UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Master in Innovation and Research in Informatics

High Performance Computing

Master thesis

Improving instruction scheduling in GPGPUs

Author: Rodrigo Huerta Gañán

Advisor: Antonio González

Co-Advisor: Jose Maria Arnau Montañes

June 23, 2022

Abstract

GPU architectures have become popular for executing general-purpose programs. Moreover, they are some of the most efficient architectures for machine learning applications which are among the most trendy and demanding applications these days. GPUs rely on having a large number of threads that run concurrently to hide the latency among dependent instructions.

This work presents SOCGPU (Simple Out-of-order Core for GPU), a simple out-of-order execution mechanism that does not require register renaming nor scoreboards. It uses a small Instruction Buffer and a tiny Dependence matrix to keep track of dependencies among instructions and avoid data hazards. Evaluations for an Nvidia GTX1080TI-like GPU show that SOCGPU provides a speed-up up to 3.76 in some machine learning programs and 1.58 on average for a variety of benchmarks, while it reduces energy consumption by 17.6%, with only 3.48% area overhead when using the same number of warps as the baseline. Moreover, we show that SOCGPU can reduce the number of concurrently running warps without hardly affecting performance, which can provide significant reductions in area, especially in the register file and the instruction scheduler logic, as well as other hardware structures of the GPU cores.

Keywords: GPU, GPGPU, out-of-order issue, instruction scheduling.

Acknowledgements

First, I want to thank my advisors. Particularly to Antonio González for his wise advice and for letting me learn and research GPGPU microarchitectures.

Also, I really appreciate the support of my parents during my studies.

Finally, I would like to thank all the mates that I had during the master, especially Aurora.

Contents

1 Introduction **1**

2 Background **5**

3 Baseline **7**

4 SOCGPU **11**
4.1 Alternative Instruction Selection Policies 14
4.2 Smart dual-issue 15

5 Evaluation Methodology **17**

6 Results **21**
6.1 Performance 21
6.2 Area and Energy Consumption 25
6.3 Instruction Buffer Size Sensitivity Analysis 25
6.4 Impact of the Local Instruction Selection Policy 26
6.5 Reducing the Number of Warps per SM 29
6.6 Smart dual-issue evaluation 30

7 Related Work **33**

8 Conclusions and future work **35**

List of Figures

1.1 GP104 SM [1]. 2
1.2 Percentage of cycles with issued instructions in the baseline GPU. 3

3.1 Stages of the baseline GPU architecture, and the most relevant hardware structures for the work. 7
3.2 Example of SIMT execution. 8

4.1 SOCGPU architecture diagram. 12
4.2 Dependence matrix structure. 13

6.1 Speed-up of SOCGPU versus the baseline. 22
6.2 Number of Flushes per 100 warp instructions of each benchmark. 23
6.3 Percentage of Barriers/Branches/Returns instructions in each benchmark. . 24
6.4 Energy savings of SOCGPU with respect to the baseline. 26
6.5 Comparison among different entry sizes for the Instruction Buffer with the First Ready instruction selection policy. 27
6.6 Comparison among different instruction selection policies with an Instruction Buffer of 8 entries. 28
6.7 Speed-up of SOCGPU and the baseline with a different number of warps per SM against the baseline with 64 warps. 29
6.8 Energy savings of SOCGPU and the baseline with a different number of warps per SM against the baseline with 64 warps. 30
6.9 Speed-up comparison of single-issue and dual-issue instructions per cycle. . 31

List of Tables

5.1 GPU specification 18
5.2 Selected benchmarks 19

CHAPTER 1

Introduction

GPU architectures have become popular for executing general-purpose programs [2] in addition to graphics workloads. Moreover, GPGPUs are among the most efficient architectures [3] for machine learning applications, which have become very popular nowadays. These architectures have many cores called Streaming Multiprocessor (SM) that share an L2 cache. Each core is normally subdivided into different sub-cores (usually 4). In Figure 1.1, we can see the design of an SM of the Nvidia Pascal architecture. These sub-cores have an issue scheduler in charge of dispatching instructions into the different SIMD (single instruction multiple data) units local to each sub-core, and a local register file for reading and writing operands. Current GPUs issue instructions in program order from an Instruction Buffer and use a Scoreboard to solve any potential hazards caused by dependencies among instructions.

Out-of-order (OoO) architectures have become very popular in the CPU field. They allow issuing instructions without following the program order while ensuring program correctness. The main CPU designers such as Intel, AMD, and ARM provide this feature in the vast majority of their CPUs. Moreover, products based on the open-source RISC-V ISA have recently appeared with this characteristic [4]. However, there is not any commercial GPU that uses out-of-order issue.

Figure 1.1: GP104 SM [1].

In this work, we show that in traditional GPGPUs, the schedulers cannot find any ready instruction for a large percentage of the cycles, as seen in Figure 1.2. This results not only in a performance loss, but also in additional energy consumption due to augmented execution time, which increases the static energy consumption. To tackle this source of inefficiency, we propose SOCGPU (Simple Out-of-order Core for GPUs), a simple out-of-order issue mechanism that does not require renaming. This mechanism increases the energy consumption of the issue logic but this overhead is more than offset by the overall reduction in energy consumption of the whole GPU. Overall, SOCGPU provides an average increase in performance of 58% and a reduction of energy of 17.6% with a minor cost in extra area of 3.5%. To sum up, in this work we make the following contributions:

- We propose a new design of the Instruction Buffer and the instruction scheduler that requires minor extensions to current GPGPU core microarchitectures.
- We analyze its performance, power and area impact, and demonstrate that this design is both performance and energy efficient for GPGPUs.

The rest of this work is organized as follows. In chapter 2 we present some background and in chapter 3 we describe the baseline GPU architecture. The implementation of the SOCGPU scheme is described in chapter 4. In chapter 5 we describe the evaluation methodology that is later used in chapter 6 to analyze the benefits of the proposed scheme. Chapter 7 discusses related work, and we conclude in chapter 8.

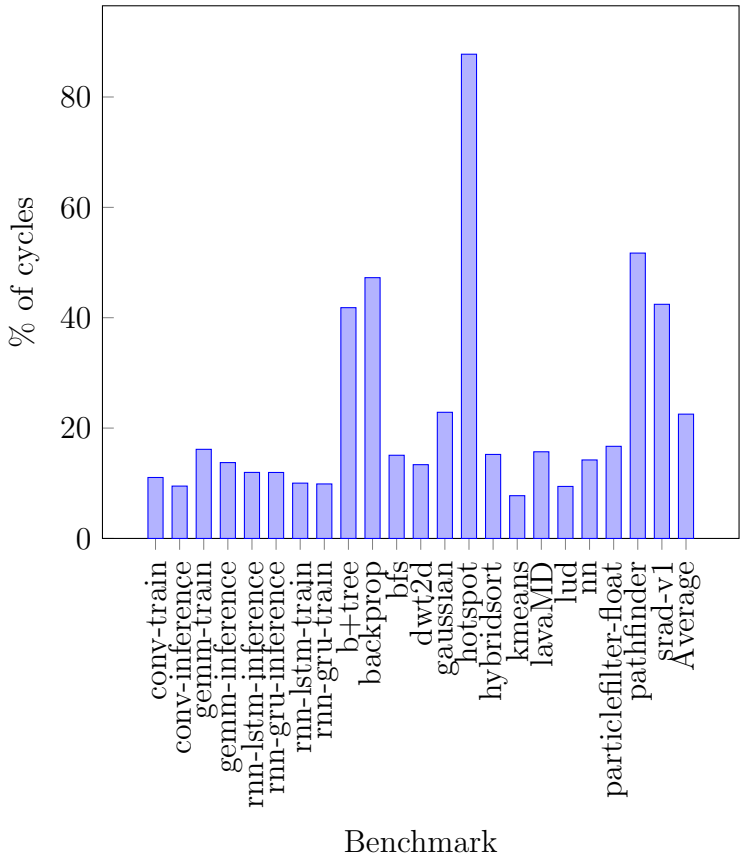

Figure 1.2: Percentage of cycles with issued instructions in the baseline GPU.

CHAPTER 2

Background

OoO execution is prevalent in traditional CPUs. It allows reordering the issue of instructions to minimize pipeline bubbles caused by dependencies and structural hazards while ensuring the correctness of the program. There are many different OoO proposals for CPUs since the first one, Tomasulo [5]. Our work is inspired in early work by Goshima et al. [6] that presents an instruction scheduling scheme that uses matrices to represent the dependencies between instructions. Later, that matrix scheduler was improved by Sassone et al. [7] to provide better scalability or performance.

GPU programming models are based on having a huge amount of threads that are arranged into Cooperative Thread Arrays (CTA). Each CTA is mapped to a core (aka Streaming Multipocessor or SM for short). Threads in a CTA can easily get synchronized and share data through a configurable scratchpad memory inside each SM, normally referred to as Shared Memory.

Once a kernel (a task executed in a GPU) is launched, CTAs are placed into SMs. Threads in a CTA are grouped into sets (typically of 32 or 64 threads each), that are referred to as warps (also known as wavefronts). All threads in a warp execute in parallel in a lockstep mode, which is known as SIMT (single instruction multiple threads) execution mode. Each SM has various sub-cores and the warps of each CTA are distributed among them. A sub-core is a simple compute unit (like a CPU) that issues instructions in order from a set of warps using a particular scheduling policy. An example of a popular issue policy in the literature is Greedy Then Oldest (GTO) [8].

