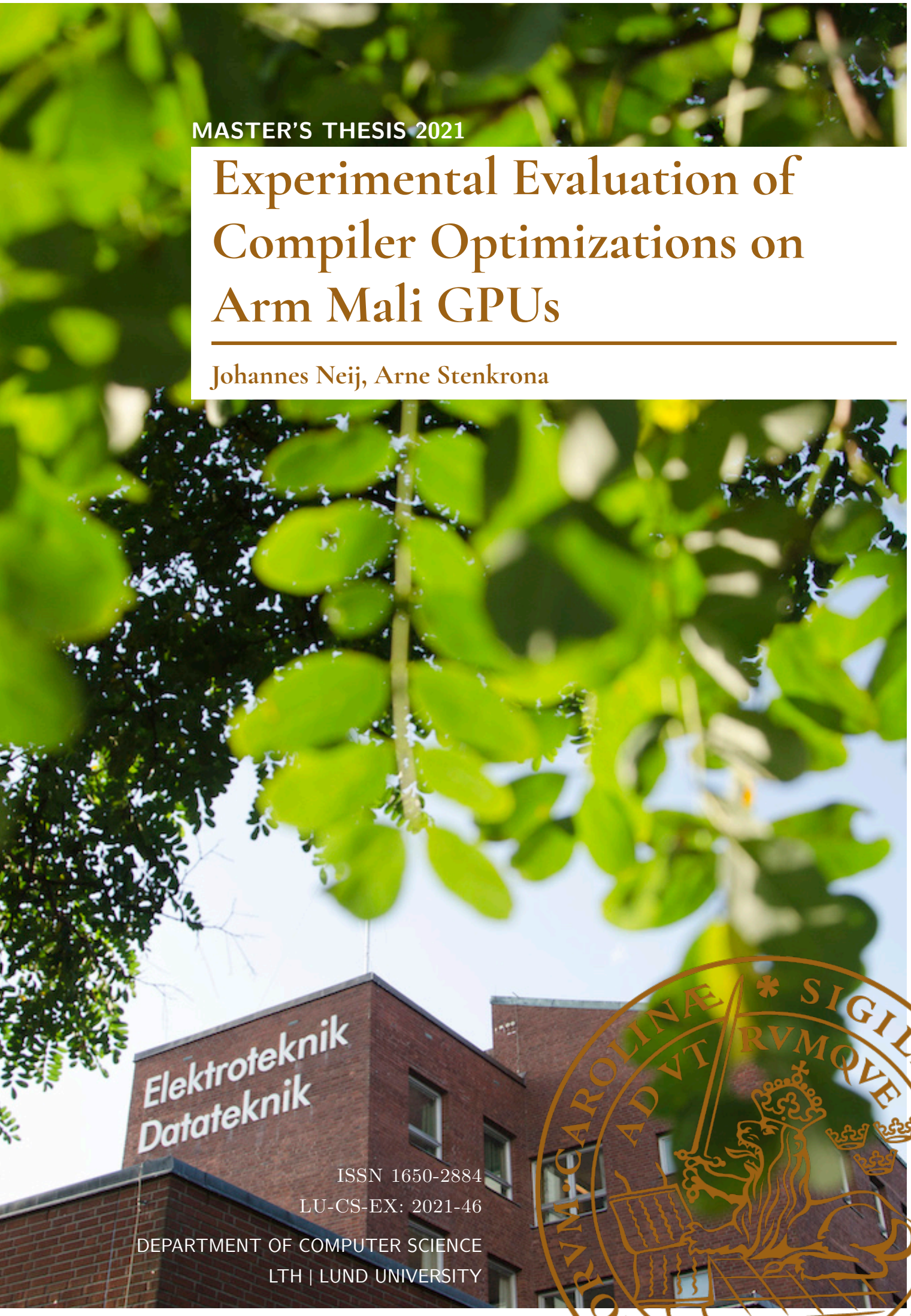


MASTER'S THESIS 2021

# Experimental Evaluation of Compiler Optimizations on Arm Mali GPUs

Johannes Neij, Arne Stenkrona

ISSN 1650-2884
LU-CS-EX: 2021-46

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-46

# Experimental Evaluation of Compiler Optimizations on Arm Mali GPUs

## Experimentell Utvärdering av Kompilatoroptimeringar för Arm Mali GPU:n

**Johannes Neij, Arne Stenkrona**

# Experimental Evaluation of Compiler Optimizations on Arm Mali GPUs

Johannes Neij
johannes.neij@gmail.com

Arne Stenkrona
arne.stenkrona@me.com

October 15, 2021

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Tobias Gutzmann, tobias.gutzmann@arm.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

**Abstract**

GPU compiler development is subject to shorter development cycles in comparison to traditional CPU compilers. This limits the amount of optimization passes that can be reasonably implemented in a GPU compiler. These limitations can be mitigated by estimating potential performance gains of various optimizations before actual implementation. We have evaluated the potential performance gains of peephole optimizations and time-unconstrained optimization. In the case of peephole optimization we found that high prevalence of underlying patterns seem to lead to more consistent performance gains. However, even in cases of low prevalence the performance gains can be higher than expected. In the case of time-unconstrained optimization heavily relies on the ability of static metrics to reliably translate to real world performance, which is not always the case. However, it can help identify possible areas of improvement such as ineffective passes and missed optimization opportunities. It can also help identify compiler errors.

**Keywords**: MSc, GPU, Arm, optimization, peephole optimization, superoptimization, iterative compilation

# Acknowledgements

We extend our sincerest gratitude towards our supervisor Tobias Gutzmann. Without his expertise and extensive help, both in technical matters and in writing, our work would not have been possible. We also offer a special thanks to our supervisor Jonas Skeppstedt, whose contagious enthusiasm for compiler technology motivated us to pursue a thesis within this subject. We also thank Simone Pellegrini at Arm for his great help in matters regarding the Mali compiler and time-unconstrained optimization. Finally, we thank Arm for providing the tools and technology that enabled us to do our thesis in collaboration with them.

Arne Stenkrona wishes to express his deepest thanks towards Disa Gustafsson for her loving support and helpful discussions without which this thesis would have greatly suffered. He also expresses gratitude towards his family who have stood behind him in all endeavors.

Johannes Neij would like to express his gratitude to Dijana Aleksić, whose love and support helped him stay motivated and focused. He would also like to thank his family for all of their support.

# Contents

# Chapter 1

# Introduction

Computer graphics has always been a computationally expensive task. GPUs were introduced in order to achieve acceptable performance for real-time graphics rendering by utilizing specialized hardware. Early GPUs were typically fixed function, forcing developers to rely on intrinsic hardware behavior. In order to accommodate a higher degree of control modern GPUs have long provided programmable behavior through shader programs [1]. Shaders are programs that allow developers to perform general computations on GPUs. With the advent of shader based architectures the need for GPU compilers has arisen. When attempting to achieve high performance for high intensity tasks, compiler optimizations are of great importance. This thesis hopes to give further insight into which compiler optimizations matter for typical GPU applications on Arm Mali GPUs.

## 1.1 Problem

GPU compiler development occurs within a much smaller time frame compared to traditional CPU compilers. A particular CPU architecture will generally have a long life time without drastic changes. This gives developers time to implement many different optimization techniques, allowing the compiler to mature. GPU architectures, on the other hand, change frequently. For example, since 2007 Mali, an Arm series of GPUs, has gone through the Utgard, Midgard, Bifrost, and Valhall architectures[2][3]. The limited window of time reduces the number of optimizations that can be implemented. Furthermore, since most GPU architectures are proprietary compiler development does not have a large open source community at its disposal. Given limited scope in both time and developer resources each GPU generation risks missing out on significant performance gains. Therefore it is desirable to estimate potential performance gains in advance in order to scope out promising avenues of development.

GPU compilers are further constrained by strict compilation time requirements. Pre-compiling before application deployment is unfeasible as the end-user GPU hardware and driver is not known beforehand. Instead, compilation occurs on the end-user's device during application runtime. In order to maintain acceptable responsiveness fast compilation times are very important.

## 1.2 Goals

The main goal of this thesis is to investigate potential avenues for performance gains in the Mali GPU compiler. We have approached this goal with two different strategies. Firstly, we have investigated peephole optimization. Since sub-optimal instruction sequences are frequently detected it is useful to gauge the pay-off of replacing these sequences with better sequences. Secondly, we have investigated time-unconstrained optimization to find improvements that the current production compiler has not considered. If gains are significant it can show compiler developers where to focus their efforts.

## 1.3 Research Questions

In order to achieve these goals we have attempted to answer the two following questions:

1. How is the performance of typical GPU applications impacted by various peephole optimization on Arm Mali GPUs?
2. How can we improve performance for Arm Mali GPUs if we are not constrained by compilation time?

## 1.4 Previous/Related Work

The work we have performed covers peephole optimization, iterative compilation, and superoptimization. These topics have been explored in literature before. We outline the previous work that has been performed in these areas below and how this relates to our approach.

### 1.4.1 Peephole Optimization

Peephole optimization was first introduced in 1965 by McKeeman [5]. Since then, different techniques have been used when implementing peephole optimization. It is widely used in modern optimizing compilers [6].

In McKeeman's original introduction of peephole optimization he notes that compilers often emit redundant instructions. He found that local inspection is often enough to remove many redundancies. By going through instructions one by one and applying rule based logic on one or two preceding instructions the final code size can be reduced. For example, if a constant is loaded twice into the same location one instruction can be omitted. He concludes that a narrow peephole is sufficient for considerable improvements in object code.

McKeeman leaves his definition of peephole optimization quite open to interpretation. In summary, his approach is to apply a set of rules to each sequential group of 2-3 instructions and in case of a match perform necessary modifications to the object code.

Since McKeeman's original article the technique has been widely used and studied. Chakraborty highlights its widespread usage in compilers in his summary [6] of the history of peephole optimization. He notes that peephole optimizations have succesfully been used to eliminate redundant instructions, improve control flow, perform algebraic simplification and enable better use of machine idioms.

Our peephole optimizer is implemented using an existing pattern matching framework developed by Arm. The framework was developed specifically for detecting programmer specified patterns within assembly instruction sequences. The pattern matching was used by our peephole optimizer to detect sub-optimal instruction sequences. We have extended the framework to be able perform peephole optimization on detected sequences. The pattern matching method used in this thesis is more sophisticated than the one originally described by McKeeman. For example, the pattern matcher can utilize semantic knowledge of the program. It may also match sequence of arbitrary length that do not need to be contiguous.

Aktolga [7] has given a thorough investigation of pattern matching for peephole optimization. While we are unaware if the pattern matching framework made at Arm, that is used in this thesis, is inspired by Aktolga's work, there are striking similarities. In short, Aktolga divides peephole optimization into two parts, pattern matching and replacement. The pattern matching finds sub-optimal sequences which are then replaced with a better sequence with the same behavior.

The work we have performed involves adding a replacement part to the existing pattern matching framework. This stems naturally from the purpose of peephole optimization, that is the replacement of sub-optimal instruction sequences. As such, there are similarities to the methods described by Aktolga though we have not based our approach on his work.

Because peephole optimization is used in order to achieve better target code it is natural to perform it as an optimization pass within a compiler. Since we have implemented our peephole optimizer using a pattern matching framework that works as a stand-alone process we are instead performing optimization outside the compiler. We have not found any other work on peephole optimization performed independent of the compiler tool-chain.

### 1.4.2 Time-unconstrained Optimization

Superoptimization is not quite as old of an idea as peephole optimization. It was introduced in 1987 by Massalin [8]. The idea was to find the optimal version of any given code sequence, with the limitation of being very slow if attempted at longer code sequences. Equivalence between sequences was tested with a probabilistic test that aimed to find edge cases and fail the generated code as fast as possible. This was an improvement from the first strategy proposed in the same paper, which they described as boolean verification. Boolean verification meant comparing the boolean minterms of the generated and original code. This was not a fast process, and other papers have tried to optimize the speed of it, such as *Probabilistic verification of Boolean functions* [9]. The output code of the superoptimizer was often very convoluted, but it

was able find some large improvements. This superoptimization was performed on machine code.

In 1992, inspired by the aforementioned paper, Granlund and Kenner also created a superoptimizer and used it to eliminate branch instructions with *gcc* [10]. The idea behind this was to eliminate as many jump instructions as possible, since they were considered expensive. A limitation for superoptimization noted by the researchers was the large search space which was approximately exponentially in relation to the instruction sequence length.

In their 2002 paper, Knijnenburg et al. implemented and discussed a different time-unconstrained optimization technique, iterative compilation [11]. Iterative compilation did not aim to produce the optimal code, but rather to produce well optimized code within a more reasonable time frame. Optimal code is very difficult to produce, and it is not time efficient to do so. The ideas towards iterative compilations provided in this paper provide many of the general concepts regarding iterative compilation used in this thesis.

In more recent times, a superoptimizer has been developed by a group of google employees[4][12]. In their paper, Phothilimthana et al. discuss strategies for pruning the search space for a superoptimizer [13]. This superoptimizer will be directly used in this thesis, and is in part a product of the previously mentioned research in superoptimization.

## 1.5 Contributions

Arne: abstract, 1.1, 1.2, 1.3, 1.4.1, 1.5, 2.2, 2.3, 3.1, 3.3, 4.1, 5.1, 6.1, 7, A.1

Johannes: abstract, 1.1, 1.2, 1.3, 1.4.2, 1.5, 2.1, 2.4, 3.2, 3.3, 4.2, 5.2, 6.2, 7, A.2

# Chapter 2

# Theory

The goal of this chapter is to introduce the necessary theory for understanding the approach, implementation and motivations for choices made in this thesis.

## 2.1 Architecture

Mali is a group, or series, of GPUs made by Arm. The product series has contained multiple different *micro architectures* and *instruction set architectures* (ISAs) with the most recent one being Valhall.

The work in this report was carried out on the Mali G78 GPU, a Valhall architecture based GPU. This is the most recent premium model GPU out of the Mali series [14], and as such, the most representative of the challenges faced at Arm regarding graphics optimization.

The Valhall architecture uses a processing engine that is divided into three arithmetic processing pipelines [15]:

- The FMA (fused multiply accumulate) pipeline, which handles complex math operations.
- The CVT (conversion) pipeline, which handles simple math operations.
- The SFU (special function) pipeline, which handles special functions.

The SFU pipeline has a 4 times lower throughput than the other two pipelines [16]. In Valhall, the number of threads that a program can use is dependent on the number of registers used for a shader. The registers are 32 bit, and up to 64 of them can be used. However, 32 registers is the maximum number that can be used while fully utilizing all threads [15].

## 2.2 Shader Programs

Shaders programs are used to give programmers control over GPU hardware[17]. In typical GPU applications these programs determine the visual output of the application. In order for shader code to run on a GPU it needs to be compiled to suitable object code. Due to wildly different target hardware, as well as the need to procedurally generate shader code during application runtime, shader code is typically not pre-compiled[18][17]. Instead, during application runtime, the GPU driver will compile shader source code written in GLSL ES [19]. Realtime rendering applications often prioritize responsiveness, which puts time constraints on the compilation. As such, compiler optimization passes must be chosen carefully.

## 2.3 Peephole Optimization

In this section we will introduce the basics of peephole optimization. We outline the role of pattern matching within peephole optimization. We also highlight the benefits and drawbacks of compiler independent peephole optimizations.

### 2.3.1 Background

Generated code will likely never be fully optimal despite a compiler's best effort. While it may be too late to employ most *optimization passes* once the compiler has selected instructions for the program, some optimizations are still possible. By searching through the object code it is possible to detect sequences of instructions which are known to be sub-optimal. These can be replaced with an equivalent, yet closer to optimal instruction sequence. What is more optimal may be determined by sequence length, power consumption, or perhaps another desired metric. This type of optimization is called peephole optimization as it uses a limited view of the code, as seen through a peephole. The fast implementation and the decoupling from prior optimization passes, allowing optimization to be performed in parallel on different parts of the code, contribute to the appeal of peephole optimization.

