ENDSEM PROJECT

KERNEL AWARE WARP SCHEDULER (KAWS)

Ishan Mardani (21CS01023)

Akshit Dudeja (21CS01026)

Joshua Jose Dias Barreto (21cs01075)

Vishnu Tirth Bysani(21cs01077)

Tushar Joshi(21cs01078)

Table Of Contents

Introduction

- There are a lot of parallel logical units built into modern GPUs. A GPU's high level of throughput and remarkable compute power are its distinguishing characteristics.
- The ability of GPUs to instantly swap contexts to mask instructions with a long latency is the primary source of their processing power.
- A GPU often accomplishes this via a warp scheduler.

Introduction

- However, GPUs consistently display under-utilization of hardware resources in the absence of software optimisation.
- To increase resource utilisation, it is necessary to change various warp scheduling strategies during various kernel execution times.
- Our goal is to inform current warp schedulers about the kernel development so they can offer an efficient scheduling scheme.
- Additionally, it has been seen that sharing stalling warps with particular warp schedulers could increase resource utilisation for GPUs with numerous warp schedulers.

Background

- All existing warp scheduler schemes address different aspects of GPU architectures.
- Each warp receives the same priority from LRR. It successfully takes advantage of an inter-warp locality but due to the nature of continually shifting warps that occur with each cycle, it is unable to use intra-warp locality.
- Warps are prioritised more effectively in GTO. Because it keeps issuing the same warp until it stalls, then switches to the most recent warp to be issued, intra-warp locality is kept.

Background

- For the duration of the kernel execution, the majority of warp schedulers constantly use the same scheduling algorithm. However, during one period of kernel execution, one scheduling policy may be more effective than the other.
- It is inefficient to use the same warp scheduling policy throughout the whole kernel execution process.
- We are going to talk about 2 techniques to solve this problem. Before that, lets briefly go through the GPU architecture.

GPU Architecture

- Streaming Multiprocessors (SM) is equipped with an advanced memory hierarchy consisting of register files (RF), L1 cache, shared memory, L2 cache, and off-chip DRAM.
- There are typically two or four warp schedulers in an SM.
- An operand collector unit (OCU) is assigned to a warp instruction when it is issued. This is performed to load a source operand value for it.
- When all the required operands are collected, the instruction is ready to be issued to the SIMD (single instruction, multiple data) execution unit.

GPU Architecture

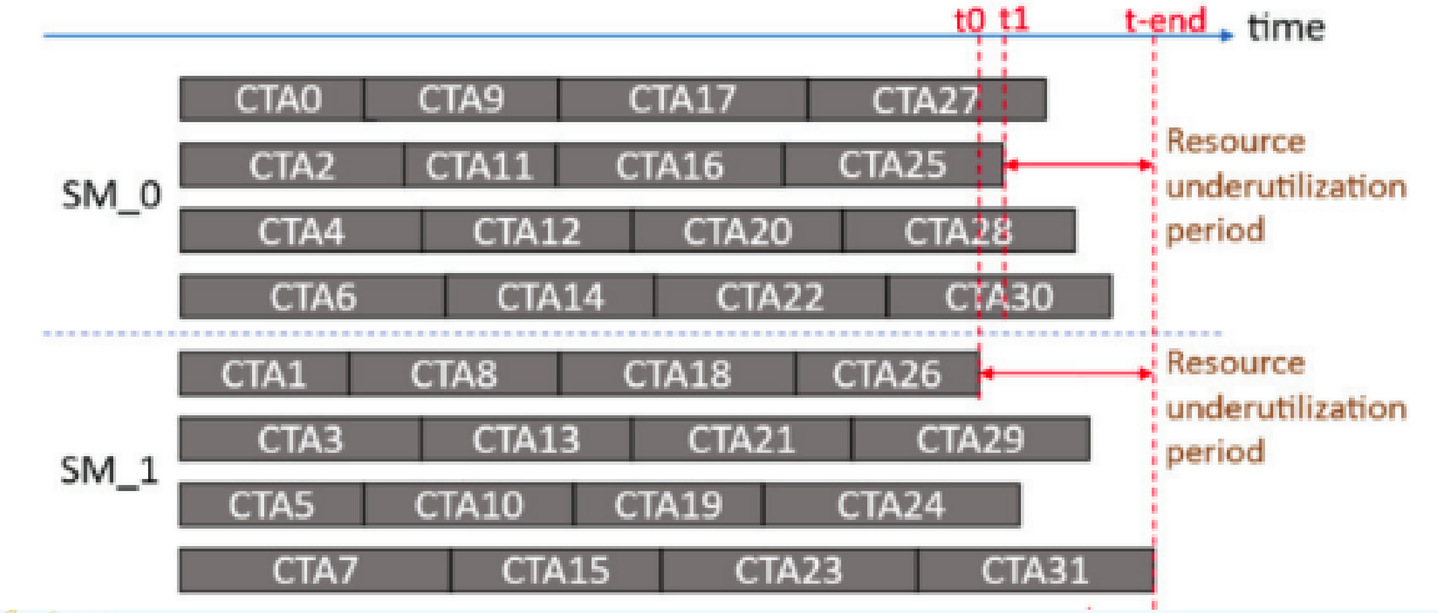
- SIMD consists of streaming processor (SP), special function unit (SFU), and memory (MEM).
- The SP unit executes ALU, INT, and SP operations. The SFU unit executes double-precision (DP), sine, cosine, log, and other operations. The MEM unit is responsible for load/store operations.
- Each unit has an independent issue port from the OCU. Therefore, the warp scheduler cannot issue new warp instructions corresponding to an execution unit if the OCU of that unit is stalled.

