GPGPU Pipeline Visualization for RISC-V SIMT Architecture

Yu-Yu Hsiao
National Cheng Kung University
Tainan, Taiwan
n28111534@gs.ncku.edu.tw

Liang-Chou Chen
National Cheng Kung University
Tainan, Taiwan
n28111542@gs.ncku.edu.tw

Chung-Ho Chen
National Cheng Kung University
Tainan, Taiwan
chchen@mail.ncku.edu.tw

Abstract

The increasing complexity of modern computer architectures demands more effective analysis and visualization techniques to optimize performance and microarchitecture. One such key approach is pipeline visualization, which offers valuable insights into the architectural design of processors. Although numerous pipeline visualization tools have emerged for CPUs, it is remarkable that research specifically on SIMT pipelines has been overlooked in the existing research. To bridge this gap, this paper proposes a visualization framework specifically for SIMT pipelines. We describe the methodology for generating and visualizing the SIMT pipeline trace and present three case studies including latency hiding, warp scheduling, and memory coalescing to demonstrate the effectiveness of our visualization framework. Our visualized pipeline traces provide insightful observations that align with the quantitative results, demonstrating the framework's capability of analyzing SIMT processors.

Keywords

GPGPU, pipeline visualization, RISC-V, SIMT

ACM Reference Format:
Yu-Yu Hsiao, Liang-Chou Chen, and Chung-Ho Chen. 2024. GPGPU Pipeline Visualization for RISC-V SIMT Architecture. In *Proceedings of Eighth Workshop on Computer Architecture Research with RISC-V (CARRV'24)*, 7 pages.

1 Introduction

As performance-demanding applications emerge, general-purposed GPUs (GPGPUs) have become popular in the high-performance computing domain. GPGPUs rely on the single-instruction, multiple-thread (SIMT) execution model, which leverages data parallelism to deliver flexible computing power while preserving programmability. Meanwhile, RISC-V [2], an open-source instruction set architecture (ISA), has been gaining attraction in both academic and industry realms due to its flexibility and scalability. This versatility has enabled innovative research in computer architecture across a wide range of applications. Building on this momentum, recent studies [8, 21] have proposed solutions for enabling SIMT execution model on RISC-V processors. However, these implementations are often presented at the register transfer level (RTL), providing excessive details that make it difficult to understand the interaction between individual instructions and the processor as a whole. In contrast, C-based simulators [3, 4, 11] are widely used in this scenario, allowing developers to abstract away low-level details and focus on modeling processor behavior, thereby enabling more effective performance analysis and optimization.

To better understand the architectural design of a processor, pipeline visualization is one of the primary techniques adopted by CPU developers. Various tools have also been developed to provide a comprehensive analysis of visualized pipeline trace [22, 5, 17, 7].

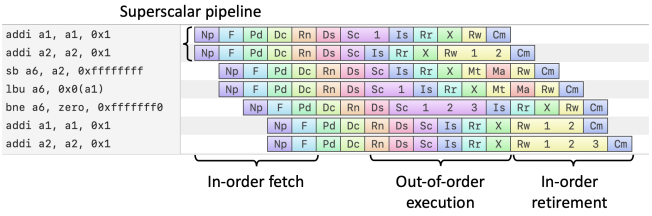


Figure 1: Pipeline trace example of RSD processor executing Dhrystone.

While there exist various simulation frameworks for GPGPU, there has been a notable absence of research that concentrates on the development of visualizations for SIMT pipelines.

Fig. 1 illustrates a pipeline trace example, which is generated by running Dhrystone [23] on a RISC-V out-of-order superscalar processor, RSD [13]. The figure provides visual insight into various details, including:

- **Superscalar pipeline**: Two instructions are executed simultaneously at the same stage.
- **Out-of-order execution**: Instructions are executed out of program order.
- **In-order fetch and retirement**: Despite being executed out-of-order, the instructions are fetched and committed in the program order.

For SIMT processors, we believe that inspecting the pipeline trace can also offer valuable insights into execution details, enabling more effective exploration of architecture and design parameter space. Therefore, we present a visualization framework in this paper as part of our ongoing project to develop a RISC-V-based SIMT processor.

