Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Lars Murud Aurud

# Improving Fetch and Issue Bandwidth in the Vortex GPU

Masteroppgave i Datateknologi
Veileder: Magnus Jahre
Juni 2023

NTNU
Kunnskap for en bedre verden

Lars Murud Aurud

# Improving Fetch and Issue Bandwidth in the Vortex GPU

Masteroppgave i Datateknologi
Veileder: Magnus Jahre
Juni 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU
Kunnskap for en bedre verden

# Assignment Text

The objective of this master thesis is to work towards leveraging the Vortex soft Graphics Processing Unit (GPU) to create an FPGA-accelerated evaluation infrastructure for GPUs at NTNU's Computer Architecture Lab. The starting point of the thesis is the (potential) performance issues that the candidate identified in Vortex during his project work by enabling Vortex to generate CPI stacks. In the master thesis, the candidate should propose, implement, and evaluate solutions to at least one of the (potential) performance issues. If time permits, the candidate should broaden the analysis by evaluating the observed problems and proposed solutions with benchmarks from commonly used GPU benchmark suites (e.g., Rodinia).

# Abstract

While software simulation is a common method for performing computer architecture research, it is slow for highly parallel architectures such as GPUs. A detailed simulation of a fully sized GPU can take several days. FPGAs are reconfigurable integrated circuits, which can be used to accelerate computer architecture simulations. They are thus a middle ground between slow software simulations and expensive hardware prototypes.

Vortex is a RISC-V based GPGPU capable of being FPGA-accelerated, thus being a good candidate for GPU architecture research. In my project thesis [1], I added support for generating cycles per instruction (CPI) stacks for Vortex, which enabled me to identify potential performance bottlenecks in Vortex' frontend and schedulers. These issues inhibit Vortex from exploiting parallelism and hiding stalls, reducing its throughput.

In this thesis, I implement and evaluate three improvements to the Vortex microarchitecture. First, I implement *ready scheduling*, enabling Vortex to know which warps are ready before issuing them. Secondly, I improve the throughput of Vortex' frontend by allowing it to fetch instructions without stalling. I also implement stall prediction to make Vortex learn when stalls are required. Finally, I implement a greedy then oldest (GTO) scheduling algorithm and compare its performance with the existing loose round-robin (LRR) scheduler.

In my project thesis, the generation of CPI stacks was closely connected to the existing issue scheduler. It only sampled the stall cause of the warp selected by the scheduler. In this thesis, I expand upon this method, sampling the stall cause of all warps in the issue stage. This gives a greater overview of why Vortex is stalling. Additionally, I broaden Vortex' lacking benchmark suite by porting 16 benchmarks from Rodinia, a commonly used set of GPU benchmarks. This involves altering core components of Vortex' system for reading performance data and changing the benchmarks' source code to accommodate for some missing OpenCL functionality.

Implementing *ready scheduling*, removes all missed opportunities for issuing warps. For benchmarks with low-latency stalls, such as *psort*, this change is enough to hide the stalls, reducing CPI by 20%. *Ready scheduling* does however have less impact on benchmarks bounded by long-latency stalls. The frontend improvements are able to increase the frontend bandwidth, reducing the average number of frontend-related stalls by 71%. For *sfilter*, all frontend stalls are removed. This is because the improved frontend only stalls to handle control flow, and *sfil-*

*ter* does not have any control flow instructions. However, the CPI is not reduced to the same degree. On average, the combined improvement of the frontend and *ready scheduling*, reduces CPI by 5.4%. This is because the latencies are too long to be hidden by the current Vortex configuration, which is too small. The bottleneck is thus moved to the backend. The size of the Vortex configuration is also the reason why there is no significant performance difference between using an LRR and GTO scheduler.

# Sammendrag

Softwaresimulering er en mye brukt metode for å forske på datamaskin arkitekturer. Dessverre er det tregt, spesielt for større parallelle arkitekturer, som GPUer. En detaljert simulering av en GPU kan ta opptil flere dager. FPGAer er konfigurerbare integrerte kretser som kan brukes for å akselerere datamaskin arkitektur simuleringer. De er dermed en middelvei mellom trege softwaresimuleringer og kostbare prototyper.

Vortex er en RISC-V basert GPGPU som kan FPGA-akselereres, og er dermed en god kandidat for forskning innen GPU arkitekturer. Gjennom min prosjektoppgave [1], la jeg til støtte for å generere CPI stacks for Vortex. Det gjorde at jeg identifiserte mulige ytelsesproblemer knyttet til Vortex sin frontend og skedulerer. Disse problemene hindrer Vortex i å utnytte parallellitet og skjule venting, som reduserer gjennomstrømningen av instruksjoner.

I denne oppgaven, implementerer jeg og evaluerer tre forbedringer til Vortex sin mikroarkitektur. Først implementerer jeg *klar skedulering*, som gjør det mulig for Vortex å vite hvilke instruksjoner som er klare før de blir utstedt. For det andre, øker jeg gjennomstrømningen av instruksjoner i Vortex sin frontend ved å gjøre det mulig å hente instruksjoner uten å blokkere. I tillegg implementerer jeg stansforutsigelse for at Vortex skal lære når den må blokkere. Til slutt implementerer jeg en *grådig så eldst* (GTO) skeduleringalgoritme, og sammenlikner dens ytelse med den eksisterende *loose round robin* (LRR) skedulereren.

I prosjektoppgaven min var genereringen av CPI stacks koblet til den eksisterende skedulereren. Den samplet bare stansårsaken til instruksjonen valgt av skedulereren. I denne oppgaven utvider jeg denne metoden ved å sample stansårsaken til alle instruksjonene. Dette gir et bedre overblikk over hvorfor Vortex står stille. I tillegg utvider jeg Vortex sin testbenk ved å overføre 16 testprogrammer fra Rodinia, et mye brukt sett med testprogrammer for GPUer. Denne overføring involverer å endre sentrale komponenter av Vortex sitt system for å lese ytelsesdata, og endre testprogrammene sin kildekode for å tilrettelegge for noe manglende OpenCL funksjonalitet.

Implementasjonen av *klar skedulering* fjerner alle sykler hvor instruksjoner er klare, men ingen blir utstedt. For testprogrammer med lav-latens blokkader, som *psort*, er denne endringen nok for å skjule at andre instruksjoner må vente. Dermed fører det til en CPI reduksjon på 20%. *Klar skedulering* har en mye mindre effekt på testprogrammer som er bundet av høy latens. Forbedringene av fron-

tenden øker frontendens båndbredde og reduserer det gjennomsnittlige antallet blokkeringer relatert til frontenden med 71%. For *sfilter*, blir alle blokkeringer relatert til frontenden fjernet. Dette er fordi den forbedrede frontenden bare blokkerer for å håndtere flytkontroll, mens *sfilter* ikke har noen flytkontroll instruksjoner. CPIen blir derimot ikke redusert i samme grad som blokkeringene relatert til frontenden. Ved å kombinere alle endringene, blir den gjennomsnittlige CPIen redusert med 5.4%. Dette er fordi latensen blir for stor til at den kan skjules av den nåværende Vortex konfigurasjonen, som er for liten. Flaskehalsen blir dermed flyttet til backenden av GPUen. Størrelsen på Vortex konfigurasjonen er også grunnen til at det ikke er noen forskjell i ytelse for LRR og GTO skedulererene.

# Contents

**Assignment Text** . . . **iii**
**Abstract** . . . **v**
**Sammendrag** . . . **vii**
**Contents** . . . **ix**
**Figures** . . . **xi**
**Tables** . . . **xiii**
**Code Listings** . . . **xv**
**Acronyms** . . . **xvii**
**Glossary** . . . **xix**
**1 Introduction** . . . **1**
1.1 Motivation . . . 1
1.2 Assignment Interpretation . . . 2
1.3 Contributions . . . 3
1.4 Outline . . . 3
**2 Background** . . . **5**
2.1 GPU Overview . . . 5
2.1.1 GPU Programming Model . . . 5
2.1.2 GPU Architecture . . . 6
2.2 Warp Scheduling . . . 8
2.3 GPU Simulation . . . 9
**3 Changes to Vortex** . . . **11**
3.1 Vortex Architecture . . . 11
3.1.1 Vortex ISA . . . 11
3.1.2 Pipeline . . . 13
3.1.3 Workload Distribution . . . 15
3.2 Scheduling Algorithms . . . 16
3.2.1 Ready Scheduling . . . 16
3.2.2 Greedy then Oldest . . . 17
3.2.3 Matching Warp and Issue Schedulers . . . 18
3.3 Frontend . . . 19
3.3.1 No Stall Scheduling . . . 19
3.3.2 Stall Prediction . . . 20
3.3.3 Back Pressure Reduction . . . 22
**4 Adding Rodinia to Vortex** . . . **23**

4.1 Reading Performance Data . . . 23
4.2 Adapting Benchmarks for Vortex . . . 25
4.2.1 Offline Compilation . . . 26
4.2.2 Memory Allocation . . . 26
4.2.3 Selecting Work Sizes . . . 27
4.3 Fast-Forward, Warm-Up and Early-Exit . . . 27
**5 CPI Stacks for Vortex . . . 29**
5.1 GPU Stall Inspector . . . 29
5.2 CSV Overview . . . 30
5.3 Improving CSV . . . 31
**6 Experimental Setup . . . 33**
6.1 Vortex Configuration . . . 33
6.2 Benchmarks . . . 33
6.3 IDUN Cluster . . . 36
**7 Results and Evaluation . . . 39**
7.1 CPI Stack Overview . . . 39
7.2 Reduction of Control Stalls . . . 40
7.3 Utilization of Functional Units . . . 42
7.4 Memory Stalls . . . 43
7.5 Workload Distribution . . . 46
7.6 Warp Scheduling . . . 48
7.7 Sensitivity Analysis . . . 50
**8 Conclusion and Further Work . . . 53**
8.1 Conclusion . . . 53
8.2 Further Work . . . 54
**Bibliography . . . 55**

# Figures


1.1 High-level overview of the Vortex GPU . . . . . 2
1.2 Thesis outline . . . . . 4

2.1 Relation between the kernel, thread blocks, threads and warps . . . 6
2.2 High-level block diagram of a GPU . . . . . 7
2.3 Thread block scheduling . . . . . 8
2.4 Demonstration of how LRR and GTO selects warps for scheduling . 9
2.5 Demonstration of how LRR and GTO handles long-latency stalls . . 9

3.1 Vortex RISC-V 5-stage pipeline of a streaming multiprocessor . . . . 12
3.2 Clustering of streaming multiprocessors in Vortex. . . . . 12
3.3 Illustration of Vortex' icache-stage. . . . . 14
3.4 Illustration of Vortex' issue stage . . . . . 14
3.5 Demonstration of the unready baseline issue scheduler . . . . . 16
3.6 Illustration of the new issue stage. . . . . 17
3.7 Implementation of greedy then oldest (GTO). . . . . 17
3.8 Demonstration of Vortex' find-first warp scheduler. . . . . 18
3.9 Illustration of the improved icache-stage. . . . . 20
3.10 Illustration of the stall table. . . . . 21
3.11 Illustration of back pressure from the ibuffer to the icache-stage. . . 22

4.1 Timeline of the three stages of multi-kernel benchmarking. . . . . 25
4.2 Timeline explaining fast-forward, warm-up and early-exit. . . . . 28

5.1 Examples of TIP-inspired stall classification with 4 warps in an SM. 31
5.2 Flowchart for CSV's cycle attribution. . . . . 32

6.1 Vortex simulation stack . . . . . 34
6.2 Dcache hit rates over time during startup. . . . . 36

7.1 Normalized CPI stacks before and after the changes. . . . . 40
7.2 Normalized CPI attributed to the frontend . . . . . 41
7.3 Normalized CPI for benchmarks with missed schedule opportunities. 42
7.4 Normalized CPI for benchmarks where memory stalls are revealed . 43
7.5 Average bandwidth usage between the L2 cache and main memory. 43

7.6 Average dcache hitrate . . . . . 44
7.7 Average memory latency on logarithmic scale . . . . . 45
7.8 Distribution of executed instructions in clusters for Hotspot3D . . . 46
7.9 Distribution of executed instructions per SM. . . . . 47
7.10 Normalized CPI stacks for benchmarks with idle cycles . . . . . 48
7.11 Normalized CPI stacks comparing schedulers. . . . . 49
7.12 Normalized CPI stacks when using 8 warps per SM. . . . . 50
7.13 Normalized CPI stacks with double the available memory bandwidth 51
7.14 Normalized CPI stacks when using only L1 caches . . . . . 52

# Tables

4.1 Rodinia benchmarks added to Vortex . . . . . 27

6.1 Configurations for the Vortex architecture. . . . . 34
6.2 Overview of benchmarks and the adjusted input sizes. . . . . 35

# Code Listings

4.1 Example of using the perf macros to create and use the initialization and dump kernels . . . . . 25

# Acronyms

**ALU** arithmetic logic unit. 15, 30

**ASIC** application specific integrated circuit. 10, 54

**BPR** back pressure reduction. 22, 39, 40, 49, 50

**CAL** computer architecture lab. 2, 10

**CPI** cycles per instruction. v–viii, xi, xii, xvii, 1–3, 29, 39–43, 48–53

**CSR** control status register. 13, 15, 23

**CSV** CPI stacks for Vortex. xi, 1–3, 29–32, 53

**FPGA** field-programmable gate array. v, vii, 2, 10, 25, 33, 54

**FPU** floating point unit. 15

**FU** functional unit. 6, 16

**GPGPU** general purpose graphics processing unit. v, 1, 10

**GPR** general purpose registers. 15

**GPU** graphics processing unit. v, vii, viii, xi, 1–3, 5–8, 10, 15, 23–26, 30, 48, 54

**GSI** GPU stall inspector. 29–31

**GTO** greedy then oldest. v–viii, xi, 8, 9, 17–20, 39, 40, 45, 48, 49, 53

**HPC** high-performance computing. 1

**IL** intermediate language. 9

**IPDOM** immediate postdominator. 11

**ISA** instruction set architecture. 10, 11, 26

**LRR** loose round-robin. v–viii, xi, 8, 9, 16–19, 39, 45, 48, 53

**LSB** least-significant bit. 21

**LSU** load-store unit. 15, 30, 44

**MLP** memory level parallelism. 6, 51

**NoC** network on chip. 6, 45, 46, 48, 51

**NSS** no stall scheduling. 19, 20, 22, 39–41

**PC** program counter. 11, 20, 21

**ROP** render output unit. 10

**RTL** register transfer level. 10, 29

**SIMT** single instruction multiple threads. 1, 5, 11

**SM** streaming multiprocessor. xi, xii, 2, 6–8, 11–13, 15, 19, 23, 24, 27, 28, 30, 31, 33, 34, 42, 45–48, 50, 51, 54

**TB** thread block. xi, 5–9, 13, 15, 18–21, 26, 27, 40, 41, 45, 46, 48–51, 54

**TLP** thread level parallelism. 6

**UUID** universally unique identifier. 19

# Glossary

**PoCL** PoCL (Portable Compute Language) is an open source implementation of the OpenCL 1.2 standard with some OpenCL 2.0 features. 26

**RISC-V** RISC-V is an open instruction set architecture based on RISC (Reduced Instruction Set Computer) principles. v, vii, xi, 10–12, 26

**Rodinia** Rodinia is a benchmark suite designed for heterogeneous computing infrastructures with OpenMP, OpenCL and CUDA implementations. v, vii, 3, 26

**Vortex** Vortex is an open source RISC-V based GPGPU architecture aiming at enabling architecture research and FPGA-accelerated simulation. v–viii, xi, xiii, xvii, 1–3, 5, 10–15, 18, 19, 23–30, 33, 37, 39, 40, 42–45, 48, 50, 51, 53, 54

# Chapter 1

# Introduction

## 1.1 Motivation

Computer architects and engineers have for a long time been able to increase computer performance by increasing clock frequencies and the reduction of feature sizes following Moore's Law [2]. However, with the end of Dennard scaling [3], we have hit a power wall [4], forcing designers to look at other possibilities for increasing performance.

Accelerators are specialized hardware for a specific domain or application, and they are a key component of improving computing power in high-performance computing (HPC). Graphics processing units (GPUs) are by far the most common type of accelerator [5], as most desktop and laptop computers have dedicated GPUs. GPUs utilize SIMT and exploit data level parallelism to achieve high throughputs for large parallel workloads. In the beginning, they were mainly used for graphics applications, but later generations of GPUs consists of a set of highly parallel and programmable cores for more general purpose computation [6]. Due to the prevalence of GPUs, and the increased possibilities for general purpose computation, GPU research is becoming more essential.

Vortex [7] is an open-source GPGPU with a focus on enabling architecture research. Vortex comes with its own simulation environment, giving cycle-accurate software simulations. The main attraction of using Vortex in GPU research is that the simulation can be FPGA-accelerated. This bridges the gap between slow software simulations and expensive prototyping. FPGA-acceleration also allows for simulating larger systems than what is realistically possible in software. Being able to simulate larger systems is important to be more representative of real world GPUs.

In this thesis, I continue the work done in my project thesis [1] and the master's thesis of M. Rekdal [8]. In my project thesis, I implemented *CPI stacks for Vortex* (CSV) breaking down, and classifying the cycles of Vortex. I used CSV to further investigate the performance of Vortex. I found that Vortex was stalling mostly because the available instructions were waiting for memory requests to resolve, making it latency bound. This was possibly due to problems with Vortex'

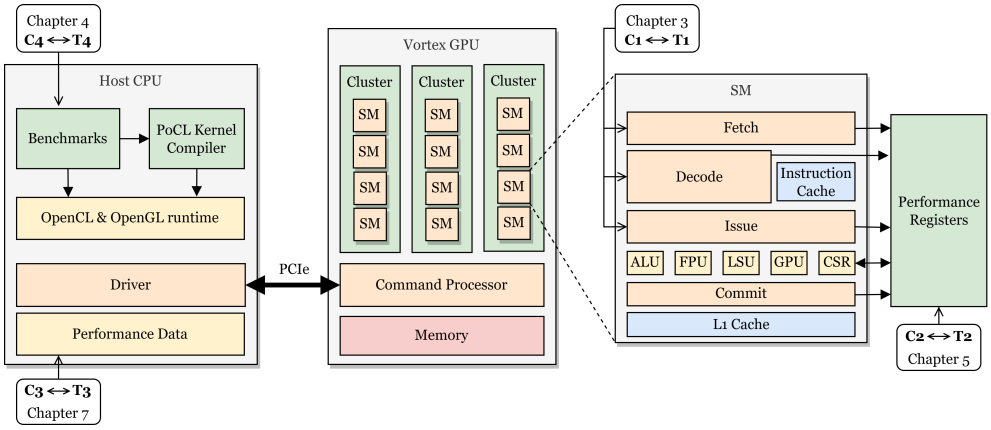



**Figure 1.1:** High-level overview of the Vortex GPU based on [7]. The overview also shows how the tasks and contributions relate, and where in the project the contributions are made.

schedulers and frontend. The issue scheduler was at times unable to issue ready instructions, and the frontend was struggling to fetch enough instructions. This reduced Vortex' throughput and its ability to exploit parallelism.

Figure 1.1 provides a high-level overview of the Vortex GPU and its environment. It also shows how the tasks and contributions presented in Section 1.2 and 1.3 relate to Vortex and to each other. I have first and foremost proposed and implemented improvements to the frontend and schedulers of Vortex' streaming multiprocessors (SMs). On average, these changes give a 71% decrease in frontend related stalls and a 5.4% decrease in CPI. Additionally, I have broadened the existing benchmark suite by porting 16 benchmarks from Rodinia [9, 10]. Lastly, I improve CSV, to give a better overview of why Vortex is stalling, allowing me to give a better evaluation of the improvements.

## 1.2 Assignment Interpretation

In this thesis, I will continue the work done in my project thesis [1]. The overarching goal of the project- and master thesis is to aid the Computer architecture lab (CAL) at NTNU to simulate and evaluate GPUs using FPGAs. I define the following list of tasks based on my interpretation of the assignment text:

- **T1** Propose and implement a set of improvements to the Vortex GPU based on results and CPI stacks obtained in my project thesis.
- **T2** Improve CSV to give a better overview of the issue stage.
- **T3** Evaluate the implemented improvements.
- **T4** If time permits, evaluate the observed problems and proposed solutions using a commonly used GPU benchmarks suite such as Rodinia.