Chakraborty [6] has summarised the history of peephole optimization for the interested reader.

### 2.3.2 Pattern Matching

Peephole optimization can be thought of as consisting of two parts. The first involves finding a sub-optimal instruction sequence. The second part is to replace it. In this thesis we will use an existing pattern matching framework developed by Arm to perform the first part.

Pattern matching, in the context of peephole optimization, involves tokenizing an assembly file and detecting patterns in these tokens. Spinellis, for example, uses three token classes: labels, branch instructions, and other code [20]. As we will see in our approach in chapter 3, pattern matching offers us a great deal of flexibility. This flexibility means that the instruction sequences that a single pattern can match may differ in many ways. Therefore, a pattern matching approach demands flexible replacement capabilities.

As mentioned in chapter 1, the pattern matching framework was not created by us. It exists as a internal tool at Arm. Our work has been concerned with adding instruction replacement functionality. Furthermore, the pattern matching capabilities were insufficient for all of our needs and we have worked to extend it. These extensions consists of the addition of new *initial patterns*. Initial patterns are used to find candidate instruction sequences that may or may not correspond to a desired pattern. These are outlined in more detail in chapter 3. The work also involved the implementation of specific patterns, described further in chapter 4.

### 2.3.3 Compiler Independent Peephole Optimization

Because of the local nature of peephole optimization it can be implemented independently from the rest of the compiler. The benefits from this are twofold.

Firstly, it is typically very demanding to implement new passes within an optimizing compiler. By implementing peephole optimization as a stand-alone process after compilation has finished we circumvent a lot of complexity inherent to compiler development.

Secondly, because peephole optimization is able to operate on compiled object code the performance gains may be measured reliably. Small modifications to optimization passes within the compiler will likely produce cascading changes leading to vastly different object code.

For example, consider the pseudo-code in listing 2.1 and assume that `r0` may be used later in the program. A compiler may wish to improve it by changing the conditional operation and removing the second instruction as seen in listing 2.2. This frees up a register and therefore changes register pressure which may greatly affect later decisions the compiler takes.

```
1  r0 = x + 2
2  r1 = r0 + 2
3  ...
4  cond = r1 > 4; last use of r1
```

**Listing 2.1:** A simple code sequence.

```
1  r0 = x + 2
2  ...
3  cond = r0 > 2
```

**Listing 2.2:** An optimized version of the code in listing 2.1.

This difference in decision-making can be seen in compilers using an LLVM tool-chain [21]. LLVM uses an *intermediate representation*, or IR, for code transformations but must translate this into the target code language during code generation. The code generator is given an overview in LLVM's official documentation. In summary, IR are translated to the target instruction set in a process called *instruction selection*. Although the program is now expressed in the target language, these instructions use virtual registers of which there are an infinite

number. The instructions are then scheduled, meaning that their order is determined. At this point the compiler may perform machine code optimizations, such as peephole optimizations. Ultimately, the virtual registers have to be replaced with hardware registers. This happens in the subsequent *register allocation* step. It might be impossible to fit all virtual registers into hardware registers while satisfying variable liveness constraints. As such, register allocation introduces register spilling, meaning the compiler has to introduce store and load instructions here. After this step it is may perform late machine code optimizations and finally code emission.

As we can see, if peephole optimization is implemented before register allocation there are still many decisions for the LLVM back-end to take before code emission. These decisions may be greatly affected by earlier steps in the compilations. Minor differences before register allocation may cascade into more major difference later in the compilation process.

These cascading effects makes it difficult to reliably assess code improvements in isolation. In contrast, if peephole optimizations are not followed by any further optimization passes we avoid cascading effects. Measuring improvements is therefore fairly straight-forward. If we assume that the cascading effects will on average have a neutral effect on performance a stand-alone peephole optimizer offers a way to estimate performance gain of a compiler implemented peephole optimization. That is, we assume that any trends that we discover using our stand-alone approach would present in a compiler implementation, albeit with more noise.

An obvious downside of using a stand-alone peephole optimizer is the fact that it cannot realistically be used in a production setting. This is because a stand-alone process complicates the work-flow and introduces unnecessary performance overhead. Such a process will only be used for assessment of potential performance gains. For the purposes of our thesis a compiler independent peephole optimizer is ideal. Since we are mainly interested in investigating potential performance improvements we are not concerned by the inability of compiler integration.

### 2.3.4 Importance of Peephole Optimization in Mali GPUs

Using a pattern matching approach to peephole optimization it is straightforward to see how often a pattern occurs in an application. It is simply a matter of checking the number of occurrences within all shaders that are used by the application. However, it is not as simple to determine the performance impact of the corresponding peephole optimization of such patterns. In addition to the cascading effects mentioned in the previous section, GPU performance is also largely affected by texturing operations and memory accesses.

In Mali, cost of texturing operations will depend texture format and filtering mode [22] which can be changed during runtime through the graphics API [19] [23]. A memory access may only require a single instruction to execute but due to latency it can take hundreds of cycles before the result is available. GPU hardware hides latencies from such operation by running many threads in parallel, enabling switching of threads when such operations causes a stall [24]. This latency hiding is also subject to the dynamic behavior of the GPU. It is therefore

difficult to know the true performance impact of memory accesses and texturing. The relative impact of reducing arithmetic instruction count compared to memory accesses and texturing is therefore also difficult to ascertain. It is simply not yet known if peephole optimization is important for performance in Mali GPUs.

## 2.4 Time-unconstrained Optimization

In this section the theory behind iterative compilation and superoptimization is explained, as well as some background as for why these techniques are relevant for graphics content.

### 2.4.1 Background

When compiling code, one of the variables that needs to be accounted for is time. There are benefits to having a compiler spend more time optimizing a code sequence, such as improved *execution time*, or *memory footprint*. However, in practice, there needs to be a limit to the amount of time a compiler can spend. This is even more so true for graphics compilers, such as for those in the Arm Mali GPU series. GPUs need to compile *shaders* during application runtime, meaning that compilation time is a vital component to performance of the system.

Time-unconstrained optimization is, as the name suggests, conceptually unbounded by compilation time. As compilation time is vital, this can not be used for actual compilation in GPUs. However, time-unconstrained optimization can still be utilized to obtain information about optimization pass behavior and finding good *optimization pass sequences*. This can give insight into possible future performance gains. Time-unconstrained optimization can come in multiple forms, such as iterative compilation or superoptimization.

### 2.4.2 Iterative Compilation

The goal of iterative compilation is to compile the same code multiple times, with differences each time based on feedback from previous iterations. An iteration could be the generation and evaluation of one selection of compiler parameters. In other words, an iterative compiler modifies things around the compiler, and then compiles the code multiple times in order to generate better code. Iterative compilation gives no guarantee that it will find the optimal version of the code. Iterative compilation needs to use an algorithm in order to generate improved code. This can be done in different ways, for example using *Markov chains*, or *genetic algorithms* in order to drive the iterative compiler forwards. What metrics are used for evaluation can vary. Execution time, memory usage, code size or other *static metrics* are common. The idea is often to find sequences similar to those that have yielded the best results previously and have these impact the next iteration, thus moving forward.

Iterative compilation does introduce the risk of finding *local optima* in the search space as the there are no guarantees that the search tends towards the optimal solution. Different algorithms that are used to implement iterative compilation attempt to mitigate or avoid this in different ways.

### Markov Chains

Markov chains are a type of *Markov processes* [25]. The defining feature of a Markov process is that the next future state only depends on the current state. This means that it has no direct memory of previous states and does not directly use them in order to calculate the future state. Markov processes depend on time, either continuous or discrete. The *state space* can be either continuous or discrete. A Markov process is a Markov chain if the state space is discrete. The transition between states is done by modeling using a *matrix of transition probabilities*. Within the context of an iterative compiler, each state could be seen as an iteration. Due to the inherent randomness of probabilities, the exact behavior of the chain at a given time can not be predicted. In regard to the problem of finding optimization pass sequences, the chance of generating sequences containing optimization passes that are found in the best performing optimization pass sequences is increased.

### Genetic Algorithms

Genetic algorithms are *heuristic algorithms* that can be used for solving optimization problems [26]. A population of individual solutions is created and a *fitness function* is used to evaluate them. Each iteration, a subset of the population is selected to create new individuals through processes called *selection*, *recombination* and *mutation*. A common selection method is picking the most fit individuals, also known as *elitism*. The selected individuals are then paired up and recombined through a genetic combination operation and possibly mutated.

As previously mentioned, iterative compilation risks tending towards local optima where the fitness of a solution is better than other close solutions in the search space [27]. One way of mitigating this problem is mutation. Mutation increases the diversity of the population, which helps avoid local optima. How mutation works depends on what is being optimized. For a problem where an individual is represented by an array or matrix of genes, where the genes either exist or not, a mutation could be changing a gene from on to off or the other way around. An example of such a problem would be the *knapsack problem* [28].

## 2.4.3 Superoptimization

The goal of a superoptimizer is to find the optimal set of instructions generating the same results for a given set of instruction. It is entirely possible to create a superoptimizer that searches the entire *search space* for the optimal solution, which would be a very time-consuming operation. However, superoptimizers try to limit the search space in different ways. The LENS algorithm [13] made for souper, which will be introduced in the next paragraph, was able to be 11-times faster than other existing *enumerative strategies* due to aggressive *pruning*.

Souper is a superoptimizer which optimizes a purely functional subset of the LLVM IR using integers and scalars [12]. Mali uses LLVM *intermediate representation* (IR) internally and can as such take advantage of souper. Souper has two uses: being used as an optimization pass, but also to help compiler developers find new possible optimizations by working as a separate program. In order to automatically verify an optimization, it tries to find counterexamples that break the equivalence between the optimized code, or right hand side (RHS), and the

original code, or left hand side (LHS). Many of the optimizations generated by souper have also been manually verified for correctness using the LLVM verifier Alive[29][30][12]. As souper exploits undefined behavior, and some external benchmarks rely on undefined behavior, not all external benchmarks are able to verify the optimizations. This means that souper could generate optimizations that are completely correct, but may not mirror the expected behavior in some cases.

Souper uses *synthesis* in order to find optimizations. This means using an *equivalence checker* in order to find an RHS equivalent to a provided LHS. In order to find the smallest equivalent RHS an algorithm has to be deployed. Iterating through every single possible RHS from smallest to largest is not feasible according to the paper [12]. Instead it tries to find equivalences for one single input. If that equivalence then holds for all inputs, a RHS has been synthesized. If it does not, the synthesizer can add conditions which narrow the search space. As a *satisfiability modulo theory* (SMT) solver, the Z3 theorem prover [31] is used. It is used to check if the equivalences are unsatisfiable. Souper will be used in order to try to utilize superoptimization for graphics code for the Arm Mali GPUs.

Graphics code relies on *floating point arithmetic*, and as such the current integer nature of souper is of limited use for graphics content. As per [12], there is also no support for *vector instructions*, another construct heavily utilized by graphics content.

# Chapter 3

# Approach

This chapter describes how we approached the questions we have outlined in chapter 1. The first section deals with peephole optimization. We go over the pattern matching framework used as well as the mechanisms for instruction replacement. The second section deals with time-unconstrained optimization. Both iterative compilation and superoptimization are discussed. The third section deals with benchmarking. Here we explain the common measurement strategy for both peephole optimization and time-unconstrained optimization.

## 3.1 Peephole Optimization

In order to investigate how performance on Mali GPUs are impacted by peephole optimizations we have implemented a compiler independent peephole optimizer based on an existing pattern matching framework. Within this optimizer, peephole optimization is performed by replacing instruction sequences found by a general pattern matcher. An existing proprietary framework for pattern matching was provided by Arm. The framework parses Valhall assembly files and constructs an internal model of the shader program. Compiler developer defined patterns are run on the internal model and subsequent matches are collected in a report. Our work included implementing several patterns as well as extending the frameworks capabilities in order to handle more pattern types. Specifically, we have added what we have dubbed *multiple instruction patterns* and *segmented multiple instruction patterns*, described in further detail in subsection 3.1.4. Furthermore, it involved creating a peephole replacement scheme in order to replace matched instruction sequences with better sequences where possible.

The experimental evaluation of Peephole optimization has some unique qualities that are highlighted at the end of this section.

### 3.1.1 Code Format

Peephole optimization was performed on shader programs compiled into Valhall assembly using the Mali offline compiler [32]. As the Mali ISA is proprietary we will explain the pattern matcher using a generic placeholder language. This is a simple three-address code format. The instructions follow the format seen in listing 3.1.

```
OPCODE destination operand_1 operand_2
```

**Listing 3.1:** An example of the generic placeholder language used in this section.

### 3.1.2 Pattern Matching Overview

Although the pattern matching framework is not the result of our work it will still be given an in-depth description. It is critical that we understand how it operates. Both the patterns implemented in this thesis and the peephole optimizer we have created are built upon the foundation of this framework.

The operation of the pattern matcher starts by parsing an assembly file. Then, for each pattern we are interested in the framework does the following:

- Construct a matching context
- Perform initial match using an initial patterns
- Validate matches
- Produce match result

The match results are then sent to peephole replacement routines corresponding to the respective patterns. A major advantage of the framework over regex is its internal model of the assembly language and semantic knowledge of the program. This is useful when validating matches as this allows patterns to effectively target patterns based on ISA specifics and data dependencies.

### 3.1.3 Op-code Pattern

An Op-code pattern is the basic building block for our initial patterns. It consists of a set of op-codes. An op-code pattern matches a single instruction if that instruction has any of the op-codes in the set. Examples are shown in listings 3.2 and 3.3

```
{MUL}
```

**Listing 3.2:** Example of an op-code pattern. This pattern matches a `MUL` instruction.

```
{MUL, ADD}
```

**Listing 3.3:** Example of an op-code pattern with multiple op-codes. This pattern matches any `MUL` or `ADD` instruction.

### 3.1.4 Initial Pattern

An initial pattern searches through a basic block and finds all instruction sequences that match the given pattern. The initial pattern is some collection of op-code patterns. The exact matching logic depends on the type of initial pattern. The different types are outlined below.

#### Single Instruction Pattern

A single instruction pattern simply consists of a single op-code pattern. The pattern simply matches any instruction that matches the op-code pattern. For example, the single instruction pattern in listing 3.4 matches any `MUL` instruction, as seen in listing 3.5.

```
{MUL}
```

**Listing 3.4:** Example of a single instruction pattern. This pattern matches all `MUL` instructions.

```
MUL r3 r1 r2
```

**Listing 3.5:** Example of an instruction sequence that would match the pattern in listing 3.4.

#### Multiple Instruction Pattern

A multiple instruction pattern consists of a contiguous sequence of op-code patterns. It matches a contiguous sequence of instructions. Specifically, a match occurs if all entries in the op-code pattern list appear in order in the sequence. That is, each entry ordered at position $i$ in the op-code pattern list matches the instruction sequence at position $i$.

An example can be seen in listing 3.6. In listing 3.7 we show an example of a sequence that would match this pattern. Listing 3.8 shows a sequence that would fail a match.

```
{MUL}
{MUL}
{ADD}
```

**Listing 3.6:** Example of a multiple instruction pattern. This pattern matches two `MUL` instructions followed by an `ADD`.

```
MUL r7 r0 r6
MUL r8 r1 r6
ADD r9 r2 r6
```

**Listing 3.7:** Example of an instruction sequence that would match the pattern in listing 3.6.

```
MUL r7 r0 r6
ADD r8 r1 r6
MUL r9 r2 r6
```

**Listing 3.8:** Example of an instruction sequence that would fail to match the pattern in listing 3.6. The op-codes do not appear in the correct order.

## Segmented Multiple Instruction Pattern

A segmented multiple instruction pattern consists of several contiguous sequences of op-code patterns. Each contiguous sequences is called an op-code segment. Similarly, a contiguous sequence of instructions is called an instruction segment. The pattern matches a contiguous sequence of instruction segments. Each segment may be separated by any arbitrary instruction sequence. The instruction segments must be in the same order as the patterns op-code segments. Additionally, all instruction segments must fit within the same basic block.

