CU cannot execute any instruction because all of its wavefronts are waiting on memory operations. So, lowering the L1 cache hit rate increases stall cycles, which leads to GPU performance degradation.

Comparing GPU performance of Cholesky [Figure 8 (c)] and GPU stall cycles [Figure 9 (c)], it can be observed that the stall cycles directly correlate with the performance degradation of QP-based scheduling techniques for medium and coarse granularity tasking, confirming that benefits from active wavefronts are negated by interference from greater numbers of wavefronts. However, for fine-grain tasks, although QP scheduling encounters a reduced cache hit rate [Figure 9], the relatively large number of active wavefronts [Figure 7 (c)] overcome the cache hit rate challenge by overlapping long latency memory accesses with wavefront execution. This can be seen from Figure 9 (c) where QP-based scheduling schemes have fewer stall cycles than RR variants. This reduced stall cycles directly translate to application performance improvement [Figure 8 (c)].

In addition to GPU execution time, the overall application execution is also impacted by idle cycles. We show the overall application speedup with different scheduling policies for different task granularities in Figure 10. Since the input size is fixed and only the tasking granularity is changed, all of these experiments perform the same amount of work. As such, we have normalized this speedup against coarse-grain RR-timeout. For NW, although different scheduling policies have similar idle cycle reduction for all three tasking granularities [Figure 6 (a)], the fine-grain tasks are able to better utilize the GPU hardware in terms of active cycles [Figure 7 (a)]. Because of this, QP scheduling polices are the most effective for fine-grain tasking and achieve 26x speedup over coarse-grain RR-timeout. In comparison, coarse-grain and medium-grain tasking achieve a 23x and 24x speedup, respectively. Overall, fine-grain tasks benefit the most with QP-imm and achieve 11% speedup over QP-imm with coarse-grain tasks.

The speedup of LUD for different task granularities are given in Figure 10 (b). Unlike NW, the fine-grain tasking does not outperform coarse-grain tasking by a huge margin with QP scheduling. Although GPU execution speedup with fine-grain LUD is 1.6x faster with QP-immediate, medium and coarse are not far behind with a 1.25x and 1.4x speedup, respectively [Figure 8 (b)]. So, in this case, different granularities benefit similarly in terms of GPU execution speedup. However, the application speedup of QP-based scheduling improves by more than 5x for all tasking granularities because of the reduction in idle cycles [Figure 6 (b)].

Finally, application performance of Cholesky is shown in Figure 10 (c). The application speedup for different task sizes follows the same pattern as GPU speedup except for the RR-blind scheduling policy. Since QP schedulers do not significantly reduce idle cycles as compared with optimized RR schedulers, the application performance is determined exclusively by GPU performance. For RR-blind, although its GPU performance is comparable to RR-opt for medium and coarse-grain tasks, a greater percentage of idle cycles reduces overall application performance.

6.4 Scalability of Scheduling Policies

To demonstrate the scalability of scheduling policies, we run the benchmarks with three different input sizes and fine-grain tasks. Figure 11 (a) shows the speedup of NW for different input sizes, normalized against their respective RR-timeout. The 512, 1024 and 1536 in the x-axis denotes an input matrix size of 512x512, 1024x1024 and 1536x1536, respectively. The tasking granularity is fixed to the fine-grain tasking size of 6 tiles per task across all these input sizes. From that figure, it can be seen that the performance of RR-based schedulers is significantly inferior compared to QP-based schedulers, highlighting the need for better scheduling policies for oversubscribed queues. The RR-tout and RR-imm show similar speedup. Although RR-imm unmaps a queue immediately after that queue gets empty, there is no guarantee that the newly mapped queue has the right set of tasks. This highlights the fact that searching for the right set of tasks by mapping and unmapping queues in a round-robin fashion is the biggest overhead for RR scheduling policies. QP-based scheduling polices eliminate this overhead and perform much better, as depicted in Figure 11 (a). QP-imm achieves 45x speedup for the largest input size. Compared to RR-blind, QP-imm is 91x faster. Figure 11 (a) also illustrates that QP-based scheduling policies take advantage of the immediate unmapping of an empty queue resulting in QP-imm outperforming QP-timeout in all cases. While both QP-imm and QP-timeout have the same queue mapping-unmapping sequence, QP-immediate is able to map the next queue immediately without waiting for the scheduling quantum to complete. This is why QP-immediate outperforms QP-timeout. LUD speedup [Figure 11 (b)] follows a similar pattern of NW with larger input sizes getting higher benefit from QP scheduling. However, the speedup saturates at the 768x768 input size.