Our in-house cycle-based RISC-V SIMT simulator is capable of running execution-driven simulations and generating detailed pipeline traces of every fetched instruction. This allows the developers to navigate throughout the whole executed program, verifying the correctness of the design and finding potential performance bottlenecks under different workloads. Our visualization framework leverages Konata [17], a pipeline visualization tool, as the GUI front-end for rendering the SIMT pipeline. To the best of our knowledge, this work is the first visualization framework dedicated for SIMT pipelines.

2 Related Work

2.1 RISC-V GPGPU

Simty [8] proposed by Collange demonstrates the feasibility of leveraging RISC-V, a general-purpose ISA, to achieve microarchitecture-level SIMT execution. Elsabbagh *et al.* introduce Vortex GPGPU

[10], which extends the RISC-V ISA with minimal ISA extension. The design includes runtime kernel library support so that it requires no compiler modification to execute the SIMT program. Tine *et al.* extends the Vortex project[21] in 2021, which enhances the capability of 3D graphic rendering for RISC-V processor. The design of Vortex also incorporates high-bandwidth caches to enhance performance. By supporting the OpenCL framework [19] and integrating seamlessly with LLVM-based compilation tools, Vortex possesses a tighter connection with the open-source community, further enhancing its capabilities as a comprehensive GPGPU research platform.

2.2 GPGPU Simulation Frameworks

In recent studies, several simulation frameworks have been proposed specifically for GPGPU simulation. Beckmann *et al.* introduce the AMD gem5 APU simulator and model heterogeneous computing system in gem5 [4]. GPGPU-Sim [3, 16], proposed by Bakhoda *et al.* is a detailed GPGPU simulator that can model how contemporary GPUs execute programs written in CUDA [14] and OpenCL [19] frameworks. Also, GPGPU-Sim is able to model the power consumption [12] of the GPGPU system for a better understanding of power performance. Based on GPGPU-Sim, Khairy *et al.* propose Accel-Sim [11], which extensively updated GPGPU-Sim's performance model to increase its level of detail, configurability, and accuracy. Our previous GPU studies [20] with open-sourced HSAIL ISA [18] are developed in electronic system level (ESL) methodology, which allows developers to explore GPU micro-architecture toward machine-learning-based applications and verify the design with a full-system software stack.

2.3 Pipeline Visualization for CPU

Pipeline visualization has become a well-established technique in the CPU architecture domain, allowing developers to quickly identify performance bottlenecks and troubleshoot complex system setups. Weaver *et al.* present a graphical pipeline viewer [22] that enables users to efficiently spot areas of inefficiency, zooming in or out to uncover high-level trends or inspect specific code sequences to diagnose slow-downs. The state-of-the-art system architecture simulation platform, gem5 [5], features a text-based viewer for its out-of-order CPU model. This viewer enables users to visualize the specific stage of an instruction across every clock cycle. While it is useful for understanding the microarchitecture of the pipeline, the text-based interface's rendering limitation makes it difficult for users to navigate through the entire executed program. Shioya presents Konata [17], a graphical pipeline visualizer capable of parsing gem5's `O3PipeView` format as well as its native log format, `Kanata`. Konata features a user-friendly interface, making it easier to grasp the execution flow. Additionally, the open and well-documented log format enables seamless integration with our own simulator.

3 RISC-V SIMT Architecture and Pipeline Visualization Framework

Most SIMT instruction sets are essentially scalar, which drives our GPGPU research to shift focus from HSAIL-based ISA to the more trendy RISC-V-based ISA. This section introduces our modeled RISC-V-based SIMT architecture, demonstrates the workflow of our pipeline visualization framework, and provides guidance on interpreting the rendered pipeline trace.

3.1 RISC-V-based SIMT Architecture

A challenge for RISC-V ISA in enabling SIMT execution model is that branch divergence cannot be easily resolved by using native RISC-V instructions alone. To address this issue, previous works have proposed solutions such as path tracking with minimal PC [8, 9] and immediate post-dominator (IPDOM) stack-based re-convergence scheme [10, 21]. In this work, we implement the **Inner-Conditional-Statement-First (ICS-First)** re-convergence mechanism in our RISC-V-based SIMT simulator. As proposed in our prior research on enabling SIMT execution on homogeneous multi-core system[6], the ICS-First algorithm relies on the compiler's assistance to identify and prioritize inner conditional statements, allowing the outer statement to wait until there are no more diverging code blocks left for execution.