An example of a segmented multiple instructio pattern can be found in listing 3.9. The sequences in listings 3.10 and 3.11 would match this pattern whereas the sequence in listing 3.12 would fail to match.

```
{ADD}
{MUL}
...
{MUL}
{ADD}
```

**Listing 3.9:** Example of a segmented multiple instruction pattern. This pattern matches the two specified sequences separated by an arbitrary instruction sequence.

```
ADD r9 r2 r6
MUL r7 r0 r6
MOV r11 r10
XOR r11 r6 r7
MUL r8 r0 r6
ADD r12 r2 r6
```

**Listing 3.10:** Example of an instruction sequence that would match the pattern in listing 3.9.

```
ADD r9 r2 r6
MUL r7 r0 r6
MUL r8 r0 r6
ADD r12 r2 r6
```

**Listing 3.11:** Example of an instruction sequence that would match the pattern in listing 3.9. Note that the instruction sequence separating the two segments may be empty.

```
MUL r7 r0 r6
ADD r9 r2 r6
MOV r11 r10
XOR r11 r6 r7
ADD r12 r2 r6
MUL r8 r0 r6
```

**Listing 3.12:** Example of an instruction sequence that would fail to match the pattern in listing 3.9. The segments do not appear in the correct order.

### Chained Instruction Pattern

A chained instruction pattern matches instruction sequences who are linked by use-define chains. That is, each additional op-code pattern needs to have a parameter defined by the previous instruction.

An example is shown in listing 3.13. Listing 3.14 shows a sequence that would match this pattern. Listing 3.15 shows a sequence that would fail to match.

```
{MUL}
→
{MUL}
→
{ADD}
```

**Listing 3.13:** Example of a chained instruction pattern.

```
MUL r7 r0 r6
MUL r8 r1 r7
ADD r9 r2 r8
```

**Listing 3.14:** Example of an instruction sequence that would match the pattern in listing 3.13. The second `MUL` uses the version of `r7` defined in the first `MUL`. The `ADD` uses the version of `r8` defined in the second `MUL`.

## 3.1.5 Parsing

The framework contains a parser which can read Valhall assembly files The parser processes the assembly file line by line. Similar to a compiler, the parser performs lexical, syntax, and

```
MUL r7 r0 r6
MUL r8 r1 r7
ADD r9 r2 r6
```

**Listing 3.15:** Example of an instruction sequence that would fail to match the pattern in listing 3.13. The use-define chain is broken between the `ADD` and the second `MUL`.

semantic analysis. First it splits the program into tokens which are assembled into a node hierarchy of token elements. The tokens are given meaning by identifying functions, basic blocks, instructions, parameters, and so on. While the semantic analysis does construct parameter types and performs basic control flow analysis, it does not verify that the assembly file is a valid program.

### 3.1.6 Matching Context

A matching context is constructed for each basic block. It contains references to the following:

- The originating assembly file
- The label of the basic block
- The instructions of the basic block
- If applicable, the function corresponding to the basic block

Subsequent matching is dependent on the contents of the matching context. Because of this, patterns can be highly context aware. This provides additional flexibility compared to lexical pattern matching. This is both used in the initial matching in the case of chained instruction patterns, in order to find use-define chains, as well as when validating matches, where the validation routine may take decisions based on the context.

### 3.1.7 Initial Matching

An initial matching is performed through a fixed matching routine. The routines are supplied with the matching context, as well as an initial pattern. The initial pattern operates on the basic block of the matching context and returns all matched sequences. The matched sequences are used to create a match results which are then submitted for final validation.

The sequences may overlap which may become an issue when the peephole optimizer performs an instruction replacement. The replacement routine is responsible for handling such issues.

### 3.1.8 Match Result

A match result is created for each matched instruction sequence. It contains references to the following:

- The matching context
- The matched instruction sequence

This information is necessary for the peephole optimizer to perform its work.

### 3.1.9 Validation

The final part of the pattern matching consists of a user programmer defined Boolean validation function. The routine has access to the match result of the initial matching. It may perform arbitrary computation in order to output either true or false. An output of true results in a pattern match and the previous match result is passed along to the peephole replacement routine.

Validation is often required as an initial patterns are often not sophisticated enough to detect a desired pattern. Consider the code in listings 3.16 and 3.17.

```
ADD r1 r2 1
ADD r1 r1 1
```

**Listing 3.16:** A mergeable pair of `ADD` instructions.

```
ADD r1 r2 2
```

**Listing 3.17:** A single `ADD` instruction which performs the same work as listing 3.16.

In both examples we perform the same calculation, namely adding `2` to the register `r2` and stored the result in `r1`. In this case we can safely replace the sequence in listing 3.16 with listing 3.17.

In order to detect such a sequence we could start by using the chained instruction pattern shown in listing 3.18

```
{ADD}
→
{ADD}
```

**Listing 3.18:** Example of an initial pattern that could be used to find potentially mergeable `ADD` pairs.

However, this will match any use-define chain consisting of two adds. In listing 3.19 the two `ADD` instructions write to different destinations and may not be merged.

```
ADD r1 r2 1
ADD r3 r1 1
```

**Listing 3.19:** Example of a false positive of a mergeable `ADD` pair. The results are written to different registers.

```
ADD r1 r2 1
MUL r4 r1 1
ADD r1 r1 1
```

**Listing 3.20:** Example of a false positive of a mergeable `ADD` pair. The result of the first `ADD` is used before it is overwritten.

In listing 3.20 a `MUL` instruction reads the result from the first `ADD`. Merging the two `ADD` instructions will destroy this intermediate result. Therefore we are prevented from performing this optimization.

In this example, we need to validate our initial match to ensure that we only find instruction sequences that may actually be merged. An example of such a validation routine can be seen in algorithm 1

**Algorithm 1:** A validation routine that can ensure that the pair of `ADD` instrucitons is mergeable.

**Input :** Matching Context *context*
function *ValidateMatch*(*context*)
*add*1 ← *retrieve first* **ADD** *instruction from* **context**
*add*2 ← *retrieve first* **ADD** *instruction from* **context**

*dest*1 ← *retrieve destination register of* *add*1
*dest*2 ← *retrieve destination register of* *add*2

`// Ensure that both instructions write to the same register`
**if** *dest*1 ≠ *dest*2 **then**
  **return** False
**end**
*res*1 ← *retrieve variable produced by* *add*1
*uses*1 ← *retrieve set of all instructions that use* *res*1

`// Ensure that the intermediate result is used only by` *add*2
**if** *uses*1 ≠ {*add*2} **then**
  **return** False
**end**
**return** True

### 3.1.10 Replacement

Once we have produced a final match result we may perform peephole optimization. We have devised two strategies for replacement, *direct replacement* and *generalized replacement*. Different strategies are needed for different patterns.

Once peephole optimization has performed we need to write the altered program to a new assembly file. The node hierarchy of token elements needs to be traversed and outputted in

order. This is reverse parsing, or *unparsing*, as it constructs a file from the parse tree.

### Direct Replacement

Direct replacement simply replaces the instruction sequence in the match result with a new sequence. As such, it only involves changes local to the matched pattern.

### Generalized Replacement

Generalized replacement may take any sequence of instructions found in the basic block of the matching context and replace it with a new sequence. It may do this for any number of sequences. This is useful when non-local changes are needed. For example, in order to perform a certain peephole replacement we might need to rename register for instructions outside the matched sequence. In other cases, we are interested in replacing dependencies of the matched pattern, instead of the pattern itself.

### Overlapping Sequences

In theory, matched instruction sequences may overlap. This could cause concerns as an instruction replacement may destroy a sequence found in another match. We have resolved this by verifying that any sequence that is about to be replaced is still present within the program. If the sequence is no longer present the replacement routine is aborted. In practice, however, we have yet to encounter this issue.

### Unparsing

Unparsing, that is outputting a new assembly file after parsing and altering an existing assembly file, is essential if we are to use our peephole optimizations. Assembly files are tokenized when parsed. This tokenization keeps track of the tokens' original position within the file. Unparsing a tokenized file is simply a matter of printing the tokens in their original order.

When performing peephole optimization it is important to update previous tokens as instruction replacement moves all subsequent parts of the file. Peephole optimization does not always simply remove instructions. It may also insert instructions, even if the overall length of a sequence is lowered. It is necessary to provide tokenization for these newly inserted instructions. Once tokens have been created for new instructions, and affected tokens have been updated, unparsing can be performed in the same manner as for an unaltered assembly file.

The peephole replacement routine, along with unparsing, is outlined in algorithm 2.

### Optimization Metric

By replacing instruction sequences we are hoping to improve the program according to some optimization metric. Peephole replacement allows us map a instruction sequence to any arbitrary replacement sequence. As such we may decide our metric on a per-pattern basis, be it reduction in instruction length, using instructions that lower power consumption, avoiding

**Algorithm 2:** Program for performing peephole optimization on assembly file *asm*.

```
   Input : Assembly program asm
 1 function PeepholeOptimize(asm)
 2 foreach Pattern p do
 3     foreach Basic Block b ∈ asm do
 4         context ← construct matching context
           // The below loop may be executed in parallel
 5         foreach Sequence s ∈ context that matches initial pattern for p do
 6             m ← construct match result
 7             seq ← retrieve matched sequence from m
 8             if seq is not still present within the program then
 9                 continue
10             end
11             if Validate(p, m) then
12                 Replace(p, m)
13                 UpdateTokens(asm)
14             end
15         end
16     end
17 end
18 foreach Token t in asm in order do
19     write t to optimized assembly file
20 end
```

memory accesses, etc. In practice, however, we have chosen to replace sequences with shorter ones. We have used the underlying assumption that fewer instructions will reduce execution time. This may not always be true in practice, as differences in instruction length will affect cache behavior. This may cause a given program to end up slower despite using fewer instructions. However, we assume that the likelihood of decreased execution time increases as instruction count goes down.

## NOP padding

By replacing instruction sequences with shorter sequences we are altering the length of the program. This will in turn affect memory layout. Cache boundaries for other shaders loaded into memory will differ. Execution differences due to peephole optimization risk being hidden by noise introduced by changes in caching behavior.

NOP instructions can be added to the end of a program without affecting execution. If reduction in instructions are offset by NOP padding we retain the original program size. Cache alignment between loaded shaders will remain consistent.

### 3.1.11 Measurement Strategy

When evaluating implemented patterns we are interested in pattern prevalence, performance impact, and correctness. Prevalence is useful in order to gauge the significance of the performance impact. Performance impact and correctness are measured according to the common benchmarking method described in section 3.3.

#### Generating Disassemblies and Optimized Programs

Our optimizations have been performed in the shaders found within GPU traces. These are prerecorded snippets of GPU applications that can be replayed in order to perform benchmarking. We describe traces in more detail in section 3.3.

All shaders within a trace are compiled with a non-released development version of the Mali Offline Shader Compiler and subsequently disassembled. This compiler version does perform peephole optimizations. However, these peephole optimization are distinct from the patterns that we have implemented. Disassembled files are fed through our peephole optimizer to produce an optimized collection of disassemblies. These are in turn assembled into machine code once again which form an optimized trace.

#### Pattern Prevalence

We have measured the prevalence statically by observing the number of occurrences of a particular pattern within a trace. Although the number of occurrences should give an indication of how often corresponding instruction sequences are executed, it does not account for program flow. For example, a pattern might only occur once but if this occurrence is within a loop that execute many times the pattern's effect on the performance may be more significant than the lone occurrence suggests.

Pattern prevalence does not tell us enough to know actual performance impact which is why we also perform benchmarking. It may seem unnecessary to measure prevalence if we also have real performance measurements. Still, a relationship between prevalence and performance impact could be examined. The result may tell us if prevalence is a useful proxy for performance gains.

#### Serialized Jobs

The Mali driver enables parallel execution of different workloads[15]. This means that vertex processing of a particular frame may start before the fragment stage of the previous frame has finished. This introduces noise into our performance measurements. We will instead force jobs to execute in a serialized manner in order to better isolate the performance impacts of our optimizations. Serialized execution does not effect the final program output. The program will still behave correctly.

## 3.2 Time-unconstrained Optimization

In our approach, we have investigated the potential gains of time-unconstrained compilation by examining iterative compilation and the possibility of using a superoptimizer. In this section we outline our motivations for the chosen algorithms used for iterative compilation. We have also examine the workflow of souper [4], a superoptimizer.

### 3.2.1 Iterative Compilation

An iterative compiler was created and used in order to generate better optimized optimization pass sequences. The iterative compiler was run for each pipeline in the provided *shader database*. A pipeline contained shader programs of all necessary types required, for example fragment and vertex, in order to generate a shader. If the output from the iterative compiler we got was better than the baseline, we put it in a mirrored structure to that of the shader database, but with only the optimized shaders. We always compiled the entire pipeline for each shader evaluation. First, we compiled and optimized all fragment shaders and then the vertex shaders. This meant that the shaders were optimized separately and independently of each other. No other shader types were considered for this thesis largely due to time constraints as well as perceived possible impact. These two shader types were considered the most likely to impact performance, and due to time limitations, were selected. Optimizing a shader could take the iterative compiler from 1 minute to approximately 15 minutes for the largest shaders. This was a considerable amount of time considering that the shader database contained over 10000 fragment shaders and over 10000 vertex shaders.

The iterative compiler utilized lists of both fixed and optional optimization passes. The generated pass sequence started and ended with fixed optimization passes that were always put at the same places in the optimization pass sequence. The contents of these lists were based on a previous project by an Arm employee.

A number of optimization passes were selected by the iterative compilers algorithm from two optional lists. The selected optional passes were concatenated with the fixed passes and the pipeline was compiled. The optional lists contained optimization passes that were currently used in the compiler, and LLVM optimization passes that were currently not part of the compiler pipeline, but that that was considered possibly useful. The finalized list of optimization passes was ordered like this: *Fixed passes + generated optimization pass sequence + fixed passes* where the generated optimization sequence was made of passes from two lists, as previously mentioned.

The fixed lists included many passes of varying functionality. Some passes were necessary in order to generate correct code. Some of those were specific to the target ISA and some were machine passes and some added or removed transformations done in order to enable other optimization passes. Passes for the target ISA were necessary in order for the code to work on the target architecture.

There were also optimization passes that had to be specifically placed before or after other passes, or other groups of passes, in order to improve performance. Often, passes closer towards the middle of the finished sequence were more likely to be optimization sequences which transformed the IR. Passes towards the end of the finished sequence were more likely to

be related to instruction scheduling or register allocation. Some passes were post-processing which optimized the IR. Not all passes had an impact on the instruction count of the generated code, but were still be necessary for other reasons.

## Comparison Between Markov, Random and Genetic

The iterative compiler used a genetic algorithm implemented by us. The motivation for choosing a genetic algorithm over a Markov algorithm or a purely random algorithm came from some experiments early on in the process.

The three algorithms were implemented and tested by us. They were compared against each other in order to decide which to use in the iterative compiler. The random algorithm was the simplest, being random selection of optimization passes from the optional pass lists. For the experiment, the maximum number of iterations was set to be 3000 iterations, which was an equal or larger amount of iterations than the other algorithms in most expected cases. The choice of first choosing the number of iterations for the random algorithm was made because of the random algorithm being the only algorithm that allowed for setting an exact number of iterations, and as such we could modify parameters for the other algorithms to perform a similar number of iterations. The random algorithm was intended as a baseline, but as can be seen later in this section, it performed better than the Markov chain based algorithm.

The Markov algorithm used was previously created by an Arm employee. It was modified to fit the evaluation metrics for this thesis, but the core design choices were kept the same as it was originally designed for a very similar objective. It utilized Markov chains in order to find optimized sequences. The algorithm first generated random optimization pass sequences until it had 15 that compiled. Then, it used the fitness function evaluation of these sequences in order to modify the likelihood of selecting sequences going forward. The three best sequences were kept for the next group of generated passes. Once it had 15 working sequences again, it repeated the process. The algorithm terminated at a maximum of 3000 compiled sequences (iterations) or if the 7 best sequences in the list of 15 sequences were the same.

