

MASTER'S THESIS 2025

Enabling Profile Guided Optimizations (PGO) for Graphics

Emma Jansson

Elektroteknik
Datateknik

ISSN 1650-2884

LU-CS-EX: 2025-13

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY



EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2025-13

**Enabling Profile Guided Optimizations
(PGO) for Graphics**

Aktivera profilstyrda optimeringar för
grafik

Emma Jansson

Enabling Profile Guided Optimizations (PGO) for Graphics

Emma Jansson
`emma.jansson@live.se`

June 7, 2025

Master's thesis work carried out at Arm Sweden AB.

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

Examiner: Christoph Reichenbach, `christoph.reichenbach@cs.lth.se`

Abstract

This master thesis presents an implementation for enabling profile-guided optimizations (PGO) for mobile phone GPUs. PGO is an optimization technique that uses runtime profiling data, like block frequency and function call frequency, to guide compiler optimizations. The implementation is done by adapting the existing PGO infrastructure in LLVM to serve the architectural differences between CPUs and GPUs.

Analyzing how PGO functions in LLVM/Clang for CPUs, necessary adaptations were identified to fit the GPU architecture. A major difference includes GPUs not being able to allocate their own memory, requiring modifications to where profile counters and necessary profile information about each function are stored. For this, a new compiler pass has been developed to collect the important profile information for each function, while the storing of profile counters has been outsourced to the GPU driver, which also manages writing the final profile data file that is later given back to the compiler.

Evaluation of exposing the collected PGO data to existing compiler passes shows valuable performance improvements, with up to 3.74% fewer clock cycles and 6.32% fewer message instructions between threads. This proves that enabling PGO for graphics can lead to more efficient execution of workloads on GPUs.

Keywords: MSc, GPU, PGO, LLVM, Compiler optimization

Acknowledgements

I would like to express my deepest gratitude to my supervisors at Arm: Tobias Gutzmann, Mikael Nilsson, and Simone Pellegrini. Thank you for your great support in both technical and writing matters, and for always answering my endless questions in less than five minutes. This work would not have been possible without your guidance.

A special thank you to my supervisor at the institution, Jonas Skeppstedt, for your support and for the excellent university courses that lead me into this exciting field.

Thank you to my sister, parents, grandmother, and boyfriend for their care, for helping me organize my thoughts during stressful times, and for trying to understand what a compiler is.

Contents

1	Introduction	7
1.1	Research questions	7
1.2	Contributions	8
1.3	Approach	8
2	Background	11
2.1	Theory	11
2.1.1	GPU	11
2.1.2	PGO	12
2.1.3	LLVM	14
2.1.4	PGO with Clang/LLVM	16
2.2	Previous work	20
2.2.1	Sample-based vs Instrumentation-based PGO	20
2.2.2	Optimizations	21
2.2.3	PGO in LLVM	22
3	PGO Implementation Details	25
3.1	Arm Mali GPU	25
3.1.1	Rendering pipeline	25
3.1.2	Architecture	26
3.1.3	Memory allocation	26
3.2	PGO	27
3.2.1	Compiler pass	27
3.2.2	Buffer allocation	28
3.2.3	PGO file writing	29
3.2.4	PGO file utilization	30
4	Evaluation method	33
4.1	GPU traces	33
4.2	Measurements	33

4.2.1	Collect PGO data	34
4.2.2	Hardware counters	34
4.2.3	Selected traces	35
4.3	Verification	35
5	Results	37
5.1	PGO in LLVM	37
5.1.1	Static results	37
5.1.2	Hardware results	38
5.1.3	Spilling performance impact	39
5.1.4	Optimizations	41
5.2	Collection of PGO data	41
6	Discussion	45
6.1	Static results	45
6.2	Hardware results	46
6.2.1	Single frame optimization	47
6.3	Spilling performance impact	47
6.4	Optimizations	48
6.5	Collection of PGO data	48
7	Conclusion	49
7.1	Future work	50
7.1.1	Investigate performance improvement	50
7.1.2	Value profiling	50
7.1.3	Optimizations	51
7.2	Threats to validity	51
	References	53

Chapter 1

Introduction

In modern computing, profile-guided optimizations (PGO) is a widely used technique to improve program performance by collecting runtime data and using it to guide compiler optimizations. While PGO is a well-established technique in CPUs, its application in GPU environments, where workloads are highly parallel and dynamically structured, is largely unexplored. Existing GPU compilers are static, meaning they are applying the same optimizations across all programs and code paths, not adapted to real-world execution patterns. This thesis explores how PGO can be enabled for GPUs in mobile phones to improve performance. While the initial focus is on mobile GPUs, this implementation is expected to be transferable to GPUs in other platforms, such as laptops and desktops.

GPUs are highly parallel processors, handling extensive workloads distributed across a vast number of threads. This makes optimization challenging but important. A minor improvement can have a significant effect when scaled across thousands of threads. Exposing profiling information, such as block frequency and branch probability, to the GPU compiler can potentially improve performance. However, adapting the existing PGO infrastructure in LLVM presents significant challenges, as CPUs and GPUs differ in many ways, most notably by GPUs being highly parallel and lacking the ability to allocate their own memory.

This thesis investigates these challenges and examines how leveraging profiling data can guide and improve optimizations within the LLVM framework.

PGO is also known by other names, such as feedback-directed optimizations (FDO) or profile-based optimizations (PBO). This thesis will use the term PGO, but they all refer to the same technology.

1.1 Research questions

The following research questions will be explored in this thesis:

1. How can the existing PGO infrastructure in LLVM be adapted to support GPU architectures?

2. What is the impact of exposing profiling information, such as block frequency, to existing GPU compiler passes on the performance of the generated code?
3. What types of novel GPU-specific optimizations can be developed by leveraging the gathered profiling data?

1.2 Contributions

This thesis explores how PGO techniques, typically used for CPU optimizations, can be adapted and applied to GPUs to improve the performance of graphics applications. This thesis makes the following contributions:

- Implementing a PGO infrastructure to collect and utilize profile data for GPU execution.
- Demonstrating performance improvements on GPUs by leveraging PGO in LLVM, with improvements of up to 3.74% fewer clock cycles, when the compiler utilizes profiling data.
- Make suggestions about GPU-specific optimizations that could be developed by leveraging profile data.

1.3 Approach

This section outlines the approach for the research conducted in this master's thesis. The aim was to enable PGO for GPUs in mobile phones and evaluate its impact on performance.

Since PGO is a well-known approach utilized in CPUs, the first step was to study how PGO functions in Clang/LLVM. That study is found in section 2.1.4. It can be seen in the study that Clang/LLVM uses three PGO specific LLVM passes, `PGOInstrumentationGen` and `InstrProfiling` for generating instrumentation, and `PGOInstrumentationUse` for using the collected PGO data.

Due to significant differences between CPUs and GPUs, with the GPU not being able to allocate its own memory, the LLVM pass `InstrProfiling` is exchanged with a new pass adapted to the requirements of the GPU. The details for the implementation and how LLVM is adapted to support the GPU architecture are described in chapter 3.

To evaluate the impact on performance, we look at hardware counters collected while running GPU traces on an FPGA board simulating a Mali Gxx GPU. Three runs are made for each trace, and runs one and three are compared:

1. Baseline run, without PGO enabled.
2. Runs to collect PGO data.
3. Optimizing run, using the collected PGO data.

The runs to collect PGO data are done on selected frames from the trace. The frames are meant to represent the whole trace and the data simulates a user running an application. However, data collected from one user could be used to optimize for another user running the same application. The evaluation methodology is explained in chapter 4, and the results are shown in chapter 5.

Due to time limitations, possible GPU-specific optimizations are proposed in section 7.1.3 as future work, focusing on minimizing branch divergence and the optimization possibilities with value profiling.

Chapter 2

Background

This chapter begins by presenting all the necessary theory needed to understand the implementations and decisions made in this thesis. This is followed by a section on previous work related to PGO that is relevant to this thesis.

2.1 Theory

This section aims to explain all the necessary theory for understanding this thesis. It explains what a GPU is and how PGO works, which is the main focus for this thesis. The most basic and important things to know about LLVM are also presented to understand the implementation part for this thesis. Lastly, a study how PGO works for Clang/LLVM is presented.

2.1.1 GPU

A GPU is a processor designed specifically for rendering images on devices with screens, such as computers and smartphones [36]. Prior to the development of GPUs, graphics controllers in PCs and video game consoles relied on the computer's central processing unit (CPU) to execute graphics-related tasks. The CPU is often referred to as the main processor of a computer and can be seen as its "heart" and "brain". It runs the machine's operating system and apps, which involves numerous instructions to interpret, process, and execute [3]. However, as personal gaming and computer-aided design (CAD) became more demanding, the industry needed a faster and more efficient way of rendering images.

The GPU was specifically developed for managing demanding graphics and extensive workloads, constantly performing complex mathematical calculations. Therefore, the architecture includes smaller, more numerous, and more specialized cores that are designed to process multiple small tasks in parallel. These tasks are assigned to warps.

Warps

A warp is a group of threads in a GPU that share the same Program Counter (PC) and executes the same instruction simultaneously [22][13]. Within a warp, threads share resources such as memory, which is beneficial for collaboration and overall performance. Grouping threads into warps simplifies thread management and data sharing. The number of threads in a warp depends on the GPU architecture.

Shader Programs

Shader programs are used to render an image the way the user wants and determine the visual output of the running application [1]. They give the developer control over what should be executed in the rendering pipeline on the GPU [2]. Shaders are written in high-level shading languages such as OpenGL Shading Language (GLSL) [39]. This code is compiled to an intermediate representation by the GPU driver. An intermediate representation can also allow shader programs to be compiled and stored offline. However, in real-time rendering applications, shader code is typically not pre-compiled as it needs to be generated procedurally during runtime.

Uniforms

Shader programs can contain uniforms, which can be explained as global variables for the shader. Their values are set before the shader gets executed and will remain constant for every iteration of the shader [1]. The values for the uniforms can be changed the next time the GPU is asked to execute the shader. For example, a certain type of shader may provide the color of a light source as a uniform value. The number of available registers for uniforms is large as the uniform inputs are stored once for each shader and then reused through the entire iteration.

2.1.2 PGO

PGO is an optimization technique that utilizes runtime profiling data, e.g. block frequency and function call frequency, to guide compiler optimizations [12]. Figure 2.1 shows an overview of how PGO works.

During the first compilation phase, the source code is compiled as normal, with the normal optimizations. That gives an executable that, when it is executed, generates runtime statistics (the profile data). The same source code is then compiled a second time with access to the profiling data. This profile data gets integrated into the optimizations to make them more powerful. To give an example of when PGO is useful, let's assume we have the C code shown in listing 2.1.

When compiling, the compiler typically aims to inline the callee functions, `bar` and `baz` in this case, into the caller `foo`, to improve performance by eliminating function call overhead. However, the compiler can only inline one of these functions or the size of the final binary will be too large. Without profiling data, the compiler lacks the information to determine which function is executed most frequently and would benefit most from inlining. To solve this, PGO can be used according to the flow explained in figure 2.1 and obtain the necessary profiling information to make the best inline decision for this case.

