This is the peer reviewed version of the following article:


Institute of Electrical and Electronics Engineers Inc.

Terms of use:
The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher’s website.

25/07/2024 14:37

(Article begins on next page)
Reconciling QoS and Concurrency in NVIDIA GPUs via Warp-Level Scheduling

Jayati Singh¹ Ignacio Sañudo Olmedo² Nicola Capodieci² Andrea Marongiu² Marco Caccamo³

¹University of Illinois Urbana-Champaign, United States
²University of Modena and Reggio Emilia, Italy
³Technical University of Munich, Germany

¹jayati@illinois.edu ²name.surname@unimore.it ³mcaccamo@tum.de

Abstract—The widespread deployment of NVIDIA GPUs in latency-sensitive systems today requires predictable GPU multi-tasking, which cannot be trivially achieved. The NVIDIA CUDA API allows programmers to easily exploit the processing power provided by these massively parallel accelerators and is one of the major reasons behind their ubiquity. However, NVIDIA GPUs and the CUDA programming model favor throughput instead of latency and timing predictability. Hence, providing real-time and quality-of-service (QoS) properties to GPU applications presents an interesting research challenge. Such a challenge is paramount when considering simultaneous multikernel (SMK) scenarios, wherein kernels are executed concurrently within each streaming multiprocessor (SM). In this work, we explore QoS-based fine-grained multitasking in SMK via job arbitration at the lowest level of the GPU scheduling hierarchy, i.e., between warps. We present QoS-aware warp scheduling (QAWS) and evaluate it against state-of-the-art, kernel-agnostic policies seen in NVIDIA hardware today. Since the NVIDIA ecosystem lacks a mechanism to specify and enforce kernel priority at the warp granularity, we implement and evaluate our proposed warp scheduling policy on GPGPU-Sim. QAWS not only improves the response time of the higher priority tasks but also has comparable or better throughput than the state-of-the-art policies.

I. INTRODUCTION

Modern automotive applications feature compute-intensive workloads in which tasks must be processed within defined timing requirements. Applications such as object tracking, lane-following, and obstacle avoidance are safety-critical and must therefore meet hard deadlines [1]. In contrast, augmented/virtual-reality applications like rendering, SLAM and eye-tracking feature soft deadlines or softer quality-of-service (QoS) requirements. These applications are characterized by high processing power demands that cannot be met by traditional multi-core platforms. In this context, traditional multi-core CPUs are often coupled to massively parallel accelerators. General-purpose graphics processing units (GPGPUs, henceforth GPUs), offer a cost-effective architectural solution on account of their high performance per watt ratio.

GPU vendors’ road-maps typically foresee the release of “bigger” GPUs at each new generation, where the computing resources (CUDA cores and streaming multiprocessors in NVIDIA terminology) increase with each chip release. This increase in the streaming multiprocessor (SM) count and the compute capability of individual SMs is not always matched by an
increase in the parallelism that application kernels can expose [2] [3], which brings developers to explore the integration of multiple kernels onto the same GPU to keep computing resources busy.

Multi-kernel execution in GPUs has been widely studied in the last years [4] [5], considering both spatial partitioning (SP) – where multiple kernels are distributed across SMs – and simultaneous multikernel execution (SMK) – where multiple kernels concurrently share the same SMs. SMK leads to resource arbitration at the level of individual groups of threads, called warps in NVIDIA terminology.

SMK is an effective approach to increase GPU utilization compared to SP [6], but it poses additional challenges when QoS requirements and real-time constraints (i.e., kernel priorities) are concerned. CUDA streams offer a priority mechanism that allows, to control QoS requirements in SMK. However, this mechanism operates at the granularity of a whole thread-block, which is too coarse to safely bound kernel latencies.

