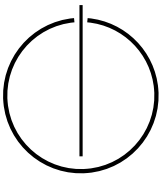
Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Profiling GPU Shaders for Profile-Guided Optimizations

Sebastian Neubauer



Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Profiling von GPU Shadern für Profilgeführte Optimierungen

Profiling GPU Shaders for Profile-Guided Optimizations

Author:	Sebastian Neubauer
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Dr. Nicolai Hähnle
	Dr. Matthäus Chajdas
Submission Date:	15. Oktober 2019

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Ort, Datum

Sebastian Neubauer

Acknowledgments

I would like to thank Nicolai Hähnle and Matthäus Chajdas for their support and excellent explanations about GPUs. Thanks to professor Gerndt for supervising this topic. Also, a big thank you to Andreas Leitner, who often helped me with problems concerning graphics and games.

Abstract

Today, GPUs are getting used in various fields for an increasing number of diverse purposes. One of the parts which enable this development are compilers, but so far, the compilers for GPUs have a shortcoming when it comes to optimizations. At many points, a compiler uses heuristics to guess how code will behave at runtime. If real runtime information is provided to the compiler, it can optimize better and more aggressively as the chance of a misjudgment is lower.
On CPUs, collecting runtime data and providing it to the compiler is called profile-guided optimizations (PGO). These optimizations are common on CPUs, but they do not yet exist for GPU programs. While many tools that assist with profiling on GPUs exist, very few of them provide insight into what happens inside a shader.
This work introduces profile-guided optimizations on GPUs with a focus on collecting profiling information.
As a first part, the basic block counters from LLVM are adjusted to support GPUs. They indicate if basic blocks are executed at all. Up to 30 % of blocks are unused in tested games and can be removed by the compiler for testing purposes. Analyzing the number of allocated registers shows that removing unused code does not have a great effect on register usage.
We test five games with LLVM's PGO with basic block counters, leading to performance changes of at most 1 %. Some games get faster, some get slower. A sample shader that implements a switch-case based virtual machine gets more than 20 % faster by using PGO and removing unused basic blocks. The overhead of counter instrumentation ranges from 0 % to 43 %, averaging at 16 %.
Apart from execution counters, uniformity of branches and variables is an interesting property on graphics cards. We implement LLVM passes to log the uniformity of branches and variables. This work shows the collected data about uniform variables and register usage in multiple diagrams, visualizing the runtime behavior of shaders.

Contents

1 Introduction

This section motivates the optimization of code running on the graphics processing unit (GPU) and explains why runtime analysis is needed for some optimizations. We follow that the knowledge of runtime information is beneficial for the performance of GPU programs. In the end, we list our contributions.

1.1 Motivation

History of GPUs GPUs started as simple co-processors that were able to draw basic primitives such as lines and triangles onto a frame buffer which was displayed on a screen. Vertex coordinates had to be computed on the central processing unit (CPU), which fed the so-called *fixed function pipeline* on the GPU. The GPU rasterized the primitives and determined the colors of the resulting pixels by interpolating vertex colors or by simple texturing. Over time, the capabilities of GPUs increased. The fixed function pipeline was first made increasingly configurable by adding new functions such as multi-texturing and eventually, it was made programmable by allowing pixel shaders. A pixel shader is a program which determines the color of each rasterized pixel and it is executed by general-purpose processing units on the GPU. At the same time, vertex processing was moved to the GPU and made programmable using the same processing units. The overall computational power of GPUs increased substantially by providing increasing memory bandwidth and by instantiating numerous processing units operating in parallel. This computational power combined with their ubiquity made GPUs attractive for applications outside of real-time rendering. [27]

While this General Purpose GPU (GPGPU) usage was not the primary purpose of shaders, multiple APIs and libraries emerged to provide good support using GPUs for different purposes. GPUs soon gained the ability to run compute shaders (also called compute kernels) outside the graphics pipeline, but using the same general-purpose processing units. Nowadays, the computing power of GPUs stems from their large amount of general-purpose arithmetical processing units and the parallel computing power is used for games, high performance computing and even tasks like offloading computations from spreadsheets or databases [22, 28].

Optimization In general, code should run as fast as possible. This saves time, energy and money or allows us to handle larger problems. To achieve this goal, code runs through multiple stages of compilation and optimization before it is executed. The time when

optimizations happen differs depending on the used programming languages and tools. E.g. C++ code is optimized ahead of runtime when compiling code. On the other hand, scripting languages like JavaScript or bytecode languages like Java and C# are optimized at runtime by a just-in-time compiler (JIT). On a GPU, code is compiled ahead of runtime, similar to C++. The compiled code is then copied over the Peripheral Component Interconnect Express bus (PCIe bus) to the GPU and executed.

For ahead-of-time compiled programming languages, optimizations are generally based on static analyses of code. For example, constant propagation can be implemented by first analyzing the possible values of variables using a fixed point iteration over the control-flow graph (CFG) and afterward transforming the CFG such that known variable values are used [34].

Another possible optimization is the decision where to place code in a resulting executable. It is beneficial for performance if seldom executed instructions are put outside of the usual control-flow. On an average execution, these instructions are not executed. If they are in between other code, the processor needs to jump over them at some point. If they are somewhere else instead, e.g. after the return statement, this one jump can be saved in the average case and the code runs faster. The general process of deciding the order of code in the executable is called *linearization* [34]. The compiler has to decide a linear order of the code, which is previously represented as a control-flow graph.

There is a difficulty with this optimization though: The compiler often does not know which branch will be taken (hot) and which code is reached seldom (cold). Therefore it has several means to figure out which branches are hot and which cold. Most often, the compiler uses heuristics. For example, error handling code is probably taken less often than a path without errors. As another variant, the code author themselves can tell the compiler which branch is taken more often. An example is the Linux kernel, which often makes use of `likely` and `unlikely` markers in `if`-conditions.

There are cases though, where the outcome of a condition depends on the concrete input into the application or the compilers heuristics take the wrong guess. In theory, a developer could work around this by marking every one of those conditions with `likely`. In general, this is undesirable as the compiler should automate as much as possible and developers can be mistaken when guessing the likelihood of branches. For JIT compilers, which are working at runtime, this optimization is not a problem because the interpreter knows which branches are hot and which cold. The knowledge of concrete input data enables them to output more efficient code than ahead-of-time compilers.

Profile-Guided Optimization There is a remedy for compiled languages though: Developers can provide sample input data to the compiler. Then an ahead-of-time compiler has access to the same information as a JIT compiler and it can make use of information e.g. about hot branches to generate more efficient code. Popular C and C++ compilers like clang and msvc implement this in the form of profile-guided optimizations (PGO) [12, 30]. This optimization can yield performance improvements of more than 10 % in some cases [20].

Listing 1 shows an exemplary usage of PGO. The procedure consists of three different steps:

1. At first, the program is compiled with static optimizations only and instrumentation instructions are inserted.

2. As a second step, the compiled program is run on sample input data. In this step, the instrumentation code collects useful information about the execution and usually

```
# Compile with instrumentation
clang++ -O2 -fprofile-instr-generate program.cpp -o program
# Run program
./program input.txt
# Transform raw profile data into the right format
llvm-profdata merge -output=program.profdata default.profraw
# Compile again with pgo
clang++ -O2 -fprofile-instr-use=program.profdata program.cpp -o program-pgo
```

Listing 1: *Compiling a C++ program with profile-guided optimizations using clang.*

creates a file containing this data. In the case of clang, this data has to be post-processed with the `llvm-profdata` executable.