The genetic algorithm started with a base population of 720 optimization pass sequences. The best $\frac{1}{6}$ sequences were paired up and recombined in order to replace the worst $\frac{1}{6}$. A big reason for the numbers chosen regarding the genetic algorithm (base population, number of selected sequences,mutation rate, maximum generations, repopulation), was that they resulted in approximately 3000 iterations being performed by the iterative compiler when it was run with a genetic algorithm. A similar amount of iterations between the different algorithms would make comparisons more accurate. The new sequences were created by one-point crossover. For one-point crossover the sequences were split at random points and the leftmost part of the first sequence was combined with the rightmost part of the second sequence and vice versa. These new sequences were added to the population and then every individual sequence had a 20% chance to mutate. If the mutation occurred, the sequence could continue mutating, in other words it did not automatically stop mutating after one mutation. The effect of a mutation was one optimization pass randomly getting replaced by another random optimization pass from the lists of optional optimization passes. For each generation, the best individual sequence was saved in a separate list. This was done to avoid a sequence being mutated away and never being found again while possibly better than any other the algorithm produces. In order to avoid the population size shrinking due to se-

quences that would not compile, the algorithm created new random sequences at the end of a generation that compensated for the difference between the population starting size of 720 and the existing current population.

After a maximum of 18 generations of replacements and mutations, the algorithm was considered finished. The algorithm would also finish if the fittest individual sequence had been the same for 8 consecutive generations. For shaders with a baseline length of less than 100 lines of assembly code, the limit was 4 consecutive generations instead.

If all compilations succeeded, this would lead to the algorithm going through 2880 iterations. However, due to the repopulation in order to compensate for failed compilations, the maximum number of possible iterations would be $720 + 720 \cdot 18 = 13680$. In the case of this test, the population never exceeded 3050 which was still deemed acceptable to compare with the other algorithms by us.

All three types of algorithms were tested against each other which can be seen in figure 3.1 and 3.2. One non-trivial shader was compiled 100 times with each of the algorithms. The choice of 100 was to mitigate variance between runs and allow for patterns in the data to more likely not be caused by chance. Each run also did take around 10 minutes, and as such we did not want to be stuck at algorithm selection for too long early on in the process. The baseline compilation for the selected shader did not have any spills, and as such there was no difference in spills between the algorithms. The baseline compilation had a register usage of 64. It is important to note that the metrics used were prioritized from top to bottom as per the graphs, with no other weights. This means that the iterative compiler would always consider adding any number of instructions to lower the register usage to at most 32 registers as beneficial. The motivation for this was the idea that even if the iterative compiler initially found an expensive way to lower the register usage, it could lower the instruction count of the shader in later iterations. The length metric was the least important, as a simple algorithm utilized at the end of each iterative compilation removed passes which did not positively impact the other static counters. As such, having more passes in the optimization pass sequence only negatively impacted compile time during iterative compilation.

The comparison between the algorithms described in the previous paragraph will be presented here. Its relation to the goal of the thesis only pertains to the selection of a method and not the end results. As such, it will not be presented in the results section. Comparing genetic and random, as per figure 3.1, it could be seen that the genetic algorithm was able to lower the register usage faster. However, the random algorithm caught up with more iterations. That is reasonable, as genetic algorithms have a risk of getting stuck in local optima. The difference between random and genetic algorithms for register usage was it being lowered from 64 to 32 in one more test for the random algorithm. While the number of times the algorithms lowered the register usage was fairly even, the genetic algorithm lowered the number of instructions significantly more than the random algorithm.

As for the Markov algorithm, it could be quickly noted that it got stuck in local optima for the shader tested, as can be seen in figure 3.2. The telling part is that it was unable to lower register usage, unlike the other algorithms. The reason why, is that the optimization pass sequence necessary for this was not all that similar to one required to lower the maximum number of instructions at 64 registers used. It fell behind the random algorithm, which was the same random algorithm data as when compared to the genetic algorithm, very quickly. As

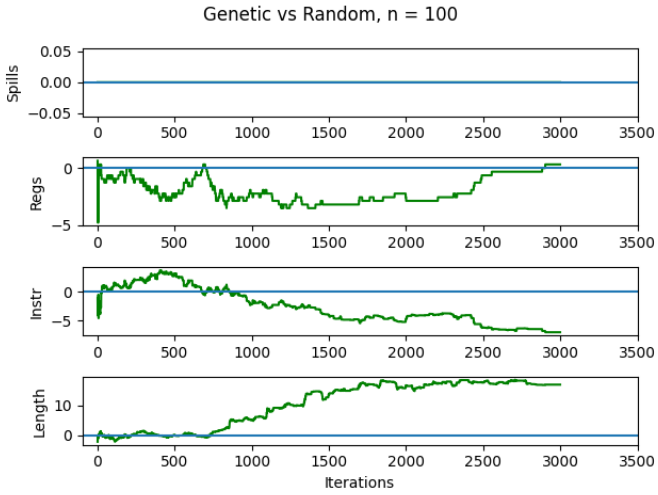


**Figure 3.1:** Genetic algorithm vs random algorithm. Average difference in metrics over iterations from compiling the shader with each algorithm $n$ times. X-axis represents the i-th iteration for the algorithms, which is the same as the i-th compilation. Green line represents $\boldsymbol{random - genetic}$ for the average metric value at that point. Blue line is $y = 0$. As an illustrating example: When it comes to the maximum number of instructions, the green line is above 0 roughly between iterations 0 and 700. This means that the genetic algorithm on average has a higher maximum number of instructions. Past the 700th iteration, we can see that the number steadily goes down, meaning that with future iterations the genetic algorithm lowers the maximum number of instructions more than the random algorithm.

the evaluation metric prioritized register usage over instruction count, the Markov algorithm performed worse than the other two.

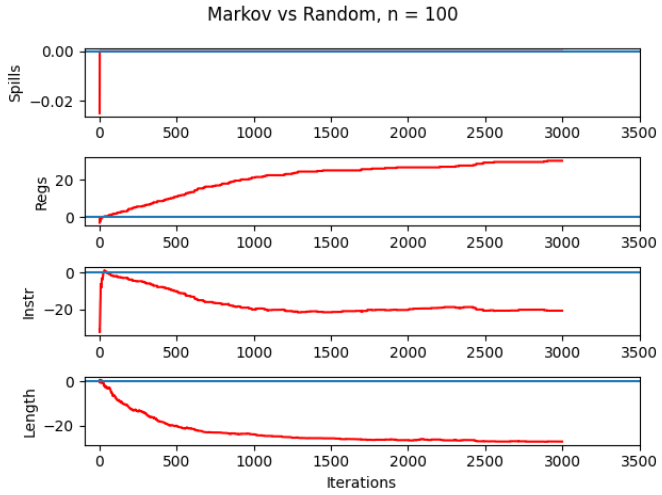


**Figure 3.2:** Markov algorithm vs random algorithm. Average difference in metrics over iterations from compiling the shader with each algorithm $n$ times. X-axis represents the i-th iteration for the algorithms, which is the same as the i-th compilation. Red line represents $random - markov$ for the average metric value at that point. Blue line is $y = 0$.

### Pass Manager Modification

In order to try to find more possible benefits for the Arm Mali GPU compiler from the implementation of the iterative compiler, some more experiments were performed by us. The iterative compiler was used to find hints towards what possible inefficiencies were in the compiler. Passes which the iterative compiler rarely picked were removed from the pass manager for the production compiler one at a time, and the difference in static metrics was generated using an internal tool. We also modified the order of certain passes and tested the results.

## 3.2.2 Superoptimization

In order to evaluate the possibilities of using a superoptimizer to improve code for the Mali series GPUs, souper was chosen by us. One major reason for the choice of souper was that it already worked with LLVM IR, which the Mali series GPUs used. The main goal of this thesis was to look into the possibilities of using souper on graphics content, and what would be required to do so.

We attempted to run souper on multiple shader types, such as fragment, vertex and compute shaders. The main focus was on compute shaders, as they tend to use more integer computations in practice, and as such only have one of the problems concerning vector instructions and floating point instructions.

We selected entire shaders from the shader database as targets for souper to optimize. The compiler was able to produce LLVM IR. The LLVM assembler, *llvm-as* was utilized for generating LLVM bitcode from LLVM IR. The bitcode was then passed to souper along with the path to a local installation of the z3 theorem prover. As souper analyzed the code, it printed out its internal representation and the original code of any possible optimization it found.

Initially, we assumed that the selected shaders had to contain relatively few lines of code for souper to work within reasonable time. However, as most of the graphics content contains a large amount of vector instructions and potentially floating point instructions, souper only performed work on a small part of it. As such, the relationship between shader code size and processing time for souper was not that simple. This meant that it was harder for us to in advance have an idea which shaders would have sequences that souper could optimize.

## 3.3 Benchmarking

In this section we will describe how we measure performance impact. It is important that our measurements correspond well to real world examples. This can be achieved by using GPU traces. We will be looking at hardware counters on both baseline and optimized versions to draw relevant comparisons.

### 3.3.1 GPU Traces

A graphics application schedules work on a GPU by submitting API calls executed by the driver. As such, it is possible to record the behavior of the GPU by capturing driver calls. Such a capture is called a *trace* and can be replayed on compatible devices.

In order to benchmark a particular application, a range of frames were recorded and the resulting trace were replayed on the target device. Because API calls are implementation agnostic, the same trace could be used to evaluate different drivers. Furthermore, it was possible to swap the shader programs in order to evaluate the same application but with different shaders. The shaders within a trace exist within a folder structure. An identical folder structure containing different shaders could be specified to allow the trace to execute using these shader replacements. As such, we could replay a trace with both baseline and optimized versions of the shaders in order to measure performance differences.

A selection of recorded traces was provided by Arm. This included traces of both GLES and Vulkan applications. We used the applications *patrace*[33] and *vktrace*[34] for replaying GLES and Vulkan traces respectively. Furthermore, methods for replacing shaders in the traces as well as collecting various hardware counters were also provied.

### 3.3.2 Measurements

Peephole optimization and iterative compilation were to an extent evaluated using static values such as instruction count. However, static values can not fully capture the actual dynamic

performance. In order to assess performance differences between baseline and optimized traces we relied upon hardware counters available in the Mali architecture.

As Arm does not produce its own hardware implementations of its GPUs, ISAs testing were carried out on a Field-programmable gate array (FPGA). All measurements used an FPGA board simulating a Mali G78 GPU (revision r1p1).

We established an average of each metric on a given trace by running the measurement three times. This average was first determined for a baseline version of the trace. The measurements were then repeated by replacing shaders in the trace with optimized versions, if applicable. For iterative compilation, measurements were made with only the optimized fragment shaders, as well as with both optimized fragment and vertex shaders at the same time. The best version for each trace was selected.

## Hardware Counters

Performance metrics were stored in the GPUs hardware counters [35] by the simulation. These counters measure the number of GPU cycles, as well as the number of instructions executed within the different processing pipelines outlined in section 2.1. The specific counters used are:

- The number of cycles there is work scheduled on the GPU (GPU_ACTIVE)
- The number of executed FMA instructions (EXEC_INSTR_FMA)
- The number of executed CVT instructions (EXEC_INSTR_CVT)
- The number of executed SFU instructions (EXEC_INSTR_SFU)

These metrics were of interest as lower a GPU workload and/or fewer instructions executed both indicate better performance. GPU_ACTIVE being the most important metric as it is the best indicator of overall performance of the GPU.

## Selected Traces

For the static values collected for peephole optimization and iterative compilation we have used a selection of important traces.

We attempted to record hardware counters on all the available traces. However, some traces exhibited issues during replay, such as crashing, forcing us to exclude these from our performance measurements. Additionally, time constraints forced us to prioritize some traces over others. In the case of peephole optimizations we excluded some traces with low prevalence. Traces with low prevalence would likely see little change in performance. Measuring such traces would likely not give additional insight into the relationship between prevalence and performance impact.

In the case of iterative compilation we prioritized traces with good static metrics and quick baseline execution times. We motivated this by noting that in part the aim of this thesis was finding where performance gains can be made.

We do not intend to make claims of universal performance gains or losses. This would be unfeasible as no set of benchmarks could adequately cover all potential applications. Rather we wanted to highlight the extent of the effects that could be seen as a result from our optimizations, and to establish a relationship between static counters and performance.

For iterative compilation, optimized shaders which added too many instructions in order to lower the register usage were excluded. As discussed in the approach, the iterative compiler allowed for increasing maximum instruction count in order to lower register usage with the hope of then lowering the maximum instruction count later. The shaders that did not sufficiently decrease enough of the added maximum instruction count, were not included. The limit was set at a maximum of two extra maximum instructions.

### 3.3.3 Verification

In order to verify the correctness of the optimizations, frames were dumped and compared. There being no visible difference was considered good enough, and bit comparisons were not made.

Neither the OpenGL[36], OpenGL ES 2.0[19], nor Vulkan[23] specification are pixel exact, meaning that an optimization may produce bit-level differences and still be valid. Consider the following equality:

$$a + b - a = b$$

While this holds mathematically, floating point arithmetic may produce different results. For very large $a$ and very small $b$ the equality will not hold. For reasons such as this optimizing arithmetic expressions may change the final bit output. By allowing bit-level differences a compiler may still optimize arithmetic expressions. Since some of our optimizations are in fact arithmetic optimizations we could expect small differences in the output while remaining specification compliant.

# Chapter 4

# Implementation

This chapter describes the implementation of our approach. This includes further implementation details as well as problem that had to be solved in order to proceed. The first section concerns peephole optimization and describes pattern implementation and their corresponding replacements. The second section concerns time-unconstrained optimization and is divided into iterative compilation and superoptimization.

## 4.1 Peephole Optimization

We have implemented 10 pattern matching routines with corresponding replacement routines. The amount was limited to 10 to fit within the time-frame of this thesis. They are based on sub-optimal sequences identified by compiler developers at Arm. The particular patterns chosen are known to occur in important shaders, i.e. shaders that are frequently executed in widely used applications.

### 4.1.1 Patterns

The patterns we have found are sub-optimal mainly due to three reason.

Firstly, some generated sequences that compute a mathematical operation are based on expressions that can be further simplified, or expressed differently, in such a way that they may be implemented with fewer instructions.

Secondly, some sequences do not reason well enough about their input. Because of this they are longer than needed as they try to compute something too general. For example, if an operand of an instruction is the result of a logical operation it is known that its value is

boolean, i.e. either zero or one. Despite this, the code generator may have generated a sequence that behaves correctly over all floating point values. By only trying to behave correctly over the smaller domain of $\{0, 1\}$ the implementation may end up using fewer instructions.

Finally, some sequence simply contain redundant instructions. These patterns find cases were the same result is computed from multiple instructions. Redundant instructions can be omitted, reducing sequence length. This will require register renaming in case the redundant results are not outputted to the same register.

The patterns find opportunities to reduce instructions executed in the FMA, CVT, and SFU pipelines. The patterns find sequences varying between 2 and 6 instructions in length. The sequences are reduced in length by 1 or 2 instructions.

The pattern specifics will not be disclosed in this report as the Valhall ISA is not public. Though table 4.1 summarizes the effect of the peephole optimizations that corresponds to each pattern.

**Table 4.1:** Brief summary of the effects of the peephole optimization for each implemented pattern.

| Pattern | Effect |
|---|---|
| **Pattern 1** | Reduces FMA instruction count by 1 |
| **Pattern 2** | Reduces FMA instruction count by 1 |
| **Pattern 3** | Reduces CVT instruction count by 1 |
| **Pattern 4** | Reduces CVT instruction count by 1 |
| **Pattern 5** | Reduces FMA instruction count by 2, increases CVT instruction count by 1 |
| **Pattern 6** | Reduces CVT instruction count by 1 |
| **Pattern 7** | Reduces CVT instruction count by 1 |
| **Pattern 8** | Reduces FMA instruction count by 1 |
| **Pattern 9** | Reduces FMA instruction count by 2 |
| **Pattern 10** | Reduces SFU instruction count by 1 |

## 4.2 Time-unconstrained Optimization

This sections describes the implementation and implementation details for the iterative compiler and superoptimization.

