Coordinating the fetch and issue warp schedulers to increase the timing predictability of GPUs

Noïc Crouzet, Thomas Carle, Christine Rochange

Coordinating the fetch and issue warp schedulers to increase the timing predictability of GPUs

Noïc Crouzet, Thomas Carle and Christine Rochange
Institut de Recherche en Informatique (IRIT)
Université de Toulouse, CNRS, Toulouse INP, UT3, France
{crouzet, carle, rochange}@irit.fr

***Abstract*—The verification of a time-critical system requires precise analysis of execution times, which makes assumptions on the system's behavior, especially when it is poorly documented. One of those assumptions, when the system features a GPU accelerator, relates to the policy that each Streaming Multiprocessor (SM) follows to schedule warps. We argue that the literature overlooks the lack of synchronization between the instruction fetch and instruction issue schedulers, while this is likely to make the behavior of existing GPUs unpredictable. We propose to coordinate the action of the fetch and issue stages in GPU pipelines in order to enable reliable static timing analysis. We implement our approach in Vortex, a RISC-V-based open-source GPU. We report experiments that show that it makes warp scheduling predictable with little performance costs.**

***Index Terms*—RISC V, GPU, real time, timing predictability, warp scheduling**

I. Introduction

The considerable computing power of GPUs makes them prime candidates for the deployment of increasingly complex applications in the field of embedded systems, particularly autonomous vehicles. The most time-critical tasks, however, are subject to a verification process to ensure that deadlines are always met. Among other things, this process evaluates the worst-case execution and response times (WCET/WCRT) of critical software tasks.

Measurement-based methods can hardly offer execution time guarantees as soon as the analyzed system (hardware and software) is so complex that we cannot be sure of covering all possible scenarios. This is why static analysis techniques are preferred whenever possible. Such techniques have been widely studied for standard processors (CPUs) [1], but much less so for GPUs, which feature a very specific execution model (SIMT model, i.e. execution of a set of threads – a warp – in lockstep mode) that requires specific analyses. In recent work [2], an approach to represent the code of GPU programs in a form comparable to a control-flow graph, taking possible branch divergence within a warp into account, has been proposed. The aim was to support the analysis of the execution time of a warp executed in isolation. In this paper, we focus on the analysis of how several active warps are scheduled and how this impacts the execution time of the program. Our goal is to ensure that warp scheduling is predictable for static timing analysis.

Off-the-shelf GPUs are extremely poorly documented (even less so than standard processors). As a consequence it is very difficult to figure out the details of their microarchitecture and of their scheduling strategies. In this paper, we consider an open-source GPU architecture based on the RISC-V ISA: Vortex [3]. This allows us to consider hardware solutions, implement them and evaluate their impact on performance.

The paper is organized as follows. Section II is devoted to warp scheduling: general principles are outlined and the state of the art in this field is reviewed. In Section III, we pose the problem: what makes warp scheduling unpredictable for static analysis? Section IV introduces our proposed solution. Its implementation in Vortex and experimental evaluation are presented respectively in Sections V and VI. Section VII concludes the paper.

II. Overview of GPUs and warp scheduling

A. Background on GPUs

We first give an overview of how a GPU works. On the hardware side, a GPU is composed of a large number of Streaming Multiprocessors (SM), as illustrated in Figure 1. Each SM is capable of handling a large number of threads and executing one (or more) group(s) of threads in lockstep mode, thus implementing the SIMT (Single Instruction Multiple Threads) execution model. Such a group of threads is called a *warp* and is typically composed of 32 threads.

On the software side, an application is initially executed on the CPU and invokes *kernels*, that is functions that are to be run on the GPU. To favor concurrency, kernels can be launched in *streams*. Each kernel is to be executed by a very large number of threads, organized into *blocks* or *workgroups*. Each block is assigned to a given SM and its threads share the resources provided by that SM (functional units and internal shared memory).

B. Warp scheduling strategies

A complete analysis of scheduling mechanisms implemented in Nvidia GPUs is provided in [4]. It highlights the presence of a hierarchy of schedulers that determine the order in which the following elements are executed: streams, kernels, blocks and warps. In this paper, we focus on the scheduling of warps.

Various white papers published by Nvidia (e.g. [5]) show one or more warp schedulers in each SM. As far as we know, whether the schedulers share the same warp queue or have

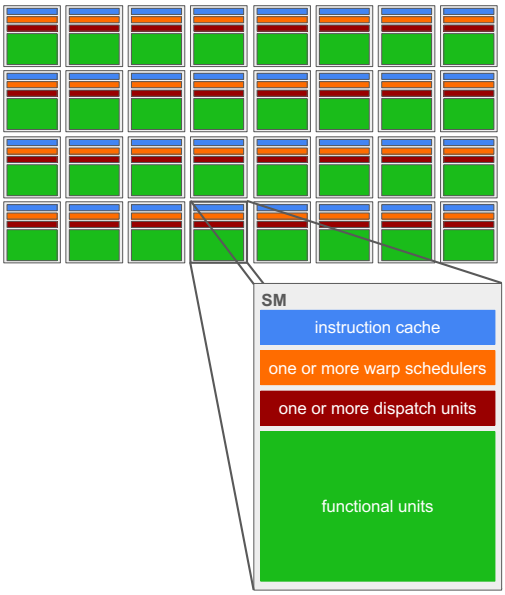

Fig. 1. Generic architecture of an Nvidia GPU

separate queues is not documented. In this paper, we restrict ourselves to a single warp scheduler per SM.