3. In the third and last step, the program is compiled again. This time, the compiler has access to collected runtime/profiling information so it can produce a better optimized version of our application.

For clang there is also a second option available. Instead of using instrumentation to capture runtime information, it can also use data collected by standard profiling tools like the Linux `perf` program [23]. Profilers like `perf` generally incur a lower runtime overhead compared to instrumentation. The downside is that the resulting information is less detailed [12].

The usage example in listing 1 shows that using PGO today is well-supported by popular tools and easy to use for developers. In the first part of this introduction, we looked at the evolution of GPUs and how they became highly parallel, general-purpose processors. Profile-guided optimizations however are not yet supported for GPU code. The problem is the lack of tools for profiling or instrumentation of shaders.

GPU Profiling There exist many analysis tools to analyze shader performance [17, 32]. They give information about how long a shader needs to execute or how much of the memory bandwidth it uses. But almost all of them stop at this level and are not able to give instruction-level insights into the performance of shader code like where it spends the most time, which branches are taken, how long different memory fetches take, etc. There exist static analysis tools that approximate this information [3]. However, as seen before with the comparison of ahead-of-time and JIT compilers, the knowledge of concrete input data enables some optimizations that are otherwise impossible.

Before we can start to implement more sophisticated optimizations in the compiler that rely on profiling data, detailed profiling information are needed. The focus of this thesis is to collect the necessary runtime information of GPU code, to use it for PGO. In the workflow of PGO, this is part of step one, namely inserting instrumentation when compiling code for the first time. The collected information from the second step can then be fed back into the compiler for use in optimizations.

1.2 Terminology

There exist quite a few different terms for the logical and physical parts of a GPU. Multiple vendors invent different terms and CPU notations like “thread” can be ambiguous when used in the context of GPUs. The logically similar meaning of a thread on a GPU would be a single lane for the SIMD units. On the other hand, a thread could be a description

for a complete SIMD unit which executes one program at a time, which makes sense from a hardware point of view.

Throughout this paper, we use the term *SIMD lane* for a single work-item. For a group of work-items that are run on the same SIMD unit, we use the expression *SIMD unit*. AMD refers to this group of items as wavefront while NVIDIA uses warp.

Concerning profile-guided optimizations, the term *benchmark* labels an automated, scripted execution of an application that is used to gather profiling data.

1.3 Contributions

This work contributes several instrumentations for Vulkan on AMD GPUs in the AMDVLK driver. We use these instrumentations on popular games and analyze the collected data. Using this data, we evaluate some possible, profile-guided optimizations on these games.

The first instrumentation inserts counters at basic blocks (BBs)[1] which gives us information about how often each BB is executed. This allows the compiler to detect hot paths in shaders and optimize them more aggressively. We measure and evaluate the effect of optimizations in LLVM when accurate BB execution counts are available.

Accurate counters also allow detecting unused basic blocks that are never executed in the selected benchmarks. We test the impact of removing this code and check if PGO and unused code affect the register allocation of shaders.

The second instrumentation counts how often certain variables are uniform or divergent at runtime. A uniform variable has the same value on all active lanes on a SIMD unit, the opposite is a divergent variable where the values differ across the lanes. Uniform variables permit optimizations that are impossible for divergent variables. E.g. for branches, the uniformity is important because the GPU might need to execute both branches on a SIMD unit and needs to run them sequentially, one after the other. We show statistics about the uniformity of branches in games. Additionally, our instrumentation allows us to analyze the uniformity of memory loads, which we present too.

To accomplish our goals, we reuse the existing infrastructure in LLVM for profile-guided optimizations. We extend and modify LLVM and the AMD Linux driver to support our instrumentations and use the collected data. To support PGO in the driver, we fixed various bugs in both, the hardware management layer of the driver PAL and in LLVM. In addition, we implemented loading executable and linkable format (ELF) sections onto the GPU and applying relocations to the code.

Finally, we discuss the effect of inserting instrumentation counters on the SIMD unit level[2] compared to inserting them on the lane level.

[1] See section 2.3 for an explanation of basic blocks

[2] After structurizing the CFG as explained in section 2.4

2 Background

This section introduces the hard- and software of graphics processors. Further, we explain concepts that are used in the implementation, like the ELF format.

2.1 Graphics Processing Unit

To understand how profiling data can – and cannot – be collected on GPUs, a basic understanding of how code is run on graphics processors is needed. This section gives a short explanation of the underlying hardware and in which points it differs from CPUs. Afterwards, we take a closer look at the communication between CPU and GPU. This is important to collect runtime information about shaders because this is the part where the collected analysis data is fetched and written into a file. The hardware section is largely based on a blog series about compute shaders by Matthäus Chajdas [10].

The general concepts, approaches, and results of this thesis should apply to all GPU vendors. For simplicity and because of the publicly available resources and code, we focus the work on AMD graphics cards and their open-source Vulkan driver AMDVLK.

To understand why GPUs are built the way they are, it is helpful to understand how developers write GPU code. Similar to CPU code, developers like to write simple code that does computations on a single data point. Much of the hard work like choosing instructions or parallelizing loops with single instruction, multiple data (SIMD) instructions can be done automatically by the compiler. This works similarly for GPU code. A developer writes code e.g. computing the color of one pixel in a game. The compiler then chooses the right instructions and in the end, the GPU does the computation for each pixel of a rendered triangle. The way SIMD instructions are used on CPUs and GPUs is quite different though.

On a CPU, SIMD instructions are often used to vectorize multiple iterations of a loop that are independent of each other. An equivalent way of expressing this would be to write SIMD instructions by hand. In summary, multiple instructions of the code a developer writes (which may be from multiple loop iterations) are fused into one SIMD instruction.

On a GPU however, loops are not vectorized. Instead, a program gets duplicated for multiple input data. For games, this means the program is started multiple times — one time per pixel — to compute the color for every pixel. If the program flow is the same for all instances and only the input values are different, the GPU runs (almost) every instruction of a shader on a SIMD unit. The multiple input and output data of the SIMD

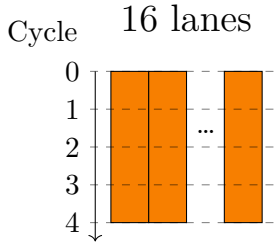


Figure 1: *Executing an instruction on a SIMD unit*

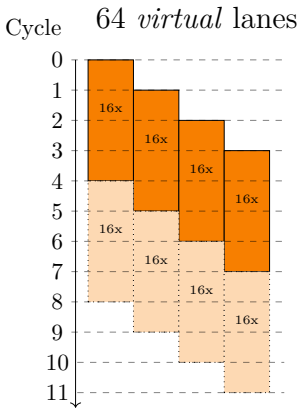


Figure 2: *Executing instructions on a SIMD unit with pipelining*

unit are from multiple instances (pixels) of the shader. As we see later, this poses a problem if the program flow is not the same. On CPUs, this concept of parallelism is similar to different threads, which only have limited access to each other and run the same code on different data.

Hardware This section describes the hardware architecture of AMD Graphics Core Next (GCN) GPUs as this is the architecture we are most familiar with. The GCN architecture was introduced in 2011. In 2019, the Radeon DNA (RDNA) architecture was published to supersede GCN, we will not dive into it and instead focus on GCN. The general concept of SIMD units is the same on NVIDIA graphic cards, though a difference is e.g. that there are 32 concurrent floating-point operations while there are 64 on AMD GCN cards.