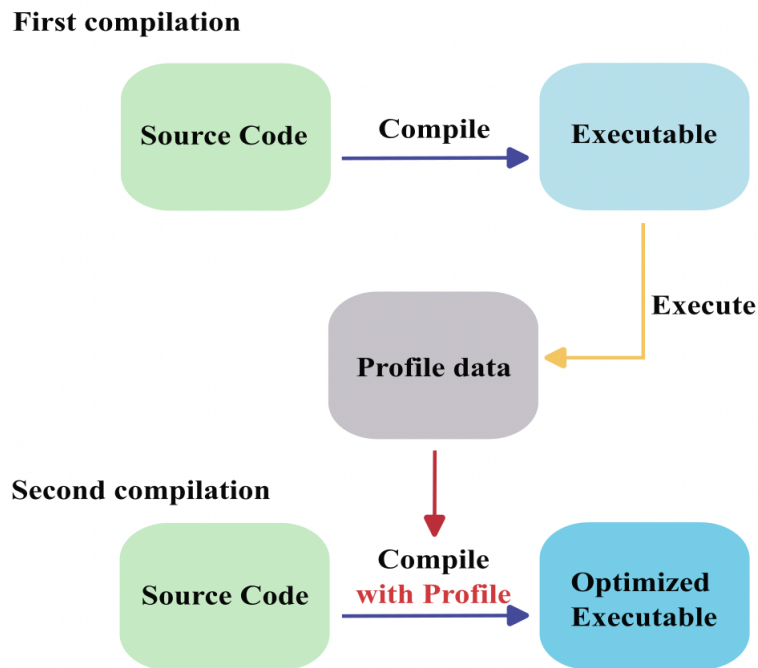


Figure 2.1: Overview of the PGO workflow. The same source code is compiled twice: the first compilation and execution generate profile data, which the second compilation uses to produce a more optimized executable.

```

1 void foo(int i) {
2     if (i == 1)
3         bar();
4     else
5         baz();
6 }
  
```

Listing 2.1: An example of a C program that might require PGO before the compiler can optimize it.

There are primarily two ways of collecting the profiling data, sample-based or instrumentation-based profiling, and both are described below. This thesis implements instrumentation-based profiling, and the motivation for this choice is discussed in section 2.2.1.

Instrumentation-Based Profiling

Instrumentation-based PGO inserts instrumentation code into the target program during the first compilation phase [12]. This instrumentation gets inserted into specific parts of the program, for example, basic blocks and functions, and the results are written to a file when the instrumented program is executed.

An advantage of instrumentation-based PGO, compared to sample-based PGO, is that it usually generates more precise profiling data since the compiler can insert instrumentation

where it gives the most benefit for the optimizations. However, inserting instrumentation also means modifying the target program, which will give a regression in performance.

Sample-Based Profiling

Sampling-based PGO depends on external tools to collect profiling data [12]. This means that there is no need to modify the target program to collect the data, and therefore its impact on performance is minimal. However, frequent sampling can still introduce some overhead.

The downside with sampling-based PGO is that it is more difficult to map the profiling data back to the original source code during the second compilation, which makes it less precise. This is due to aggressive optimizations changing the source code, for example merging two blocks, which makes it difficult to correlate the execution count from the merged blocks in the binary back to the two blocks that were merged [11].

2.1.3 LLVM

LLVM is a framework for compiler optimization and code generation [12] [30]. It helps developers create high-quality optimizing compilers by offering hundreds of libraries and thousands of different APIs, providing an extensive set of features to manage the challenging task of building a production-ready compiler. The APIs are used to generate instructions in a format known as LLVM Intermediate Representation (LLVM IR). LLVM then compiles LLVM IR to a standalone binary that matches the instruction set of a specific processor, as LLVM is designed to support multiple architectures and targets.

LLVM Pass

Transformations to the LLVM IR are done with LLVM Passes [12]. Passes are a fundamental component of LLVM and are designed to optimize the code, perform static analysis, or gather information for further optimizations. The Passes can be chained together, allowing multiple transformations to be performed sequentially, known as a Pass pipeline.

LLVM IR

As mentioned, LLVM IR is a target-independent intermediate representation used for program analysis and compiler transformation. The primary optimizations that improve the performance and reduce the size of the input program after the compilation process are generally performed at the LLVM IR level [12].

LLVM IR has a hierarchical structure, shown in figure 2.2. The top of the hierarchy is referred to as a module, which usually represents a source file. Each module contains functions and global variables. The functions contain a list of basic blocks (BB), each of which contains a list of instructions. All instructions in a basic block will be executed if the basic block is entered. This hierarchical structure makes it simpler to iterate over a program and find interesting parts of the code where transformations or analysis algorithms could act.

Listing 2.2 shows the C++ code of a simple function. The corresponding IR for this function is shown in listing 2.3, with comments on rows 7, 8, and 9 marking the IR units from

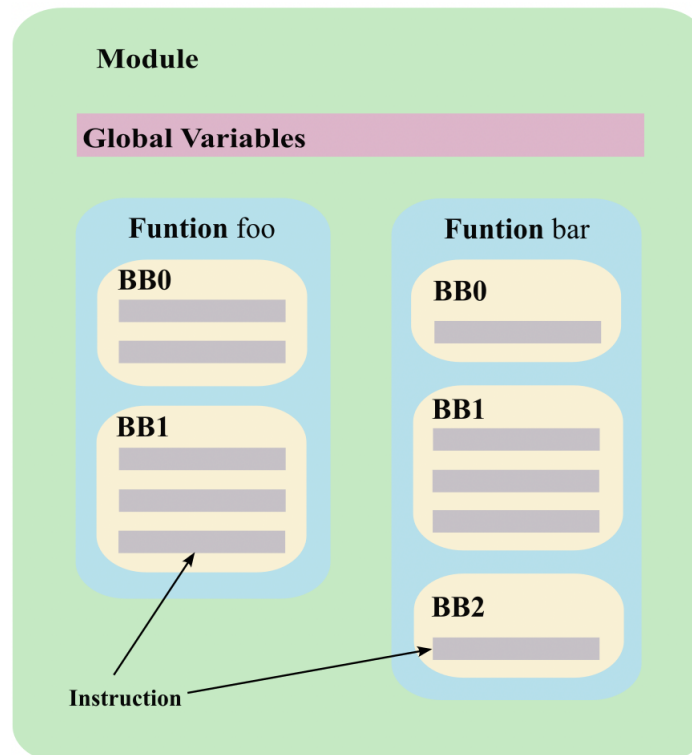


Figure 2.2: An illustration of the LLVM IR hierarchy structure, showing the relationships between modules, functions, basic blocks (BB), instructions, and global variables.

figure 2.2. Each line of LLVM IR corresponds to operations such as loading a variable, performing arithmetic's, and returning the result. Comments have been added to listing 2.3 to illustrate how the LLVM IR corresponds to the C++ code in listing 2.2. This provides insight into how the program will be executed by the hardware.

```

1 int func(int x) {
2     int y = x;
3     for (int i = 0; i < x; ++i) {
4         y *= x;
5     }
6     return y;
7 }

```

Listing 2.2: A C++ example of a simple function containing a for-loop.

```

1 ; ModuleID = 'code.cpp'
2 source_filename = "code.cpp"
3 target datalayout = ...
4 target triple = ...
5
6 ; Function Attrs: ...
7 define i32 @func(i32 %x) #0 {    // Function
8 entry: // Basic Block
9     %x.addr = alloca i32, align 4    // Instruction
10    %y = alloca i32, align 4         // Allocate memory for y

```

```

11  %i = alloca i32, align 4          // Allocate memory for i
12  store i32 %x, i32* %x.addr, align 4
13  %0 = load i32, i32* %x.addr, align 4
14  store i32 %0, i32* %y, align 4    // int y = x (row 2)
15  store i32 0, i32* %i, align 4     // int i = 0 (row 3)
16  br label %for.cond              // Branch to execute for condition
17
18  for.cond:                        ; preds = %for.inc, %entry
19  %1 = load i32, i32* %i, align 4
20  %2 = load i32, i32* %x.addr, align 4
21  %cmp = icmp slt i32 %1, %2        // i < x (row 3)
22  br i1 %cmp, label %for.body, label %for.end
23      // if (i < x) execute loop body; else return;
24
25  for.body:                        ; preds = %for.cond
26  %3 = load i32, i32* %x.addr, align 4
27  %4 = load i32, i32* %y, align 4
28  %mul = mul nsw i32 %4, %3         // y * x (row 4)
29  store i32 %mul, i32* %y, align 4  // y = y * x (row 4)
30  br label %for.inc               // Branch to execute ++i
31
32  for.inc:                         ; preds = %for.body
33  %5 = load i32, i32* %i, align 4
34  %inc = add nsw i32 %5, 1          // i + 1 (row 3)
35  store i32 %inc, i32* %i, align 4  // i = i + 1
36  br label %for.cond, !llvm.loop !8
37      // Branch to execute for condition
38
39  for.end:                         ; preds = %for.cond
40  %6 = load i32, i32* %y, align 4
41  ret i32 %6                       // return y (row 6)
42 }

```

Listing 2.3: LLVM IR for the C++ example in listing 2.2, with comments indicating the corresponding LLVM IR units from figure 2.2, and referencing the corresponding lines from the original C++ code.

LLVM intrinsics

The compiler can inject intrinsic functions during code optimizations at IR level. Intrinsics are built into the compiler, which has predefined implementations for them. During compilation, the compiler replaces the intrinsics with optimized machine instructions for a certain backend [34]. LLVM has a reserved word for these intrinsic functions, and therefore, the function names will start with the prefix "llvm.". These functions are always external, so developers can only call them but cannot modify their implementation.

2.1.4 PGO with Clang/LLVM

Since the implementation of PGO in CPUs is well-established, it inspires this thesis implementation on the GPU. To show an example of how PGO functions in CPUs, a simple C/C++

program is compiled using the Clang/LLVM tool for profiling [37], and the LLVM IR output is studied.

Compile with profile instrumentation

The C++ example in listing 2.2 is reused for this study to show the difference in the IR with and without the inserted profiling instrumentation. The following command compiles the program into readable LLVM IR:

```
clang --target=arm -fprofile-generate -S code.cpp \
-mllvm -print-after-all
```

The target for compilation is set to Aarch64 [17], profiling is done with instrumentation by the flag `-fprofile-generate`, and `-mllvm -print-after-all` prints the IR after each pass.

Profile instrumentation generation

The program gets instrumented by two passes, the first pass is called `PGOInstrumentationGen` followed by the second pass `InstrProfiling`.

The first pass determines where in the IR the profile counters should be inserted and inserts intrinsics in those places looking like this:

```
1 define i32 @func(i32 %x) #0 {
2 entry:
3   call void @llvm.instrprof.increment(...) // Inserted intrinsic
```

Where the intrinsics get inserted is based on a paper by Knuth and Stevenson [14], which presents an algorithm for calculating the minimum number of measurement points needed to accurately determine all execution frequencies.

The `InstrProfiling` pass job is to lower these intrinsics, i.e., replace them with simple load, add, store instructions. It is also responsible for managing the data structures and initialization code required to update the counters accurately and to emit the profile at program termination. The following part explains more in-depth how the instrumented code will look after the instrumentation passes have been run.

The instrumented IR contains three global variables:

```
1 @__profc_func = private global [2 x i64] zeroinitializer ...
2 @__profd_func = private global { i64, i64, i32, i8*, i8*, i32, [2
   x i16] } ...
3 @__llvm_prf_nm = private constant [18 x i8] ...
```

- `__profc_func` is a global array of size 2. The size corresponds to how many instrumentation counters have been inserted into that function.
- `__profd_func` is a global struct containing metadata to map profile counters back to instrumented code areas [32].
- `__llvm_prf_nm` is a constant with a concatenated string of all function names in the program.