Several papers discuss the policy followed by the warp scheduler: some introduce new policies and analyze their performance [6]–[13]; others apply a retroengineering approach to identify the policy implemented in off-the-shelves GPUs [4], [14], [15].

Before examining the various scheduling policies considered in the literature, we point out that papers do not have all the same view: some of them consider warp scheduling in the fetch stage [7], others in the issue stage [9], [10]. We argue that both stages need a scheduler to select the next warp to process, raising the question of the synhronization between the two schedulers.

The terminology used to name scheduling policies is not always consistent among different papers. Here we specify our own conventions. Different warp selection algorithms can be implemented in a scheduler:

- *Round-Robin*: warps are selected in a fixed order.
- *Oldest-First*: the oldest ready warp is selected.
- *Icount*: the warp with the fewest number of fetched but not yet issued instructions is selected. The underlying objective is to favor fast executing warps [6], [16].

Orthogonally to these algorithms, we distinguish four main categories of policies:

- *Strict*: if the warp selected by the algorithm at a given cycle is not ready, the stage is idle during that cycle. One possible reason why a warp is not ready in the fetch stage is that it is waiting for a branch to be resolved. In the issue stage, a warp is not ready if e.g. its operands are not available. A new warp is selected at each cycle.
- *Loose*: if the selected warp is not ready, its turn is skipped and the next warp (according to the algorithm) is selected. A new warp is selected at each cycle.
- *Greedy*: unlike the previous two categories, the warp selected at a given cycle remains selected until it is stalled [8]. A warp might be stalled in the fetch stage when it triggers a cache miss, or in the issue stage when it executes a long latency operation.
- *Adaptative*: the greediness of the scheduler is dictated by external considerations such as the relative progress of different warps [6].

According to [4], Nvidia's Maxwell, Pascal, Volta and Turing architectures use the Loose Round Robin (LRR), which refers to *Greedy then Loose Round Robin* (GTLRR) in our classification: the selected warp runs until it is blocked by a long-latency operation; control is then passed to the next warp, in round-robin order; if a warp is not ready to execute, its turn is simply skipped. Other work [13] assumes a Greedy Then Oldest (GTO) policy [8] (*Greedy then Loose Oldest* or GTLO in our terminology) whereby a warp that becomes stalled transfers control to the oldest ready warp.

III. Predictability of warp scheduling

The advent of GPUs in embedded systems is extending to mission-critical systems. Timing predictability is therefore a key issue in the perspective of certification.

A. Static timing analysis of a GPU-executed kernel

In its simplest form, a kernel is executed by a set of thread blocks, each organized into warps (groups of threads executing in lockstep). Determining the execution time of a kernel therefore requires calculating: (i) the execution time of a warp in isolation; (ii) the makespan of the execution of all warps assigned to the same SM, taking into account the scheduling policy; (iii) conflicts between accesses to the shared memory (intra-SM); (iv) interference in the global memory between warps executing on different SMs. Very little work has been published on these topics, and much remains to be done. The approach proposed in this paper is intended to facilitate the computation of component (ii). The other components are outside the scope of this paper.

We assume that component (i) can be computed, although we know that work is still in progress. First, the executable program has to be decoded, which is already a challenge when the instruction set is closed [17]. Next, all possible execution paths must be identified, which is akin to building a CFG for a CPU program [2], [18], [19]. This enables identifying branches and their direction associated with each path. A precise hardware model must be established to specify instruction execution times in functional units [20] and an analysis of cache behavior can be used to determine the latency of each access (hit or miss) [21]. Although we are not aware of any comprehensive tool for determining the execution time of a warp in isolation, we consider that existing work argues in favor of its feasibility in the short term (for documented GPUs) and we focus instead on warp scheduling.

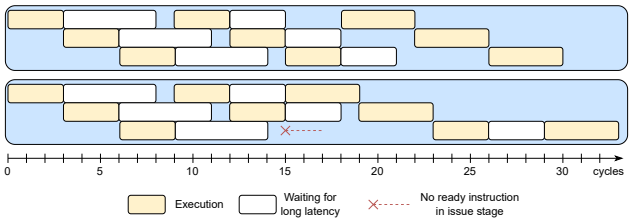

Fig. 2. Example of uncoordinated fetch and issue policies with a LRR issue

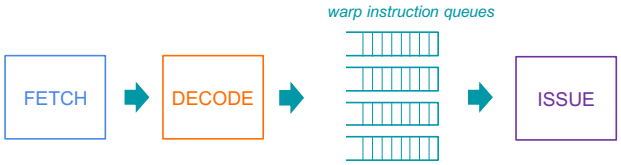

Fig. 3. Pipeline of a generic GPU

B. Analyzing the interleaving of warps in an SM

A few papers examine how warps are scheduled within an SM [22], [23]. The focus is on the issue, assuming a Loose Round Robin (LRR) warp scheduling policy – in [23] the scheduling is additionally greedy. The two papers derive the makespan of a set of warps executing the same kernel and mapped to the same SM. In [22] simplifying assumptions are done on the executed code (considering only two types of instructions: calculation or memory access) and an integer linear program (ILP) formulation is built to compute the schedule of warps. In [23], the makespan is derived based on equations that express the scheduling model and on timing estimations provided by a static analyzer.

