Performance Analysis of the Vortex GPGPU

1st Lars Murud Aurud
Department of Computer Science
NTNU
Trondheim, Norway
larsmaur@stud.ntnu.no

***Abstract*—Vortex is an open source GPGPU designed for architecture research and FPGA-accelerated simulation. It allows for exploring all stages of the system, such as compiler, driver and hardware stack. Previous investigations found that the performance of Vortex with existing benchmarks did not scale well with increasing numbers of cores. In this paper we implement *CPI stacks for vortex* bringing hardware support for generating CPI stacks. This paper utilize the CPI stacks to investigate the performance of Vortex, and find potential bottlenecks. We find that the *psort*, *vecadd*, *sfilter* and *sgemm* benchmarks scale much better with the number of cores, when increasing the input size beyond default. We also find that *nearn* and *saxpy* are unable to utilize multiple cores. When using reasonable input sizes, we observe that Vortex's CPI is largely dominated by dependency stalls, both from compute and memory. To solve this, we propose possible improvements to the schedulers and front-end.**

***Index Terms*—Vortex, GPGPU, CPI stacks**

I. Introduction

A common method for evaluation in computer architecture is software simulation [1] [2] [3]. Simulating entire systems is rather slow, especially for large multi-core architectures [4]. Software simulations are thus troublesome for evaluating GPU architectures. There are multiple levels of abstractions when running simulations, ranging from the OneIPC model to more precise, but slower, cycle-accurate simulations [5]. To obtain accurate performance metrics, reflecting the absolute performance of an architecture, cycle accurate simulations are required. Cycle accurate simulations can thus give us insight into why the architecture performs as it does.

FPGA-acceleration can be used to speed up architecture simulations. RAMP-gold [4], an FPGA-based multicore simulator, achieved a 263× speedup over the software based simulator GEMS [6]. Thus FPGAs serves as a middle ground between software simulation and prototypes, which are costly and have a long turnaround time. FPGAs allow for evaluating the parallel multicore workloads of GPUs in an efficient manner. There are even cloud based systems running Firesim [7] which allows for scalable FPGA-accelerated cycle-accurate simulations.

Vortex is a RISC-V based GPGPU presented at MICRO 21 [8]. It was designed with the intention of enabling architecture research and FPGA-accelerated simulation. Vortex has two main issues: the benchmark suite, and the difference in FPGA and ASIC clock frequencies in relation to memory speeds. The benchmark suite is quite small and multiple benchmarks does not work, or has performance issues, which are not well understood. A previous analysis of Vortex's working benchmarks by M. Rekdal [9] found that the kernel size was a bottle neck for some of the benchmarks. In other cases the benchmarks had poor performance scaling with the number of cores. When simulating architectures, having a realistic relationship between the clock frequency, and memory bandwidth and latency is required to obtain realistic results. As the clock frequencies for FPGAs are lower than for ASICs, the memory speed have to be reduced accordingly. Firesim [7] solved this problem using a token mechanism. The latter issue will not be addressed further in this paper, as we do not run Vortex on an FPGA.

In this paper we implement *CPI stacks for vortex* (CSV) to bring insight into the performance issues of the benchmarks. CSV brings hardware support for identifying what is done in each cycle. It is implemented in rtl and will thus work for future FPGA implementations. The collected metrics are used to generate cycle stacks for a set of configurations and benchmarks. We observe that the *psort*, *vecadd*, *sfilter* and *sgemm* benchmarks require input sizes much larger than their default sizes to utilize all the cores of vortex. *Psort* has the best performance scaling, improving with both input size and number of cores. The CPI of *vecadd*, *sfilter* and *sgemm* does not always scale well with an increasing number of cores due to stalling for dependant memory and compute instructions. Lastly we observe that the *saxpy* and *nearn* benchmarks are unable to utilize more than one core, resulting in poor performance. To address some of these issues we suggest possible improvements to the Vortex front-end and schedulers. In addition, further analysis could be improved by using a common GPU benchmark suite such as Rodinia [10].

II. Background

A. GPU execution model

GPUs are accelerators designed to run highly parallelizable applications. The kernel is the part of the application which can be parallelized. The programming model of the GPU is data-parallel, that is the kernel is divided such that each part performs the same operations on different data. To achieve high performance for these programs, GPUs utilize multiple compute units often called streaming multiprocessors (SMs), the Vortex team refers to these as cores. Each core is designed to execute multiple threads with the same instructions (SIMT) in lockstep. The grid layout of a GPU kernel is shown in Figure 1. The threads of a kernel are grouped into waves,

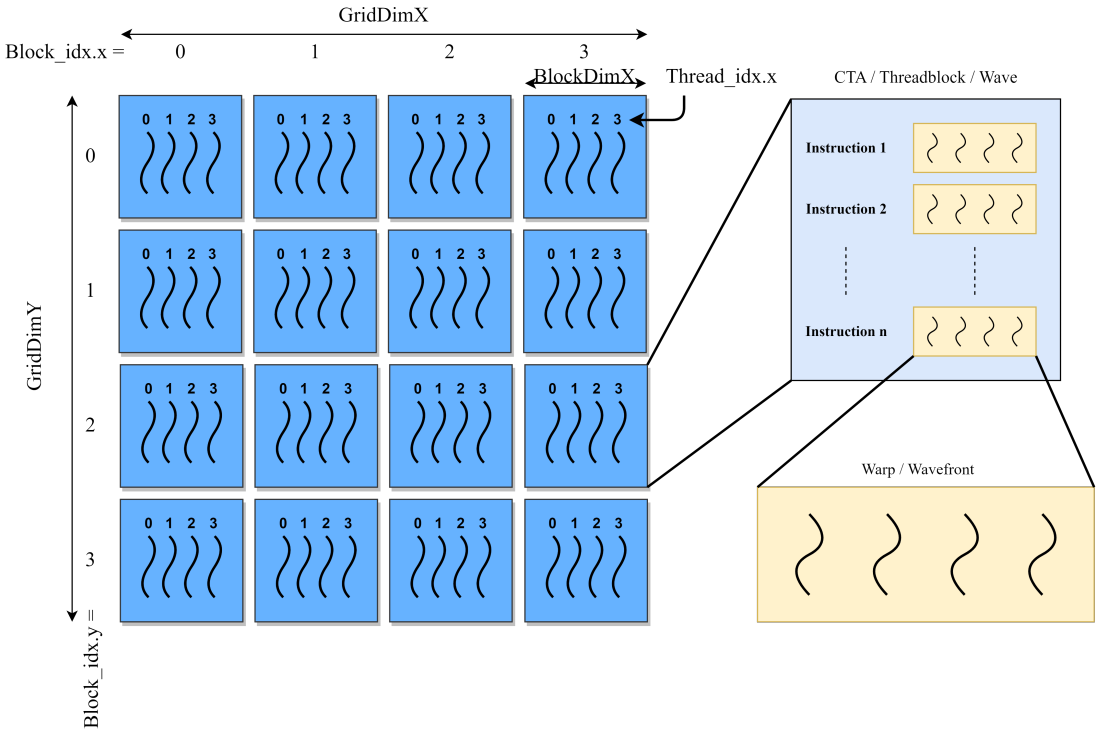


Fig. 1: GPU kernel represented as a grid of waves/thread blocks. Each wave has a set of threads with the same instructions but different data such that they can run in lockstep. The instructions in the waves are called wavefronts or warps.