As this example only has one function, it will contain only one of each variable `profc` and `profd`. Good to know is that each function in a program gets its own of these variables. Therefore, as the number of functions increases, so does the number of corresponding `__profc_<function_name>` and `__profd_<function_name>` variables.

Continuing, increments for the counters in `__profc_func` are inserted in two places for this example. This is where the intrinsics were first inserted by the pass `PGOInstrumentationGen`, and will be lowered as shown in listing 2.4 below.

```

1 define i32 @func(i32 %x) #0 {
2   entry:
3     ...
4     %pgocount = load i64, i64* getelementptr inbounds ([2 x i64], [2
      x i64]* @__profc_func, i32 0, i32 0), align 8
5     %0 = add i64 %pgocount, 1
6     store i64 %0, i64* getelementptr inbounds ([2 x i64], [2 x i64]*
      @__profc_func, i32 0, i32 0), align 8
7     ...
8     br label %for.cond
9
10  for.cond:                                ; preds = %for.inc, %entry
11    ...
12    br i1 %cmp, label %for.body, label %for.end
13
14  for.body:                                ; preds = %for.cond
15    %pgocount1 = load i64, i64* getelementptr inbounds ([2 x i64],
      [2 x i64]* @__profc_func, i32 0, i32 1), align 8
16    %0 = add i64 %pgocount1, 1
17    store i64 %0, i64* getelementptr inbounds ([2 x i64], [2 x i64]*
      @__profc_func, i32 0, i32 1), align 8
18    ...
19    br label %for.inc
20
21  for.inc:                                  ; preds = %for.body
22    ...
23    br label %for.cond, !llvm.loop !8
24
25  for.end:                                  ; preds = %for.cond
26    %6 = load i32, i32* %y, align 4
27    ret i32 %6
28 }
```

Listing 2.4: Instrumented IR, including additional instructions inserted by the compiler to collect runtime profile information.

This will give profiling information of how many times the function is executed and how many times the for body is executed.

Lastly, some static calls are inserted to register the `__profd_func` and `__llvm_prf_nm` variables.

```

1 call void @__llvm_profile_register_function(@__profd_func)
2 call void @__llvm_profile_register_names_function(@__llvm_prf_nm)
```

Listing 2.5: Static calls to register profiling variables.

These calls ensure that the profiling runtime is aware of these data structures and can properly associate them with the corresponding functions and counters. At runtime, this registration process allows the profiling library to collect the execution data stored in these variables, making it possible to map the gathered profile information back to specific sections of the code.

Profile data file

By default, the profile data will be written to a binary file, called `default.profraw` if no filename is given, in the current directory when the first generated executable is run. There exists a runtime library called `compiler-rt` which contains an implementation of the low-level specific hooks required by code generation and other runtime components, one of them being profiling [28]. The binary file gets created and written with the profile information by the profile runtime.

The runtime initialization hook calls `__llvm_profile_initialize`, which is responsible for gathering data from variables `__profc_func`, `__profd_func` and `__llvm_prf_nm` and write them to the `default.profraw` file. This function calls the C library function `atexit(__llvm_profile_write_file)` which will make sure the file `default.profraw` gets written to when the program terminates [8].

The binary file is composed of a header, data, counters, and name variables. The header's content can be summarized as table 2.1 [8], with the values for this example in parenthesis. The counters delta corresponds to the in-memory address difference between profile meta-data (`profd`) and the start of the counters [29]. Value kind last seems to be always 1 [8].

00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F
magic	version
number of <code>profd</code> variables (1)	padding before counters
number of total counters in the program (2)	padding after counters
the length of the <code>__llvm_prf_nm</code> variable (18)	counters delta
address of <code>__llvm_prf_nm</code>	value kind last

Table 2.1: Header content for `default.profraw`.

After the header follows the data part where each `profd` variable get its own section [8]. The `profd` variable in this case contain seven fields, the six of these will be written to the binary file and those fields are [29]:

1. The hash of the function's PGO name.
2. The function's CFG hash.
3. The in-memory delta to the counters this function corresponds to.
4. The address to the function it refers to.
5. The pointer to the value profiles¹, if any.

¹Value profiling tracks the values that variables and expressions take during runtime, along with the frequency of those values. This feature is not implemented in this thesis, as explained further in section 7.1.2.

6. The number of counters for the instrumented function.

After follows the counters that are stored contiguously and in an order that is consistent with the instrumentation [31], and lastly the `__llvm_prf_nm` is written as the final part of the `default.profraw` file [8].

To be able to use the profile data in the second compilation phase, the tool `llvm-profdata` [37] has to be used to change the file format to the format expected by the compiler, clang in this case.

```
llvm-profdata merge -output=default.profdata default.profraw
```

And after that the profile data can be utilized to generate an even more optimized executable with the following command:

```
clang --target=arm -fprofile-use=default.profdata code.cpp \  
-o code
```

Profile instrumentation utilization

The pass that interprets the profiling data file is called `PGOInstrumentationUse`. It will read the edge count profiles and annotate the block counts.

It builds on the algorithm [14] that the `PGOInstrumentationGen` pass uses, so when `PGOInstrumentationUse` is finished, each block will have a count set to it, even if the block didn't have a counter inserted, thanks to the algorithm.

2.2 Previous work

This section presents previous work related to PGO and optimizations that have been done based on PGO. To begin, section 2.2.1 explains two different sample-based PGO tools and why instrumentation-based PGO is beneficial for GPUs despite the benefits of sample-based PGO. Section 2.2.2 details optimizations that have been done on GPUs utilizing PGO and could be of interest to this thesis. Lastly, section 2.2.3 explains how existing LLVM passes primarily utilize PGO.

2.2.1 Sample-based vs Instrumentation-based PGO

As described in section 2.1.2, there exist two types of PGO: sample-based and instrumentation-based. A lot of previous work has been done on sample-based PGO to make it more convenient to use for developers, despite it being dependent on external tools. In [27], a tool called `SamplePGO` is presented that has been developed to target this and be more accessible for developers.

The traditional way of doing PGO is by inserting instrumentation. Novillo finds in [27] up to 40% performance improvements in many internal applications at Google when using instrumentation-based PGO with GCC. This makes PGO essential in their optimization

pipeline. However, these performance improvements come with some costs that make developers hesitant to use it as it requires significant modifications to the build process. The inserted instrumentation also increases runtime overhead which also makes it less attractive.

To make PGO more accessible for all developers, [7] developed a PGO model in the GCC compiler that is sample-based. With inspiration from that, SamplePGO describes an LLVM implementation of a sample-based optimization model, however, it is not completely LLVM as the sampling is done by an external tool that utilizes available hardware performance counters. A tool to convert the collected profile to the desired format expected from LLVM needs to be used before LLVM can read it and incorporate it into the IR.

As instrumentation-based PGO is chosen for this thesis implementation, implementation details for SamplePGO are not important, but its results, presented in [27], are more interesting and provide valuable insights into the performance improvements achievable with PGO. Using the GCC implementation of SamplePGO on their internal benchmarks including, for instance, big tables, indexing, video conversion, and compression, the authors experience peak improvements up to 30% with a geometric mean performance improvement of 7% to 9%. In comparison, instrumentation-based PGO achieves peak improvements of up to 40% with a geometric mean between 6% and 11%.

AutoFDO [6] is another sample-based profiling tool that was implemented by Google for the same reasons as SamplePGO, i.e. usability. When Google compared AutoFDO’s performance gains against FDO, they found that AutoFDO achieves approximately 85% of the performance improvement of instrumentation-based PGO. However, while benchmarks suggest this is the case, many developers have found that, in practice, the performance is equivalent, likely because more representative input compensates for the less precise profiles.

Both [27] and [6] agree that sample-based PGO is more beneficial for simpler optimizations like inlining, which is where most of the performance gains come from. However, for workloads that require peak performance, instrumentation-based PGO will likely remain the preferred choice. We believe that for highly parallel and compute-intensive GPUs, where simple optimizations like inlining are always applied, instrumentation-based PGO is the best choice for obtaining more precise profile data, leading to better optimizations.

[27] highlights an important issue, stating that LLVM optimizers are not fully utilizing PGO, as many passes in LLVM do not make use of profile data if it is provided. As a result, the developers of SamplePGO find that the performance gains from SamplePGO in LLVM are no more than 5%.

2.2.2 Optimizations

Although PGO has been largely unexplored for GPUs, some previous work on optimizations for GPUs based on PGO exists and is further explained below.

Zeroploit

Zeroploit [33] is one optimization that has been developed for shader programs in modern gaming applications on an NVIDIA® GeForce RTX™ 2080 GPU. By using an offline value profiler, the developers of *Zeroploit* observed that one of the register operands of several multiply, logical-and, and similar operations is likely to be dynamically zero. To take advantage

of this behavior, *Zeroploit* introduces a zero-specialized fast path and a default slow path. The fast path provides performance gains in two ways:

- If one operand is zero, the computation of the other operand (its backward slice) can often be dynamically skipped, provided that the operand is only used in the given instruction.
- Future instructions dependent on the given instruction can be zero-specialized, potentially triggering additional backward slice specializations.

Zeroploit achieves an average frame-rate improvement of 2.8%, and an average speedup of 35.8% for targeted regions. This proves that there are possible performance gains in understanding operand value locality in modern gaming applications. However, the authors implemented *Zeroploit* manually for a research purpose, and highlight that more work is needed on how to productize this optimization along with a value profiler.

Branch divergence

Branch divergence occurs when threads within the same warp follow different execution paths due to conditional branches. This will lead to serialized execution and has a significant impact on the performance of GPU programs. In [35], the authors come up with a novel profile-guided approach to identify and optimize divergent branches. The key idea for this paper is to focus on branches inside data-parallel loops that can potentially cause branch divergence. The authors approach is to instrument all branches in the parallel loops by adding profiling function calls that prints **L** for the left basic block and **R** for the right basic block. This instrumented program is then executed to generate profiling data. By leveraging repetitive branch patterns, loops can be reconstructed so that corresponding iterations traverse the same branch path, and the warps become convergent.

The authors tested their approach on Rodinia benchmarks [5], which is well-known and includes applications that target GPU platforms, and found that it is possible to achieve up to 48% performance improvements.

2.2.3 PGO in LLVM

As mentioned above in section 2.2.1, not many LLVM passes make use of profile data when it is provided. However, some passes do [23], and their primary focus will be briefly described in this section.

Three main types of passes make use of profile data: control flow graph (CFG) simplification, loop transformation, and instruction scheduling.

Profile data helps simplify the CFG by identifying infrequently taken branches and rarely executed paths. If a branch is rarely taken, it can be removed or redirected, reducing the number of conditional branches and branch mispredictions. Profile data also supports the compiler in prioritizing frequently executed paths when merging blocks and optimizing conditional statements, improving the execution efficiency of hot paths.

Block placement also benefits from profile information. By placing blocks from a hot path sequentially in memory, unnecessary jumps are avoided, and cache locality is improved.

Furthermore, profile data improves loop behavior predictions, allowing the compiler to make more informed decisions for loop transformations. Common transformations include loop unrolling and loop unswitching, which improve execution efficiency when a loop is executed frequently. However, these transformations increase code size, which can place additional pressure on the instruction cache. If a loop is rarely executed, the overall performance gain from these transformations is minimal and it is therefore better to avoid increasing the code size.