We underline that both papers assume that the selected warp is always ready (its instruction queue is not empty), disregarding, this way, the fetch policy. In this paper, our point is that this assumption is not always valid.

In practice, fetched and decoded instructions are stored in the instruction queue of the warp they belong to. A warp cannot be eligible for issue if its queue is empty. In this case, the scheduler may select another warp (Loose strategy). This means that the effective scheduling of warps at issue is not completely dictated by the intended policy but also depends on the pace at which instructions are fetched. This depends in turn on the fetch scheduling policy but also on other factors such as the instruction cache behavior and the memory latency. This is not an issue from a functional perspective. However, when it comes to accurately predicting the actual order of execution of the various warps in a block to derive the execution time of the application, this becomes problematic because the scheduling policy is not respected when a warp runs out of fetched instructions. Such a situation is illustrated in Figure 2. Three warps are depicted in two configurations, with a GTLRR scheduler at the issue stage. The execution of each warp is modeled as a sequence of execution cycles (in yellow) in which the warp issues an instruction and of idle cycles (in white) in which the warp is not ready to issue because it is waiting for the result of a long latency operation (e.g. a memory load). At the top, it is assumed that instructions are always available when a warp is selected by the warp scheduler at the issue stage. Consequently, the schedule rigorously follows the GTLRR policy. On the contrary, at the bottom, it is assumed that the third warp has no available instruction in its queue at cycle 15, forcing the GTLRR scheduler to schedule the first warp instead. This results in a different schedule with a significant impact on the execution time.

In section VI, we present measurement results that show that this phenomenon is not anecdotal. For our baseline GPU, the fraction of cycles during which the issue scheduling policy is not strictly respected because the warp that was initially selected has no ready instruction ranges from 0.10% to 81.12% depending on the benchmark and the scheduling policies.

In this paper, we propose a way to coordinate fetch and issue schedulers to ensure that warps are actually executed in the intended order. This will contribute to make static analysis of thread block execution times possible.

IV. Coordinating Instruction Fetch and Issue

A. Pipeline model

Figure 3 shows the front end of a generic GPU's pipeline, inspired from [24]. The first stage requests one instruction per cycle to the instruction cache. Instruction requests are pipelined and have a latency of t_F cycles in case of a *cache hit*. The second stage decodes one instruction per cycle, and inserts it into the instruction queue of the warp it belongs to. In the issue stage, one instruction is selected for execution among the heads of the instruction queues. An instruction is said to be *ready* for execution when its operands are available (data dependencies are resolved) and the required functional unit is free.

Modern GPUs do not implement branch prediction: there are generally very few branches in GPU code and the cost of a branch predictor is not justified, since the control can easily switch to another warp while waiting for a branch to be resolved. However, given that a branch must reach the decode stage to be identified as such (and possibility suspend the instruction fetch for the warp), it seems reasonable to assume a basic scheme that continues to load instructions in sequence when a branch is decoded and simply invalidates them in the queue if the branch is resolved as *taken*. This amounts to a static *not-taken* branch predictor.

B. Coordinated warp scheduling

In order to guarantee that the intended issue scheduling policy is strictly applied, i.e. that warps are issued in the expected order, it must be enforced that the selected warp is never short of instructions in its instruction queue. Our solution to achieve this is threefold: (i) using instruction queues of appropriate length, (ii) building a common scheduler for the fetch and issue stages, (iii) inserting NOPs whenever needed to avoid instruction shortage.

1) Capacity of the instruction queues: First of all, it is necessary to ensure that the instruction queue capacity is sufficient to hide the latency of loading an instruction from the cache. We assume a pipelined cache where an instruction can be requested each cycle and becomes available after t_F cycles. If the duration of the decode stage is one cycle, that means that the first instruction of a warp can be ready only t_F+1 cycles after the warp has been selected by the scheduler. If we want to ensure that the warp can execute an instruction as soon as it is selected, we need an instruction queue of capacity $t_F + 1$, initialized with as many NOPs. Those NOPs can be taken into account at analysis time. Later, each time the warp is selected, one instruction is popped of the queue to be executed (if ready) and a new instruction is fetched. This way, the instruction queue of a warp always contains the same number of instructions. Note that whether the head instruction is ready or not can be anticipated by static analysis.