also known as cooperative thread arrays (CTAs) or thread blocks. Each wave holds a fixed number of threads which can execute in sequence or parallel, depending on branch divergence. Each instruction in a wave is called a wavefront or warp as shown in Figure 1. The waves are allocated on the cores of the GPU, and occupies the core until all its threads have finished execution. Each core will typically have multiple waves allocated at a time. During execution, the core schedules a wavefront from one of the available waves. When a wave is stalled, the GPU can hide the stall by scheduling wavefronts form other waves. This allows the GPU to maintain a high throughput and execute a large number of instructions per cycle.

III. Methodology

A. Vortex

Vortex [8] is a RISC-V based general-purpose graphics processing unit (GPGPU). It was presented at MICRO 2021 and is still in development. The microarchitecture of Vortex is shown in Figure 2. Each core implements a five-stage in-order RISC-V pipeline. The pipeline is augmented with SIMT components, such as a wavefront scheduler, thread masks and an immediate postdominator (IPDOM) stack, and banked general-purpose registers for each thread in each wave. The purpose of the IPDOM stack is to handle diverging threads in a wave. It saves the not-taken threads such that they can be restored at a later point. Thread masks enable the GPU to execute a selections of the threads in case of divergence. To enable wave-level synchronization, barrier control signals can be sent to the scheduler. The vortex cores are kept quite simple to save power and area. There are for example no branch prediction or forwarding. Dispite this it can maintain a high throughput by focusing on thread level parallelism (TLP) and not instruction level parallelism (ILP). A stalling wave is not detrimental to the performance of the GPU as it can schedule wavefronts from another wave. The Vortex cores can be configured and grouped into a number clusters, where each cluster has its own L2 cache. All the clusters also share an L3 cache as shown in Figure 2.

Vortex has three schedulers, these will be referred to as:

- Wave scheduler
- Wavefront scheduler
- Instruction scheduler

The **wave scheduler** allocates waves to each core. We have been unable to find information in regards to the algorithm used in Vortex.

Vortex's **wavefront scheduler** is situated in the fetch stage, and decides what to fetch from the I-cache and send to decode. It contains three bit-masks: *active waves*, *stalled waves* and *barrier stalled waves*. The set of ready waves are calculated as:

$$\text{ready}_i = \text{active}_i \wedge \neg(\text{stalled}_i \vee \text{barrier}_i) \tag{1}$$

This creates a bit-mask where each index corresponds with the id of a wave in the core. The wavefront scheduler selects which of the waves in the set of ready waves to schedule next. Vortex