All the threads of a warp advance at the same pace. Warp divergence appears in the case of conditional branches since some threads have to execute the taken path whereas others require to execute the not taken path. This means that both paths need to be serially executed, or executed in an interleaved manner. This warp divergence can be managed through a SIMT stack that stores the different program counters (PCs) and re-convergence PCs of the entries, although other solutions are also possible.

CHAPTER 3

Baseline

As a baseline system for our studies, we model an architecture based on a Nvidia Pascal GP102 GPU. In that design, each SM is subdivided into four sub-cores. In Figure 3.1, we can see the different stages of the architecture in green. The figure also shows some components used in these stages in orange and some backward connections among these stages and components in blue.

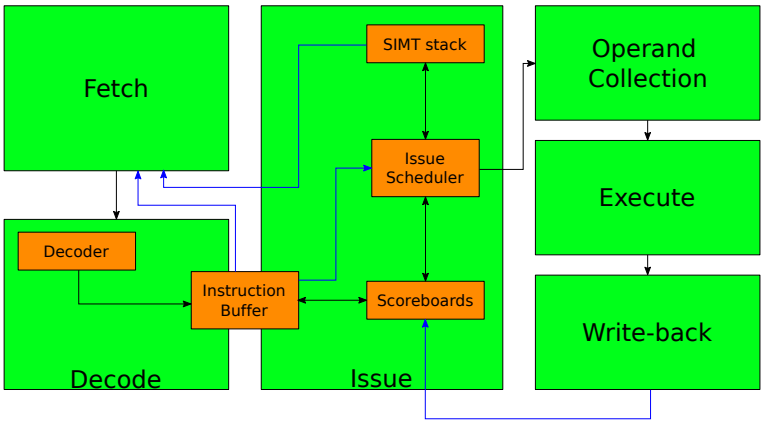

Figure 3.1: Stages of the baseline GPU architecture, and the most relevant hardware structures for the work.

The first stage is the Fetch stage, where a round-robin scheduler selects a warp with an empty Instruction Buffer to start a fetch request of two consecutive instructions from the L1 instruction cache. When the request is ready, both instructions are decoded in the Decode stage and placed into the Instruction Buffer of the corresponding warp. In the issue stage, a warp among all the eligible ones in each of the sub-cores is selected to issue its oldest instruction. A warp is eligible to be scheduled if it has an instruction in the Instruction Buffer and that instruction does not have any dependence on previously

executed instructions pending to finish. This condition is typically checked using a scoreboard. Once an instruction is issued, it is placed in an Operand Collect Unit (OCU), where it waits until all its source register operands are retrieved. Each sub-core register file has multiple banks with one port per bank to allow for multiple accesses in a single cycle. An arbiter deals with the possible conflicts among several petitions to the same bank. When all source operands of an instruction are in the OCU, the instruction is dispatched to the proper execution unit (e.g. memory, single-precision, special function) whose latencies differ depending on the type of unit. When the instruction reaches the write-back (WB) stage, it clears its dependencies with all the following instructions.

Branches cause what is known as warp divergence. This consists in the fact that different threads of the same warp need to follow different control flow. In order to deal with warp divergence, the baseline architecture uses a SIMT stack per warp [9], updated each time the warp issues an instruction. When a new path appears due to a branch instruction, a new entry (each entry has a thread mask, next PC, and re-convergence PC) is pushed to the top of the stack. The next PC of the top of the stack is used to point to the new instruction when a fetch request is performed and to detect control hazards in the issue stage. When the current next PC reaches the re-convergence PC, the entry at the top of the stack is popped and the execution continues with the next PC in the new entry at the top of the stack, which will correspond to an alternative path or the re-convergence point.

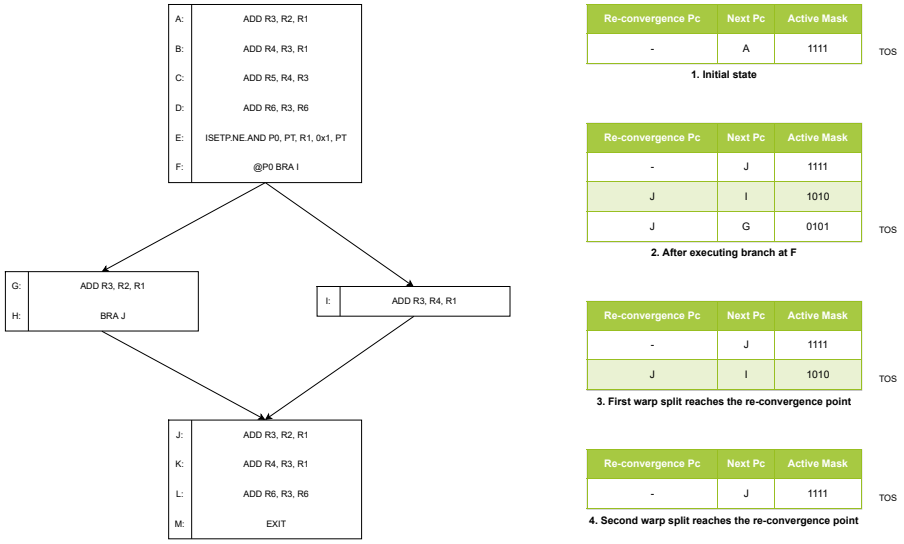

Figure 3.2: Example of SIMT execution.

In order to explain better the behavior of the SIMT stack, we can see an example in Figure 3.2. In that figure, we have written a letter representing the PC next to each instruction. Moreover, for the sake of simplicity, we use a warp size of four threads. In the beginning, there is only one entry in the SIMT-Stack, which contains the PC of the first instruction of the program. It has an Active mask with `1111` that indicate that all threads will execute the instruction. The warp continues executing instructions and updating the next PC until the instruction at PC **F** (the first branch) is executed. Threads belonging to

this warp diverge in two paths. Threads 0 and 2 will perform the taken path, and threads 1 and 3 will do the not-taken. So, the SIMT stack pops the entry and pushes 3 new entries, as seen in the second point of Figure 3.2. The first entry is for the re-convergence point (PC **J**), the second one is for the taken path (PC **I**), and the last one is for the not-taken (PC **G**). Then, the top of the stack (TOS) is updated to point to the last entry (PC **G**). After that, instructions at PC **G** and **H** are executed only by threads 1 and 3 as indicated in the active mask. As all active threads execute the branch instruction at **H**, there is no divergence, and no extra entries in the SIMT stack are needed. Once the next PC matches the re-convergence PC, the entry at the TOS is popped, as seen in the third stage of Figure 3.2. Next, instruction at **I** is executed only by threads 0 and 2. Then, this warp split reaches the re-convergence PC, and the TOS has popped again, reaching the fourth stage in Figure 3.2. Finally, all the threads have re-converged due to the active mask being `1111`.

CHAPTER 4

SOCGPU

SOCGPU is a micro-architectural proposal focused on modifying the issue stage of GPGPUs. Nowadays, GPGPUs issue instructions in program order. SOCGPU is a lightweight technique that allows issuing instructions out-of-order with very small extra hardware requirements. In particular, unlike other alternatives, SOCGPU does not use register renaming in order to simplify the hardware.

SOCGPU identifies ready instructions that can be issued in each warp and chooses one instruction among all of them. An instruction is a candidate to be selected if it fulfills several requirements: not having unresolved dependencies with previous instructions, not having been issued, not being younger than an outstanding barrier, and not being younger than any instruction that can trigger a SIMT stack change.

In Figure 4.1, we can see a block diagram of the proposed architecture for a single warp. In green rectangles, we can see the involved stages. The already available components used in the baseline architecture are depicted in blue, whereas the added components are in orange. Red arrows are used to point to activities related to the SIMT stack. Finally, in purple, we can see backward connections from the write-back stage to components that belong to previous stages.

The proposal consists of 3 different phases. In the first phase, we insert a new instruction from the Decode stage (❶). We check if the new instruction is dependent on previous instructions that are stored in the Instruction Buffer and are valid. Once we know the dependencies of the new instruction, we store the new instruction in any free entry of the Instruction Buffer and the dependencies associated with this instruction are stored in the Dependence matrix (❷).

The main extra component of SOCGPU is the Dependence matrix, which has a row and a column associated to each entry of the Instruction Buffer. The content of each cell of this matrix is a single bit that indicates if the entry associated with the row has a dependence with the entry associated with the column. An instruction can be issued only when all the columns of its associated row are clear, which is determined by a NOR gate of all its entries. This structure replaces the scoreboards used in the baseline architecture.

