Masteroppgåve

NTNU
Noregs teknisk-naturvitskaplege universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Markus Rekdal

Investigating the Performance Scalability of the Vortex GPU

Masteroppgåve i Datateknologi
Rettleiar: Magnus Jahre
Juni 2022

NTNU
Kunnskap for ei betre verd

Markus Rekdal

Investigating the Performance Scalability of the Vortex GPU

Masteroppgåve i Datateknologi
Rettleiar: Magnus Jahre
Juni 2022

Noregs teknisk-naturvitskaplege universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Assignment Text

The objective of this master thesis is to work towards enabling an FPGA-accelerated evaluation infrastructure for Graphics Processing Units (GPUs) at NTNU's Computer Architecture Lab (CAL). We will use the Vortex soft GPU which was presented at MICRO'21 as our starting point. The main task is to get a collection of benchmarks to run on Vortex, evaluate their performance, and identify any performance bottlenecks. The candidate should then analyze one or more bottlenecks in depth and provide approaches for overcoming them. If time permits, the candidate should explore how Vortex can be extended to enable simulating systems with different amounts of memory bandwidth than what is available on the FPGA platform Vortex is run on.

Abstract

Software simulated computer architecture evaluation is slow, especially for large multi-core architectures. Using FPGA-accelerated evaluation one can bridge the gap between simulation and creating a prototype, enabling significantly more efficient evaluation than the software simulated alternative. To be able to evaluate multiple architectures well, one requires a good baseline, an example that performs well. Vortex [1], a RISC-V based software GPU initially seemed like a good candidate for a baseline for GPU evaluation.

In this thesis we will continue upon the work done in my project thesis [2] and take a closer look at Vortex. We will perform a thorough analysis of Vortex's benchmarks to figure out how to squeeze the most power out of it. We will attempt to reconfigure Vortex to give us better performance and take a closer look at why some benchmarks seem to have problems with performance scaling.

We found that Vortex's scheduler is at times unable to properly divide its work among the separate cores. This issue seems to be at least partially caused by the limitation of the compute kernel size to one. Adjusting the kernel size is shown to give a substantial boost in performance for one of the benchmarks. This and other issues that were discovered means that at the time of writing Vortex is perhaps not the best candidate for a baseline GPU.

Samandrag

Programvaresimulert datamaskinarkitektur-evaluering er treigt, spesielt for store fleir-kjerne arkitekturar. Ved å bruke FPGA-akselerert evaluering kan ein minske avstanden mellom simulering og prototyping, slik at ein kan oppnå mykje meir effektiv evaluering enn med det programvaresimulerte alternativet. For å kunne evaluere fleire arkitekturar på ein god måte, trenger ein ein kjent referansearkitektur, ein som ein veit korleis den kjem til å oppføre seg. Vortex [1], ein RISC-V basert programvare GPU såg innleiingsvis ut som ein god referansearkitektur for vidare GPU-evaluering.

I denne oppgåva skal vi fortsette på arbeidet gjort i mi prosjektoppgåve [2] og ta ein nærare kikk på Vortex [1]. Vi skal gjennomføre ei nøye evaluering av Vortex sine referanseprogram for å finne ut korleis ein kan skvise ut betre yting frå Vortex. Vi vil prøve å rekonfigurere Vortex til å gi oss betre yting og ta ein nærare kikk på kvifor enkelte av referanseprogramma ser ut til å ha problem med å skalere med arkitekturen.

Vi oppdaga at Vortex sin skedulerer av og til har problem med å dele arbeidet sitt mellom dei ulike kjernene. Dette problemet ser ut til å vere i allefall delvis forårsaka av at storleiken på utrekningskjernane er satt til ein. Vi viser at å justere storleiken på desse utrekningskjernane gir ei klar auke i ytinga. Dette og andre problem som blei oppdaga undervegs tydar på at Vortex kanskje ikkje er den mest eigna referanse-GPUen for vidare datamaskinarkitektur-evaluering.

Contents

Figures

Tables

Acronyms

AFU Accelerator Functional Unit. 9

AMD Advanced Micro Devices, Inc.. 6

CPU Central Processing Unit. 1, 5, 6

DDR Double Data Rate. 8, 22

DRAM Dynamic Random Access Memory. 8

FPGA Field Programmable Gate Array. v, vii, 1, 2, 6

GDDR Graphical Double Data Rate. 22

GPGPU General Purpose Graphics Processing Unit. 6, 11

GPR General Purpose Register. 8

GPU Graphics Processing Unit. v, vii, xi, 1–3, 5, 6, 29

IPC Instructions per Cycle. xi, 13, 15–25, 29

ISA Instruction Set Architecture. 6, 8

NoC Network on Chip. 11

NOP No Operation. 29

OpenCL Open Computing Language. 6–9, 11, 24, 25

OpenGL Open Graphics Library. 7

POCL Portable Computing Language. 8

RAM Random Access Memory. 2, 12, 22

RISC-V Reduced Instruction Set Computer, Five Stage. v, vii, 6, 11

RTL Register-Transfer Level. 9

SIMD Single Instruction Multiple Data. 1, 5

SIMT Single Instruction Multiple Thread. 6, 11

VM Virtual Machine. 7

Glossary

nearn A Vortex benchmark that calculates the nearest neighbours in a Cartesian coordinate system. xi, 12, 15–17, 21–25

psort A Vortex benchmark that sorts a list. xi, 12, 15, 16, 18–20

saxpy A Vortex benchmark that multiplies and adds a constant to each element in a vector. xi, 12, 16, 21, 22, 24

sfilter A Vortex benchmark that applies a 3x3 kernel to a matrix. xi, 12, 15, 16, 20

sgemm A Vortex benchmark that multiplies two matrices. xi, 12, 15, 18–20

vecadd A Vortex benchmark that adds two vectors. xi, 12, 15–18, 22, 24–26

Chapter 1

Introduction

1.1 Motivation

With the end of Dennard Scaling and the looming end of Moore's law, increased parallelism and higher core counts are now the most promising venues for achieving higher computing power. For this purpose we have accelerators, compute units that specialise in one specific domain of computing. The most common type of accelerators by far are GPUs. A GPU is a SIMD architecture that is designed to exploit the data level parallelism that is typical in the graphical world. GPUs are also often used for general purpose programming for large parallelised workloads.

GPUs achieve this by having drastically simpler compute units than CPUs such that they are both smaller in physical size as well as cheaper to produce. This allows for a core count potentially orders of magnitude larger than for a CPU allowing a drastically increased computational bandwidth.

Given a particular computer system, one will of course want to evaluate how efficient this system is and compare it to other similar systems. Simulating the systems on the same simulation interface will allow one to efficiently compare them against eachother. Simulation also allows for a large variety of abstraction levels, from cycle-accurate to more abstract models such as the OneIPC model [3].

Prototyping is also an option when one has to evaluate large workloads, as simulation tends to be quite slow, particularly for highly parallel workloads. However, creating a prototype is both far more time consuming and much more expensive.

FPGA-accelerated evaluation allows one to bridge the gap between a simulation and a physical prototype. It will typically be cheaper than creating a physical prototype, but will still allow for far greater speed than a simulation of the computer system in software. This means that evaluating a large heavily parallelised many-core workload efficiently can be both attainable and cheap. With cloud based FPGA-accelerated simulation interfaces such as Firesim [4], this can be scaled up to systems that needs several thousand cores.

In the project thesis [2] that precedes this thesis we analysed which of several software GPUs were the most suited for integration to a FPGA. We concluded that the most likely candidates were MIAOW [5] and Vortex [1]. MIAOW turned out to have issues regarding the use of licensed software that we were not able to attain. A move away from this to something open source, such as Verilator [6], would have required a large rebuild of the entire GPU and was deemed to be out of scope for this project. This left Vortex as the GPU that appeared most suitable for further analysis.

Vortex was thus deemed to be the sole GPU of those analysed that was feasible to integrate into a FPGA. The project then continued with a analysis of the working benchmarks included in vortex. From this analysis we could categorise the benchmarks by behaviour when changing the amount of cores and input sizes. In this thesis we will first do a more thorough behaviour analysis of the available benchmarks including what happens when the core structure is changed as well as the type and speed of RAM.