On the hardware level, the basic building block of a modern GPU is a big SIMD unit. On the AMD GCN architecture, the same instruction is run on 16× 32-bit floating-point numbers simultaneously as shown in fig. 1. We call these 16 different numbers *lanes*. For most instructions, the pipeline has 4 stages and an instruction takes 4 cycles to complete. This also means that after the first cycle, the GPU can issue a second instruction, computing on another 16 floating-point numbers. This could be the next instruction in the program but then the GPU would have to care about data dependencies between instructions. This would add a lot of complexity so AMD decided to do something else instead. They virtually enlarge the SIMD unit to work on a multiple of 16 numbers and execute the same instruction again. The GPU can execute the same instruction 4 times, then the first 16 lanes finished the instruction. This means at one point in the pipeline, there are 4 instructions in flight, each computing on 16 numbers, which makes a total of 64 different lanes. For a developer, this looks like a SIMD unit that runs computations on 64 floats in parallel. An execution of two consecutive instructions is visualized in fig. 2.

In addition to the SIMD unit, a GCN GPU contains a few other processing units. One of them is the *scalar unit*. This unit comes along the SIMD units, so a program that is run on the GPU contains both, vector instructions for the SIMD units and scalar instructions for the scalar unit in an interleaved manner. The scalar unit is meant for computations which are constant among all SIMD lanes. Using the scalar unit instead of SIMD units is a lot more efficient in memory and energy because the data only needs to be stored once instead of 64 times, and operations only need to be executed once. Data values that are the same for a whole SIMD unit are called *uniform*, in contrast to *non-uniform*/divergent

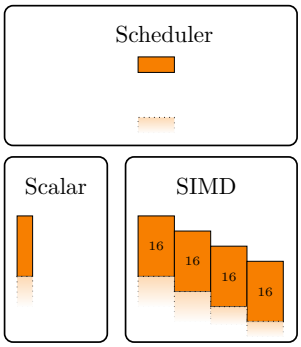


Figure 3: *Adding a scalar unit and an instruction scheduler*

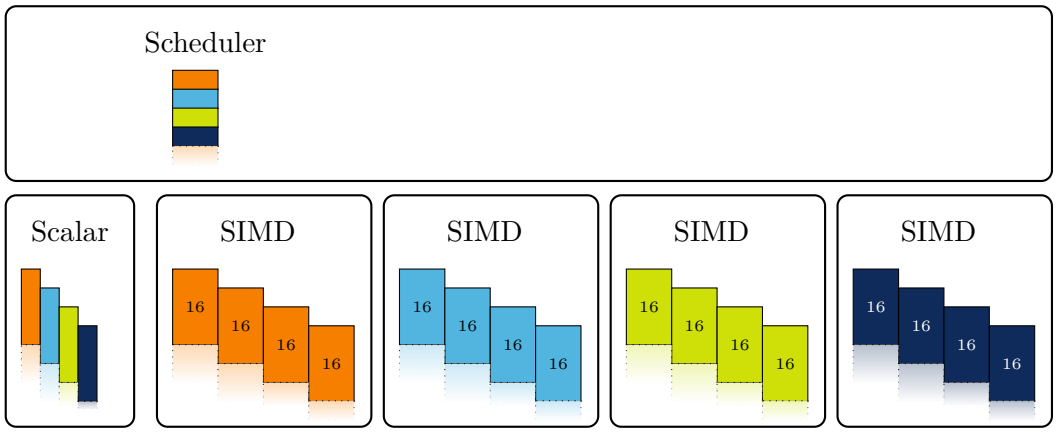


Figure 4: *The scalar unit, SIMD units and scheduler of one compute unit in the AMD GCN architecture*

values which are different in each lane. The scalar unit is also responsible for managing the control-flow of the SIMD unit. The vectorized instructions do not support jumps because a jump cannot be executed on a subset of lanes. Instead, jump instructions are executed on the scalar unit. We come back to that later, with a more detailed explanation of branching.

Another unit on a GCN GPU is the instruction scheduler. It is responsible for fetching new instructions and handing them to the responsible unit, e.g. the scalar unit or the SIMD unit. Figure 3 shows the three different components. For our current setup with one SIMD unit, one scalar unit and one instruction scheduler, the scalar unit will only get an instruction every 4 cycles because it works in synchronization with the SIMD unit and the instruction scheduler will only output instructions every 4^{th} cycle. The only unit which is occupied the whole time is the vector unit. For this reason, every scalar unit and instruction scheduler are responsible for 4 SIMD units. Figure 4 displays the pack of 4 SIMD units, one scalar unit and one instruction scheduler, which is called Compute Unit (CU). Additionally, one CU contains some local memory called local data share (LDS). One GPU consists of multiple such packs, for example, a recent AMD Radeon VII GPU contains 60 CUs. For a floating-point instruction such as an addition, which takes 4 cycles, this accumulates to a total of $60 \cdot 4 = 240$ SIMD units and a maximum of $240 \cdot 64 = 15360$ SIMD computations running in parallel. Every cycle, $240 \cdot 16 = 3840$ will get ready. And as a computation takes 4 cycles, 15360 are currently “in-flight”. The processor runs at a base

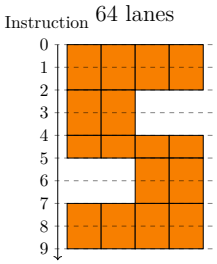


Figure 5: *Executing an* if-else *on SIMD units with an* EXEC mask

frequency of 1.4 GHz. At this speed, it finishes $5.376 \cdot 10^{12}$ vector instructions per second.

As emphasized before, having divergent control-flow is not simply possible for a program instances that get executed on one SIMD unit. For this purpose, AMD GCN GPUs have an *EXEC mask*, which contains one bit for every lane of a SIMD unit. If this bit is one, it means a lane is active, if it is zero, a lane is inactive and all SIMD operations are a no-op for this lane. When the program encounters a branch instruction, there are several possibilities. If all lanes take one branch, the scalar unit can execute a jump and execution continues as normal. If a part of the lanes take one branch and the rest takes another branch, both branches have to be executed. Typically, the execution mask for the first branch is set and the first branch is executed. Then the execution mask is flipped and the second branch is executed. After both branches are executed, the execution mask is reset to all ones again. An illustration of an *if-else* construct is shown in fig. 5.

Software This section explains how developers run code on GPUs. Developers who write code for GPUs do not have to handle all details of the hardware. Similar to CPUs where the operating system initializes the processors when starting and handles the resource allocations of processes, GPUs are managed by a *driver*. The driver abstracts from the concrete underlying hardware and gives developers an interface that is easier to use.

Similar to different operating systems and standards like POSIX and the Windows application programming interface (API), there exist different interfaces to communicate with graphic drivers. Two open standards, which are implemented by multiple vendors, are OpenGL and Vulkan. A specialty in comparison to CPUs is that the compiler is integrated into the driver in many APIs. For example, in OpenGL this would be accomplished with the `glShaderSource` and `glCompileShader` function. In Vulkan, there is `vkCreateShaderModule` for this purpose.