GPU Architecture

- The GPU programming model consists of one or several kernels. Each kernel groups the threads into cooperative thread arrays (CTAs). The threads within a CTA execute in a group of 32 threads called a warp.
- At the SM level, the CTA scheduler assigns CTAs to each SM in each clock cycle until it reaches a maximum number. The maximum number of CTAs that can concurrently run in an SM is determined by the number of threads per CTA.

Kernel Aware Warp Scheduling

For convenient demonstration, we have used a GPU with two SMs. Assume that a kernel contains 32 CTAs and that each SM can manage a maximum of four concurrent CTAs.



Kernel Aware Warp Scheduling

Whenever a CTA completes execution, a new CTA is assigned to the empty CTA slot in the next cycle. At time t_0, CTA26 completes its execution in SM_1. However, no new CTA remains in the kernel to fill the CTA slot. This leaves three CTAs to run freely in SM_1. The number of running CTAs in SM_1 after the completion of CTA24 and CTA29 remains two and one, respectively. This underutilization continues until the final CTA (CTA31) completes its execution. In SM_0, resource underutilization begins to occur from time t_1.

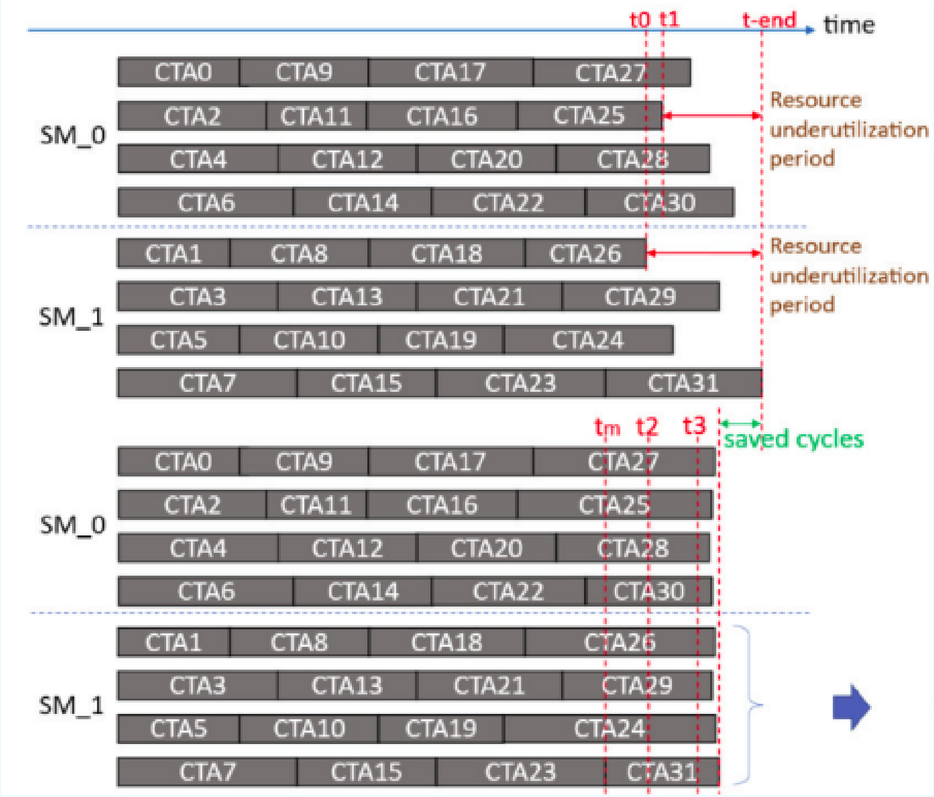
Kernel Aware Warp Scheduling

- From this example, we understand that resource underutilization occurs in SM_1 when any CTA in SM_1 (in this example: CTA26 at t_0) completes its execution after SM_1 receives the final CTA (in this example, CTA31) from the CTA scheduler.
