Simple Out of Order Core for GPGPUs

Rodrigo Huerta
rodrigo.huerta.ganan@upc.edu
Polytechnic University of Catalonia
Barcelona, Spain

Jose-Maria Arnau
jose.maria.arnau@semidynamics.com
Semidynamics
Barcelona, Spain

Antonio González
antonio@ac.upc.edu
Polytechnic University of Catalonia
Barcelona, Spain

ABSTRACT

GPU architectures have become popular for executing general-purpose programs which rely on having a large number of threads that run concurrently to hide the latency among dependent instructions. This approach has an important cost/overhead in terms of low data locality due to the increased pressure on the memory hierarchy of the many threads being run concurrently and the extra cost of storing and managing the on-chip state of those many threads.

This paper 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 Tesla V100-like GPU show that SOCGPU provides a speed-up of up to 2.3 in some machine learning programs and 1.38 on average for a variety of benchmarks, while it reduces energy consumption by 6.5%, with only 2.4% area overhead.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**.

KEYWORDS

GPGPU, out-of-order execution, microarchitecture, issue logic

1 INTRODUCTION

GPU architectures have become popular for executing general-purpose programs [1] in addition to graphics workloads. These architectures have many cores, also known as Streaming Multiprocessor (SM) in Nvidia terminology, that share an L2 cache. Each core is normally subdivided into different sub-cores (usually 4). Each sub-core has 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.

GPUs rely on a highly-multithreaded approach in order to hide the latency among dependent instructions. While the oldest instruction of a thread is stalled waiting for its operands to be ready, the scheduler issues instructions from other independent threads. This approach may be effective but at the expense of significant overhead. Since the execution latency of some instructions such as memory instructions is very high, a huge number of threads must be run concurrently to hide these high latencies. In Figure 1, we show the speed-up of an up-scaled version with 128 warps (and double the register file size) against the baseline (64 warps). As can be seen, having more warps does not improve the performance of most applications. Running a huge number of threads concurrently has an important overhead in terms of the extra cost of storing and managing the state of those many threads on-chip. Besides, it increases the pressure on the memory hierarchy, since it has to store the working set of those many threads, which results in a decrease in its effectiveness to exploit locality, and thus in extra cache misses that have an important overhead in terms of energy consumption and performance.

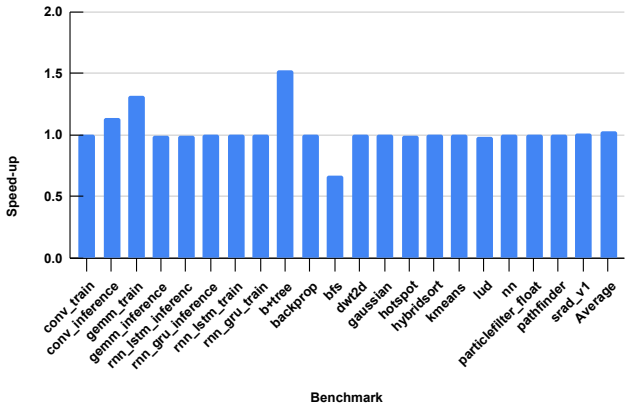

Figure 1: Speed-up of the baseline with 128 warps per SM.

On the other hand, out-of-order (OoO) architectures are very popular in the CPU field. They allow issuing instructions without following the program order while ensuring program correctness, which can potentially hide the latency among dependent instructions but in a different/complementary manner to multithreading. In this case, while some instructions are stalled waiting for their operands, other instructions of the same thread can be issued since the scheduler is not constrained to issue instructions in program order.

In this work, we propose SOCGPU (Simple Out-of-order Core for GPUs), a simple out-of-order issue mechanism that does not require register 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 38% and a reduction of energy of 6.5% with a minor cost in extra area of 2.4%.

The rest of this paper is organized as follows. In section 2 we present some background. The implementation of the SOCGPU scheme is presented in section 3. In section 4 we describe the evaluation methodology that is later used in section 5 to analyze the benefits of the proposed scheme. Section 6 discusses some related work, and we conclude in section 7.