Figure 11 (c) shows the performance of Cholesky for different input sizes. As discussed in Section 6.1, Cholesky has high concurrency with larger input sizes and fine-grain tasking, enabling multiple tasks to be ready for execution at any point. So, even the optimized RR policies are not suffered from low GPU utilization, as indicated by the idle cycle percentage [Figure 6 (c)]. Because of this reason, the opportunities for QP schedulers decrease under larger inputs. This explains the relatively lower performance improvement for larger input sizes with QP scheduling for Cholesky.

7. Related Work

Task scheduling schemes: There has been considerable amount of research done on efficiently scheduling tasks to improve resource utilization and power efficiency [6][24][36][44][60][5][19][20]. Chronaki et al. [15] proposed a policy for scheduling dependent tasks onto the asymmetric cores based upon criticality of each task. Daoud et al. [18] proposed a compile-time task scheduler based on the longest dynamic critical path algorithm. Specifically, this algorithm attempts to identify the most important tasks and assigns a higher priority to them. Then, the runtime scheduler will take into account these priorities. Topçuoğlu et al. [54] proposed the Heterogeneous Earliest Finish Time (HEFT) algorithm and Critical

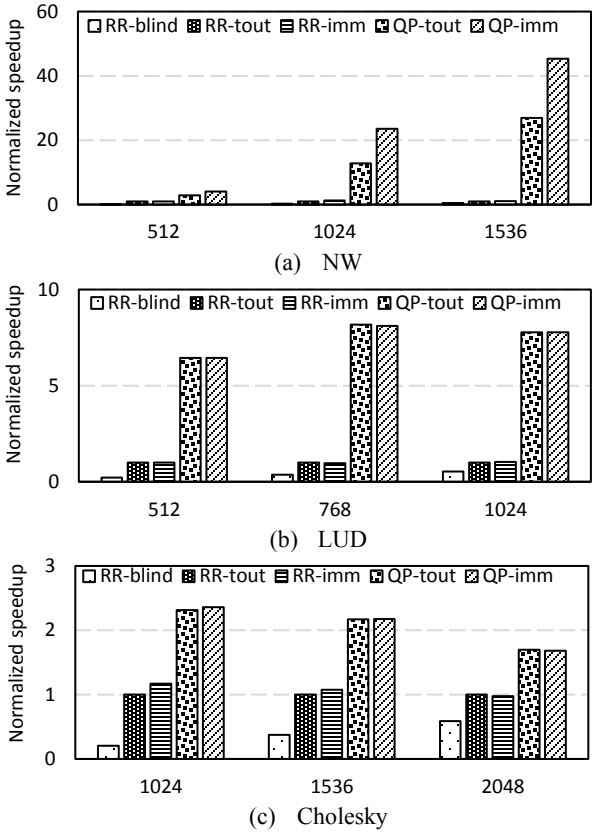

Figure 11. Application speedup for different input sizes with fine-grain tasks. The speedup of each input size is normalized to its RR-tout.

Path on a Processor (CPOP) algorithm that use upward/downward rank for fast task scheduling in heterogeneous systems. We used this HEFT scheduling algorithm for ranking our tasks. However, it is important to note that the oversubscribed queue challenge described in this paper is not inherently a task scheduling problem but a queue scheduling problem. We proposed one solution that uses the notion of task scheduling to mitigate the queue oversubscription challenge.

Task prioritization techniques: In addition to providing insights into task scheduling, Topçuoğlu et al. [54] also propose several algorithms that can be used to assign priorities to tasks according to which can be executed in parallel and how great the computation cost of a task is. Wu et al. [59] proposed modified critical path based prioritization for task distribution in message passing systems. Hwang et al. [26][26] presented earliest time first scheduling algorithm that reduces the communication delays. Tao et al. [53] proposed a cost driven workflow scheduling algorithm based on the Markov Chain-based resource availability prediction model. Wang et al [58] proposed locality aware task prioritization in GPUs. Compared to all prior efforts, we focus on prioritizing tasks by prioritizing the queues to which that task is enqueued, improving the hardware resource utilization and reducing the idle time of oversubscribed command queues.