- After the last CTA is issued, a more efficient scheduling policy is required to reduce resource underutilization.
- As soon as KAWS detects that the final CTA in the kernel has been issued, it switches to a progress-based prioritization policy.

Kernel Aware Warp Scheduling

- CTAs are prioritized based on the CTA progress. CTA progress is defined as the number of instructions issued from one CTA.
- Older CTAs (or CTAs issued earlier) tend to issue more instructions and are likely to be completed early. Therefore, we deprioritize these to speed up younger CTAs (or CTAs issued more recently).
- Thereby, CTAs issued more recently can be completed earlier whereas the execution time of previously issued CTAs can be prolonged.
- As a result, all the CTAs are adjusted for them to be completed almost simultaneously.

- As soon as CTA31 (last issued CTA in the kernel) is assigned (at time = t_m), a progress-based scheduling policy is applied.
- Initially, this policy provides the highest priority to warps that belong to CTA31 in SM_1.
- Then, the scheduler dynamically prioritizes warps based on the progress of CTA24, CTA26, CTA29 and CTA31 in SM_1.



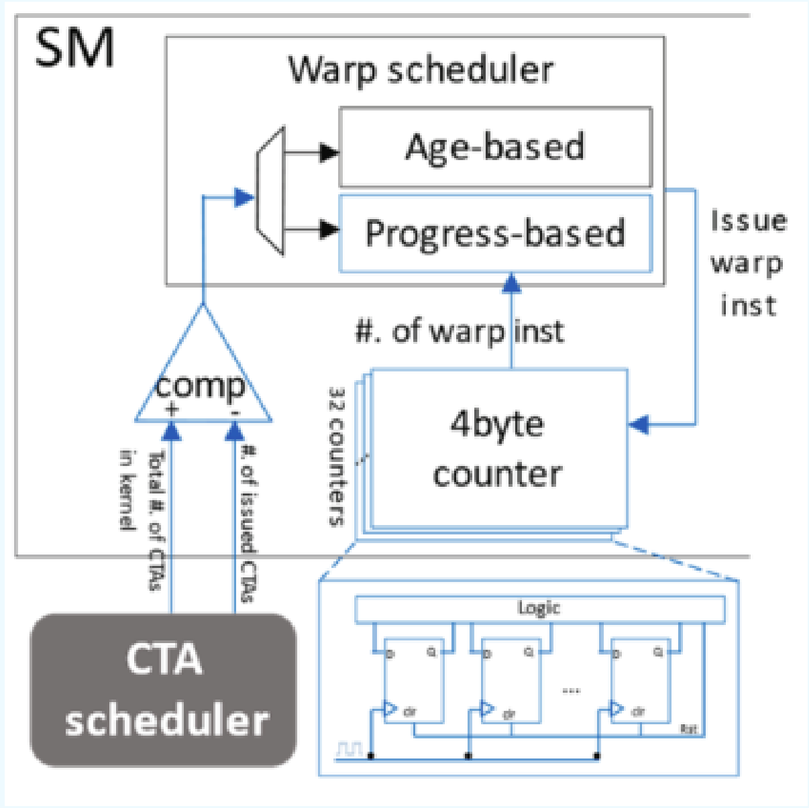
Kernel Aware Warp Scheduling

- Finally it is observed that all the CTAs in the SM are completed almost simultaneously, which prevents resource underutilization and accounts for several saved cycles.
- Note that underutilization depends significantly on the execution time of the last issued CTA in the kernel.