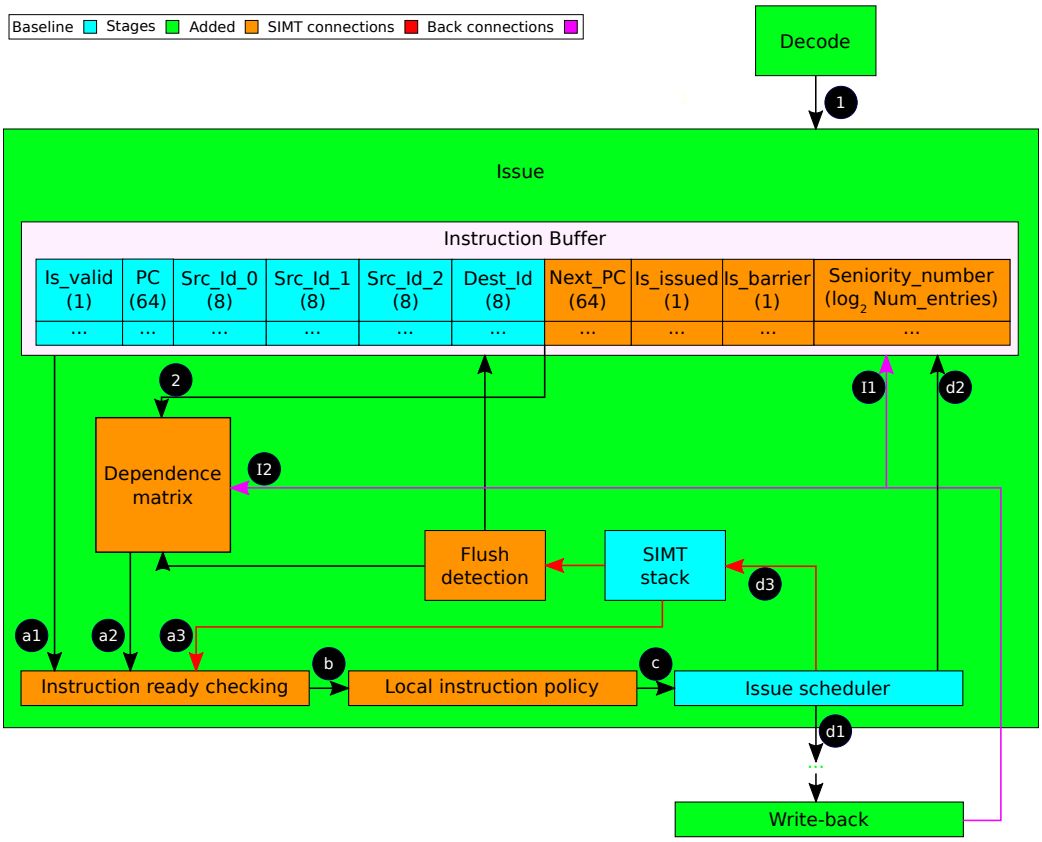

Figure 4.1: SOCGPU architecture diagram.

Decoded instructions are simultaneously placed in the Instruction Buffer and the Dependence matrix physically in the first empty entry available regardless of the program order. In Figure 4.2, we can see an example of a Dependence matrix with size 4. The diagonal is always filled with `0` because an entry cannot have a self-dependence. In this example, the Instruction Buffer and their corresponding rows in the Dependece matrix contains 4 instructions that are not consecutive and are placed with no particular order. Instructions at entries 0 and 3 are candidates for being issued because they do not have any dependence pending to be resolved. However, instruction at entry 1 will not be a candidate to be issued until the instruction at entry 0 finishes its execution (i.e., it reaches the write-back stage).

An instruction A has a dependence on a previous instruction P in the following cases:

- A's destination register is the same as any source or destination operand of P (WAR and WAW dependencies).
- Any of the source operands of A is the same as the destination register of P (RAW dependence).
- If A is a barrier, a branch, or a return from a function call. When A is a barrier, we must guarantee that P has been executed before proceeding with A to guarantee correctness. Branches and returns from functions can produce a change in the

control flow that provokes a flush in the Instruction Buffer. Therefore, we need to ensure that older instructions such as P are issued before this event.

- If P is a barrier, a branch, or a return from a function call. In the case of the barriers, P must be executed before A to maintain the semantics of barriers in the programming model. Branches can cause a modification in the control flow. Consequently, instructions younger than P such as A must not be executed until the branch is solved. Returns from functions are a similar case to branches.

If any of the previous cases is detected, we will store a `1` in the cell corresponding to the row associated to A and the column associated to P to mark the dependence of A with P.

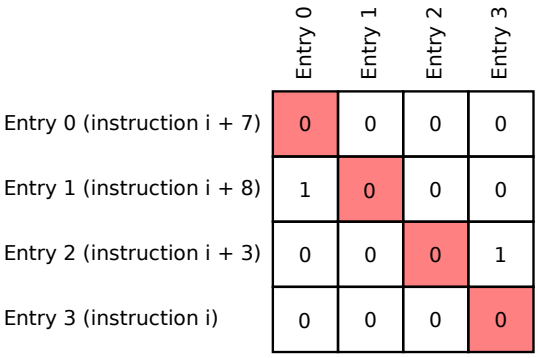

Figure 4.2: Dependence matrix structure.

In the second phase, instructions are issued. First, in the instruction ready checking, we inspect which entries can be a candidate to be issued. An entry is a candidate if it fulfills three requirements:

- It is valid and has not been issued (**a1**).
- It has no dependencies with other instructions. In other words, the whole row in the Dependence matrix associated to that entry has all columns set to `0` (**a2**).
- The next PC of the instruction is not greater or equal to the current SIMT stack's top entry re-convergence PC (**a3**), unless it is the oldest instruction. This requirement is to avoid executing instructions that should not be executed because of a change in the program's control flow. That change in the control flow is caused by reaching a re-convergence point. In the case of being the oldest instruction in the Instruction Buffer, it is allowed to be issued because it is the instruction in charge of triggering the SIMT stack change.

Once we know which entries can be issued (**b**), only one of them is selected as the candidate instruction for each warp. This is done in the Local instruction policy, following a particular heuristic. In section 4.1, we explain the different policies that have been explored. When the candidate is chosen, it is sent to the issue scheduler (**c**). Each cycle, this scheduler chooses only one warp to be issued depending on the issue policy

(e.g., GTO). After issuing an instruction (d1), its corresponding entry in the Instruction buffer is marked as issued (d2).

The third phase is applied when the instructions reach the write-back stage. First, the valid bit in its corresponding entry of the Instruction Buffer is cleared (I1). Second, all the cells of the column of the Dependence matrix associated to that entry are set to `0s` (I2) to clear all dependencies with other instructions.

The Instruction Buffer is a crucial structure in SOCGPU. There is one Instruction Buffer per warp which stores the decoded instructions that later will be checked if they can be issued.

In the baseline, it has two entries, and the fetch stage retrieves two instructions when it is empty and the fetch policy chooses this warp. Only the oldest instruction in the Instruction Buffer is considered as a candidate to be issued. Once a instruction is issued, the entry is freed in the Instruction Buffer.

In SOCGPU the instructions must be maintained in the Instruction Buffer until they reach the write-back stage to correctly check dependencies. Besides, a larger Instruction Buffer allows for more opportunities to discover ready instructions. Therefore, in our studies we have increased its size to eight.

In Figure 4.1, we can see the different fields of each entry of the Instruction Buffer. There are some fields that are the same as in the baseline design: validity, PC, source and destination register identifiers.

In addition to the baseline fields, we need to include some extra ones. The number of required bits of each field are represented into brackets in Figure 4.1. There is a field that indicates if the entry stores a barrier instruction (barriers, branches, or function returns), another to know if the entry has been issued, and a field with its seniority with respect to the other entries. Furthermore, each entry keeps the expected not taken next PC of each instruction, which is used to properly manage the control flow with the SIMT stack. In particular, an instruction can be issued only if its next PC field is not greater than or equal to the re-convergence PC at the top of the SIMT stack, or if it is the oldest entry (which is in charge of triggering the SIMT stack change and Instruction Buffer flushes).

Sometimes, a flush of the Instruction buffer is required. In particular, a flush is triggered when the next PC pointed by the SIMT stack of the warp is different than the next not taken PC of the last issued instruction of that warp. When this happens, it means that there is a change in the program flow due to a modification in the SIMT stack. The SIMT stack can be altered when a branch pushes new entries. Also, when the current entry reaches a re-convergence point and therefore it is popped. Once a flush is triggered, all the entries in the Instruction Buffer that have not been issued are eliminated, and the issued ones will be squashed when they reach the write-back stage.

4.1 Alternative Instruction Selection Policies

In order to decide which instruction is selected from each warp at each cycle as a candidate to be issued, different policies have been investigated:

- **First Ready**: The first entry of the Instruction Buffer (in physical ascending order)

with an instruction ready, valid, and not issued is chosen regardless of its seniority.

- **Older**: The older instruction is chosen among all the valid, ready, and not issued. In this policy, all the seniority numbers of the entries are compared, and the smallest one is selected.
- **Younger**: The youngest instruction among all the valid, ready, and not issued is chosen. In this policy, all the seniority numbers of the entries are compared, and the largest one is selected.
- **Most Ready**: A counter to each entry is added, and each cycle that the instruction in that entry is valid, ready, and not issued, it is increased by 1. The counter is cleared when the instruction is flushed, or a new instruction is allocated. The entry with the largest number is selected among all the entries that are ready, valid, and not issued
- **Least Ready**: It is similar to the previous policy, but the entry with the smallest number on the counter is selected among all the entries that are ready, valid, and not issued.
- **Random**: An instruction is chosen randomly between the ones that are valid, ready, and not issued.