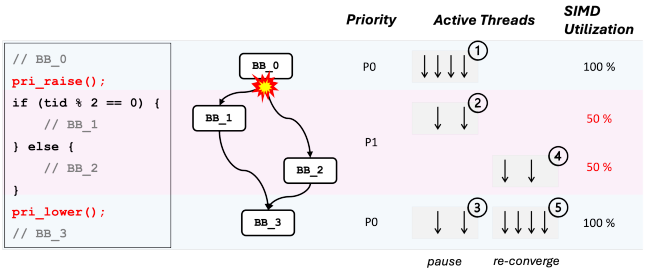


Figure 2: An example of control divergence.

Fig. 2 demonstrates an example of using the ICS-First algorithm to handle control flow divergence. Consider a conditional branch that leads to divergent paths `BB_1` and `BB_2`. ① The processor starts by executing `BB_0` with all threads enabled. ② After encountering the branch instruction, the processor jumps to `BB_1` with only a subset of threads enabled. Note that a priority adjustment instruction pair is inserted around the divergent code block (`BB_1` and `BB_2`) to promote the priority of the diverging threads. ③ In this case, when the execution reaches `BB_3`, i.e., the re-convergence point, the priorities of these threads are lowered and they must pause until their diverging counterparts complete execution. ④ Therefore, the execution resumes to `BB_2` due to the elevated priority. ⑤ Finally, all of the threads reach `BB_3` and re-converge because they now share the same priority. To integrate with the ICS-First re-convergence scheme, we introduce 3 additional instructions to the original RISC-V ISA, as shown in Table 1.

Table 1: Proposed RISC-V SIMT ISA Extension[1].

Instructions	Description
`fsa.pri.raise`	Raise the threads' priority
`fsa.pri.lower`	Lower the threads' priority
`fsa.pri.reset`	Reset the threads' priority

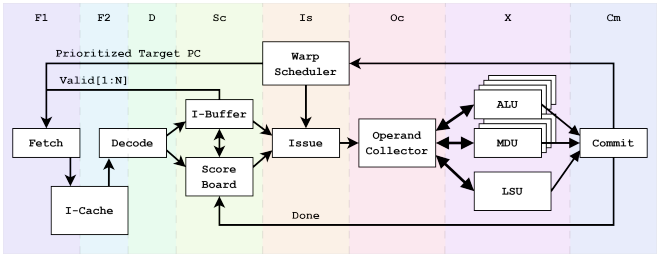


Figure 3: Proposed SIMT pipeline architecture.

Fig. 3 illustrates the architecture of the modeled SIMT core, which is roughly equivalent to NVIDIA's Streaming Multiprocessor (SM) [15] or AMD's Compute Unit (CU) [1]. The SIMT core features a dedicated fetch scheduler in the front-end to fetch instructions from the I-Cache. After an instruction is fetched and decoded, it is sent to the instruction buffer, awaiting scheduling. The scoreboard logic is used to identify data dependencies among instructions within the same warp. The warp scheduler selects a warp instruction according to its scheduling policy and issues it to the operand collector unit, which gathers necessary data from the register file and dispatches the instruction to the back-end function units, including the arithmetic logic unit (ALU), multiplication/division unit (MDU), and load/store unit (LSU).

3.2 Simulation and Trace Generation for the SIMT Pipeline

Fig. 4 illustrates the workflow of our simulation and trace generation for the SIMT pipeline. The process begins by compiling the SPMD program into a RISC-V binary with our patched LLVM compiler that supports the ICS-First algorithm. Next, our cycle-based simulator loads the binary file into memory and initiates the simulation.

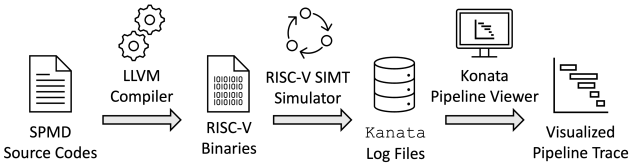


Figure 4: Workflow of proposed visualization framework.

To accurately model the propagation of instructions between pipeline stages and emit the trace, we encapsulate each fetched instruction as an object. As the cycle progresses, every instruction object either **1) remains in its current stage** or **2) updates the architectural state and propagates to the next stage** depending on the status of the pipeline. For instance, a load instruction object can only progress from the execute stage to the commit stage once it has completed memory read operations. Once the instruction enters a new stage, a corresponding log is emitted to inform the pipeline viewer to render the instruction's current stage. The emitted logs are formatted in `Kanata` log format, which records events such as fetch, decode, and execute as they occur throughout the pipeline.