Hardware

- Since we have assumed to have 32 CTAs, we will use 32 instruction counters to maintain the entire CTA progress. We will also use a 4 B register per CTA to record the number of warp instructions issued during CTA execution. This consumes 4×32 bytes of additional storage.
- We use an additional comparator to be aware of the kernel execution. When the number of issued CTAs becomes equal to the total number of CTAs in the kernel the comparator sends a notification signal to the warp scheduler to switch to a progress-based scheduling policy.

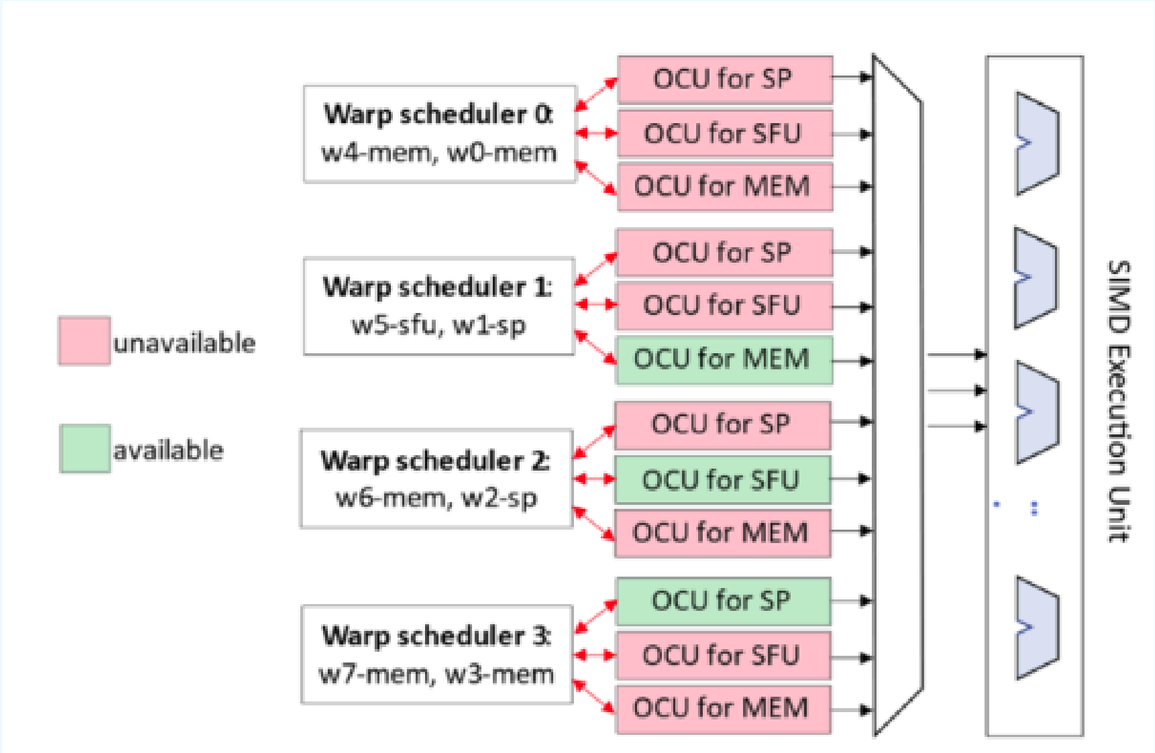
Hardware


SM
Warp scheduler
Age-based
Progress-based
Issue warp inst
#. of warp inst
comp
32 counters
4byte counter
Total #. of CTAs in kernel
#. of issued CTAs
Logic
CTA scheduler

Warp Sharing Mechanism

- A warp scheduler cannot issue new warp instructions if the OCU, which is supposed to load operands corresponding to that specific warp instruction, is unavailable.
- Owing to the multiple-warp-scheduler configuration, OCUs from different warp schedulers are likely to be available.
- Our concept is to utilize these to maintain the pipeline and prevent pipeline stall. This concept is called “warp sharing mechanism.”