Another loop optimization involves moving loop-invariant computations to be executed outside of the loop to reduce redundant execution. However, this optimization is only beneficial if the source block inside the loop is executed more frequently than the destination block outside the loop. Profile data helps determine which instructions that would benefit from being hoisted.

With multiple passes in LLVM leveraging profile data, their impact will be evaluated in the tests conducted for this thesis to address RQ2.

Chapter 3

PGO Implementation Details

The goal for the implementation was to adapt the existing PGO infrastructure in LLVM to support GPU architectures. This implementation of PGO is carried out on the Arm Mali GPU. The first section gives an overview of the Arm Mali GPU and its impact on the implementation. This is followed a detailed description of the PGO implementation.

3.1 Arm Mali GPU

This section gives an overview of the Arm Mali GPU. It begins with an explanation of the rendering pipeline to provide context for what programs run on the GPU, followed by a brief explanation of the architecture and lastly a section about memory allocation on the GPU.

3.1.1 Rendering pipeline

The rendering pipeline refers to the sequence of processes through which a complete image transitions from an application request to the expected visual output [2]. Figure 3.1 gives a simplified overview of the pipeline.

The CPU, which is the first stage of the pipeline, runs the application and the GPU driver, and is responsible for sending the rendering commands to the GPU. The command processing stage receives the rendering commands from the CPU. The command processing stage's job is to process the information in the rendering commands and give instructions to the geometry processing step. The geometry processing stage will then produce a set of vertices and primitives based on the information it receives from the command processing stage. Lastly, the pixel processing stage will color the vertices and primitives and write them out to the frame buffer.

The last two steps, in the rendering pipeline, do many things to produce the wanted image, but what matters for this thesis is that they contain two different mandatory shaders. A vertex shader is located in the geometry processing step and draws the vertices. The pixel

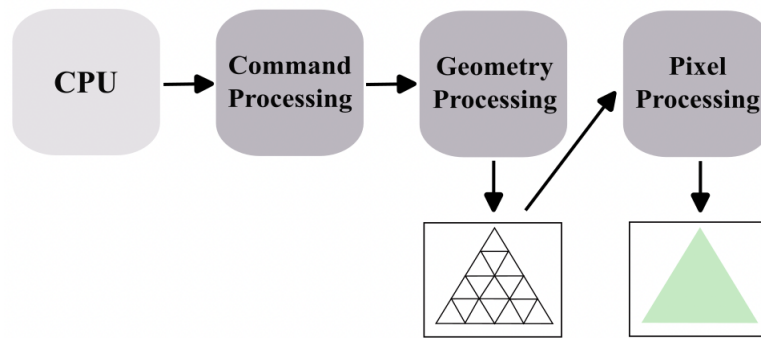


Figure 3.1: Simplified overview of the rendering pipeline for the Arm Mali GPU.

processing stage contains a fragment shader, which does the coloring. The shaders are what the programmer can control and what the GPU compiler receives as input, i.e. the shaders will get the PGO instrumentation inserted.

3.1.2 Architecture

The Arm Mali GPU has a unified shader core architecture, meaning that it only has a single hardware shader processor design, which can execute all types of shader code [15]. Mali GPUs are tile-based renderers, meaning all geometry processing for a render pass must finish before pixel processing can start. A render pass is a single execution of the rendering pipeline [16]. Geometry processing and pixel processing can run in parallel for different render passes. The workload for a single render pass is typically large and highly parallelizable, so the hardware will distribute it across all the shader cores that are available in the GPU. This parallel execution needs to be considered when implementing PGO.

3.1.3 Memory allocation

In old graphics APIs the resource owns the memory, i.e., the memory is allocated when the resource is created, which gives no control over where that memory is placed. In modern graphics APIs, it is up to the user to allocate memory [24]. The reason behind this is that it can be more efficient, as the user, if they have knowledge about the driver and the memory architecture of the device it is running on, can make informed memory allocations.

This is the reason why the compiler won't be able to insert data resources into the IR as the profiling instrumentation does for clang as shown in section 2.1.4. Since the modern graphics APIs don't know how to allocate memory for it, neither does the GPU. Just like the graphics APIs, the driver is responsible for allocating the memory where the data and counters will be stored.

Most embedded graphics systems, including Arm GPUs [20], use a unified memory architecture meaning the CPU and GPU share memory. To manage this shared memory, the GPU has a Memory Management Unit (MMU) [9]. The application running on the CPU will have allocated memory and set up all the data required to render a scene. The GPU's MMU make sure the scene data is available to the GPU by mapping the memory from the CPU's address

space into the GPU's address space. It is the MMU that will assign the actual memory for the data and counters allocated by the driver in this case.

3.2 PGO

The study in section 2.1.4 sets the starting point for this implementation. For this to work on the GPU, the architectural features of the Arm Mali GPU, stated in section 3.1, need to be considered. This implementation was done in four steps:

- Write a compiler pass that modifies the IR based on GPU specific requirements.
- Allocate the counter buffer that will replace the `profc` array.
- Write the `default.profrac` file.
- Give the profile data file back to the program and use it.

3.2.1 Compiler pass

As described in section 2.1.4, the program gets instrumented by two LLVM passes when running on the CPU. The first pass, `PGOInstrumentationGen`, inserts instrumentation in the least amount of places necessary according to a well worked out algorithm. As this implementation also needs instrumentation inserted, that pass was added to the pass pipeline. The second pass, `InstrProfiling`, lowers the intrinsics, but it also inserts data resources and static function calls to register these data resources. As explained in section 3.1.3, this will not work as the GPU will not be able to allocate memory for the data resources. Therefore, a new pass was written to lower the intrinsics and gather the important function data (the data stored in `profd` struct from the example in section 2.1.4), without writing it to the IR. Replacing the `profc` array with a static buffer and writing the profile data file are outsourced to the GPU driver.

To be able to lower the intrinsics, one will need to know where in the IR they are located. Thanks to the structure of LLVM IR, as shown in figure 2.2, iterating over the program is simple. The intrinsic instructions were collected according to algorithm 1.

Algorithm 1 Collect intrinsics

```

for all Function F in Module M do
  for all BasicBlock BB in F do
    for all Instruction I in BB do
      if I is a profile intrinsic then
        Save I to an array
      end if
    end for
  end for
end for

```

Section 3.2.2 explains in depth how the intrinsics are lowered and incremented correctly, and how the counters are allocated and stored in a buffer for this implementation. The counter buffer replaces the `profc` array that got inserted in the IR for storing the counters when looking at the study in section 2.1.4.

Continuing, looking at section 2.1.4, `profd` and `prf_nm` contain important metadata needed to map the counters back to instrumented code areas when writing the profile data file. As it will not be possible to insert these into the IR, this information has to be stored somewhere else. For this, we will use a descriptor [4]. Both the compiler and the driver can write and read information in these descriptors, which makes them a suitable place to store information that would otherwise be inserted into the IR.

`profd` have nine fields of data in the struct [32], but the only important ones for this implementation are (name of the corresponding struct field within parentheses) :

- The hash of the function's PGO name (`NameRef`).
- The function's CFG hash (`FuncHash`).
- The in-memory address difference to the counters, from the profile data struct, this function corresponds to (`CounterPtr`).
- The number of counters for the instrumented function (`NumCounters`).

The compiler pass collects information about `NameRef`, `FuncHash` and `NumCounters` simply by using LLVM's internal class `InstrProfInstBase` and its subclasses. All profile instrumentation intrinsics can be cast to a version of that class, which contain functions to retrieve the name, function hash and the number of counters for the function the intrinsic belongs to. The compiler pass writes this information to the descriptor so it is available for the driver when it writes the profile data file.

`prf_nm` contains a compressed version of all the function names in one shader, which is also collected and written to the descriptor by the compiler pass.

More information how this data is used when writing the profile data file follows in section 3.2.3.

3.2.2 Buffer allocation

As the GPU cannot allocate memory on its own, the driver must handle memory allocation. The driver needs to allocate memory for the counter buffer, which replaces the `profc` array that was seen in section 2.1.4. As both the GPU compiler and the driver need to know where this buffer is placed in memory, the pointer to the counter buffer is created as a uniform. As explained in section 2.1.1, uniforms are set before the shaders are executed and remain constant throughout the execution of all shaders. While the uniform itself is stored in constant memory, the memory it points to, i.e. the counter buffer, is modifiable. This setup ensures that the pointer remains unchanged during shader execution, while the buffer it points to can still be written to for updating the profile counters.

Algorithm 2 shows a simplification of what instructions are inserted into the IR by the compiler pass to update the counter buffer and replace the intrinsics. Index *i* corresponds to the index to which counter should be incremented in the shader. As explained in section

3.1.2, one shader runs in parallel distributed on many cores, and all threads update the same counter buffer, therefore, the counters need to be incremented atomically for the Arm Mali GPU.

Algorithm 2 Replace intrinsics

Input: index i

Compiler pass replaces the intrinsics with the following instructions into the IR:

```
counter_buffer = get_counter_buffer()
atomic_add(counter_buffer[i], 1)    // Increment by 1
```

The driver will allocate one large buffer that will fit all counters for all shaders. For the shaders to write to different places in this buffer, the pointer to the counter buffer needs to point to different spots for different shaders. This is decided based on the function hash written to the descriptor for each shader. For each new shader calling to set up the pointer, the pointer is moved one step in the allocated memory according to a fixed offset that is large enough to ensure that each shader has enough space in the buffer for its counters.

3.2.3 PGO file writing

To be able to reuse the collected profile data it must be written to a file that can later be given back to the compiler. As described in section 2.1.4, the runtime library `compiler-rt` handles the file writing. The runtime initialization hook calls `__llvm_profile_initialize` which is responsible for gathering data from the variables `__profc_func`, `__profd_func` and `__llvm_prf_nm`, where `func` corresponds to a function named `func`. This data is then written to the PGO (`default.profrac`) file. As discussed earlier, these variables are not available in the IR and the writing can therefore not happen from the program and the runtime initialization calls. It is the driver that will need to initialize and write the file, by calling the functions in the `compiler-rt` library that would otherwise be inserted in the IR and called from the program.

To begin, the function `__llvm_profile_initialize_file` [38] is called to initialize the file. The shader program will then run and fill up the counter buffer with the incremented profile counts. When the program terminates, writing of the `default.profrac` file is initiated. Section 2.1.4 explained the structure of the file and its components. Since the driver will still use the functions from the `compiler-rt` library to write the `default.profrac` file, it is essential to define the data resources (counter array and function metadata) that would otherwise have been generated in the IR and register them using the same functions as in listing 2.5.

The in-memory representation of the profile metadata struct is `__llvm_profile_data` [32] and the only fields that needs to be filled in order for `__llvm_profile_write_file` to have all the information it needs are:

- NameRef
- FuncHash
- NumCounters

- CounterPtr

`NameRef`, `FuncHash` and `NumCounters` have been written to the descriptor as explained in section 3.2.1. `CounterPtr` corresponds to the delta between where the counters for this shader start and the in-memory address of `__llvm_profile_data`. Algorithm 3 shows how this is done for each shader. The `profile_buffer_ptr` represents the pointer to the counter buffer and, as explained in section 3.2.2, points to different areas in the counter buffer for each shader. `name_str` in algorithm 3 corresponds to `llvm_prf_nm`. Calling `__llvm_profile_write_file` for each shader will not create multiple files, instead, it will append the new profile information to the existing file.