© Authors | ACM 2023. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in: Rodrigo Huerta, Jose-Maria Arnau, and Antonio Gonzalez. 2023. Simple Out of Order Core for GPGPUs. In Proceedings of the 15th Workshop on General Purpose Processing Using GPU (GPGPU '23). Association for Computing Machinery, New York, NY, USA, 21–26. http://dx.doi.org/10.1145/3589236.3589244

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 pioneer designs in the 60s [17] [18]. Our work is inspired by 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. [16] 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 Multiprocessor 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 assigned to 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) [15].

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 [4] that stores the different program counters (PCs) and re-convergence PCs of the entries, although other solutions are also possible [3].

3 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 issues 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, and not being younger than the re-convergence PC of the current pointed entry of the SIMT unit. Moreover, memory instructions cannot be younger than an outstanding barrier and only loads can be reordered between them.

In Figure 2, we can see a block diagram of the proposed architecture for a single warp. In gray rectangles, we can see the involved pipeline stages. The already available components used in the baseline architecture are depicted in blue, whereas the added components are in orange. White arrows are used to point to activities related to the SIMT unit. Finally, in green, we can see backward connections from the write-back stage to components in previous stages.

The scheme consists of three different phases. In the first phase, we insert a new instruction from the Decode stage (①) in the Instruction Buffer and the Dependence matrix. We first check if the new instruction is dependent on previous instructions that are stored in the Instruction Buffer. Once we know its dependencies, 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 (②).

The main extra component of SOCGPU is the Dependence matrix, which has a row and a column associated with 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 depends on 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.

Decoded instructions are simultaneously placed in the Instruction Buffer and the Dependence matrix in the first empty entry available regardless of the program order. In Figure 3, we can see an example of a Dependence matrix of 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 Dependence matrix contain 4 instructions that are not consecutive and are placed in 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 depends 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 memory instruction and P is a barrier in order to maintain the semantics of barriers in the programming model.
- If A is a store and P is a load or a store.
- If A is a load and P is a store.
- If A is a branch, or a return from a function call because they 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 branch, or a return from a function call. 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 similar case to branches.

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

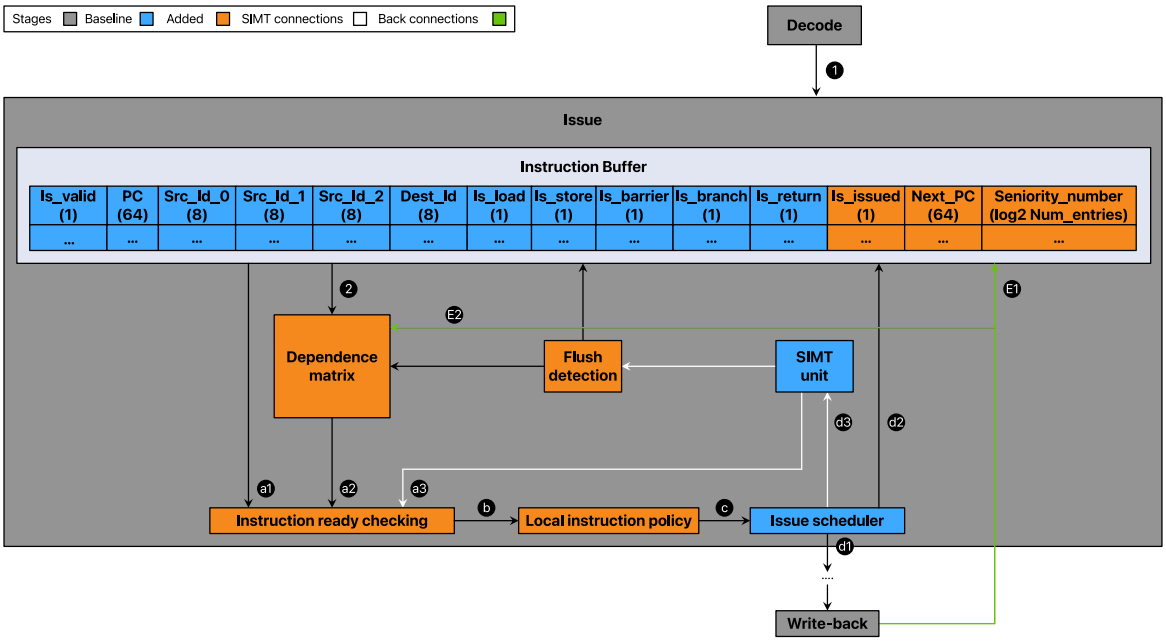

Figure 2: SOCGPU architecture diagram.