2) A unique warp scheduler: Since we want instructions of a warp to be fetched at the same pace as they are issued (so that inputs and outputs to/from a warp instruction queue are balanced), it seems natural to consider the same warp scheduling policy in both stages. However, having two identical but separate schedulers would not be enough to reach our goal: the selection of a warp in the issue stage depends not only on the scheduling policy, but also on which instructions are ready (with available operands and a free functional unit). For this reason, the warp selected for execution may not be the first one dictated by the policy, but one of the following warps. To ensure that the same decision is taken in the fetch stage, it is desirable to implement a unique warp scheduler that is shared by the fetch and issue stages.

3) Compensating breaks in the instruction flow: There are two reasons why the fetch stage would not be able to sustain the pace imposed by the common warp scheduler: (i) when a request to the instruction cache results in a cache miss; (ii) when the instruction flow contains a taken branch.

In case of a cache miss, the instruction fetch must be suspended for the warp, meaning that the warp is not scheduled anymore until the cache miss is resolved. It is thus necessary to notify the cache miss to the warp scheduler. However, (i) the warp still issues an instruction in the current cycle and (ii) when the corresponding cache block is loaded, the warp has to fetch the instruction again. To compensate for the not-yet-fetched instruction, a NOP is inserted in the warp instruction queue. As mentioned before, part of the static analysis process consists in characterizing the cache behaviour for the kernel code. It is then possible to anticipate cache misses and the resulting deviations from the warp scheduling policy, as well as additional instruction fetch latencies. The NOP that replaces the not-yet-fetched instruction ensures that the warp does not lack a ready instruction when scheduled again later. As explained above, we assume static branch prediction where instructions are fetched in sequence until a branch is executed and resolved as *taken*: the instructions fetched along the wrong path must be flushed and the instruction fetch must resume from the branch target address. In order to keep the warp instruction queue fed with instructions, our solution replaces flushed instructions with NOPs instead of simply removing them from the queue. Again, static analysis is able to anticipate branch mispredictions and to predict the number of inserted NOPs.

V. Example implementation

In this section, we report how we have implemented our approach towards timing-predictable GPUs in a forked version of the open-source Vortex [3].

A. Vortex: an open-source RISC-V-based GPU

Long considered unrealistic, designing your own processor is now a real possibility, thanks in particular to the success of the RISC-V ecosystem. The open-source GPU named Vortex [3] is part of this movement, and offers interesting prospects compared to the closed, poorly documented devices from Nvidia. Vortex supports an extended version of the standard RISC-V ISA. It includes six new instructions that enable SIMT (Single Instruction Multiple Threads) execution for up to 1024 threads on 64 cores and control synchronizations. Its Verilog model can be synthesized on an FPGA.

B. Our baseline GPU: a variant of Vortex

To get closer to the pipeline model presented above, we have produced a variant of Vortex (v2) and made the following modifications:

- In the original version of Vortex (v2), a new instruction for a warp cannot be fetched until the previous instruction has been decoded (no branch prediction). While this naturally favors timing predictability, it also limits performance when combined to a 4-cycle instruction cache hit latency and a limited number of warps (32). We have implemented static (*not-taken*) branch prediction to allow fetching one instruction per cycle, possibly several cycles in a row for the same warp. In case of a taken branch, instructions fetched on the wrong path are simply discarded.
- We have modified the instruction cache so that it is now able to signal a cache miss one cycle after a request has been submitted. This is needed to implement our approach.
- Additional warp scheduling policies have been implemented in both the fetch and issue stages. In the original Vortex GPU, warps are scheduled by a *Fixed-Priority* policy in the fetch stage and by a LRR (*Loose Round-Robin*) policy in the issue stage. We have added the (*Loose Round Robin*), GTLO (*Greedy Then Loose Oldest*) and GTLRR (*Greedy Then Loose Round Robin)* policies to both stages.

All the experimental results labeled *baseline* have been collected for this variant of Vortex.

TABLE I
BENCHMARKS

Name	Function	Source	#instr
blackscholes	Black-Scholes formula	NVIDIA SDK	192,052
hotspot	physics simulation	Rodinia	17,554
hotspot3D	physics simulation	Rodinia	39,971
kmeans	k-means clustering	Rodinia	164,077
psort	parallel sorting	Vortex	492,237
sgemm	matrix multiplication	Vortex	1,728

C. Implementing our approach

We have applied the recommendations given in Section IV to our *baseline* device to produce the *SWaS (Synchronized Warp Scheduling)* variant. Given the latencies of the fetch (with cache hit) and decode stage (resp. 3 and 1 cycles), we have set the lengths of the instructions queues to 4. At start-up, instructions queues are filled with NOP instructions.

We have merged the two warp selection functions into a single scheduler used synchronously by the fetch and issue stages: at each cycle, a warp is selected and (i) the head instruction of its instruction queue is issued while (ii) a new instruction is requested to the cache for this warp. In case of a cache miss, a NOP is sent to the decode stage and the warp is temporarily suspended: this is made possible by notifying the scheduler when the miss occurs as well as when it is resolved.

Finally, we have modified the way branch mispredictions (in case of a taken branch) are handled, so as to replace any flushed instruction by a NOP, avoiding this way to create bubbles in the instruction queue.

In the following section, we report experiments that validate our approach by highlighting that the scheduler behaves in a way that static timing analysis can anticipate. We also examine the impact on performance of SWaS variant compared to *baseline* GPU.