Algorithm 3 Allocate and initialize `__llvm_profile_data` for all shaders and write to file

Input: Shaders, `profile_buffer_ptr`

```
*pd = (__llvm_profile_data *)malloc(sizeof(__llvm_profile_data));
for all Shader s in Shaders do
    pd->NameRef = s.name_ref;
    pd->FuncHash = s.func_hash;
    pd->NumCounters = s.num_counters;
    pd->CounterPtr = profile_buffer_ptr - pd;

    __llvm_profile_register_function(pd);
    __llvm_profile_register_names_function(s.name_str);
    __llvm_profile_write_file();
end for
```

`__llvm_profile_register_function` only saves the memory address of the pointer, not the actual values. For this reason, `__llvm_profile_write_file` is called once for each shader while the `__llvm_profile_data` struct still contains the values for the latest registered shader (function).

The more traditional way of doing this is the way Clang does it, in section 2.1.4, by registering each function separately by multiple succeeding calls to `__llvm_profile_register_function` and then finish with one call to `__llvm_profile_write_file`. For this thesis implementation, that seemed more complicated and would have required more code and memory allocations of the `__llvm_profile_data` struct, one for each shader.

3.2.4 PGO file utilization

As the instrumentation is inserted with help from `PGOInstrumentationGen`, the simplest way to read back the profile data is with `PGOInstrumentationUse`. The profile data file is given to the compiler with an environment variable and passed as an argument to `PGOInstrumentationUse`. As the instrumentation is inserted into the IR, it is important that the pass interpreting the profiling data file sees the same IR as the pass generating the instrumentation. This means the two passes `PGOInstrumentationGen` and `PGOInstrumentationUse` must be added to the same place in the compiler pass pipeline.

As in the study in section 2.1.4, `llvm-profdata` [37] has to be used to get the right format of the file that `PGOInstrumentationUse` expect, i.e.:

```
llvm-profdata merge -output=default.profdata default.profrac
```


Chapter 4

Evaluation method

We have evaluated the performance impact of exposing profiling information to the compiler by looking at hardware counters. This was done by running GPU traces on an FPGA board simulating a Mali Gxx GPU.

4.1 GPU traces

A GPU trace is a recording of an application's interaction with a specific graphics API. Section 3.1.1 explained the rendering pipeline and how it starts with the CPU sending application requests to the GPU, that the GPU then executes. A GPU trace is recorded by capturing those application requests, and that recording can later be replayed on compatible devices.

A selection of recorded traces from Vulkan applications were provided by Arm. The applications `gfxrecon-replay` [10] and `vktrace` [25] were used for replaying the traces. The possibility to replace the generated shader code in the traces and collect hardware counters were also provided.

4.2 Measurements

Since Arm does not manufacture its own hardware implementations of its GPUs, performance evaluation was conducted on an FPGA board simulating a Mali Gxx GPU. This provides insights into the dynamic performance of the GPU.

To evaluate the performance, three types of runs were made for each trace:

1. Baseline run, without PGO enabled.
2. Runs to collect PGO data.
3. Optimizing run, using the collected PGO data.

The baseline run represent running a trace without the impact of PGO. Optimizing run is where the collected PGO data is given back to the compiler, and optimizations can be made with the knowledge of profiles. Performance is evaluated by comparing hardware counters from these two runs, as explained in section 4.2.2.

Additionally, we analyzed static counters to measure spills and how they correlate with dynamic performance.

4.2.1 Collect PGO data

Due to the shaders being heavily executed during each trace, collecting profile data for the complete trace is both time consuming and will often give too large counters that will not fit in the 64-bit integer that is allocated for each counter. To solve this, we limited gathering profile data to a subset of frames. If more than one frame is used for data gathering, the `default.profrac` file from each frame run is merged together, as shown below. Merging the files will leave one representative `default.profddata`, containing the sum of the profile counters, to use for the optimizing run.

```
llvm-profddata merge -output=default.profddata default*.profrac
```

4.2.2 Hardware counters

Performance is measured by looking at hardware counters that are collected when running a trace on the FPGA. The specific counters that are interesting for this thesis are:

- **GPU_ACTIVE**, the number of cycles the GPU are actively executing instructions.
- The number of instructions executed by the processing pipeline [19], specifically:
 - **FMA (Fused Multiply-Add)**: instructions that perform combined multiplication and addition, widely used in floating-point computations.
 - **CVT (Conversion)**: instructions that perform simple operations, such as format conversion and integer addition.
 - **SFU (Special Function Unit)**: instructions that perform complex functions, such as exponentials and trigonometric operations.
 - **MSG (Message)**: instructions used for communication and synchronization between shader cores or with memory.

A lower value of these metrics can indicate better performance. GPU_ACTIVE is the most valuable metric for evaluating overall performance of the GPU. Reducing the MSG counter can lead to significant performance gains, as this communication can be a significant source of overhead. The MSG counter is, therefore, a key indicator of how much time and effort is spent on these essential but often slower tasks, which can impact the overall performance of the system, especially in high-core-count runs or on real hardware. Even a small percentage reduction in this counter is worth a lot.

4.2.3 Selected traces

Since PGO is most effective on complex programs, we have selected traces that offer more complexity and optimization potential than typical mobile phone traces. As simple mobile game shaders often lack loops or branches, this PGO implementation is evaluated on more complicated traces where its performance impact and capabilities can be properly tested.

4.3 Verification

To verify that the collected PGO data is correct, and thus also the implementation of PGO in the GPU, statements that will cause a crash are inserted into the executed shaders. If a crash does not appear and the profile count for that block is zero, the PGO data and implementation will be accepted as correct.

This is done by first running a trace with PGO instrumentation enabled and dumping all shaders that were run in that trace. By compiling the shaders with the PGO data file and the option `-mllvm=pgo-view-raw-counts=graph`, using the Mali Offline Compiler [18], one will get a picture with the control flow graph (CFG) and the corresponding block counts, as seen in figure 4.1.

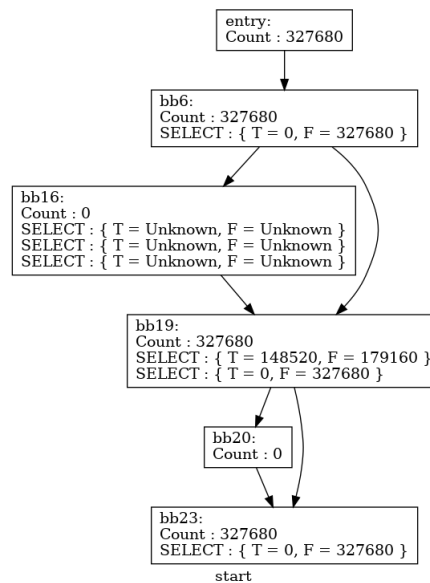


Figure 4.1: CFG for a shader, including profiling counts for the basic blocks.

This makes it simple to see interesting parts of the code to insert the crash statements in. For this case, it can be seen that two blocks, bb16 and bb20, have the block count zero. By inserting crash statements in the shader code, replacing that shader on the FPGA and then run the same trace again, we were able to verify that the collected PGO data is correct by not seeing any crash statement triggering from the two blocks with count zero.

Chapter 5

Results

This chapter presents the results comparing the baseline run with the optimized run that uses PGO data. The traces and frames are anonymized by numbers, and the numbers are consistent for all results. Each run was done once.

5.1 PGO in LLVM

This section presents the results from exposing profiling information to existing GPU compiler passes. The only modification that was done to the pass pipeline for this test was inserting the `PGOInstrumentationUse` pass early in the pipeline to ensure the majority of the passes had access to the profile data.

5.1.1 Static results

When compiling all shaders for all traces, with and without profile data available, we can get static metrics on how each shader is affected by the profile data. Table 5.1 presents the summed-up percentage change for the top three shaders with the highest execution counts in a trace when compared to the baseline for that trace. The frames correspond to the different frames that have been used to collect the profile data and merged correspond to the merged profile data files from all frames. Arm provided information about which shaders are the top three most executed, gathered from previous model runs. For trace 3, they did not provide this information, and static data for that trace is therefore not presented

The gathered data in table 5.1 include the percentage change in load and store spills and total instructions. How many of the top three shaders were affected is also presented, and it can be observed that not all top shaders show a difference in static measurements when profile data is available. Instructions can still have been scheduled differently in these shaders without affecting spilling and total instructions.

The total number of instructions is minimally affected, with a slight overall increase. This is not necessarily significant, as PGO doesn't primarily focus on reducing the code size.

The most interesting metric is spilling where we can observe a significant increase for the majority of frames. Trace 1 is the only trace seeing a decrease in spilling for all frames. Trace 2 and 5 show a significant change in spilling between their selected frames. Trace 4 and 5 have similar spilling for the merged metric, however trace 5 has two affected shaders compared to all three top shaders being affected for trace 4. This indicates that the two affected shaders for trace 5 contains more spilling than the three affected shaders for trace 4.

An increase in the static counters doesn't have to be a problem for performance, which will be seen in the coming section.

		Spilling	Total instructions	Affected shaders
trace 1	merged	-8.9%	-0.1%	2/3
	frame 1	-8.9%	-0.1%	2/3
	frame 2	-8.9%	-0.1%	2/3
	frame 3	-8.9%	-0.1%	2/3
trace 2	merged	Counter overflow.		
	frame 1	+40.8%	+0.9%	3/3
	frame 2	+64.2%	+1.2%	3/3
	frame 3	+200%	+2.6%	3/3
trace 3	merged	Nothing to merge.		
	frame 1	-	-	-
trace 4	merged	+166.2%	+1.3%	3/3
	frame 1	+166.2%	+1.7%	2/3
	frame 2	+80.9%	+0.8%	1/3
	frame 3	+116.3%	+0.6%	2/3
trace 5	merged	+164.1%	+2.4%	2/3
	frame 1	+249.1%	+3.3%	2/3
	frame 2	0.0%	+0.1%	1/3

Table 5.1: Percentage change in static counters for the top three most executed shaders in each frame.

5.1.2 Hardware results

Table 5.2 presents the difference in hardware counters for each trace, when comparing the baseline run with the optimized run for the whole trace. This is the most important result, as it best reflects real-world PGO performance in GPUs. The `default.profrac` files for trace 2 couldn't be merged due to counter overflow and we have therefore made three runs, one for each profile data file from the frames.

By observing the two most important counters GPU_ACTIVE and MSG, we can see an improvement for all traces, except for trace 2, where the MSG counter decreases most on average compared to the other metrics. GPU_ACTIVE and CVT also decreased more on average compared to FMA and SFU. Trace 3 has the most even improvement seen over all counters, with all counters being within one percentage unit. Trace 3 also has the largest improvement in four of the five counters, with trace 4 improving most in the MSG counter.

It is noticeable that there is a large difference in performance between the frames in trace 2, with frame 1 performing the best and frame 3 being worst of all runs across all traces. This variability in frame performance creates an uncertainty in selecting which frames to use for profile data collection. However, trace 3 has the overall best improvement, despite only having data from one frame. This contrasts with our initial assumption that merging profile data from multiple frames would yield the best results, as the merged data would be more representative of the entire trace.