Task **T2** was added during the development of the improvements, as I discovered that my existing version of CSV was too connected to Vortex' issue scheduler. This blocked CSV from giving a good overview of the issue stage and stall causes.

## 1.3 Contributions

In this thesis, I make the following key contributions:

**C1** I propose and implement changes to Vortex' fetch, decode and issue stage to increase fetch and issue bandwidth, solving the problems identified in my project thesis.

**C2** I improve upon CSV, enabling it to identify the stall cause for all warps in the instruction buffer and attributing them accordingly.

**C3** I evaluate the implemented improvements using CPI stacks and other collected performance metrics. I find that the changes move the frontend bottleneck to the backend of the Graphics processing unit (GPU), showing that Vortex is unable to hide latency stalls.

**C4** I expand Vortex' benchmark suite by porting a majority of the Rodinia benchmarks. Additionally, I improve core components of Vortex' mechanisms to collect performance metrics, making it more accurate and allowing multi-kernel programs to be used for benchmarking.

The tasks and contributions are linked one-to-one, i.e. **Tx** $\longleftrightarrow$ **Cx**. Figure 1.1 also describe how the tasks and contributions relate and where in the project the contributions are made.

## 1.4 Outline

Following is an outline of the rest of the thesis:

- **Chapter 2** covers background information regarding GPUs, the GPU programming model and GPU simulation.
- **Chapter 3** first describe in detail the main components of Vortex, and why the frontend is a bottleneck. Then it covers my proposed changes to solve the problems.
- **Chapter 4** contains details regarding how I ported a wide range of Rodinia benchmarks to run on Vortex.
- **Chapter 5** describes my methodology for collecting performance metrics to generate CPI stacks.
- **Chapter 6** includes information regarding the experimental setup and Vortex configuration.
- **Chapter 7** contains the results and evaluation of the proposed changes, in addition to a sensitivity analysis.
- **Chapter 8** contains the conclusion and thoughts regarding further work.

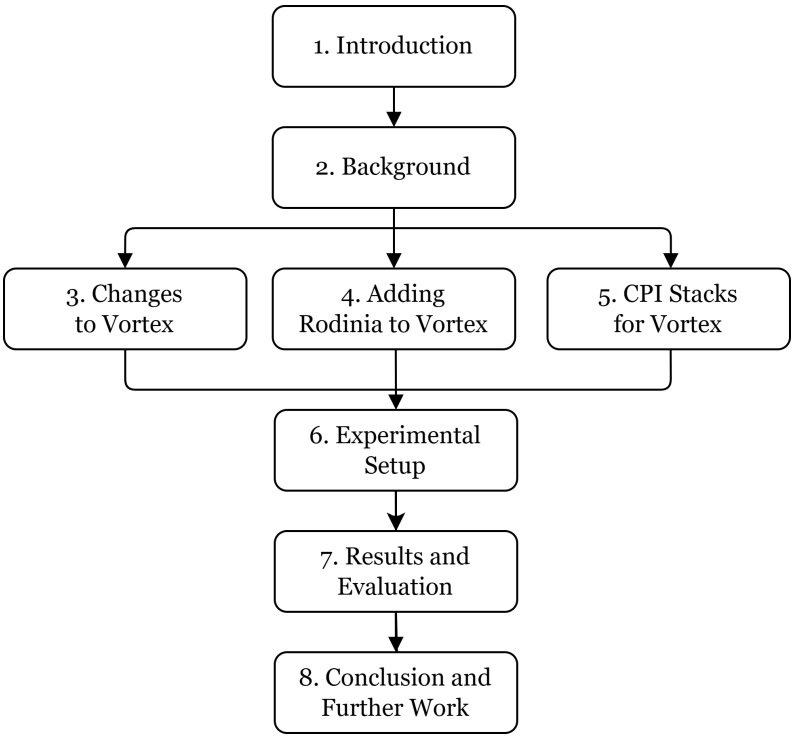



**Figure 1.2:** Outline of the thesis. Chapter 3, 4, 5 and 7 contain the contributions made by me.

# Chapter 2

# Background

## 2.1 GPU Overview

Graphics processing units (GPUs) are designed for executing highly parallelizable workloads. They achieve high throughput by exploiting SIMT. This section will first cover the GPU programming model, before describing the general architecture of a GPU.

### 2.1.1 GPU Programming Model

OpenCL is a commonly used programming framework for heterogeneous parallel computing for cross-vendor and cross-platform hardware. As Vortex supports OpenCL, I will refer to the OpenCL programming model [11], although the concepts described are transferable to CUDA. The program running on the GPU is known as the kernel. The kernel is a function which can be executed in parallel over a predefined number of dimensions. The kernel is divided into a set of *threads*, also referred to as *work-items*. As illustrated in Figure 2.1, these threads are grouped into *thread blocks* (TBs) which are also known as *work-groups* or *cooperative thread arrays* (CTAs).

All the threads execute the same kernel function in an N-dimensional domain over a region of memory, i.e. the threads execute the same instructions on different data, based on their thread and TB index. The size and number of dimensions of the TBs are determined by the *local work sizes*. The dimensions of the kernel grid and the number of TBs are then given by the *global work sizes* divided by the number of threads per TB. Both the *local* and *global work sizes* are given by the application when executing the kernel on the GPU.

After defining the grouping of the threads into TBs, each TB will be executed concurrently within a compute unit. All the threads within the TB will execute in lockstep, i.e. they will all execute the same instruction at the same time. An instruction executed by all the threads in a TB is known as a *warp*. If the kernel contains branches, the threads within a TB might diverge and have to execute different execution paths. As the threads are executed in lockstep, one execution

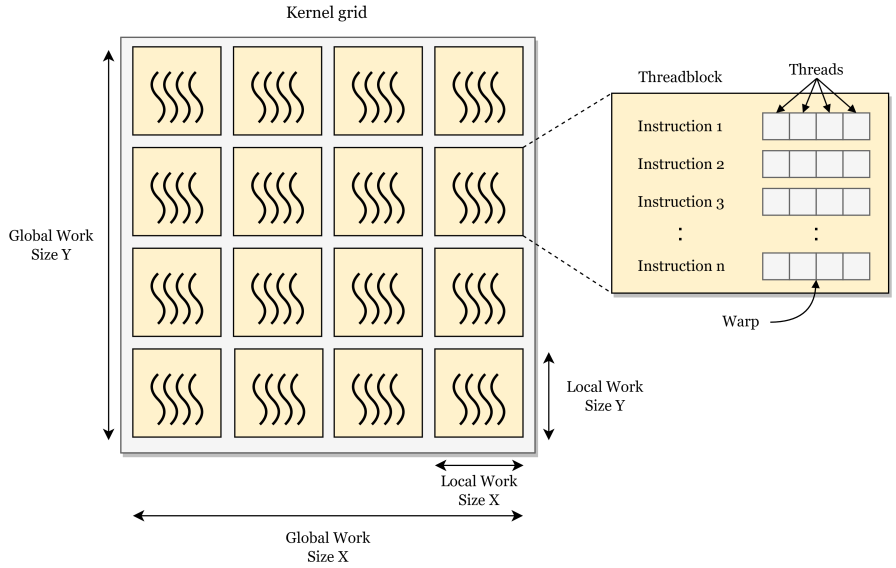



**Figure 2.1:** Illustration of how the kernel is divided into a grid of thread blocks with threads running in lockstep. The number of threads in the kernel is determined by the global work sizes, and the number of threads in each thread block is set by the local work sizes

path has to be executed before the other. When executing branching paths, threads are masked out to make only the correct threads execute the instructions. How this is handled varies based on the architecture of the GPU.

### 2.1.2 GPU Architecture

A high-level block diagram of a GPU is shown in Figure 2.2. To achieve high throughput for highly parallel workloads, GPUs dedicates a large number of its transistors to computation. GPUs have a number of *streaming multiprocessors* (SMs) all having a set of parallel execution cores. The SMs can execute a set of logically independent threads by executing each thread on an execution core. The threads do however run in lockstep, as the control logic is shared among the execution cores.

Typically each SM have a dedicated L1 cache, and share an L2 cache with the other SMs in the same cluster. These caches are often smaller than CPU caches as GPUs are less latency sensitive. By having multiple TBs allocated to each SM, the GPU can mask stalls by interleaving their warps. As SMs have multiple functional units (FUs) to execute different instructions, it is likely that if a warp stalls, a warp from another TB can be executed. This results in a high degree of thread level parallelism (TLP) and memory level parallelism (MLP). Due to the amount of MLP, GPUs require high memory bandwidth [12]. The NoC and memory system have to handle this high bandwidth requirement, which is why GPUs typically use specialized memory types such as GDDR or HBM.

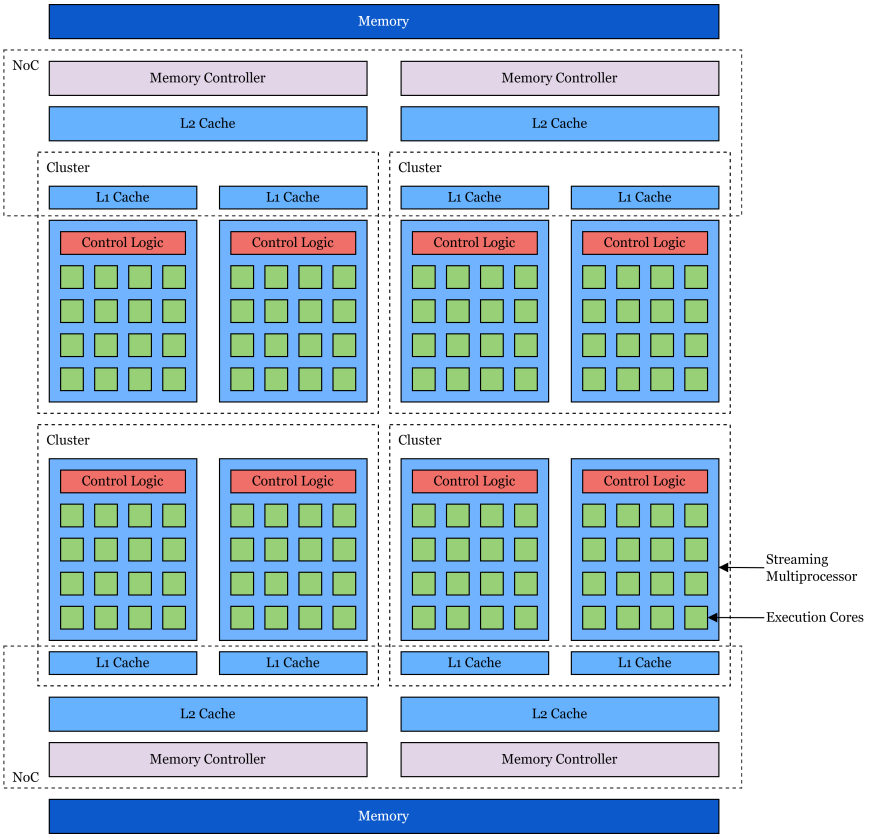



**Figure 2.2:** High-level block diagram of a GPU

When a kernel is executed on a GPU, the TBs have to be divided among the SMs. This is the job of the TB scheduler. The TB scheduler attempts to balance the workload evenly among the SMs during execution. Figure 2.3 shows an example of how a TB scheduler might distribute TBs among SMs. For Nvidia GPUs, it attempts to maximize TB occupancy [13], i.e maximize the number of TBs in SMs at all times. This is done by periodically obtaining information from every SM regarding the available resources over a dedicated network, and selecting the SM best fit for the next TB. The TB scheduler can account for factors such as data locality when selecting the best fitting SM. In the case of clustered SMs, the TB scheduler can also aim at distributing TBs evenly among the clusters, or map close TBs to the same cluster [14]. Having good load balancing will allow the SMs to execute the kernel efficiently and balance the load evenly to obtain high utilization of the SMs and reduce idling.

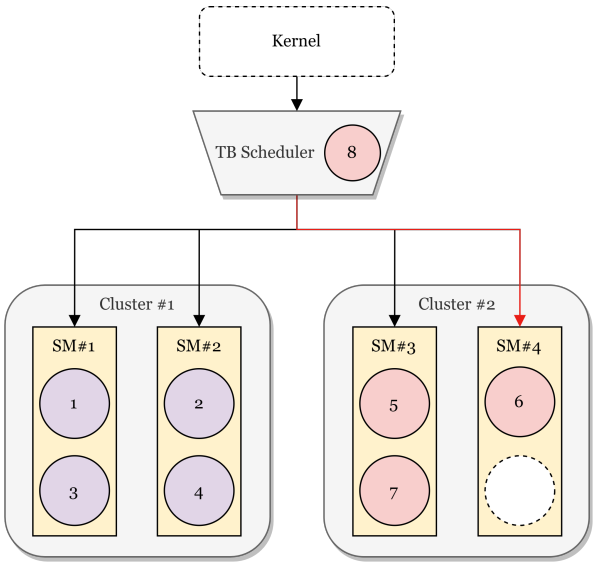



**Figure 2.3:** Thread block scheduler scheduling TBs in a round-robin order among the SMs in the same cluster, but attempting to map neighbouring blocks to the same cluster to exploit locality.

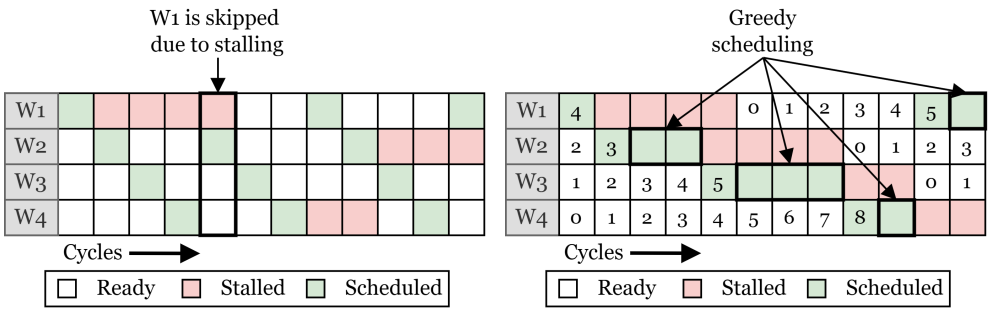
## 2.2 Warp Scheduling

The warp scheduler is a part of the SM's control logic, it selects the next warp to be executed. It does this by selecting one of the allocated TBs and scheduling its next warp. The scheduling algorithm used by the warp scheduler can be integral to the performance of the GPU. Two commonly used warp scheduling algorithms are loose round-robin (LRR) and greedy then oldest (GTO) [15]. Figure 2.4a and 2.4b respectively illustrate how LRR and GTO schedules warps. LRR schedules warps in a round-robin order. If a warp is stalled, it is skipped, and the next warp can be scheduled. GTO selects the TB with the oldest ready warp and schedules warps from the TB until it stalls [16].

In the case of memory intensive applications, LRR can cause a situation where all warps arrive at long latency stalls at the same time [17], as shown in Figure 2.5a. If all warps are stalled, the long latency stalls cannot be hidden, resulting in low throughput. The goal of the GTO scheduling algorithm is to reach a stall for a single warp before scheduling other warps. This may enable more stalls to be hidden, as shown in Figure 2.5b, and even achieve better cache locality [16, 17].

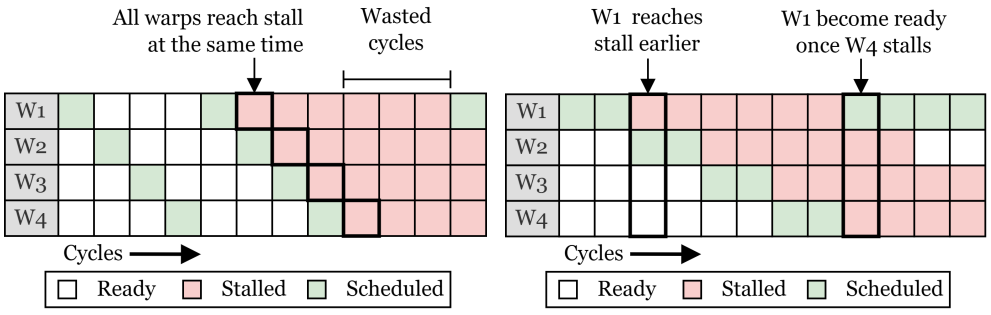
There are also other notable warp schedulers which can give somewhat better performance than GTO, but also require more information about the state of the GPU and the warps.

*Cache-conscious wavefront scheduling* (CCWS) [16] attempts to dynamically determine how many, and which warps should be allowed to access, using feedback from the L1 cache. The goal of CAWS is to reduce the number of memory




**(a)** Loose round-robin **(b)** Greedy then oldest

**Figure 2.4:** Demonstration of how LRR and GTO selects warps for scheduling. For GTO, the ages of the ready warps are written in the cells




**(a)** Loose round-robin **(b)** Greedy then oldest

**Figure 2.5:** Demonstration of how LRR and GTO performs in conjunction with long-latency stalls. GTO is able to hide the stalls, while LRR is unable to because all warps stall at the same time

accesses between re-references by the same TB, to keep cache hit rates high.

*Criticality-aware warp scheduling* (CAWS) [18] attempts to improve the execution time of critical warps, i.e. warps which require more time and resources to complete. By doing this, the execution time disparity between the TBs is equalized, reducing the total execution time.

*Lazy warp scheduling* (LWS) [15] attempts to schedule low-latency warps first, such that they can be masked by scheduling longer latency warps afterwards. By prioritizing low-latency warps, there are fewer stall cycles.

## 2.3 GPU Simulation

GPGPU architecture research is mainly focused on using software simulations [19–21] modeling the architecture at the intermediate language (IL) level, such as PTX and HSAL. There are however significant deviations between GPU simulators using high-level abstractions and real hardware. Simulating IL instructions

can add up to 33% error when comparing the absolute runtimes to real hardware [22]. High-level abstraction models have substantially less functional state associated with the instructions. Thus they are unable to model important micro-architectural interactions, such as instruction fetching and control flow divergence.

To obtain results which most accurately reflect the performance of an architecture, cycle-accurate simulations are required. A solution to the inaccurate high-level abstraction simulations is register transfer level (RTL) implementations. These implementations are cycle-accurate, but require substantially more time and memory to simulate. This is because the entire state of the system is represented. There exist several RTL implementations of open-source GPGPUs, such as MIAOW [23], Nyami [24] and FlexGrip [25]. However, the ISAs used in these GPUs are either custom or proprietary, which restricts application support. Vortex solves this by basing its ISA on the open source RISC-V ISA and including a custom compiler. By additionally having OpenCL support, adapting applications for Vortex becomes easier.

In addition to creating Vortex, the team at Georgia Tech presented Skybox [26] at ASPLOS 2023. Similarly to Vortex, Skybox can be FPGA-accelerated, but is more focused on rendering graphics. It has support for Vulkan, a modern graphics rendering API, and it has a hardware rasterizer and render output unit (ROP). This results in a GPU more suitable for graphics workloads than Vortex, as the Vortex GPGPU is mostly suited for compute workloads.

Simulating entire systems in software is rather slow, especially for large parallel systems as the simulations are difficult to parallelize due to fine-grained synchronization [27, 28]. To speed up architecture simulation, FPGA-acceleration can be used. RAMP-gold [29], an FPGA multicore simulator, achieved a 263× speedup over GEMS [30], a software-based simulator. FPGA-acceleration serves as a middle ground between software simulation and ASICs. As high-end FPGAs are becoming more prevalent in the consumer market, implementing full-feature GPGPUs is becoming a possibility.

There is however a critical problem when running FPGA-accelerated simulations, modeling the timing and behaviour of I/O and peripherals [31], e.g. DRAM. To obtain representative results using FPGA-accelerated GPUs, both the memory bandwidth and the latency needs to be scaled to match the discrepancy between FPGAs and ASICs. Chipyard [31] achieves this using Firesim [32]. Firesim, use a token mechanism which can stall individual SoCs of the simulated system to advance the system in target time. Vortex does not have any mechanisms to solve this problem, however, work is currently being done at NTNU's computer architecture lab (CAL) to implement Vortex into Chipyard. Meanwhile, I have to simulate Vortex and DRAM in software to obtain representative results.