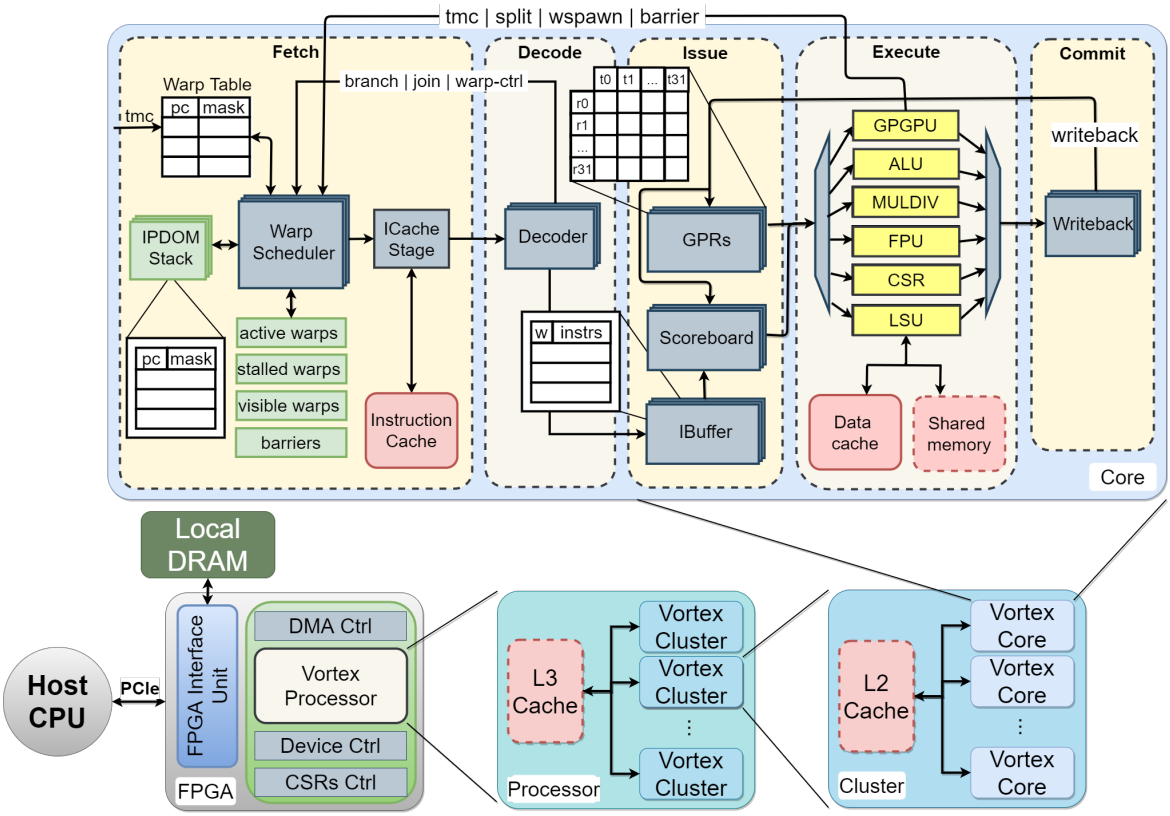


Fig. 2: Vortex microarchitecture

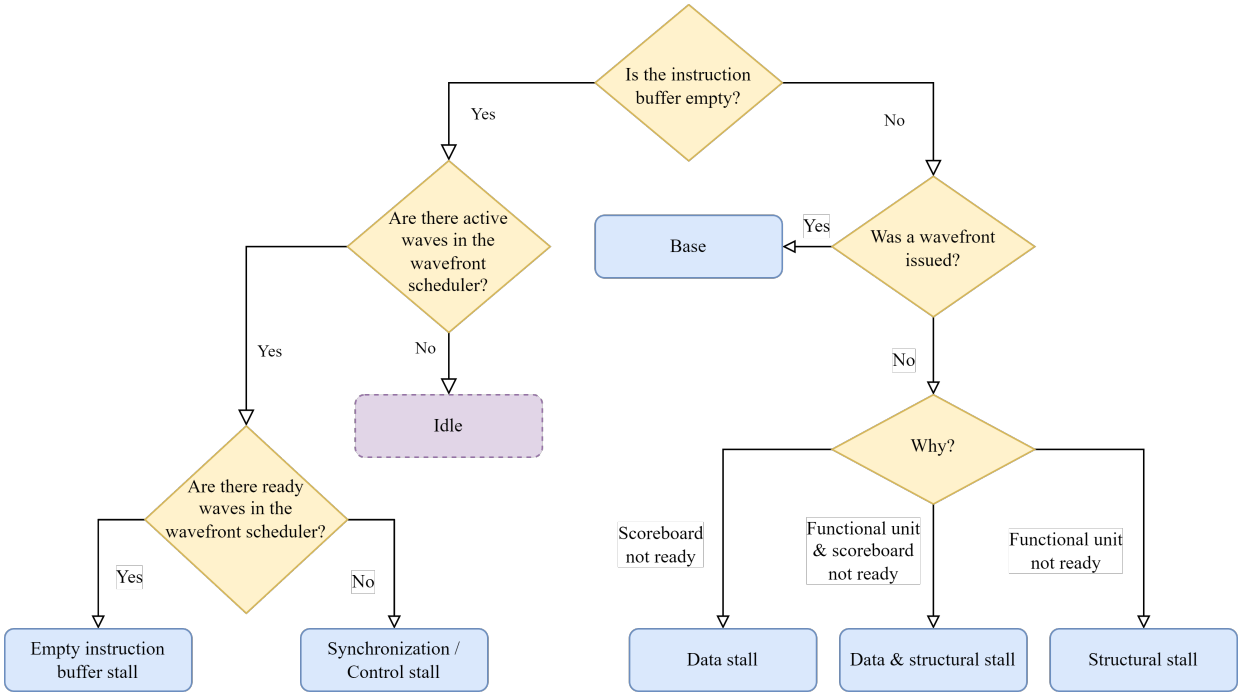


Fig. 3: Flowchart for attributing cycles to stall classes

use *find first* scheduling, which prioritizes the waves based on their index, i.e it selects the ready wave with the lowest set bit in the ready bit-mask.
After a wavefront is scheduled in Vortex, the corresponding wave is stalled for at least one cycles. Vortex has to stall the wave until the wavefront has been decoded, to know if the wavefront can change control flow. Vortex has to do this because it cannot flush the pipeline. Thus scheduling the wrong instruction could result in incorrect program behaviour. If the decode stage identifies that the wavefront cannot change control flow, a signal is sent to the scheduler, no longer marking the wave as stalled. If the instruction can change control flow, the wave continues to be stalled until the instruction has executed.

The **instruction scheduler** is a part of the instruction buffer in the issue stage. The instruction buffer can hold a number of wavefronts from each wave, as illustrated in 5b. In each cycle, one of the waves with a wavefront in the buffer is selected using round-robin. The oldest wavefront in the selected wave is then attempted to be issued. An instruction can only be issued if the required functional unit is ready, and the source and destination registers are not in-use by another instruction. To maintain the latter restriction, Vortex uses scoreboarding. Note that each wave has a separate register bank, such that there are no inter-wave dependencies. If the selected wavefront cannot be issued, no wavefront is issued that cycle, even if the wavefront of another wave could be issued. This is because the instruction buffer does not have any information regarding the state of the scoreboard or dispatch unit.

B. GPU Stall classification

To profile the GPU, J. Alsop et al. [11] propose the GPU Stall Inspector (GSI). In their paper they introduce a classification scheme for GPU stalls. They define an *instruction stall* as when a specific instruction cannot be issued. A *stall cycle* is then defined as a cycle where no instructions can be issued. GSI attributes stall cycles in two stages. In the case of a stall cycle, a stall type is first assigned to each instruction. This classification is based on the cause which is most strongly preventing the execution. They argue that the strength of the stall is linked to the likelihood that the instruction will remain stalled in the next cycle. The second stage of the classification considers the stall causes of each wavefront and classifies the cycle based on the weakest stall cause, i.e. the instruction closest to issuing.

In the version of Vortex used in this paper, we do not have to apply this two stage classification. This is because Vortex only attempts to issue one wavefront each cycle. This attempt is done without any knowledge of the availability of dependencies or functional units. If the attempt fails, the issue stage stalls for one cycle, before it attempts to issue the next available wavefront. Thus the classification can be done in one stage without any assumptions of strong or weak stall causes.

The cycle classes from GSI can, with some minor changes to better fit the architecture, be utilized for vortex. Below, we list the classes and their definitions. A flowchart for how we classify the cycles are presented in Figure 3.

- A **Base cycle** is a cycle where an instruction was issued.
- An **Idle stall** is when there are no instructions to attempt to issue and no active waves to schedule.
- A **Synchronization / Control stall** is caused by there being no instructions in the instruction buffer and no ready waves to schedule. i.e there are active waves, but all of them are stalled or blocked.
- An **Empty ibuffer stall** is a fallback for *Idle stall* and *Synchronization / Control stall*, it occurs if there are no instructions in the instruction buffer, but there are ready waves.
- A **Memory data stall** occurs when an instruction cannot issue because its operands are dependant on the result of a pending load.
- A **Memory structural stall** occurs when a memory instruction cannot be issued because the load-store unit (LSU) is occupied.
- A **Compute data stall** occurs when the operands of an instruction are dependant on the result of a pending compute instruction. I.e. all instructions to functional units other than the LSU.
- A **Compute structural stall** occurs when a compute instruction cannot be issued because the corresponding functional unit is occupied.
- A **Data & structural stall** is when an instruction is waiting for both waiting for its operands and the corresponding functional unit to be ready

C. Data collection in Vortex

The version of Vortex used in this paper already included a way of collecting performance data. Each core in the GPU tracks its performance metrics which are available as control status registers (CSR). At the end of the application, these registers are read and written to memory. Reading the performance metrics may influence them, as the read instructions is a part of the application running on the GPU. This would be problematic, as it is desirable that all the metrics capture the same state of the GPU. To deal with this problem, we implemented a *performance lock* register. This register can be set by writing to the corresponding CSR register. When set, the performance registers no longer update. By locking the performance registers before reading the data, it ensures that all the metrics come from the same state. The added benefit of this is that the performance collection wont influence the performance of the benchmarks. This is important as some of the benchmarks are quite short in terms of cycles. In addition, changing the collection program or number of collected metrics wont influence the results.