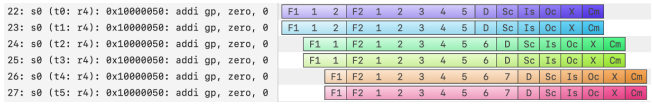
3.3 SIMT Pipeline Visualization

The generated log file is then loaded into Konata, the pipeline viewer, to render the pipeline trace. Upon loading the log file, Konata displays the serial ID, PC, and mnemonic of each warp instruction on the left-hand side of the interface. On the right-hand side of the interface, a detailed pipeline trace for every instruction is rendered. The rendered pipeline trace is essentially a Gantt chart that illustrates the progress of every fetched instruction. Each row of the trace represents a fetched instruction, while the symbol displayed in every colored column indicates the pipeline stage currently occupied by the corresponding instruction at that cycle. The meaning of the symbol is shown in Table 2 which can be mapped to the components of the processor pipeline depicted in Fig. 3.

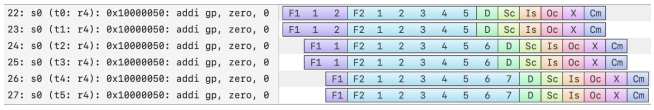
Table 2: Symbols shown in the pipeline trace.

Symbol	Description
`F1`	The fetch request of the instruction is issued
`F2`	I-cache responds to the instruction fetch request
`D`	The instruction is decoded
`Sc`	The instruction is waiting for scheduling
`Is`	The instruction is scheduled and issued
`Oc`	The instruction is collecting operands
`X`	The instruction is being executed by computing units
`Cm`	The instruction commits architectural changes
`123...`	The extra elapsed cycles in the stage

To accommodate various analysis needs, Fig. 5 shows two available color schemes for the generated pipeline trace **a) Coloring by warp ID**: The color of a pipeline trace is determined by each instruction's warp ID. This color scheme is particularly useful to inspect how different warps' instructions are scheduled. **b) Coloring by pipeline stage**: In this scheme, different colors are used for various stages, making it suitable for tracing the execution of individual instructions throughout the pipeline.



(a) Coloring by warp ID



(b) Coloring by pipeline stage

Figure 5: Two color schemes for visualizing the pipeline.

Additionally, the performance characteristics of a simulated SIMT core can be evaluated by examining the slope of the pipeline

[1] "fsa" is currently the codename of this project.

trace. A steeper slope indicates that the SIMT pipeline is issuing and committing instructions more frequently, suggesting efficient execution of parallel threads and a higher IPC (instructions per cycle). Conversely, a gentler slope indicates pipeline congestion due to long-latency instructions. By monitoring the slope, one can gain insights into the local performance characteristics of specific program segments, facilitating targeted optimization and refinement.

4 Case Studies

In this section, we provide three case studies that illustrate the effectiveness of our visualization framework in tracing and analyzing the behavior of our RISC-V SIMT processor during execution. For demonstration, these case studies utilize a mix of real-world examples, such as an inner product of two vectors, alongside some synthetic examples to provide comprehensive explorations of our framework's capabilities.

4.1 Latency Hiding Through Warp Scheduling

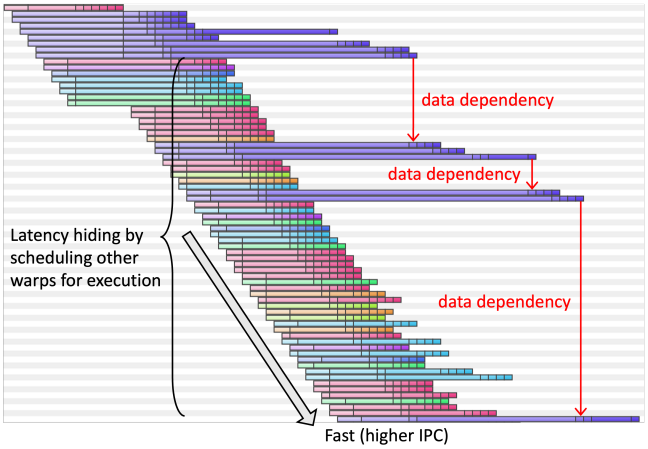
A significant challenge in SIMT architecture lies in the warp scheduler's capability to anticipate and schedule the warp instructions in a manner that effectively minimizes the latency introduced by control or data dependencies. In situations where an instruction relies on the result of a previous long-latency operation, SIMT architecture could effectively mitigate the latency by scheduling another independent warp into the back-end pipeline. This is possible because SIMT processors can accommodate multiple warps simultaneously, thereby enabling them to maintain high throughput while minimizing the impact of long-latency instructions. This approach is also referred to as latency-hiding.