### 4.2.1 Iterative Compilation

We wrote the iterative compiler in Python. All evaluation was performed for the G78 Mali compiler (revision r1p1) using modified versions of the default compilation files for shaders from a shader database. As previously mentioned, it utilized a genetic algorithm in order to iteratively compile the shaders. Static metrics were generated by an analysis pass that already existed. The static counters from the pass that we used for the static evaluation were the following: The maximum number of instructions executed, the maximum number of

registers used, the maximum number of spill stores and spill loads. Our comparison between two sets of static metrics was performed element-wise. If one had lower maximum spills that the other, it was selected, if not it continued to which had lower registers used and so on. The metrics were chosen with guidance from Arm.

The fitness function could not use hardware counters for the benchmarking as generating hardware counters once could take up to 1 hour for some traces. Up to 3000 iterations per shader at such a speed would not be a sufficiently time-efficient solution. As an example, the longest time the iterative compiler took for a single shader was 15 minutes. A single benchmark run generating hardware counters could take up to one hour. If we were to take the same 3000 iterations of doing this, it would take 3000 hours, or 125 days, to compile one shader.

### 4.2.2 Pre-Register Allocation Scheduler Heuristics

In order to further increase the probability that the iterative compiler was able to optimize shaders, 3 variations of a compiler register heuristic were used. The important thing here is the idea of introducing variation points in the production compiler outside of the iterative compiler. Exactly what the heuristic did, can not be disclosed and is also not of much interest. No register heuristic would be optimal for all possible inputs. As such, any of the three register heuristic variations could yield better optimzied code compared to the other approaches, which would help the iterative compiler produce even better optimized code. This could be expanded upon to other variation points in the compiler, but due to limitations with respect to time, one heuristic was selected.

### 4.2.3 Bugs

#### Static Metrics Bug

It was noticed by another Arm employee during development of the iterative compiler that the static metrics outputted by the utilized compiler analysis pass was not always correct. It was noticed due to large differences between the metrics for the baseline version and the optimized version, but was not guaranteed to be limited to these shaders. To mitigate the impact of this bug, shaders with large observed differences were not be used for measurements. The risk of the bug having impacted shaders with seemingly normal static metric values could not be mitigated in any reasonable way.

#### Empty Shader

For a certain shader, a specific optimization pass followed by another optimization pass generated a corrupted shader. Passes after these two passes further optimized the corrupted shader, leading to a partially empty shader. A part of the code necessary for the second pass was moved somewhere it shouldn't be, and corrupted the results that the second pass relied on. The cause of this issue was that there was an implicit expected ordering of these passes, and other passes expected in-between the passes, which is fulfilled by the production compiler. However, the iterative compiler did not guarantee this ordering, nor did we deem it worth it to do so.

This bug was handled by removing the first pass from the optional optimization pass list, as it was almost never used by the iterative compiler in other situations.

### 4.2.4 Souper

One problem with souper was that it did not support floating point or vector instructions. Both of these were commonly used in graphics content. Compute shaders frequently used integers, while other shader types rarely used integers, which is why computer shaders were the main focus. However, all of them, including compute shaders, still heavily utilized vector instructions. As such, the amount of work that souper could perform on graphics shaders was very limited.

# Chapter 5

# Results

In this chapter we summarize our results. The first section lists interesting findings for peephole optimization. The second section lists interesting findings for time-unconstrained optimization. For a more detailed view of our results we direct the reader to the tables in appendix A. The traces are anonymized as numbers. The numbering is consistent for both peephole optimization and time-unconstrained optimization.

For repeated measurements the differences in hardware counters between each run were always beneath 0.03%. Measured performance gains and regressions larger than this is most likely not just noise.

## 5.1 Peephole Optimization

This section shows the measurement results of our peephole optimizer. This includes pattern prevalence in the form of pattern matches, number of performed optimizations and reduction in instruction count. It also includes performance impact as measured through hardware counters.

### 5.1.1 Pattern Prevalence

Table 5.1 shows percentage of reduction in instruction count for all patterns combined in all traces that were available to us. We did not detect any instances of overlapping pattern occurrences.

Table 5.1 shows prevalence of each pattern in all traces.

**Figure 5.1:** Percentage of reduction in instruction count for each trace sorted in descending order. Each bar represents a single traces.

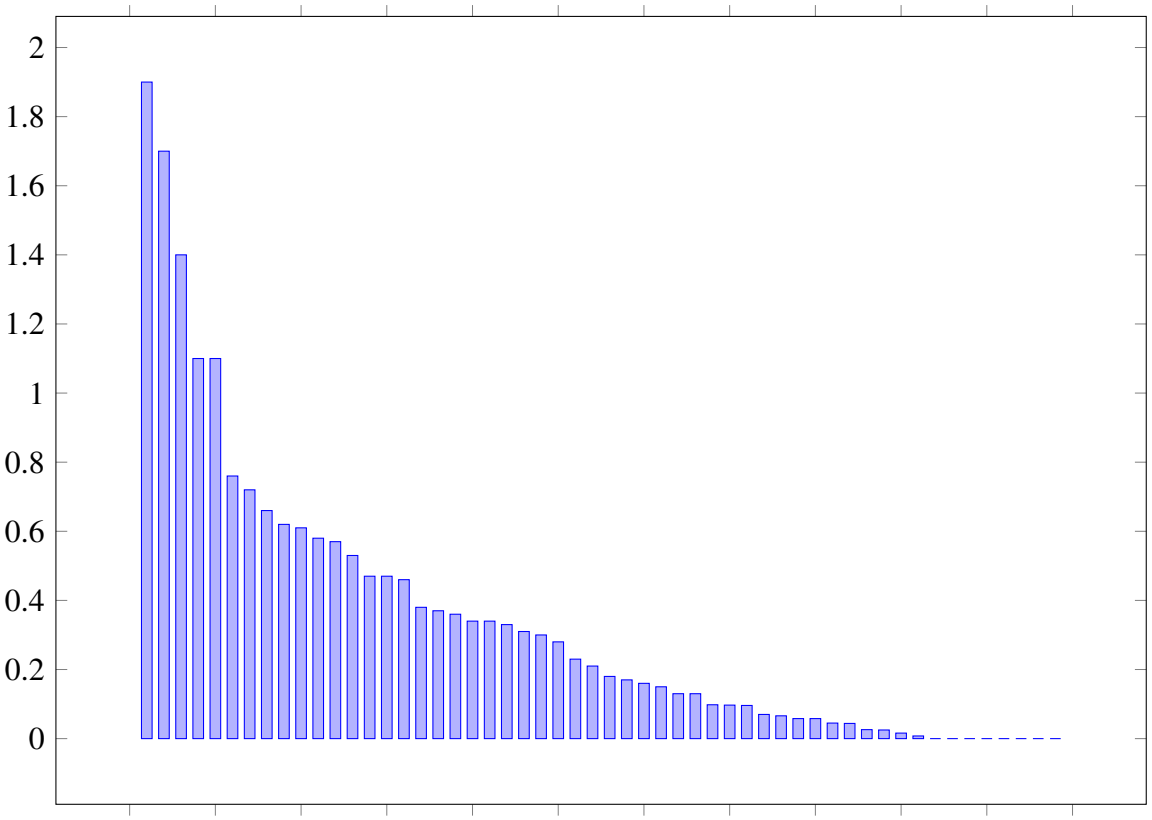


**Table 5.1:** Pattern prevalence for in all traces. *Pattern matches* denotes the number of pattern matches. *Peephole count* denotes the number of successful peephole optimizations. *Reduction count* denotes the reduction in instruction count. Percentages are in relation to the number of instructions in the trace.

| Pattern | Instructions | Pattern matches | Peephole count | Reduction count |
|---|---|---|---|---|
| **Pattern 1** | 4114868 | 133 | 129 | 129 (0.0031%) |
| **Pattern 2** | 4114868 | 3011 | 2691 | 2691 (0.065%) |
| **Pattern 3** | 4114868 | 253 | 248 | 248 (0.006%) |
| **Pattern 4** | 4114868 | 455 | 455 | 455 (0.011%) |
| **Pattern 5** | 4114868 | 41 | 22 | 22 (0.00053%) |
| **Pattern 6** | 4114868 | 26 | 26 | 26 (0.00063%) |
| **Pattern 7** | 4114868 | 0 | 0 | 0 (0%) |
| **Pattern 8** | 4114868 | 3592 | 3582 | 3582 (0.087%) |
| **Pattern 9** | 4114868 | 1685 | 1679 | 3360 (0.082%) |
| **Pattern 10** | 4114868 | 301 | 301 | 301 (0.0073%) |
| **Total** | 4114868 | 9497 | 9133 | 10814 (0.26%) |

### 5.1.2 Performance Impact

Scatter plots showing the relationship between pattern prevalence and hardware counter reduction are shown in figures 5.2, 5.3, 5.4, and 5.5.

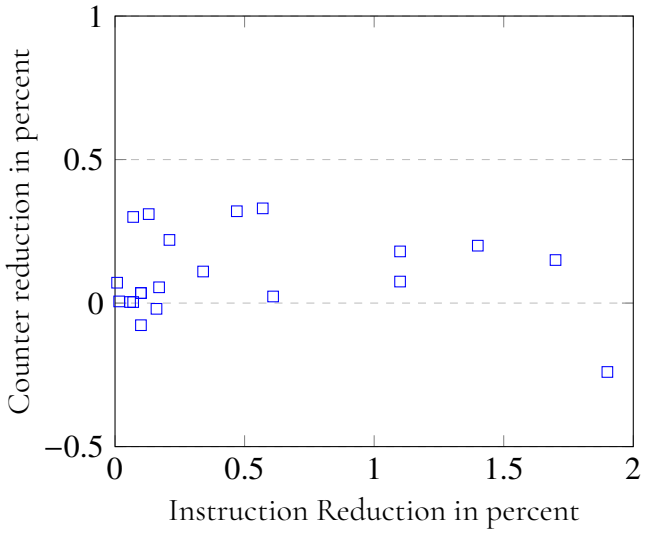


**Figure 5.2:** GPU_ACTIVE reduction percentage plotted against prevalence. (Higher is better)

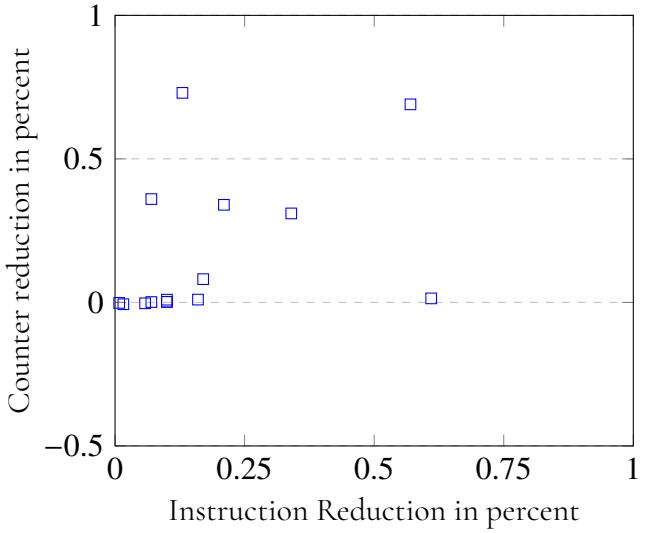


**Figure 5.3:** EXEC_INSTR_FMA reduction percentage plotted against prevalence. (Higher is better)

## 5.2 Time-unconstrained Optimization

This section contains the tables detailing the measured differences for the explored metrics in a selection of traces. For a full reference of static and hardware results for all traces, see appendix A.2.

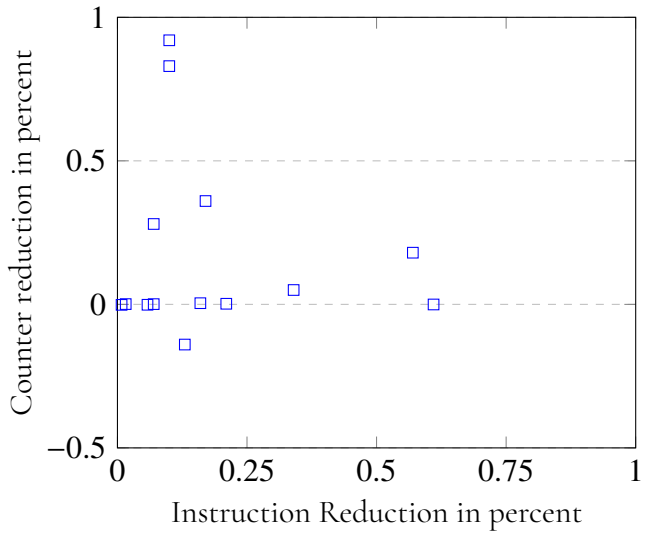


**Figure 5.4:** EXEC_INSTR_CVT reduction percentage plotted against prevalence. (Higher is better)

Hardware counter improvement plotted against prevalence. (Higher is better)

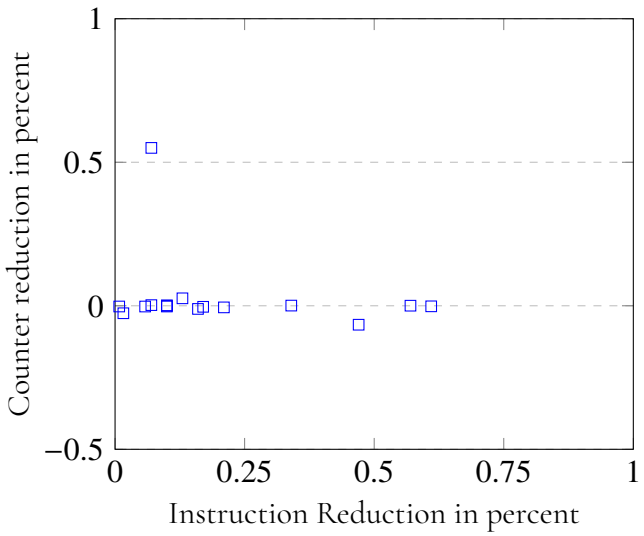


**Figure 5.5:** EXEC_INSTR_SFU reduction percentage plotted against prevalence. (Higher is better)

### 5.2.1 Iterative Compilation

#### Static Results

The static results are the values of the metrics used by the iterative compiler to evaluate different optimization pass sequences. These are the changes of all the individual shaders in a trace summed up as a percentage change from the original value for the trace. Table 5.3 contains the difference in static metrics for vertex shaders from a select number of traces. Table 5.2 contains the same kind of data, but for fragment shaders. As can be seen in table 5.2 and 5.3, all static metrics always improved on average, except for trace 45, where the number of instructions went up slightly. Some traces, such as trace 27, had the number of spills decrease significantly. Trace 8 had almost 10% of its shaders drop in register usage.

**Table 5.2:** Percentage change in static counters for fragment shaders. Spills = **1.5**· spill stores + **1**· spill loads, this was done in order to give slightly higher priority to spill stores.

| | Maximum spilling | Maximum registers used (% of files) | Maximum # of instructions executed |
|---|---|---|---|
| **Average all traces** | -7.61% | -3.83% | -1.29% |
| **trace 4** | 0.00% | 0.00% | -0.46% |
| **trace 8** | 0.00% | -10.83% | -1.42% |
| **trace 12** | 0.00% | -9.91% | -0.43% |
| **trace 27** | -85.95% | -7.09% | -0.67% |
| **trace 37** | 0.00% | 0.00% | -1.73% |
| **trace 45** | 0.00% | -13.85% | +0.15% |

**Table 5.3:** Percentage change in static counters for vertex shaders. Spills = **1.5**· spill stores + **1**· spill loads, this was done in order to give slightly higher priority to spill stores.

| | Maximum spilling | Maximum registers used (% of files) | Maximum # of instructions executed |
|---|---|---|---|
| **Average all traces** | -5.49% | -1.25% | -1.36% |
| **trace 4** | 0.00% | 0.00% | 0.00% |
| **trace 8** | 0.00% | -6.74% | -1.59% |
| **trace 12** | -61.28% | -3.51% | -1.63% |
| **trace 27** | -67.57% | -9.39% | -1.63% |
| **trace 37** | 0.00% | 0.00% | -4.39% |
| **trace 45** | 0.00% | -3.45% | -1.77% |

#### Hardware Results

Table 5.4 contains the difference in hardware counters for a select number of traces with notable results. The values are from the best of either only optimized fragment shaders or both fragment and vertex shaders (always from the same type of test for each trace). The averages for each hardware counter can be seen in table A.5. EXEC_INSTR_CVT decreased more on average, compared to the other metrics. Trace 37 had a large improvement for EXEC_INSTR_SFU. GPU_ACTIVE never had as large changes as the others, but trace 4 had a relatively large improvement, and trace 45 a relatively large regression. Table 5.5 contains results for traces where there were individual compiled shaders which worked, but were not correct. Trace 11, 4 and 47 had incorrect shaders which were caught during validation

which improved the benchmarking results, but had to be removed. Trace 39 had incorrect shaders which worsened benchmarking results.