In this section, we explain the structure of compiling and running shaders with the AMDVLK Vulkan driver on Linux. This driver is based on the LLVM compiler framework. Typically, shaders for Vulkan are written in the OpenGL shading language (GLSL). The developer first has to translate this source coded to the intermediate language *SPIR-V*. The resulting SPIR-V code can be passed to the Vulkan API. From there it is passed to the llvm pipeline compiler (LLPC). The SPIR-V code gets then converted into the LLVM intermediate representation (IR). After some intermediate transformations on this IR, the LLVM compiler gets called. In this part, most optimizations happen. The LLVM framework also converts the IR to another IR, the *SelectionDAG* — as the name says a directed, acyclic graph of instructions [25]. The SelectionDAG gets linearized and transformed into *MachineIR* [25]. MachineIR is the intermediate representation which is most similar to the native Instruction Set Architecture (ISA) of the graphics card. The conversion to the ISA happens in the last step. The output of LLVM is a file in the executable and linkable

format (ELF). The LLPC applies some patches to the generated ELF file and then returns it.

When running a shader, the compiled ELF file is passed to the Platform Abstraction Library (PAL). This library is responsible for talking to the operating system and the in-kernel part of the driver also called kernel-mode driver (KMD). It reads meta-information from the ELF file and uploads the compiled code to the GPU to execute it. The stages to run code on a GPU thus look like this:

1. Developer writes human-readable GLSL code
2. `glslangValidator` converts GLSL to SPIR-V bytecode
3. Developer passes bytecode to Vulkan driver with `vkCreateShaderModule`
4. LLPC converts SPIR-V to LLVM IR code
5. LLPC transforms the LLVM IR to make it valid for the LLVM framework
6. LLVM optimizes the IR and converts it to the SelectionDAG
7. LLVM transforms the SelectionDAG to MachineIR
8. LLVM converts MachineIR to GPU ISA and returns an ELF file

The compilation is done now, the next steps are loading and executing the code.

9. PAL reads the ELF file and loads it onto the GPU
10. The GPU runs the code

GPUs wait a lot of time when fetching values from memory. To deliver good performance, they employ multiple techniques to hide memory latency and keep the arithmetic cores busy. Like CPUs, GPUs use simultaneous multithreading (SMT) to execute other code while waiting for memory in one program. The difference to CPUs is that GPUs can switch between programs a lot faster. A single cycle is enough for a context switch. Additionally, the amount of programs sleeping and waiting for memory concurrently is higher than on CPUs. While CPUs typically run two programs simultaneously, GPUs can have up to ten programs on a single SIMD unit [1].

However, this does not come for free. A high level of SMT needs lots of registers for all simultaneously executed programs as the registers cannot be fetched from memory for the fast context switches. The current approach of GPUs tries to get the best of both worlds, a large number of usable registers and a high level of concurrency. The amount of registers that a shader uses is dynamically allocated. If a shader uses a low amount of registers, it allows multiple other shaders to be run on the same SIMD unit with SMT. If a shader uses a high amount of registers, this level of concurrency will be lower. The amount of shaders that can run simultaneously is called *occupancy*. If the occupancy is too low, the GPU stalls when waiting for memory and will deliver degraded performance. Thus, the fewer registers a shader uses, the better the performance can get, up to a certain point.

2.2 Profile-Guided Optimizations in LLVM

The LLVM compiler framework has an existing infrastructure for profile-guided optimization. This infrastructure can be used by all frontends to LLVM, e.g. the C/C++ compiler `clang` or the Rust compiler. LLVM supports two different profiling techniques, a sampling profiler, and instrumentation. This work makes use of the instrumenting profiler.

2.3 Basic Block Counting

Compilers often use a control-flow graph as their internal representation of a program. This graph consists of nodes and edges. The edges are jumps or branches in the program, each node consists of a list of instructions that are executed in order. Inside a node, no control-flow exists, there are no jumps to other instructions and edges which lead to a node always start with the first instruction. The nodes are called basic block (BB) as they are the building block for every program.

One information that can be collected through profiling is how often each basic block is executed. The simplest version of such an instrumentation would insert one counter at each basic block. Let us consider an example with an if-else-statement. The instrumentation would insert four counters: Before the branching, in the if-block, in the else-block and after the branching in the following block. However, the same information can also be obtained with only two counters. If the if-part and the total executions are collected, it is possible to get the counter of the else-part by subtracting the if-part counter from the total amount.

In 1973, Knuth and Stevenson [18] showed an algorithm to find a minimal set of blocks where a compiler needs to insert counters. This algorithm is implemented in the instrumentation profiling of LLVM, and automatically gets used when turning on basic block counting.

2.4 Structurizing the Control-Flow Graph

A single SIMD unit cannot execute divergent control. However, the shader code that the compiler gets is not in a form which can handle divergent branches. Therefore, the compiler needs to change the CFG in such a way, that it gets suitable for SIMD units and gracefully handles diverging control-flow. In LLVM, this pass is called `StructurizeCFG`.

The CFG structurization converts the control-flow so it sets the EXEC mask and runs both, the if- and the else-branch if necessary. The result of such a transformation is shown in fig. 6. Loops are transformed similarly, such that the whole unit runs the loop until the last lane finishes.

2.5 The Executable and Linkable Format

The executable and linkable format (ELF) is a file format for programs, libraries object files and coredumps. It is used mainly on Unix systems and is the main format for executables on Linux. In the AMD Vulkan driver, LLVM outputs an ELF object file. This ELF object is then passed to PAL, which uploads the code to the GPU and starts the execution. Some parts of the ELF are interesting to us so this section provides a summary of their structure and functionality.

An ELF object file contains a variable amount of sections. The `.text` section, for example, contains the code, also called program text. The `.data` and `.rodata` sections

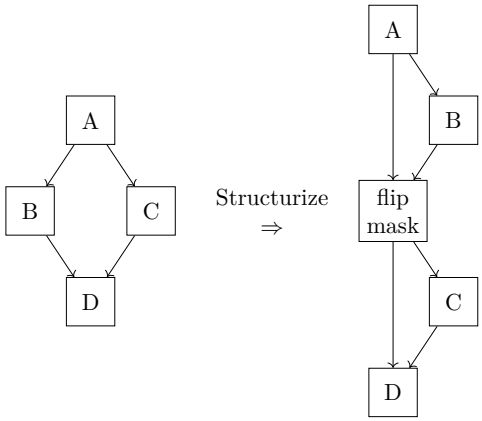


Figure 6: *Transforming the CFG for an* if-else *on SIMD units with the* `StructurizeCFG` *pass*

contain writable and read-only data that will be available to the program at runtime. There are also sections that contain meta-information for the loader. The loader is the application that loads the ELF file, in AMDVLK this is a part of PAL.

An essential part of the meta-information are the symbols. A symbol associates a certain region in a segment with a name. E.g. exported functions from libraries are marked with a symbol. The ELF loader reads the symbols that an executable needs and matches them with the exported symbols of libraries.

Relocations Another use of symbols are relocations. When compiling code into an object file, the compiler does not yet know at which address the `.text` and `.data` sections will be placed. However, the code can contain references to the data section for reading and writing static variables, so the compiler needs to output instructions that use addresses in other sections. This is where relocation sections come in. The compiler uses zeroes as placeholders for the address and writes them into the `.text` section, for example for variable access — relocations can be used in any section. Then the compiler adds a relocation entry that instructs the loader to fill out the address in the `.text` section at runtime when it is known.

Each relocation section connects one symbol section with the section where the loader should fill out the addresses, for example, the `.text` section. The relocation section references the symbol section in its `sh_link` field — where `sh` means section header — and the target (`.text`) section in the `sh_info` attribute. Each relocation entry describes which symbol to use, how to compute the final address value and where to write it.