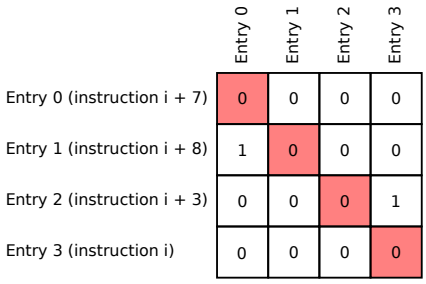

Figure 3: Dependence matrix structure.

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 (ⓐ1).
- It has no dependencies with other instructions. In other words, the whole row in the Dependence matrix associated with that entry has all columns set to `0` (ⓐ2).
- The next PC of the instruction is not greater or equal to the current SIMT unit's entry re-convergence PC (ⓐ3), 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 changes to the SIMT unit.

Once we know which entries can be issued (ⓑ), only one of them is selected as the candidate instruction for each warp. This is done by the *local instruction policy*, following a particular heuristic. Specifically, 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. Once the candidate is chosen, it is sent to the *issue scheduler* (ⓒ). Each cycle, this scheduler chooses only one warp to be issued depending on the issue policy (e.g., GTO). After issuing an instruction (ⓓ1), its corresponding entry in the Instruction Buffer is marked as issued (ⓓ2).

The third phase is applied when an instruction reaches the write-back stage. First, the valid bit in its corresponding entry of the Instruction Buffer is cleared (Ⓔ1). Second, all the cells of the column of the Dependence matrix associated to that entry are set to `0`s (Ⓔ2) to clear all dependencies with other instructions.

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

In SOCGPU the instructions must be kept in the Instruction Buffer until they reach the write-back stage to check dependencies correctly. Besides, a larger Instruction Buffer allows for more opportunities to discover ready instructions. Therefore, in our studies we assume a default size of eight, unless otherwise stated.

In Figure 2, we can see the different fields of each entry of the Instruction Buffer. The number of required bits of each field is represented in brackets. Some fields are the same as in the baseline design: validity, PC, source and destination register identifiers and the type of instruction (load, store, barrier, branch, and function returns). In addition to them, we need extra fields 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 manage the control flow through the SIMT unit properly. 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 elected entry of the SIMT unit unless it is the oldest entry.

Sometimes, a flush of the Instruction buffer is required. In particular, a flush of a warp is triggered when a branch, a return from a function call, or an instruction that reaches re-convergence is issued, and its next not taken PC is different than the PC pointed by the SIMT unit. When this happens, it means that there is a change in the program control flow due to a modification in the SIMT unit. The SIMT unit can be altered when a branch pushes new entries or there is a change of the chosen entry in the SIMT unit. Also, when the issued instruction 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.

Note that GPUs do not need to recover a precise state in case of exceptions. Therefore, a Re-order Buffer (ROB) is not required. Moreover, write-backs do not need to be in order, as it already happens in the baseline. For example, an instruction *i* can write much later than an instruction *i+1* if the latter has lower execution latency or has left the OPC stage earlier than the first one and there is no dependence between these instructions.

4 EVALUATION METHODOLOGY

To get performance metrics, we have modeled the baseline and the SOCGPU architectures through the Accel-sim [9] simulation infrastructure. Our baseline configuration resembles an Nvidia Tesla V100. We use the SASS trace execution mode, but we have also used the PTX mode to verify that OoO execution is correct. Also, we have extended AccelWattch [8] to get average energy measurements of the baseline and our proposal in addition to power.

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 write into 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.

We can see the main configuration parameters in Table 1 for both the baseline and our proposal. The benchmarks used belong to Rodinia 3.1 [2] and Deepbench[13] suites. In Table 2, we can see the complete list of benchmarks.