VI. Experimental evaluation

A. Evaluation methodology

1) Benchmarks: We use six benchmarks from different suites: Rodinia [25], the NVIDIA SDK and a collection embedded in the Vortex project. They are listed in Table I. Each kernel is executed by 32 warps of 32 threads. They all perform floating-point computations, which have long latencies: 4 cycles for `fmadd`, `fmul` and `fadd`, 5 cycles for `fcvt` and 16 cycles for `fsqrt` and `fdiv`.

2) Experimental setup: All benchmarks are run on an RTL simulator of Vortex[1] built with Verilator[2]. Vortex is configured to include a single SM with enough resources to run at most 32 warps of 32 threads concurrently. The DRAM is emulated using ramulator[3] [26]. OpenCL kernels are compiled to executable code to be run by a single so-called *workgroup* and this code is dispatched onto the GPU by a custom launcher. This launcher uses custom RISC-V CSRs (*Control and Status Registers*) to register basic metrics such as the number of cycles needed to run the kernel or the number of committed instructions (the list of metrics is given in Section VI-A5).

3) Scheduling policy configurations: Nine configurations of scheduling policies are considered or our baseline GPU – three variations (GTLRR, GTLO and LRR) in the fetch stage combined to the same three variations in the issue stage.

Whatever the fetch scheduling policy, a warp is considered *not fetch-ready* either when its previous request resulted in a cache miss *or* when its instruction buffer is full. Whenever the selected warp is not fetch-ready, its turn is skipped and the next warp (in the sense of the scheduling policy) is selected. In the issue stage, a warp is said *not issue-ready* when the operands of the instruction at the head of the instruction buffer are not yet available (data dependency) *or* the instruction buffer is empty.

The simplest policy in the fetch (resp. issue) stage is LRR: it fetches (resp. issues) one instruction for another warp at each cycle. Greedy policies, GTLRR and GTLO, keep instead the same warp as long as it is fetch- (*resp.* issue-)ready. With GTLRR, the next warp is determined with a round-robin algorithm (as for LRR), while GTLO gives priority to the oldest warp (warp 0, then warp 1, etc. since warps are numbered in the same order as they are activated).

4) Scheduling errors in the baseline GPU: To evaluate the predictability of warp scheduling, we want to measure the number of discrepancies between the intended and effective (observed) warp scheduling policies. For this purpose, we augment the design of the GPU with two oracle warp schedulers at issue stage: `osched1` and `osched2`. The first one, `osched1`, considers that instruction buffers always contain ready instructions: it is used to simulate the intended scheduling policy. Any discrepancy in the selected warp when using `osched1` instead of the implemented warp scheduler means that the policy is not strictly respected and is accounted for (at each cycle). For greedy scheduling policies, `osched1` is synchronized with the implemented scheduler after a discrepancy has been observed: this way, this discrepancy is accounted for only once. Discrepancies are due to three reasons: (i) cache misses – due to their long latency, they prevent the instruction buffer to be fed for a while, (ii) mispredicted branches – the instruction buffer is flushed out after a branch is found mispredicted, and (iii) absence of synchronisation between the fetch and issue warp schedulers. Since situations (i) and (ii) can be anticipated at analysis time, only situations (iii) are to be considered as deviations of the scheduling policy. To identify them, we use `osched2` that considers that the selected warp always has a ready instruction in its buffer except if it has experienced a cache miss less than (1 + *cache_miss_latency* + *cache_hit_latency*) cycles ago or a taken (then mispredicted) branch less than (1 + *cache_hit_latency*) cycle ago (the scheduler keeps track of this information thanks to internal signals).

5) Metrics: We define here the metrics collected during experiments (only during the execution of the OpenCL kernel,

[1]https://github.com/vortexgpgpu
[2]https://veripool.org/verilator/
[3]https://github.com/CMU-SAFARI/ramulator