The different ways to compute addresses are encoded in the type field and are dependent on the hardware architecture. Two different types that are found for many architectures are absolute and relative relocations. An absolute relocation simply writes the address of a symbol (plus an optional offset). A relative relocation subtracts the target address (where the loader writes the result) before writing the resolved address. This is used for relative addressing, where the code usually loads the instruction pointer and adds a constant. [7]

After the compiler emits object files, the linker merges multiple object files into an executable. It merges similar sections into *segments* (also called *program headers* in ELF). At runtime, the segments of one library are loaded at a random offset address but with fixed distances between each other. This means the linker knows the relative distance between two addresses in the code (and data) and is able to resolve all relative relocations that refer to a symbol in the same ELF object. Of course, it cannot resolve

relative relocations to other libraries as their distance is not fixed.

This could work in the same way on GPUs. However, PAL does not support allocating memory at fixed addresses at the moment and it does not perform relocations (the latter was fixed as part of this thesis). The AMDVLK driver needs to skip the linker step and compiled shaders are stored as ELF object files, like `.o` files when compiling C code.

Position Independent Code While relocations are still in use today, the plain use of relocations has several disadvantages. As the loader changes the content of the code section, the kernel cannot share these sections in memory between multiple programs or even multiple instances of the same program because the addresses are different each time. Another disadvantage is that the loader has to go through all usages of every relocated variable and function which can cause a long loading time for programs. The solution is to write all addresses that have to be linked into one section and only once. This section is called the global object table (GOT). When the address of a linked variable is needed, its address is loaded from the GOT and dereferenced. Functions are called similarly. This Position Independent Code (PIC) approach enables sharing the code sections between processes because only the GOT is different. A disadvantage is that two dereferences are needed compared to only one with plain relocations. [8]

Today, PIC is the standard technique for libraries on Linux. For GPUs, the advantages of PIC are currently not relevant as there are no shared libraries that have to be linked so no code is shared between processes. On the other hand, the disadvantage of an additional dereference is relevant, so currently, it is not advantageous to use PIC on GPUs.

3 Design

In this section, we explain various possible profile-guided optimizations and the data we need for them. This sets the foundation for our instrumentation. On a high level, we describe how the needed data can be collected by instrumentation.

3.1 Workflow

PGO is a common technique for applications running on CPUs. Developers are already used to the way PGO is applied. When using a compiler that builds upon the LLVM framework, the workflow is the following:

1. The application is compiled with profiling instrumentation
2. The application is run n times and produces an output file on each run
3. The n files are merged into a single file which contains all profiling information
4. The application is compiled again and optimized using the data from the generated file

As the AMD Vulkan driver is based on LLVM and developers already know this workflow, it suggests itself to use the same workflow for PGO on GPUs. When using PGO with `clang`, command-line arguments tell LLVM which options should be used. When invoking the compiler through the Vulkan API, using command-line arguments is not possible. To keep the usage and implementation of options simple, we decided to use environment variables to pass filenames and other options.

3.2 Optimizations

The goal of this thesis is to build the instrumentation, which is used in the first step of the PGO workflow. The instrumentation collects information at runtime and is needed to perform optimizations afterward. Before we can start collecting data, we need to know which information is needed. Therefore, it is part of this thesis to find out which information can enable optimizations and should be collected and also how they can be collected.

The needed data is defined by the optimizations we want to perform in step four. In the following, we list some possible optimizations and the respectively needed profiling data.

The optimizations stem from various sources, e.g. PGO optimizations for CPUs [30] and optimizations tailored to GPUs, as there are some peculiarities that have to be taken into account. Some described optimizations are specific to the current architecture of GPUs and have no close counterpart on CPUs.

3.2.1 Improve Linearization

Description As described in the introduction, moving seldom taken branches out of the usual control-flow improves performance.

Background This optimization is implemented in LLVM [24]. To avoid generating worse code, it is only applied when profiling data is available or the compiler assigned a very high probability to the hot branch.

Needed data Branch probabilities

Example

```
    ; v0 == 0
    v_cmp_neq_f32_e32 vcc, 0, v0
    s_and_saveexec_b64 s[0:1], vcc
    s_cbranch_execz else
    ; then-branch
    s_branch endif
else:
    ; else-branch
endif:
    s_endpgm
```

Listing 2: *Linearization — unoptimized*

```
    v_cmp_neq_f32_e32 vcc, 0, v0
    s_and_saveexec_b64 s[0:1], vcc
    s_cbranch_execz else
    ; then-branch
endif:
    s_endpgm     ; no jump if the
        condition is true
else:
    ; else-branch
    s_branch endif
```

Listing 3: *Linearization — optimized*

3.2.2 Hoist Texture Loads from Branches

Description Texture fetches that happen in branches are sometimes moved before the branch by the compiler to be executed unconditionally. This has the effect that while waiting for the texture content to be loaded from memory, the shader can do computational work and does not have to wait. The optimization speeds up the execution if the branch condition for the texture fetch evaluates to true. However, if the condition evaluates to false, it will degrade performance because the GPU unnecessarily fetches memory and occupies bandwidth.

Background Some GPU drivers are hoisting texture fetches out of branches, with guessing the probability that a branch is taken.

Needed data Branch probabilities

Example

```
    v_cmp_neq_f32_e32 vcc, 0, v0
    ; Set EXEC mask to cmp result
    s_and_saveexec_b64 s[0:1], vcc
    ; Skip branch if no lane is active
    s_cbranch_execz endif
    ; Load from texture
    image_sample v[0:3], v[5:6], s[0:7],
         s[12:15] dmask:0xf
    ; Wait until texture is loaded
    s_waitcnt vmcnt(0)
endif:
    s_or_b64 exec, exec, s[0:1]
```

Listing 4: *Load Hoisting — unoptimized*

```
    image_sample v[0:3], v[5:6], s[0:7],
         s[12:15] dmask:0xf
    v_cmp_neq_f32_e32 vcc, 0, v0
    s_and_saveexec_b64 s[0:1], vcc
    s_cbranch_execz endif
    s_waitcnt vmcnt(0)
endif:
    s_or_b64 exec, exec, s[0:1]
```

Listing 5: *Load Hoisting — optimized*

3.2.3 Improve Register Allocation and Occupancy

Description The number of registers that a shader uses can limit the number of shaders that run on a single compute unit in SMT fashion. If a single instance of a shader needs many registers, it prevents multiple instances of them running on the same SIMD unit because the register memory is not large enough to hold the registers for more instances at the same time. Therefore, reducing the registers that a shader needs or spilling registers which are rarely needed into memory can improve performance. However, registers in the hot path should not be spilled, as this will degrade performance. Branch probabilities can provide the information, where the register allocator can spill registers and where not.

Background Deciding spilled registers based on profiling information is already done for CPUs [4]. A difference is that on CPUs, the amount of registers is not dynamic but static. So storing variables in memory instead of registers to use fewer registers would never result in a performance gain.

Needed data Branch probabilities

Example For example, a game has a single shader for most of the objects that get drawn. This shader implements all material properties though not all properties are active for each object. One of the effects — e.g. subsurface scattering — needs a lot of registers but is enabled for few objects only. The high maximum number of used registers forces a low occupancy. In this case, spilling registers in the subsurface scattering implementation leads to lower register usage, higher occupancy and improves the performance.

3.2.4 Value Profiling

Description If a variable has a constant value in most executions, the compiler can introduce a branch and apply constant propagation to provide a fast path. This increases the code size, as parts of the code are duplicated, but can speed up the shader for the common case.