- Assume that eight warps are scheduled in four warp schedulers.
- S0 denotes Warp scheduler 0, and w0-mem indicates that Warp 0 is carrying an instruction that would be executed in the MEM unit. Similarly, sp and sfu denote the instructions to be executed by the SP and SFU units, respectively.



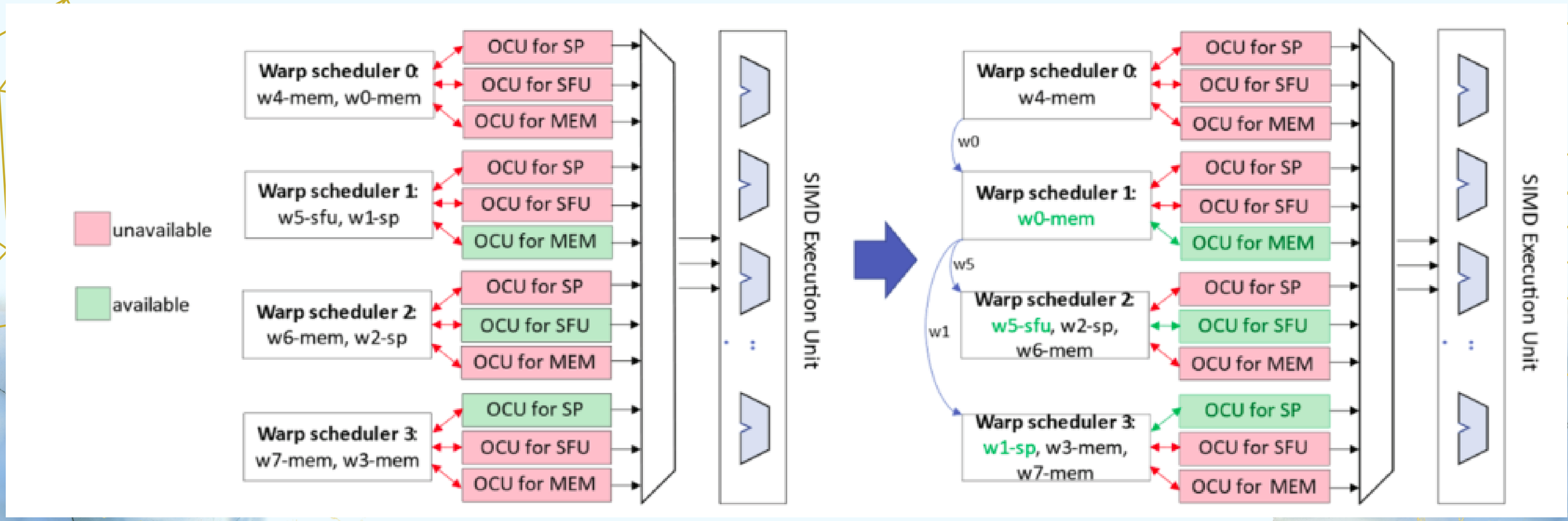
Warp Sharing Mechanism

- Presently, S0 cannot issue w0 and w4 because the OCU for the MEM unit corresponding to S0 is unavailable. An identical scenario occurs with w6 at S2, and w3 and w7 at S3.
- Meanwhile, the OCU for the MEM unit corresponding to S1 is free. However, S1 does not issue warp instruction because the instructions w1 and w5 are supposed to issue the OCU for the SP and SFU units, respectively.