1.2 Assignment Interpretation

In this thesis we will continue on the work done in my project thesis [2]. The goal is to analyse what would make a GPU suitable for integration into a FPGA. We can define the following tasks based on our interpretation of the problem description:

- T1 Get a collection of benchmarks to run on Vortex and analyse their behaviour.
- T2 Identify and attempt to overcome at least one performance bottleneck in Vortex.
- T3 Explore how Vortex can be extended to use different amounts of memory bandwidth than what is available on the FPGA platform Vortex is run on.

It was however decided early on that actually getting Vortex to run on a FPGA would be out of scope for this thesis, which meant that task **T3** was no longer relevant.

1.3 Contributions

In this thesis we have made the following contributions:

- C1 An analysis of Vortex's working benchmarks and their performance. We found a large discrepancy in performance between the version of Vortex used at MICRO'21 and the updated version released on GitHub.
- C2 We identified the kernel size as a major performance bottleneck for some of the applications and showed that a substantial performance increase can be attained by adjusting it.

These two contributions directly correspond to tasks **T1** and **T2** respectively.

1.4 Thesis Outline

The outline for the rest of this thesis is as follows:

- Chapter 2 This chapter contains a short summary of other GPUs that could have been chosen for this project instead of Vortex.
- Chapter 3 This chapter contains an explanation of how Vortex was set up to allow the execution of the benchmarks.
- Chapter 4 This chapter contains a analysis of Vortex's benchmarks and how they perform, as well as a further analysis into some avenues for attaining a performance gain.
- Chapter 5 This chapter contains a conclusion as well as some ideas for further work that could have been included in this thesis if not for a lack of time.

Chapter 2

Background

2.1 GPUs

GPUs differ from CPUs mainly in their highly parallel architecture that allows a SIMD execution of large data sets. Their main commercial use is for graphical processing, as the SIMD execution is well suited for image and video rendering. However, GPUs are also useful for highly parallel workloads in general.

A simple GPU architecture is shown in Figure 2.1. A GPU differs from a CPU by having a significantly larger amount of much simpler cores. This allows for a highly parallel architecture enabling SIMD execution. GPUs trade latency for throughput, meaning that while computing a single instruction may take longer than on a CPU, computing a large batch of data will be faster on a GPU.

In this thesis we are mainly concerned with soft GPUs, that is GPUs that are implemented in software. While typically slower and less efficient than hardware implemented GPUs, soft GPUs are still useful as they are far easier to modify.

2.2 Soft GPUs

Here follows a short summary of some software GPUs that potentially could have been used instead of Vortex [1] in this thesis. Most of these were evaluated on whether they could be a candidate for this thesis in my project thesis [2]. For my

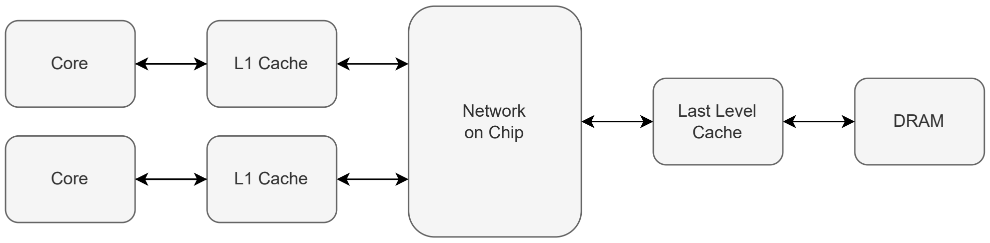

Figure 2.1: A simple GPU architecture.

project thesis the main factors to consider were whether they had available source code, such that they could easier be modified to suit our needs, whether they were fairly modern or not, and whether they used a custom software stack. When this evaluation was performed, the only GPU known to fit all three criteria was Vortex.

MIAOW [5] is an open source implementation of the AMD Southern Islands GPGPU ISA. It was made by the Vertical Research Group at the University of Wisconsin-Madison. It is capable of running OpenCL-based applications, and would thus be capable of using Rodinia [7] based benchmarks just like Vortex. Unfortunately MIAOW was too dependent on a closed source dependency that required licensing we were not able to get. For that reason it was deemed unsuitable for my project thesis.

Flexigrip [8, 9] is a software GPU that implements the NVIDIA G80 architecture. At the time of writing my project thesis it had already been abandoned for several years. It also used its own custom software stack, which would have made running custom applications on it substantially more cumbersome. For that reason it was deemed to be unsuitable for further use in my project thesis.

The **Pixilica** RISC-V GPU [10] is a GPU based on the vector extensions for the RISC-V architecture. It attempts to stay fairly close to a traditional RISC-V CPU and thus manages to maintain a fairly simple architecture. At the time of writing my project thesis Pixilica was still very early in its development, and as such no source code was publicly available.

Harmonica [11] is a configurable multithreaded SIMT processor core design. During the writing of my project thesis I was of the belief that the source code for Harmonica had been taken off the internet. However, the authors of Harmonica had simply moved it to a new repository with the release of the next major version, Harmonica2.

NyuziRaster [12] is a software GPU with rendering support. It is still being actively maintained despite being quite old even at the time of writing my project thesis. However, due to the fact that NyuziRaster had opted for using a custom software stack it was deemed to be unsuitable for use in my project thesis.

Simty [13] implements a specialised RISC-V architecture which supports SIMT execution. However, only the microarchitecture was implemented as a proof of concept. It also does not support any software stack.

FGPU [14] is a GPU-like architecture intended for use on FPGAs. While it originally used a custom software stack, it was updated to be able to run OpenCL applications not long before I wrote my project thesis. Unfortunately this was not known at the time, as it would have made FGPU a viable candidate for further evaluation.

2.3 Vortex RISC-V GPGPU

Vortex [1] is a RISC-V based general purpose GPU made by a team at the Georgia Institute of Technology. At the time of writing, Vortex was still in active development having originally been published for the MICRO 2021 conference. It supports

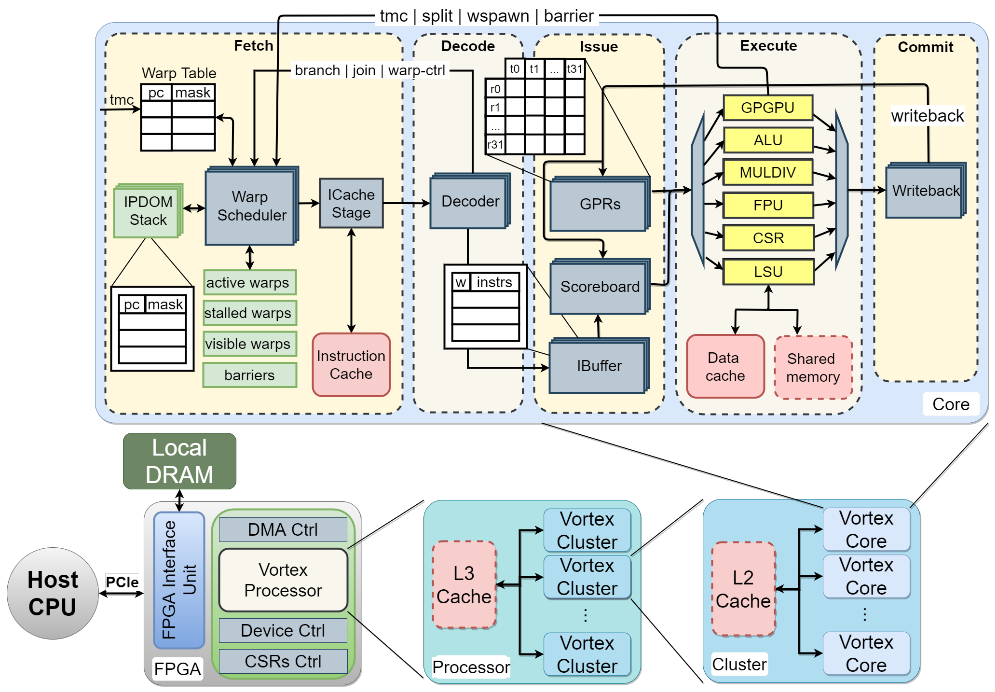

Figure 2.2: Vortex Architecture (Reproduced from [1]).

executing both OpenCL based applications and OpenGL based applications.

The setup used for Vortex is branched out from the upstream version of Vortex published on GitHub[1] on the 10th of January 2022. This downstream version can be found in the ntnu_main branch of the vortex-ntnu repository under the EECS-NTNU organisation on GitHub[2]. At the time of writing, the most recent version of Vortex released on GitHub was unable to build properly, which is why a slightly older version was used instead.