While we have used HEFT algorithm for task prioritization, that algorithm is not suitable for dynamic task graphs where the task graph is not known a priori. For prioritization of dynamic task graphs, a heuristic based or dynamic critical-path based algorithms can be used.

Fine-grain parallelism: Fine-grain parallelism/tasking is an effective and widely-used approach to improve resource utilization on heterogeneous and homogeneous multi-core systems [33][12][16][21]. However, it requires sophisticated task scheduling policies to avoid the load imbalance of relatively small tasks and the overhead they create. Kumar et al. [34] described Carbon, which uses hardware queues and messaging protocols to distribute tasks across cores. Blumofe et al. [9] proposed work-stealing to allow less busy processors to take work from busy processors to better balance the load. Sanchez et al. [50] presented asynchronous direct messages for fast communication between threads in CMP, which is then used in their scheduler to enable efficient task stealing. Our work uses these asynchronous fine-grain task-based benchmarks to highlight the importance of better queue scheduling policies.

Wavefront scheduling: Numerous researches has proposed efficient wavefront scheduling to hide long latency operations [39][35][37], maintain cache locality [49][31], and resolve control divergence [22][48][11]. Narasiman et al. [39] described two-level warp scheduling that allows groups of threads to execute long latency operations in an interleaved fashion. Such interleaving can better hide the long latencies. Rogers et al. [49] proposed cache conscious wavefront scheduling which improves performance by avoiding thrashing the cache. Fung et al. [22] dynamically regroups the threads into groups to reduce the intra-group control divergence. The focus of our work is command queue scheduling to reduce the starvation of work on a GPU whereas the wavefront scheduling efforts try to increase the efficiency of GPU while doing that work. Hence, our work is orthogonal to these wavefront scheduling efforts and can be used with any of these wavefront scheduling policies.

Kernel launch overheads: The overhead of launching kernels from both host and device has been the investigation topic of many studies [40] [52][56][57]. Wang et al. [56] characterized dynamic applications and showed that the overhead, both in terms of memory space and launch latency, is exponentially increased along with the number of launched kernels. Wang et al [57] proposed DTBL on NVIDIA GPU which launches thread blocks instead of kernels to avoid the launch latency. Chen et al. [13] proposed compiler support which reuses the parent thread to execute the child kernels. Hajj et al. [25] proposed a runtime kernel management which fuses the child kernels since the child kernels are usually light-weight. Tang et al. [52] pointed out that launching arbitrary number of child kernels can encounter hardware restrictions. Our work is orthogonal to these works that aim at reducing kernel launch overhead.

8. Conclusion

Multiple command queues are used in a GPU to expose application concurrency. Although a large number of command queues can increase concurrency, they also cause queue oversubscription and can lead to idle resources. This paper brings to attention this largely overlooked problem of queue oversubscription in modern GPUs.

We also showed that this queue oversubscription challenge can be mitigated by guiding the hardware to service queues in a better order. Specifically, we evaluate queue prioritization, where the programmer uses prioritization to create a schedule for servicing the queues. With our proposed queue prioritization scheme, we were able to reduce the GPU idling to less than 2% for the evaluated benchmarks and achieve speedups up to 45x as compared to a naive round-robin policy.

ACKNOWLEDGMENT

AMD, the AMD Arrow logo, Radeon, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] AMD. "Asynchronous shaders". http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf

[2] AMD. "AMD FirePro GPUs". http://www.amd.com/en-us/innovations/software-technologies/apu

[3] AMD. "AMD GCN Architecture". https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[4] ATMI: https://gpuopen.com/compute-product/atmi/

[5] A. Agarwal and P. Kumar. "Economical Duplication Based Task Scheduling for Heterogeneous and Homogeneous Computing Systems". IACC 2009, 2009.