# Chapter 3

# Changes to Vortex

In this chapter, I will first give an overview of the Vortex architecture and pipeline. Then I will describe the identified bottlenecks, and how I propose fixing them.

## 3.1 Vortex Architecture

Other than what is written in the Vortex paper [1], Vortex is mostly undocumented. Most of my understanding of Vortex' architecture and software stack is therefore derived from reading the source code. The Vortex microarchitecture is illustrated in Figure 3.1 and 3.2. Each SM is a 5-stage pipeline containing a fetch, decode, issue, execute and commit stage. The SMs of Vortex all have their own L1 and instruction caches. There are also options for Vortex to include L2 and L3 caches, which are shared among SMs, as shown in Figure 3.2. If the L2 or L3 caches are not included, they are replaced by memory arbiters. Vortex' pipeline is elastic [33], i.e. it is using *ready* and *valid* signals to communicate between the modules and pipeline stages. This makes it easier to add new or configure existing components, as the modules can be more flexible in the number of cycles they use.

### 3.1.1 Vortex ISA

Vortex extends the RISC-V ISA [34] with six new instructions. The added instructions are essential primitive for supporting the SIMT execution model and graphics processing. All of the instructions are RISC-V R-Type instructions and fit in one opcode.

- **wspawn**: Controlling warps by activating a number of warps at a specified PC.
- **tmc**: Controlling threads by activating or deactivating threads within a warp.
- **split & join**: Handling control divergence. *Split* pushes information about the current state of the thread mask and branching to the immediate postdominator (IPDOM) stack, and *join* pops the information off the stack to reconverge the branches.

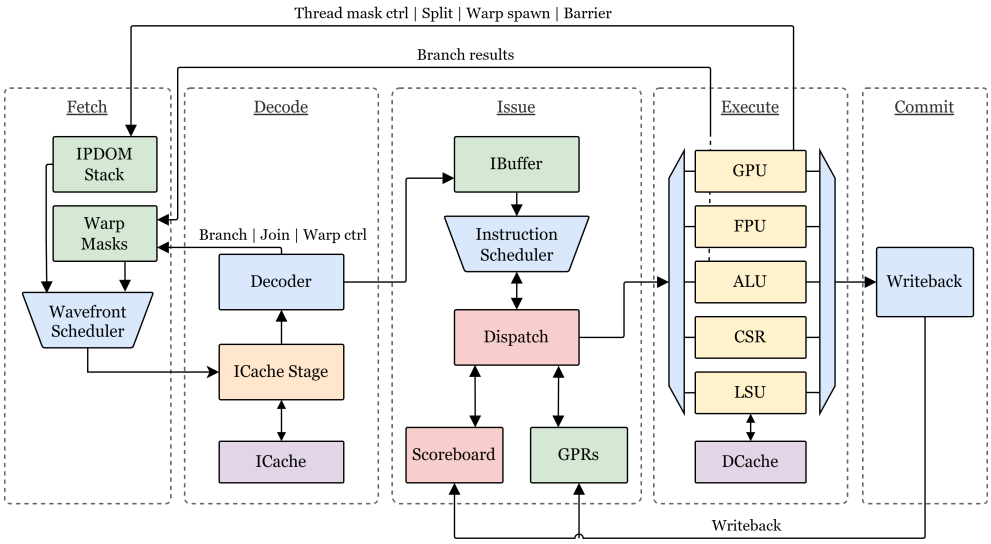



**Figure 3.1:** Vortex RISC-V 5-stage pipeline of a streaming multiprocessor

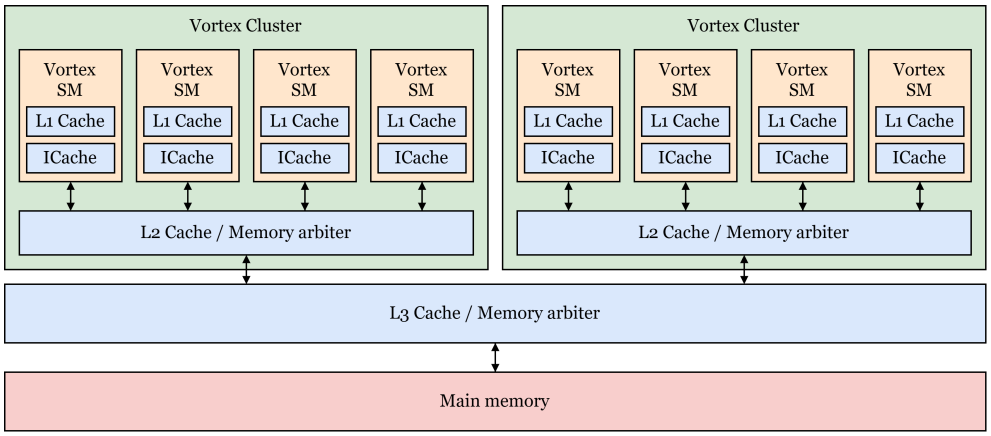



**Figure 3.2:** Example of SMs clustering in Vortex using 2 clusters and 4 SMs per cluster. The L2 and L3 caches are optional, if they are not included, memory arbiters are used.

- **bar**: Synchronizing warps, both intra-core and inter-core, using barriers. The barrier is released when the expected number of warps has reached it.
- **tex**: Texture lookup using normalized coordinates and texture mipmap level. Other specifications regarding the texture lookup, such as dimension and format are configurable via CSRs.

### 3.1.2 Pipeline

Following is a description of the five pipeline stages shown in Figure 3.1.

**The fetch stage** is responsible for scheduling warps, and keeping track of branches, divergence and barriers. The warp scheduler keeps track of the active, stalled and blocked warps using bitmasks. To select which warp to schedule next, the warp scheduler uses a *find-first* algorithm. The find-first algorithm prioritizes warps based on their warp ID. This will be further explained in Section 3.2.3. Upon scheduling a warp, the warp scheduler sends an instruction fetch request to the icache-stage in decode. In the baseline version of Vortex, only one warp from each TB can be fetched concurrently. Since Vortex does not support branch prediction, the warp has to be stalled until it is known that it cannot change the control flow. The first point in the pipeline where this can be known is after decoding the instruction. If the instruction is not a branch, barrier or thread mask control, the warp will be marked as ready in the warp scheduler. Otherwise, the warp continues to be stalled until the instruction completes execution.

**The decode stage** contains the decoder and the icache-stage. The icache-stage is illustrated in Figure 3.3. The purpose of the icache-stage is to enable fetching instructions from multiple TBs concurrently. The icache-stage uses dual-port RAM with one address per TB in the SM, to store information about the request. The request is simultaneously sent to the icache. Upon receiving a response from the icache, the corresponding request information is read from RAM and combined with the instruction data to create the response. The response is then sent to the decoder, which decodes the instruction using combinational logic. The decoder also informs the fetch stage whether or not the instruction can change the control flow. After decoding, the instruction is sent to the instruction buffer in the issue stage.

**The issue stage**, shown in Figure 3.4, is responsible for issuing the warps to the functional units. To handle data dependencies, Vortex utilize scoreboarding. The warps are issued in-order, but are committed as soon as their execution completes. When instructions are decoded, they are transferred to the instruction buffer (ibuffer). The ibuffer contains a queue for each TB in the SM. The issue stage has an instruction scheduler which schedules warps from the front of the ibuffer and dispatches it to the corresponding functional unit in the execution stage.

The instruction scheduler used in the baseline version of Vortex is a round-robin scheduler. For a warp to be issued, the source and destination registers must be available in the scoreboard and the corresponding functional unit must be available in dispatch. The instruction scheduler attempts to issue the warps by selecting

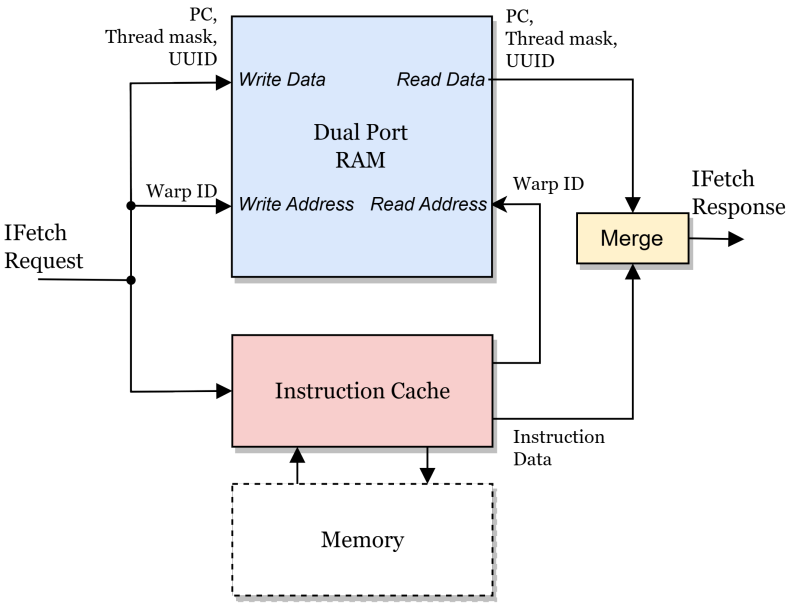



**Figure 3.3:** Illustration of Vortex' icache-stage.

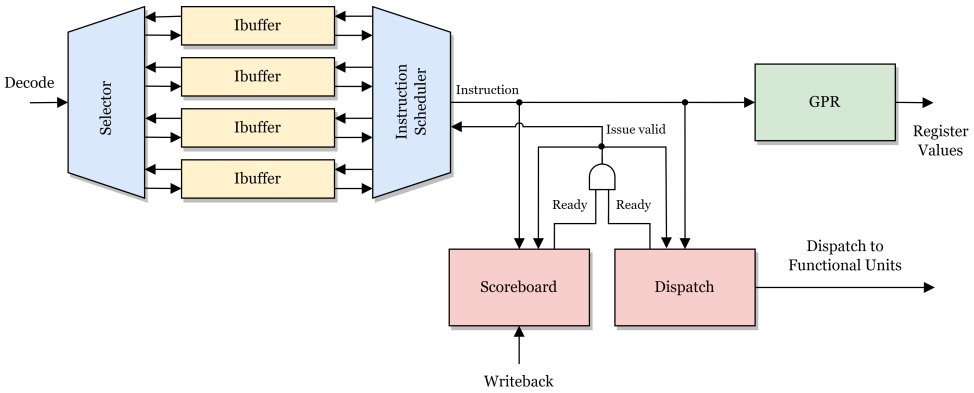



**Figure 3.4:** Illustration of Vortex' issue stage

one of the warps available in the ibuffer. The issue stage can however only check for one warp whether it is ready or not in each cycle. Thus if the warp selected by the instruction scheduler cannot be issued, no other warp can be issued in that cycle.

**The execute stage** consists of five functional units. The *arithmetic logic unit* (ALU) performs logic and integer arithmetic operations in addition to handling branches. The *load-store unit* (LSU) performs memory loads and stores. The *control status register* (CSR) holds a number of status registers which can be read and written to using CSR instructions. Some of these influence how graphics are rendered, by controlling sampling modes, texture addresses etc. The CSR also tracks a number of performance metrics, such as the number of committed instructions and the number of cycles used. The *graphics processing unit* (GPU) performs texture sampling as well as sending control signals to the fetch stage in regards to warp spawning, divergence, barriers and thread masks. Lastly the *floating point unit* (FPU) performs floating point operations.

**The commit stage** is the last stage in the pipeline. When instructions are finished in the execute stage, the results are written back to the general purpose registers (GPR). The registers, which were reserved by the instruction, are also released in the scoreboard.

### 3.1.3 Workload Distribution

Vortex' workload distribution and TB scheduling is performed statically. Upon starting the execution of a kernel, each SM calculates the total number of active SMs required. The number of SMs required is calculated as:

$$\#SMs_{active} = \min\left(\frac{\#TBs}{\#WarpsPerSM \times \#ThreadsPerSM}, \#SMs\right) \tag{3.1}$$

The TBs are then divided evenly among the active SMs. If there are more TBs than the total number of slots available in Vortex, the last SM is allocated all of the remaining TBs. This can result in very inefficient workload distribution. To get the best performance, programs have to be adjusted before compilation to best fit the architecture. Static workload distribution also has another weakness. If for some reason the SMs require different amounts of time to complete their workload, the SMs which finish early, end up idling while waiting for the other SMs to complete their workloads. The difference in execution time can stem from multiple causes. The memory system could for example prioritize requests from certain SMs, or the workloads could differ in terms of divergence. This results in unused computational power and reduces the throughput of the GPU. While I do not propose or implement any solutions to this problem in this thesis, it is relevant for understanding the results and evaluation in Chapter 7.

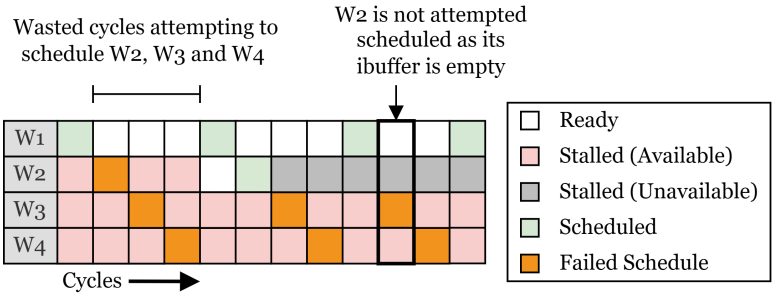



**Figure 3.5:** Demonstration of the baseline issue scheduler. The *unready* LRR scheduler wastes cycles by attempting to issue stalled warps from the ibuffer

## 3.2 Scheduling Algorithms

### 3.2.1 Ready Scheduling

The baseline instruction scheduler, described in Section 3.1.2, attempts to issue an available warp before checking if it is ready. As the scheduler is round-robin, this can potentially be detrimental to the performance. If 1 out of $N$ warps are ready, while the $N-1$ other warps are stalling, the scheduler could potentially waste $N$ cycles before issuing the warp. An example of this is illustrated in Figure 3.5. While this is the most extreme case, cases of similar severity are likely to occur. Using a *ready scheduler*, which checks for ready warps before scheduling, will guarantee that it only takes 1 cycle to schedule a warp if at least one is ready.

Figure 3.6 illustrates the new issue stage proposed by me. It performs the ready check for all warps before the instruction scheduler selects which warp to issue. Each warp checks if it is ready by: ① Checking in dispatch if the corresponding FUs is available. ② Checking in the scoreboard if the required operands are available. ③ Use bitwise AND to create a bitmask of ready warps. ④ Create a bitmask of valid warps which can be issued, by finding the bitwise AND of the ready warps and the warps available in the ibuffer. The instruction scheduler can then select a ready warp from this bitmask. ⑤ The selected warp ID is then sent back to the ibuffers, dispatch and scoreboard to update their values. These changes result in a ready loose round-robin (LRR) scheduler as described in Section 2.2.

The implementation of the new issue stage does not require much additional hardware. All the information is already available in the scoreboard and dispatch, thus it mainly requires selectors for reading the correct registers. The scheduler does not need to be changed, as it continues to select a warp from a bitmask.

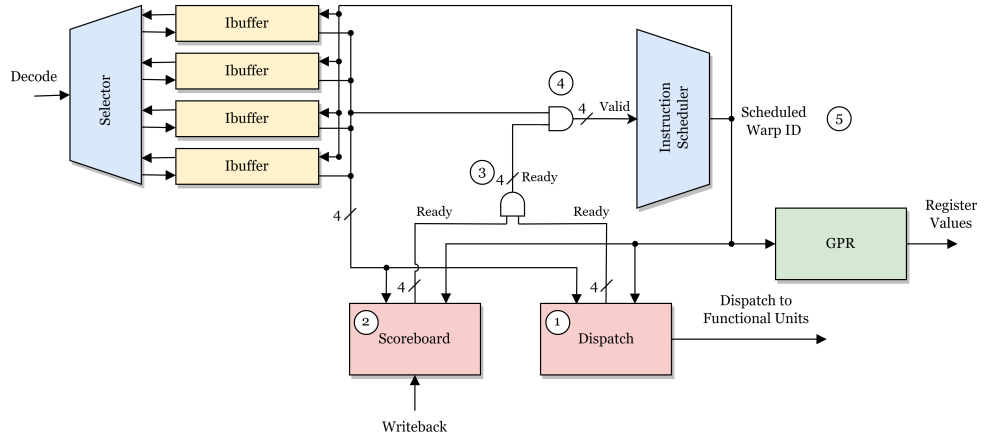



**Figure 3.6:** Illustration of the new issue stage allowing the instruction scheduler to check for ready warps before selecting which warp to issue.

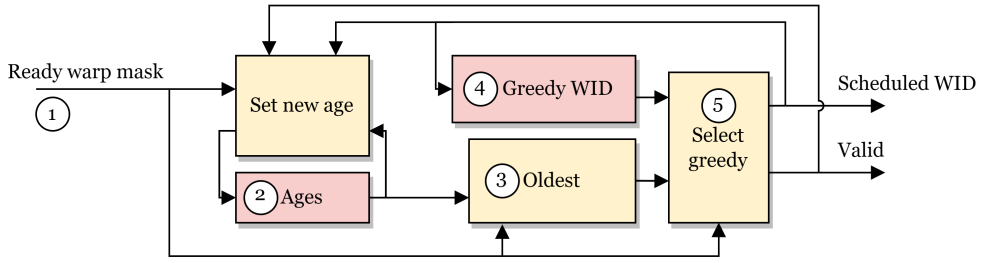



**Figure 3.7:** Implementation of greedy then oldest (GTO).

### 3.2.2 Greedy then Oldest

After implementing ready scheduling, the possibility of implementing other scheduling algorithms becomes available. Greedy then oldest (GTO) is a common and easy-to-understand algorithm with a potential to give improved performance over loose round-robin (LRR). I was unable to find implementation details of existing GTO schedulers, thus I had to implement my own. Figure 3.7 illustrates my implementation of GTO. ① Is the set of ready warps in the form of a bitmask. ② Contains the ages of the warps, i.e the number of valid schedules which occurred, while the warp was ready. The age is calculated as described by Equation 3.2. Issuing a warp might cause structural stalls for the other warps. Thus an alternative would be to use the warps' presence in the instruction buffer rather than it being ready, as a condition for incrementing the age. ③ Use combinational logic to find the oldest ready warp using the ready mask and the age of each warp. ④ Is the warp ID of the previously scheduled warp. If the previously scheduled warp ID is still ready, the greedy selector ⑤ continues to select this warp, otherwise, it selects the oldest ready warp.

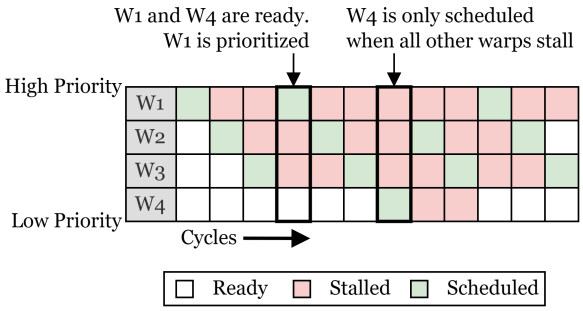



**Figure 3.8:** Demonstration of Vortex' *find-first* warp scheduling algorithm. When the length of the stalls is shorter than #Warps − 1 cycles, the low-priority warps are scheduled substantially less than the high-priority warps.

$$\text{age}_i = \begin{cases} \text{age}_i, & \text{if valid} = 0 \text{ or ready}_i = 0 \\ 0, & \text{if valid} = 1 \text{ and Scheduled WID} = i \\ \text{age}_i + 1, & \text{otherwise} \end{cases} \tag{3.2}$$

### 3.2.3 Matching Warp and Issue Schedulers

The scheduling algorithm currently used by the warp scheduler in fetch is *find-first*. This algorithm prioritizes warps based on warp ID. Figure 3.8 illustrates how this algorithm can end up scheduling low-priority warps substantially less than high-priority warps. This can make some warps finish long before others, resulting in idle cycles. This does not happen for the baseline version of Vortex, as there are few warps and too many frontend stalls. However, as I am going to reduce the number of frontend stalls in the next section, it is probably better to use a fairer algorithm.