Table 1: GPU specification

Parameter	Value
Clock	1530 *MHz*
SP/DP/INT/SFU/MEM/Tensor Units per SM	4/4/4/4/1/4
Warps per SM	64
Number of registers per SM	65536
Warp Width	32
Issue Scheduler policy	GTO
Number of SMs	80
Sub-cores per SM	4
Number of Collector Units per sub-core	2
L1/Shared cache size	128 KB
L2 cache size	6 MB
Memory Partitions	32
SOCGPU Instruction Buffer Size	**8**

Table 2: Selected benchmarks

Rodinia 3.1	Deepbench
b+tree	conv_train
backprop	conv_inference
bfs	gemm_train
dwt2d	gemm_inference
gaussian	rnn_lstmtrain
hotspot	rnn_lstm_inference
hybridsort	rnn_gru_train
kmeans	rnn_gru_inference
lud	
nn	
particlefilter_float	
pathfinder	
srad_v1	

5 RESULTS

In this section, we analyze the benefits and overheads of SOCGPU with respect to the baseline. We first evaluate performance, then we evaluate area and energy consumption.

5.1 Performance

Figure 4 shows the speed-up of SOCGPU over the baseline architecture. For some Deepbench benchmarks, such as rnn_gru_train or conv_train, SOCGPU achieves a huge speed-up, 2.3 and 1.6 respectively. For the Rodinia 3.1 benchmarks, speed-ups are still important although lower than for Deepbench, and only for two benchmarks, kmeans and particlefilter_float, SOCGPU shows a minor performance slowdown compared to the baseline. On average, the speed-up of SOCGPU is 1.38.

The benefits of SOCGPU are reduced if the program has many branches, or return calls because these instructions limit the size of the effective instruction window. Each time that any of these instructions is found, SOCGPU does not consider any further younger instruction as a candidate for issue, until this instruction has been issued. We call this kind of dependency Stop. Moreover, the benefits

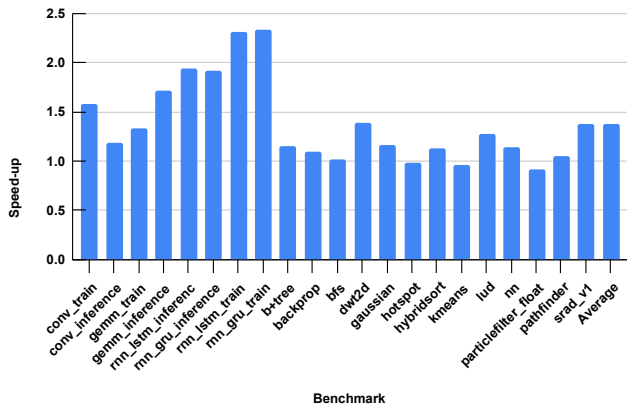

Figure 4: Speed-up of SOCGPU versus the baseline.

of out-of-order execution depend on the amount of RAW, WAW, and WAR dependencies. In the case of the last two (WAW and WAR), they can be mitigated through a renaming mechanism if available. In Figure 5, we can see the average number of dependencies (WAW-WAR, RAW and Stop) among instructions in the Instruction Buffer. This figure depicts for each instruction how many dependent instructions (and the type of dependence) it encounters in the Instruction Buffer when it is inserted. As the percentage of Stop and RAW dependencies in Deepbench is much lower than in most Rodinia benchmarks, which is the main reason why the speed-up of SOCGPU is much higher for Deepbench than for Rodinia. The correlation coefficient between Speed-up and RAW dependencies is −0.73, indicating a high correlation. The fewer RAW dependencies we find in a benchmark, the more performance we obtain.

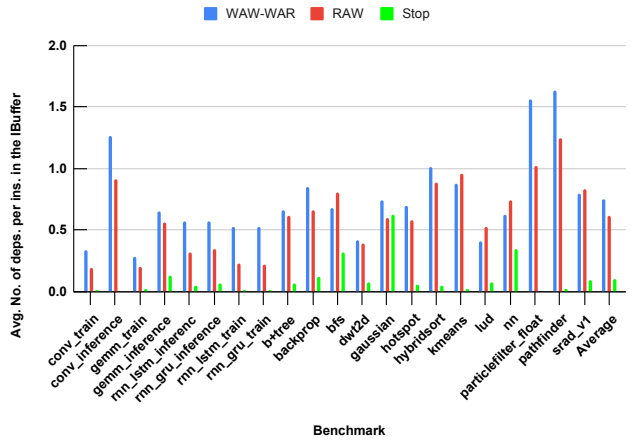

Figure 5: Average number of dependencies per instruction in the Instruction Buffer by type of dependence.

5.2 Area and Energy Consumption

Our baseline architecture resembles an Nvidia Tesla V100, which has a die size of 815 mm^2 [14].