Acting at the warp level has significant advantages over stream prioritization (see Section II-B). However, the NVIDIA warp scheduler inside each SM lacks a notion of user-defined priority; this makes the enforcement of QoS requirements among kernels next to impossible when using SMK. Therefore, in this paper we propose a QoS-Aware Warp Scheduling (QAWS) policy to effectively prioritize concurrent kernels according to their QoS requirements. More specifically:

1. We demonstrate the need for a QoS-aware warp scheduler when concurrent kernels execute in NVIDIA GPUs.
2. We propose a budget-based QAWS policy that uses information about the QoS requirements of each kernel.
3. We implement the proposed QAWS policy on GPGPU-Sim and evaluate it against state-of-the-art warp scheduling algorithms deployed on NVIDIA GPUs, namely greedy-then-oldest (GTO) and loose round-robin (LRR).

The experimental results show that our QAWS policy reduces the response time of homogeneous high priority kernels by 22% on average and by 10% on average for heterogeneous kernels without affecting the total system throughput.

II. SCHEDULING HIERARCHY IN NVIDIA GPUs

NVIDIA GPUs are built as a composition of several streaming multiprocessors (SM). A SM is a computing cluster which hosts different ALUs (or CUDA cores). SMs might internally be further organized as a set of sub-cores or processing blocks that consist of 32 ALUs sharing register and memory resources and hosting the execution of as many threads. This group of 32 threads goes by the name of a warp. All the threads within a warp execute the same instruction in lockstep (SIMT).

The arbitration of GPU applications involves scheduling decisions in a hierarchical manner [7], [8]:
1) The application scheduler manages applications launched in different memory spaces in a TDMA fashion.
2) Applications may be composed of multiple streams. A stream is an abstraction of a queue of compute or copy commands that are offloaded to the GPU. Commands within the same stream are executed in FIFO order.
3) Compute commands (i.e., kernels) are organized in thread blocks, which are distributed to available SMs following a resource usage-aware variant of round robin [8]. Kernels issued in different streams can execute concurrently if enough GPU resources are available. Preemption can be applied at block boundaries.
4) Blocks are composed of warps, which are the atomic scheduling unit. Each sub-core in an SM contains a warp scheduler that is in charge of dispatching ready warp instructions to various
SIMD lanes like INT, FP64, etc. (see Fig. 1). Warsps are distributed through the sub-cores in a round-robin manner [9], [10].

A. Warp scheduling

The warp scheduler organizes ready-to-execute instructions from a set of available warps. Anytime a warp stalls at an instruction, the warp scheduler chooses another warp to be executed. Each warp scheduler maintains a pool of warps it can choose from at every GPU clock cycle. The warp scheduling policy determines which instruction from a ready warp is issued every cycle. This policy is typically modeled as loose round-robin (LRR) or greedy-then-oldest (GTO) [11]. Related work has not been able to reach a consensus on the scheduling policy implemented in NVIDIA GPUs. Some work [8], [12] suggests that it follows the LRR policy, while others [9], [11] claim that it follows GTO. According to the LRR policy, the same warp is issued by the warp scheduler every cycle until an instruction stalls, at which point the warp scheduler chooses the next warp in round robin order (See Fig. 3b).

In the GTO policy, the same warp is issued by the warp scheduler every cycle until an instruction stalls, at which point the warp scheduler chooses the next instruction from the oldest ready warp (See Fig. 3a). LRR assigns equal priority to all warps, ensuring that all warps make equal progress. LRR is beneficial if the warps in the SM have spatial locality and share cache lines and DRAM row buffers, thus increasing cache and row buffer hits. However, if there is no inter-warp locality, all the warps reach long latency operations at the same time and there are no warps left to hide this latency, resulting in idle cycles and performance loss.

Let us consider LRR in an SMK scenario, where multiple kernels share the GPU concurrently under the LRR warp scheduling policy. Warsps belonging to different kernels will be given the same priority, which could lead to a higher-QoS/priority kernel missing its deadline, in order to achieve the “fairness” as promised by LRR.

GTO attempts to overcome the long latency problem by allowing unequal progress across warps. With GTO, the long-latency periods of the warps do not overlap, ensuring that there are always enough warps to hide a long latency stall. From a real-time perspective, however, GTO can be more harmful than LRR, since GTO gives a higher priority to older warps. Thus, if a lower-QoS/priority kernel is launched before a higher QoS/priority kernel and the two can execute concurrently, GTO will prioritize the warps of the lower-priority kernel, significantly increasing the response time of the higher priority kernel, ultimately leading to a deadline miss.