The interaction between the instruction scheduler and the warp scheduler is likely important for the algorithms to achieve their goals. For example, if the warp scheduler fetches warps in LRR order, a GTO instruction scheduler will not have enough warps from the same TB to see the effects of greedy scheduling. It is thus desirable to use scheduling algorithms in the warp and instruction schedulers which can support each other. To do this, I also implemented the LRR and GTO algorithms in the warp scheduler.

Implementing the algorithms in the warp scheduler is quite simple, as the modules can be reused. The warp masks in fetch can also continue to be used as input for the warp scheduler. An issue which arises when implementing GTO in the warp scheduler is that GTO ideally wants to fetch warps from the same TB multiple cycles in a row. As explained in Section 3.1.2, the warps are stalled after being scheduled to avoid potential control flow hazards. Thus a GTO warp scheduler will not have the intended behaviour when integrated into the existing pipeline. Section 3.3.1 will describe changes to the frontend, which solve this issue.

## 3.3 Frontend

In my project thesis [1], I found that Vortex' frontend was unable to fetch enough instructions to the issue stage, impeding its ability to hide stalls. This section describes a set of changes done to improve the throughput of the frontend.

### 3.3.1 No Stall Scheduling

The root of the frontend problem is that TBs are stalled whenever their warps are fetched. Most of the instructions do however not require stalling and are unstalled after being decoded. As the default configuration of Vortex has 4 TBs per SM, it is unable to hide the latency of the instruction fetch. The high number of stalls thus reduces the throughput of the frontend. To avoid this, I propose *no stall scheduling* (NSS), allowing the frontend to schedule warps without stalling, and instead flush the frontend if stalling is required. To further improve the efficiency of NSS, I propose *stall-prediction* to reduce the number of flushed instructions, and to make the frontend stall only when required.

To implement NSS, two mechanisms are required. First, the icache-stage needs to handle multiple concurrent instruction fetches from the same TB. The responses to these requests have to be reordered, as the icache is non-blocking. Secondly, the requests have to be flushed in case a stall is required. Figure 3.9 illustrates my design of the improved icache-stage. Instead of using dual-port RAM, which allows only one concurrent fetch request for each TB, the new design uses an *insert queue*, which allows for up to 8 concurrent instruction fetches per TB. It is possible to implement an even larger queue, but I found that the number of in-flight requests per TB rarely exceeded 8. The insert queue has two ports for writing data. ① First the fetch request is pushed into the queue and sent to the icache. By using a queue, the order of the requests is maintained. ② Secondly, as the response is returned, the instruction data is inserted into the queue. The UUID of the request and response is used to match the data with the request. When the first element in the queue is ready, i.e ③ it contains both a valid request and the instruction data, it becomes available for ④ the scheduler to forward it to decode.

A scheduler is required, as it is possible for multiple queues to have ready elements at the same time. This is because the response data can be reordered, or the ibuffer can be full, causing back pressure. I elected to use GTO for the icache-stage scheduler as it is likely to conform with both an LRR and GTO issue scheduler. I do however believe that the choice of algorithm will not impact the performance. Some preliminary testing also support this belief.

To not stall every time a fetch request is sent, the warp scheduler has to predict if the instruction requires stalling or not. NSS initially predicts that instructions do not require stalling. This scheme has low overhead, as the scheduling continues as if nothing changed. Upon decoding an instruction, it might require the frontend to stall. When this happens, ⑤ all requests in the corresponding queue are flushed. This is done in the insert queue by setting the valid bit of each entry low. While the

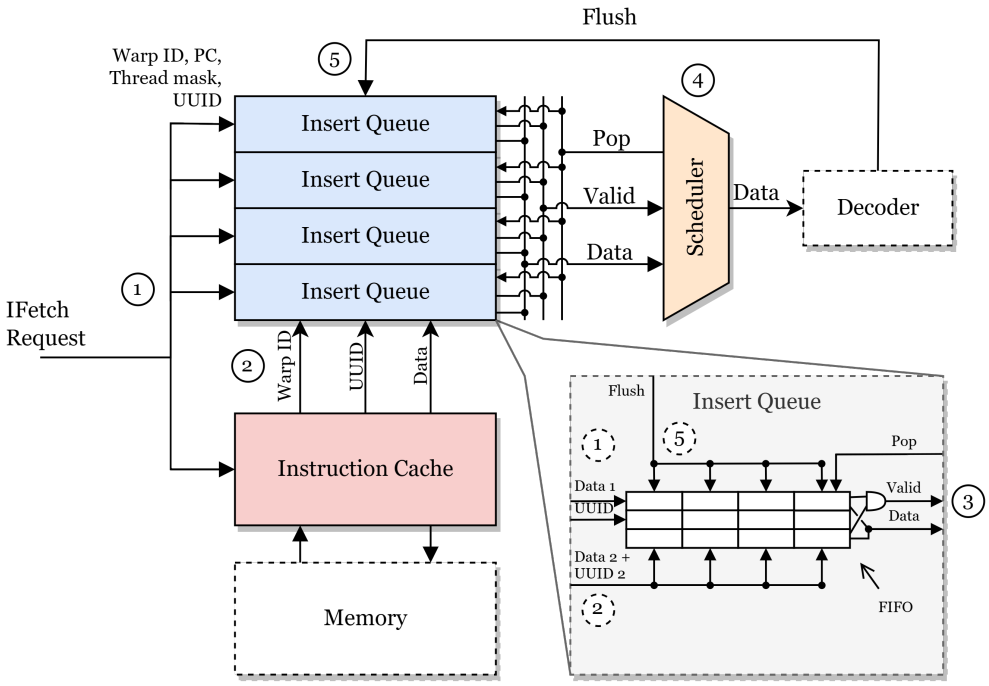



**Figure 3.9:** Improved icache-stage allowing for multiple concurrent instruction fetch requests from each TB

requests are flushed from the insert queue, the requests continue to be processed by the instruction cache. When data from flushed requests are returned, there will not be any valid matches in the queue, and the data will be ignored. Note that in the case of a join instruction, it would be possible to change the thread mask after fetching the instructions, and not flush. This would however require additional control logic and is not done in this thesis.

Upon flushing, the warp is marked as stalled in the warp scheduler, and the PC is set to address after the instruction requiring the flush. This is done by sending the PC of the stalling warp back to the warp scheduler from decode. As the instructions now only stall when required, there are more opportunities for what order the warps can be fetched. Because of this, using a GTO warp scheduler is now possible.

### 3.3.2 Stall Prediction

The implementation of NSS presented in Section 3.3.1 has one main weakness. The cost of mispredicting is large. Mispredicting will likely waste cycles, as there is a high probability that the GPU could fetch other non-speculative warps. The cost is therefore not relative to not fetching any warps, but rather relative to scheduling a number of other warps. To combat this issue I propose implementing *stall-prediction*, allowing the warp scheduler to learn if an instruction will require the frontend to stall.

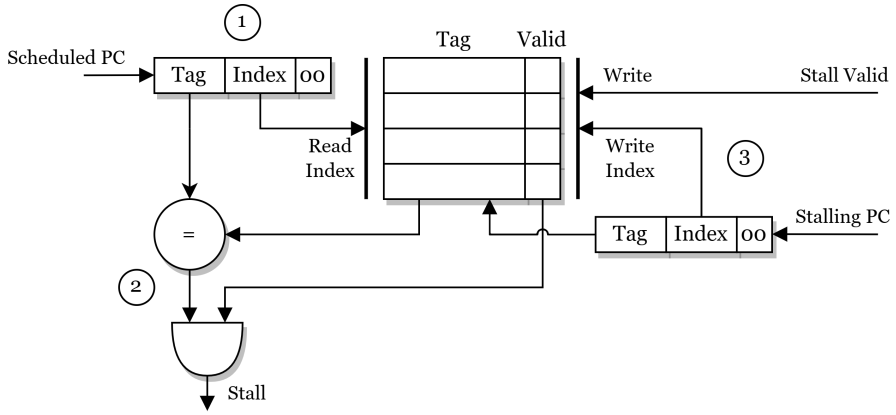



**Figure 3.10:** Illustration of the stall-table, which is used to check if an instruction is known to require stalling

Figure 3.10 shows the *stall-table*, which is used to learn which instructions are causing a frontend stall. The stall-table is included in the fetch stage and is used to check whether the next warp should be blocked from being fetched. ① The PC of the fetched warp is used as an index in the stall-table. It is split into a number of tag and index bits. The number of index and tag bits is dependant on the size of the table, see Equation 3.3. ② If the tags are matching and the table entry is valid, the instruction will require the next warp to stall. By stalling the next warp, other warps can be fetched instead of continuing to fetch instructions which will be flushed.

$$\begin{aligned} \#IndexBits &= \lceil \log_2(TableSize) \rceil \\ \#TagBits &= \#PCBits - \lfloor \log_2(TableSize) \rfloor - 2 \end{aligned} \tag{3.3}$$

For the stall-table to learn, ③ the PC and stall status of the decoded instructions are sent to the stall-table. If the instruction is a stall, the tag and the valid bit are set at the corresponding index. Note that this may invalidate a previous entry by overwriting the tag. If the instruction does not cause a stall, nothing is written to the stall-table. Because of this, the stall-table contains only instructions which are known to cause a stall.

The index bits can be any subset of the PC bits, but I elected to use the least-significant bits (LSBs), except the last two which are always zero. By using the LSBs as the index, the stall-table will be able to handle consecutive stalling instructions. The stall-table will only be effective when instructions are executed multiple times, i.e. loops, or TBs executing the same instructions. Thus it is more valuable for the stall-table to have high accuracy within loops, than wide coverage. Preliminary testing showed that having 128 entries gave a high hit rate in the stall-table. Increasing the size further gave diminishing returns and did not improve the hit rate by much.

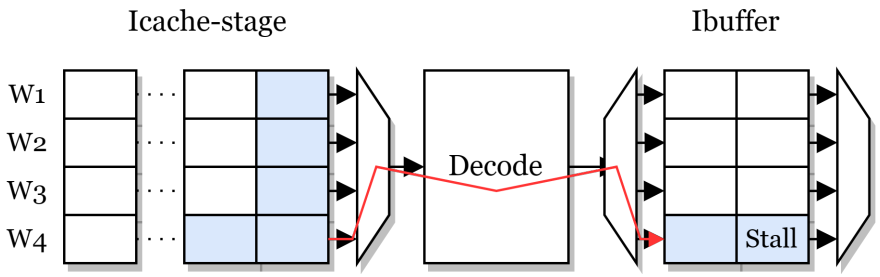



**Figure 3.11:** Back pressure from the instruction buffer to decode. Ready warps in the icache-stage are blocked from being decoded and sent to the instruction buffer because the instruction buffer of the decoded warp is full. The instruction buffers of $W_1, W_2$ and $W_3$ are empty, and have to wait for $W_4$ to issue before being filled

### 3.3.3 Back Pressure Reduction

When implementing NSS and stall-prediction, the throughput of the frontend is increased. The increase in throughput may cause the instruction buffer to fill up, which can result in back pressure, as illustrated in Figure 3.11. This occurs when the decoder decodes an instruction that there is no space for in the instruction buffer. As the decoder is unable to send the instruction to issue, it is blocking until there is space in the ibuffer. This may be problematic because the back pressure could prevent ready instructions from being decoded. This will impair the throughput, as these warps might be able to issue.

This problem can be resolved by implementing what I will refer to as back pressure reduction (BPR). BPR is quite simple a bitmask sent from the ibuffer to the icache-stage indicating which ibuffers are full. This allows the icache-stage to always decode instructions which will not cause back pressure. This signal is also sent to the warp scheduler such that it can schedule other warps.

# Chapter 4

# Adding Rodinia to Vortex

One of Vortex' weaknesses is its benchmark suite. Vortex includes only a few benchmarks, all of which are quite small and which behaviours does not match real-world applications. When lacking reasonable benchmarks, it is difficult obtain conclusive results about the performance of Vortex. To solve this, I brought Rodinia [9, 10], a commonly used set of benchmarks for parallel computing [35], to the Vortex ecosystem. This chapter describes the work done to enable Rodinia benchmarks to run on Vortex.

## 4.1 Reading Performance Data

All the benchmarks included with Vortex execute one kernel once. The existing setup for gathering performance data, created by the Vortex team, took advantage of this. Vortex utilize internal performance counters to collect performance data. Each SM has its own counters, which are accessible as addressable CSR registers. Before a kernel is enqueued, the GPU and the performance registers are reset. After the reset, the kernel begins executing. Throughout the execution, different metrics are collected and stored in the CSR registers. When an SM finishes the execution of its allocated workload, it ends by reading the relevant CSR registers and writing their contents to memory. The code related to dumping the performance counter to memory is included in every kernel by a stub. At the end of the benchmark, the host can read the memory, and collect the performance metrics for each of the SMs, and the GPU as a whole.

Most of the benchmarks in Rodinia have more than one kernel, or execute a single kernel multiple times. When executing a multi-kernel benchmark, only the last execution would be recorded, because the GPU would reset between kernel executions. Obtaining performance data for all kernels in a benchmark is important as they might possess radically different behaviours. An example of this would be the *streamcluster* benchmark which has two kernels: *memset* and *pgain*. The memset kernel sets global memory to a given value. This results in a short loop and many writes to memory. The pgain kernel is more complex, having multiple

branches, loops, calculations and memory operations. As the kernels are so different, it is important to capture the behaviour of both to be representative of the benchmark's performance.

The method used to read the performance data also has other issues which affect the accuracy and performance of the simulation.

1. The process of reading performance registers and writing data to memory, alters the results while reading. The performance metrics read first, will not represent the same state as the ones read last.
2. If SMs does not finish at the same time, performance metrics are written to memory while other SMs are running. Writing to memory may reduce the available memory bandwidth for the remaining cores.
3. For programs running multiple kernels, or a single kernel multiple times, it is unnecessary to write the data to memory at the end of each kernel. This is especially significant when using software simulation, as it increases the simulation time.

To solve these problems, I had to stop the performance metrics from being reset between kernel executions, and only collect performance metrics at appropriate times. To do this, I introduced two new CSR registers: *perf-lock* and *perf-reset-lock*. Setting *perf-lock* stops the collection of performance metrics. When *perf-reset-lock* is set, the performance metrics are not reset when the GPU resets. However, as *perf-reset-lock* controls what is reset, it cannot be controlled by reset itself. To ensure the performance metrics are reset before starting the benchmark, I have to first execute a kernel to specifically set the lock low. This solves the problem of benchmarking multiple kernels.

Problems 2 and 3 listed above, can be resolved by moving the code responsible for writing the performance data to memory, into a separate kernel which is executed after the benchmark has been completed. Doing this makes sure that all of the SMs complete their workloads before starting the process of reading performance data. The kernel reading the performance data, sets the *perf-lock* to ensure that it does not interfere with the data while reading, thus also solving problem 1.

The final solution, illustrated in Figure 4.1, can be divided into three phases. ① Before starting the execution of a benchmark, the initializing kernel is executed. It is responsible for setting the *perf-reset-lock* low, to allow the performance metrics to reset correctly. ② The kernels belonging to the benchmark are then executed. They also set the *perf-reset-lock* high, to keep the data from resetting. The benchmark continues to execute its kernels until it has been completed. ③ Finally, the performance dump kernel is executed, setting the *perf-lock* high to stop it from altering the results.

To make this solution viable to implement for all benchmarks, I created a header file, containing macros for creating and starting the two new kernels. To add new benchmarks to Vortex, it is only required to include the header and insert the macros at the correct location. The macros take the current OpenCL context, command queue and device ID as input. An example snippet of how this can be

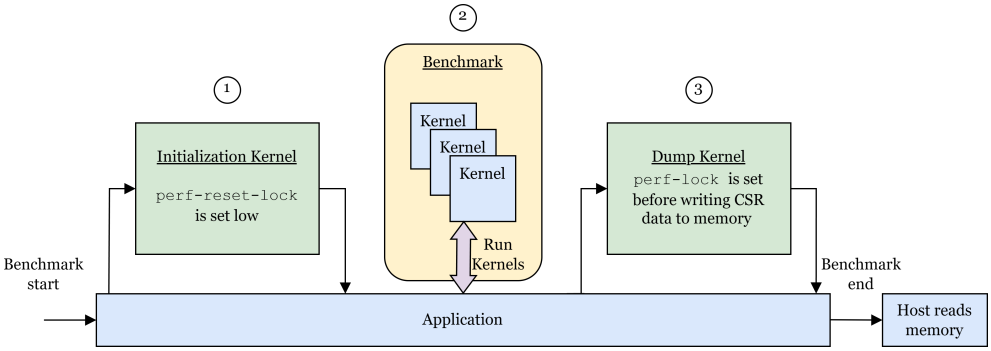



**Figure 4.1:** Timeline of the three stages of multi-kernel benchmarking: *initialization*, *execution of benchmark* and *dumping of performance data*

used is shown in Code listing 4.1. Doing this allowed for more rapid adjustments to the implementation, in addition to making the process of adding new benchmarks quicker.

**Code listing 4.1:** Example of using the perf macros to create and use the initialization and dump kernels

```
#include "../vortex_perf.h"             // Header containing perf macros

PERF_VARIABLES // Macro declaring variables required to run perf kernels

// -- Code for setting up OpenCL: create context, command-queue and device

PERF_CREATE_PROGRAM(context, device_id) // Macro for creating the perf kernels
PERF_ENQUEUE_INIT_KERNEL(command_queue) // Macro for starting the init kernel

// -- Code for running the benchmark

PERF_ENQUEUE_DUMP_KERNEL(command_queue) // Macro for starting the dump kernel
PERF_CLEANUP                            // Macro for perf kernel cleanup
```

## 4.2 Adapting Benchmarks for Vortex

When simulating Vortex in software, Verilator [36] is used to compile the SystemVerilog implementation of Vortex into a vortex library. This library can simulate the entire GPU cycle-accurately. When executing a benchmark, it is linked with the Vortex driver and Vortex library generated by Verilator. This allows the host to treat the simulated GPU as if it was a normal GPU using an OpenCL interface. The software and FPGA simulations thus provide the same interface. Because of this, the changes done in this section should also work for FPGA-accelerated simulations.

### 4.2.1 Offline Compilation

Vortex has support for OpenCL, and uses PoCL [37] to implement the compiler and runtime software, which provides an OpenCL interface for the host. The compiler has been modified to support the generation of kernel binaries targeting the extended RISC-V ISA used by Vortex. The PoCL driver does however not support online compilation. Thus all kernels have to be compiled offline and loaded as binaries. This means that variables can not be passed from the program to the kernel before compilation. Because of this, the buffer sizes of some of the benchmarks had to be hard-coded, instead of changing with the input of the benchmark. This makes the benchmarks less flexible.

The *hybridsort* and *myocyte* benchmarks from Rodinia have kernels which require specific functions. This includes `atomic_add`, `expd`, *powdd* and `sqrtd`. These functions were missing from the PoCL compiler. These benchmarks could therefore not be included in Vortex' benchmark suite. After further inspections, it became apparent that none of the atomic functions are included by the PoCL compiler.

### 4.2.2 Memory Allocation

While porting Rodinia benchmarks to Vortex, I observed that the OpenCL functions `clEnqueueNDKernels`, `clEnqueueCopyBuffer` and *clCreateBuffer* sometimes aborted or resulted in segmentation faults. Looking into the cause of the crashes, I found that `clEnqueueNDKernels` caused a segmentation fault when the kernel contained local parameters. It seems like the driver is unable to dynamically allocate local memory for the TBs using `clSetKernelArg`. To solve this, I removed the local parameters of the kernels, and instead defined local buffers in the kernel code. By doing this, the driver does not have to allocate the buffers dynamically. As the size of the local buffers is now defined at compile time of the kernels, I had to hard-code the buffer sizes according with the selected input of the benchmarks.

The `clCreateBuffer` function creates a buffer on the GPU to store data. If the `GL_MEM_ALLOC_HOST_PTR` flag of `clCreateBuffer` is set, the allocated memory should be accessible by the host. For Vortex it seems like using this flag results in a segmentation fault. To resolve this, I instead created the buffer without the flag, and used `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` to access the memory whenever needed by the host. This issue is similar to the one above, as it seems to be caused by issues regarding the permission to dynamically allocate memory in specific locations.