4.2 Smart dual-issue

Some GPU architectures such as Maxwell or Pascal can issue up to two instructions per cycle. However, this is hard to happen because it is needed to fulfill some conditions:

1. The instructions have to belong to the same warp.
2. The Instruction Buffer has to contain two or more instructions.
3. The second instruction cannot have a dependence with the first instruction.
4. The second instruction cannot go to the same execution unit as the first one.

As it can be seen, some of these requirements are very hard to satisfy in the baseline. Nevertheless, the nature of SOCGPU makes that many of the requisites can be solved, but condition **4** is still a problem.

Therefore, we have designed a modified scheme on the top of the policies explained in section 4.1. It consists of a better exploration of the execution units for the second instruction, considering the execution unit used by the first instruction. This way, we take advantage of the out-of-order to select a better candidate for the second instruction.

CHAPTER 5

Evaluation Methodology

To get performance metrics, we have modeled the baseline and the proposed architecture through the Accel-sim [10] simulation infrastructure. Our baseline configuration resembles an Nvidia GTX 1080TI. We use the SASS trace execution mode, but we have used the PTX mode too to verify that OoO executions are correct. Also, we use AccelWattch [11] to get power measurements of both the baseline and our proposal.

We have enhanced the simulator and its tools to make it more accurate for our purposes. First, we have updated the simulator to take into account all types of dependencies. For this purpose, we extended the tracer tool to include the use of predication registers which are omitted in the original simulator and are important to model an out-of-order execution pipeline. Predication instructions now generate a destination register, and a predication register is treated as a source operand when an instruction is predicated. Moreover, we have added an extra scoreboard to the baseline model called the WAR scoreboard to correctly handle WAR dependencies since the original simulator ignored these dependencies, as reported elsewhere[12]. When an instruction is issued, the source operands of this instruction are marked in this scoreboard. Once the instruction reaches the WB stage, it clears the bits of the register source operands in the scoreboard.

In addition, we have extended AccelWattch to report the average power measurements of the whole application instead of doing it separately per kernel.

We can see the main configuration parameters in Table 5.1 for both the baseline and our proposal.

Table 5.1: GPU specification

Parameter	Value
Baseline	
Clock	1480 MHz
Maximum Instructions Issue per Warp	2
SP/SFU/MEM Units per sub-core	4/4/1
Number of SMs	28
Number of Collector Units per SM	8
Warps per SM	64
Number of registers per SM	65536
Warp Width	32
L1/Shared cache size	12 KB/32 KB
Sub-cores per SM	4
Instruction Fetch throughput	4
Schedulers per sub-core	1
Memory Partitions	11
Issue Scheduler policy	GTO
SOCGPU	
Instruction Buffer Size	8
Local policy	First

The benchmarks used to measure the impact of the changes belong to Rodinia 3.1 [13] and Deepbench[14]. The parameters and number of kernels of the applications are the same as used in the default traces provided by the simulator developers for an NVIDIA V100. We have chosen only one application of each type for those applications that have more than 20% of occupancy in a V100 (therefore, Myocyte and NW are not used due to its low occupancy). The input dataset that have been selected for each benchmark is the one that achieves higher IPC in the baseline among the various dataset provided by the simulator developers. In Table 5.2, we can see the complete list of benchmarks.

Table 5.2: Selected benchmarks

Rodinia 3.1	Deepbench
b+tree	conv-train
backprop	conv-inference
bfs	gemm-train
dwt2d	gemm-inference
gaussian	rnn-lstm-train
hotspot	rnn-lstm-inference
hybridsort	rnn-gru-train
kmeans	rnn-gru-inference
lavaMD	
lud	
nn	
particlefilter-float	
pathfinder	
srad-v1	

CHAPTER 6

Results

In this section we analyze the benefits and overheads of SOCGPU with respect to the baseline architecture. The section starts evaluating the performance in section 6.1. Then, the area and energy effect are analyzed in section 6.2. Moreover, we have studied how the size of the Instruction Buffer affects the performance of SOCGPU in section 6.3. Section 6.4 analyzes how important is the Local instruction policy of SOCGPU. Next, in section 6.5, we evaluate the impact of using a reduced number of warps per SM. Finally, in section 6.6, we analyze the impact of the smart dual-issue policy.

6.1 Performance

In Figure 6.1, we can see the speed-up of SOCGPU over the baseline architecture. On the one hand, we can see that there are benchmarks belonging to Deepbench, such as gemm-train or conv-inference, that achieve a huge speed-up, 3.76 and 3.18 respectively. On the other hand, most of the benchmarks of Rodinia 3.1 obtain lower speed-ups compared to Deepbench, but there is not any benchmark that loses performance. On average, the obtained speed-up is 1.58.

The benefits of out-of-order execution are reduced if the program has a lot of branches, barriers, or return calls because these kinds of instructions limit the instruction window. Each time that any of these instructions is found, SOCGPU does not consider any further younger instruction as candidate for issue, until this instruction has been issued. Moreover, if the program has a lot of flushes because it has a lot of branches, it flushes the Instruction Buffer very often. As the Instruction Buffer has to be refilled to take advantage of the entire instruction window, SOCGPU is less effective in these programs.

The main reasons why SOCGPU achieves better speed-up for Deepbench than for Rodinia 3.1 can be explained by the statistics reported in Figure 6.2 and Figure 6.3. The first figure shows the number of flushes per 100 warp instructions. The second figure shows the percentage of instructions limiting out-of-order issue (branches, barriers, and returns

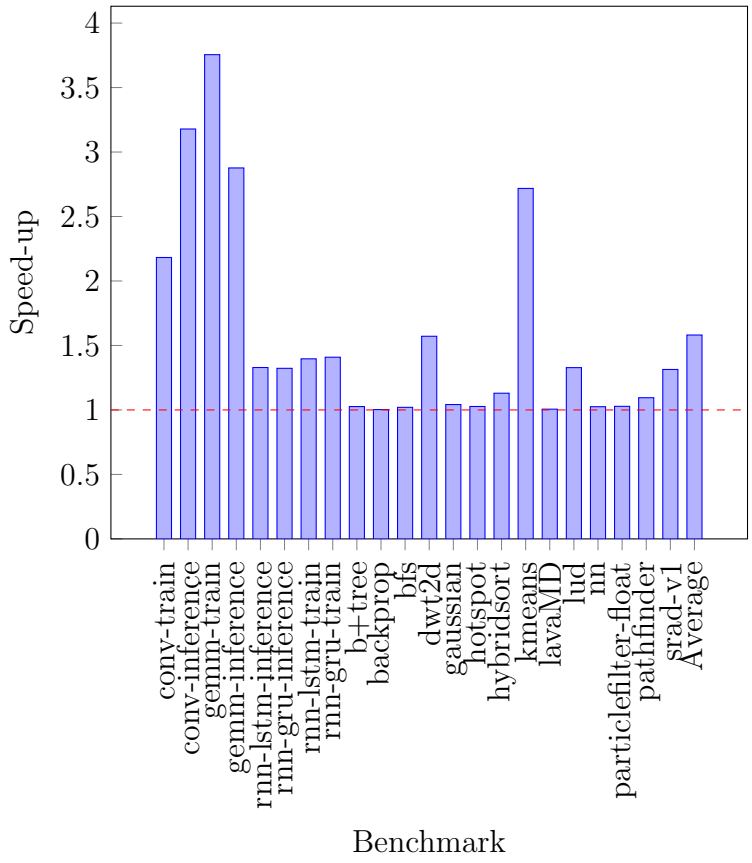

Figure 6.1: Speed-up of SOCGPU versus the baseline.

from functions). We can see in these figures that the number of flushes or instructions limiting out-of-order issue is much lower in Deepbench than in Rodinia 3.1, which explains the difference in the speed-up obtained by SOCGPU. Moreover, as Rodinia 3.1 has more flushes of the Instruction Buffer, it does not take advantage of the increase in the number of entries of the Instruction Buffer as much as Deepbench does, because it gets flushed more frequently, and more flushes provoke the need for more fetches of instructions and a smaller effective instruction window size.