		GPU_ACTIVE	FMA	CVT	SFU	MSG
trace 1		-0.47%	+0.07%	-2.17%	+0.018%	-0.74%
trace 2	frame 1	-1.27%	-0.093%	-1.74%	-0.028%	-1.88%
	frame 2	-0.7%	-0.024%	+0.92%	-0.021%	-1.2%
	frame 3	+4.23%	-0.94%	-0.76%	-0.17%	+3.62%
trace 3		-3.74%	-3.13%	-3.6%	-2.78%	-3.23%
trace 4		-1.59%	-0.028%	-0.93%	-0.026%	-6.32%
trace 5		-0.53%	-0.033%	-0.28%	-0.02%	-1.73%

Table 5.2: Percentage change in hardware counters for entire traces.
Lower is better.

Single frame optimization

Table 5.3 presents the difference in hardware counters for the selected frames for each trace when executed with their corresponding profile data collected from that frame. The expected result would be a greater improvement in the hardware counters compared to the result for the entire traces in table 5.2. The reason we expect better results here is because of the profile data being specifically collected for this frame, i.e., the optimizations should be optimal.

Traces 1, 2, and 3 show results that differ significantly compared to the results of the full traces. This difference suggests a possible issue during frame testing, as the results appear inconsistent. Traces 4 and 5 perform more as the expected results. It can be observed that trace 4 shows great improvements in the important counters GPU_ACTIVE and MSG, with the improvement in GPU_ACTIVE being slightly lower than for the full trace, and MSG showing greater improvements for frames 1 and 2, while frame 3 shows less improvement in that counter. For the two frames for trace 5, it is observed that PGO led to better performance for the single frames compared to the full trace. A significant difference between the two frames is also observed, with one frame showing a notably higher improvement than the other.

5.1.3 Spilling performance impact

This section contains scatter plots showing the relationship between spilling, from table 5.1, and the hardware counters GPU_ACTIVE and MSG, from table 5.2 and 5.3. Figure 5.1 plots the relationship between spilling and GPU_ACTIVE, while figure 5.2 plots the relationship between spilling and MSG.

		GPU_ACTIVE	FMA	CVT	SFU	MSG
trace 1	frame 1	-0.17%	-0.14%	-1.73%	-0.14%	-0.83%
	frame 2	-0.22%	+0.096%	+0.46%	+0.48%	+0.24%
	frame 3	-0.21%	+0.25%	-0.074%	+0.3%	-0.037%
trace 2	frame 1	-0.039%	-0.04%	-1.35%	+0.054%	-2.04
	frame 2	-2.11%	+0.015%	+0.5%	-0.043%	-0.77%
	frame 3	+0.67%	-1.07%	-0.9%	-0.3%	+0.54%
trace 3	frame 1	-0.15%	+0.16%	-0.66%	+0.12%	+0.046%
trace 4	frame 1	-1.15%	-0.041%	-0.82%	-0.042%	-7.03%
	frame 2	-1.28%	-0.006%	-1.27%	-0.02%	-7.98%
	frame 3	-1.09%	+0.008%	-0.88%	+0.013%	-3.37%
trace 5	frame 1	-1.42%	+0.047%	-0.65%	+0.068%	-5.83%
	frame 2	-0.72%	-0.038%	-0.32%	-0.06%	-2.44%

Table 5.3: Percentage change in hardware counters for single selected frames. Lower is better.

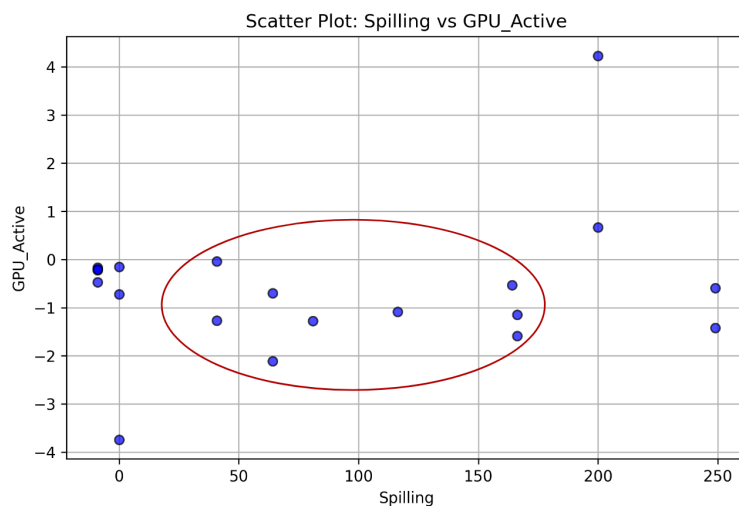


Figure 5.1: GPU_ACTIVE reduction plotted against spilling. The red circle marks the area where GPU_ACTIVE decrease most on average.

Observing figure 5.1, the most significant decrease in GPU_ACTIVE appears when spilling is unchanged (zero), while the greatest increase happen when spilling has increased 200%. The red circle marks where a decrease in GPU_ACTIVE seems most stable, improving around 1% on average.

Figure 5.2 indicates the same for the MSG counter. The greatest increase appears when spilling has increased 200%. The blue circle marks where a decrease is most certain, and include the point where MSG improves most, where spilling increased 80%.

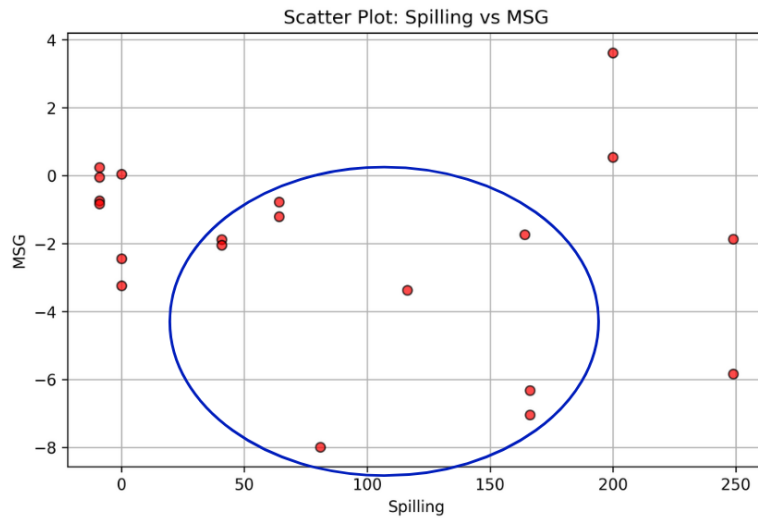


Figure 5.2: MSG reduction plotted against spilling. The blue circle marks the area where MSG decrease most on average.

5.1.4 Optimizations

From what we have observed in table 5.2, PGO provides obvious improvements in the two most important counters GPU_ACTIVE and MSG, however, we haven't been able to localize all passes that contribute to these improvements. The most important pass types that leverages PGO in LLVM is brought up in section 2.2.3, and provides insight into which passes have contributed to the performance improvements.

One thing that has been observed is that the simplification of the CFG is less aggressive when PGO is enabled. The LLVM pass **SimplifyCFG** performs folding and speculative execution of basic blocks when possible, provided that the amount of instructions in the removed basic block allows it. Listing 5.1 provides an example of how this optimization is performed using if-conversion and figure 5.3 illustrate an example from a shader where this optimization were disabled when profile data was available.

5.2 Collection of PGO data

From what we have observed when gathering the profile data, the slowdown is approximately 10%. For now, the work in this thesis will only be used offline, and a decrease in performance is therefore not an issue. However, if this work were to be productized, a more critical concern is the possibility for overflow in the profile counters.

```
1 Source code:
2 if (x < y) {
3     sub = 0;
4 } else {
5     sub = x - y;
6 }
7 ...
8
9 IR of sourcecode:
10 BB:
11     %cmp = icmp ult %x, %y
12     br i1 %cmp, label %EndBB, label %ThenBB
13 ThenBB:
14     %sub = sub %x, %y
15     br label BB2
16 EndBB:
17     %phi = phi [ %sub, %ThenBB ], [ 0, %EndBB ]
18     ...
19
20 IR after folding and if-conversion:
21 BB:
22     %cmp = icmp ult %x, %y
23     %sub = sub %x, %y
24     %cond = select i1 %cmp, 0, %sub
25     ...
```

Listing 5.1: Example of IR transformation using folding and if-conversion to eliminate branches by replacing them with select instructions.

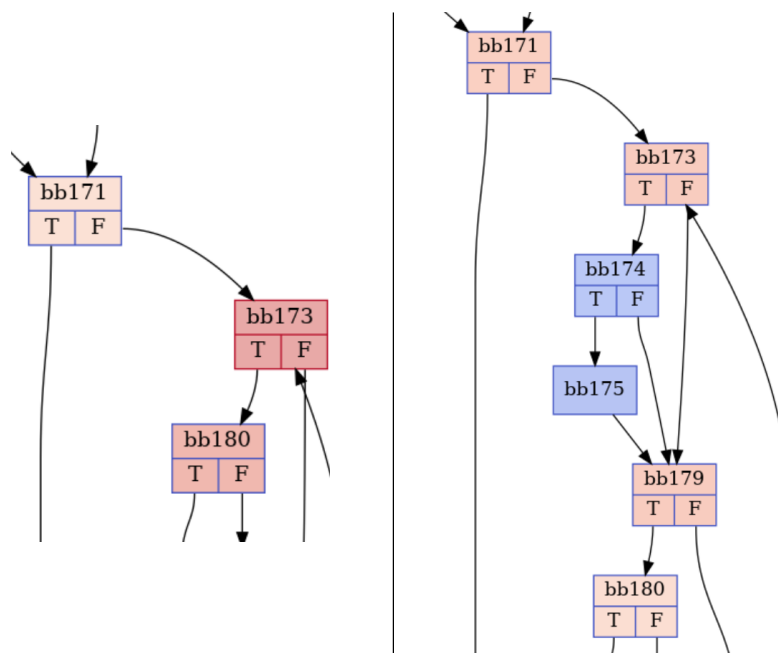


Figure 5.3: Basic blocks bb174, bb175 and bb179 have not been folded when PGO is available (right picture), compared without PGO available (left picture). Blue color of blocks indicate low profile count (only apply for right picture with PGO available).

Chapter 6

Discussion

This chapter discusses the result from evaluating the implementation of PGO.

6.1 Static results

Table 5.1 showed static results for all traces and their selected frames when compiled with profile data. We choose to only include the top three most run shaders for each frame. The reason for this decision is that shaders outside the top three often do not run more than around one percent and, therefore, have a smaller effect on the result. We generalized this for practical considerations.

First, by analyzing table 5.1, we observed that not all top shaders affect the static measurements. The reason for this is that the optimizations utilizing PGO do not focus on removing instructions. Instead, as explained in section 2.2.3, they primarily perform CFG simplifications, loop transformations, and instructions scheduling. This means that a shader can still be affected and optimized by the PGO data, even if it does not show static changes in terms of spilling and instruction count.

We observe that changes to the total instructions count are minimal, although increasing mostly. Since this is not the primary goal for PGO, to decrease the total instructions, we will not focus on this measurement when analyzing the results.

The most noticeable result is the great increase in spilling for most traces. While the increase in spilling might seem like a problem for performance, we can observe that the increase arises from the compiler moving the spills into blocks that are rarely executed when PGO is available. Figures 6.1 and 6.2 show the same loop in the same shader with and without PGO. Figure 6.2 that uses PGO contains more blocks due to PGO disabling if-conversion, explained in section 5.1.4. We can observe that the blocks inside the loop contain more spilling with PGO and that the spilling in the block after the loop is removed with PGO. The blocks inside the loop are executed 0.03% compared to the loop entry and exit block. This example, along with the improved results for most traces observed in table 5.2, suggests that the

increased spilling is not expected to be a concern in most cases.