**Table 5.4:** Percentage change in hardware counters. Lower is better.

| | GPU_ACTIVE | EXEC_INSTR_FMA | EXEC_INSTR_CVT | EXEC_INSTR_SFU |
|---|---|---|---|---|
| **Average all traces** | -0.06% | -0.04% | -3.65% | -0.05% |
| **trace 4** | -2.84% | -0.67% | -5.56% | -0.01% |
| **trace 8** | -0.56% | -0.26% | -11.32% | +0.23% |
| **trace 12** | -0.03% | -0.03% | -0.09% | -0.05% |
| **trace 27** | +0.33% | -0.11% | -1.66% | +0.29% |
| **trace 37** | -0.01% | -2.21% | -4.23% | -10.79% |
| **trace 45** | +2.21% | +3.59% | +3.66% | +4.93% |

**Table 5.5:** Percentage change in hardware counters for traces with incorrectly compiler shaders. Lower is better.

| | GPU_ACTIVE | EXEC_INSTR_FMA | EXEC_INSTR_CVT | EXEC_INSTR_SFU |
|---|---|---|---|---|
| **trace 11 incorrect** | -1.24% | -13.15% | -4.83% | -11.97% |
| **trace 4 incorrect** | -25.27% | -30.40% | -28.22% | -38.22% |
| **trace 47 incorrect** | -30.93% | -36.39% | -32.77% | -36.40% |
| **trace 39 incorrect** | -0.01% | +0.57% | -3.19% | +0.46% |

## Possible improvements

Two sequences were found that showed a possible inefficiency in the current optimization passes for the production compiler. The sequences showed that there were optimizations enabled by passes outside of the production compiler that current compiler passes with similar functionality did not enable.

## Pass Manager Modification

As mentioned in the chapter 3.2.1, production compiler passes that were rarely used by the iterative compiler, and non-production compiler passes that were used frequently by the iterative compiler were tested further outside the iterative compiler setting. Removing production compiler passes (that the iterative compiler rarely used) in the pass sequence for the production compiler yielded varying results. A total of 6 passes were experimented on, with one pass being used twice in the production pass sequence. Below is a list of what removing each individual use of the rarely used passes changed. These are all individual removals of the pass, not one after the other (removing pass 4 means removing only pass 4, not 1, 2, 3 and then 4).

- Pass 1, use 1: Modified 4 shaders in the database, 2 regressions and 2 improvements.
- Pass 1, use 2: Changed nothing
- Pass 2: Large improvements of up to a 99.3% decrease in maximum instructions. Also some regressions. Occurred in a very large amount of shaders.
- Pass 3: Regressions in all affected shaders. Occurred in 32 shaders.

- Pass 4: Both regressions and improvements in affected shaders. Almost all changes affected maximum instructions. Occurred in 90 shaders.
- Pass 5: Both regressions and improvements in affected shaders. Mostly maximum instructions changed, but also some regressions in terms of spill stores. Occurred in 90 shaders.
- Pass 6: Large regressions of many different static metrics in 100 shaders.

Most modification of pass ordering led to crashes. Changing the placement of pass 5 in the pass manager to occur directly after pass 4 instead of a few passes before, also changed behavior. This change led to regressions in almost every shader it changed. However, for shaders in trace 33 the changes were mainly improvements in instruction count instead.

The addition of one pass, to the production compiler pass sequence, that was used frequently in the iterative compiler was also done. It showed changes too large to warrant further investigation, but another pass similar to it was also tested. It showed improvements in a few compute shaders, and no regressions. As for why it was not found by the iterative compiler, it was due to it being an integer only pass, and the iterative compiler only being run on fragment and vertex shaders.

### 5.2.2 Superoptimization

Souper was able to find optimization opportunities in parts of compute shaders. It found a few very similar sequences of three instructions that could be optimized to be one instruction. There were no findings in fragment or vertex shaders. The exact details of the pattern found can not be disclosed.

# Chapter 6

# Discussion

This chapter discusses the results obtained for both peephole optimization and time-unconstrained optimization. Limitations of our approach are also discussed.

## 6.1 Peephole Optimization

Peephole optimization appears to have had a positive impact on performance. Performance gain correlates with high pattern prevalence. Though low prevalence traces can sometimes see improvements in hardware counters.

### 6.1.1 Pattern Prevalence

Through peephole optimization we reduce the instruction count by anywhere between 1.9% and 0%. At no point is instruction count increased. Most matches result in successful peephole optimizations though in traces such as 33 and 21 some matches do not result in optimization. The majority of these cases are due to difficult instances of a pattern were more effort is needed to properly replace the instructions. For example complicated register renaming schemes could be required to perform the replacement. Though given the limited number of instances where we are prevented from performing optimization we decided to focus our efforts elsewhere. The remaining cases are due to global memory constraints, not taken into account by the pattern matcher.

In the five most prevalent traces we see that reductions are entirely due to patterns 1, 2, 4, and 8. Patterns 2 and 4 are expected as together they account for over half of all matches. Patterns 1, 3, 4, 5, 6, 9, and 10 are not as frequent which can explain why only some of them are present in the most significant traces. It is surprising that pattern 9, despite being very prevalent over

all, does not occur in the most significant traces. It appears that pattern occurrence is heavily dependent on what trace is being examined.

Pattern 7 does not occur in any examined trace. The pattern is known to occur in content where its presence is considered important. This content is outside the scope of our experiments and was not included in measurements.

In summary, pattern prevalence vary heavily between traces, where some traces do not have any matches at all. Most matches result in a successful optimization. The prevalence is not evenly distributed over the patterns. Notably patterns 2, 8, and 9 account for over 90% of all matches. However even a common patterns are missing in some traces.

### 6.1.2 Performance Impact

We can see clear impact on hardware counters. Table A.2 in the appendix show that reductions in GPU_ACTIVE up to 0.33%. While this may not appear to be much it is important to note that this is from our limited set of 10 patterns. Within the Mali GPU compiler there are many more patterns that are both already implemented and possible to implement. This should be seen as an indication that peephole optimizations do contribute to noticeable improvements in performance.

While some increases in hardware counters were seen these are generally below the established noise levels of 0.03% and are most likely not actual regressions. Though a few cases of higher increases can be seen. For example, trace 32 saw an increase of 0.24% in GPU_ACTIVE, despite a decrease in all other measured hardware counters. The cause of this is unknown but may be due to changes in dynamic behavior, such as caching.

Trace 37 saw a decrease in EXEC_INSTR_FMA of 5.9% accompanied by an increase in EXEC_INSTR_CVT of 3.5%. This is most likely due to pattern 5, as it reduces FMA instructions by two but increases CVT instructions by one for each replacement.

The scatter plot in figure 5.2 seem to show that a reduction in instruction count leads to a decrease in GPU_ACTIVE, though this trend is quite noisy. Generally, it seems that after a certain instruction reduction threshold we find more consistent improvements. Fewer number of instruction reductions do not always lead to significant reduction in GPU_ACTIVE but some outliers do see significant performance gains.

Similar positive trends seem to hold for EXEC_INSTR_FMA and EXEC_INSTR_CVT. This is expected as the performance gain from peephole optimization should be coming from a decrease in the number of executed instructions. Furthermore, the implemented patterns focus mainly on FMA and SFU instructions. Patterns 1, 2, 5, 8, and 9 all focus on reducing FMA instructions and constitute a majority of matches. Patterns 3, 4, 6, and 7 focus on CVT instructions though these are not as prevalent. In fact, pattern 7 does not occur at all.

EXEC_INSTR_SFU remains largely unaffected, except for a couple of outliers. Pattern 10 is the only peephole optimization targeting SFU instructions and it is absent from most traces. It is therefore natural that most traces show no difference for SFU.

It is worth to note that despite low prevalence of patterns within a trace the performance gains can still be unexpectedly high. In fact, for all hardware counters the highest reduction

were in traces that had less then half of the reduction in instructions compared to the trace with the largest instruction reduction.

### 6.1.3 Limitations

While the optimizations were shown to produce the same result visually as compared to baseline results incorrectness is still a possibility. We did not verify bit correctness with the justification that GLES and vulkan do not require bit exact results. Also, we do not know if potential bit inexactness is within acceptable bounds. Furthermore, we did not employ any formal verification tools to guarantee optimization correctness. While we believe our implementations to be correct it is difficult to eliminate all uncertainties.

Naturally, we found a larger number of traces with a low prevalence of patterns compared to traces with high prevalence. Our assertions about the positive impact of high prevalence on performance draw on relatively few data points. In other words, our knowledge about low prevalence traces is more certain than high prevalence traces. This could be mitigated by recording more traces or implementing more peephole patterns.

It would have been useful to implement optimizations both independently and within the compiler and compare the results. This could have shown to what extent the performance impact of compiler independent peephole optimization translate into performance impact in the production compiler. This was discussed during our work but was unfortunately not possible due to time constraints.

## 6.2 Time-unconstrained Optimization

### 6.2.1 Shader Improvement

There were a few traces that showed large improvements such as trace 4 and trace 47. However, upon closer inspections, this was due to corrupted shaders. Contrary to those traces, trace 39 got better results when its corrupted shaders were removed. Overall, most traces showed very small improvements or regressions when it came to GPU_ACTIVE, which is considered the most important metric. EXEC_INSTR_CVT did decrease on average for the optimized shaders, but there is no clear correlation for individual shaders.

For example, trace 26 had the number of instructions in its vertex shaders reduced by 9.38%, yet had almost no changes in any of its hardware counters. Another example of discrepancy is trace 4 where the only static improvement was a reduction in maximum number of instructions in fragment shaders by 0.46%. This trace had a 2.84% reduction in GPU_ACTIVE, which was the second best out of all traces. Part of the reason for this discrepancy is that not every shader was run the same amount, and neither was each path in the shaders. Some optimizations might have modified less frequently executed code paths in the shaders, or rarely used shaders overall.

There was also no clear correlation between traces where maximum number of spills were lowered and improved hardware counters. This is surprising as spills are often attributed as very expensive in terms of execution time.

One conclusion that can be drawn from the data appears to be that the static metrics favored lowering EXEC_INSTR_CVT more than the other metrics. This could be either due to the selection of shaders that it was tested on, or due to the static metrics used for static evaluation.

The fact that there was such clear regression for trace 45 is also of interest. There were most likely one or two shaders which had a large negative impact on the overall performance. It was also the only trace where the iterative compiler increased the maximum number of instructions on average. This was due to the iterative compiler preferring to reduce the maximum number of instructions over reducing the number of instructions if given the choice. The regression in maximum number of instructions was not very significant compared to the improvements for many other traces, though. The total number of instructions in its fragment shaders went up by 0.15%. Not all of these shaders ended up being used, as shaders which added more than two instructions were not added to the testing. As such, the two shaders where the iterative compiler added 31 instructions to reduce the register usage from 64 to 32 were not included. As previously mentioned, the idea was that adding any amount of instructions to lower register usage might allow the iterative compiler to later lower the number of instructions again. If it did not, the resulting shader was not used. However, it is not always a good trade-off to add instructions to lower the number of registers used.

### 6.2.2 Limitations

Even though each individual compilation was not constrained by time, there was still an implicit time constraint. Due to the large number of shaders in the shader database and the time it took to compile shaders, time was limited. There were also many possible modifications that could be made to the iterative compiler, that could not be explored. Parameters for the genetic algorithm could be tweaked, but other modifications such as pass list changes or fitness function changes could also be made. Making such changes and comparing them over a large set of traces in order to see the impact would be unfeasible with regards to time.

There was also only a weak connection between the goal and what could easily be measured. The evaluation had to be done using static counters. Such metrics include register spilling, numbers of registers used and code size. As could be seen from the results, the metrics did lack a direct connection to performance on hardware. There was, as previously mentioned, no clear connection between static metric improvement and hardware results. Some traces had large hardware improvements, while the improvements in static counters were smaller than for other traces with worse hardware improvements. This limitation did make it hard to predict which traces would perform well, but also to understand what the static metrics should look for. Lowering the register usage might be very impactful for one shader, whereas lowering the maximum instructions might be more impactful for another. This created a situation where it was close to impossible to decide if a particular shader had a positive impact on compilation without extensive manual testing.

This flaw came with the method of using a compilation strategy that relied on instant feedback and multiple iterations. Each shader pipeline could not be tested on its own within a reasonable amount of time. We had access to an internal Arm tool whose aim was to find the shaders which regressed on hardware. However, due to the fact that it had to test the en-

tire trace on hardware for each iteration of its process, it took the tool a significant amount of time for a single trace. This made it an unfeasible solution due to the number of traces. As trace 45 suggests that certain shaders might face regressions even without regressions in static values, it is very likely that a subset of improved could generate better results than all of them together. One possible explanation for regressions could be that a certain optimization would include reducing the number of maximum instructions by increasing usage of a certain pipeline (e.g reducing CVT instructions, but adding FMA instructions). However, that might create a regression on hardware if the CVT pipeline is already heavily used at that point of time in execution.

Another limitation is that of what the compiler was allowed to do. Since the compilers main job was to modify the optimization pass list, the observed potential gains were tied to that. Iterative compilation might be able to visualize more significant potential gains when used in other areas of the Mali compiler environment.

As mentioned multiple times, the lack of full support from souper makes the results very limited. As part of the For other content, it might be able to find more missed optimizations.

### 6.2.3 Possible Compiler Improvements

As mentioned in the results and implementation sections, a few discoveries regarding possible improvements, bugs or otherwise were made. At least one appears to show that there are cases which the current optimization passes do not cover.

Discoveries regarding the current optimization passes used in the production compiler were also found. As mentioned, some passes appear to impact static metrics very little. Due to time constraints, the modifications had to be tested with static counters. The few passes that appeared to not have a significant impact were all related. As such, further investigation is necessary.

### 6.2.4 Pass Manager Modification

The pass manager modification did yield some interesting results, but not all of them may be usable. Pass 2, which did not do anything, is a very inexpensive pass when it does not do anything, and as such does not matter much. Due to the large number of instructions that pass 2 appears to create, it is assumed to be intentional and necessary. Pass 3, which is paired with pass 2 only improved shaders, and as such did not need to be looked into further.

The fact that both pass 4 and 5 caused regressions and improvements was of great interest. Removing either of them caused a few improvements, when the pass was supposed to not supposed to be able to negatively impact . Pass 6 was the selected pass that was used the least by the iterative compiler, but it clearly impacted quite a few shaders. This could mean that while it never was part of optimized optimization pass sequences, it did on average positively impact less optimized optimization pass sequences.

For the ordering, the fact that trace 33 was very positively affected by moving pass 4 should also be investigated. It could be the sign of other missed optimization opportunities within

other earlier passes for the shader, or that pass 5 did not perform certain beneficial optimizations that pass 4 did.

### 6.2.5 Superoptimizaton

Superoptimization did expose some optimization opportunities, and may be promising for future use. However, its reach does need to be expanded before that is possible.