The last OpenCL function I encountered issues with was `clEnqueueCopyBuffer`. `clEnqueueCopyBuffer` copies data from one buffer on the GPU to another. I is unclear why this is causing a crash, but it could be easily resolved by reading the data from the device to the host using `clEnqueueReadBuffer`, before writing it back to the destination buffer using `clEnqueueWriteBuffer`. The workarounds described in this using the problematic OpenCL features should not affect the measured performance, as measurements are made only during kernel execution.

**Table 4.1:** Rodinia benchmarks added to Vortex

| **Applications** | **Domains** |
|---|---|
| B+ Tree | Graph Traversal |
| Back propagation | Pattern Recognition |
| Breadth-First Search | Graph Algorithms |
| CFD Solver | Fluid Dynamics |
| Gaussian elimination | Linear Algebra |
| GPUDWT | Image/Video Compression |
| Heart Wall | Medical Imaging |
| HotSpot | Physics Simulation |
| HotSpot3D | Physics Simulation |
| Kmeans | Data Mining |
| LavaMD2 | Molecular Dynamics |
| LU Decomposition | Linear Algebra |
| Needleman-Wunsch | Bioinformatics |
| Particle Filter | Medical Imaging |
| SRAD | Image Processing |
| Streamcluster | Data Mining |

### 4.2.3 Selecting Work Sizes

Table 4.1 shows the list of Rodinia benchmarks which can execute on Vortex after performing the adaptations from Section 4.2.1 and 4.2.2. However, the benchmarks are not scheduled efficiently, by Vortex' static TB scheduler. The scheduler struggles to divide the work among Vortex' SMs. For most benchmarks only a small subset of the SMs are scheduled any work. Rekdal [8] encountered the same issue with some of the existing Vortex benchmarks. The solution was to reduce the local work size when enqueuing the kernels. This reduces the number of threads per TB, resulting in more TBs, which can be scheduled to more SMs. This method worked to some degree for most benchmarks, except *heart wall*, which I was unable to improve upon.

## 4.3 Fast-Forward, Warm-Up and Early-Exit

Most of the added Rodinia benchmarks require substantially more time to complete execution than the already included benchmarks. When using software simulation, it becomes infeasible to simulate the entire benchmarks, due to long simulation times. One way of solving this would be to reduce the input sizes, however, this would affect the TB scheduler and cache behaviour. A more common approach for reducing simulation time is using *fast-forwarding* and *warm-up* [38]. Fast-forwarding is to run a less accurate simulation to a given point in execution, and then continue the real simulation from there. This skips the initialization, and

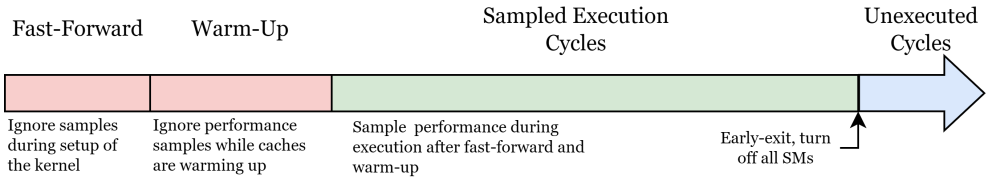



**Figure 4.2:** Timeline explaining fast-forward, warm-up and early-exit.

samples only the part of the code most representative of the whole benchmark. After fast-forwarding using a less accurate simulator, branch predictors and caches will not be in the same state as if an accurate simulation was used. A solution is to then run the accurate simulation until the cache hit rate stabilizes, before collecting performance metrics. This is called warm-up and removes the cold-start bias.

There is no emulator or less accurate simulator for Vortex, thus fast-forwarding and warm-up will have limited versatility. To further reduce simulation time, I introduce *early-exit* to Vortex. Early-exit terminates the benchmark after a given number of cycles. As the number of simulated cycles scales linearly with the simulation time, early-exit allows me to easily regulate the simulation time. When implementing early-exit, it becomes more important to skip the startup, i.e. using fast-forward and warm-up, as it becomes a larger portion of the execution time. Figure 4.2 illustrates how fast-forward, warm-up and early-exit affect which execution cycles are sampled.

Fast-forward and warm-up are implemented by setting the *perf-lock* register, described in Section 4.1, high for a given number of cycles at the beginning of the simulation. Thus no performance data will be collected during the startup. Caches and other components can also be warmed during this period. Note that this is only performed for the first kernel in a benchmark. If a benchmark executes multiple kernels before exiting, the startup cycles become representative of the kernel's performance. Early-exit is implemented by deactivating all the SMs after a given number of cycles. This indicates to the driver that Vortex has completed its execution.

This method of implementing early-exit is somewhat problematic. When terminating early, the result of the kernels will not be correct. Some of the benchmarks are affected by the results returned by the kernel execution. *Streamcluster* does for example execute the kernel until the result is not getting better. If the result is wrong due to early-exit, it might result in an infinite loop of starting and exiting the kernel. To resolve this, I had to manually add safeguards for *streamcluster* and *kmeans*, setting a maximum number of iterations.

# Chapter 5

# CPI Stacks for Vortex

Cycles per instruction (CPI) stacks are a breakdown of the execution cycles into a set of classes. The breakdown aids in identifying potential bottlenecks in the architecture. In my project thesis [1], I implemented a register transfer level (RTL) solution to create CPI stacks for Vortex (CSV). My solution utilized a classification scheme based on GSI [39] which will be described in the next section. Section 5.2 will then briefly describe how my implementation of CSV differs from GSI, and Section 5.3 will describe how I improved CSV in this thesis.

## 5.1 GPU Stall Inspector

The GPU stall inspector (GSI) [39] is a stall attribution tool that enables detailed classification of memory stalls. They also present a set of classes for instruction stalls, which is also used by GCoM [40]. GSI's classification is done in two separate steps. First, if no warps are issued, a stall type is attributed to each warp in the issue stage. The attribution is done based on the cause most strongly preventing execution, i.e. the stall cause most likely to remain in the next cycle. The details regarding the priority can be found in the GSI paper [39]. Once each warp is classified, the issue cycle is classified based on the inverse priority[1] of the stalled warps, i.e. the cause of the warp which is least likely to continue stalling in the next cycle.

Following is a list describing the classes used by GSI:

- **Base**: If any of the warps are issued, the cycle is a base cycle.
- **Idle**: The warp is not active, indicating that the kernel is not fully utilizing the GPU, because of poor load balancing or because there is not enough work.
- **Control stall**: The warp instruction supplied by the instruction buffer is not the next instruction to be executed by the warp. This might be due to a high degree of divergence in the kernel.

[1]The priority is not exactly inversed, as memory and synchronization stalls are prioritized over compute stalls in both steps.

- **Synchronization stall**: The warp is blocked due to a barrier, to synchronize with other warps.
- **Memory data stall**: The warp cannot issue because the operands are dependent on the result of a pending load.
- **Memory structural stall**: The warp is a memory instruction requiring the LSU, but it cannot be issued because the LSU is not ready.
- **Compute data stall**: The warp cannot issue because the operands are dependent on the result of a pending compute instruction. Compute instruction refers to every non-memory instruction.
- **Compute structural stall**: The warp is a compute instruction, but the required functional unit is not ready.

## 5.2 CSV Overview

This section describes how I adapted the GSI classification scheme in my project thesis [1], to create CSV. As the baseline version of Vortex only selects one warp to attempt to issue, the two step scheme of GSI was not required. Instead, CSV only sampled the stall cause of the warp selected by Vortex' issue scheduler. Every cycle, each SM attributes its cycle to the class corresponding with its stall cause. Later, the data from each SM is read and accumulated to create the results for the entire GPU.

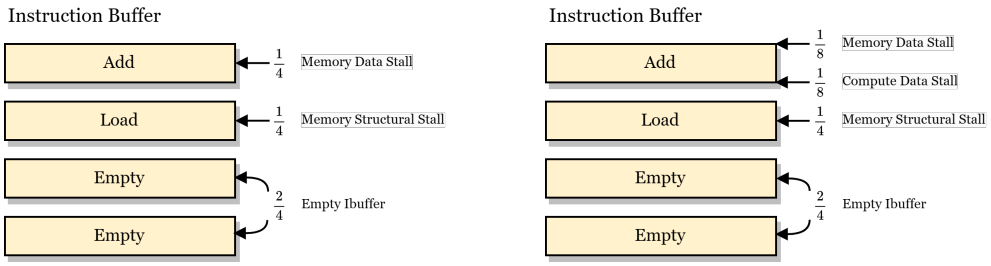
When a data stall occurs, the functional unit or combination of functional units, which reserved the operand register(s), are used to track the type of data stall. If both of the operands are reserved, half of the cycle is attributed to each of the sources. For example, if a warp is waiting for results from both the ALU and the LSU, one half of the cycle is attributed as a *memory data* stall and the other half is attributed as a *memory structural* stall.

To simplify the implementation, I grouped *control* and *synchronization* stalls into one class: *sync & control*. *Sync & control* stalls occur if there are no warps in the instruction buffer, and there are no ready warps in fetch's warp scheduler. Additionally, I included a class for *empty ibuffer* which occurs if there are no warps in the instruction buffer but there are ready warps in the warp scheduler. This could occur if there is significant latency between fetching an instruction and it being available in the instruction buffer.

An issue with sampling *idle* stalls is that the SMs terminate at different times. The SMs will therefore not be able to track idle cycles after completing, while other SMs continue execution. The idle cycles can thus not be read by the SMs themselves. To solve this, the number of idle cycles can instead be calculated as

$$C^i_{idle} = \max(C^i_{active}) - C^i_{active} \tag{5.1}$$

where $\max(C_{active})$ represents the number of cycles used to run the program, and $C^i_{active}$ represents the number of active cycles for the $i$th SM.




**(a)** The add instruction is waiting for the result of a load. The load is waiting for the LSU to be ready while the two remaining buffers are empty

**(b)** The add instruction is waiting for the result of a load and a computation. The load is waiting for the LSU to be ready while the two remaining buffers are empty

**Figure 5.1:** Examples of TIP-inspired stall classification with 4 warps in an SM.

While the *data* and *structural* stalls described by GSI can occur individually, an instruction can also be stalled by both at the same time. This is why I included a *data & structural* stall class. If a warp is waiting for both data and the functional unit to be ready, it would thus be attributed to a separate class rather than having to prioritize one class over another.

## 5.3 Improving CSV

As I see it, GSI and the version of CSV implemented in my project thesis has a key weakness: it does not describe the entire state of the issue stage. For example, if multiple warps are available, and all are memory structural stalls except one compute structural stalled warp, the cycle will be classified as compute structural stall. This may mislead us into thinking that the throughput of the compute functional units is the bottleneck. Resolving the 'issue' would just reveal the cause of all the other warps. Another example would be a case where the GPU frontend does not fetch enough instructions. This would result in many data stalls, due to having a small selection of warps in the issue stage. However, the real problem might be control or synchronization stalls. As the classification is not proportional to the number of each stall cause, the solutions will not lead to a proportional change. It will also be hard to say if the proposed solutions just revealed existing stalls or actually caused new behaviour.

To resolve this weakness I took inspiration from TIP [41] to get a better overview of why warps stall. When a stall occurs, all warps in the SM attribute the cause(s) of their stall. Then $1/N$th of the cycle is attributed to the stall class of each warp, where $N$ represents the number of warps per SM. Figure 5.1a shows an example of how the cycles may be attributed. If a warp has multiple causes for stalling, i.e. a combination of data stalls and/or a structural stall, the $1/N$th cycle is further divided evenly among the stalls, as illustrated in Figure 5.1b. By doing

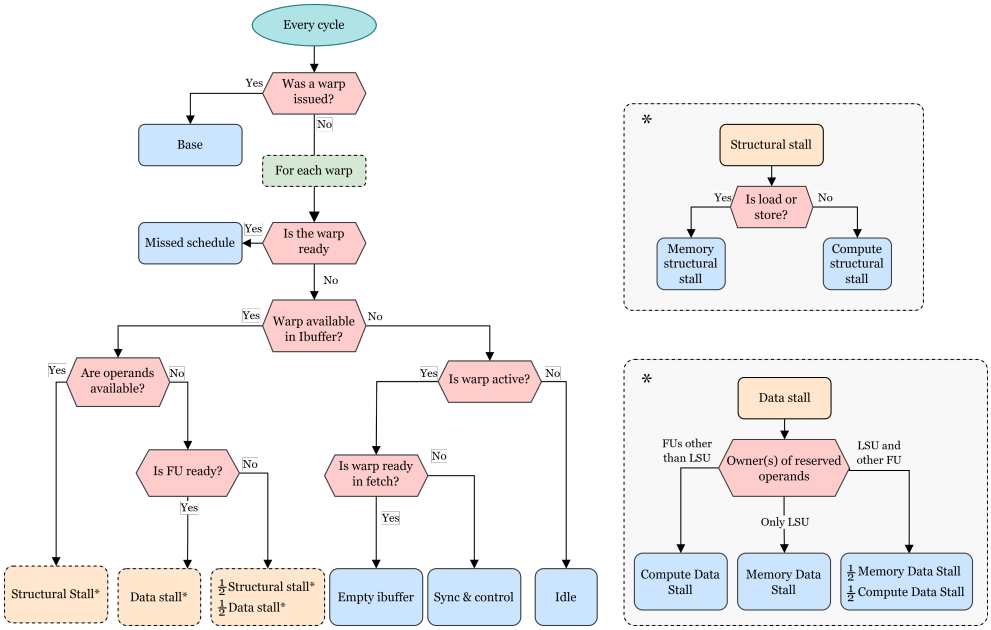



**Figure 5.2:** Flowchart for CSV's cycle attribution. Data and structural stalls are divided if they occur together. Data stalls may also be further divided if an instruction is waiting for results from both memory and compute.

this, the *data & structural* can be removed, as the causes will be attributed proportionally. By attributing the cycles in proportion to the occurrence of the stalls, we get a better overview of the entire issue stage and why no warp can be issued.

With my new attribution scheme, I have to include an additional stall class: *missed schedule*. This is required as the baseline instruction scheduler might stall due to selecting a stalling warp, while a ready warp is available. In this case, the ready warps are attributed as *missed schedule*. Figure 5.2 shows a flowchart for how the cycles are attributed.

# Chapter 6

# Experimental Setup

## 6.1 Vortex Configuration

Section 2.3 explained how FPGA-accelerated simulations have to correctly scale memory bandwidth and latency to obtain representative results. As work is still being done at NTNU to integrate Vortex into chipyard [31], I have to use software simulations. The Vortex project has a built-in script for running benchmarks. The script allows for setting several parameters and configurations for the benchmarks, architecture and simulator. Vortex' simulation stack, includes four simulation environments shown in Figure 6.1. In this thesis, I use *VLSIM*. VLSIM use Verilator [36] to simulate the full RTL design and implements the accelerator functional unit (AFU) interface in software. Memory is also simulated in software using Ramulator [42]. The configurations I used are listed in Table 6.1, while the benchmarks are listed in Table 6.2. The number of warps and threads per SM is based on Vortex' default configuration. In my project thesis [1], I tested a range of configurations with different numbers of SMs and found that all benchmarks worked as intended for up to 32 SMs. For 64 SMs, some of the benchmarks returned erroneous results. Because of this, I elected to continue evaluating Vortex with 32 SMs.

## 6.2 Benchmarks

In my project thesis [1], I found that the benchmarks included by Vortex required a much larger input size than the default to give reasonable performance results. That is, having enough work to have realistic memory usage and to activate all the SMs. The adjusted input sizes, as well as the inputs for the new benchmarks, are displayed in Table 6.2.

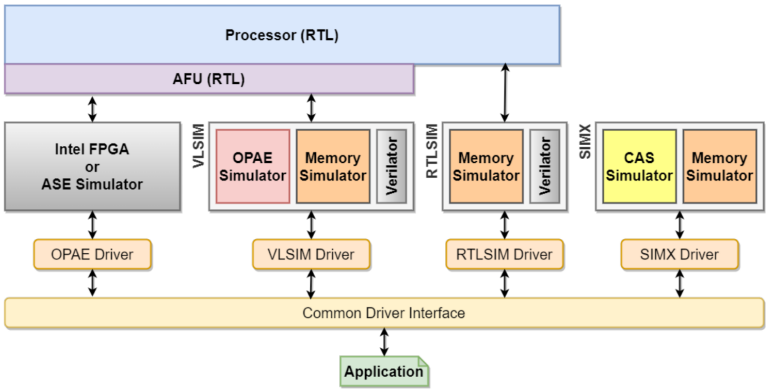


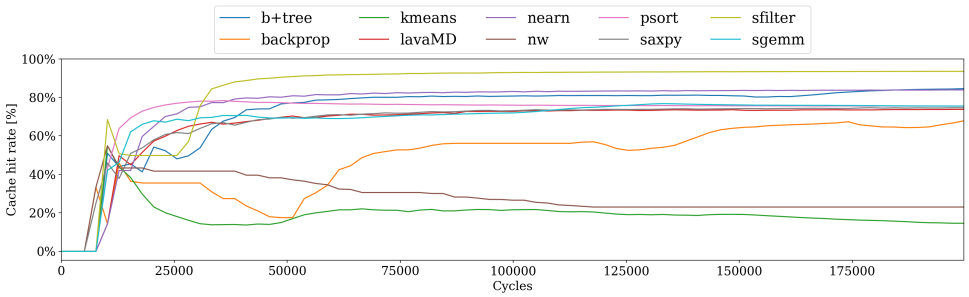

**Figure 6.1:** Vortex simulation stack reproduced from [7].

**Table 6.1:** Configurations for the Vortex architecture.

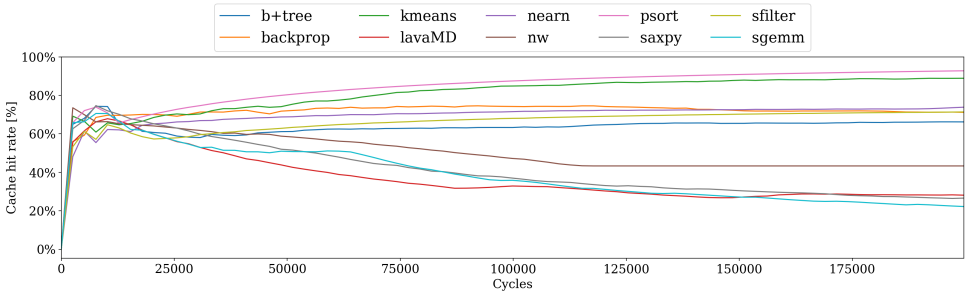
| **Vortex Configuration** | |
|---|---|
| GPU | 32 cores, 1.2GHz, 16 threads/SM,<br>4 threads/warp, 4 warps/SM |
| Clustering | 8 cores/cluster (4 clusters) |
| GPU L1 Cache | 16KiB per SM, direct mapped,<br>16B blocks, 4B words |
| GPU L2 Cache | 128KiB per cluster, direct mapped<br>64B blocks |
| GPU L3 Cache | No L3 cache |
| DDR4 | DDR4 2400R (1200 MHz), 19.2GB/s,<br>4Gbx8, 1 channel, 1 rank/channel<br>$t_{CL} = 16$, $t_{RCD} = 16$, $t_{RP} = 16$ |
| NoC | Hierarchical tree structure |

**Table 6.2:** Overview of benchmarks and the adjusted input sizes. The first 6 benchmarks are the benchmarks not ported by me

| **Benchmark** | **Short Name** | **Default Input Size** | **Adjusted Input Sizes** |
|---|---|---|---|
| Vector Addition | Vecadd | 64 | 32768 |
| General Matrix Multiply | Sgemm | 32x32 Matrix | 256x256 Matrix |
| Matrix Filter (3x3 kernel) | Sfilter | 16 | 1024 |
| Sorting | Psort | 16 | 8192 |
| A times X plus Y | Saxpy | 16 | 262144 ($2^{18}$) |
| Nearest Neighbour Search | Nearn | 40k Records | - |
| B+ Tree Graph Traversal | B+tree | 1M Elements | 10K Elements |
| Back propogation | Backprop | 65536 | - |
| Breadth-First Search | BFS | 4096 Nodes | 65536 Nodes |
| CFD Solver | CFD | fvcorr.domn.097K | missile.domn.0.2M |
| Gaussian elimination | Gaussian | 16x16 Matrix | 512x512 Matrix |
| GPUDWT | DWT2D | 192x192 Bitmap | - |
| Heart Wall | HW | 20 Frames | - |
| HotSpot | Hotspot | 512 2 2 | 512 2 2 |
| HotSpot3D | 3D | 512 8 100 | 512 8 4 |
| Kmeans | Kmeans | 100 Points,<br>100 Features | 2048 Points<br>128 Features |
| LavaMD2 | LavaMD | -boxes1d 10 | -boxes1d 16 |
| LU Decomposition | LUD | 1024x1024 Matrix | - |
| Needleman-Wunsch | NW | 2048x2048 Matrix<br>10 Penalty | - |
| Particle Filter | PF | -x 128 -y 128<br>-z 10 -np 10000 | -x 512 -y 512<br>-z 10 -np 300 |
| SRAD | SRAD | 502x458 Image | 251x229 Image |
| Streamcluster | SC | 65536 Points | - |




**(a)** L1 dcache hit rate during startup




**(b)** L2 dcache hit rate during startup

**Figure 6.2:** Dcache hit rates over time for a subset of the benchmarks during startup.

To keep the simulation time reasonable, I implemented fast-forward, warm-up and early-exit, as described in Section 4.3. To do this, I have to find how many cycles are required for the caches to become warm and get past the startup. Figure 6.2a and 6.2b respectively show the cache hit rate during startup of the L1 and L2 caches when running a subset of the benchmarks. After 30k cycles, the cache hit rates are stabilizing for both the L1 and L2 caches. While the L1 hit rate for *backprop* is fluctuating, it is probably due to its access pattern. To have some extra margins, I chose to use 50k fast-forward and warm-up cycles. For early-exit, I let the benchmarks run for up to 10M cycles before terminating.