We demonstrate the importance of having a sufficient number of warps available for selection by the scheduler in the example illustrated in Fig. 6. Fig. 6a shows a scenario where sufficient warps are available for scheduling. When the purple-colored warp encounters intensive data dependencies among long-latency instructions, the scheduler can effectively alleviate the associated performance penalty by proactively selecting other warps for execution. However, if we intentionally restrict the scheduler to only two warps for selection, as depicted in Fig. 6b, the exposure of latency becomes inevitable due to the lack of available independent warps that can be scheduled during the execution of long-latency instructions. This ultimately leads to sub-optimal performance.

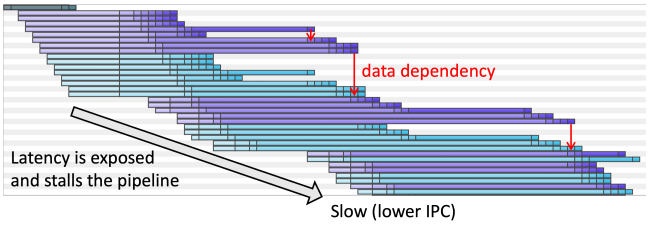
To further explore the impact of warp counts on performance, we conduct simulations of three workloads, relu, 2mm, and vecmul on different core configurations. With the total number of threads held constant, we vary the number of warp per core and threads per warp. The results shown in Fig. 7 indicate that as more warps become available for scheduling, performance tends to improve accordingly, which also corresponds to the observation in the visualized pipeline trace.

4.2 Warp Scheduling Policies

We implement two of the most common scheduling policies in our cycle-based simulator, Loose Round-Robin (LRR) and Greedy-then-Oldest (GTO). This enables a comprehensive evaluation of their effects on the performance of our SIMT architecture. The LRR policy follows a round-robin schedule for warps, ensuring that each warp



(a) More warps available for scheduling



(b) Less warps available for scheduling

Figure 6: The capability of latency hiding through warp scheduling.

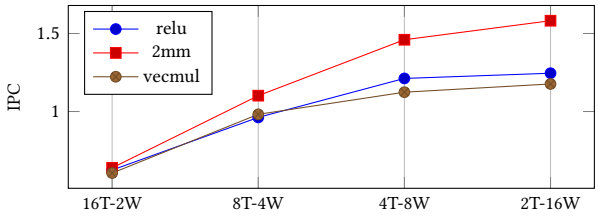
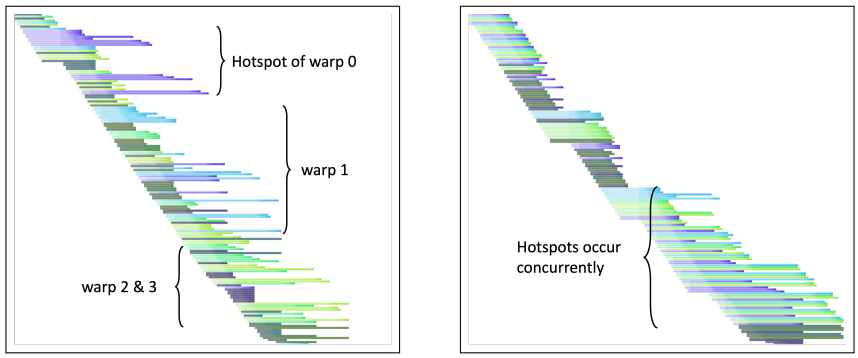


Figure 7: IPC results for different core configurations. IPC is calculated by dividing the total number of committed warp instructions by the number of elapsed cycles.