In my project thesis [2], the version of Vortex used was the one released for the Micro 2021 conference. This was a precompiled version packaged in a Vagrant Box VM. This version was used for my project thesis as the most recently published source code published at the time did not build correctly.

This version was not ideal as it was extremely unstable. Even the most minute of changes would break the precompiled builds, rendering it useless for a deeper analysis. For this reason a more recent build was used for this thesis, as a working build was now available.

2.3.1 Vortex Architecture

A detailed view of the Vortex architecture is shown in Figure 2.2. The Vortex microarchitecture implements the standard five stage in-order pipeline for each core

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

with some modifications. Each core has its own 16KB L1 data cache and 16KB shared memory. Several cores can then be joined into a cluster, which may have its own optional L2 cache. Finally, all clusters may share a common L3 cache, if this is enabled.

When Vortex is being simulated it will use Ramulator [15], a cycle-accurate DRAM simulator, to simulate its DRAM. This allows for a large amount of configurability in the DRAM setup. Ramulator allows the user to change the memory type, capacity, speed, layout and more. By default, Vortex uses 2400MHz DDR4 with 8 lanes of 4Gb for a total of 4GB of memory. This DRAM is shared between all cores in Vortex, independently of the how the cores are organised into clusters.

The Vortex microarchitecture also implements what they call a hardware wavefront scheduler. The team behind Vortex has not entirely decided whether it should name its schedulable units a wavefront or a warp. The Vortex team has mainly used the term wavefront in their papers [1] and the term warp in Vortex's documentation. For consistency we will use the term warp in this thesis.

The warp scheduler contains the program counter, thread mask register and an immediate post-dominator (IPDOM) stack. The purpose of the warp scheduler is to decide what should be fetched from the instruction cache. It has a set of warp masks to choose the next warp as well as a warp table that has information about the warps. The scheduler uses four thread masks to gain information about the warps. The active and stalled warp masks indicates whether a wavefront is active or stalled respectively. The barrier mask shows if the warp is waiting at a barrier instruction and the visible mask is used to implement a hierarchical scheduling policy.

The barrier instructions are used to synchronise warps. This is done by maintaining a count of how many warps still need to execute the barrier and keeping a mask of the warps that are currently stalled at the barrier in a barrier table.

The Vortex microarchitecture also makes use of banked GPRs to contain all of the GPRs for all threads in a warp. These can be found in the issue stage of the pipeline.

2.3.2 Vortex Software Stack

All of the benchmarks used for the analysis in this thesis contain a OpenCL kernel. Vortex uses the POCL [16] open-source framework to implement the compiler and runtime software for OpenCL. The POCL compiler that Vortex depends on has been modified to be able to generate kernel programs that target the Vortex ISA. The steps taken to generate a Vortex binary are shown in Figure 2.3.

All of the Vortex benchmarks mentioned in this thesis are based on OpenCL benchmarks from the benchmark suite Rodinia [7]. This means that getting Vortex to run on a larger set of benchmarks should in theory be quite achievable. However, given how many of Vortex's own benchmarks simply does not work, this might be more cumbersome in practise.

Vortex allows for the selection between several different simulation environ-

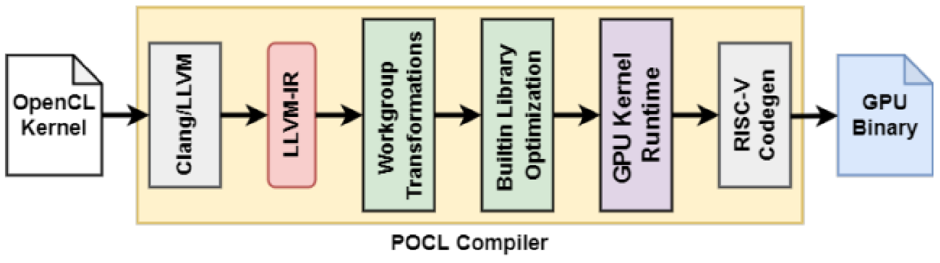

Figure 2.3: Vortex Binary generation steps for OpenCL applications (Reproduced from [1]).

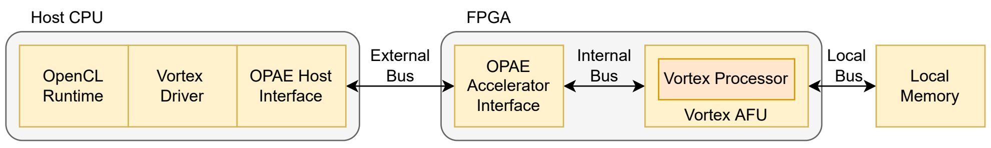

Figure 2.4: Vortex Driver stack (Based on a figure from [1]).

ments. The available simulation environments are the OPAE driver, which makes use of Intel's AFU Simulation Environment; VLSIM, which uses Verilator [6] to simulate the RTL design and implements the AFU interface and memory simulation in software; RTLSIM, which also uses Verilator for the RTL design, but does not make use of the AFU; and SIMX, which implements a cycle-level simulator for Vortex.

A simplified diagram of the Vortex driver stack when using the VLSIM simulation environment is shown in Figure 2.4.

Chapter 3

Experimental Setup

3.1 Vortex Configuration

The Vortex RISC-V GPGPU [1] was evaluated using its builtin blackbox benchmark setup script. This benchmark setup script allows the user to set several different parameters for the benchmark, such as the number of cores and threads, enabling level 2 or level 3 cache and more. It also allows for the selection of the simulation environment.

We chose to use the VLSIM simulation environment for this thesis as it would give a better simulation of the memory system, without necessitating the use of any proprietary software.

Sadly, a large amount of the OpenCL benchmarks made for Vortex does not work with the blackbox script. Getting the other benchmarks to work was not deemed to be a suitable use of time for this thesis as the main objective is to perform a performance analysis. For that reason, the working benchmarks were deemed to be enough.

Furthermore, a significant amount of the benchmarks have very small input sizes by default. The benchmark for Gaussian elimination for example, uses a 4

Table 3.1: Default Simulated 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 Max. 1024 threads (4 warps/core, 4 threads/warp)
Scheduler	1 warp 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 3.2: Overview of benchmarks used in this thesis.

Benchmark	**Name**	**Default Input**	**Adjusted Input**
Vector Addition	vecadd	64	64
General Matrix Multiply	sgemm	32	32
Nearest Neighbour search	nearn	42764	42764
Matrix Filter (3x3 kernel)	sfilter	16	256
Sorting	psort	16	1024
A Times X Plus Y	saxpy	1024	1024

by 4 matrix. While most of the benchmarks have easily configurable input sizes, the benchmark for Gaussian elimination did not. For this reason, this benchmark was not used.

This means that for many of the applications no scaling will be shown at all with the default input sizes, as most of the execution time will be spent on setting up the applications. An overview of the configuration used for Vortex when running is shown in Table 3.1.

The applications vecadd, sgemm, nearn, psort, saxpy and sfilter were ran using 1, 2, 4, 8, 16 and 32 cores. Additionally, vecadd and sfilter were ran with 64 cores. An overview of the benchmarks ran is shown in Table 3.2. In the third column of the table the default input size is shown. This is the input size the benchmarks are ran with by default. For some of the benchmarks this input size is too small to show behaviour representative of the benchmark. For these, the input size has been adjusted. This is shown in the last column. The only applications that had their input size adjusted were sfilter and psort, as they were both heavily limited by their default input size of 16.

An attempt was made to run the benchmarks with 64 cores. At this point, the runtime becomes significantly longer and the memory usage of the simulation increases substantially. More importantly however, not all of the benchmarks compute correctly when using 64 cores. Vecadd and sfilter are the only ones that are able to return the results they are supposed to, the other benchmarks all return incorrect results when using 64 cores. For this reason, the results of running the benchmarks with 64 cores have been omitted where most of the benchmarks are unable to make use of them.

3.2 IDUN Cluster

For the project thesis all the benchmarks were ran locally on my personal computer. While this is sufficient for small and simple benchmarks it quickly becomes untenable. In my experience, one will need roughly 0.5GB of RAM per core used in Vortex. This means that running benchmarks using more than 16 cores is more or less impossible when ran locally on my personal computer. Additionally, I was only able to run one benchmark at a time.