Background The concept of value profiling exists since 1997 [9]. But there is no detailed profiling for GPUs so far, so this is only done for CPUs. In LLVM, value profiling is used to inline e.g. `memcpy` calls when the size of the copied memory region is small enough.

Needed data Variable value distribution

Example A shader may compute textures for multiple material effects and blend them in the end by using a material texture with weights for every effect. If an effect is used seldom, its weight is often zero. The compiler should skip the computation of the effect in this case.

```
1  v_mov_b32_e32 v1, 0
2  ; Lots of computations, result is in
       v1
3  ; The weight for the effect is in v0
4  v_mul_f32_e32 v1, v1, v0
```

Listing 6: *Value Profiling — unoptimized*

```
1  v_mov_b32_e32 v1, 0
2  ; Skip if v0 is zero
3  v_cmp_eq_f32_e32 vcc, 0, v0
4  s_and_saveexec_b64 s[0:1], vcc
5  s_cbranch_execz endif
6  ; Lots of computations
7  v_mul_f32_e32 v1, v1, v0
8  endif:
9  s_or_b64 exec, exec, s[0:1]
```

Listing 7: *Value Profiling — optimized*

3.2.5 Detect Uniform Computations

Description If the shader does a computation or memory fetch on the same values for every lane on a SIMD unit, it is uniform. Uniform code can be optimized to be executed only once by the scalar unit instead of the vector unit.

Background This is similar to value profiling but specialized to GPUs. It does not consider isolated executions of a program. Instead, the instrumentation needs to inspect if values are different across one parallel execution. Compared to value profiling, the actually used values are not interesting, only if a value is uniform or not has to be saved. Using scalar instructions for uniform variables yields great benefits as described in a paper by Chen [11].

Needed data Uniformity of variables on a SIMD unit

Example If a shader loads an element from a uniform buffer, and the compiler knows through instrumentation that the offset into the uniform buffer is usually the same on all SIMD lanes, it can optimize that load. The compiler can insert a branch that checks if the offset is uniform at runtime. If it is uniform, the scalar unit loads the element from the uniform buffer. If the offset is not uniform at runtime, the shader falls back to the original, non-optimized variant.

3.2.6 Skip Branches

Description If the compiler encounters a non-uniform branch, it inserts code to set the EXEC mask and a jump which skips the branch on the SIMD unit if no lane wants to execute it. If the compiler knows that the branch will be executed (almost) always, it can omit the skip-jump and save a few instructions.

Background This optimization is an example that has no counterpart for CPUs. CPUs cannot run SIMD instructions only on a part of the lanes. This is a special case of detecting uniform computations where we are interested in the uniformity of the branch-condition.

Needed data Branch probabilities and uniformity on a SIMD unit

Example This optimization saves only a few instructions. An example where this can be useful nonetheless is in a hot loop where optimizing away a single instruction has a measurable impact.

3.3 Storing Data

On CPUs, LLVM generates code for writing the collected data into a file. Most of the file writing code is implemented in a static library which gets linked in when compiling a program with PGO instrumentation. The code creates a file containing all the profile data, consisting of counters, metadata and variable values if value profiling is enabled. To call the generated code when the application exits, LLVM uses a global destructor. The destructors are a list of functions that are called when an application quits.

GPUs currently do not support the destructor list. Additionally, shader code is unable to access the file system of the host. So directly creating and writing result files from the GPU is impossible, we have to use the driver on the CPU for this task.

We want to reuse the file writing code from LLVM but inside the driver instead of inside the shader. Therefore, we link the static library of LLVM into the driver. Before calling the dump file function, PAL maps the counters in GPU memory to the CPU and tells LLVM the name of the file that should be written. The filename is set by an environment variable and it can contain the pipeline id so the data does not get overwritten when saving multiple pipelines.

3.4 Basic Block Counters

The LLVM project, their compiler framework, and their C compiler clang successfully use BB counters for profile-guided optimizations on CPUs. Thus, BB counters suggest themselves to be used for PGO on GPUs too. In LLVM, the counters are used mostly for linearization. To get the accurate information at this stage in the compiler, the CFG is instrumented after most optimizations, when it will not be changed significantly anymore. An earlier instrumentation can not give us the needed information when basic blocks get duplicated because they would be counted as the same block. An early instrumentation can also harm following optimizations, e.g. by making it more difficult to merge duplicated code when different counters get incremented.

After the optimizations, the CFG changes one more time on GPUs using the single instruction, multiple threads (SIMT) model: The CFG structurization that makes it possible to run shader code on SIMD units. The counter instrumentation can be added either before the CFG structurization or afterward. A detailed comparison of both options can be found in section 6.1. For this work, we implemented and compare both methods.

After capturing counters, the next task is to use them. As LLVM is already using basic block counters for linearization if they are available, an optimized linearization is a low-hanging fruit.

3.5 Removing Unused Code

Another interesting optimization affects the number of used registers. The number of registers that a shader needs can limit the occupancy on a compute unit, i.e. the level of multithreading, which is important to hide memory latency. If the occupancy is too low, the GPU cannot hide memory latency successfully and it will be waiting a significant amount of time. To improve that situation, some registers can be spilled to (scratch) memory and increase the occupancy in that way. To decide where to spill registers, the compiler needs to know which parts of the code are executed seldom or not at all. Basic block counters can give us this information. To estimate the possible performance improvement without

the need to fully implement this optimization, we remove all parts of the shader code that are never executed and measure the impact on used registers and performance. We cannot use this strategy in a production driver because code that is not executed in a PGO benchmark scene is not necessarily dead code. However, this test can provide us valuable information if such an optimization can be worth the effort.

The normal switch-case lowering pass in LLVM works on the machine IR level. The amdgpu backend, however, needs to structurize the branch instructions which are the result of the lowering pass to support reaching different case-blocks on multiple SIMD lanes. This currently happens on the IR level, thus before the usual switch-case lowering pass. To be able to structurize the branches of a switch-case, the amdgpu backend uses a second implementation of switch-case lowering that works on the IR level instead of the machine IR. This pass is meant to be a temporary replacement for targets that cannot yet make use of switch instructions in the machine IR and not as a permanent solution. The pass converts the switch into a binary jump tree but it does not honor basic block frequencies in the process. We want to find out if rebalancing the jump tree to reach frequent blocks faster can lead to performance improvements. A small sample program which runs a bytecode virtual machine in a shader with one large switch-case statement is used to test this.

To reach the two goals, studying register usage and optimized jump trees, we implemented a pass in LLVM which removes basic blocks that are never executed and replaces them by the unreachable intrinsic. The collected basic block counters may not be perfectly accurate because sometimes, the profiling data is dumped while a game is still running. As the counters for some blocks are computed via other counters instead of being collected via atomics, even basic blocks that are never executed can have counters greater than zero. To remedy these inaccuracies, we consider each basic block as unused code that is executed less than 0.001 % of the shader executions.

3.6 Uniform Branches

Not only can the compiler optimize the number of branch instructions in a switch-case, but it can also remove branches in an if-else if the condition is always divergent. I.e. if most of the times both branches are executed on a SIMD unit for different lanes and the branches are sufficiently short, the compiler can just set the EXEC mask and not skip a branch if the EXEC mask is zero.