D. Stall classification in Vortex

The following section will cover details of the implementation of the classification scheme presented in section III-B.

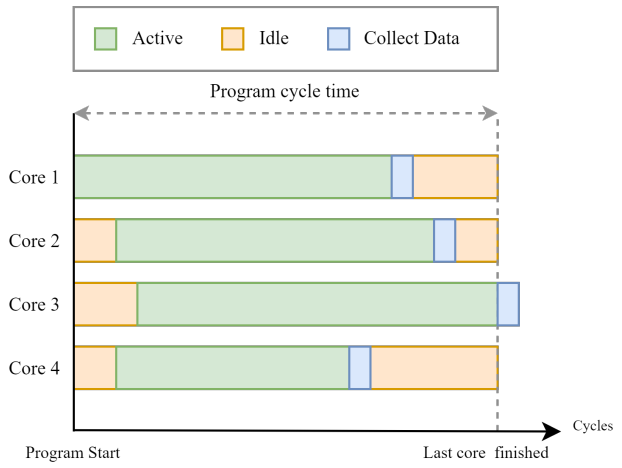


Fig. 4: Plot of arbitrary program ran on four cores. To calculate the idle cycles of a core, subtract its active cycles from the finish time of the last core

1) Idle: One problem arise when we collect performance metrics for each core when it is finished. Figure 4 illustrates how the activity of the different cores might look for an arbitrary program. It also shows how the metrics of each core is collected at different points in time. While the core would be idle until all the other cores are finished, the collected metrics does not include these cycles. To solve this, each core tracks the number of active cycles, and the number of cycles since program start. The number of idle cycles for each core can be calculated as

$$C^i_{idle} = \max(C_{total}) - C^i_{active} \tag{2}$$

where C^i represents the cycles of core i and $max(C_{total})$ is the cycles from start to end of the program.

2) Base: Cycles are attributed as *base* cycles if instructions are issued. In the case of Vortex, three conditions have to be met in order to issue an instruction. The instruction buffer needs to have at least one instruction, the corresponding functional unit needs to be ready and the operands of the instruction needs to be ready. In addition to this, the round-robin selector has to attempt to issue the instruction.

3) Front-end stalls: As stated above, if there are no instructions in the instruction buffer, no instructions are issued. The most common reason for the instruction buffer to be empty is that the wavefront scheduler is not scheduling instructions. Thus we divide these stalls into three classes, two based on the cause of no instructions being scheduled and one fall back.

- **Idle** cycles are attributed as described above.
- **Synchronization / Control stalls** are attributed if the instruction buffer is empty and there are active waves, but no ready waves. Thus all the active waves are stalled or blocked by a barrier and no wavefronts can be scheduled.
- **Empty Ibuffer** is a fall back case if the instruction buffer is empty, but there are ready waves. This could for example be because of misses in the instruction cache or cycle delay, as described below.

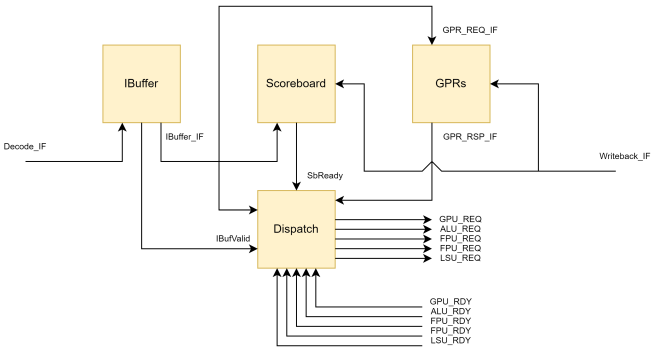
This method is not ideal as there are multiple cycles of delay between a wavefront being scheduled and it appearing in the instruction buffer. Thus if the instruction buffer is empty, there could be ready waves in the scheduler, while the reason for the instruction buffer being empty is that there were no ready waves a few cycles earlier. These cycles would be attributed as *empty ibuffer* and not *synchronization / control*. It will still give an indication of what causes the front-end stalls and could be investigated further in case front-end stalls are deemed to be an issue.

4) Data stalls: Figure 5a illustrates the issue stage of vortex in greater detail. Vortex utilizes a scoreboard to handle data dependencies. The scoreboard tracks dependencies by reserving the destination register of every issued instruction and freeing the destination register of committed instructions. For an instruction to be ready to be issued, the destination register and all source register must free (i.e not be reserved by another instruction). To track which instructions cause the data stall, the functional unit reserving the register is also tracked. When a data stall occurs, the source and destination registers of the stalled instruction can be used to lookup the functional units which reserved the registers. The owners of these registers can be used to attribute the stall to memory and/or compute data stalls.

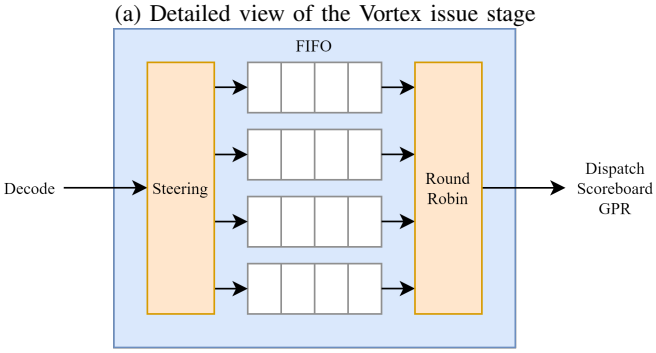
Instructions can be dependent on the results from multiple different functional units to finish before it can issue. Thus an instruction could end up waiting for both memory and compute units. In vortex, all combinations of data stalls are counted e.g. we count how many cycles an instruction stalls for only ALU, and for both ALU and LSU. As memory and compute data stalls can occur in the same cycle, we have to decide how to attribute these. An option would be to include an additional class for memory and compute data stalls. Since the number of these stalls were so small ($< 0.5\%$), we divide the cycles equally into each class. Similar to the time proportional instruction profiler [12].