For this reason, we made use of the IDUN cluster [17]. IDUN allowed me to greatly expand my pool of resources as well as allowed the parallel execution of multiple benchmarks. To be more specific, Vortex benchmarks can be ran in parallel on the same installation of Vortex as long as they all use the same configuration for the architecture, that is, they all use the same amount of cores, they have the same amount of cache and so on.

Another requirement for this to be possible is that the benchmarks can not require compilation of any part of Vortex. For this reason, it is advisable to first complete a single benchmark for any given configuration of Vortex, before running all the others with the same configuration.

In order to run Vortex on the IDUN cluster one will need to modify the install script for the Vortex dependencies such that it does not attempt to install the dependencies into a restricted part of the system. This means that the user will also have to modify the paths to these dependencies in Makefiles throughout Vortex's source code.

3.3 Performance Metrics

The main performance metric used in this thesis is IPC. IPC is a measure of how many instructions are executed per cycle. This gives a simple view of how well the architecture perform for a given workload. In Vortex the IPC is calculated as the aggregated sum of the instructions executed by all cores divided by largest amount of cycles used by a single core. As we are mostly interested in how the performance changes when we modify parts of Vortex, the IPC is normalised to the base case in most instances.

The main reason for using IPC in this thesis is that IPC is the main performance metric used by Vortex itself. While IPC is a good indicator of performance, particularly for single core applications, it is not perfect. For multi core applications, it can in some circumstances be misleading or simply wrong [18].

The main criteria we will use to determine whether IPC is a good performance metric is that the amount of instructions needed to complete the benchmark is relatively unchanged. If the change in configuration changes the amount of instructions needed, then IPC will quickly become misleading. If one, for example, is able to achieve the same result in half the amount of instructions but the amount of cycles needed remains more or less the same, then using IPC will make it seem like the performance has halved even though this is probably a better result.

For this reason we will use a different performance metric in the cases where the amount of instructions will change substantially. This is only really the case for the experiments relating to the change in kernel size. Here we will use a performance metric which is calculated by $\frac{1}{CC}$ where CC is the amount of cycles needed to complete the benchmark. This will typically be a very small number, so the measure is normalised to the base case.

This is only a good metric if the workload is more or less constant. If one for example doubles the input size, this metric will show that the benchmark needed more time in order to deal with more work. While this is not wrong, it is not useful information.

Chapter 4

Results

4.1 Baseline Performance

The resulting normalised IPC of running the benchmark with a selection of applications with varying number of cores is shown in Figure 4.1. Note that the results for 64 cores is only available for vecadd and sfilter, as those are the only benchmarks that work with 64 cores. As can be seen from the figure, the different applications have a substantially different performance with a higher amount of cores.

The only benchmark that scales even remotely well is sgemm. Sgemm has a normalised IPC that scales almost linearly with the number of cores used. It only starts to lose ground at 32 cores. This shows that Sgemm is mostly compute bound.

When using the default input size on vecadd, Vortex is actually not able to fully utilise more than 4 cores. It instead gives all of the workload to four of the cores, leaving the remaining cores more or less idle. This makes sense given the fact that the default input size for vecadd is 64. With 4 cores and 16 threads per core, each thread ends up doing the vector addition for one element only.

For this reason, using more than four cores should not really effect the performance. What we actually observe though is a substantial decrease in IPC. This seems to be caused by a substantial increase in the time used to configure and start additional cores, meaning that this overhead starts to dominate much earlier than it should for any given input size. For vecadd in particular, having a rather light workload, this causes a large negative impact on the performance. Sfilter and psort show the same pattern when using their default input sizes, although they are only able to fully utilise a single core.

For the nearn application, the IPC does not seem to change at all with increased core count. This is actually for the same reason as for sfilter and psort with all of them only being able to utilise a single core. However, here the situation is somewhat different as nearn has a substantially larger default input size than the other benchmarks.

By default nearn tries to find the five nearest neighbours amongst 42700 data

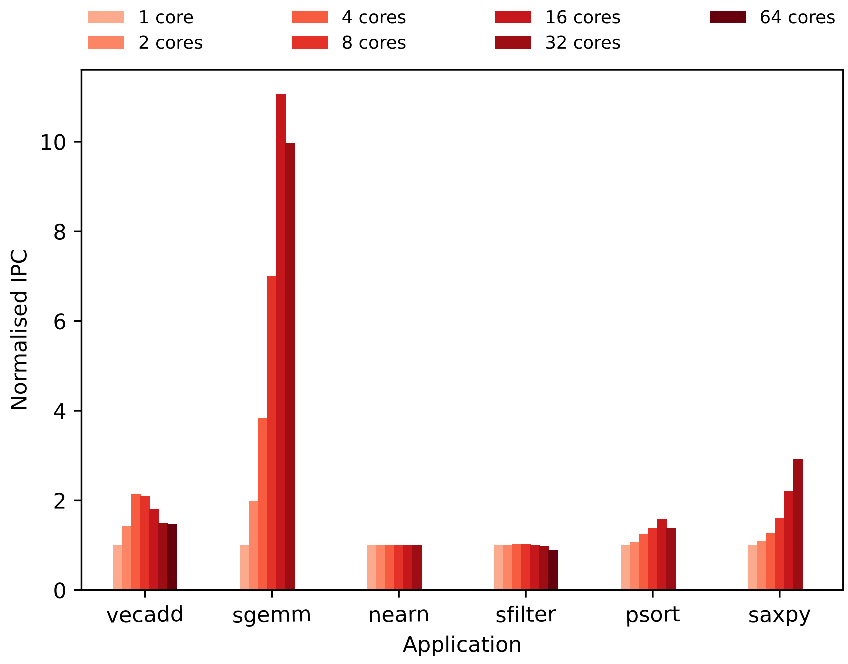

Figure 4.1: Baseline IPC for various benchmarks on current Vortex.

points. To do this nearn takes nearly 6 million cycles to complete. Most of the other applications that can only make use of one core will only need a couple of thousand cycles. This means that the setup and configuration of the superfluous threads will make up a far smaller part of nearns instructions than for the other applications.

Changing the setup of the cores, warps and threads can show that nearn is able to fully utilise 12 threads with the default input size. More specifically, with 4 cores that have two warps and two threads, cores one, two and three all have a utilisation of about 99.8% whereas core 4 has a utilisation of only 0.05% being only active for 3000 out of 5.7 million cycles.

Saxpy initially seems to behave somewhat similarly to vecadd. It initially scales somewhat worse than vecadd, but keeps scaling somewhat well with more than four cores. This is however extremely misleading. Saxpy will only make use of one core. The pattern that saxpy shows is actually similar to nearn. The increase in IPC that is shown for saxpy is due to it having a substantially lower input size and complexity, meaning that the IPC from the idle cores setting up and tearing down can influence the aggregated IPC.

A comparison with the equivalent results from my project thesis [2] where the same tests were ran on the version of Vortex used at MICRO'21 is interesting. These results are shown in Figure 4.2. Note that psort and sfilter were not ran with the default input size in my project thesis, instead using the adjusted input size shown in Figure 3.2.

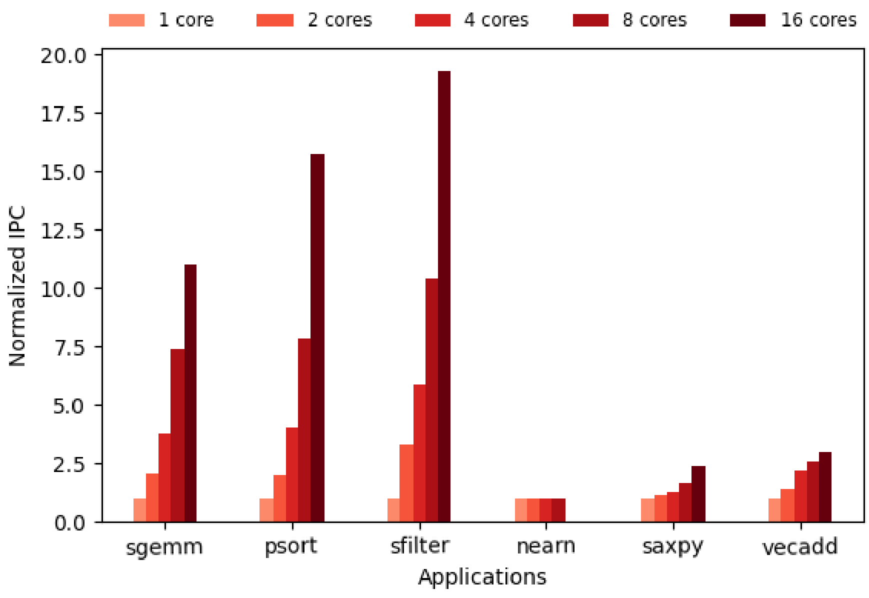

Figure 4.2: IPC for various benchmarks on Vortex from MICRO'21 (Repurposed from [2]).