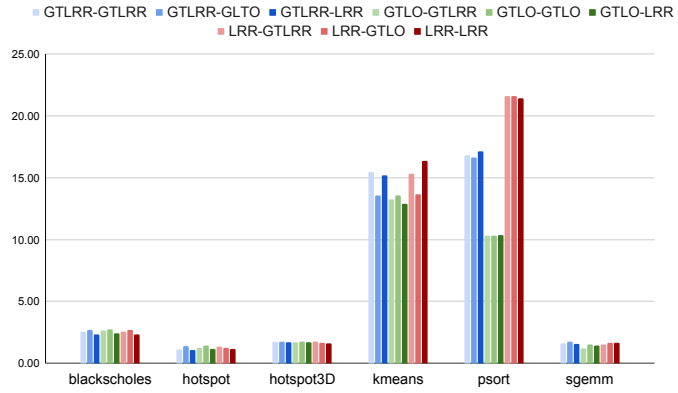

Fig. 4. IPC of the baseline GPU

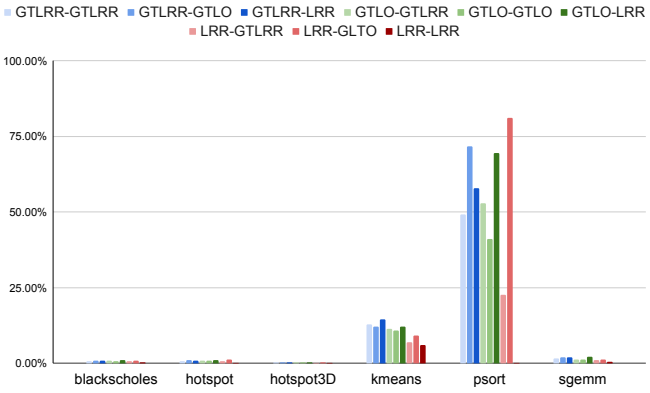

Fig. 5. Scheduling error rate in the baseline GPU

i.e. we ignore the prologue code):

- *cycles*: duration (in cycles) of the kernel execution, that is the number of cycles during which at least one thread is still executing the kernel. Since warps are launched one at a time with a few-cycle-gap between two warps, *cycles* is a few cycles longer than the execution time of a single warp.
- *committed*: number of instructions committed by any thread that executes the kernel. This number is given as *#instr* in Table I. It does not include the NOPs inserted with our approach.
- *IPC*: number of instructions committed per cycle.
  $$IPC = committed/cycles$$
- *discrepancies*: number of scheduling discrepancies, i.e. number of cycles during which the intended issue warp scheduling policy is not respected (discrepancy with `osched1`). This number is for the whole set of warps.
- *errors*: number of scheduling errors, i.e. number of cycles during which the intended issue warp scheduling policy is not respected, excluding predictable situations such as cache misses and branch mispredictions (discrepancy with `osched2`). This number is for the whole set of warps.
- *error_rate*: percentage of cycles for which an error is observed.
  $$error\_rate = errors/cycles$$

B. Experimental results

1) Behaviour of our baseline GPU: The performance (IPC) of our baseline GPU is plotted in Figure 4. Each bar stands for a configuration of fetch-issue scheduling policies. The theoretical upper bound on the IPC is 32: it would be reached if an instruction was committed at each cycle by one of the warps, that is by 32 threads. In practice, it ranges from 1.07 to 21.62 over our set of benchmarks.

The high IPC for `psort` can be explained as follows. Each thread compares one element of the array to be sorted to every other element, in a loop. Within one iteration, all threads in the same warp access the same element and are then served by a single memory access. The same happens for `kmeans` where threads in the same warp consider the same cluster at a time as a possible target for their point. For `hotspot` and `sgemm`, memory accesses cannot be coalesced, which adversely affects performance. These remarks are confirmed by the measured average latency of memory loads: it is significantly shorter for `kmeans` and `psort` than for the other programs. For instance, with the LRR-GTLRR configuration, the average latency is respectively 18 and 37 cycles for these two applications and ranges from 68 to 98 cycles for the other ones. The low IPC for `blackscholes` is related to (non pipelined) divisions that have a 16-cycle latency

On a more general note, this poor performance is also partly due to the fact that we make a restricted use of the Vortex architecture: because we want to put the focus on warp scheduling, and avoid interference from other levels of scheduling, we use a single core and we run a single workgroup (equivalent to a block in Cuda). With such a configuration, papers about the original Vortex (e.g. [3]) report similar IPC.

The impact of the scheduling policy configuration on the IPC is globally not significant. However, the diagram shows that the combinations for which the fetch policy is GTLO (green bars) result in a twice lower IPC for `psort`. This can be explained by the large number of branch mispredictions with greedy fetch policies (and particularly for GTLO): since several instructions are fetched for the same warp in a row, this does not give enough time for a branch to reach the execution stage before the next instruction is fetched. Taken loop branches are then often mispredicted.

Over the set of configurations, LRR-GTLRR and LRR-GTLO are those that globally produce the highest IPCs.

Figure 5 displays the scheduling error rate, that is the fraction of cycles during which a divergence between the theoretical and effective scheduling decisions has been observed. For some benchmarks, such as `psort` and `kmeans`, the error rate is significant and the schedule of warps considered by static analysis is expected to differ significantly from the runtime behavior. However, the low error rate for the other

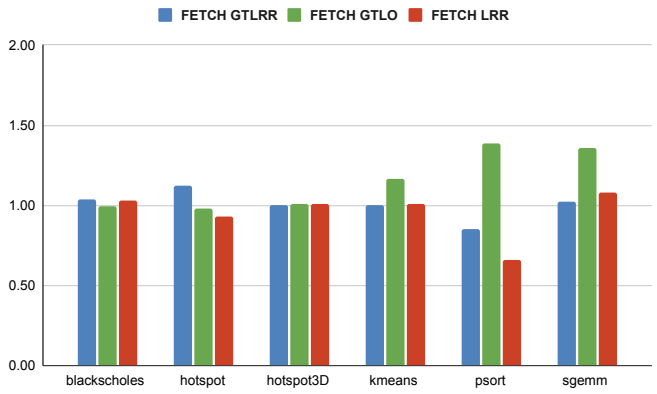

Fig. 6. Comparison between baseline and SWaS with GTLRR issue/synchronized scheduling policy