## 6.3 Impact on Society

We do not anticipate any major societal impact or ethical issues to arise from our work. Our aim has been to aid the incremental improvement of GPU compilers. While this could lead to improved user experiences this likely not something that will impact society at large.

# Chapter 7

# Conclusions

Peephole optimization seems to provide performance gains and is a viable strategy to keep improving the Mali GPU compiler. Further more, the performance impact of low prevalence peephole patterns is highly variable. If a particular peephole optimization is easy to implement, it could be advisable to do so regardless of prevalence. However, if implementation is more demanding prevalence should be examined first before spending development resources.

With this said, some low prevalence peephole patterns are exceptions to this rule. We saw some cases of significant performance increase despite lower prevalence. For this reason it might be worth to implement a low prevalence peephole optimization if it is estimated to be low effort.

Time-unconstrained compilation suffers from a large distance between the target and the approach. However, it does provide results in terms of lower EXEC_CVT_INSTR mainly. On an individual trace level, it does not show any direct correlation between static results and hardware results. As mentioned in the discussion, this may be due to each shader not being executed an equal amount of times, or each code path not being taken an equal amount of times.

## 7.1 Future Work

While we seem to find a positive correlation between peephole pattern prevalence and performance a limited amount of data points limits our certainty. In particular, we would like to measure more high prevalence traces in order to verify that this trend holds. Further experimentation with a larger number of traces could help in this area. Additionally, implementing more peephole optimizations could boost prevalence in existing traces. This could both in-

crease our samples of high prevalence traces as well as seeing how the trend develops as we increase prevalence even further.

As previously mentioned we have not explored differences between compiler independent and compiler implemented peephole optimization. A natural next step would be to implement our patterns in the compiler and compare with our results.

The iterative compiler does have its uses. However, as discussed, the limitations brought by the static metrics limits its use. An improved fitness function that could be more closely related to the observed hardware performance would improve the usefulness of the tool. It was able to reduce EXEC_CVT_INSTR on average, which is a good start. It would most likely be possible to develop a metric which more closely ties to GPU_ACTIVE, but that is most likely not trivial. Further experimenting with how many instructions is worth adding to lower the amount of registers used is also of interest. Other future work would include looking into the different optimization opportunities that the iterative compiler did expose. Fixing the few errors that were exposed and relevant would also be a good thing to do.

In terms of what would be necessary in order to get more value out of souper, implementing vector instructions first would be more efficient time wise. Support for vector instruction would allow for souper to fully analyze compute shaders. If more missed optimizations were to be found, then adding floating point support might be of interest. Since vector instructions support is necessary for the other shader types, this order of implementation seems the most beneficial.

# References


[1] C. McClanahan, "History and evolution of gpu architecture a paper survey," Georgia Tech, 2011.

[2] R. Smith, "ARM's Mali Midgard Architecture Explored." `https://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored/2`. Last Accessed: June 30, 2021.

[3] Arm, "Mali GPU architectures." `https://developer.arm.com/architectures/media-architectures/gpu-architecture`. Last Accessed: June 30, 2021.

[4] Google, "souper." `https://github.com/google/souper`. Last Accessed: June 20, 2021.

[5] W. M. McKeeman, "Peephole optimization," *Commun. ACM*, vol. 8, pp. 443–444, 1965.

[6] P. Chakraborty, "Fifty years of peephole optimization," 2015.

[7] E. Aktolga, "Pattern matching strategies for peephole optimisation," 2005.

[8] H. Massalin, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, no. 5, p. 122–126, 1987.

[9] J. Jain, J. A. Abraham, J. Bitner, and D. S. Fussell, "Probabilistic verification of boolean functions," *Formal Methods in System Design*, vol. 1, no. 1, p. 61–115, 1992.

[10] T. Granlund and R. Kenner, "Eliminating branches using a superoptimizer and the gnu c compiler," *ACM SIGPLAN Notices*, vol. 27, no. 7, p. 341–352, 1992.

[11] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle, "Iterative compilation," *Embedded Processor Design Challenges Lecture Notes in Computer Science*, p. 171–187, 2002.

[12] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," November 2017.

[13] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," *ACM SIGPLAN Notices*, vol. 51, pp. 297–310, 03 2016.

[14] Arm, "Mali-G78." `https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-g78-gpu`. Last Accessed: August 27, 2021.

[15] Arm, "Graphics and gaming development: The valhall shader core." `https://developer.arm.com/solutions/graphics-and-gaming/developer-guides/learn-the-basics/the-valhall-shader-core/single-page`. Last Accessed: May 28, 2021.

[16] Arm, "Mali valhall architecture." `https://developer.arm.com/documentation/101863/0703/Mali-GPU-pipelines/Mali-Valhall-architecture`. Last Accessed: July 1, 2021.

[17] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*. USA: A. K. Peters, Ltd., 4th ed., 2018.

[18] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[19] The Khronos Group, Inc, "OpenGL ES Common Profile Specification Version 2.0.25." `https://www.khronos.org/registry/OpenGL/specs/es/2.0/es_full_spec_2.0.pdf`. Last Accessed: July 1, 2021.

[20] D. Spinellis, "Declarative peephole optimization using string pattern matching," *SIGPLAN Not.*, vol. 34, p. 47–50, Feb. 1999.

[21] LLVM Developer Group, "The LLVM Target-Independent Code Generator." `https://llvm.org/docs/CodeGenerator.html`. Last Accessed: July 2, 2021.

[22] Arm, "Arm mali gpu best practices developer guide, texture sampling performance." `https://developer.arm.com/documentation/101897/0200/buffers-and-textures/texture-sampling-performance`. Last Accessed: Oct 9, 2021.

[23] The Khronos Group, Inc, "Vulkan 1.1.183 - A Specification." `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html`. Last Accessed: July 1, 2021.

[24] V. Volkov, *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016. Last Accessed: Oct 9, 2021.

[25] A. Tolver, *An introduction to Markov Chains*. Department of Mathematical Sciences, University of Copenhagen, November 2016.

[26] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998.

[27] M. Rocha and J. Neves, "Preventing premature convergence to local optima in genetic algorithms via random offspring generation," in *Multiple Approaches to Intelligent Systems*, (Berlin, Heidelberg), pp. 127–136, Springer Berlin Heidelberg, 1999.

[28] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc., 1990.

[29] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with alive," *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 22–32, 2015.

[30] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr, "Alive2: bounded translation validation for llvm," *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, p. 65–79, 2021.

[31] M. Research, "Z3." `https://github.com/Z3Prover/z3`. Last Accessed: August 9, 2021.

[32] Arm, "Mali Offline Compiler." `https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/mali-offline-compiler`. Last Accessed: July 5, 2021.

[33] Arm, "patrace." `https://github.com/ARM-software/patrace/`. Last Accessed: July 1, 2021.

[34] LunarG, Inc., "vktrace." `https://github.com/LunarG/vktrace`. Last Accessed: Aug 7, 2021.

[35] Arm, "Mali-G78 Performance Counters." `https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-performance-counters/mali-g78-counters`. Last Accessed: July 5, 2021.

[36] The Khronos Group, Inc, "The OpenGL Graphics System: A Specification Version 4.6." `https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf`. Last Accessed: July 1, 2021.

# Appendices

# Appendix A

# Complete Results

This appendix contains tables with complete results for both peephole optimization and time-unconstrained optimization.

## A.1 Peephole Optimization

This section contains tables detailing the result of the performed peephole optimizations. Table A.1 shows prevalence, whereas A.2 shows performance impact on hardware counters.

### A.1.1 Prevalence

**Table A.1:** Prevalence of all patterns in each trace. *Instructions* denote the number of instructions. *Pattern matches* denotes the number of pattern matches. *Peephole count* denotes the number of successful peephole optimization. *Reduction count* denotes the reduction in instruction count.

| Traces | Instructions | Pattern matches | Peephole count | Reduction count |
|---|---|---|---|---|
| **trace 1** | 517 | 0 | 0 | 0 (0%) |
| **trace 2** | 40357 | 306 | 306 | 306 (0.76%) |
| **trace 3** | 48558 | 294 | 294 | 294 (0.61%) |
| **trace 4** | 11190 | 38 | 38 | 38 (0.34%) |
| **trace 5** | 12096 | 24 | 22 | 22 (0.18%) |
| **trace 6** | 577 | 10 | 10 | 10 (1.7%) |

| | | | | |
|---|---|---|---|---|
| **trace 7** | 17579 | 128 | 116 | 116 (0.66%) |
| **trace 8** | 55736 | 64 | 54 | 54 (0.097%) |
| **trace 9** | 56028 | 13 | 9 | 9 (0.016%) |
| **trace 10** | 18899 | 63 | 56 | 57 (0.3%) |
| **trace 11** | 3907 | 1 | 1 | 1 (0.026%) |
| **trace 12** | 78184 | 54 | 53 | 55 (0.07%) |
| **trace 13** | 13597 | 45 | 45 | 45 (0.33%) |
| **trace 14** | 447 | 0 | 0 | 0 (0%) |
| **trace 15** | 919 | 0 | 0 | 0 (0%) |
| **trace 16** | 10471 | 53 | 48 | 48 (0.46%) |
| **trace 17** | 295661 | 1077 | 1073 | 1075 (0.36%) |
| **trace 18** | 182961 | 435 | 418 | 420 (0.23%) |
| **trace 19** | 11950 | 69 | 69 | 69 (0.58%) |
| **trace 20** | 44334 | 77 | 66 | 66 (0.15%) |
| **trace 21** | 81400 | 264 | 252 | 253 (0.31%) |
| **trace 22** | 27692 | 7 | 7 | 7 (0.025%) |
| **trace 23** | 456 | 0 | 0 | 0 (0%) |
| **trace 24** | 7397 | 53 | 53 | 53 (0.72%) |
| **trace 25** | 497 | 0 | 0 | 0 (0%) |
| **trace 26** | 2213 | 1 | 1 | 1 (0.045%) |
| **trace 27** | 2289517 | 3749 | 3545 | 4778 (0.21%) |
| **trace 28** | 26125 | 2 | 2 | 2 (0.0077%) |
| **trace 29** | 24665 | 94 | 94 | 94 (0.38%) |
| **trace 30** | 2419746 | 5838 | 5701 | 6665 (0.28%) |
| **trace 31** | 10083 | 53 | 53 | 53 (0.53%) |
| **trace 32** | 216 | 4 | 4 | 4 (1.9%) |
| **trace 33** | 42755 | 165 | 145 | 145 (0.34%) |
| **trace 34** | 56487 | 107 | 96 | 96 (0.17%) |
| **trace 35** | 16553 | 23 | 11 | 11 (0.066%) |
| **trace 36** | 8091 | 39 | 38 | 38 (0.47%) |
| **trace 37** | 3209 | 15 | 15 | 15 (0.47%) |
| **trace 38** | 14642 | 54 | 54 | 54 (0.37%) |
| **trace 39** | 25696 | 15 | 15 | 15 (0.058%) |
| **trace 40** | 9091 | 12 | 12 | 12 (0.13%) |
| **trace 41** | 50207 | 62 | 48 | 48 (0.096%) |
| **trace 42** | 6769 | 3 | 3 | 3 (0.044%) |
| **trace 43** | 48262 | 373 | 293 | 299 (0.62%) |
| **trace 44** | 633 | 1 | 1 | 1 (0.16%) |
| **trace 45** | 40487 | 63 | 52 | 52 (0.13%) |
| **trace 46** | 6805 | 0 | 0 | 0 (0%) |
| **trace 47** | 8577 | 49 | 49 | 49 (0.57%) |
| **trace 48** | 1639 | 18 | 18 | 18 (1.1%) |
| **trace 49** | 11484 | 127 | 127 | 127 (1.1%) |
| **trace 50** | 48176 | 60 | 47 | 47 (0.098%) |

| | | | | |
|---|---|---|---|---|
| **trace 51** | 3812 | 0 | 0 | 0 (0%) |
| **trace 52** | 8746 | 126 | 126 | 126 (1.4%) |
| **trace 53** | 30989 | 18 | 18 | 18 (0.058%) |
| **trace 54** | 516 | 0 | 0 | 0 (0%) |
| **Total** | 4114868 | 9497 | 9133 | 10814 (0.26%) |

### A.1.2 Hardware Results

**Table A.2:** Percentage change in hardware counters. Lower is better.

| | GPU_ACTIVE | EXEC_INSTR_FMA | EXEC_INSTR_CVT | EXEC_INSTR_SFU |
|---|---|---|---|---|
| **trace 1** | - | - | - | - |
| **trace 2** | - | - | - | - |
| **trace 3** | - | - | - | - |
| **trace 4** | -0.11% | -0.31% | -0.05% | -0.00067% |
| **trace 5** | - | - | - | - |
| **trace 6** | -0.15% | -0.43% | +0.0088% | +0.0034% |
| **trace 7** | - | - | - | - |
| **trace 8** | -0.035% | -0.002% | -0.92% | +0.0025% |
| **trace 9** | -0.0058% | +0.0063% | -0.00098% | +0.026% |
| **trace 10** | - | - | - | - |
| **trace 11** | - | - | - | - |
| **trace 12** | -0.0036% | -0.0013% | -0.00087% | -0.0023% |
| **trace 13** | - | - | - | - |
| **trace 14** | - | - | - | - |
| **trace 15** | - | - | - | - |
| **trace 16** | - | - | - | - |
| **trace 17** | - | - | - | - |
| **trace 18** | - | - | - | - |
| **trace 19** | - | - | - | - |
| **trace 20** | - | - | - | - |
| **trace 21** | - | - | - | - |
| **trace 22** | -0.023% | -0.014% | +0.00038% | +0.0019% |
| **trace 23** | +0.0063% | -0.0011% | -0.00025% | -0.0036% |
| **trace 24** | - | - | - | - |
| **trace 25** | -0.0012% | +0.00047% | +0.00046% | -0.0015% |
| **trace 26** | - | - | - | - |
| **trace 27** | -0.22% | -0.34% | -0.0021% | +0.0056% |
| **trace 28** | -0.071% | +0.0017% | +0.0017% | +0.0025% |
| **trace 29** | - | - | - | - |
| **trace 30** | - | - | - | - |
| **trace 31** | - | - | - | - |
| **trace 32** | +0.24% | -4% | -0.015% | -0.11% |
| **trace 33** | - | - | - | - |

| | | | | |
|---|---|---|---|---|
| **trace 34** | -0.055% | -0.081% | -0.36% | +0.0036% |
| **trace 35** | -0.3% | -0.36% | -0.28% | -0.55% |
| **trace 36** | - | - | - | - |
| **trace 37** | -0.32% | -5.9% | +3.5% | +0.066% |
| **trace 38** | - | - | - | - |
| **trace 39** | - | - | - | - |
| **trace 40** | - | - | - | - |
| **trace 41** | +0.077% | -0.0095% | -1.2% | -4.3e-05% |
| **trace 42** | - | - | - | - |
| **trace 43** | - | - | - | - |
| **trace 44** | +0.02% | -0.0098% | -0.0041% | +0.011% |
| **trace 45** | -0.31% | -0.73% | +0.14% | -0.026% |
| **trace 46** | - | - | - | - |
| **trace 47** | -0.33% | -0.69% | -0.18% | -0.00029% |
| **trace 48** | -0.18% | -0.19% | +0.00042% | -0.00024% |
| **trace 49** | -0.075% | -0.32% | -0.065% | -0.00019% |
| **trace 50** | -0.035% | -0.002% | -0.83% | -0.0015% |
| **trace 51** | - | - | - | - |
| **trace 52** | -0.2% | -0.38% | -0.7% | -0.0016% |
| **trace 53** | -0.0038% | +0.0031% | +0.0014% | +0.0026% |
| **trace 54** | - | - | - | - |
| | | | | |

# A.2 Time-unconstrained Compilation

This section contains the tables detailing the measured differences for the explored metrics. Table A.3 and A.4 show static results, whereas A.5 shows results from benchmarking.

## A.2.1 Static Results

Table A.4 contains the difference in static metrics for a select number of traces for vertex shaders. Table A.3 contains the same, but for fragment shaders.