Vecadd in particular seems to behave quite differently. The downward trend in IPC for higher core counts was simply not observed while using this version of Vortex. It does however remain unclear what exactly has been changed to cause this difference in behaviour.

4.2 Configuring Vortex

4.2.1 Adjusting Input Size

As shown in the previous section, several of the benchmarks are severely hampered by their extremely limited default input sizes. Thus, in order to properly demonstrate their complete behaviour we will explore what changes when their input size is increased.

Adjusting nearn's input size requires generating an entirely new data set for it to use. Doing this is not impossible but setting this up was not considered to be a valuable use of time, as the default data set containing over 40.000 records is in all likelihood sufficiently large enough to demonstrate how nearn will perform.

Vecadd

The results for changing vecadd's input size is shown in Figure 4.3. Vecadd maintains the same pattern as for the default input size throughout all the sizes tested here. The core count at which the IPC peaks does increase somewhat, going from

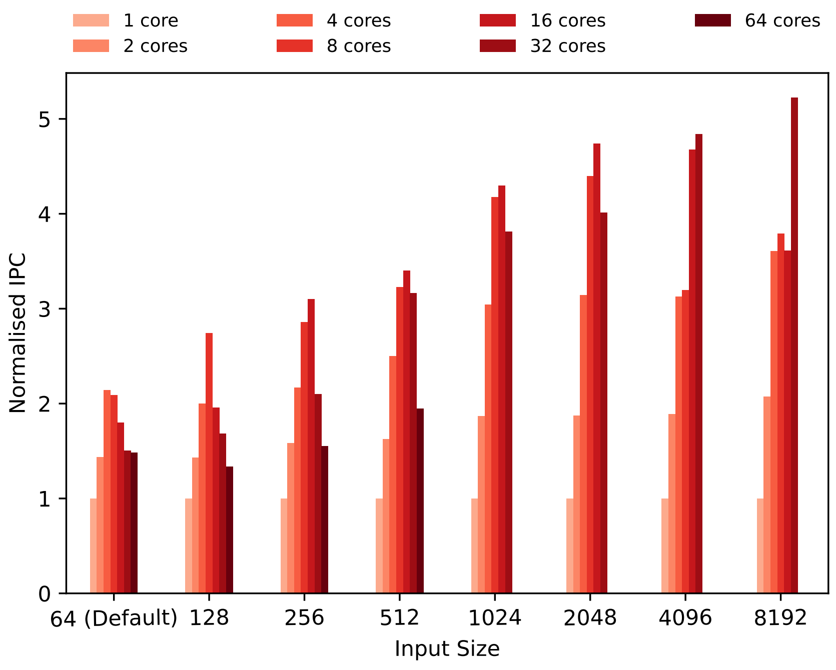

Figure 4.3: IPC for various input sizes for vecadd

4 cores at an input size of 64 to 32 cores at an input size of 4096. For an unknown reason, using input sizes larger than 512 with 64 cores causes an incorrect result to be calculated. For this reason, these results have been discarded.

Sgemm

The results for changing sgemm's input size is shown in Figure 4.4. Note that we were unable to complete sgemm with an input size of 256 with 32 cores. Sgemm scales better with an input size of 64 than it does for the default input size of 32. On the other hand, for higher input sizes than 64 it starts to scale worse with high core counts, having a substantial decrease for 32 cores and an input size of 128. This is presumably due to the matrix now filling up the cache, causing memory stalls that are detrimental to the performance.

Increasing the input size beyond 256 starts to be prohibitively resource and time intensive. This is due to the fact that the matrix multiplication algorithm used in sgemm is the naive $O(n^3)$ algorithm, which means that the simulation will also be $O(n^3)$.

Psort

The results for changing psort's input size is shown in Figure 4.5. For small input sizes, psort seems to behave similar to vecadd as shown in Figure 4.3. That is, the IPC increases up to a point, and then starts decreasing. However, this changes with