In this work, we propose QoS-aware warp scheduling (QAWS). QAWS builds upon the benefits of GTO but also considers the kernels’ timing constraints, striving to achieve the best of both worlds: performance and predictability.

B. Example

Fig. 2 clarifies the role of QoS-based arbitration at the stream- and warp-level. We consider two kernels, \( K_1 \) and \( K_2 \), with almost overlapping arrival...
Fig. 2: Scheduling Example II-B

Several warp scheduling policies have been proposed in the literature for improving the GPU performance. The two widespread policies adopted in NVIDIA GPUs are GTO and LRR [9], [11], [8], [12]. Two-level warp scheduling is proposed in [15], [16]. The two-level warp scheduler maintains warps as two subgroups, the fetch group and the ready queue, to improve performance [15], [16]. A warp in the ready queue is demoted to the fetch group when it encounters a long latency instruction. LRR and GTO policies can be used to order warps between and within the groups. Since different scheduling policies can be suitable for different workloads, [17] determines which warp scheduling policy to apply on the different phases of the kernel through compile-time analysis. [18] extends this to be dynamic, based on the instruction issue pattern at runtime. Other work [19] proposes modulating the warp scheduling policy to shape the cache access patterns to avoid cache thrashing, and subsequent misses. All the above solutions offer sophisticated solutions to improve the instructions per cycle and utilization of the GPU by preventing idle cycles. However, all these policies are optimized for warps belonging to a single kernel and perform well when a single kernel is executing in an SM. With the constant increase in GPU compute capabilities, multi-tasking is inevitable, yet little to no investigation is done on how these policies impact the system throughput and individual kernel performance in scenarios with multiple concurrently executing kernels. Furthermore, none of the aforementioned solutions is able to prioritize/schedule warps according to the timing requirements for kernels deployed in soft/hard real-time systems. In this work, we examine the performance of GTO and LRR when warps of multiple kernels are arbitrated by the warp scheduler. Furthermore, to the best of our knowledge this work...
is the first to propose a warp scheduling policy that is aware of the QoS requirements of the kernel associated with the warps, leading to predictable execution when multitasking via SMK in GPUs.

IV. PROPOSED WARP SCHEDULING ALGORITHM

We propose QAWS (QoS-Aware Warp Scheduling), which is based on the real-time priorities/QoS requirements of the corresponding kernel associated with each warp. To achieve QoS-Aware Warp Scheduling, we define budget, which is a number associated with each kernel; a higher priority/QoS requirement kernel should be assigned a larger budget.

Within every sub-core, QAWS groups warps based on the budget value of the kernel they belong to. Warps of the same kernel belong to the same group. Similar to GTO, our policy issues a warp (from a given group) every cycle until it encounters a stall, after this QAWS switches context and issues another warp from the same group. QAWS continues to issue warps from a single group until the number of context switches within that group reach the budget or all warps in that group are stalled. After the budget is reached, the policy starts issuing warps from another group and resets the context switch count for the first group. The higher the budget of a kernel, the more stalls QAWS can tolerate and the longer the warps of that kernel can execute before switching context to execute warps of another kernel. For e.g., if a kernel $K_1$ has a budget of 1 and kernel $K_2$ has budget $b > 1$, QAWS will tolerate $b$ stalls of $K_2$ before switching context to execute $K_1$ warps. Whereas, QAWS will only tolerate 1 stall of $K_1$ before switching back to $K_2$. Such a strategy provides kernels a computing capacity that is proportional to their QoS requirements while preventing starvation of kernels with smaller budgets. We implement our budget-based policy, QAWS, on GPGPU-Sim.

A. QoS-Aware Warp Scheduling Policy Implementation

At every simulation cycle in GPGPU-Sim, the warp scheduler in each sub-core invokes the SortWarps function which returns a prioritized queue of warps (called $Q_{issue}$) sorted according to a specific warp scheduling policy. The existing simulation framework has various implementations (for LRR, GTO, etc.) of this function depending on the simulator configuration. The scheduler then iterates through the warps in $Q_{issue}$ until it finds a ready warp and an available functional unit (special-function unit, load-store unit, etc.) required by the warp. It then dispatches the warp to the respective functional unit. The scheduler invokes SortWarps again in the next cycle. We present a SortWarps implementation to sort the warps based on QAWS, outlined in Algorithm 1.