The power and area overhead has been computed by modeling the baseline (scoreboards and Instruction Buffers) and the SOCGPU hardware components that are needed for the issue of instructions in Verilog with the same target clock of the Nvidia Tesla V100 in boost mode (1530 MHz, [14]). 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 Tesla V100) and all the SMs (80 in an Nvidia Tesla V100). The library used to synthesize the design is the SAED14nm of Synopsys,[11] and the software used is Synopsys Design Compiler. Then, the functional behavior was checked with RTL test benches.

Regarding the area, we obtain that the area overhead of SOCGPU is 2.4%, which represents a very small overhead given the huge benefits in performance (1.38x on average).

About the power, we get a total power overhead of 3.26 W. We have integrated the power obtained through design tools into Accelwattch [8]. Moreover, we have modified Accelwattch to obtain the average energy consumption from the original average power consumption. In Figure 6, we can see the percentage of energy savings due to the use of SOCGPU. On average, SOCGPU consumes about 6.5% 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 a higher speed-up (up to 41.9% energy reduction in rnn_gru_train). On the other hand, benchmarks that have a slight slowdown or the performance improvement is low, consume more energy (up to 24.7% more energy consumption in backprop).

In conclusion, SOCGPU not only significantly improves performance, by 1.38x on average, but it also reduces energy consumption, by 6.5% on average. These important benefits come with a very small cost of a 2.4% extra area, due to the simplicity of the proposed design, so we believe that SOCGPU represents a better design alternative for GPGPU cores than the in-order cores that have been traditionally used.

6 RELATED WORK

There have been many works focused on improving instruction scheduling in GPGPUs. A few of these works have also considered out-of-order approaches in GPGPUs. Warped Preexecution [10] is capable of continuing to issue independent instructions when the first long-latency stall appears. It has two execution modes, the N-mode which 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 back from P-mode to N-mode. Another approach is HAWS [5] which modifies the compiler to introduce some hints. The compiler adds these hints to point to instructions without dependencies, and the scheduler starts to fetch these instructions when a long latency operation causes a stall. Both works are known as Dual-Operation Mode (DOM) techniques due to having two execution modes. The latest work in this area is LOOG [7], which uses CUs as reservation

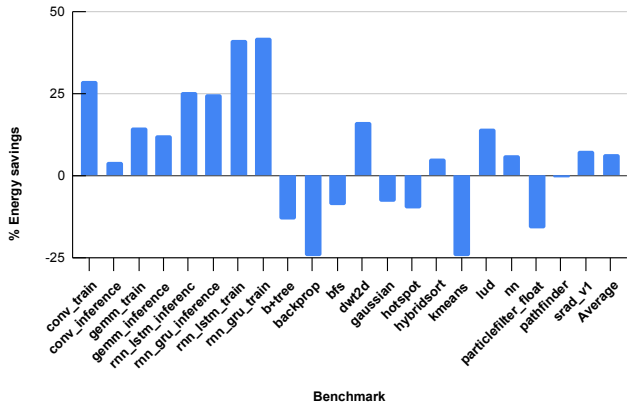

Figure 6: Energy savings of SOCGPU with respect to the baseline.

stations. The above three related works have in common the use of register renaming to solve dependencies, whereas our proposal does not require it.

7 CONCLUSIONS

In this paper, we propose SOCGPU, a scheme that can issue instructions out-of-order in a GPU core with 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 1.38 speed-up and 6.5% energy saving on average with only 2.4% area overhead. Overall, these results show that SOCGPU represents a better design alternative for GPGPU cores than the in-order cores that have traditionally been used.

ACKNOWLEDGMENTS

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, and the ICREA Academia program.

REFERENCES

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

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

[3] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M. Aamodt. 2014. A scalable multi-path microarchitecture for efficient GPU control flow. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 248–259. https://doi.org/10.1109/HPCA.2014.6835936

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

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

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

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

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

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

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

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

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

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

[14] Nvidia. 2017. *NVIDIA Tesla V100 GPU architecture the world's most advanced data center GPU*. Technical Report. Nvidia.

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

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

[17] James E. Thornton. 1980. The CDC 6600 Project. *Annals of the History of Computing* 2, 4 (1980), 338–348. https://doi.org/10.1109/MAHC.1980.10044

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