5) Structural stalls: As shown in Figure 5a, the dispatch unit receives a ready signal from all the functional units indicating if it is ready to execute a new instruction. If the ready signal corresponding to the to-be issued instruction is low, a structural stall has occurred. Tracking the type of the structural stall is quite simple, as it can be attributed to the functional unit required by the instruction. If the functional unit is the load-store unit, the stall is attributed as a memory structural stall. Otherwise it is a compute structural stall.

6) Multiple stall causes: While it is clear that front-end stalls cannot occur in the same cycle as data or structural stalls, data and structural stalls may occur in the same cycle. The results later showed few occurrences of simultaneous data and structural stalls. While these cycles could be divided and attributed in the same manner as the memory and compute data stalls, the way the data was collected did not allow for this. Thus all the occurrences of simulaneous data and structural stall are attributed to a single class of *data & structural stall*.



(a) Detailed view of the Vortex issue stage



(b) Detailed view of the instruction buffer in the issue stage. There is one FIFO corresponding to each wavefront. The round robin selector selects from the non-empty FIFOs

Fig. 5: Detailed view of issue stage and instruction buffer

IV. Experimental Setup

The Vortex project includes a built in blackbox benchmark setup. The setup scripts allow for setting several parameters and configurations for the benchmarks and architecture. Table I lists the parameters used in this paper. The benchmarks used and their input sizes are listed in Table II. The RTLSIM driver was used to simulate the Vortex processor with Verilator [13]. RTLSIM simulates the processor RTL without the accelerator functional unit (AFU) [8].

An important factor for accurate simulation results is memory simulation. Vortex uses Ramulator [14], which is a cycle-accurate DRAM simulator. In this setup, the default parameters for Vortex was used, which is to have eight chips of 4GB 2400MHz DDR4 RAM. The setup was ran using only one cluster, as Rekdal [9] found that clustering had little impact on the performance.

The version of vortex used in this paper can be found in the ntnu_main branch of the vortex-ntnu repository under the EECS-NTNU organisation on Github[1]. The version including the implementation of CSV can be found in the ntnu_main_larsmaur branch ntnu_main_larsmaur[2].

A. IDUN Cluster

For this project, all the benchmarks were ran on the IDUN Cluster [15]. Some benchmarks with low input sizes were simulated locally on configuration with low core counts to test changes. When running all benchmarks on all configurations, running on IDUN allowed for running all benchmarks with a single configuration is parallel. This saved a lot of time, as some of the benchmarks required multiple 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 makefile.

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

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

TABLE I: Configurations for the Vortex architecture.

Vortex Configuration	
No. Cores	1, 2, 4, 8, 16, 32 or 64 cores
Core resources	250 MHz, 16 SIMT width, 16KB shared memory, 4 waves/core, 4 threads/wave
Instruction buffer depth	2 Wavefronts / Wave
Scheduler	1 wavefront scheduler per core
L1 data cache	16KB per core, 8 cycle delay
L2 and L3 cache	Optional, here disabled
NoC	Hierarchical tree structure
DRAM setup	4GB, 4 channels, 1 rank
DRAM bandwidth	2400MHz DDR4, 19.2 GB/s

TABLE II: Overview of benchmarks and the input sizes used. The first input size is also the default input size

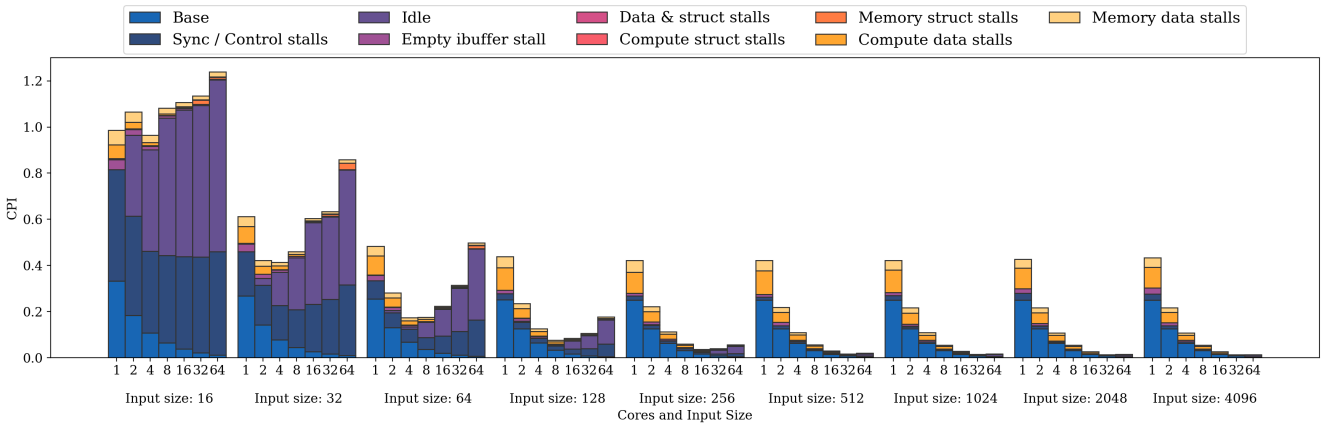
Benchmark	Name	Input Sizes
Vector Addition	vecadd	64, 128, 256, 512, 1024, 2048, 4096, 8192
General Matrix Multiply	sgemm	32, 64, 128, 256
Nearest Neighbour Search	nearn	42764
Matrix Filter (3x3 kernel)	sfilter	16, 32, 64, 128, 256, 512, 1024
Sorting	psort	16, 32, 64, 128, 256, 512, 1024, 2048, 4096
A Times X Plus Y	saxpy	1024, 2048, 4096, 8192, 16384