In this work, we limit the number of kernels in the GPU to two. However, this model can be easily scaled to accommodate more than two kernels. QAWS organizes the warps of the kernels with distinct budgets into distinct groups, say $g_0$ and $g_1$ (line 7). In Alg. 1, we use the convention that $g_0$ is the group that was prioritized in the previous cycle. When there is no history, $g_0$ is the group with the higher budget.

Similar to GTO, QAWS continues to issue the same warp greedily every cycle (lines 11-12) until it encounters a stall (line 13). At this point, the scheduler switches context to another warp from the same group $g_0$ (line 17) and executes that warp greedily. The scheduler continues to issue warps in $g_0$ until the number of context switches within $g_0$ ($g_0.ncs$ in Alg. 1) reaches the defined budget (line 18). Then, the scheduler resets the context switch count for $g_0$ (line 19) and proceeds to issue warps from $g_1$ (line 20) until the number of context switches in $g_1$ reach the $g_1$ budget, and so on. In essence, the budget represents the number of warp context switches within a group.
(g, ncs) that can be tolerated by the scheduler until it begins to issue warps from another group.

Since the warps within a group are ordered using the standard GTO policy, in the case where all the kernels are assigned the same budget there is only one group and the scheduling policy is reduced to GTO (line 9).

**Example.** (See Fig. 3) We consider two kernels K1 and K2, where K2 has a higher QoS demand than K1 but is launched just one cycle after K1. Both kernels execute the same code with only nine instructions. We consider an SM sub-core with four warps where warps \{w0, w1\} ∈ K1 and warps \{w2, w3\} ∈ K2. Fig. 3a and Fig. 3b demonstrate GTO and LRR respectively. Fig. 3c illustrates the QAWS policy when K2 has a budget of 4 and K1 has a budget of 1. In Fig. 3c, since K1 is launched first, w0 ∈ K1 executes until it stalls at cycle 1. Then, the scheduler switches context to w1 and increments the context switch count for K1 (line 16 of Alg. 1). Since the budget of K1 is 1, upon the next stall at cycle 2 (line 18 of Alg. 1), the scheduler begins to issue warps of K2 and continues to do so until cycle 12 when it switches back to issue K1 warps. Whereas, in Fig. 3d the budget of K2 is 6, hence the scheduler continues (lines 15-17 of Alg. 1) to issue warps from K2 beyond cycle 12, all the way until cycle 18.

At cycle 22 in Fig. 3c, w3 encounters a stall however since it does not switch context to a warp in the same group (because no more warps in the group are left), ncsK2 is not incremented. Since the context switch count is still less than the K2 budget, warps of K2 are still prioritized by the policy (lines 22 and 23 in Alg. 1) and w3 is issued as soon as it is ready in cycle 26. The budget sets a limit on the number of context switches and not the number of warp stalls within a warp group.