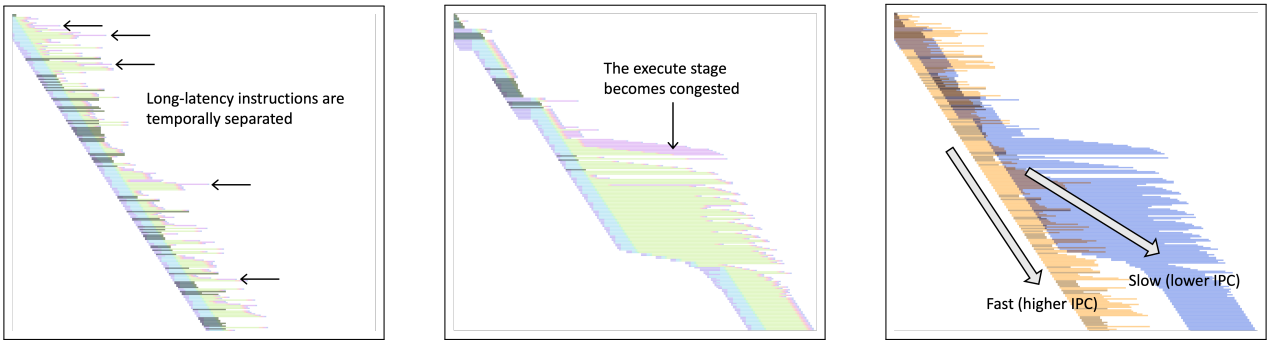
makes similar progress. In contrast, the GTO policy prioritizes the execution of a specific warp until it is blocked by instructions with long latency, such as memory access instructions. Once blocked, the scheduler then selects the least recently used warp to issue and execute.

Fig. 8 facilitates visual analysis of performance hotspots for both scheduling policies, under identical workloads. The workload involves a load-use-store pattern, characterized by significant delays during the load and store operations and the data dependency between the memory access and computations. Fig. 8a shows the simulation result under the GTO scheduler. The pipeline trace exhibits three primary program hotspots, corresponding to warps 0, 1,



(a) Using GTO scheduling policy (b) Using LRR scheduling policy

Figure 8: Comparison of hotspot positions under two warp scheduling policies.



(a) Using GTO scheduling policy (b) Using LRR scheduling policy (c) Combining GTO and LRR pipeline trace

Figure 9: The behavior of two different warp scheduling policies.

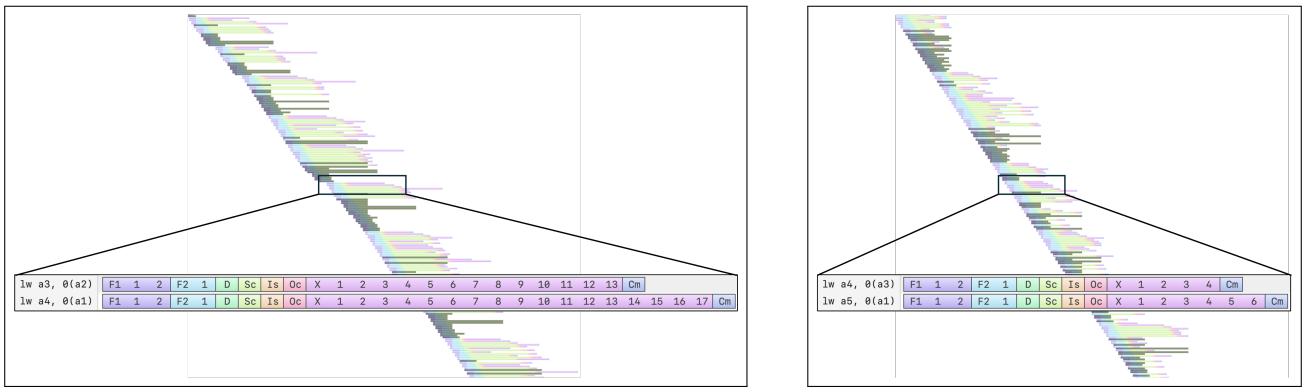
and subsequently, 2 and 3, which are executing the load-use-store pattern. The scattered hotspots are a result of GTO's scheduling strategy, which prioritizes the execution of a single warp until it encounters a blocking latency event, allowing each hotspot to develop independently. On the other hand, LRR's strategy of ensuring similar progress for each warp results in all warps reaching the program hotspot at roughly the same time, as shown in Fig. 8b. Although LRR may offer benefits in the special locality of memory access, the concurrent occurrence of program hotspots may further stall the pipeline.