V. Results

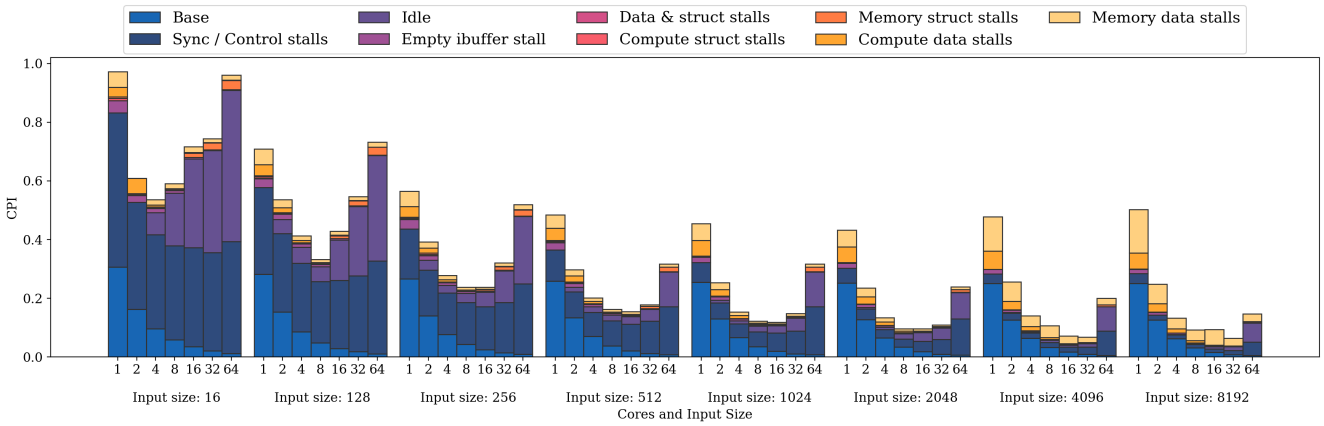
We look at the CPI stacks for each benchmarks and attempt to identify potential problems.

Psort. The CPI stacks for the psort benchmark is shown in Figure 6a. For larger input sizes (>256) it is clear that psort is able to take advantage of all of the available cores. This is because there are few idle cycles. Psort also scales well with the number of cores, when doubling the number of cores, the CPI is halved. This is not the case for lower input sizes, as idle and sync/control stalls dominate. This indicates that the kernel is too small. The sync/control stalls probably come from the setup of the cores, as when the kernel is small, these make a significant contribution to the cycle count. The sync/control stalls occur during setup as in this period, the cores only have one active wave. When only one wave is active, the wavefront scheduler is stalled every other cycle, as is explained in Section III-A.

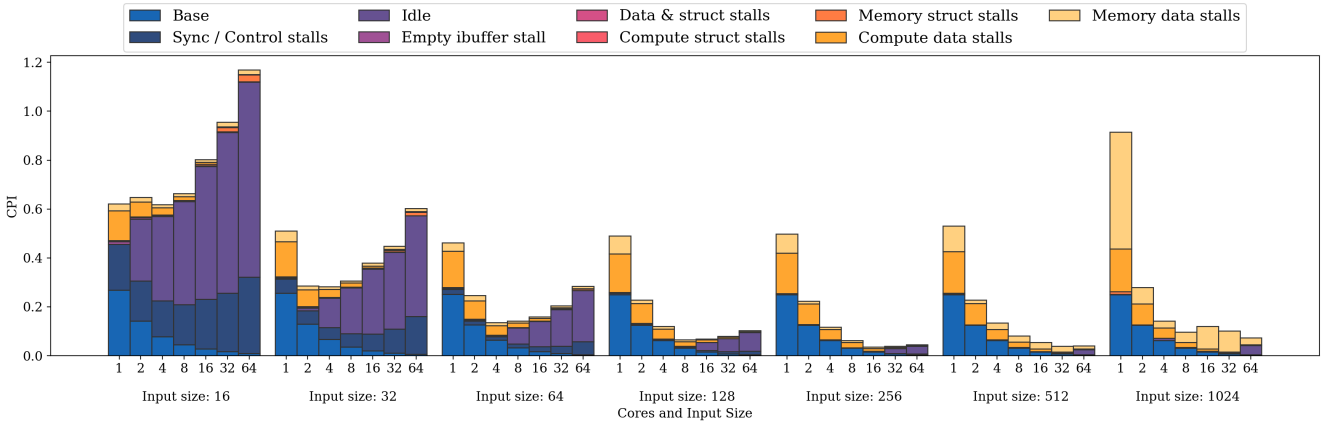
We observe that the number of memory data stalls stay constant for all input sizes. This is probably because the



(a) psort

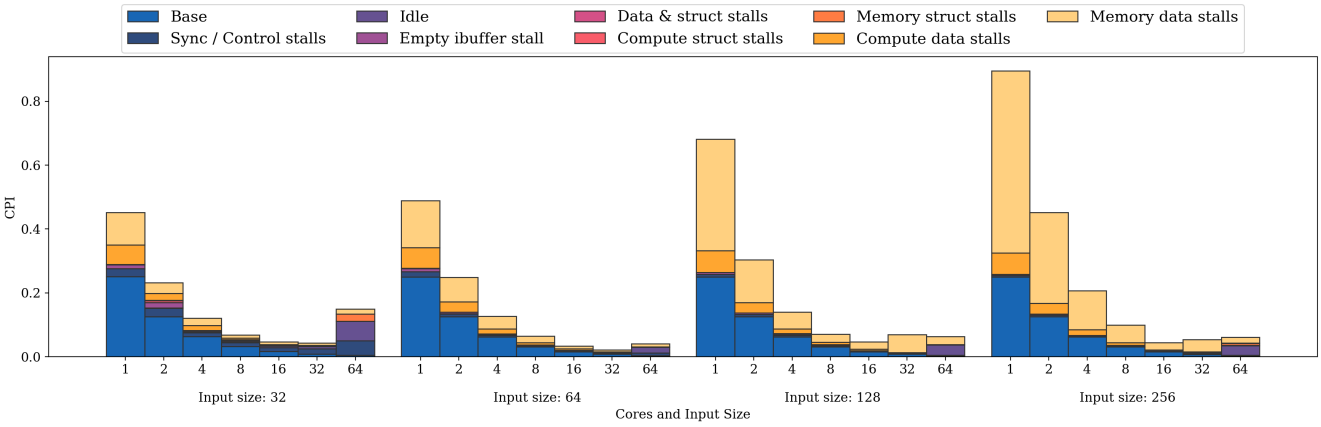


(b) vecadd



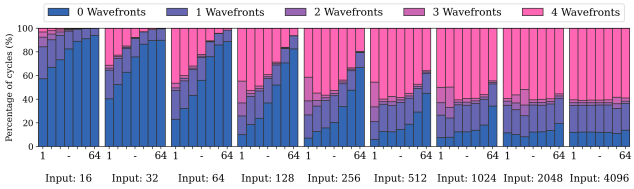
(c) sfilter

Fig. 6: CPI stacks for all benchmarks with reasonable performance results

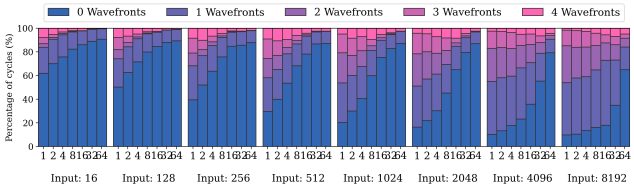


(d) sgemm

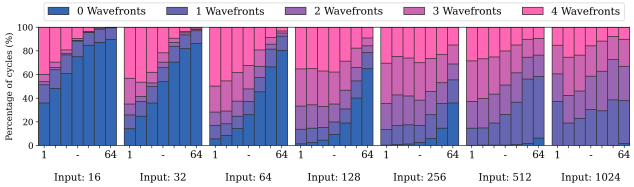
Fig. 6: CPI stacks for all benchmarks with reasonable performance results