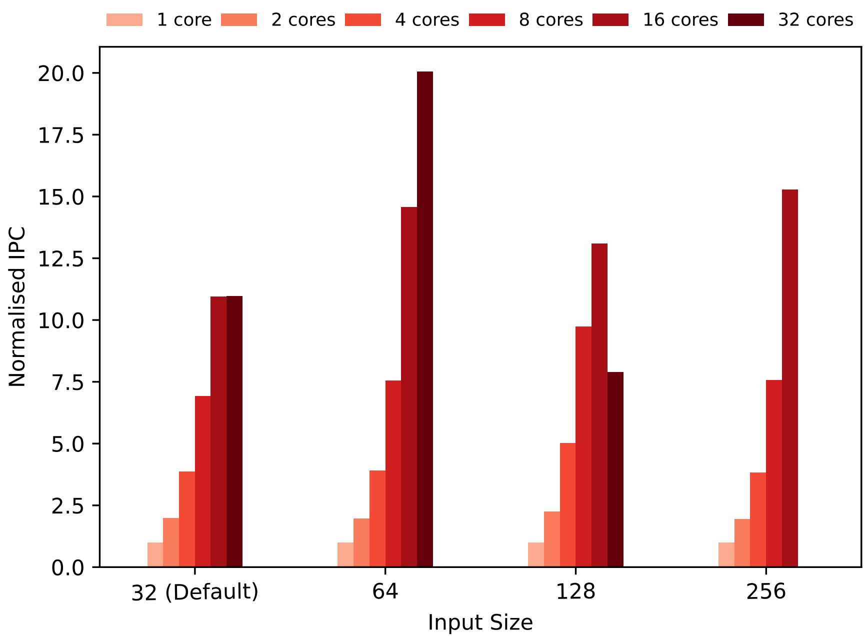

Figure 4.4: IPC for various input sizes for sgemm

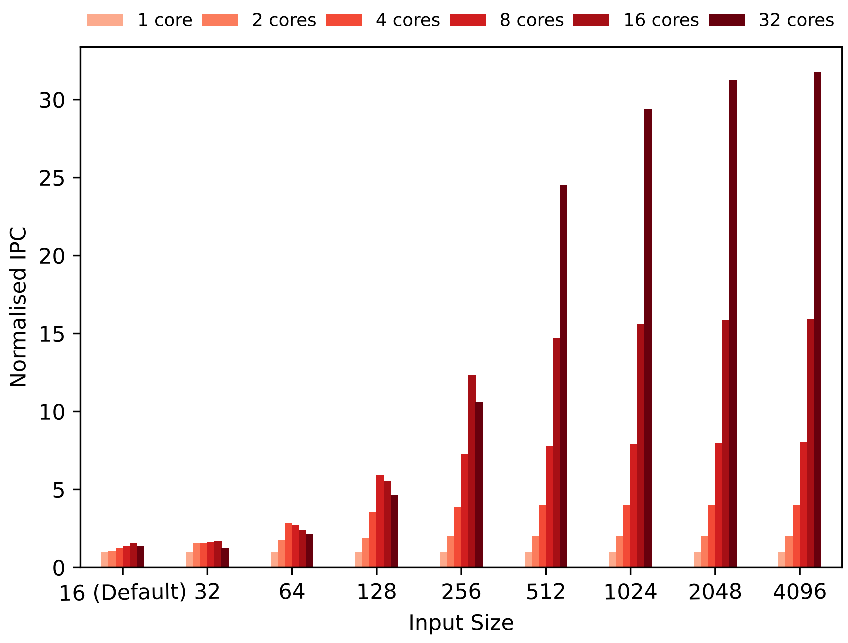

Figure 4.5: IPC for various input sizes for psort

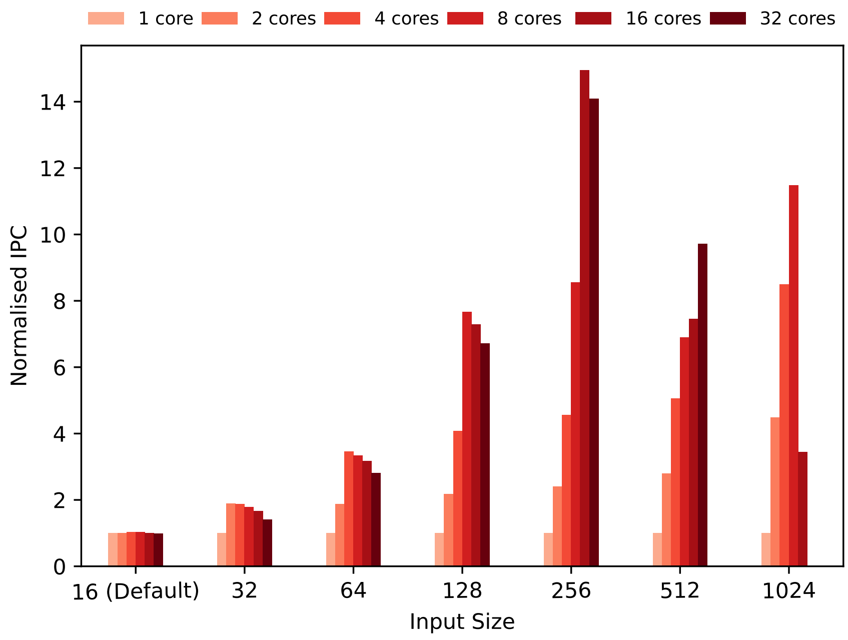

Figure 4.6: IPC for various input sizes for sfilter

an input size of 512, where the IPC keeps scaling linearly even for 32 cores. This is where psort goes from being synchronisation bound to being compute bound. This behaviour shows that psort is a compute bound application, like sgemm. Sadly, the psort benchmark does not work properly with 64 cores, so we are unable to test whether this pattern continues.

We can also see that the behaviour observed here is the same as the behaviour observed in my project thesis shown in Figure 4.2.

Sfilter

The results for changing sfilter's input size is shown in Figure 4.6. Note that we were unable to complete sfilter with an input size of 1024 with 32 cores. For small input sizes sfilter just is not able to make use of its resources well as can be seen for the input sizes up to 64. With a larger input size it initially scales well, before the performance diminishes greatly with an input size of 512 and 1024. This is presumably due to hitting some memory threshold as the matrix of size $512x512$ will take roughly $512 * 512 * 4B = 1MB$ of memory.

We can also see that the behaviour observed here is the same as the behaviour observed in my project thesis shown in Figure 4.2.

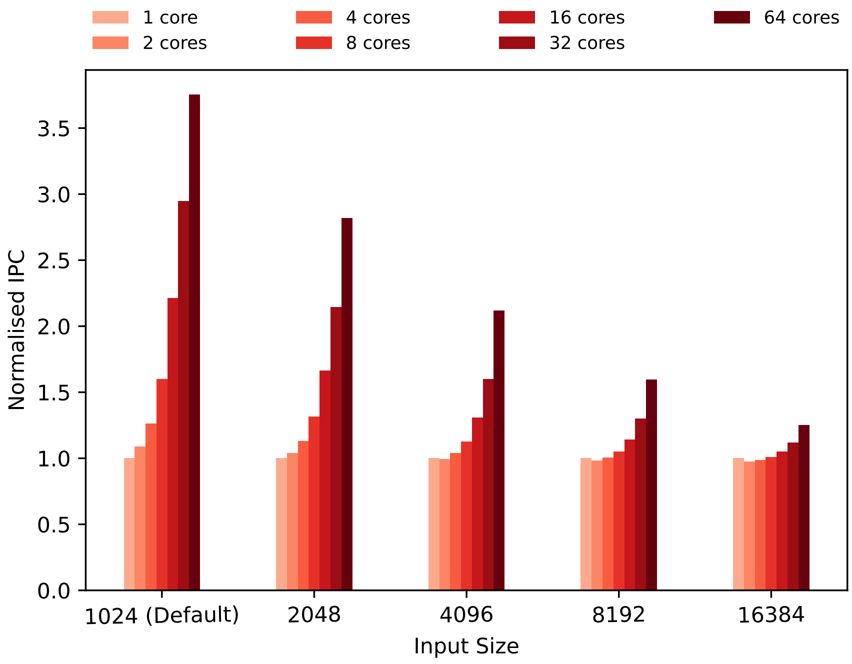

Figure 4.7: IPC for various input sizes for saxpy

Saxpy

The results for changing saxpy's input size is shown in Figure 4.7. Saxpy does not really scale all that well, it has a behaviour that actually is quite similar to nearn. It does not appear like that here though as saxpy terminates much earlier. This means that the cores that end up with no work, that only have a couple hundred instructions has a higher contribution to the aggregated IPC. As the program takes longer with a higher input size this effect diminishes, and thus we get the pattern shown in Figure 4.7. With a sufficiently large input size we would achieve the same pattern as shown for nearn in Figure 4.1.

4.2.2 Clustering

Vortex allows for a large variety of configuration in its core structure. A potential alley for performance increases is to enable the organisation of cores into clusters. Here we tested what change in performance we could achieve by having 1, 2 or 4 clusters when we are limited to a total of 16 cores. The resulting performance is shown in Figure 4.8. Rather disappointingly, the clustering of cores does not seem to affect the performance all that much.

Clusters in Vortex can optionally have a level 2 cache each, as well as a shared level 3 cache. An attempt was made to enable these to evaluate how they could affect performance. Unfortunately enabling the extra levels of cache does not seem to work correctly.

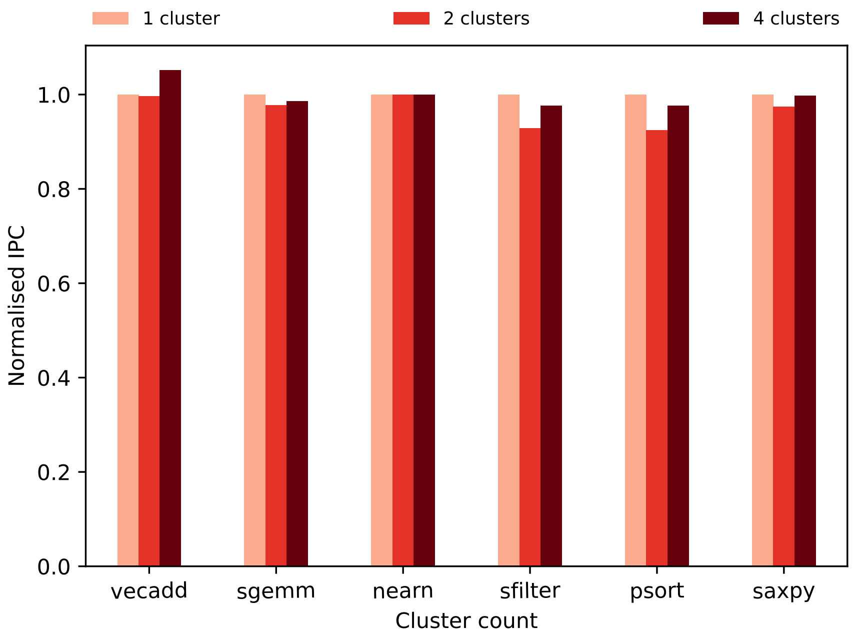

Figure 4.8: IPC for various core structures

4.2.3 Memory Configurations

Another potential avenue for performance increases is to modify the RAM configuration. Vortex uses Ramulator [15] to simulate its RAM. This allows us to very easily modify its parameters to simulate other setups By default Vortex is set up to have eight chips of 4Gb 2400MHz DDR4 RAM. By modifying this we can further evaluate what impact the RAM configuration has on Vortex's performance.

The impact of changing the memory configuration to GDDR5 is shown in Figure 4.9. Interestingly, doing this increases the IPC for all benchmarks except for nearn. However the extent of the performance increase varies wildly. Vecadd for instance has a 53% increase in IPC whereas saxpy only shows a 4.2% increase in IPC. This shows us that vecadd is far more limited by the speed of the memory than the other benchmarks.

4.3 Fixing Benchmarks

4.3.1 An Attempt at Fixing Nearn

As shown in Figure 4.1, nearn does not scale at all. It is entirely unable to utilise multiple cores. In the paper that introduces Vortex [1] this is explained by nearn having a square root in its core loop. While this square root does exist, there is no reason for it by itself to completely prevent any form of parallelism.