**Table A.3:** Percentage change in static counters for fragment shaders. Spills = **1.5**· spill stores + **1**· spill loads. Shaders with very large changes are excluded from the data.

| | **Maximum spilling** | **Maximum registers used** | **Maximum # of instructions executed** |
|---|---|---|---|
| **trace 1** | 0.00% | 0.00% | 0.00% |
| **trace 2** | -51.61% | 0.00% | -2.30% |
| **trace 3** | 0.00% | 0.00% | -1.01% |
| **trace 4** | 0.00% | 0.00% | -0.46% |
| **trace 5** | 0.00% | 0.00% | -1.65% |
| **trace 6** | 0.00% | 0.00% | 0.00% |

| | | | |
|---|---|---|---|
| **trace 7** | 0.00% | -4.29% | -1.76% |
| **trace 8** | 0.00% | -10.83% | -1.42% |
| **trace 9** | 0.00% | -0.00% | -1.77% |
| **trace 10** | 0.00% | 0.00% | -0.22% |
| **trace 11** | 0.00% | 0.00% | -3.11% |
| **trace 12** | 0.00% | -9.91% | -0.43% |
| **trace 13** | 0.00% | -3.70% | -1.46% |
| **trace 14** | 0.00% | 0.00% | 0.00% |
| **trace 15** | 0.00% | 0.00% | 0.00% |
| **trace 16** | 0.00% | -3.70% | -0.63% |
| **trace 17** | -10.96% | 0.00% | +0.76% |
| **trace 18** | 0.00% | -6.99% | -0.74% |
| **trace 19** | 0.00% | 0.00% | -0.97% |
| **trace 20** | 0.00% | -8.40% | -1.27% |
| **trace 21** | -50.00% | -17.61% | -2.33% |
| **trace 22** | 0.00% | -0.61% | -1.06% |
| **trace 23** | 0.00% | 0.00% | -1.55% |
| **trace 24** | 0.00% | 0.00% | -0.54% |
| **trace 25** | 0.00% | 0.00% | -2.30% |
| **trace 26** | 0.00% | 0.00% | 0.00% |
| **trace 27** | -85.95% | -7.09% | -0.67% |
| **trace 28** | 0.00% | -2.77% | -5.01% |
| **trace 29** | 0.00% | 0.00% | 0.00% |
| **trace 30** | -83.64% | -7.06% | -0.85% |
| **trace 31** | 0.00% | -2.33% | -3.31% |
| **trace 32** | 0.00% | 0.00% | 0.00% |
| **trace 33** | 0.00% | -0.79% | -0.31% |
| **trace 34** | -93.51% | -17.42% | -0.35% |
| **trace 35** | 0.00% | -2.47% | -0.97% |
| **trace 36** | 0.00% | -14.29% | -0.89% |
| **trace 37** | 0.00% | 0.00% | -1.73% |
| **trace 38** | 0.00% | -16.67% | -1.89% |
| **trace 39** | 0.00% | -10.64% | -1.61% |
| **trace 40** | 0.00% | 0.00% | -5.40% |
| **trace 41** | 0.00% | -9.72% | -1.43% |
| **trace 42** | 0.00% | 0.00% | -2.92% |
| **trace 43** | 0.00% | -19.88% | -2.10% |
| **trace 44** | 0.00% | 0.00% | -2.86% |
| **trace 45** | 0.00% | -13.85% | +0.15% |
| **trace 46** | 0.00% | 0.00% | -0.50% |
| **trace 47** | 0.00% | 0.00% | -0.94% |
| **trace 48** | 0.00% | 0.00% | -3.28% |
| **trace 49** | 0.00% | 0.00% | -0.50% |
| **trace 50** | 0.00% | -15.97% | -1.10% |

| | | | |
|---|---|---|---|
| **trace 51** | 0.00% | 0.00% | -0.55% |
| **trace 52** | 0.00% | 0.00% | -0.71% |
| **trace 53** | -35.29% | 0.00% | -1.52% |
| **trace 54** | 0.00% | 0.00% | -2.02% |
| **Average** | -7.61% | -3.83% | -1.29% |

**Table A.4:** Percentage change in static counters for vertex shaders. Spills = 1.5· spill stores + 1· spill loads. Shaders with very large changes are excluded from the data.

| | Maximum spilling | Maximum registers used | Maximum # of instructions executed |
|---|---|---|---|
| **trace 1** | 0.00% | 0.00% | -0.88% |
| **trace 2** | 0.00% | 0.00% | -0.18% |
| **trace 3** | 0.00% | 0.00% | -1.43% |
| **trace 4** | 0.00% | 0.00% | 0.00% |
| **trace 5** | 0.00% | -2.27% | -0.30% |
| **trace 6** | 0.00% | 0.00% | -1.37% |
| **trace 7** | 0.00% | 0.00% | -0.82% |
| **trace 8** | 0.00% | -6.74% | -1.59% |
| **trace 9** | 0.00% | -1.11% | -0.25% |
| **trace 10** | 0.00% | 0.00% | -0.26% |
| **trace 11** | 0.00% | 0.00% | -0.40% |
| **trace 12** | -61.28% | -3.51% | -1.63% |
| **trace 13** | 0.00% | -2.27% | -0.45% |
| **trace 14** | 0.00% | 0.00% | -3.88% |
| **trace 15** | 0.00% | 0.00% | -0.97% |
| **trace 16** | 0.00% | 0.00% | -0.12% |
| **trace 17** | -98.08% | -1.07% | -1.22% |
| **trace 18** | 0.00% | -0.94% | -1.88% |
| **trace 19** | 0.00% | 0.00% | -0.91% |
| **trace 20** | 0.00% | -7.39% | -1.36% |
| **trace 21** | 0.00% | -3.46% | -1.91% |
| **trace 22** | 0.00% | -2.45% | -1.92% |
| **trace 23** | 0.00% | 0.00% | -0.86% |
| **trace 24** | 0.00% | 0.00% | -0.67% |
| **trace 25** | 0.00% | 0.00% | -3.88% |
| **trace 26** | 0.00% | 0.00% | -9.38% |
| **trace 27** | -67.57% | -9.39% | -1.63% |
| **trace 28** | 0.00% | -1.40% | -1.16% |
| **trace 29** | 0.00% | -1.65% | -0.91% |
| **trace 30** | -69.50% | -3.52% | -1.34% |
| **trace 31** | 0.00% | 0.00% | -0.86% |
| **trace 32** | 0.00% | 0.00% | -7.69% |

| | | | |
|---|---|---|---|
| **trace 33** | 0.00% | -0.45% | -1.10% |
| **trace 34** | 0.00% | -3.45% | -1.91% |
| **trace 35** | 0.00% | 0.00% | -0.69% |
| **trace 36** | 0.00% | 0.00% | -0.25% |
| **trace 37** | 0.00% | 0.00% | -4.39% |
| **trace 38** | 0.00% | 0.00% | -0.31% |
| **trace 39** | 0.00% | -0.59% | -0.27% |
| **trace 40** | 0.00% | -1.39% | -1.61% |
| **trace 41** | 0.00% | -5.56% | -1.45% |
| **trace 42** | 0.00% | 0.00% | -0.27% |
| **trace 43** | 0.00% | -0.41% | -1.30% |
| **trace 44** | 0.00% | 0.00% | 0.00% |
| **trace 45** | 0.00% | -3.45% | -1.77% |
| **trace 46** | 0.00% | 0.00% | -0.12% |
| **trace 47** | 0.00% | 0.00% | -0.54% |
| **trace 48** | 0.00% | 0.00% | -1.65% |
| **trace 49** | 0.00% | 0.00% | -0.37% |
| **trace 50** | 0.00% | -7.14% | -1.52% |
| **trace 51** | 0.00% | 0.00% | -0.30% |
| **trace 52** | 0.00% | 0.00% | -0.34% |
| **trace 53** | 0.00% | 0.00% | -1.20% |
| **trace 54** | 0.00% | 0.00% | 0.00% |
| **Average** | -5.49% | -1.25% | -1.36% |

## A.2.2 Hardware Results

Table A.5 contains the difference in hardware counters for a select number of traces. The values are from the best of either only optimized fragment shaders or both fragment and vertex shaders (always from the same type of test for each trace). - signifies that the trace was not able to be tested in time for the publication of the report.

**Table A.5:** Percentage change in hardware counters. Lower is better.

| | GPU_ACTIVE | EXEC_INSTR_FMA | EXEC_INSTR_CVT | EXEC_INSTR_SFU |
|---|---|---|---|---|
| **trace 1** | -0.00% | +0.00% | +0.00% | +0.00% |
| **trace 2** | - | - | - | - |
| **trace 3** | - | - | - | - |
| **trace 4** | -2.84% | -0.67% | -5.56% | -0.01% |
| **trace 5** | -0.55% | -0.07% | -6.52% | -0.25% |
| **trace 6** | +0.25% | -0.01% | -0.01% | -0.02% |
| **trace 7** | -0.96% | -0.06% | -8.68% | +0.08% |
| **trace 8** | -0.56% | -0.26% | -11.32% | +0.23% |
| **trace 9** | +0.06% | +5.88% | -13.65% | +0.03% |

| | | | | |
|---|---|---|---|---|
| **trace 10** | +0.04% | -0.26% | -0.61% | -0.03% |
| **trace 11** | +0.33% | -0.90% | +0.70% | +0.00% |
| **trace 12** | -0.03% | -0.03% | -0.09% | -0.05% |
| **trace 13** | -0.73% | +0.19% | +1.16% | -0.47% |
| **trace 14** | +0.00% | +0.00% | -0.00% | -0.00% |
| **trace 15** | -0.00% | +0.00% | -0.00% | -0.00% |
| **trace 16** | -0.65% | -1.28% | -2.93% | +0.18% |
| **trace 17** | - | - | - | - |
| **trace 18** | +0.16% | -0.06% | -4.94% | +0.57% |
| **trace 19** | -0.81% | +3.98% | -13.96% | +0.01% |
| **trace 20** | +0.19% | -1.09% | -4.63% | +0.02% |
| **trace 21** | -1.54% | +0.82% | -4.19% | +1.32% |
| **trace 22** | +0.00% | -0.37% | +0.28% | -0.04% |
| **trace 23** | -0.02% | -0.00% | -0.01% | -0.00% |
| **trace 24** | +1.70% | +0.08% | -2.59% | +0.04% |
| **trace 25** | +1.84% | -0.19% | +0.03% | -0.59% |
| **trace 26** | -0.05% | -0.02% | -0.01% | -0.00% |
| **trace 27** | +0.33% | -0.11% | -1.66% | +0.29% |
| **trace 28** | +0.48% | -0.01% | -1.65% | +0.11% |
| **trace 29** | +0.55% | -0.02% | -0.00% | -0.03% |
| **trace 30** | +1.23% | -0.45% | -3.62% | +0.22% |
| **trace 31** | -0.94% | -1.13% | -9.96% | +0.09% |
| **trace 32** | +0.17% | -3.88% | -0.08% | +0.13% |
| **trace 33** | +0.06% | -0.06% | -0.97% | -0.50% |
| **trace 34** | -0.83% | +1.26% | -2.66% | +0.80% |
| **trace 35** | -0.35% | -0.03% | -5.48% | -0.12% |
| **trace 36** | +0.40% | +0.31% | +0.03% | +0.25% |
| **trace 37** | -0.01% | -2.21% | -4.23% | -10.79% |
| **trace 38** | +0.25% | +0.19% | -1.95% | +0.02% |
| **trace 39** | -0.35% | -0.11% | -3.98% | -0.28% |
| **trace 40** | -0.00% | +0.00% | -0.04% | -0.00% |
| **trace 41** | -0.84% | -0.46% | -10.13% | -0.12% |
| **trace 42** | +0.23% | -0.54% | -3.59% | +0.00% |
| **trace 43** | -0.10% | +0.12% | -3.85% | +0.40% |
| **trace 44** | -0.00% | -2.56% | -5.28% | -0.02% |
| **trace 45** | +2.21% | +3.59% | +3.06% | +4.93% |
| **trace 46** | +0.01% | +0.16% | -1.27% | +0.01% |
| **trace 47** | -3.08% | -1.08% | -7.44% | -0.02% |
| **trace 48** | -0.40% | +0.08% | -7.60% | +0.04% |
| **trace 49** | +1.19% | +0.22% | -7.98% | -0.05% |
| **trace 50** | -0.87% | -1.58% | -6.06% | +0.58% |
| **trace 51** | +0.24% | -0.01% | -5.31% | -0.01% |
| **trace 52** | +1.44% | +0.27% | -8.28% | +0.20% |
| **trace 53** | +0.26% | -0.16% | -9.28% | -0.00% |

| | | | | |
|---|---|---|---|---|
| **trace 54** | -0.03% | +0.00% | +0.00% | -0.51% |
| **Average** | -0.06% | -0.04% | -3.65% | -0.05% |
| | | | | |

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-08-20

**EXAMENSARBETE** Experimental Evaluation of Compiler Optimizations on Arm Mali GPUs
**STUDENT** Arne Stenkrona & Johannes Neij
**HANDLEDARE** Jonas Skeppstedt (LTH)
**EXAMINATOR** Flavius Gruian (LTH)

# Förbättring av Arm Mali grafikkort med hjälp av utvärdering av optimeringar

POPULÄRVETENSKAPLIG SAMMANFATTNING **Arne Stenkrona & Johannes Neij**

Innan kod kan köras på grafikkort behövs den översättas till maskinkod genom kompilering. För hög prestanda behöver kompilatorn utföra optimeringar. Vi har utvärderat optimeringar för att i förhand avgöra vilka som bör prioriteras.

Grafikkort används idag i stor utsräckning, inte minst i mobiltelefoner. Förbättringar i prestanda kan förbättra användarupplevelsen och minska strömförbrukningen i de miljontals enheter som förlitar sig på grafikkort. Maskinkoden som kompilatorn producerar avgör de instruktioner som grafikkortet utför. Kompilatoroptimering kan exempelvis minska mängden instruktioner som behöver utföras, samt välja mindre resurskrävande instruktioner. Detta kan ha betydande effekt för prestandan.

Utveckling inom grafikkort går fort. Ny hårdvara förändrar villkoren för kompilatorn. Tidigare optimeringar kan bli inaktuella och nya optimeringar kan krävas för att bäst utnyttja hårdvarans egenskaper. Kompilatorutveckling präglas därför av tidsbegränsningar. Vi har undersökt två olika kategorier av optimeringar, titthållsoptimering och tidsobegränsad optimering. Detta kan i förväg berätta vilka optimeringar som bör prioriteras.

Titthållsoptimering innebär att man undersöker en liten bit av koden i taget för att hitta optimeringsmöjligheter. Likt ett titthål ser man bara ett fåtal instruktioner åt gången. Dessa sekvenser undersöks för att se ifall de överenstämmer med kända ineffektiva kodmönster som kan ersättas. I vårt arbete har vi identifierat ineffektiva sekvenser och utvecklat ett redskap som ersätter dessa med färre instruktioner.

Tidsobegränsad optimering innebär att kompilatorn arbetar under en betydligt längre tid än vad som är möjligt i praktiken. Kompilering för grafikkort sker under användning och därför måste därför vanligtvis gå snabbt. I tidsobegränsad optimering undersöker kompilatorn programmet i större detalj och kan därmed öka prestandan ytterliggare. Detta kan visa outforskade optimeringsmöjligheter.

Vårt resultat visar att titthålsoptimering kan innebära påtagliga förbättringar för prestandan. Däremot krävs det att titthålsoptimeringen utförs med tillräckligt hög frekvens. När ett ineffektivt mönster har identifierats bör det också säkerställas att mönstret förekommer ofta innan resurser används för att implementera optimeringen i kompilatorn. För tidsobergränsad optimering var det svårt att hitta ett direkt samband mellan statiska resultat och resultat på hårdvara. Vi minskade dock mängden enkla matematiska intruktioner samt hittade potentiella effektiviseringar i kompilatorn.