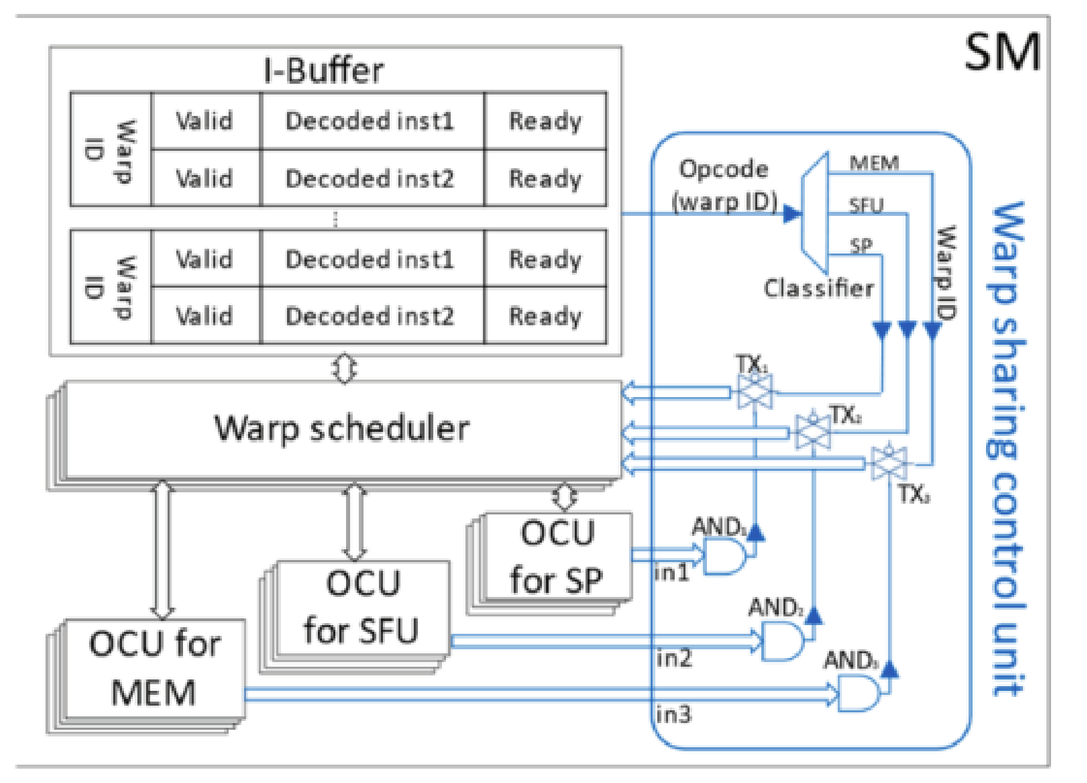
Warp Sharing Mechanism

- We design a warp-sharing control unit that enables the memory instruction of w0 to be issued by S1. Therefore, the OCU for the MEM unit of S1 is utilized.
- By applying the same rule, the warp- sharing control unit permits w5 and w1 to be issued by S2 and S3, respectively.
- Thus, the primary goal of improving resource utilization is achieved.

Warp Sharing Mechanism



Hardware


SM
I-Buffer
Warp ID
Valid
Decoded inst1
Ready
Valid
Decoded inst2
Ready
Warp ID
Valid
Decoded inst1
Ready
Valid
Decoded inst2
Ready
Warp scheduler
OCU for MEM
OCU for SFU
OCU for SP
Opcode (warp ID)
MEM
SFU
SP
Classifier
Warp ID
Warp sharing control unit
TX1
TX2
TX3
AND1
AND2
AND3
in1
in2
in3

Hardware

- The warp scheduler and OCU are stacked in four layers. illustrating that there are four warp schedulers and four corresponding OCUs for each execution unit inside the SM.
- The “warp sharing control unit” scans the opcode from the I-buffer. Each warp instruction type is classified to identify the execution unit that should be employed for individual warp instruction.
- Warps are transferred to suitable warp schedulers via a transmission gate
- The conditions to switch on the TX are combined by an AND gate fed by an “in” input status from all four OCUs.
- TX1, TX2 and TX3 are for SP, SFU and MEM respectively.