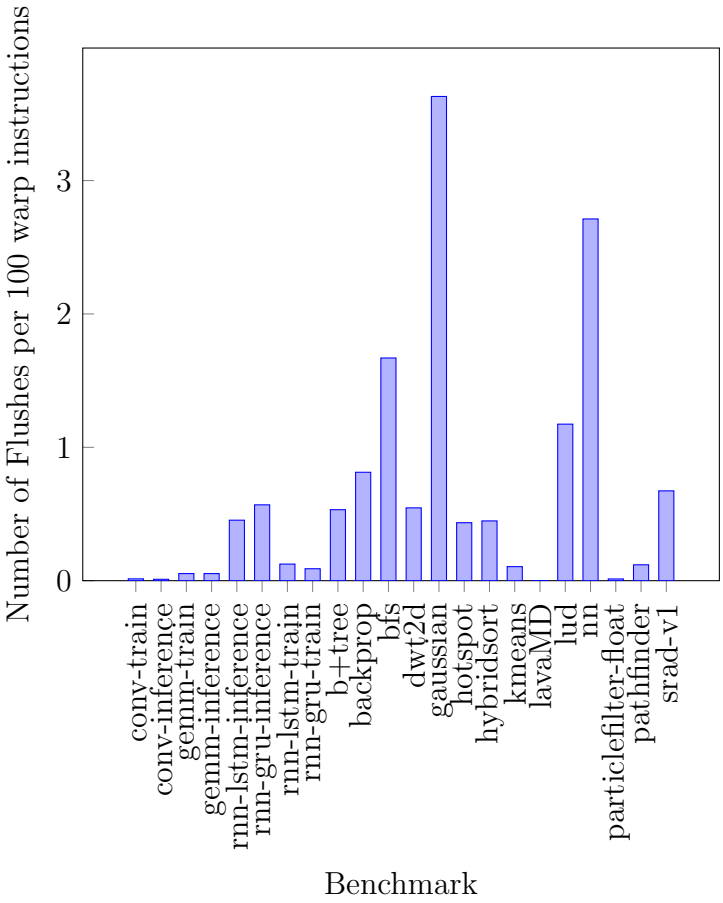

Figure 6.2: Number of Flushes per 100 warp instructions of each benchmark.

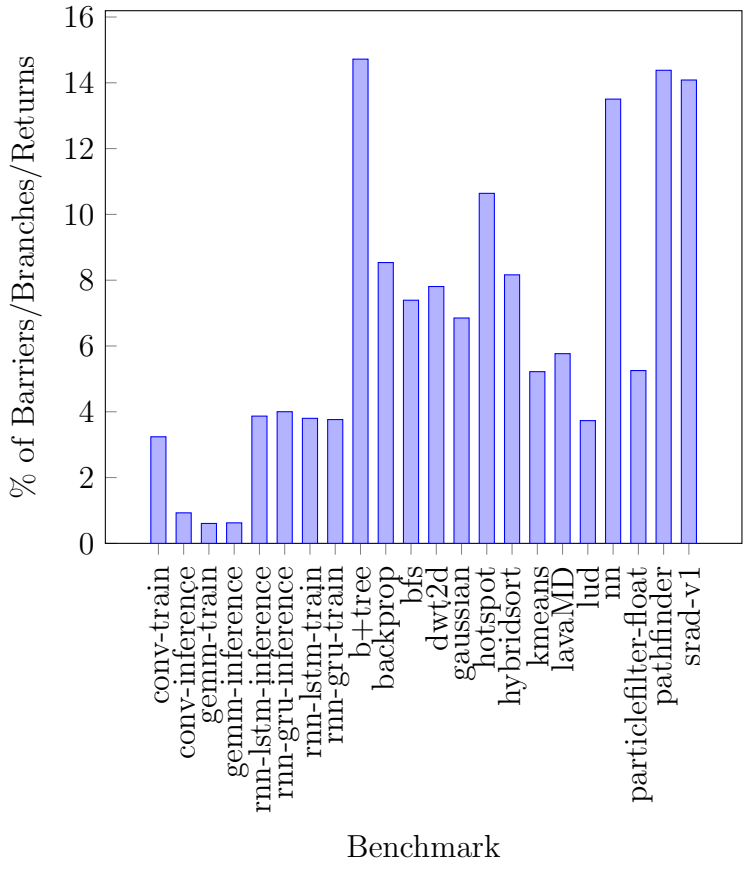

Figure 6.3: Percentage of Barriers/Branches/Returns instructions in each benchmark.

6.2 Area and Energy Consumption

Our baseline architecture resembles an Nvidia GTX 1080TI, which has a die size of 471 mm^2[15].

The power and area overhead has been computed by modeling the baseline and the SOCGPU hardware components that are needed for the issue of instructions in Verilog with the same target clock of the Nvidia GTX 1080TI (1480 MHz, [15]). The overhead of SOCGPU is computed by subtracting the power/area of the baseline from the one obtained for SOCGPU. The design is performed for one warp and is replicated for all the warps in each SM (64 in an Nvidia GTX 1080TI) and all the SMs (28 in an Nvidia GTX 1080TI). The library used to synthesize the design is the SAED14nm of Synopsys,[16] and the software used is Synopsys Design Compiler.

Analyzing the area for one warp, the issue hardware in the baseline occupies 2234.79 um^2 whereas in SOCGPU it occupies 11375.72 um^2. If we scale both for the whole GPU and compute the percentage of area overhead with respect to the whole die size, we obtain that the area overhead of SOCGPU is 3.48%.

Regarding the energy analysis, we obtain each benchmark's power numbers for the baseline using Accelwattch [11]. Then, we add to this figures the power overhead reported by Design Compiler. According to these tools, the issue-related structures of the baseline dissipate 1104 uW, whereas for SOCGPU they dissipate 1553.4 uW. After scaling the consumption for the whole GPU, we get a 0.8 W of power overhead. We have converted those power numbers to energy using the number of cycles from Accel-sim and 1480 MHz as frequency. In Figure 6.4, we can see the percentage of energy savings due to the use of SOCGPU. On average, SOCGPU consumes about 17.6% less energy than the baseline. Looking at the graph in more detail, we can see that the benchmarks for which SOCGPU saves more energy are the ones that have obtained more speed-up (up to 62.49% energy reduction in conv-inference). On the other hand, a few benchmarks that practically are not improving in performance consume more energy (up to 10.75% more energy consumption in gaussian).

6.3 Instruction Buffer Size Sensitivity Analysis

The effectiveness of the out-of-order of SOCGPU depends on the size of the Instruction Buffer. In particular, as we increase its size, we have a bigger instruction window and it is more likely to find ready instructions. Besides, in SOCGPU the instructions must be kept in the structure Instruction Buffer until they arrive at the write-back to correctly keep track of dependencies, which increases the pressure on this structure.

In Figure 6.5, we can see the speed-up of SOCGPU over the baseline when using the First Ready selection policy for different sizes for the Instruction Buffer. We can see that with size 2, SOCGPU loses performance in most of the benchmarks (and we obtain 0.99 of speed-up on average) because the instruction window is too tiny. If we increase the size to 4, we see a significant improvement in speed-up, 1.38 on average. The configuration of 8 and 16 entries achieve, on average, a 1.58 and 1.65 speed-up. It is remarkable that some benchmarks reach up to 4.23 (gemm-train) with 16 entries. However, going beyond

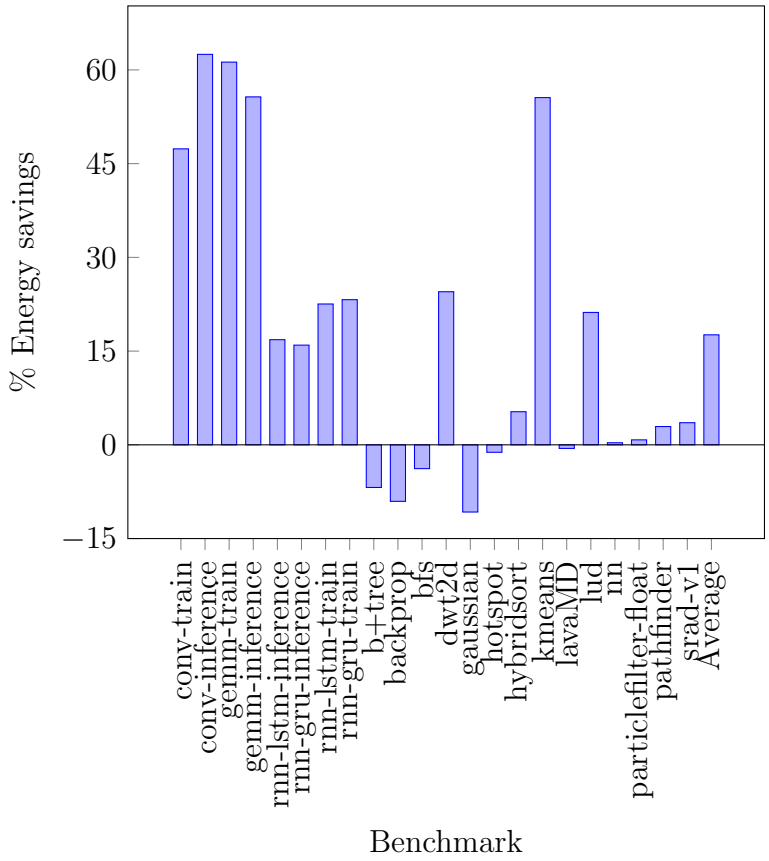

Figure 6.4: Energy savings of SOCGPU with respect to the baseline.

those entry sizes results in small improvements or even some minor performance loss (on average, speed-up is 1.66 for size 32 and 1.64 for size 64).

6.4 Impact of the Local Instruction Selection Policy

In this section, we analyze the performance difference among the different instruction selection policies explained in section 4.1. Figure 6.6 shows the speed-up of SOCGPU over the baseline for the different policies.

The most significant thing about this comparison is that no policy shows a significant difference with respect to the others. For most of the benchmarks, the results are practically the same for all policies. In a few cases such as kmeans, we can see some differences but still they are relatively minor. The simplest policy among all of them is First. It does not need additional comparators to know which is the oldest/youngest instructions among all the ones that are ready (Old and Young) or counters and comparators to know which instruction has been more or fewer times ready (Mostr and Leastr). In conclusion, First is the best choice from the point of view of performance and hardware cost. Random looks also a good choice, since it is also very simple to implement but it has a slight lower performance than First.