[6] S. Bansal, P. Kumar, and K. Singh. "An Improved Duplication Strategy for Scheduling Precedence Constrained Graphs in Multiprocessor Systems". Parallel and Distributed Systems, IEEE Transactions on, 14(6), 2003

[7] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions." In the International Conference on Supercomputing, 2012

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. "The gem5 simulator." SIGARCH Comput. Archit. News, 2011.

[9] R. D. Blumofe and C. E. Leiserson. 1999. "Scheduling multithreaded computations by work stealing". J. ACM 46, 5 (September 1999), 720-748.

[10] D. Bouvier, and B. Sander. (2014, August). Applying AMD's Kaveri APU for Heterogeneous Computing. In Hot Chips: A Symposium on High Performance Chips (HC26).

[11] N. Brunie, S. Collange and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," 2012 39th Annual International Symposium on Computer Architecture (ISCA)

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: an object-oriented approach to non-uniform cluster computing". In Proc. of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005

[13] G. Chen and X. Shen. 2015. "Free launch: optimizing GPU dynamic kernel launches through thread reuse". In Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)

[14] N. Christofides, "Graph Theory: An algorithmic Approach." 1975.

[15] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta and M. Valero. "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures". ICS 2015: 329-338

[16] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. "Solving large, irregular graph problems using adaptive workstealing". In Proc. of the 37th International Conference on Parallel Processing, 2008

[17] CUDA streams. https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[18] M. I. Daoud and N. Kharma, "Efficient compile-time task scheduling for heterogeneous distributed computing systems," 12th International Conference on Parallel and Distributed Systems - (ICPADS'06), Minneapolis, MN, 2006, pp. 9 pp.-.

[19] A. Duran, E. Ayguad´e, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. "Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures". Parallel Processing Letters, 21, 2011

[20] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. 2010. "Task Superscalar: An Out-of-Order Task Pipeline". In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO '43).

[21] M. Frigo, C. E. Leiserson, and K. H. Randall. "The implementation of the Cilk-5 multithreaded language". In Proc. of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, 1998.

[22] W. W. L. Fung, I. Sham, G. Yuan and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)

[23] G. Krishnan, D. Bouvier, L. Zhang and P. Dongara. "Energy Efficient Graphics and Multimedia in 28nm Carrizo APU" In Hot Chips: A Symposium on High Performance Chips (HC27).

[24] M. Hakem and F. Butelle. "Dynamic Critical Path Scheduling Parallel Programs onto Multiprocessors". IPDPS'05, 2005

[25] I. E. Hajj, J. Gomez-Luna, C. Li, L. W. Chang, D. Milojicic and W. m. Hwu, "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

[26] J. Hwang, Y. Chow, F. D. Anger and C. Lee. 1989. "Scheduling precedence graphs in systems with interprocessor communication times". SIAM J. Comput. 18, 2 (April 1989)

[27] HSA Foundation. (2016). "HSA Platform System Architecture Specification". Version 1.1. http://www.hsafoundation.com/standards

[28] HSA Foundation. "HSA Runtime Programmers Reference Manual. Version 1.1" (2016). http://www.hsafoundation.com/standards

[29] HSA Foundation. (2016). "HSA Runtime Specification". Version 1.1. http://www.hsafoundation.com/standards

[30] ioctl: http://man7.org/linux/man-pages/man2/ioctl.2.html

[31] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. 2013. "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance". In Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13)

[32] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: "A Task Based Programming Model in a Global Address Space". In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14), 6:1--6:11, 2014

[33] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. "Scheduling strategies for optimistic parallel execution of irregular programs". In Proc. of the 20th annual Symposium on Parallelism in Algorithms and Architectures, 2008.

[34] S. Kumar, C. J. Hughes and A. Nguyen. 2007. "Carbon: architectural support for fine-grained parallelism on chip multiprocessors". SIGARCH Comput. Archit. News 35, 2 (June 2007), 162-173.

[35] S. Lee and C. Wu. 2014. "CAWS: criticality-aware warp scheduling for GPGPU workloads". In Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT '14)

[36] C.-H. Liu, C.-F. Li, K.-C. Lai, and C.-C. Wu. "A dynamic Critical Path Duplication Task Scheduling Algorithm for Distributed Heterogeneous Computing Systems". volume 1 of ICPADS 2006, 2006.