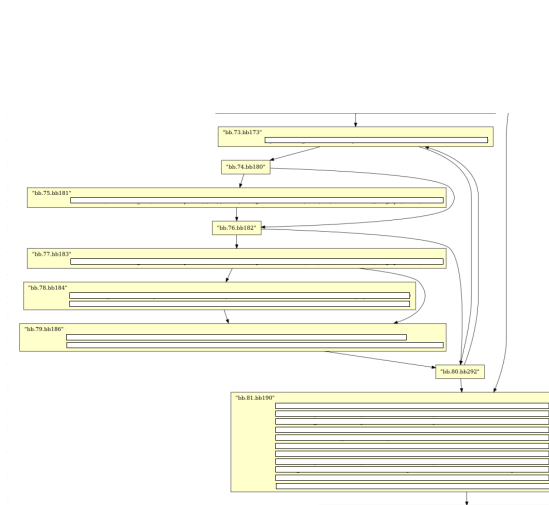


Figure 6.1: Spill distribution without PGO.

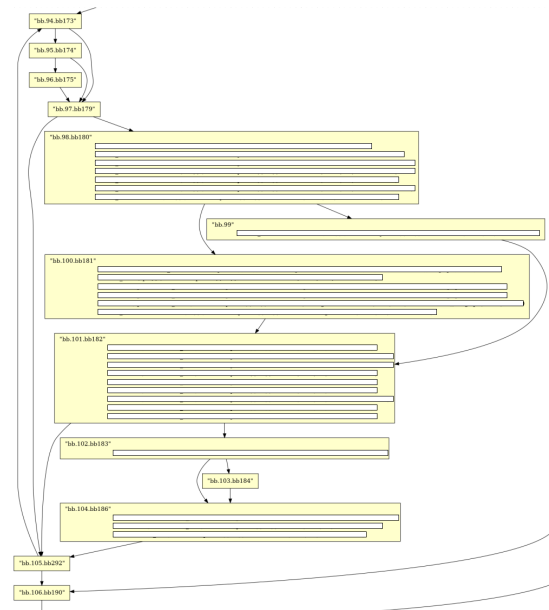


Figure 6.2: Spill distribution with PGO.

Figure 6.3: Snippet of the CFG for a shader program. White lines represent instructions that spill. It is noticeable that a large amount of the spilling has been moved to other blocks when PGO is available.

Spilling can be more costly on GPUs compared to CPUs. CPUs hide the latency caused by spilling by speculatively executing loads that have been spilled to memory. In contrast, GPUs avoid speculative execution of spilled memory due to warp synchronization, if one thread spills, all threads in the warp spill. As a result, moving spilling to rarely executed blocks can be an effective optimization for GPUs.

6.2 Hardware results

We observed in table 5.2 that the majority of traces improved in performance for the two most important counters GPU_ACTIVE and MSG. The other counters, FMA, CVT, and SFU, also improved in most runs, however, not as much, and their improvements will not affect performance in a significant way compared to GPU_ACTIVE and MSG.

The most notable results were observed for traces 2 and 3. Trace 2 because merging the profile data files resulted in a counter overflow, meaning we had to perform three optimizing runs, one for each frame. Noticeable was that the results between the three frames were significantly different. Frame 1 and 2 improved in performance, while frame 3 stood out by performing worst of all runs across all traces. This creates uncertainty in how the profile data should be collected and suggests that merged data provides a better representation of

the program, as all traces with merged data show improved performance for the important counters.

Observing the result from trace 3, it stands out, considering it performed overall best, showing the greatest improvement in four out of five metrics despite only having profile data from one selected frame. Since the top shaders for this trace are unknown, we have not provided any static results for this trace. However, one thing that could be observed when analyzing the static data is that many shaders for this trace show a decrease in used registers by 50%. We do not know how often these shaders are executed, so we cannot determine if they cause the performance improvement. A 50% decrease in register use is generally good because it doubles the number of threads that can run simultaneously.

Overall, we find the result promising, especially considering the shaders for these traces have already undergone extensive optimizations.

6.2.1 Single frame optimization

We observed in table 5.3 that traces 1, 2, and 3 gave unexpected and inconsistent results for the single frame runs compared to the achieved results for the entire traces. We find no reasonable explanation for this behavior other than an issue during testing, and more testing would need to be conducted to confirm this. Trace 3, frame 1, confirms this as profile data from this frame alone was used for optimizing the entire trace, and that trace showed the greatest performance improvement of all traces. Therefore, the observed performance for optimizing that frame alone is unlikely.

Observing trace 4, it achieved slightly less improvement in GPU_ACTIVE than the entire trace, however, it performed greater improvements for MSG in frames 1 and 2. Frame 3 still showed a significant decrease in the MSG counter, just not as much as the other frames or the entire trace. Considering all frames for trace 4, it is reasonable that merging these frames led to significant improvements for the full trace.

Trace 5 showed better performance for both of its frames compared to the full trace. While this aligns with what we hypothesized would be the result before optimizing the individual frames, it also suggests that the merged profile data from these two frames may not have been representative enough to give a significant improvement for the full trace, considering the performance difference between the runs.

Another explanation for most frames not achieving results as good as for the entire traces is that they might have already reached their peak in possible performance improvements, meaning that PGO is limited on these frames.

6.3 Spilling performance impact

Observing the plots in figure 5.1 and 5.2, we can see that the majority of the points are under 0 for the hardware counters, indicating a performance improvement. An increase is observed when the spilling increase is either close to 0 or 200%. The metrics between these spilling values are always decreasing. This proves that some increase in spilling can provide an overall performance improvement, and observing figures 6.1 and 6.2 of how spilling is moved to rarely executed blocks contributes to this conclusion.

This observation helps guide the selection of profile data. A significant increase in spilling is likely caused by the compiler receiving too aggressive counters, especially when collected for frames that may not be as representative as expected.

6.4 Optimizations

Section 5.1.4 showed a real example of how the LLVM pass **SimplifyCFG** uses PGO in one of the existing shaders. It could be observed that the CFG captured when using PGO contained more blocks than the CFG without PGO available. This occurs because the compiler performs folding to eliminate branches. In a GPU, folding blocks is desirable if there is a risk of branch divergence, as it helps maintain warp coherence. However, if one of the branches is rarely or never taken, it is more beneficial to keep the blocks separated to avoid unnecessary instructions and improve performance.

By analyzing the static results, which indicate increased spilling, together with the CFG snippets in the figures 6.1, 6.2 and 5.3, we can conclude that the compiler is leveraging PGO effectively, in the areas we have studied if given representative profile data.

6.5 Collection of PGO data

As stated in section 5.2, a slowdown in execution time and decreased performance are not an issue since this implementation will only be used offline for now. We could experience a counter overflow for the profile data files for trace 2 when trying to merge them. This is more of a concern as it is a bottleneck when gathering representative profile data. In theory, profile data from multiple frames should give a more representative overview of how the program is executed and enhances performance. One possible solution to this problem is to count every tenth counter instead of every individual counter. This approach would be valid since block frequency is always compared to the surrounding blocks, using the probability instead of the exact frequency when determining which optimizations to apply.

Chapter 7

Conclusion

In this master's thesis we have addressed the following research questions.

1. *How can the existing PGO infrastructure in LLVM be adapted to support GPU architectures?*

We have adapted the current PGO implementation in LLVM/Clang, explained in section 2.1.4, to work for the Arm Mali GPU by addressing the architectural differences between CPUs and GPUs. The most significant difference was that the GPU cannot allocate its own memory, preventing us from storing the counters and function data in the IR. Consequently, the function to write the `default.profraw` file couldn't be called from within the IR.

To address these challenges, we made the following adaptations:

- Developed a compiler pass (as a replacement for the `InstrProfiling` pass) to lower the intrinsics and collect important data information about each function, without inserting anything into the IR.
- Modified the GPU driver to allocate memory for the counter buffer.
- Modified the GPU driver to register the correct counter and function data for each function and write the `default.profraw` file.

2. *What is the impact of exposing profiling information, such as block frequency, to existing GPU compiler passes, on the performance of the generated code?*

When evaluating the impact of PGO on entire traces we achieved performance improvements for all runs but one, with up to 3.74% fewer clock cycles and 6.32% fewer message instructions between threads. The static results show that moving spilling to rarely executed blocks and avoiding folding and speculative execution when a branch is unlikely to be taken also contribute to performance improvements from PGO.

3. *What types of novel GPU-specific optimizations can be developed by leveraging the gathered profiling data?*

Due to time constraints, no novel GPU-specific optimizations were implemented. However, some ideas of what type of implementations that could be made are presented as future work, in section 7.1.3.

To summarize, our work demonstrates that enabling profile-guided optimizations for GPUs is feasible and can lead to measurable performance improvements, as observed in the results from chapter 5. Compared to previous implementations of PGO explained in section 2.2, the performance gain we achieved is relatively small. However, those implementations were either not designed for GPUs or were specifically adapted for them. In contrast, our work only shows the result of providing profile data to LLVM's existing optimization passes without making any modifications to them. Since the existing optimizations in LLVM are not tailored for GPUs, this suggests that even greater improvements could be achieved by developing GPU-specific optimizations that leverage profile data. After all, even a small percent of improvement is significant for complex parallel GPUs, and our results show great potential for using PGO in graphics. As more LLVM passes utilize PGO, further improvements in GPU performance is possible, building upon this work.

7.1 Future work

This section presents potential future work to maximize the benefits of PGO and its capabilities.

7.1.1 Investigate performance improvement

As can be seen in section 5.1.2, PGO improves performance for the majority of traces when exposed to the compiler and existing passes. Section 5.1.4 illustrates one optimization that leverages PGO. To investigate how the optimizations mentioned in section 2.2.3 contribute to the observed performance was not in the scope of this thesis. By studying in detail how PGO has affected the performance (both positively but perhaps also negatively) for the GPU, even greater performance may be achieved.

7.1.2 Value profiling

This thesis only implements block frequency profiling, however value profiling also exists in LLVM. Value profiling tracks the values that variables and expressions take during runtime, along with the frequency of those values. This allows the compiler to perform optimizations based on observed values and patterns. For the GPU, this could be helpful for tracking thread patterns and divergence within a warp.

The pass `PGOInstrumentationGen` already insert intrinsics for value profiling. In this thesis implementation, those intrinsics are removed instead of lowered, and the reason to that is memory allocation. Further work to implement value profiling includes determining how to store the values in memory. To simplify the buffer allocation for the counters, we used a static value that we knew would be enough to store all counters. With value profiling,

this becomes more difficult as the values are dynamic, and we won't know in advance how many values, or their types, that need to be stored.

7.1.3 Optimizations

To take full advantage of PGO, some new optimizations specific for the GPU should be developed.

As explained in section 2.2.2, branch divergence has a significant impact on the performance of GPU programs. Like the authors in [35] suggest, a key idea to discover branch divergence is to instrument both branches and identify patterns that indicate divergence.

A potential optimization is the one presented in [35], where loops are reconstructed to follow the, hopefully detected, patterns observed during profiling. The reconstruction may lead to warps executing loops in a convergent way, reducing divergence and improve execution efficiency.

In many cases, a block frequency profiler is enough for optimizing branch divergence. By knowing the probability of each branch, the compiler can reorder them to make the most common branch execute first. While this does not reduce the total number of cycles required to execute both branches, it will ensure that the majority of work scheduled for a warp is executed earlier. This will help when hiding latency by allowing other warps to run while waiting for divergent execution paths to finish, improving the throughput of the GPU and reducing the overall impact branch divergence has on performance.