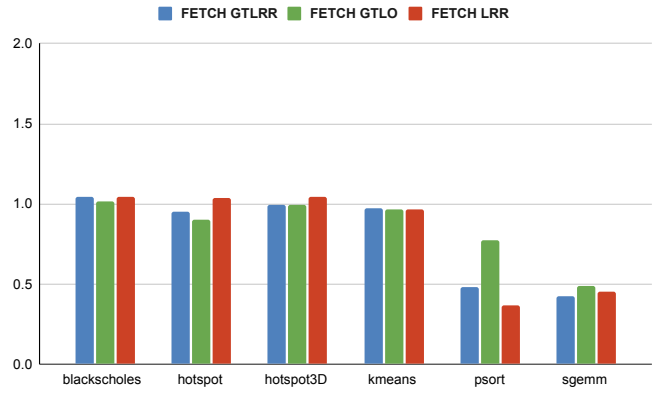

Fig. 7. Comparison between baseline and SWaS with GTLO issue/synchronized scheduling policy

benchmarks should not be underestimated. Indeed, any error is likely to invalidate the results of static analysis, which therefore cannot be guaranteed. For `hotspot3D`, that seems to exhibit the lowest rate, the raw number of errors ranges from 764 to 3,037, depending on the scheduling configuration, which is obviously far from being null. Interestingly, the benchmarks that face the highest error rates are those that show the highest IPC: because they execute instructions fast, they are more likely to empty their instruction buffer before it is refilled by instruction fetch. Conversely, the agility of the scheduler that selects another warp whenever an instruction buffer is empty contributes to increasing the IPC.

2) Behaviour of the SWaS variant: With the SWaS variant, that implements synchronized fetch/issue warp scheduling, no scheduling error is observed during the execution of the benchmarks, whatever the scheduling policy. Instruction buffers are always 100% full (sometimes with NOPs).

Figures 6 to 8 depict the ratio between the IPCs of the SWaS variant and that of the baseline GPU: a value lower than 1 indicates a loss of performance with SWaS. The three diagrams consider the same issue scheduling policy (which

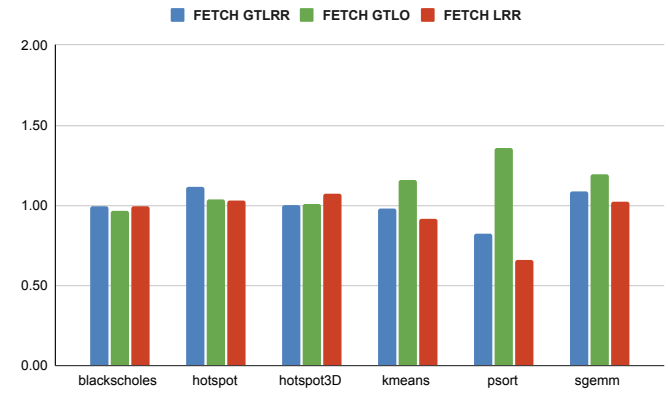

Fig. 8. Comparison between baseline and SWaS with LRR issue/synchronized scheduling policy

is also the fetch policy in SWaS, since it implements a single scheduler) for both variants. Bars correspond to different fetch policies in the baseline GPU.

Variations on the IPC between the two variants are due to two opposing phenomena. On the one hand, the synchronization of fetch and issue makes kernel execution smoother, since instructions are fetched at the same pace as they are executed. This is likely to increase the IPC. On the other hand, NOPs that are inserted in buffers to hide breaks in the instruction fetch consume execution resources and bandwidth, which lower the IPC since NOPs are not accounted for as committed instructions.

For all the benchmarks but `psort`, the impact on performance seems reasonable: the IPC is either slightly degraded or improved in the SWaS variant. This is not the case for `psort`, for which the performance cost is more significant for certain configurations: down to -63.0% for the GTLO issue policy combined to the LRR fetch policy in the baseline variant. This is because the main loop is too short (18 instructions) to hide the impact of the 4 NOPs added for the branch taken at each iteration.

The key point here is that the aim of our approach is not to improve IPC, but to make scheduling predictable. This objective is achieved, since no errors are detected: the scheduler behaves exactly as defined by the implemented policy. Contingent performance loss should then be seen as the price to pay to enable precise and safe static timing analysis.

In Table II, we compare the IPC of the two GPU variants with their best scheduling configuration. As pointed out earlier, the baseline GPU reaches highest IPC values with the LRR-GTLRR configuration (very close to the other combinations with LRR as fetch policy). For the SWaS variant, GTLRR leads to the best results. The largest performance degradation (-34%) is observed for `psort`. The number of errors with the baseline variant is significant (165,011) which underlines the necessity to address the predictability of warp scheduling. Note that synchronizing scheduling shows positive side effects on the IPC of other benchmarks.