[37] J. Liu, J. Yang and R. Melhem, "SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers," 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

[38] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill and D. A. Wood. "Multifacet's general execution-driven multiprocessor simulator (gems) toolset". SIGARCH Comput. Archit. News, 33(4):92–99, November 2005.

[39] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. 2011. "Improving GPU performance via large warps and two-level warp scheduling". In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44).

[40] NVIDIA "DYNAMIC PARALLELISM IN CUDA", http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism

[41] NVIDIA Tesla GPUs: http://www.nvidia.com/object/tesla-servers.html

[42] NVIDIA, "JP Morgan Speeds Risk Calculations with NVIDIA GPUs," 2011.

[43] OpenMP4.5 Specification. (2015). "The OpenMP Architecture Review Board".http://www.openmp.org/mp-documents/openmp-4.5.pdf

[44] A. Page and T. Naughton. "Dynamic Task Scheduling using Genetic Algorithms for Heterogeneous Distributed Computing". In Parallel and Distributed Processing Symposium, 2005.

[45] S. I. Park, S. P. Ponce, J. Huang, Y. Cao and F. Quek, "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture," 2008 37th IEEE Applied Imagery Pattern Recognition Workshop, Washington DC, 2008, pp. 1-7.

[46] G. Pratx and L. Xing, "GPU Computing in Medical Physics: A Review," in Medical physics, 2011.

[47] S. Puthoor, A. M. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann, and G. Rodgers. 2016. "Implementing directed acyclic graphs with the heterogeneous system architecture." In Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU '16). ACM, New York, NY, USA, 53-62.

[48] T. G. Rogers, M. O'Connor and T. M. Aamodt, "Divergence-Aware Warp Scheduling," 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

[49] T. G. Rogers, M. O'Connor, and T. M. Aamodt. 2012. "Cache-Conscious Wavefront Scheduling". In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)

[50] D. Sanchez, R. M. Yoo, and C. Kozyrakis. 2010. "Flexible architectural support for fine-grain scheduling". In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS XV)

[51] S. S. Stone, J. P. Haldar, S. C. Tsao, W. -m. W. Hwu, B. P. Sutton, and Z. -P. Liang. 2008. "Accelerating advanced MRI reconstructions on GPUs". J. Parallel Distrib. Comput. 68, 10 (October 2008)

[52] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir and C. Das . "Controlled Kernel Launch for Dynamic Parallelism in GPUs." In proceedings of The 23rd International Symposium on High-Performance Computer Architecture (HPCA 2017)

[53] Y. Tao, H. Jin, S. Wu, X. Shi, and L. Shi. 2013. "Dependable Grid Workflow Scheduling Based on Resource Availability". Journal of grid computing (2013): 1-15.

[54] H. Topçuoğlu , S. Hariri and Min-You Wu, "Task scheduling algorithms for heterogeneous processors," Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth, San Juan, 1999, pp. 3-14.

[55] J. D. Ullman. "NP-Complete Scheduling Problems," Journal Computer and Systems Sciences, vol. 10 pp. 384-393, 1975.

[56] J. Wang, and Y. Sudhakar. "Characterization and analysis of dynamic parallelism in unstructured GPU applications." Workload Characterization (IISWC), 2014 IEEE International Symposium on. IEEE, 2014.

[57] J. Wang, N. Rubin, A. Sidelnik and S. Yalamanchili, "Dynamic Thread Block Launch: A lightweight execution mechanism to support irregular applications on GPUs," 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)

[58] J. Wang, N. Rubin, A. Sidelnik and S. Yalamanchili, "LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)

[59] M. Y. Wu and D. D. Gajski, "Hypertool: a programming aid for message-passing systems," in IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 3, pp. 330-343, Jul 1990.

[60] C. Wakeland, A. Lyashevsky and L. Antani. "Scalable Acceleration of Real-time Audio Processing Using Hardware-Partitioned GPU Compute Units", GameSoundCon, 2016. https://www.gamesoundcon.com/2016-game-sound

[61] Z. Zong, A. Manzanares, X. Ruan, and X. Qin. "EAD and PEBD: Two Energy-Aware Duplication Scheduling Algorithms for Parallel Tasks on Homogeneous Clusters". Computers, IEEE Transactions on, 60(3), 2011.