The first idea to detect uniform and non-uniform branches is simple. For an if-else structure, the shader counts the executed basic blocks per SIMD unit, so afterward the compiler knows how often the unit ran the if-block and the else-block. These numbers are sufficient to compute how often both blocks are executed after each other. If both blocks are executed, we know that the condition is not uniform, otherwise, either the else-block or the if-block would have been executed, not both. So we can compute how often a condition is uniform or not. However, this technique does only work if there is an else-block. Without else-block, the uniformity of a condition cannot be computed this way. In addition to that problem, the uniformity of other variables, e.g. memory addresses is also interesting as we show in the next section. For these reasons, we decided to implement a generic analysis that finds out how often a variable is uniform and which can be applied to any variable.

The decision, if a branch is taken uniformly or not is dependent on the condition. If the condition variable is uniform, the branch is taken uniformly. If not, the branch diverges. Therefore, analyzing the uniformity of branches is a special case of analyzing

generic variables for uniformity where the interesting variable is the condition for a branch instruction. The analyzed variables are always booleans in this case.

3.7 Uniform Variables

This section describes how we analyze at runtime if generic (integer and float) variables are uniform or divergent. Uniformity plays an important role in the efficiency of GPUs [11], thus we want to improve the handling of uniform variables. Uniform operations need to be executed only once per SIMD unit instead of 64 times, and they can be run in parallel to other instructions that are executed per lane. These possible optimizations make it desirable to find out which variables are uniform and which divergent. LLVM contains a static divergence analysis for this purpose which uses known sources of divergence like shader input variables and marks all possibly divergent variables. All non-marked variables are known to be uniform.

However, this analysis may not find all uniform variables. Some variables may always be uniform at runtime but the compiler does not have enough knowledge about the application logic to prove that it is always the case. Additionally, it may be interesting to know variables which are uniform in *most* cases. If the compiler knows variables that are mostly uniform, it can introduce specialized fast paths and fall back to a slower, non-uniform version if necessary.

To detect, if a variable is uniform during runtime, we use the `read_first_lane` intrinsic, which transfers the value of a 32-bit variable from the first active lane to every lane. Each lane compares if the value in this lane is not equal to the value of the first lane. The `icmp` intrinsic gives us a 64-bit integer with one bit for the comparison result of each lane. If one of these bits is one, so the integer is unequal to zero, the variable is **not** uniform. In the divergent case, the shader increments a counter. At the end of the execution, if this counter is zero, the variable which is associated with this counter was always uniform. To check if a variable is statically uniform, detected by the compiler, the `DivergenceAnalysis` in LLVM is used. Unfortunately, the analysis sometimes returns inaccurate results, depending on the position in the pipeline when it is run.

4 Implementation

Our implementation happens in several steps of the PGO workflow and touches multiple parts of the AMDVLK driver. First, the LLVM compiler inserts instrumentation instructions. LLVM can already emit instrumentation instructions for counting basic blocks and it can be activated with a few lines of code in the llvm pipeline compiler (LLPC). The next step is loading the compiled binary onto the GPU. Therefore, code in the Platform Abstraction Library (PAL) needs to be adjusted as summarized in section 4.3. After loading a shader, it is run. This part is unmodified by our work. Then, when a shader pipeline is destroyed again, the driver fetches the collected information and writes them to disk as explained in detail in section 4.4. The next compilation of a profiled shader can then incorporate the collected information, which is described in section 4.5. All of this gives us the possibility to use basic block counters for PGO.

The more advanced analyses and implementations reuse this infrastructure and extend it with new LLVM passes. The implementation of these passes is described starting in section 4.7.

4.1 Workflow

As described in the design section 3.1, options are passed to the driver through environment variables. The concrete steps are:

1. Set `AMDVLK_PROFILE_INSTR_GEN` and other variables to a profile file path
2. Run the application n times which produces an output file on each run
3. Merge the n files for each pipeline
4. Set `AMDVLK_PROFILE_INSTR_USE` to the path of the merged files (and more variables) and run the application again

The paths to profile data files should contain a `%i` which will get replaced by the pipeline id when data is loaded and stored. When generating data, `%m` can also be used to insert a unique id by LLVM into the filename.

The following options can be changed by environment variables:

- `AMDVLK_PROFILE_INSTR_GEN`: Generate instrumentation in shader code, use e.g. `Pipeline_%i_%m.profraw`

- `AMDVLK_PROFILE_INSTR_USE`: Optimize shader code, e.g. `Pipeline_%i.profdata`
- `AMDVLK_PROFILE_LATE`: Add PGO passes after structurization, default is before `StructurizeCFG`
- `AMDVLK_PROFILE_PER_WAVE`: Count executions per SIMD unit instead of per lane, only use this in combination with late instrumentation!
- `AMDVLK_PROFILE_ANALYSIS`: Add analysis passes and write results to temporary file
- `AMDVLK_PROFILE_UNIFORM`: Add passes to generate and use instrumentation for uniform variables
- `AMDVLK_PROFILE_REMOVE`: Remove basic blocks which are never executed, this has only an effect when using PGO data
- `AMDVLK_PROFILE_NON_ATOMIC`: Use normal counters instead of atomic counters for instrumentation

The detailed effect of these variables and how they are implemented is described in the following sections.

4.2 Compiling Shaders

This section describes our modifications of the compiler pipeline. The changes are applied to LLPC and LLVM, which compile shaders in SPIR-V to the hardware ISA.

When the environment variable `AMDVLK_PROFILE_INSTR_GEN` is set, the compiler inserts instrumentation into shaders. There, we have to distinguish between two possibilities: In the default case, any `%i` in the filename is replaced with the pipeline id and this filename is provided to the `PassManagerBuilder`. Otherwise, if the variable `AMDVLK_PROFILE_LATE` is set, the instrumentation is inserted after the `StructurizeCFG` pass. Therefore, the `AMDGPUTargetMachine` pass adds all needed PGO passes after structurization.

To use the most basic instrumentation with the `PassManagerBuilder`, LLPC only has to set one of its member variables to the requested filename. To support more of our use cases, the `PassManagerBuilder` has a new member variable of the class `InstrProfOptions`. This allows LLPC to set more options and we do not have to modify the `PassManagerBuilder` for them. One of these options decides if atomic counters are used or not. This property is set to true except if the environment variable `AMDVLK_PROFILE_NON_ATOMIC` is set. Another change in the `PassManagerBuilder` for our implementation is that our new PGO passes are added if necessary. For example the analysis pass or the uniformity instrumentation generation pass.

When instrumenting after structurization, `AMDGPUTargetMachine` adds all needed passes. It handles the same options like atomic counters but without the `PassManagerBuilder`, so the counter instrumentation and use passes have to be added explicitly.

The PGO passes in LLVM generate an ELF file with additional data and code sections that contain e.g. basic block counters and the registered destructor to write the collected data to disk. We cannot directly use the destructor generated by LLVM because code on the GPU does not have access to the file system on the host, so we deactivate this code. Our solution to save the collected data is described in section 4.4.

4.3 Loading Shaders

After LLPC and LLVM in the AMDVLK driver compiled the code, PAL is responsible for loading the shader ELF files. This means copying the compiled code to GPU memory and running the shaders. The previous ELF support of PAL was limited as it only extracted the `.text` and `.data` section (code and data) and uploaded them to GPU memory. For PGO instrumentation, PAL also needs to load additional data sections for counters and metadata. As part of this work, we implemented loading all sections in an ELF file which are marked with write or execution flags.