The limitations of LRR scheduling become more apparent in another workload, where each thread is tasked with loading two data elements and performing multiplication operations on them. Since the multiplier resource is limited within the SIMT pipeline, an excessive number of concurrent requests for multiplication can lead to congestion in the pipeline. As depicted in Fig. 9a, when GTO is employed, the long-latency patterns are dispersed throughout the program's execution. In contrast, Fig. 9b shows that using LRR causes congestion to emerge in the execute stage of the pipeline, leading to decreased performance. We compare the two generated traces in Fig. 9c, and the result indicates a clear performance advantage for GTO over LRR in this particular scenario. This finding highlights that warp scheduling can have a substantial impact on system performance in specific workloads, emphasizing the importance of designing an appropriate scheduling strategy for optimal results.

4.3 Memory Coalescing

Memory coalescing is an optimization technique that allows optimal usage of the memory bandwidth. When parallel threads run the same instruction and access to consecutive locations in the memory, the hardware coalesces all memory accesses into a single consolidated access. In this case study, we demonstrate the fact that the varying task assignments among threads can have a direct impact on the performance of memory accesses. Consider an example where we need to perform inner-product on two vectors, A and B, with size n on an SIMT processor with total k threads. If n is a multiple of k, one straightforward task distribution is to assign each thread the calculation of a partial sum on adjacent data elements as follows:

$$psum_i = \sum_{j=0}^{n/k} A_{i\times(n/k)+j} \times B_{i\times(n/k)+j} \tag{1}$$



(a) Task distribution 1

(b) Task distribution 2

Figure 10: Example of coalescing memory accesses.

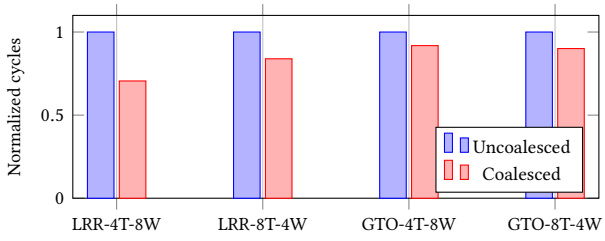


Figure 11: Cycle reduction due to coalesced memory accesses. The cycle counts of coalesced memory access have been normalized to the cycle counts with uncoalesced memory accesses.

where i represents the thread index. An alternative approach would involve distributing the calculation according to the following equation:

$$psum_i = \sum_{j=0}^{n/k} A_{j\times k+i} \times B_{j\times k+i} \tag{2}$$

In this case, each thread would process data elements that are spaced k apart.

We realize both kinds of task distributions and run the simulation on our cycle-based simulator. For this example, we set $n = 3k$. Fig. 10 shows the simulation result. For task distribution 1, as shown in Fig. 10a, each thread i handles data element $3i + j$ in every iteration j, resulting in a memory access pattern of {0, 3, 6, ...} for the first iteration, {1, 4, 7, ...} for the second, and so on. This pattern results in a larger memory footprint and potentially diminishes the opportunity for coalesced memory reads. In contrast, task distribution 2 as illustrated in Fig. 10b shows an improved spatial locality in accessing memory with lower latency. This is because each thread's memory accesses become contiguous within each iteration: {0, 1, 2, ...} for the first iteration, $\{k, k+1, k+2, ...\}$ for the second, and so on. As a result, the contiguous memory accesses are coalesced, leading to optimized performance. In addition, we run simulations with different combinations of warp schedulers and core configurations to see the effects on the coalesced memory accesses. The results shown in Fig. 11 demonstrate that coalesced memory accesses consistently run in fewer cycles than uncoalesced memory accesses, which aligns to our observation in the visualized pipeline trace.

5 Conclusion and Future Work

This paper introduces a pipeline visualization framework that helps analyze and verify the behavior of our RISC-V SIMT processor. Our cycle-based simulator is capable of generating detailed pipeline traces for every individual instruction. Leveraging the Konata [17] pipeline viewer, we can visually observe the behavior of the SIMT pipeline and perform various analyses. Furthermore, we also present three case studies to demonstrate the feasibility of using the visualization framework in highlighting key characteristics of an SIMT pipeline.

As our research group is currently developing a RISC-V-based GPGPU system, we believe that the proposed pipeline visualization framework would be highly beneficial to uncover potential design vulnerabilities and optimize the overall performance. When we progress to the RTL implementation of the processor, the visualized pipeline trace can also serve as a reference model. By comparing the RTL waveform and the trace, we can efficiently validate the design and ensure its correctness. Furthermore, future research endeavors may focus on building a more comprehensive visualization framework for state-of-the-art GPGPU simulation tools like GPGPU-Sim [3] and Accel-Sim [11]. This would further expand the applicability of our pipeline visualization framework across various GPGPU architecture simulation domains.