For this reason, we took a closer look at nearn's source code and found a minor

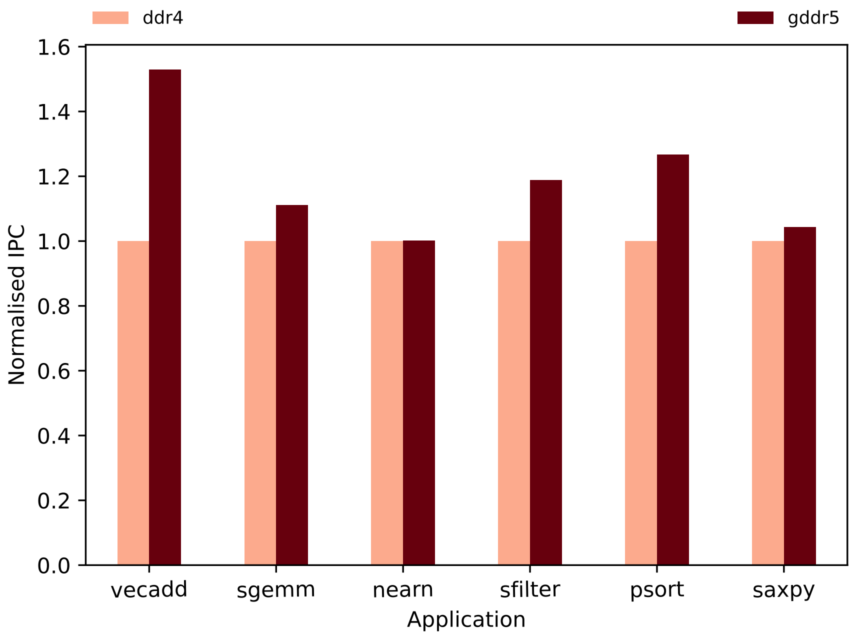

Figure 4.9: IPC for various memory configurations

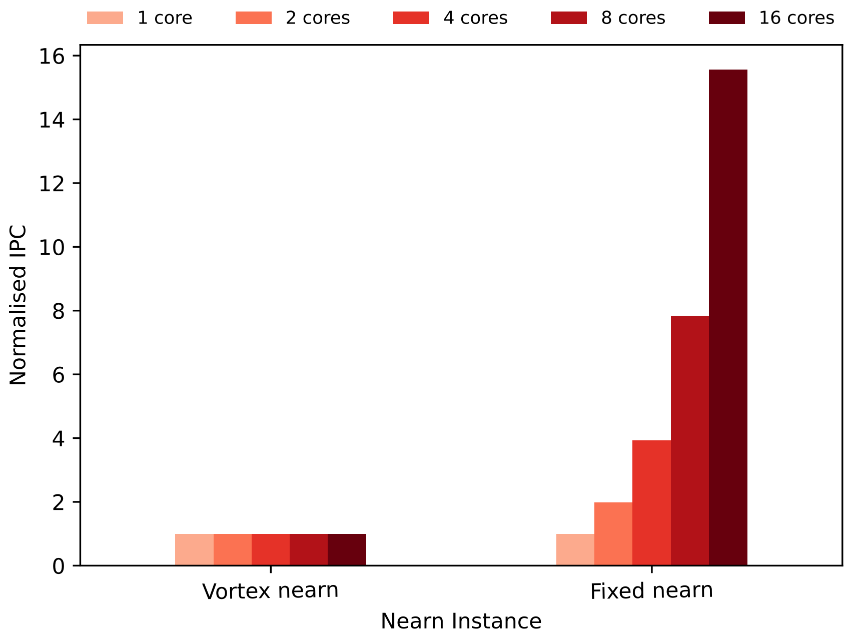

Figure 4.10: Comparison between Vortex's nearn and a fixed version

discrepancy in the way the OpenCL kernel was set up. In nearn's source code, the parameter local_work_size in the call to clEnqueueNDRangeKernel was set to null whereas in the other benchmarks it is set to one. This means that the size of the work group is left undetermined.

After aligning the OpenCL setup more towards how it is implemented in the other benchmarks, we found that nearn was indeed able to split up its workload to multiple cores.

However, at this point another discrepancy was discovered. For some indiscernible reason the reported order of the nearest neighbours were not the same for the two instances of nearn. Furthermore, the distances reported to the nearest neighbours were not calculated correctly for either instance of nearn. Somehow, the calculated distance is usually a bit wrong. The error is not very large though, so the five neighbours that the benchmark concludes are the closest are in all likelihood among the closest neighbours, but there is no guarantee that this is the case.

One of the reasons this was not discovered earlier is that while the other benchmarks all have some form of validation to ensure that the benchmark is correct, this is not the case for nearn. Nearn does appear to have some form of validation when running the benchmark, so the user has no real reason to suspect that something is amiss. However, when more closely inspecting the source code for nearn one will discover that the benchmark will always report that the results were correct and that the benchmark passed no matter what.

A comparison between the performance of Vortex's version of nearn and my fix is shown in Figure 4.10. While Vortex's version does not scale at all, the patched version scales more or less perfectly linearly with the number of cores.

A similar fix could also have been made for saxpy as it exhibits a similar behaviour. Taking a closer look at saxpy's source code reveals that it contains a similar error to the one found in nearn's source code. This was not done due to a lack of time.

4.3.2 Fixing Vecadd

The IPC scaling that is shown in vecadd is interesting. It does not really make sense for the IPC to decrease in the manner it does. This seems to be caused by the inclusion of extra cycles needed to set up and start all the extra cores. And as most of these cores are not able to be utilised fully, their individual IPC is very low. In total this will cause a decrease in the aggregated IPC. However, the fact that it causes such a large decrease means that for the higher core counts, the initialisation of the additional cores starts to dominate the total execution time. There is no reason for this to take that much time.

It is made even more interesting by the fact that this behaviour was not apparent in the version of Vortex used at MICRO 2021. If one compares the results shown in Figure 4.3 to the results shown in Figure 4.11 it becomes clear that something substantial has to have changed between the two versions. It is however still

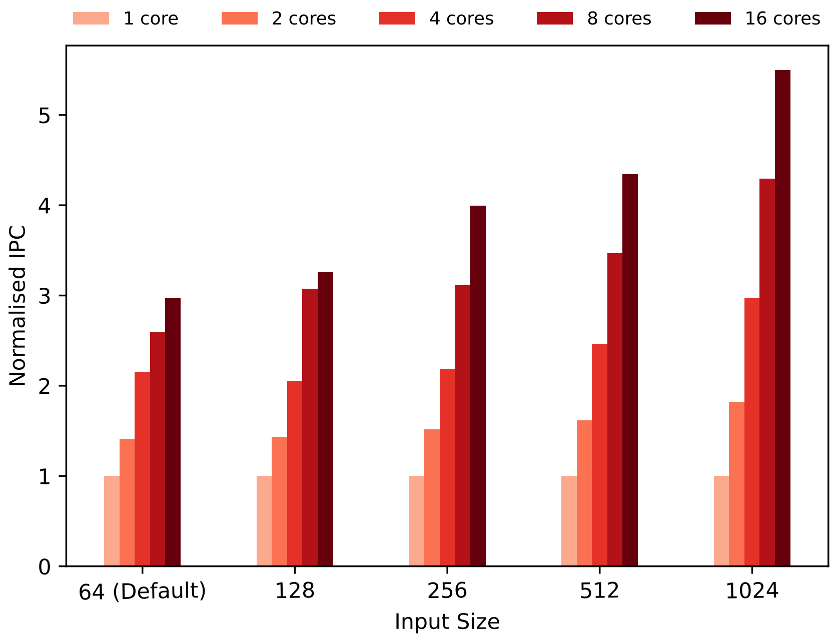

Figure 4.11: IPC for various input sizes in vecadd from MICRO'21

unclear exactly what this change is.

One potential issue that could cause such behaviour would be the fact that all the OpenCL benchmarks in Vortex are set up to use a kernel size of one. This means that the host only sends a single data point at a time. Adjusting this could then hopefully cause a significant speedup. A too large kernel size on the other hand, could cause some load balancing issues, resulting in an overall slowdown. As `vecadd` is the benchmark where these issues are the most apparent, this is where we will focus our efforts.

In both Figure 4.12 and Figure 4.13 the performance measure is $\frac{1}{CC}$ where CC is the amount of cycles the benchmark used before it completed. In Figure 4.13 Vortex was ran with 8 cores.