## 6.3 IDUN Cluster

Running the simulations required a substantial amount of time and memory. The total computing time required to run all the simulations also scales with the number of benchmarks. Adding 16 new benchmarks thus increase the total simulation time significantly. Because of this, all the simulations were run on the IDUN Cluster [43] at NTNU. Using IDUN allowed me to run all benchmarks in parallel, thus saving a lot of time, as most benchmarks required many hours to complete.

The installation of Vortex had to be modified due to missing permissions to write to some of the install locations. It also followed that the locations of these dependencies had to be changed in the Vortex makefiles.

Unfortunately, there were some discrepancies when running some of the benchmarks on IDUN. The results of *LUD*, *SRAD* were drastically different when simulated on my personal machine and on IDUN. I was also unable to run *particle filter* on IDUN without crashing, although it executed without errors on my personal machine. I was unable to identify any specific reasons for these discrepancies, but it could be due to differences in software or library versions. Because of this, these benchmarks are not included in the results in Chapter 7.

# Chapter 7

# Results and Evaluation

In this chapter, I will present the results and evaluation of the experimental setup from Chapter 6. In this evaluation, I will consider the following configurations:

- **Vortex**[1] is the baseline version of Vortex as presented at MICRO'21 [7].
- **FGTO**[2] is the configuration implementing ready scheduling, and all the frontend changes, i.e. BPR, NSS and stall-prediction. The GTO scheduling algorithm is used for both the warp and instruction scheduler.
- **FRRR**[2] is the configuration implementing ready scheduling, and all the frontend changes, i.e. BPR, NSS and stall-prediction. The LRR scheduling algorithm is used for both the warp and instruction scheduler.

## 7.1 CPI Stack Overview

Figure 7.1 shows the CPI stacks for all of the benchmarks, and the average CPI, normalized to the baseline version. The changes to Vortex have varying effects on the performance of each benchmark. In this context, performance is characterized by CPI, where lower CPI equates to higher performance. On average the *FGTO* and *FRRR* configurations respectively have a 5.03% and 5.48% reduction in CPI compared to *Vortex*. There are however large deviations from this average. The *HW* benchmark has an almost a 40% reduction in CPI for both *FGTO* and *FRRR*, while *CFD* see over 20% increase in CPI for the *FGTO* configuration. Most of the benchmarks see a drastic shift in the causes of the stalls. This shift does however not necessarily reflect in an increase or decrease in CPI. In the following sections, I will present subsets of the results shown in Figure 7.1, to explain the effects of the implemented changes.

[1] https://github.com/EECS-NTNU/vortex-ntnu/tree/ntnu_main
[2] https://github.com/EECS-NTNU/vortex-ntnu/tree/ntnu_main_larsmaur

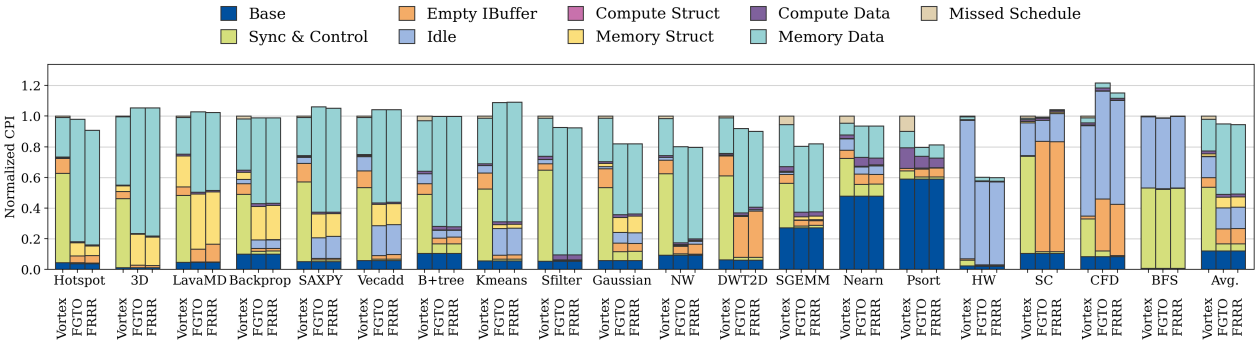



**Figure 7.1:** Normalized CPI stacks for the Vortex, and the new *GTO* and *RRR* configurations

## 7.2 Reduction of Control Stalls

Figure 7.2 shows the normalized CPI attributed to frontend stalls and base. It is clear that the frontend is the dominating cause for stalling in the baseline version. On average, 50% of the CPI is attributed to *sync & control* or *empty ibuffer* stalls. *Sync & control* stalls are however most prevalent, representing over 40% of the CPI. NSS, and stall-prediction reduces the number of control stalls by removing unnecessary stalls. Because of this, warps are now only blocked after fetching instructions capable of changing the control flow or thread mask. The new icache-stage, enables Vortex to perform multiple concurrent instruction fetches, which increases the fetch bandwidth. After implementing the new frontend, less than 5% of the average CPI is *sync & control* stalls, and the average CPI attributed to the frontend is reduced by 71%. This shows that most instructions were unnecessarily blocked by the frontend.

For *sfilter*, *FGTO* and *FRRR* reduces the number of frontend stalls from 61% to less than 0.01% of the *Vortex* CPI. This is because *sfilter* does not have any branch instructions in its kernel. The warp scheduler can thus continue to fetch instructions without stalling. The few occurring frontend stalls are during the end of the benchmark. As the instructions are never blocked in the warp scheduler, the GTO warp scheduler would never switch which TB it schedules from. However, BPR informs the warp scheduler when the ibuffer is full, blocking the warp from being scheduled. BPR is thus making the GTO warp scheduler switch if the backend is stalling, making it schedule more fairly. This becomes apparent as there are no idle stalls caused by TBs completing before others. This demonstrates how the new frontend is capable of removing all unnecessary control stalls.

While the new frontend has a large impact on most benchmarks, *BFS* seems unable to utilize NSS. The number of *sync & control* stalls does not change between *Vortex*, *FGTO* and *FRRR*. A large proportion of *BFS*' kernel are instructions pertain-

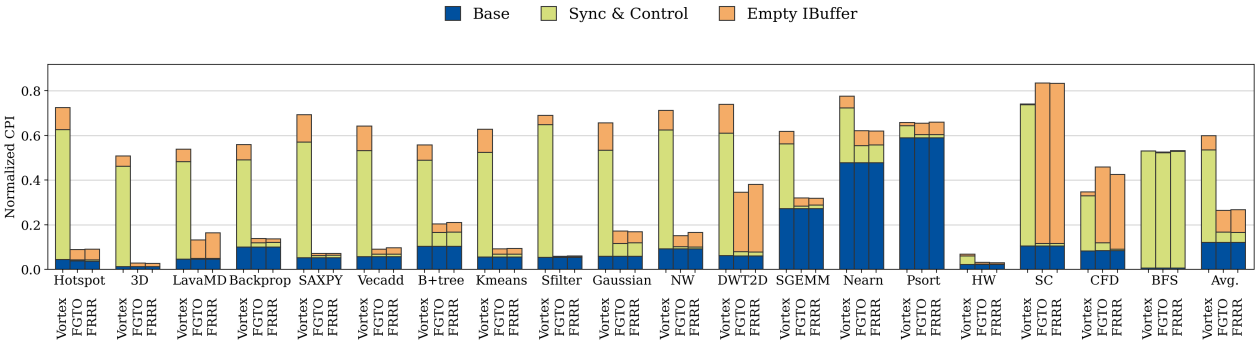



**Figure 7.2:** Normalized CPI attributed to the base cycles and stalls caused by the frontend.

ing to control flow and handling of divergence. Because of this, *BFS* has a considerable number of required frontend stalls, which cannot be improved by NSS. Stall prediction ensures that these control stalls are not mispredicted. If the control-flow instructions were not stalled, it would waste many cycles having to flush the frontend. Stall-prediction is thus ensuring that the new frontend performs at least as well as the baseline in terms of control stalls.

In the baseline configuration, all of the benchmarks have $5-10\%$ of the CPI attributed to *empty ibuffer*. These occur when a TB has no warps available in the instruction buffer, and it is not currently blocked in the warp scheduler. Some of these stall cycles are caused by the latency between the warps being un-stalled, and then being fetched, decoded and sent to the ibuffer. For all the benchmarks, excluding *lavaMD, DWT2D, SC* and *CFD*, the number of *empty-ibuffer* stalls are reduced. As the new frontend is able to cut down on the number of frontend stalls, the number of un-stalls is also reduced, which in turn reduces *empty ibuffer* stalls.

Figure 7.1 shows that *SC* has almost no stalls related to the backend for any of the configurations. That is, when a warp arrives in the ibuffer, it is issued within a few cycles. For *SC*, only the *memset* kernel is executed, because of early-exit. The *memset* kernel has almost no data dependencies. Thus the backend is almost never stalling, and the ibuffer is thus being emptied faster than it is filled. The *memset* kernel has a short loop containing a memory write. The loop does require the frontend to stall, due to control flow. There are however no *sync & control* stalls shown for the *FGTO* and *FRRR* configurations, only *empty ibuffer* stalls. This is because while the warp is stalled in the frontend, all previously fetched warps are issued, which hides the control stalls. When the control stall is released, all of the ibuffers are empty, causing *empty ibuffer* stalls. Yet, there are more *empty ibuffer* stalls when using the new frontend than *sync & control* stalls in the baseline version. This is likely because the icache-stage requires an additional cycle to re-

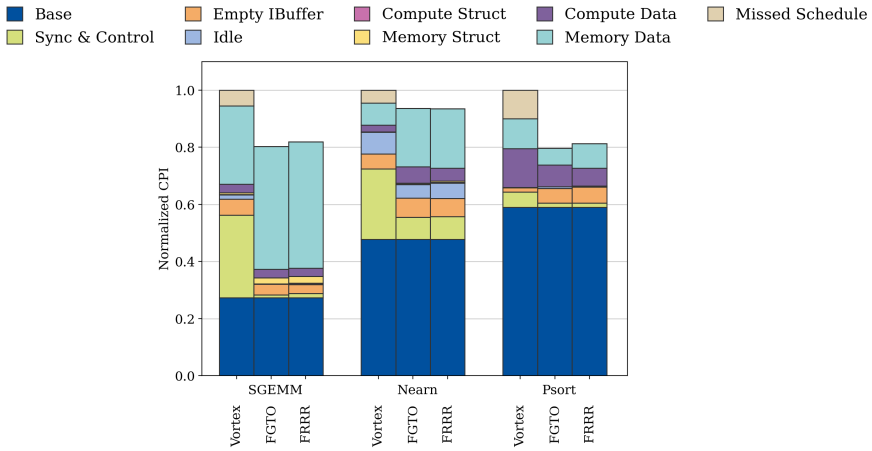



**Figure 7.3:** Normalized CPI for benchmarks with most *missed schedule* stalls.

order the icache responses. This results in a somewhat higher latency between a warp being fetched and it arriving in the ibuffer. A similar effect can be seen for other benchmarks such as *srad* and *CFD*. This problem can probably be solved by having a wider frontend, allowing for scheduling more than one instruction per cycle, or having enough warps per SM to hide the frontend latency.

## 7.3 Utilization of Functional Units

Figure 7.3 shows the CPI stacks of the benchmarks with the largest proportion of *missed schedule* stalls. For *psort*, the number of stalls is nearly halved for both *FGTO* and *FRRR*, resulting in a 20% decrease in CPI compared to the *Vortex* configuration. *Psort* has a low number of frontend stalls, which does not change after implementing the new frontend, as nearly all *sync & control* stalls are replaced by *empty ibuffer* stalls. *Psort* is thus seeing limited improvements from the frontend changes. The implementation of ready scheduling in *FGTO* and *FRRR* is able to remove all the *missed schedule* stalls. In the *Vortex* configuration, *missed schedule* stalls contribute to about 10% of the CPI, while for *FGTO* the CPI is decreased by the double (20%). This is because other than frontend and *missed schedule*, *psort* is stalled by compute data dependencies. These stalls can be hidden by the increased number of issued instructions, as they have low latency.

The *sgemm* and *nearn* benchmarks also have a significant number of *missed schedule* stalls. In addition to the improvements obtained by ready scheduling, these benchmarks see a great reduction in frontend stalls. The increased throughput of the frontend makes more warps available in the issue stage, increasing the effect of ready scheduling. Because of the significant decrease in frontend stalls, *sgemm* is also able to obtain a 20% reduction in CPI. This is similar to *psort*, but with a smaller proportion of *missed schedule* stalls in the *Vortex* configuration. The CPI of *nearn* is not reduced to the same degree as *sgemm* and *psort*. This is likely

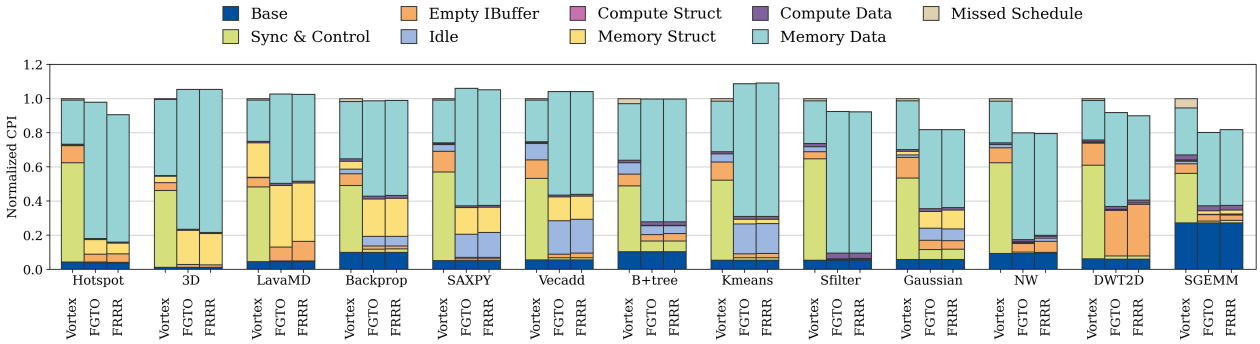



**Figure 7.4:** Normalized CPI for benchmarks where memory stalls are revealed using *FGTO* and *FRRR*

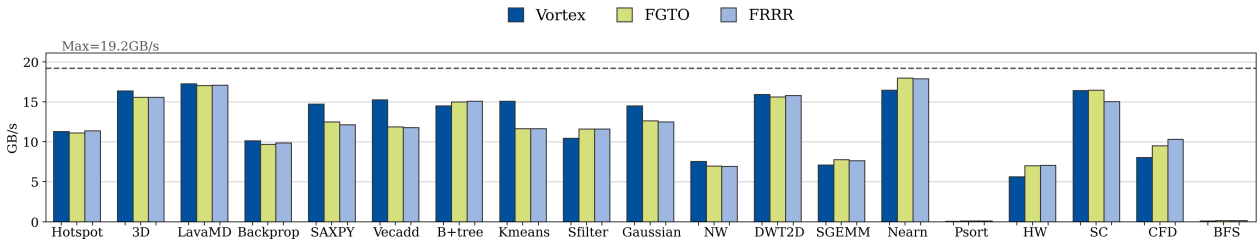
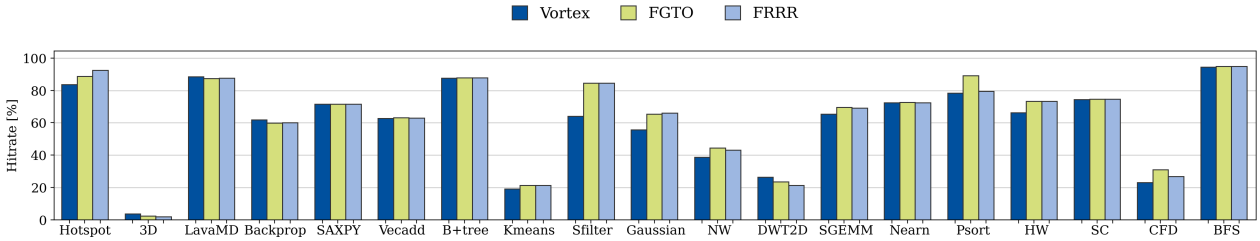



**Figure 7.5:** Average bandwidth usage between the L2 cache and DDR4 memory

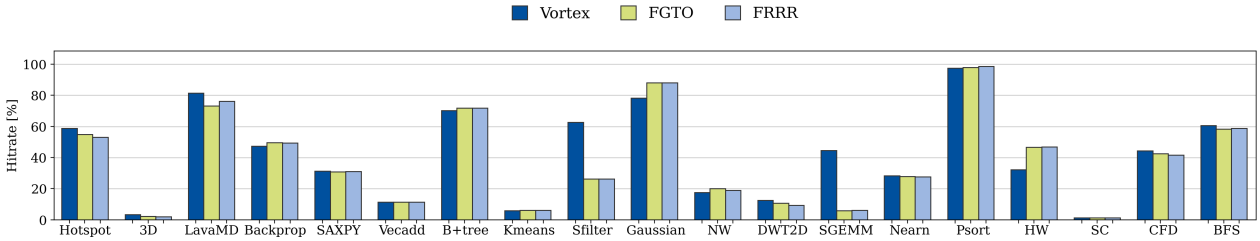
caused by a combination of multiple factors. The number of frontend stalls is only halved, thus revealing fewer additional warps in the ibuffer than *sgemm*. *Nearn* also have less *missed schedule* stalls than *psort* and more long-latency *memory data* stalls than *psort*. Because of this, there are fewer stalls which can be hidden by ready scheduling.

## 7.4 Memory Stalls

Figure 7.4 shows the normalized CPI stacks for the benchmarks where *FGTO* and *FRRR* reveal memory stalls. For all of these benchmarks, a significant number of *memory data* stalls are revealed. Additionally for *hotspot, 3D, lavaMD, saxpy, vecadd* and *gaussian*, *memory structural* stalls are also uncovered. As there is an increase in *memory structural* stalls it would be rational to assume an increase in memory bandwidth usage. However, the memory bandwidth usage illustrated in Figure 7.5, shows the opposite. For the benchmarks where the changes expose *memory structural* stalls, the bandwidth is either decreased or unchanged. Figure 7.6a and 7.6b show that the dcache hitrates are similar for *Vortex*, *FGTO* and *FRRR*. This indicates that there are other causes for the *memory structural* stalls.