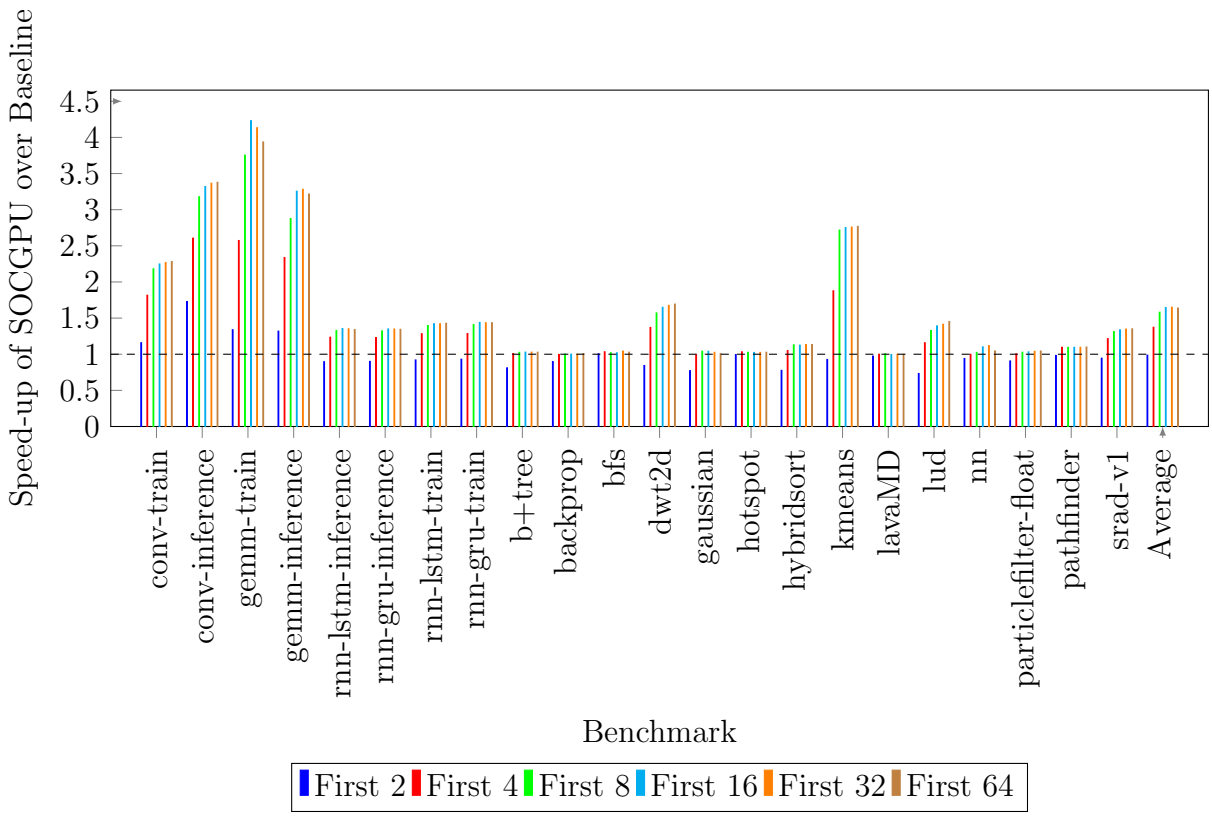

Figure 6.5: Comparison among different entry sizes for the Instruction Buffer with the First Ready instruction selection policy.

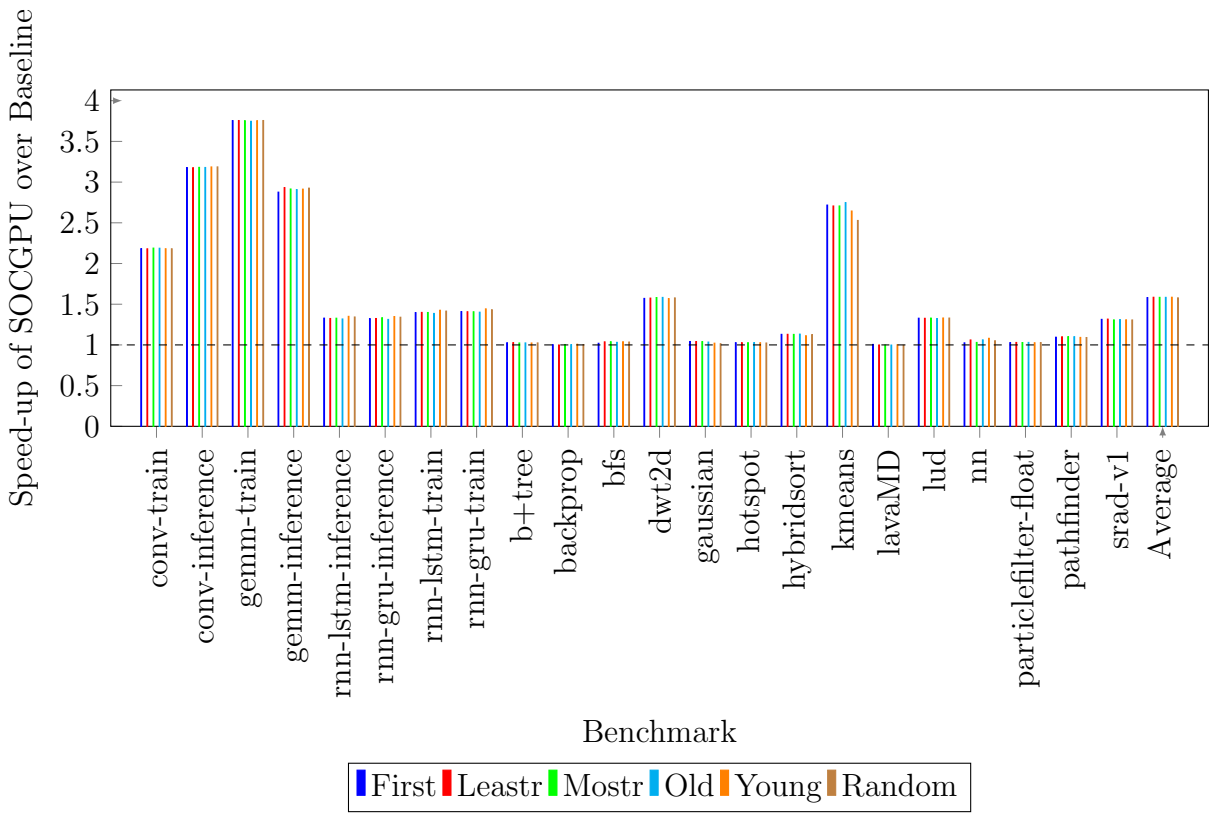

Figure 6.6: Comparison among different instruction selection policies with an Instruction Buffer of 8 entries.

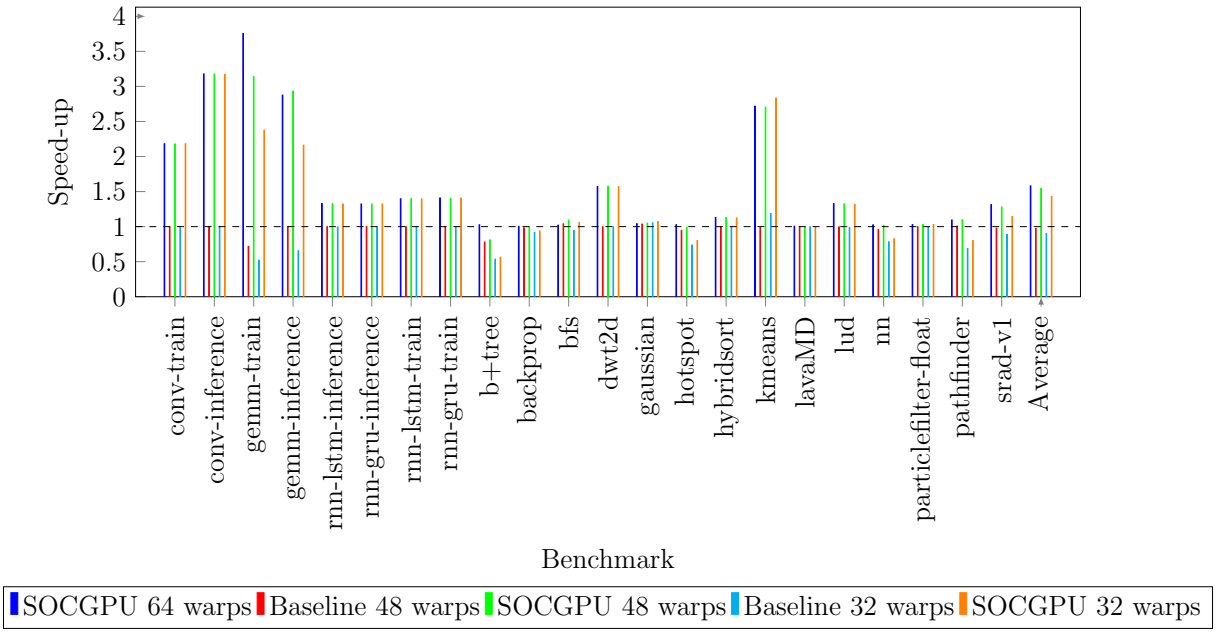

Figure 6.7: Speed-up of SOCGPU and the baseline with a different number of warps per SM against the baseline with 64 warps.