References

[1] AMD. 2020. Introducing AMD CDNA ARCHITECTURE - The All-New AMD GPU Architecture for the Modern Era of HPC & AI.

[2] Andrew Waterman and Krste Asanovic. 2017. The RISC-V Instruction Set Manual. *User-Level ISA, Document Version 2.2*, I.

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. (Apr. 2009), 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[4] Bradford M Beckmann and Anthony Gutierrez. 2015. The AMD gem5 APU simulator: Modeling heterogeneous systems in gem5. In *Tutorial at the International Symposium on Microarchitecture (MICRO)*.

[5] Nathan Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39, 2, (Aug. 31, 2011), 1–7. DOI: 10.1145/2024716.2024718.

[6] Kuan-Chung Chen and Chung-Ho Chen. 2018. Enabling SIMT Execution Model on Homogeneous Multi-Core System. *ACM Transactions on Architecture and Code Optimization*, 15, 1, (Mar. 22, 2018), 6:1–6:26. DOI: 10.1145/3177960.

[7] Chuan-Hua Chang. 2020. AndesClarity: a Performance & Bottleneck Analyzer for RISC-V Vector Processors. In RISC-V Summit. (Dec. 2020). Retrieved May 7, 2024 from https://www.youtube.com/watch?v=js97LMgQhAk.

[8] Caroline Collange. 2017. Simty: a Synthesizable General-Purpose SIMT Processor. In *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*. Boston, United States, (Oct. 14, 2017).

[9] Caroline Collange. 2011. Stack-Less SIMT Reconvergence at Low Cost. Research Report. ENS Lyon, (Sept. 2011). Retrieved May 9, 2024 from https://hal.science/hal-00622654.

[10] Fares Elsabbagh, Bahar Asgari, Hyesoon Kim, and Sudhakar Yalamanchili. 2019. Vortex RISC-V GPGPU System: Extending the ISA, Synthesizing the Microarchitecture, and Modeling the Software Stack. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*. Phoenix, AZ, USA, (June 22, 2019).

[11] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. (May 2020), 473–486. DOI: 10.1109/ISCA45697.2020.00047.

[12] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News*, 41, 3, (June 23, 2013), 487–498. DOI: 10.1145/2508148.2485964.

[13] Susumu Mashimo et al. 2019. An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. (Dec. 2019), 63–71. DOI: 10.1109/ICFPT47387.2019.00016.

[14] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6, 2, (Mar. 2008), 40–53. DOI: 10.1145/1365490.1365500.

[15] NVIDIA. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi.

[16] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. 2019. Modeling Deep Learning Accelerator Enabled GPUs. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. (Mar. 2019), 79–92. DOI: 10.1109/ISPASS.2019.00016.

[17] Ryota Shioya. 2018. Visualizing the out-of-order CPU model. In *Learning Gem5 Tutorial at the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[18] Ben Sander. 2013. HSAIL: Portable compiler IR for HSA. In *2013 IEEE Hot Chips 25 Symposium (HCS)*. (Aug. 2013), 1–32. DOI: 10.1109/HOTCHIPS.2013.7478287.

[19] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12, 3, (May 2010), 66–73. DOI: 10.1109/MCSE.2010.69.

[20] Yu-Xiang Su, Jhi-Han Jheng, Dun-Jie Chen, and Chung-Ho Chen. 2019. Development of an Open ISA GPGPU for Edge Device Machine Learning Applications. In *2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN)*. (July 2019), 214–217. DOI: 10.1109/ICUFN.2019.8806196.

[21] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Virtual Event Greece, (Oct. 18, 2021), 754–766. ISBN: 978-1-4503-8557-2. DOI: 10.1145/3466752.3480128.

[22] C. Weaver, K.C. Barr, E. Marsman, D. Ernst, and T. Austin. 2001. Performance analysis using pipeline visualization. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. (Nov. 2001), 18–21. DOI: 10.1109/ISPASS.2001.990670.

[23] Reinhold P. Weicker. 1984. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27, 10, (Oct. 1, 1984), 1013–1030. DOI: 10.1145/358274.358283.