Evaluation

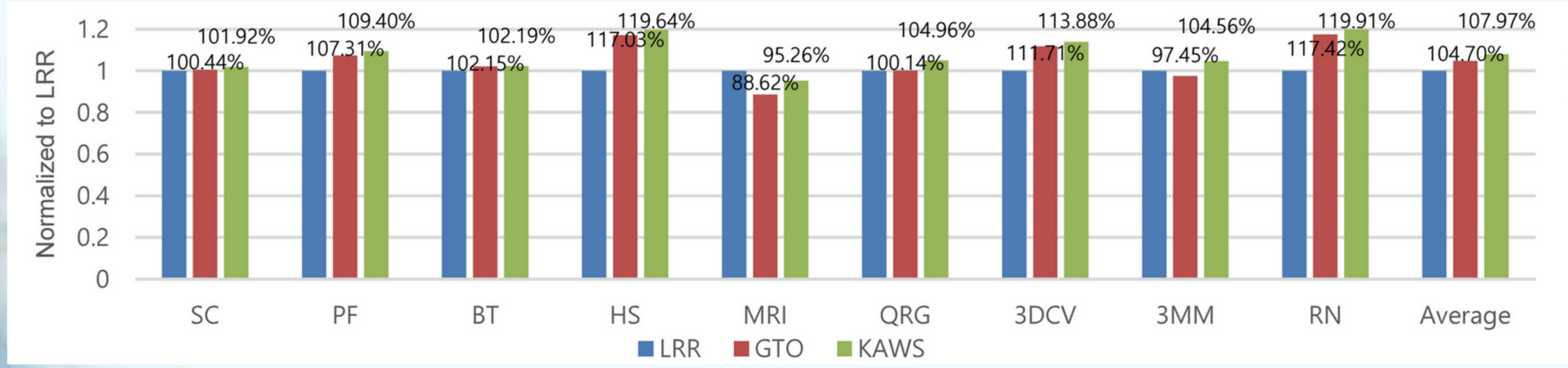
The kernel-aware and warp-sharing concepts were implemented on the GPGPU-Sim using the Titan X GPU (Pascal configuration) because it is highly effective, stable, and highly popular.

Pascal configuration (Titan X):

Parameter	Value
SIMT	28 cores
Clock	1417 (core) : 1417 (interconnection) : 1417 (L2) : 2500 (Dram)
Max. # of CTA per SM	32
L1 data cache	24 kB, 48-way
L1 instruction cache	4 kB, 48-way
L2 cache	3 MB
# of memory controllers	12
# of warps scheduler per SM	4

Performance

The performance of this warp scheduler was evaluated based on nine benchmarks Streamingcluster (SC), Pathfinder (PF), B+tree (BT), Hotspot (HS), Mri-q (MRI), Quasirandomgenerator (QRG), 3Dconvolution (3DCV), three MatrixMultiplication (3MM), and ResNet (RN). All the results have been normalized to the baseline configuration using the LRR policy.



Performance

- On an average, KAWS achieves a performance that is 7.97% and 3.26% higher than those of LRR and GTO, respectively.
- Overall, the performance of the KAWS is higher than that of LRR for eight out of the nine benchmarks. KAWS shows a significant improvement in RN (19.91%), HS (19.64%), and 3DCV (13.88%) because KAWS displays a significantly high performance in terms of latency hiding as opposed to LRR which has low latency hiding capability.

Performance

- In general, KAWS exhibits higher performance than GTO for nearly all the evaluated benchmarks because it employs warp sharing and kernel-aware implementation.
- KAWS can achieve a performance gain of up to 7.5% and 7.3% over GTO in MRI and 3MM,respectively. That is because these benchmarks require continuous computation and memory instructions.
- BT is the only benchmark where KAWS achieves almost zero improvement over GTO. This is because BT is a rich-synchronization application where warp sharing has no impact.

Resource Utilisation

The kernel-aware warp scheduling operates to reduce the execution time of the last issued CTA in the kernel while prolonging that of previously issued CTAs. Hence, the key metric to evaluate the effect of warp scheduling on SM utilization is the execution time of the last issued CTA in the kernel.

Benchmark	Number of kernels	Number of CTAs per kernel	Last CTA execution time		Normalized to GTO (%)
			GTO	KAWS	
SC	140	128	119511.6	117218.33	98.08
PF	5	463	11914.4	11631.6	97.63
BT	2	6000 or 10000	5300	5291	99.83
HS	1	1849	3372	2878	85.35
MRI	4	128	200378.3	186167.67	92.91
QRG	42	128	57937.19	48287.57	83.34
3DCV	254	256	2994.29	2859.68	95.50
3MM	3	1024	199260.7	195657.33	98.19
RN	38	64 or 256	300766.9	310851.34	103.35
Average	-	-	-	-	94.91

Resource Utilisation

We compare our design only with GTO. In LRR, prioritization is intrinsically distributed equally to all the warps, whereby CTAs are completed almost simultaneously.