**(a)** L1 dcache hitrate




**(b)** L2 dcache hitrate

**Figure 7.6:** Average dcache hitrate

*Backprop* and *lavaMD* both have a large number of *memory structural* stalls, but very different bandwidth utilization. *LavaMD* use $17GB/s$ of memory bandwidth on average while *backprop* use $10GB/s$, which is only about half of the available bandwidth. For *lavaMD*, the available bandwidth is likely the main cause of *memory structural* stalls, as its average usage is close to the available bandwidth. This is not the case for *backprop*. Unlike *lavaMD*, *backprop* uses a lot of memory barriers. When a memory barrier is issued to the LSU, all in-flight memory instructions have to complete before the LSU will be ready. *Memory structural* stalls are therefore occurring while these barriers are resolving. Because of this, the average memory bandwidth usage can be low while still producing *memory structural* stalls. This is also why *hotspot* has an increase in *memory structural* stalls. It is however a smaller increase, as it is dependant on the latency of the requests and the frequency of the barrier instructions.

For all of the benchmarks in Figure 7.4, a large number of *memory data* stalls are revealed when the number of frontend stalls is reduced by *FGTO* and *FRRR*. Because warps are waiting for the results of memory requests, these stalls are caused by memory latency. It is most likely that these memory requests are already in-flight, and thus the increased frontend throughput just exposes the stall sooner. As there is such a large proportion of *memory data* stalls, it is clear that Vortex is incapable of hiding them. There is simply not enough work to do while waiting for the memory requests. The average latency for each benchmark is shown on a logarithmic scale in Figure 7.7. It is a clear trend that *FGTO* and *FRRR* is increasing the memory latency. This is likely because the memory requests are

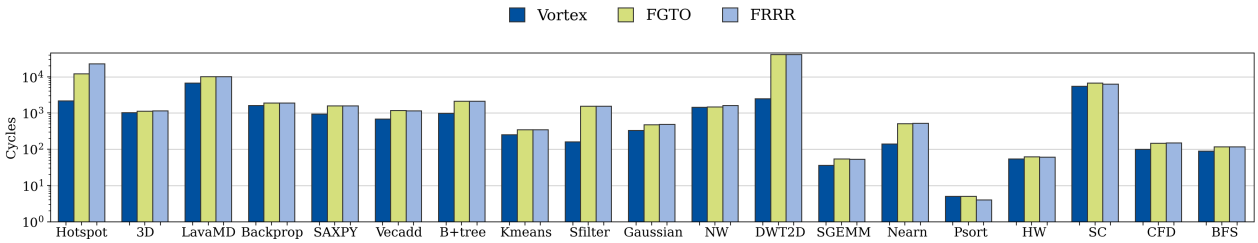



**Figure 7.7:** Average memory latency on logarithmic scale

being queued in the memory system. It is possible that *FRRR* and *FGTO* make the memory requests within a shorter time span than *Vortex*, due to the frontend throughput. This would generate uneven memory bandwidth usage, and result in queuing. This would explain why the latency is increased, while the average bandwidth does not. Section 7.6 will discuss further why *FGTO* does not perform better than *FRRR* in this situation.

Figure 7.6 shows the L1 and L2 dcache hit rates for all the benchmarks. For most of the benchmarks, the configuration does not affect the L1 cache hit rate. However, *hotspot, sfilter, gaussian* and *NW* see a significant increase in L1 hit rate for *FGTO* and *FRRR* compared to *Vortex*. A potential cause of this increase is that *Vortex*' find-first warp scheduler is causing some warps to run ahead of others, making them evict cache lines of memory shared between the TBs. This is not happening when using GTO or LRR warp scheduling as they schedule the warps more fairly.

Figure 7.8 shows the distribution of executed instructions by each of Vortex' SMs for the *3D* benchmark. It is clear that within each cluster, *SM0* is executing close to 5× more instructions than *SM7*. For each increasing index within the cluster, the SMs are executing fewer instructions. As there are no idle cycles for *3D*, this cannot be because less work is being allocated to the SMs. If the benchmark would have finished instead of exiting early, there would probably be idle cycles, as the SMs with lower indices would finish before the higher indexed SMs. The difference in the number of executed instructions is likely caused by the NoC distributing the available bandwidth unfairly. A similar effect can be observed for *lavaMD* in figure 7.9k, as both *lavaMD* and *3D* exit early and have bandwidth utilization close to the available memory bandwidth.

*SAXPY, vecadd, kmeans* and *gaussian*, see a significant decrease in bandwidth utilization. This decrease is mainly caused by an increase in idle cycles. As some of the cores idle, they stop sending memory requests, lowering the average bandwidth utilization. Section 7.5 will explain further why these benchmarks idle.

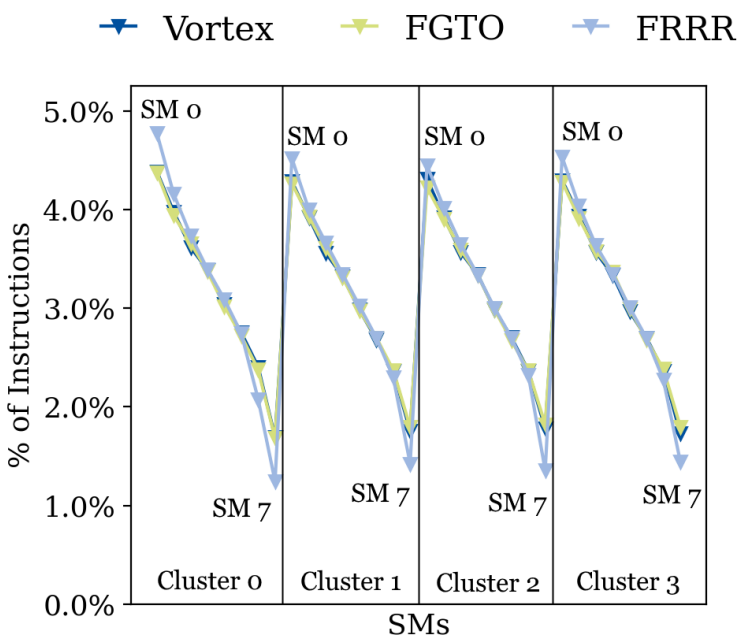
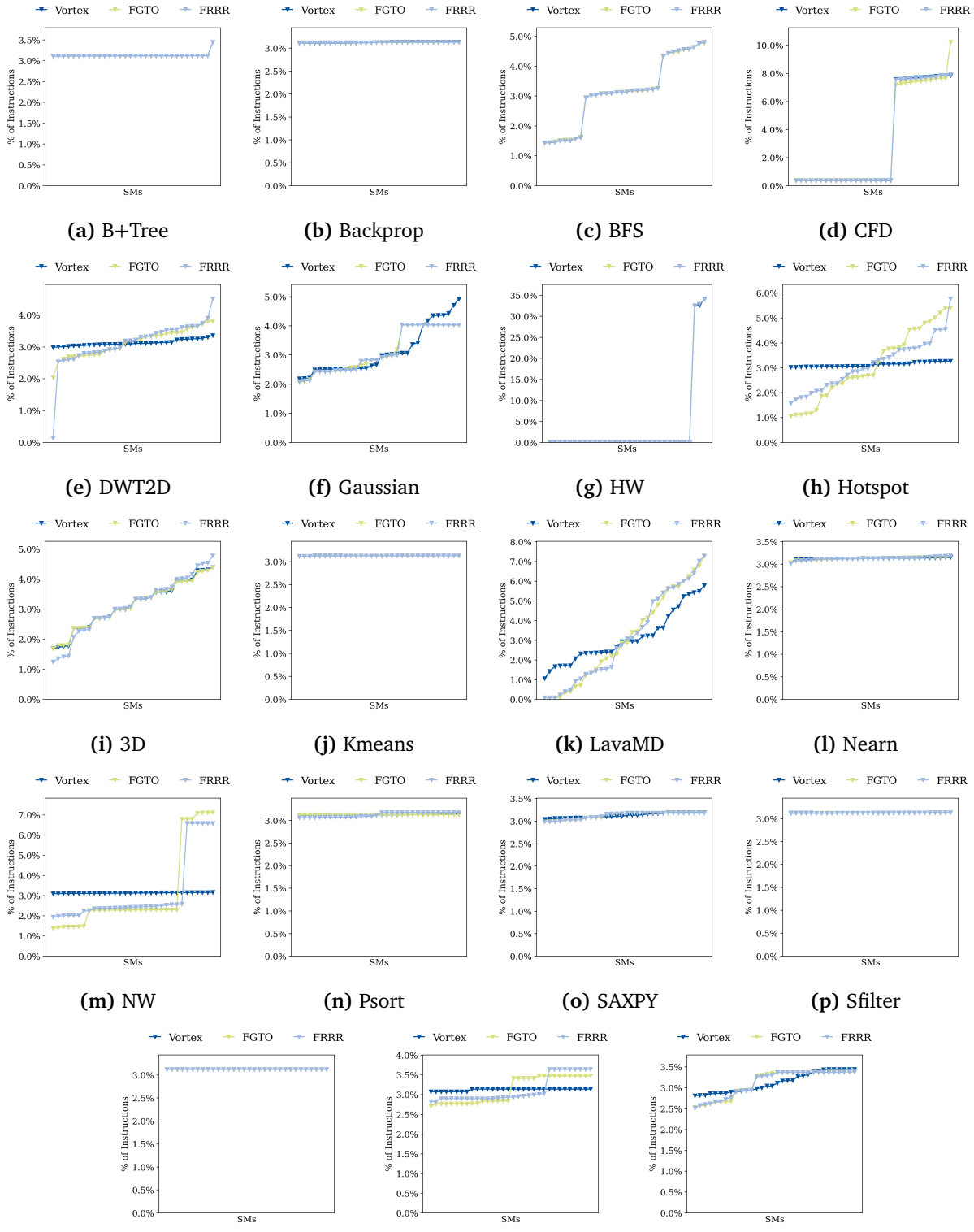



**Figure 7.8:** Distribution of executed instructions per SM and cluster for Hotspot3D *(3D)*

## 7.5 Workload Distribution

*Backprop, saxpy, vecadd, b+tree, kmeans* and *gaussian* all gain a significant number of idle cycles on *FGTO* and *FRRR*, as shown in Figure 7.10. All of these benchmarks either finish execution, or complete multiple kernels before exiting early. The effects of uneven memory bandwidth distribution can thus be observed as idle cycles. As described in Section 7.4, the NoC is distributing the available bandwidth unfairly. This makes some SMs have higher throughput than others. As the TBs are statically distributed at the beginning of the kernel, the throughput difference will cause some of the SMs to finish earlier than others. The SMs will then have to idle until the kernel has finished execution. This is why the average bandwidth becomes lower for these benchmarks after implementing the changes. The skew in throughput cannot necessarily be observed in Figure 7.9 as these benchmarks finish execution, which levels out the number of executed instructions by SMs idling.

For the benchmarks which do not complete any kernels, the skewed memory bandwidth can be observed in the distribution of executed instructions. Because the kernels do not complete, the SMs are not idling, which is why it cannot be observed as idle stalls. Looking at *hotspot* (Figure 7.9h), *DWT2D* (Figure 7.9e), *lavaMD* (Figure 7.9k) and *NW* (Figure 7.9m), the difference between the SMs executing the most and least instructions is increased for *FGTO* and *FRRR* compared




**(a)** B+Tree **(b)** Backprop **(c)** BFS **(d)** CFD

**(e)** DWT2D **(f)** Gaussian **(g)** HW **(h)** Hotspot

**(i)** 3D **(j)** Kmeans **(k)** LavaMD **(l)** Nearn

**(m)** NW **(n)** Psort **(o)** SAXPY **(p)** Sfilter

**(q)** SGEMM **(r)** SC **(s)** Vecadd

**Figure 7.9:** Distribution of executed instructions per SM for each of the benchmarks. The SMs are sorted in ascending order of their share of the executed instructions

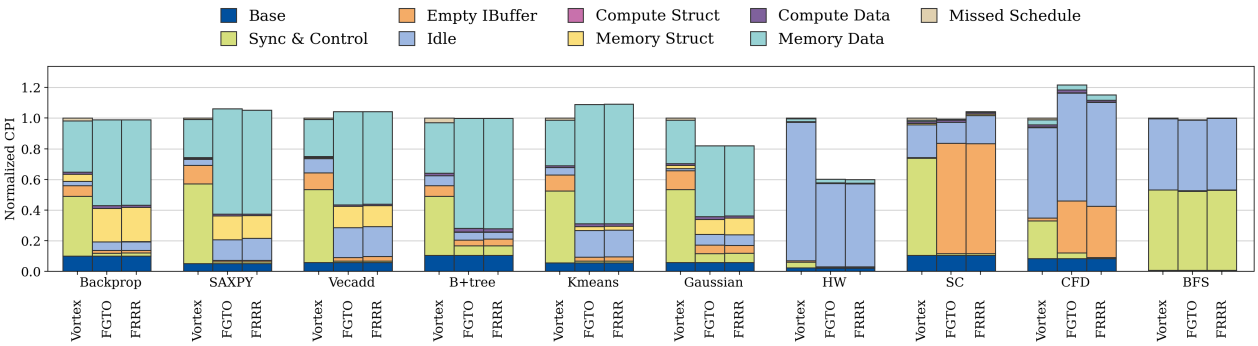



**Figure 7.10:** Normalized CPI stacks for benchmarks with idle cycles

to *Vortex*. This is because *FGTO* and *FRRR* allow the SMs to issue more instructions, but the GPU is throttled by the NoC, which is why there are only small improvements in performance.

For benchmarks such as *SC, CFD, HW* and *BFS*, the proportion of idle cycles stays mostly the same for all three configurations. These idle cycles are caused by the TB scheduler not being able to distribute workload to all of the SMs. This is apparent in Figure 7.9c, 7.9d, 7.9g and 7.9r, where most of the SMs are not executing any instructions. I attempted to solve this by altering the local and global work sizes. This did however not affect the workload distribution. A better understanding of the TB scheduler is probably required to solve this.

In Figure 7.10, the CPI of *HW* is almost halved for *FGTO* and *FRRR*. It appears as if the improvements are a result of a reduction in idle cycles. However, as shown in Figure 7.9g, the same number of SMs continue to idle. Because of low memory contention, and the increased frontend and issue bandwidth, the performance of the active SMs improve drastically. The appearance of reduced idle cycles is thus caused by an increase in executed instructions by the active SMs, which in turn reduces the number of idle cycles per instruction.

## 7.6 Warp Scheduling

The motivation for implementing new schedulers for Vortex, was to improve performance over the existing find-first and LRR algorithms. In this section, I will compare combinations of the GTO and LRR algorithms in the instruction and warp schedulers. The configurations will be described as *fetch-algorithm+issue-algorithm*. All of the configurations discussed in this section implement all the frontend improvements as well as ready scheduling.

For most of the benchmarks shown in Figure 7.11, the scheduling algorithm does not make a large difference. Looking at the average CPI, the performance seems to be somewhat worse for *GTO+LRR*, but the average CPI stays within 1.5%

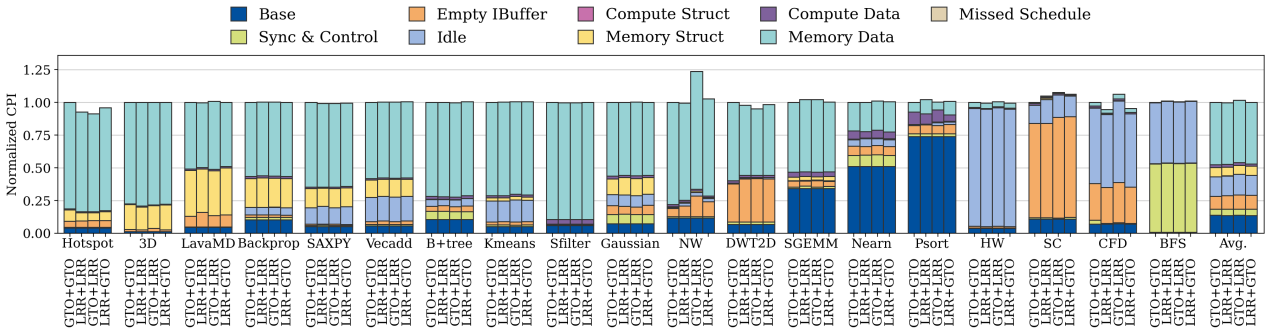



**Figure 7.11:** Comparing combinations of scheduling algorithms used in the issue stage and fetch stage.

for all the other configurations. Most of the average CPI increase for *GTO+LRR* is coming from *NW*. It is difficult to pinpoint an exact reason for this being an outlier. For *CFD*, the performance is somewhat better when using a round-robin warp scheduler. This is because it has a large number of *empty ibuffer* stalls. A round-robin warp scheduler will be able to fetch warps from different TBs, reducing the number of empty ibuffers, and thus reducing *empty ibuffer* stalls. On the other hand, a GTO warp scheduler will attempt to fetch multiple warps from a single TB, and thus only fill one ibuffer. If a warp fetched by GTO cannot be issued, it is therefore less likely to be other warps available in the instruction buffer to hide the stall.

The motivation for using GTO was to improve cache usage and reach long-latency stalls earlier. This is clearly not happening, since there is no significant difference in the number of *memory data* stalls between the configurations. There are multiple possible reasons why the GTO scheduler might not perform as expected:

- The current implementation of GTO does not differentiate between long- and short-latency stalls. If a warp is stalled due to a short-latency stall, e.g. a cache hit, its age is set to 0. Thus it will become a low-priority warp before it hits a long-latency stall. This might also lose potential cache hits, as the GTO scheduler will then begin to schedule other warps, which can evict cache-lines used by the short-latency stalled warp. Setting the age of the warp to 0 only upon detecting long-latency stalls would allow the warp to keep high priority and hit long-latency stalls sooner. This might improve cache usage and reduce the number of stall cycles due to long-latency memory requests.
- The interaction between the warp scheduler, issue scheduler and BPR might not be ideal for GTO. To get the most out of GTO scheduling, it should be able to consecutively issue as many warps from the same TB as possible. If the currently selected greedy warp ID of the two schedulers are different, i.e.

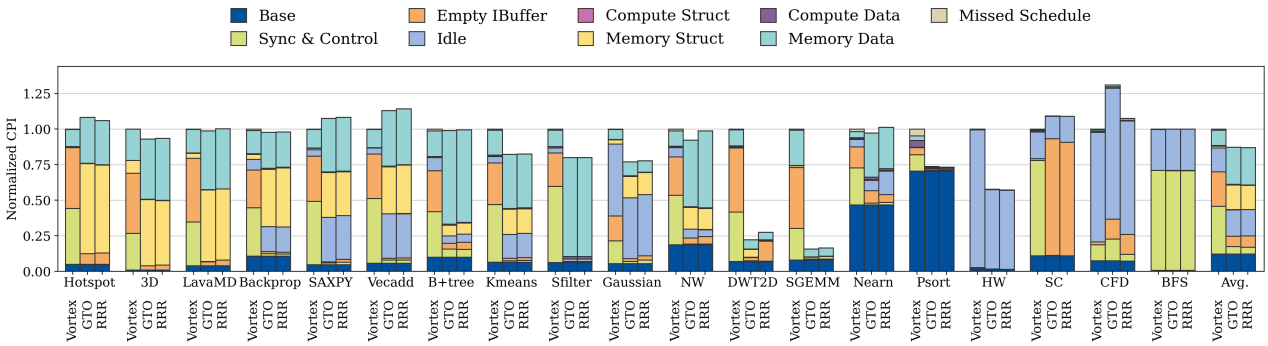



**Figure 7.12:** Normalized CPI stacks for the baseline and final versions using 8 warps per SM

they schedule warps from different TBs, the instruction buffer might empty before hitting the desired long-latency stall. BPR forces the warp scheduler to fetch warps from a different TB if the instruction buffer is full. This can prevent the warp scheduler from fetching enough instructions to hit long-latency stalls. A solution to this would be to increase the queue size of the instruction buffer.

- With only 4 TBs per SM, there might be too few warps to be able to hide any significant number of long-latency stall cycles.