Interestingly, adjusting the kernel size does not increase the IPC but rather decreases it. While initially this might seem like a performance decrease has occurred, this is not the case. The reduction in IPC stems from a reduction in the number of instructions required to complete the benchmark. This shows that IPC is not always a great metric for performance [18]. In most cases, there is also a smaller reduction in the number of cycles needed to complete the benchmark as well. However, this is not universal.

This reduction in the amount of instructions needed seems to come from a reduced amount of overhead caused by the increased kernel size. The benchmark still computes the correct result, as unlike `nearn` it actually has proper testing. This means that the observed reduction in IPC is actually a performance increase.

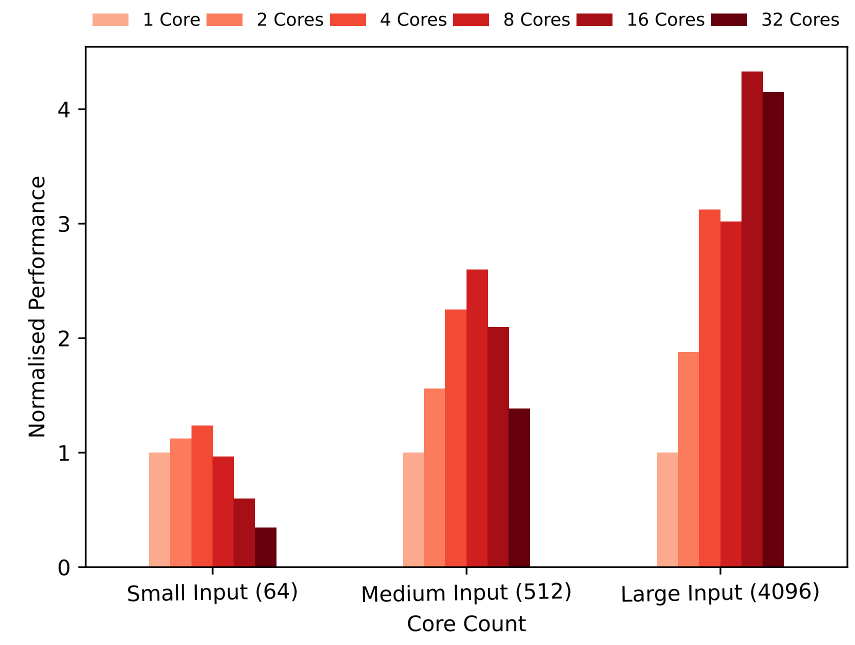

Figure 4.12: Performance for vecadd for various core configurations

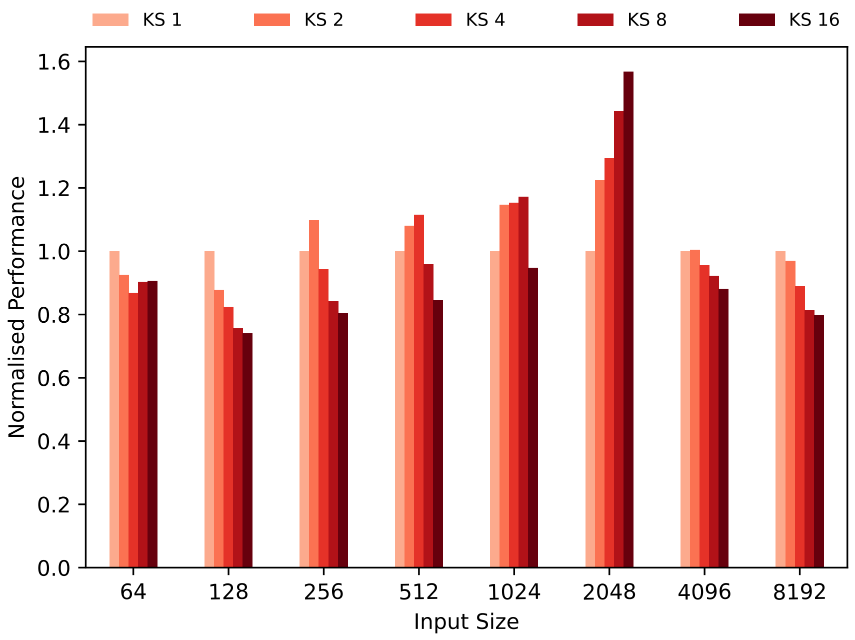

Figure 4.13: Performance for vecadd with varying input sizes and kernel sizes

As shown in Figure 4.13 adjusting the kernel size can give a substantial increase in performance even when disregarding the change in the amount of instructions. Interestingly, from an input size of 128 up to an input size of 2048 the ideal kernel size scales along with the input size, going from 2 to 16.

Chapter 5

Conclusion and Further Work

5.1 Conclusion

We performed an analysis of the available benchmarks for Vortex and found that most of them are not able to fully utilise the available resources. In some instances one can observe a decrease in IPC when increasing the core count. This comprises task **T1** which corresponds to contribution **C1**.

This seems to be at least partially due to the way that the scheduler hands out work to the cores. By default it is done one work item at a time. Adjusting this parameter gave a substantial decrease in IPC as the amount of NOPs decreased. This comprises task **T2** which corresponds to contribution **C2**.

While Vortex does have some issues, it is still in active development. Hopefully, most of the issues that have been encountered during this thesis can be resolved. Regardless, it remains a flexible and performant software GPU that is sure to aid many future research projects.

5.2 Future Work

We found that adjusting the kernel size can give a substantial boost to performance. However, in our experiments, the kernel size was always hard coded. One idea for future work could therefore be to calculate a good kernel size based on the specific Vortex configuration used, that is the number of cores, the numbers of threads and warps per core and so on, as well as the application itself.

All of the benchmarks used in this thesis were Vortex's own with some minor modifications. Said benchmarks are also all quite small. While they are all based on benchmarks from Rodinia [7], they are not representative of a typical complete GPU application. Thus, another idea for future work would be to get Vortex to run on a larger sample of more complete benchmarks from a well known benchmark suite, such as Rodinia.

Bibliography

[1] B. Tine, K. P. Yalamarthy, F. Elsabbagh and K. Hyesoon, 'Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics,' *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[2] M. Rekdal, 'Toward FPGA-Accelerated GPU Evaluation,' Norwegian University of Science and Technology, Tech. Rep., Dec. 2021.

[3] 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*, 2011, pp. 1–12.

[4] 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. Asanović, 'FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud,' in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18, Los Angeles, California: IEEE Press, 2018, pp. 29–42.

[5] V. Gangadhar, R. Balasubramanian, M. Drumond, Z. Guo, J. Menon, C. Joseph, R. Prakash, S. Prasad, P. Vallathol and K. Sankaralingam, 'MIAOW: An open source GPGPU,' in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2015, pp. 1–43.

[6] E. Schkufza, M. Wei and C. J. Rossbach, 'Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience,' in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 271–286.

[7] 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 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, USA: IEEE Computer Society, 2009, pp. 44–54.

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

[9] J. E. R. Condia, B. Du, M. Sonza Reorda and L. Sterpone, 'FlexGripPlus: An improved GPGPU model to support reliability analysis,' *Microelectronics Reliability*, vol. 109, p. 113 660, 2020.

[10] Pixilica. 'Pixilica RISC-V GPU.' (2021), [Online]. Available: `https://www.pixilica.com/graphics` (visited on 05/03/2022).

[11] C. D. Kersey, H. Kim and S. Yalamanchili, 'Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM,' in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, Alexandria, Virginia: Association for Computing Machinery, 2017, pp. 49–59.

[12] J. Bush, M. A. Khasawneh, K. Z. Mahmoud and T. N. Miller, 'NyuziRaster: Optimizing Rasterizer Performance and Energy in the Nyuzi Open Source GPU,' in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 204–213.

[13] C. Collange, 'Simty: Generalized SIMT Execution on RISC-V,' in *CARRV 2017: First Workshop on Computer Architecture Research with RISC-V*, vol. 6, Boston, United States, Oct. 2017, p. 6.

[14] M. Al Kadi, B. Janssen and M. Huebner, 'FGPU: An SIMT-Architecture for FPGAs,' in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16, Monterey, California, USA: Association for Computing Machinery, 2016, pp. 254–263.

[15] Y. Kim, W. Yang and O. Mutlu, 'Ramulator: A Fast and Extensible DRAM Simulator,' *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[16] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala and H. Berg, 'POCL: A Performance-Portable OpenCL Implementation,' *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, Aug. 2014.

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

[18] A. Alameldeen and D. Wood, 'IPC Considered Harmful for Multiprocessor Workloads,' *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.