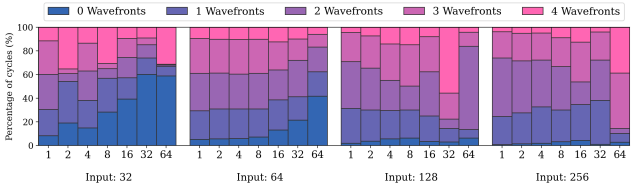
(a) psort



(b) vecadd



(c) sfilter



(d) sgemm

Fig. 7: Distribution of number of available wavefronts from different waves in the instruction buffer for active cores.

working set of psort can fit in the L1 cache for all of the input sizes. Increasing the input size further would increase the size of the working set beyond the L1 cache size.

Looking at Figure 7a, we observe that for larger input sizes ($>$256) the instruction buffer contains wavefronts from all four waves in over 60% of the cycles. Still most of the stalls are dependency stalls. This is somewhat expected, as the kernel contains a short loop with a load and some simple arithmetic operations. It is still possible that the stalls could be hidden by being more considerate when selecting what wavefront to issue. When a load completes, the wave should be able to execute all its arithmetic operations. Instead the naive round-robin algorithm of the instruction buffer does not allow a single wave to execute multiple cycles in a row which will reduce TLP.

Vecadd. The vecadd benchmark mostly follow the same trends as psort, see Figure 6b. When input size is increased, the performance scales better with the number of cores. When running with the largest input size and 64 cores, the CPI is still dominated by idle and sync / control stalls. It is probable that an even larger input size would solve this issue, as there is a clear trend of decreasing CPI for 64 cores when increasing input size.

When increasing the input size beyond 2048, the CPI starts to increase. This is due to an increase in memory data stalls. This makes sense as the size of the working set ($3 \times size \times sizeof(float)$) become larger than the L1 cache. The performance is still increased for the configurations using more cores, as multiple cores can perform memory requests, instead of stalling.

Figure 7b shows the distribution of available instructions in the instruction buffer. For the largest input size, the instruction buffer contains either zero or one instruction for over 60% of the cycles. This proportion only increases with the number of cores. Comparing these results with the results from psort, it seems like the front-end is struggling to schedule enough

instructions. As the CPI of psort and vecadd are quite comparable, the difference in available waves in the instruction buffer is an indication of a problem in the front-end. A key difference between vecadd and psort kernel is that vecadd has no branches or loops, because of this, the waves will never stall for more than one cycle in the wavefront scheduler. This results in scheduling more wavefronts from the first two waves, and less from the two last waves. This will reduce how much TLP Vortex is able to expose, resulting in more stalls.

Sfilter. Sfilter follow the same pattern as vecadd. When increasing the input size, the the performance scales better with the number of cores. It is clear that when increasing the input size from 512 to 1024, the number of memory data stalls increase drastically. The size of the working set is $2 \times size^2 \times sizeof(float)$. Thus for input sizes of 512 and 1024, the working sets are both much larger than the L1 cache. As the kernel accesses addresses with an interval of `size`, a too large input size is likely to cause more cache misses. Indeed, the cache hitrate is increasing with the input size until 512 and 1024, having 86% and 74% hitrate respectively, when using one core. We also observe an increased average memory latency from 31 to 81 cycles when increasing the input size from 512 to 1024. Similar trends arise when using more cores. All these factors result in more stalls.

It is clear that increasing the input size allows for utilizing more cores. Interestingly the number of idle cycles per instruction does not seem to change much for 64 cores when increasing beyond an input size of 256. Figure 7c shows the distribution of available wavefronts in the instruction buffer for active cores. As there are no cycles with zero available wavefronts for the input size of 1024 and 64 cores, it might be that the wave scheduler is unable to allocate the waves evenly to the cores.

Sgemm. The CPI stacks for the sgemm benchmark is displayed in Figure 6d. Note that while the CPI stacks for 64 cores are included in the plot, the results produced by the application is not correct. For most configurations, the CPI increase with the input size. The increase in CPI is caused by memory data dependencies. This makes sense as sgemm does matrix multiplication with two input and one output matrix. The size of the working set is thus given by $3 \times size^2 \times sizeof(float)$, scaling quadratically, similar to sfilter.

Figure 7d shows that for an input size of 32 and 64, the front-end of the pipeline is unable to achieve a steady state. For most configurations, there are a significant number of cycles with no available instructions in the instruction buffer. With an input size of 128 and 256, almost all cycles have at least one available instruction in the issue stage. As the distribution is similar for all cores, we can assume that it is in a steady state. When using the sgemm benchmark, it is reasonable to use 128 and 256 as input sizes and configurations up to 32 cores.

All working benchmarks. For most of the above mentioned benchmarks the number of memory data stalls increase with the input size, but the number of compute data stalls stays constant. At reasonable input sizes, these memory and compute data stalls are the only stalls. As there are few structural stalls, it is probable that the functional units are ready such that other wavefronts could be issued. Thus it is possible that some of the data stalls could be hidden using a different instruction scheduler.

Saxpy. The CPI stacks from the saxpy benchmarks shown in Figure 8a are quite different than most of the other benchmarks. A larger number of cores, does seem to lower the number of cycles per instruction. But for all instances with more than one core, there are a substantial number of idle cycles. Looking at the activity of the cores, it appears as each core has one active wave throughout the execution. The reason for the decreasing CPI is actually the increasing number of instructions due to setting up more cores. This is because setup and tear take a substantial number of cycles and instructions [9]. The benchmark is actually only utilizing a single core to run the kernel. Thus it is not worth using the benchmark in its current state.