6.5 Reducing the Number of Warps per SM

GPUs rely on a high number of warps per SM to hide the latencies of waiting for dependent instructions, specifically memory instructions whose latency can be very high. However, out-of-order as SOCGPU allows searching instructions that do not have pending dependencies inside a warp, so it can potentially reduce the number of concurrent warps needed to hide these latencies.

In this section, we investigate how the baseline and SOCGPU behave in configurations with fewer warps per SM. The baseline is configured with 64 warps in each SM. The reduced designs use 48 and 32 warps per SM, and the register file has been scaled by the same factor (3/4 and 1/2 respectively). Decreasing the number of warps below 32 is not allowed since CUDA programming model requires to support up to 1024 threads in a block (32 warps).

In Figure 6.7 and Figure 6.8, we can see the impact on performance and energy consumption when the number of warps per SM is reduced compared to the baseline with 64 warps per SM. As it can be seen, the different configurations reduce their performance and increase their energy consumption in a similar ratio. However, SOCGPU still gains a lot of performance compared to the baseline with 64 warps even with a reduced number of warps (1.54 with 48 warps and 1.43 with 32 warps). Regarding energy efficiency, SOCGPU with reduced configurations gets a slight decrease in efficiency compared to the configuration with 64 warps, but it still provides quite significant energy benefits.

Moreover, as we have pointed out before, the area impact of SOCGPU is 3.48% with 64 warps, but it is significantly reduced with 48 warps per SM to 2.61% and 1.74% with 32 warps per SM. In addition, reducing the number of warps in an SM allows us to decrease

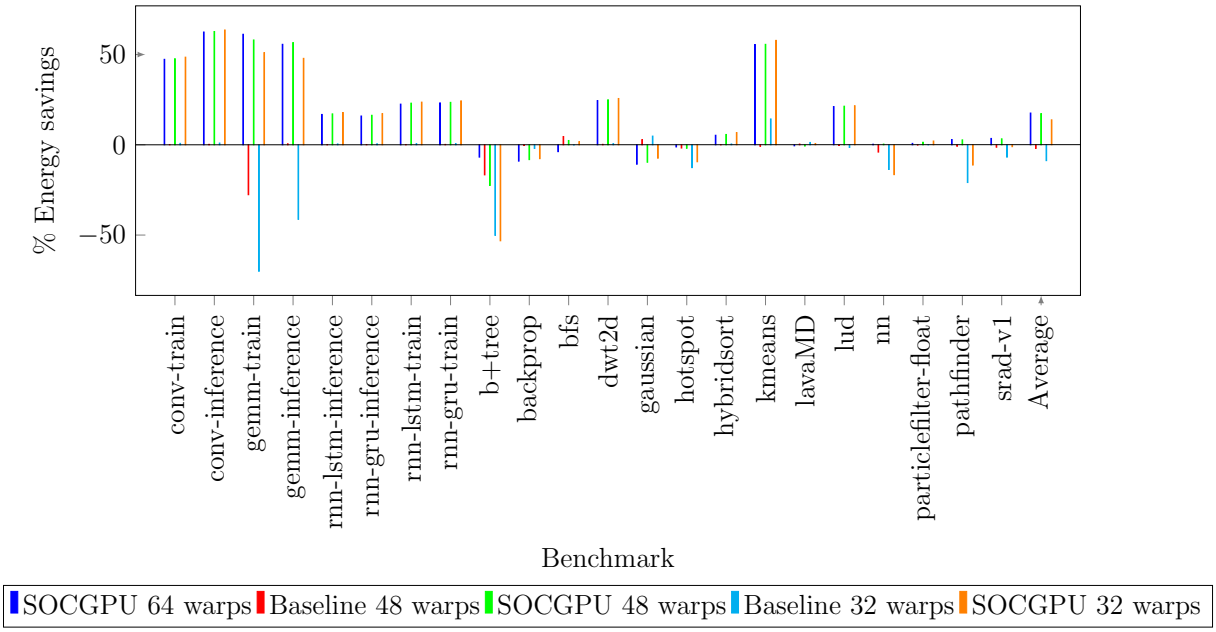

Figure 6.8: Energy savings of SOCGPU and the baseline with a different number of warps per SM against the baseline with 64 warps.

the area used by other structures such as the register file. The percentage of area reduction of the register file is 11.58% with 48 warps and 54.63% with 32 warps according to the McPAT model [17] integrated with AccelWattch. Besides, some other components that store information per warp (e.g., schedulers, SIMT stacks) can be reduced and provide additional benefits in area, although we have not quantified them.

Overall, SOCGPU with 48 warps seems the most competitive configuration since it achieves hardly the same performance and energy benefits than the configuration with 64 warps while providing a significant reduction in area.

6.6 Smart dual-issue evaluation

In this section, we have analyzed how the smart dual-issue improvement affects the proposal.

In Figure 6.9, we can see a comparison of speed-up between the baseline and different configurations. One of the settings is the baseline but can issue only one instruction per cycle instead of two. The other designs belong to SOCGPU. One is doing the dual-issue but without any purpose of improving the choice of the second instruction. There is another one that can only issue one instruction per cycle. The last one is the proposal that enables the smart dual-issue improvement to select better the second instruction of the warp.

As it can be appreciated in the plot comparison, having enabled smart dual-issue does not make any big difference in achieving 1.59 of speed-up against the baseline in front of 1.58 being disabled. However, if we compare the baseline doing a single-issue against the baseline doing a dual-issue, we can see that there is not any significant difference too.

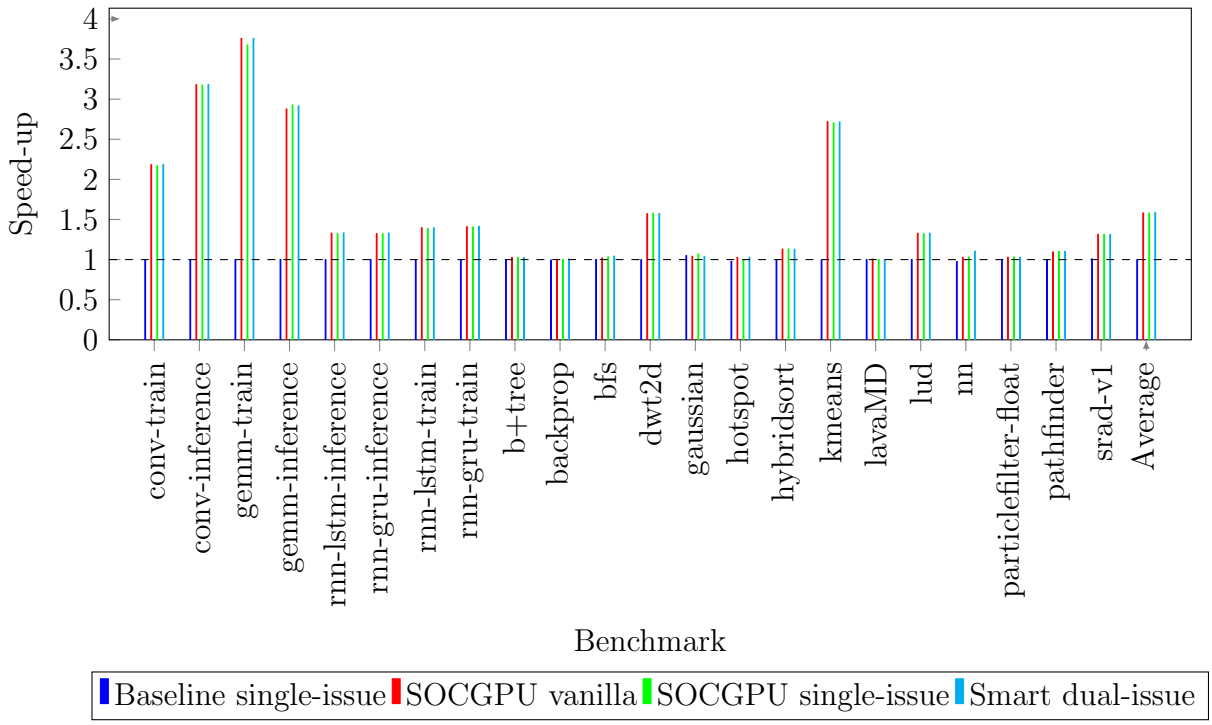

Figure 6.9: Speed-up comparison of single-issue and dual-issue instructions per cycle.

The same happens with SOCGPU and SOCGPU with single-dispatch. Therefore, there seems to be a bottleneck in another part of the baseline architecture that is preventing us from taking advantage of the smart dual-issue feature.