**Analysis.** Consider the response times of K1 and K2 for the four cases in Fig. 3. In Fig. 3a, GTO

```
Algorithm 1 Sort Warps with QAWS policy
1: state warps ▷ Warps assigned to this sub-core
2: state W_greedy ▷ Warp issued in last cycle
3: state issued_head ▷ If head of Q_issue issued last cycle
4: state g0, g1 ▷ Warps grouped by budget
5: state out Q_issue ▷ Prioritized queue of warps
6: function SortWarps
7:   g0, g1 ← GETGROUPS(warps)
8:   if g1 is ∅ then
9:     return SORT(g0) ▷ Resort to GTO
10:   end if
11:   if issued_head then
12:     return SORT(g0) ⊕ SORT(g1) ▷ ⊕: Concat
13:   else ▷ Head of Q_issue stalled last cycle
14:     if W_greedy ∈ g0 then
15:       if g0.ncs < g0.budget then
16:         g0.ncs ← g0.ncs + 1
17:         return SORT(g0) ⊕ SORT(g1)
18:       else ▷ ncs reaches budget
19:         g0.ncs ← 0 ▷ Reset
20:         return SORT(g1) ⊕ SORT(g0)
21:       end if
22:     else ▷ All warps ∈ g0 stalled last cycle
23:       return SORT(g0) ⊕ SORT(g1)
24:     end if
25:   end if
26: end function
27: function GETGROUPS(warps)
28:  return groups g0, g1 classified by warp budgets
29: end function
30: function SORT(gi)
31:  if W_greedy ≠ NullPtr and W_greedy ∈ gi then
32:    return warps ∈ gi sorted by greedy-then-oldest
33:  else
34:    return warps ∈ gi sorted by oldest warp first
35:  end if
36: end function
```
achieves the best response time for $K_1$, but the worst response time for $K_2$. This is highly undesirable since $K_2$ has a higher QoS demand than $K_1$. **Priority inversion** is observed in GTO because it executes the warps of the lower priority kernel $K_1$ before $K_2$ warps. This is because $K_1$ warps are older than $K_2$ warps: GTO only sees the age of each warp and is kernel-agnostic. The LRR policy in Fig. 3b achieves fairness at the expense of the response times of both kernels. We see that QAWS with $K_2$ budget of 4 in Fig. 3c has a much better $K_2$ response time than GTO/LRR. Increasing the $K_2$ budget to 6 in Fig. 3d further reduces the $K_2$ response time and in fact has the best $K_2$ response time. This highlights that we can control the kernel response times to meet QoS requirements through the relative budgets of the kernels.

Additionally, QAWS in Fig. 3d achieves an average execution time of 31 cycles compared to an average of 29 cycles achieved by GTO in Fig. 3a. This is a small price to pay for the significantly improved response time of a higher QoS kernel. While this is a hand-constructed example, we observe similar trends in real workloads (Section V). The budget-based mechanism of QAWS enables us to control the response times or throughput of tasks running on the GPU based on the system requirements. Due to space constraints, we leave the analysis of the hardware cost of QAWS as a part of future work.

Fig. 3: An illustration of the warp execution order in LRR, GTO and QAWS. $K_1$ requires a lower QoS than $K_2$.

V. EVALUATION

A. Methodology

The GPGPU-Sim configuration used is described in Table Ia. Note that we used a configuration that models the NVIDIA TITAN V as it is the most recent tested configuration released at the time of writing. We evaluate the proposed policy using benchmarks presented in Table Ib. The benchmarks are extracted from the CUDA API Samples, Rodinia [20] and Polybench [21]. pc is a purely compute benchmark wherein each thread executes only one load and one store instruction, and loops over compute instructions.

In all our experiments, we launch two kernels, $K_1$ and $K_2$ and run them concurrently on the GPU. We assume that $K_2$ requires a higher QoS than $K_1$. However, we launch $K_2$ right after $K_1$ in the CUDA code, since that is the most pessimistic case from a QoS perspective. We enforce SMK in the GPU by (i) launching both kernels in separate streams, and (ii) implementing the kernels using persistent threads to ensure that every SM has an equal number of warps from $K_1$ and $K_2$. With this setup, we evaluate the response times of $K_1$ and $K_2$ when scheduled with LRR, GTO and QAWS. Furthermore, we also analyze the average execution time of $K_1$ and $K_2$ to demonstrate the effect of QAWS on the system throughput.

We consider two sets of experiments. One with
### Table 1: a) GPGPU-Sim setup b) Benchmarks used in the experiments.