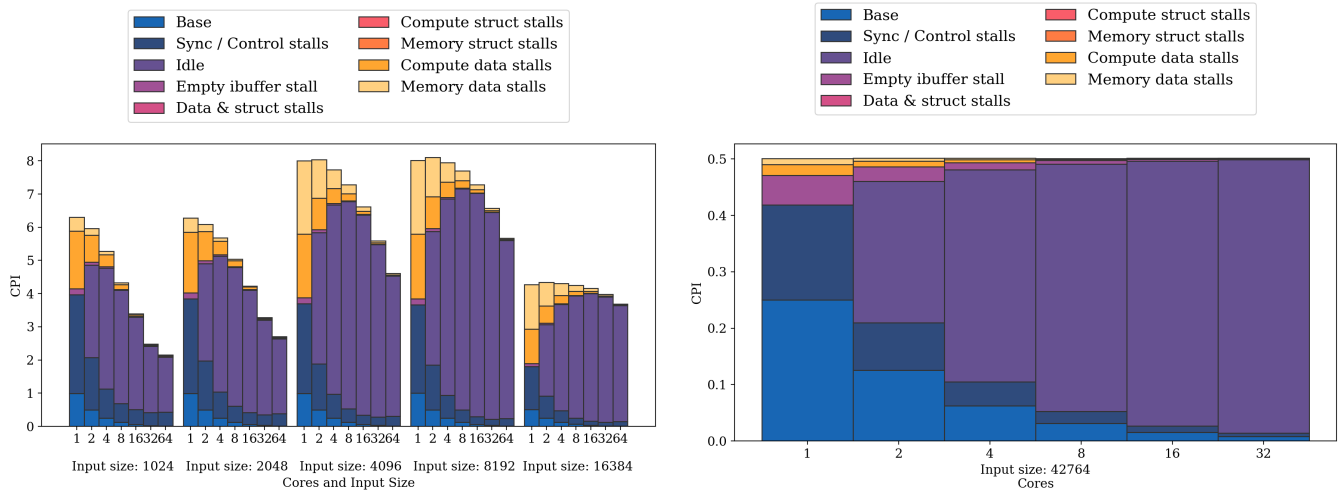
Nearn. In the case of nearn shown in Figure 8b we observe the same pattern as in saxpy, except that the perceived performance does not seem to improve with the number of cores. Looking at the activity of the cores, it becomes apparent that one core has three active waves, while all the other cores are idle. The setup and tear down cycles does not affect the results of nearn to the same degree as in saxpy. This is because nearn has a much longer runtime of 6 million cycles compared to the 25 thousand of saxpy. This explains why the CPI does not change when increasing the number of cores, but the proportion of idle cycles increase. Because of this, nearn is also not worth using as a benchmark in its current state.

VI. Improving Vortex

The results found in this paper give a pointer to what changes might lead to improved performance of the Vortex GPGPU. Following is a list of proposals for changes or areas to investigate further.

Scheduling and issuing. The different scheduling algorithms in the wavefront scheduler and the instruction scheduler could cause undesirable interactions. Investigating other combinations of algorithms could lead to better performance. In addition giving the instruction scheduler more information about the state of the operands and functional units would open for implementing for more efficient algorithms. Using a more fair wavefront scheduler, would probably make more wavefronts from different waves available in the issue stage allowing for more TLP.

Reduce control stalls. If a core has few active waves, stalling a wave for one cycle after scheduling it will have an impact on the rate at which wavefronts can be scheduled. Bringing the logic identifying whether a wavefront can cause control



(a) saxpy

(b) nearn

Fig. 8: CPI stacks for all benchmarks with unreasonable performance results

flow changes or not, from decode, to fetch would remove the need for these unnecessary stalls. As this logic is quite simple, it will likely not create a new critical path nor incur any cost in power or area.

Memory configurations. For workloads with larger working sets, memory data stalls become the dominant factor for performance. Investigating other memory configurations such as clustering could result in better performance. While Rekdal [9] found that clustering had little effect, it is unclear what input size was used for the benchmarks. Larger working sets might benefit more from clustering.

Benchmarks. Expanding the benchmark suite to include better understood benchmarks, such as Rodinia [10], would improve the analysis. This would also make it easier to compare Vortex to other architectures.

VII. Related Work

Multi-Stage CPI Stacks. Eyman et al. [16] propose to messure multiple CPI stacks during program execution, one for each stage of the pipeline. The multi-stack represention give a more complete view of the performance of applications and reveals all performance bottlenecks.

VIII. Conclusion

In this paper we implemented CSV to investigate the performance of the Vortex GPGPU. CSV brings hardware support for generating CPI stacks, such that it can be utilized in FPGA-accelerated simulations. Using CSV we found that the *saxpy* and *nearn* benchmarks were not suitable, as they were unable to utilize more than one core. Further we identified that *psort*, *vecadd*, *sfilter* and *sgemm* required input sizes significantly larger than default to produce reasonable metrics. *Psort* scaled well with both input size and number of cores. For *vecadd*, *sfilter* and *sgemm* we found that they were unable to utilize all 64 cores even for the largest input sizes used in this paper. In addition we observed that most of the non-base CPI for these benchmarks were dependency stalls. The number of memory stall cycles increase with input size while compute data stalls is mostly independent of input size.

We also propose a set of plausible improvements to Vortex. First, changing both the wavefront scheduler and the instruction scheduler is likely to improve the performance of the benchmarks affected by dependency stalls. Secondly, reducing the single cycle control stalls in the wavefront scheduler could improve the performance of benchmarks with lower input sizes. After implementing these fixes, expanding the benchmark suite would aid in identifying other issues as well as comparing with the results of other architectures. Further, with an extended benchmark suit, investigating other memory configurations would be possible.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, p. 1–7, aug 2011.

[2] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 473–486, 2020.

[3] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "MGPUSim: Enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 197–209, Association for Computing Machinery, 2019.

[4] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP Gold: An FPGA-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), p. 463–468, Association for Computing Machinery, 2010.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

[6] 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," *SIGARCH Comput. Archit. News*, vol. 33, p. 92–99, nov 2005.

[7] 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 *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 29–42, 2018.

[8] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the RISC-V ISA for GPGPU and 3d-graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 754–766, Association for Computing Machinery, 2021.

[9] M. Rekdal, "Investigating the performance scalability of the Vortex GPU," *NTNU Open*, June 2022.

[10] 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 *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[11] 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 *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 172–182, 2016.

[12] B. Gottschall, L. Eeckhout, and M. Jahre, "TIP: Time-proportional instruction profiling," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 15–27, Association for Computing Machinery, 2021.

[13] W. Snyder, "Verilator," 2022.

[14] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 15, p. 45–49, jan 2016.

[15] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure," 2019.

[16] S. Eyerman, W. Heirman, K. Du Bois, and I. Hur, "Multi-stage CPI stacks," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 55–58, 2018.