CHAPTER 7

Related Work

There have been a lot of works focused on improving instruction scheduling in GPGPUs. Some of these works have also considered out-of-order approaches in GPGPUs. Warped Preexecution [18] is capable of continuing to issue independent instructions when the first long-latency stall appears. To do it, it has two modes. The N-mode is in-order, and when a long-latency operation appears, it changes to P-mode. In this second mode, it continues to fetch and decode successive instructions that do not have any dependencies. After the long latency operation is completed, the warp switches from P-mode to N-mode. It achieves a speed-up of 1.2, saving a 0.3% of energy. The area overhead is not reported in this work. Another approach is HAWS [19] which modifies the compiler to introduce hints. The compiler adds these hints to point to instructions without dependencies, and the scheduler starts to fetch these instructions when there is a long waiting operation stall. The obtained speed-up of HAWS is 1.15 with an area overhead of 0.4%; there is not any measurement of the energy impact. LOOG [20] uses collector units as reservation stations. The reported results of LOOG are 1.23 of speed-up, 12% of energy savings, and 1.29% of increased area. Something that the three related works have in common is the use of register renaming to solve dependencies, whereas our proposal does not require it. In comparison with these previous works, SOCGPU provides much higher speed-up (1.58) and energy savings (17.6%) with a bit more area overhead (3.48%). However, it is important to point out that each work uses different simulators, GPU models, tools, benchmark suites, and metrics, which can make a big difference in evaluating power, area, and performance.

On the other hand, many works do not include out-of-order execution in their approaches and just focus on improving the issue policies to improve performance. Some of them ([21], [22], [8], [23]) use a issue scheduler with two different levels. The first level has the active warps that are the ones considered to be issued and the second level includes the ones that are stalled. The main idea is to improve the locality of the data. OWL [24] tries to prioritize some group of warps that belong to a given CTA over other CTAs assigned to the same issue scheduler. Other approaches such as CAWS [25] and

CAWA [26] try to identify the critical warps and prioritize those warps over the others. A different idea is used by PCAL [27] that assigns a certain amount of cache usage resource to each warp. If a warp tries to use more than those resources, it is prevented from going beyond that memory operation. Moreover, other works like DYNCTA [28] try to allocate the optimal amount of CTAs per SM depending on the workload.

As SOCGPU only focuses on the scheme to choose candidate instructions to be issued in each warp, it is compatible with any improvement that tries to improve performance by modifying the issue scheduling policy or CTA assignment to SMs.

CHAPTER 8

Conclusions and future work

In this work, we propose SOCGPU, a scheme that can issue instructions out-of-order in a GPU core with a minimal area overhead and without using register renaming. We show that a very small Instruction Buffer and Dependence matrix can provide important benefits in performance and energy consumption. SOCGPU achieves a 1.58 speed-up and 17.6% energy saving on average with only 3.48% area overhead. Moreover, since out-of-order execution contributes to hide the latency of dependent instructions, we show that SOCGPU is hardly affected by a reduction in the number of warps per SM from 64 to 48, which provides significant benefits in terms of area savings in the instruction scheduler, the register file and other components of the GPU cores.

Additionally, we have tried to improve the dual issue scheme with the SOCGPU proposal. However, we have not been able of coming up with any improvement. There seems to be a bottleneck in the baseline because there is not a big difference between dual or single issue simulations.

Finally, even though the proposal achieves good performance, there are opportunities to continue improving the IPC. SOCGPU works better in ML workloads than in other general-purpose applications. Therefore, there is a chance to still enhance these kinds of applications. In addition, the proposal might need some adaptions to work in the most modern architectures that do not use the SIMT stack anymore. This will also need further investigation to see if there are more opportunities to get extra performance from the change of SIMT execution path management.

Bibliography

[1] NVIDIA. Nvidia geforce gtx 1080 gaming perfected, 2016. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[2] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *Proceedings - 2012 IEEE International Symposium on Workload Characterization, IISWC 2012*, pages 141–151, 2012. ISBN 9781457720642. doi: 10.1109/IISWC.2012.6402918.

[3] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. oct 2014. doi: 10.48550/arxiv.1410.0759. URL https://arxiv.org/abs/1410.0759v3.

[4] Christopher A Celio Pi-Feng Chiu Borivoje Nikolic David Patterson Krste Asanović, Christopher Celio, Pi-Feng Chiu, Borivoje Nikoli, David Patterson, and Krste Asanovi. BOOM v2: an open-source out-of-order RISC-V core BOOM v2 an open-source out-of-order RISC-V core. 2017. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html.

[5] R M Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967. ISSN 0018-8646. doi: 10.1147/rd.111.0025.

[6] Masahiro Goshima, Kengo Nishino, Yasuhiko Nakashima, Shin Ichiro Mori, Toshiaki Kitamura, and Shinji Tomita. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 225–236, 2001. doi: 10.1109/micro.2001.991121.

[7] Peter G. Sassone, Jeff Rupley, Edward Brekelbaum, Gabriel H. Loh, and Bryan Black. Matrix scheduler reloaded. In *Proceedings - International Symposium on Computer Architecture*, pages 335–346, 2007. ISBN 1595937064. doi: 10.1145/1250662.1250704.

[8] Timothy G. Rogers, Mike Oconnor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings - 2012 IEEE/ACM 45th International Symposium on Microarchitecture, MICRO 2012*, pages 72–83. IEEE Computer Society, 2012. ISBN 9780769549248. doi: 10.1109/MICRO.2012.16.

[9] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 407–418, 2007. ISBN 0769530478. doi: 10.1109/MICRO.2007.30.

[10] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings - International Symposium on Computer Architecture*, volume 2020-May, pages 473–486. Institute of Electrical and Electronics Engineers Inc., may 2020. ISBN 9781728146614. doi: 10.1109/ISCA45697.2020.00047.

[11] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. AccelWattch: A power modeling framework for modern GPUs. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 738–753, oct 2021. ISSN 10724451. doi: 10.1145/3466752.3480063. URL `https://doi.org/10.1145/3466752.3480063`.

[12] Michael Mishkin. Write-after-Read Hazard Prevention in GPGPUsim. 2016.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, pages 44–54, 2009. ISBN 9781424451562. doi: 10.1109/IISWC.2009.5306797.

[14] S. Narang and G. Diamos. GitHub - baidu-research/DeepBench: Benchmarking Deep Learning operations on different hardware, 2016. URL `https://github.com/baidu-research/DeepBench`.

[15] GeForce 10 Series Graphics Cards, 2022. URL `https://en.wikipedia.org/wiki/GeForce_10_series`.

[16] Vazgen Melikyan, Meruzhan Martirosyan, Anush Melikyan, and Gor Piliposyan. 14nm Educational Design Kit : Capabilities , Deployment and Future. In *Proceedings of the 7th Small Systems Simulation Symposium*, number February, pages 37–41, 2018.

[17] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[18] Sangpil Lee, Won Woo Ro, Keunsoo Kim, Gunjae Koo, Myung Kuk Yoon, and Murali Annavaram. Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In *Proceedings - International Symposium on High-Performance Computer Architecture*, volume 2016-April, pages 163–175. IEEE Computer Society, apr 2016. ISBN 9781467392112. doi: 10.1109/HPCA.2016.7446062.

[19] Xun Gong, Xiang Gong, Leiming Yu, and David Kaeli. HAWS: Accelerating GPU Wavefront Execution through Selective Out-of-order Execution. *ACM Transactions on Architecture and Code Optimization*, 16(2), apr 2019. ISSN 15443973. doi: 10.1145/3291050. URL `https://dl.acm.org/doi/abs/10.1145/3291050`.

[20] Konstantinos Iliakis, Sotirios Xydis, and Dimitrios Soudris. Repurposing GPU Microarchitectures with Light-Weight Out-Of-Order Execution. *IEEE Transactions on Parallel and Distributed Systems*, 33(2):388–402, feb 2022. ISSN 15582183. doi: 10.1109/TPDS.2021.3093231.

[21] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU performance via large warps and two-level warp scheduling. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 308–317, 2011. ISSN 10724451. doi: 10.1145/2155620.2155656.

[22] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. *Proceedings - International Symposium on Computer Architecture*, pages 235–246, 2011. ISSN 10636897. doi: 10.1145/2000064.2000093.

[23] Yang Zhang, Zuocheng Xing, Cang Liu, Chuan Tang, and Qinglin Wang. Locality based warp scheduling in GPGPUs. *Future Generation Computer Systems*, 82:520–527, may 2018. ISSN 0167739X. doi: 10.1016/j.future.2017.02.036.

[24] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 395–406, 2013. doi: 10.1145/2451116.2451158.

[25] Shin Ying Lee and Carole Jean Wu. CAWS: Criticality-aware warp scheduling for GPGPU workloads. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 175–186, 2014. ISSN 1089795X. doi: 10.1145/2628071.2628107.

[26] Shin Ying Lee, Akhil Arunkumar, and Carole Jean Wu. CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. *Proceedings - International Symposium on Computer Architecture*, 13-17-June-2015:515–527, jun 2015. ISSN 10636897. doi: 10.1145/2749469.2750418.

[27] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. Priority-based cache allocation in throughput processors. *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015*, pages 89–100, mar 2015. doi: 10.1109/HPCA.2015.7056024.

[28] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pages 157–166, 2013. ISSN 1089795X. doi: 10.1109/PACT.2013.6618813.