## 7.7 Sensitivity Analysis

I performed various sensitivity analyses based on the memory bandwidth, the number of warps per SM and the caches. The goal is to see how these factors affect *FGTO* and *FRRR* compared to *Vortex*, not necessarily how they affect the overall performance.

**Increase warps per SM**. Figure 7.12 shows the CPI stacks for the same configuration as described in the experimental setup but with 8 warps per SM. On average *FGTO* and *FRRR* respectively reduce CPI by 12.9% and 13.2% over *Vortex*. This is a greater reduction than when using 4 warps per SM. When using 8 warps per SM, a larger proportion of the stalls in *Vortex* are frontend stalls. The baseline configuration thus continues to be inhibited by the frontend throughput. The increase is mainly due to the increase in *empty ibuffer* stalls. This is because the frontend is unable to fetch enough instructions to fill the increased number of instruction buffers. The *FGTO* and *FRRR* are clearly still able to fill the instruction buffer, as there are close to no frontend stalls in any of the benchmarks. *SRAD, SC, BFS* and *CFD* continue to experience the same issues as described in Section 7.2.

The increased frontend throughput allows *FGTO* and *FRRR* to utilize memory bandwidth and hide data stalls to a greater degree, as shown by the increased

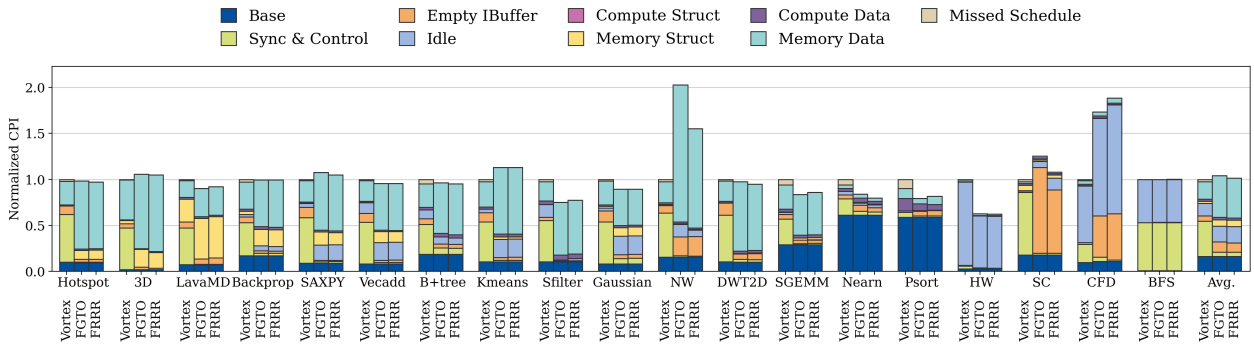



**Figure 7.13:** Normalized CPI stacks when using two memory channels, giving a bandwidth of $34.8GB/s$

proportion of *memory structural* stalls. The CPI of *sgemm* is reduced to a much larger degree by *FGTO* and *FRRR* when using 8 warps than when using 4 warps. When using 4 warps, *sgemm* is largely dominated by *memory data* stalls. By increasing the number of warps to 8, the low average latency of *sgemm* can be hidden, resulting in a drastic performance improvement. The same goes for *psort* which is close to never stalling for both *FGTO* and *FRRR*.

The increased MLP caused by having more warps, is increasing the memory bandwidth requirement. Because of this, the NoC's skewed bandwidth distribution is likely to have an even greater effect. This is why benchmarks such as *vecadd* are seeing a larger proportion of *idle* stalls. For *gaussian*, the increased proportion of *idle* stalls is likely a combination of skewed bandwidth distribution and the workload not being divided into enough TBs to fill the increased number of TB slots per SM.

**Increase DDR4 bandwidth.** Figure 7.13 shows the CPI stacks for Vortex using 2 channel DDR4 memory, resulting in $34.8GB/s$ of bandwidth, giving 2× the available bandwidth of the memory configuration described in the experimental setup. For benchmarks with high bandwidth utilization, such as *lavaMD*, the additional available bandwidth enables both *FGTO* and *FRRR* to reduce CPI, instead of increasing it. For most other benchmarks, the additional memory bandwidth does not alter how the changes affect CPI. The benchmarks continue to be dominated by *memory data* stalls, and are thus unable to utilize the bandwidth. Thus both *FGTO* and *FRRR* remain latency bound. It is difficult to determine why the CPI of *NW* and *CFD* for are almost doubled for *FGTO* and *FRRR* compared to *Vortex*. I believe it might be an issue with the simulation, as I previously experienced issues, where the simulator gave erroneous results.

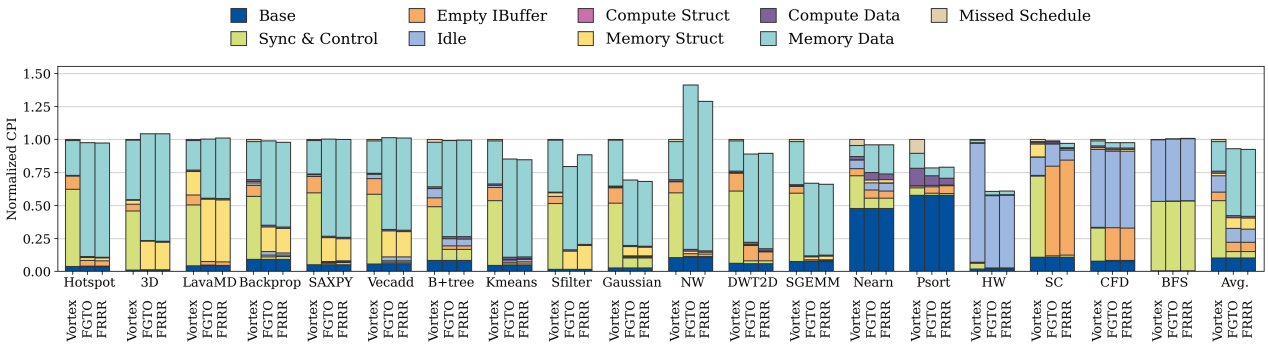



**Figure 7.14:** Normalized CPI stacks when using only L1 caches

**Only L1 caches.** Figure 7.14 shows the normalized CPI stacks when using only L1 caches, i.e. not using a L2 cache for each cluster. For most of the benchmarks, the increased memory latency of having no L2 cache makes *FGTO* and *FRRR* have less impact on performance as a smaller proportion of the stall cycles can be hidden. Most interestingly, the *idle* stalls incurred by skewed bandwidth distribution in *backprop, saxpy, vecadd, b+tree, kmeans and gaussian* are gone. This indicates that the L2 cache bandwidth might be too low, or the memory arbiters replacing the L2 caches, are distributing the memory bandwidth differently. Interestingly *FGTO* and *FRRR* reduce CPI by 7.02% and 7.42% respectively, which is more than when using L2 caches. The extra improvement mostly comes from there being no additional idle stalls.

The CPI is drastically increased for *NW* in the *FGTO* and *FRRR* configurations. This increase is due to a large increase in *memory data* stalls. The increased frontend throughput might cause the requests to queue up in the memory system, as described in Section 7.4. This effect becomes even stronger as the L1 cache hit rate is low and the L2 cache is removed. Because of this, the latency is increased which in turn increases *memory data* stalls.

# Chapter 8

# Conclusion and Further Work

## 8.1 Conclusion

In this thesis, I proposed and implemented improvements to Vortex' frontend, increasing the bandwidth of the fetch, decode and issue stages. The proposed changes were based on the work and findings in my project thesis. This accounts for contribution **C1** corresponding with task **T1**.

I improved CSV by making it consider the stall cause of all warps in the issue stage. This gives a better overview of the issue stage and enables it to show frontend stalls for individual warps. This improvement comprises contribution **C2** which corresponds with task **T2**.

Through evaluating the implemented changes, I find that the increased frontend throughput and issue bandwidth has varying effects on the performance. The improvements to the frontend, reduce frontend related stalls by 71% on average, and even remove all frontend stalls from *sfilter*, as it has no control-flow instructions. The improved ready instruction scheduler removes all missed scheduling opportunities. This is reducing the CPI of *psort* by 20%, as low-latency stalls can be hidden. However, the CPI is only reduced by 5.4%. This is because the bottleneck is moved to the backend of Vortex, as warps have to wait for long-latency memory stalls. I find that the uncovered *memory data* stalls are a consequence of the Vortex configuration being too small. This is also why there is no significant performance gain when using GTO instead of LRR scheduling. This evaluation constitutes contribution **C3** corresponding with task **T3**.

I adapt Rodinia benchmarks to enable them to run on Vortex. This is improving the analysis, by adding a set of benchmarks more representative of real workloads. By additionally enabling Vortex to profile multi-kernel programs, it becomes possible to add a wider range of benchmarks. This comprises contribution **C4** which corresponds with task **T4**.

## 8.2 Further Work

Though the improvements I proposed in this thesis improved frontend and issue bandwidth, they did not improve the performance to the same degree. The reason behind this is most likely that the SMs are too small to hide latency. Having more warps per SM together with a more fitting memory system is likely required to solve this problem. However, it is infeasible to continue using software simulation while increasing the size of the SMs and the GPU. To solve this FPGA-acceleration is required. Thus the natural next step is to integrate Vortex into Chipyard and FireSim to resolve the issues regarding the discrepancies between FPGA and ASIC clock frequencies.

There are also other issues which need to be resolved to make Vortex a reasonable GPU. Vortex is unable to perform efficient workload balancing. In some cases, the TB scheduler is unable to distribute work to all of the SMs, making them idle. In other cases, SMs finish execution long before others. Gaining a better understanding of Vortex' TB scheduler and workload distribution is probably required to understand the source of the problem. It is likely better to implement a dynamic system, similar to the one made by Nvidia [13].

# Bibliography


[1] L. M. Aurud, 'Performance Analysis of the Vortex GPGPU,' Dec. 2022.

[2] D. Burg and J. H. Ausubel, 'Moore's Law revisited through Intel chip density,' *PLOS ONE*, vol. 16, pp. 1–18, 2021.

[3] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous and A. LeBlanc, 'Design of ion-implanted MOSFET's with very small physical dimensions,' *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, 1974.

[4] G. M. Amdahl, 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,' in *Proceedings of the Spring Joint Computer Conference*, 1967, pp. 483–485.

[5] A. R. Brodtkorb, T. R. Hagen and M. L. Sætra, 'Graphics processing unit (GPU) programming strategies and trends in GPU computing,' *Journal of Parallel and Distributed Computing*, vol. 73, pp. 4–13, 2013.

[6] C. McClanahan, 'History and evolution of gpu architecture,' *A Survey Paper*, vol. 9, pp. 1–7, 2010.

[7] B. Tine, K. P. Yalamarthy, F. Elsabbagh and K. Hyesoon, 'Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics,' in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021, pp. 754–766.

[8] M. Rekdal, 'Investigating the Performance Scalability of the Vortex GPU,' *NTNU Open*, 2022.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee and K. Skadron, 'Rodinia: A benchmark suite for heterogeneous computing,' in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[10] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang and K. Skadron, 'A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads,' in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2010, pp. 1–11.

[11] J. Tompson and K. Schlachter, 'An introduction to the OpenCL programming model,' *Person Education*, vol. 49, p. 31, 2012.

[12] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo and L. Eeckhout, 'Get out of the Valley: Power-Efficient Address Mapping for GPUs,' in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018, pp. 166–179.

[13] A. Ukarande, S. Patidar and R. Rangan, 'Locality-Aware CTA Scheduling for Gaming Applications,' *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, 2021.

[14] L. Wang, 'Modeling and Minimizing Memory Contention in General-Purpose GPUs,' Ph.D. dissertation, Ghent University, 2020.

[15] S. G. Pandey and S. Gopalakrishnan, 'Improving GPGPU Performance Using Efficient Scheduling,' in *Proceedings of the International Conference on Intelligent Sustainable Systems (ICISS)*, 2019, pp. 570–577.

[16] T. G. Rogers, M. O'Connor and T. M. Aamodt, 'Cache-Conscious Wavefront Scheduling,' in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012, pp. 72–83.

[17] Y. Zhang, Z. Xing, C. Liu, C. Tang and Q. Wang, 'Locality based warp scheduling in GPGPUs,' *Future Generation Computer Systems*, vol. 82, pp. 520–527, 2018.

[18] S.-Y. Lee and C.-J. Wu, 'CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads,' in *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2014, pp. 175–186.

[19] J. Power, J. Hestness, M. S. Orr, M. D. Hill and D. A. Wood, 'Gem5-gpu: A heterogeneous cpu-gpu simulator,' *IEEE Computer Architecture Letters*, vol. 14, pp. 34–36, 2015.

[20] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, 'Analyzing CUDA workloads using a detailed GPU simulator,' in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.

[21] R. Ubal, B. Jang, P. Mistry, D. Schaa and D. Kaeli, 'Multi2Sim: A simulation framework for CPU-GPU computing,' in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 335–344.

[22] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain and T. Rogers, 'Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level,' in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.

[23] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol and K. Sankaralingam, 'Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU,' *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, 2015.

[24] J. Bush, P. Dexter, T. N. Miller and A. Carpenter, 'Nyami: a synthesizable GPU architectural model for general-purpose and graphics-specific workloads,' in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 173–182.

[25] K. Andryc, M. Merchant and R. Tessier, 'FlexGrip: A soft GPGPU for FPGAs,' in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230–237.

[26] B. Tine, V. Saxena, S. Srivatsan, J. R. Simpson, F. Alzammar, L. Cooper and H. Kim, 'Skybox: Open-Source Graphic Rendering on Programmable RISC-V GPUs,' in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 3, 2023, pp. 616–630.

[27] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep and A. Agarwal, 'Graphite: A distributed parallel simulator for multicores,' in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[28] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. Hill, D. Wood, S. Huss-Lederman and J. Larus, 'Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator,' *IEEE Concurrency*, vol. 8, pp. 12–20, 2000.

[29] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson and K. Asanovic´, 'RAMP gold: An FPGA-based architecture simulator for multiprocessors,' in *Proceedings of the Design Automation Conference (DAC)*, 2010, pp. 463–468.

[30] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill and D. A. Wood, 'Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset,' *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 92–99, 2005.

[31] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović and B. Nikolić, 'Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,' *IEEE Micro*, vol. 40, pp. 10–21, 2020.

[32] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach and K. Asanovic, 'FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud,' in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[33] J. Cortadella, M. Kishinevsky and B. Grundmann, 'Synthesis of Synchronous Elastic Architectures,' in *Proceedings of the Design Automation Conference (DAC)*, 2006, pp. 657–662.

[34] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee and D. Patterson, 'The RISC-V instruction set manual,' *Volume I: User-Level ISA', version*, vol. 2, 2014.

[35] M. Naderan-Tahan and L. Eeckhout, 'Cactus: Top-Down GPU-Compute Benchmarking using Real-Life Applications,' in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2021, pp. 176–188.

[36] W. Snyder. 'Verilator.' (2022), [Online]. Available: `https://www.veripool.org/verilator/`.

[37] P. Jääskeläinen, C. S. Lama, E. Schnetter, K. Raiskila, J. Takala and H. Berg, 'pocl: A Performance-Portable OpenCL Implementation,' *Int. J. Parallel Program.*, vol. 43, pp. 752–785, 2015.

[38] E. Perelman, G. Hamerly and B. Calder, 'Picking statistically valid and early simulation points,' in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003, pp. 244–255.

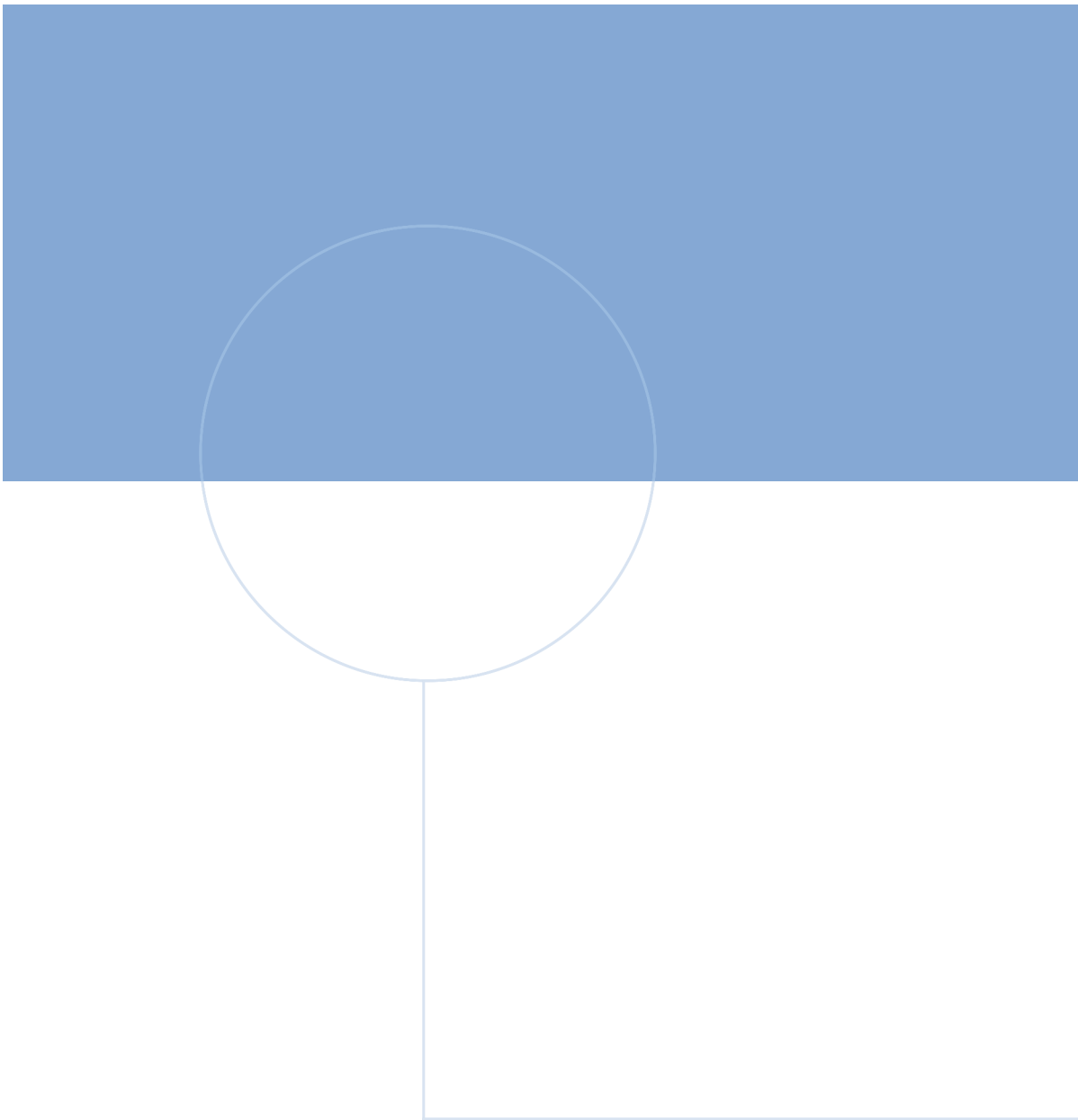
[39] J. Alsop, M. D. Sinclair, R. Komuravelli and S. V. Adve, 'GSI: A GPU Stall Inspector to characterize the sources of memory stalls for tightly coupled GPUs,' in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 172–182.

[40] J. Lee, Y. Ha, S. Lee, J. Woo, J. Lee, H. Jang and Y. Kim, 'GCoM: A Detailed GPU Core Model for Accurate Analytical Modeling of Modern GPUs,' in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022, pp. 424–436.

[41] B. Gottschall, L. Eeckhout and M. Jahre, 'TIP: Time-Proportional Instruction Profiling,' in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021, pp. 15–27.

[42] Y. Kim, W. Yang and O. Mutlu, 'Ramulator: A Fast and Extensible DRAM Simulator,' *IEEE Computer Architecture Letters*, vol. 15, pp. 45–49, 2016.

[43] M. Själander, M. Jahre, G. Tufte and N. Reissmann, *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*, 2019.



NTNU

Kunnskap for en bedre verden