Additionally, block frequency profiling enables the compiler to optimize for the most frequently executed path. This includes guided speculative execution and instruction scheduling, leading to better resource allocation, pipeline utilization and fewer execution stalls.

However, with a value profiler, more advanced optimizations can be applied. In [21], a thread-data remapping scheme is presented to dynamically re-group threads so that each warp follows the same execution path, and therefore further minimizing branch divergence. Here, runtime profiling data is invaluable, compared to handling branch divergence statically.

Another optimization that proves the benefit of having a value profiler is *Zeroploit* [33], explained in section 2.2.2. Instead of optimizing for a block, the compiler can optimize for a common value, leading to even better performance.

7.2 Threats to validity

This thesis has explored enabling PGO for graphics and its impact on performance. While the results in section 4.2.2 show improved performance, there are some factors that threaten the validity of these results.

The first issue is that this thesis implements PGO for mobile phones but does not test on typical traces for mobile phones, we cannot guarantee that the performance improvements are comparable. Performance may also vary depending on the user and the program.

The second issue is that the profile data is not fully verified. While we have done a simple verification of one shader, as seen in section 4.3, this only confirms that the data was correct for the blocks that were not executed. What it doesn't confirm is whether the counters in the blocks that execute are correct. However, with the improved result and the `default.profrac` not appearing abnormal, we assume that the counters are correct.

The third issue is that the traces are only run once for each test. Naturally, the hardware counters can have some noise, meaning that our performance gains might not be as they appear, or they could have been even better. To get more correct results, the traces should have been run multiple times and the result should have been the average of those runs. According to this thesis [26] done on the same hardware, repeated measurements showed that the differences in hardware counters between each run were always lower than 0.03%. Since the majority of our performance gains are more than this, it is likely not just noise.

Lastly, we didn't run any tests to validate that the results are correct in terms of the expected frame output. However, since the optimizations that contributed to the performance improvements were already part of the pass pipeline from the beginning, i.e. they were not introduced specifically for this thesis, we assume that they are now making more informed optimizations, without corrupting the code.

References

- [1] Tomas Akenine-Möller, Eric Haines, and Naty Homan. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 2018.
- [2] Arm. Arm GPU Training - Episode 1.2: The rendering pipeline, 2024. <https://developer.arm.com/Additional%20Resources/Video%20Tutorials/Arm%20Mali%20GPU%20Training%20-%20EP1-2>. Accessed: 2025-01-08.
- [3] Arm. Control Processing Unit (CPU), 2025. <https://www.arm.com/glossary/cpu>. Accessed: 2025-01-08.
- [4] Victor Blanco. Descriptor Sets, 2020. <https://vkguide.dev/docs/chapter-4/descriptors/>. Accessed: 2025-01-20.
- [5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.
- [6] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO '16*, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery.
- [7] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers*, 62(2):376–389, 2011.
- [8] Leo Di Donato. Demystifying the profraw format, 2022. <https://leodido.dev/demystifying-profraw/>. Accessed: 2025-01-13.
- [9] Sean Ellis. Memory Management on Embedded Graphics Processors, 2013. <https://community.arm.com/>

- arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/memory-management-on-embedded-graphics-processors. Accessed: 2025-01-10.
- [10] Brad Grantham. GFXReconstruct – Tools to Capture and Replay Graphics API Calls. 2023. https://vulkan.org/user/pages/09.events/vulkanised-2023/vulkanised_2023_gfxrreconstruct_tools_to_capture_and_replay_graphics_api_calls.pdf.
- [11] Wenlei He, Hongtao Yu, Lei Wang, and Taewook Oh. Revamping sampling-based pgo with context-sensitivity and pseudo-instrumentation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 322–333, 2024.
- [12] Min-Yih Hsu. *LLVM Techniques, Tips, and Best Practices*. Packt Publishing Ltd., 2021.
- [13] Minwook Je. [CUDA] Warps and SIMT, 2024. <https://medium.com/@minkj1992/cuda-warps-and-simt-9fb83d164a24>. Accessed: 2025-03-28.
- [14] Donald E. Knuth and Francis R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13(3):313–322, September 1973.
- [15] Arm Limited. Fourth-generation Mali GPU architecture, 2025. <https://developer.arm.com/documentation/102203/0100/Fourth-generation-Mali-GPU-architecture>. Accessed: 2025-01-10.
- [16] ARM Limited. How Render Passes Work, 2025. <https://developer.arm.com/documentation/102479/0100/How-Render-Passes-Work>. Accessed: 2025-03-31.
- [17] Arm Limited. Learn the architecture - A64 Instruction Set Architecture Guide, 2025. <https://developer.arm.com/documentation/102374/latest/>. Accessed: 2025-01-08.
- [18] Arm Limited. Mali offline compiler, 2025. <https://developer.arm.com/Tools%20and%20Software/Mali%20offline%20Compiler>. Accessed: 2025-02-12.
- [19] Arm Limited. Shader core functional units, 2025. <https://developer.arm.com/documentation/102626/0108/Shader-core-functional-units?lang=en>. Accessed: 2025-02-28.
- [20] Arm Limited. The Mali GPU Memory Model, 2025. <https://developer.arm.com/documentation/dui0538/f/opengl-concepts/the-mali-gpu-memory-model>. Accessed: 2025-01-10.
- [21] Huanxin Lin, Cho-Li Wang, and Hongyuan Liu. On-GPU Thread-Data Remapping for Branch Divergence Reduction. 15(3), October 2018.
- [22] DigitalOcean LLC. The Role of Warps in Parallel Processing, 2025. <https://www.digitalocean.com/community/tutorials/the-role-of-warps-in-parallel-processing>. Accessed: 2025-02-12.

-
- [23] LLVM Project. LLVM's Analysis and Transform Passes, 2025. <https://llvm.org/docs/Passes.html#llvm-s-analysis-and-transform-passes>. Accessed: 2025-02-20.
 - [24] Jordan Logan. Memory management in Vulkan. Vulkan Developer Day - Montreal, 2018.
 - [25] LunarG, Inc. vktrace, 2025. <https://github.com/LunarG/vktrace>. Accessed: 2025-02-27.
 - [26] Neij, Johannes and Stenkrona, Arne. Experimental Evaluation of Compiler Optimizations on Arm Mali GPUs, 2021. Student Paper.
 - [27] Diego Novillo. SamplePGO - The Power of Profile Guided Optimizations without the Usability Burden. In *2014 LLVM Compiler Infrastructure in HPC*, pages 22–28, 2014.
 - [28] LLVM Project. "compiler-rt" runtime libraries, 2025. <https://compiler-rt.llvm.org/>. Accessed: 2025-01-08.
 - [29] LLVM Project. Header, 2025. <https://llvm.org/docs/InstrProfileFormat.html#header>. Accessed: 2025-01-13.
 - [30] LLVM Project. LLVM Documentation, 2025. <https://llvm.org/docs/>. Accessed: 2025-01-08.
 - [31] LLVM Project. Profile counters, 2025. <https://llvm.org/docs/InstrProfileFormat.html#profile-counters>. Accessed: 2025-01-13.
 - [32] LLVM Project. Profile metadata, 2025. <https://llvm.org/docs/InstrProfileFormat.html#profile-metadata>. Accessed: 2025-01-08.
 - [33] Ram Rangan, Mark W. Stephenson, Aditya Ukarande, Shyam Murthy, Virat Agarwal, and Marc Blackstein. Zeroploit: Exploiting Zero Valued Operands in Interactive Gaming Applications. 17(3), August 2020.
 - [34] Suyog Sarda and Mayur Pandey. Advanced IR Block Transformations. In *LLVM Essentials*, chapter 5. Packt Publishing Ltd., Birmingham, UK, 2015.
 - [35] Santonu Sarkar and Sayantan Mitra. A Profile Guided Approach to Optimize Branch Divergence While Transforming Applications for GPUs. pages 176–185, February 2015.
 - [36] Ian Smalley and Mesh Flinders. What is a graphics processing unit (GPU)?, 2024. <https://www.ibm.com/think/topics/gpu>. Accessed: 2025-01-08.
 - [37] The Clang Team. Profile-guided optimization, 2025. <https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization>. Accessed: 2025-01-08.
 - [38] The Clang Team. Using the Profiling Runtime Without Static Initializers, 2025. <https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#using-the-profiling-runtime-without-static-initializers>. Accessed: 2025-01-13.
-

- [39] Inc The Khronos Group. OpenGL Shading Language, 2021. https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language. Accessed: 2025-01-08.

EXAMENSARBETE Enabling Profile Guided Optimizations (PGO) for Graphics**STUDENT** Emma Jansson**HANDLEDARE** Jonas Skeppstedt (LTH), Tobias Gutzmann (Arm)**EXAMINATOR** Christoph Reichenbach (LTH)

Snabbare grafikkort med profildata

POPULÄRVETENSKAPLIG SAMMANFATTNING Emma Jansson

För att kunna göra så exakta kod-optimeringar som möjligt hjälper det att veta hur ofta olika delar av ett program körs. Genom att samla in profildata från en körning kan det användas vid nästa körning för att skapa ett ännu snabbare program. Detta examensarbete möjliggör denna process på grafikkort för mobiltelefoner.

För att kunna köra kod på ett grafikkort behöver koden kompileras till maskinkod. Det innebär att koden skrivs om från läsbar kod som en användare har skrivit till maskinkod som grafikkortet förstår. Under kompileringen utförs flera optimeringar som gör koden snabbare, exempelvis att minska antalet instruktioner som körs eller flytta om kod.

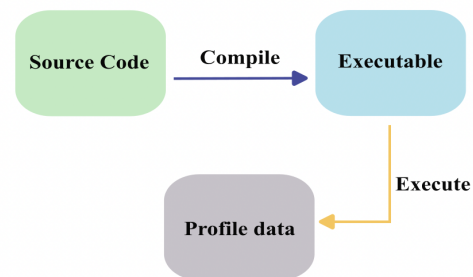
Genom att veta vilka delar av ett program som körs mest, eller inte körs alls, kan man göra mer informerade beslut om vilka optimeringar som kommer ge bäst resultat. För att veta detta behöver profildata samlas in från en körning så att den kan användas vid nästa kompilering.

Att samla in och använda profildata är redan väldigt vanligt för vanliga processorer. Att göra det för ett grafikkort är betydligt svårare. I det här exjobbet har vi tagit fram en lösning för att göra detta på ett Arm Mali grafikkort.

Vi använde det befintliga kompilatorramverket LLVM, vilket hjälpte till med de mest utmanande delarna. LLVM satte ut räknare i olika delar av koden som ökade varje gång den delen kördes, sparade resultaten från räknarna i en fil, och använde dem vid nästa kompilering för att göra programmet snabbare.

Det som ställer till det är att grafikkort inte kan hantera sitt eget minne. Det betyder att det

First compilation



Second compilation



inte själv kan skapa plats för där räknarna ska sparas under programmets gång. I stället måste grafikdrivrutinen, som fungerar som grafikkortets hjälpre, göra detta i förväg.

Genom att köra samma program med och utan profildata kan vi se en förbättring när profildata används. Detta sker enbart genom att låta LLVM använda profildatan i sina befintliga optimeringar. För att uppnå ännu bättre resultat drar vi slutsatsen att optimeringarna behöver anpassas mer specifikt för grafikkortets behov.