<table>
<thead>
<tr>
<th>Description</th>
<th>Configuration</th>
<th>Benchmark</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>TITAN V</td>
<td>pc</td>
<td>Purely Compute</td>
<td>CI</td>
</tr>
<tr>
<td>GPU Clusters</td>
<td>40</td>
<td>pf</td>
<td>Pathfinder</td>
<td>CI</td>
</tr>
<tr>
<td>SMs per Cluster</td>
<td>2</td>
<td>2dce</td>
<td>2D Convolution</td>
<td>CI</td>
</tr>
<tr>
<td>Total SMs</td>
<td>80</td>
<td>dxe</td>
<td>DXTC</td>
<td>CI</td>
</tr>
<tr>
<td>Schedulers per SM</td>
<td>4</td>
<td>bin</td>
<td>Binomial Options</td>
<td>CI</td>
</tr>
<tr>
<td>Warp Scheduler</td>
<td>GTO/LRR/QAWS</td>
<td>vec</td>
<td>Vector Add</td>
<td>MI</td>
</tr>
<tr>
<td>L1 + SHM Size</td>
<td>128KB</td>
<td>mm</td>
<td>Matrix Multiply</td>
<td>MI</td>
</tr>
<tr>
<td>L2 Size</td>
<td>4.5MB</td>
<td>his</td>
<td>Histogram</td>
<td>MI</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>10.1</td>
<td>atax</td>
<td>A^T.AX</td>
<td>MI</td>
</tr>
</tbody>
</table>

(a) (b)

**homogeneous kernels** where $K_1$ and $K_2$ are different instances of the same kernel. Doing so ensures that the baseline execution times of $K_1$ and $K_2$ (in isolation) are identical, making it easier to highlight the effect of the warp scheduling policy. Second, we combine the benchmarks using **heterogeneous kernels**. We limit the number of experiments to five because the various kernels have different baseline execution times and hence we choose kernels with baseline execution times in the same order of magnitude to simplify analysis.

### B. Results

We first consider the response times of the homogeneous kernels case as shown in Figure 4a (left-most). All response times are normalized w.r.t. the $K_1$ response time in LRR. We can see that with GTO scheduler, $K_2$, despite requiring higher QoS and being launched only a few cycles (8 cycles in GPGPU-Sim) after $K_1$, finishes long after $K_1$. This is because $K_1$ is launched first, and the $K_1$ warps are older than $K_2$ warps. Hence, the kernel with a lower QoS is implicitly given a higher priority by the scheduling policy. QAWS resolves this priority inversion by letting the developer assign the warp budget according to the needs of each kernel in the system. In our experiments, we pick the budget of $K_1$ warps as 1 and the budget of $K_2$ warps as $b$, where $b \in \{2, 4, 8\}$. While all these values of $b$ give similar performance, we choose the $b$ that gives the lowest response time for $K_2$. The exact relationship between the warp budget and kernel execution time is not straightforward, and will studied thoroughly in future work. We see that QAWS outperforms LRR for both $K_1$ and $K_2$ for all kernels except atax and his. $K_2$ response time is the least with QAWS, which is clearly demonstrated in the Speedup plot (center) in Figure 4a. Note that we see very little improvement in the atax and his kernels and this is because they are latency-bound, i.e. most of the warps are always stalled on long latency instructions. Therefore, more or fewer warp issue slots make no difference since none of the warps are ready to issue. On an average among all the workloads, QAWS improves the $K_2$ response time by 22%.

The response time when executing heterogeneous kernels are similar and are shown in Figure 4b. Note that the kernel response times shown in the left-most plot of Figure 4b also show the response times when $K_1$ and $K_2$ are executed serially and in isolation. We plot this to highlight the difference in the execution times of $K_1$ and $K_2$. On an average among all the workloads, QAWS improves the $K_2$ response time by 10%.

We plot the average execution times of $K_1$ and $K_2$ to observe the effect of QAWS on the system throughput as seen in the rightmost plots of Figures 4a and 4b resp. QAWS outperforms LRR and is as good as GTO for most kernels. This is because QAWS uses GTO to order warps within a group. Hence, with QAWS, applications enjoy the benefits of GTO (high throughput), without the curse of priority inversion.

### VI. Conclusion

In this work we explore a mechanism to facilitate predictable SMK in NVIDIA GPUs. We propose a budget-based QoS-aware warp scheduling policy and evaluate it on the state-of-the-art GPGPU-Sim simu-
Fig. 4: Evaluation of LRR, GTO and QAWS when executing two kernels. $K_2$ demands a higher QoS than $K_1$.