TABLE II
COMPARISON BETWEEN SWAS GTLRR AND BASELINE LRR-GTLRR

Benchmark	IPC SWaS/baseline	# errors
blackscholes	1.02	16,657
hotspot	0.92	3,147
hotspot3D	1.00	1,662
kmeans	1.01	23,643
psort	0.66	165,011
sgemm	1.08	403

VII. CONCLUSION AND FUTURE WORK

Accurate static timing analysis requires both full knowledge of the hardware platform on which the system is deployed and predictable execution mechanisms. Precise information on the hardware is difficult to collect when using commercial CPUs and GPUs. Open hardware gives access to all the details (even if the effort involved in digging the Verilog code is considerable) and is an asset for better analysis of execution times. The use of Vortex, a RISC-V-based GPU, for the work presented in this paper has been of valuable support.

CPUs and GPUs often feature mechanisms in which the behavior is not fully predictable. This is particularly true for mechanisms that exploit the dynamic state of the system to achieve better performance. In a GPU, the warp scheduler is usually work-conserving: when the selected warp is not ready for execution, it selects another warp. As a result, the way warps are scheduled differs from what the scheduling policy dictates. This is very difficult, if not impossible, to predict at analysis time because this would require analysing the whole system (i.e. all warps) together and this is usually not tractable.

In this paper, we propose an elegant approach to make warp scheduling predictable, that is to ensure that the warp scheduler behaves exactly as expected from the scheduling policy. It consists of a unique synchronized warp scheduler with artificial filling of instructions buffers to avoid stalls. This way, it is possible to derive offline the makespan of the execution of all the warps/threads that execute a kernel. This approach is implemented as a variant of Vortex (SWaS). Experiments show that the approach is reliable: the scheduling policy is strictly respected. Interestingly, the cost in terms of performance is low.

This work can be considered a proof of concept and the approach could be applied in future COTS GPUs if timing predictability is a concern. As future work we plan to investigate complementary and alternative architectural solutions to improve timing predictability beyond warp scheduling. We will also consider multiple SM and multiple schedulers per SM.

Acknowledgement. *This work was funded by the French National Research Agency under the ANR-21-CE25-0012 reference.*

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embedded Computing Systems*, vol. 7, no. 3, 2008.

[2] L. Jeanmougin, P. Sotin, C. Rochange, and T. Carle, "Warp-Level CFG Construction for GPU Kernel WCET Analysis," in *Workshop on Worst-Case Execution Time Analysis (WCET)*, 2023.

[3] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[4] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[5] "Nvidia a100 tensor core gpu architecture," www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, retrieved in April 2024.

[6] N. B. Lakshminarayana and H. Kim, "Effect of instruction fetch and memory scheduling on GPU performance," in *Workshop on Language, Compiler, and Architecture Support for GPGPU*, vol. 88, 2010.

[7] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Intl Symp. on Microarchitecture*, 2011.

[8] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *IEEE/ACM International Symposium on Microarchitecture*, 2012.

[9] ——, "Divergence-aware warp scheduling," in *IEEE/ACM International Symposium on Microarchitecture*, 2013.

[10] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," *ACM Computer Architecture News*, vol. 43, 2015.

[11] J. Anantpur and R. Govindarajan, "PRO: Progress Aware GPU Warp Scheduling Algorithm," in *IEEE Intl Parallel and Distributed Processing Symposium*, 2015.

[12] "Locality based warp scheduling in GPGPUs," in *Future Generation Computer Systems*, 2018.

[13] J. Singh, I. S. Olmedo, N. Capodieci, A. Marongiu, and M. Caccamo, "Reconciling QoS and Concurrency in NVIDIA GPUs via Warp-level Scheduling," in *Design, Automation & Test in Europe (DATE)*, 2022.

[14] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *Real-Time Systems Symposium (RTSS)*, 2017.

[15] N. Otterness and J. H. Anderson, "Exploring AMD GPU scheduling details by experimenting with "worst practices"," *Real Time Systems*, vol. 58, no. 2, 2022.

[16] H. Levy, J. L. Lo, J. Emer, R. Stamm, S. Eggers, and D. Tullsen, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *Intl Symp. on Computer Architecture (ISCA)*, 1996.

[17] A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang, "Decoding CUDA Binary," in *IEEE/ACM Intl Symposium on Code Generation and Optimization (CGO)*, 2019.

[18] A. Betts and A. Donaldson, "Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[19] V. Hirvisalo, "On Static Timing Analysis of GPU Kernels," in *Workshop on Worst-Case Execution Time Analysis (WCET)*, 2014.

[20] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," 2018.

[21] Y. Huangfu and W. Zhang, "WCET Analysis of GPU L1 Data Caches," in *High Performance extreme Computing Conference (HPEC)*, 2018.

[22] K. Berezovskyi, K. Bletsas, and B. Andersson, "Makespan Computation for GPU Threads Running on a Single Streaming Multiprocessor," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[23] Y. Huangfu and W. Zhang, "Static WCET Analysis of GPUs with Predictable Warp Scheduling," in *International Symposium on Real-Time Distributed Computing (ISORC)*, 2017.

[24] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, *General-purpose Graphics Processor Architectures*. Morgan & Claypool, 2018.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[26] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, and O. Mutlu, "Ramulator 2.0: A modern, modular, and extensible dram simulator," 2023.