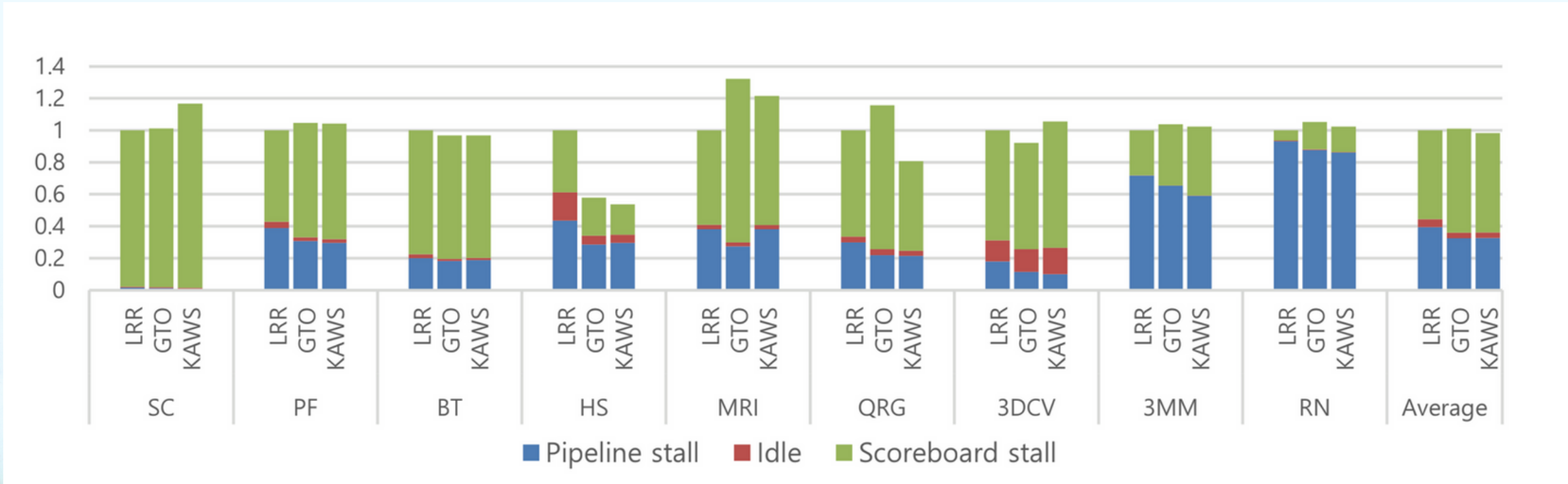
- On an average, KAWS reduces the last CTA execution time by 5.09% compared with that of GTO.
- In particular, there is a significant degradation of execution time in MRI and QRG. It corresponds to performance gains of 7.5% and 4.81%, respectively, over GTO.
- KAWS increases the last CTA execution time by 3.35% in RN.

Stall Cycles

- **Idle**: All available warps are issued to the pipeline, and none are ready to execute the next instruction. Reasons: Warps are waiting at the barrier, empty I-buffer, and control hazard.
- **Scoreboard stall:** All available warps waiting for data from memory. The scoreboard prevents WAW and RAW dependency.
- **Pipeline stall:** All the execution pipelines are full regardless of having valid instructions with available operands. It occurs because of the limited number of existing execution units.

Stall Cycles

- The warp sharing mechanism enables warps that fall into the pipeline stall to be issued in different warp schedulers by available operand collector and execution units Thus, there is pipeline stall reduction.
- Unlike pipeline stall, our concept introduces more scoreboard stalls compared to LRR. This is because it increases the communication traffic among multiple warp schedulers, which can cause conflict in the sharing of OCU.



Conclusion

- The KAWS is a warp scheduling policy, which adapts to the progress of the kernel being executed. By prioritizing more recently issued CTAs, KAWS significantly reduces resource underutilization.
- The Warp Sharing Mechanism addresses resource sharing and allocation among warp schedulers within an SM. By allowing OCUs from different warp schedulers to share resources, this mechanism helps maintain the pipeline and prevent stalls, leading to improved resource utilization.
- Collectively, KAWS and the Warp Sharing Mechanism offer a comprehensive solution to make the most of GPU architecture.

THANK YOU