To increment counters, the code in the compiled object file references the counters in the data section. As the final addresses are not yet known at compile-time, LLVM adds relocation entries for them as described in section 2.5. We implemented applying these relocations in PAL. The steps in PAL to load a shader are:

1. Allocate memory for all sections on the GPU, so the driver knows their final addresses
2. Copy sections to the allocated GPU memory by mapping the GPU memory to the CPU
3. Iterate through all relocation sections and perform the relocations

When performing the relocations, PAL reads from the loaded ELF file on the CPU and writes to the GPU mapped memory. Reading from GPU mapped memory would mean the CPU has to wait until the data arrives through the PCIe bus, which takes longer than reading from CPU memory. On the other hand, writing is fine because the CPU does not have to wait until data is written, it can just continue, thus hiding the latency.

PAL's ELF parser contained a bug so it only supported `sh_link` and `sh_info` references to already loaded sections. This bug was fixed as part of this thesis.

4.4 Fetching and Storing Data

After loading and running a shader, the driver must save the counter values. As described in section 3.3, the shader itself cannot create the data files. Instead, the driver has to fetch the data from GPU memory and write it to files.

Most parts of the file writing code of LLVM are inside a library. The library gets statically linked to the compiled program when PGO instrumentation is enabled. This does not happen for shaders because they are never linked in the current implementation. We decided to link that library into PAL, so the driver can access the file writing code. The needed information to write a file are the addresses and sizes of the PGO related ELF sections. PAL can read this information from the pipeline ELF file.

When a pipeline gets destroyed because the application does not need it anymore, PAL writes the collected counter data to disk. However, not all games destroy pipelines in the end, some just quit. To get the counter data for these games, we start a background thread that dumps the counter data every ten seconds. A problem with this approach is that collected counters can be inconsistent when a shader is in use while the data is dumped. For applications that destroy pipelines, we get accurate counters in the end, this problem only exists for programs that do not delete pipelines. To collect counters, the driver maps the sections from the GPU memory to the CPU, writes their addresses into global variables and calls the LLVM PGO file dump function. We explicitly specify the filename and replace any `%i` inside the name with the current pipeline hash.

There is a single problem with counting on the GPU and dumping the data from the CPU: The metadata contains the absolute memory addresses of the counters and LLVM also dumps the absolute address of the counter section. The counter addresses are set through relocations when loading the ELF file which means they are virtual addresses on the *GPU*. The address of the counter section, on the other hand, is a virtual address on the *CPU* which maps to GPU memory. To get a correct dump file, these addresses need to refer to the same address space. Therefore, before writing the data to a file, PAL replaces the GPU virtual addresses in the metadata with CPU virtual addresses.

4.5 Loading and Using Profile Data

Similar to compiling shaders, the LLVM `PassManagerBuilder` needs the filename with PGO data. As before, we replace a `%i` in the filename with the pipeline hash before handing the name to LLVM. LLVM will then add passes to set the basic block frequency information, which is used by subsequent passes for optimizations.

A problem that turned up when using profiling data is the `ControlHeightReduction` pass [38]. This pass is specific to profile-guided optimizations and it duplicates some basic blocks and — based on a condition — executes one or the other. For some shaders in Dota 2, it duplicated the last block which contains the `export` intrinsic call. This intrinsic is responsible for exporting a position in the vertex shader or a final color in the pixel shader. While the `export` instruction still gets executed once per SIMD lane after the block duplication, it now gets executed multiple times per SIMD unit. This leads to problems because it has a meaning for the whole SIMD unit, e.g. the `done` flag tells the GPU that this is the last `export` instruction on this unit. Executing multiple `export` instructions with the `done` flag set leads to GPU hangs. To fix this bug, we marked the `export` intrinsic as *convergent* which means it shall not depend on additional control-flow and we modified the `ControlHeightReduction` pass so it does not duplicate regions that contain convergent intrinsics.

4.6 Basic Block Counters

Counting executions of basic blocks is implemented in three different LLVM passes. The pass `PGOInstrumentationGen` inserts increment intrinsics into some basic blocks. After collecting data, `PGOInstrumentationUse` loads a `.profdata` file, reads the counters and sets the block frequency data. Creating the counter data sections, storing the metadata and emitting the actual add instructions happens in the `InstrProfiling` pass. This pass iterates over all counter intrinsics from the `PGOInstrumentationGen` pass and replaces them by add instructions.

These passes are already part of LLVM. We only modified the `InstrProfiling` pass to support per unit counters. The details of this change are described in section 4.11.

4.7 Unused Code

After collecting basic block counters, we can find code which is never executed. To get this information, we could look into the captured profiling dumps and search for counters which are zero. But not every basic block has its own counter, as some counters can be computed from others. As we do not want to miss these blocks, we wrote LLVM passes instead. One pass which needs information about unused code is our analysis pass, which

we use to collect information for diagrams in this thesis. Another pass is responsible for removing unused basic blocks if the environment variable `AMDVLK_PROFILE_REMOVE` is set. These passes run after the `PGOInstrumentationUse` pass so they have access to the block frequency information. As described in the design section, because of inaccuracies, not every unused basic block contains zero in the counter. Therefore, if a basic block is executed less than $< 0.001\,\%$ of the entry block executions in the containing shader, it is considered unused. This does not work perfectly for unused blocks in loops, where the inaccuracy could be greater, but it seems to suffice in our case.

Before removing an unused block, all references to this block need to be removed. If a conditional branch instruction references the block, the reference to the unused block can be removed by transforming the instruction to an unconditional branch to the second destination. There are more complicated cases, e.g. an unconditional branch to the unused block or references in a switch. All these references are redirected to a new basic block that only contains the unreachable intrinsic. Then, when no more references exist, the unused block is removed.

4.8 Uniform Branches

The design section for uniform branches explained that detecting the uniformity of branches is equal to testing if the branch condition is a uniform variable. The implementation details to find out if a variable is uniform or not are explained in section 4.9. This section explains peculiarities when applying this to conditions and branches.

As with basic block counters, the uniformity of branches can be captured before or after the `StructurizeCFG` pass or corresponding to that, per SIMD lane or per unit. Before structurizing, we should count the number of lanes that are affected by a divergent variable. So when a SIMD unit currently has 14 active lanes and the variable of interest is not uniform across these lanes, 14 is added to the counter. On the other hand, after structurization we want to count the units instead of lanes. This means for a non-uniform variable, the counter is incremented by one, no matter how many lanes are active.

The uniformity of branches is a special case when using this instrumentation after structurization. The structurization is the pass which handles and eliminates divergent branch instructions, thus afterward, every branch is statically known to be uniform. Hence, after the structurization pass ran, the normal conditions of branches cannot be inspected anymore because they do not represent the branches from the shader code. Instead, we have to take a look at what happens with conditions in the `StructurizeCFG` pass. The original conditions are used as an argument for the `amdgcn.if` intrinsic. This intrinsic returns a boolean that signals if the whole unit can skip the branch which is then used for the branch instruction. Therefore, if branches are instrumented before structurization, the compiler directly instruments the condition arguments of branches. After structurization, it looks at the argument of `amdgcn.if` intrinsics instead.

To find out, if a branch is statically known to be uniform, we use the `DivergenceAnalysis` before structurization. This analysis tells us for a condition if the variable is uniform or divergent. In single cases, it may not work perfectly, e.g. if a function argument for the shader is not known to be uniform at this stage in the compilation pipeline but overall it is working good. After structurization, if a block contains the `if` intrinsic, the condition is divergent. On the other hand, if a block contains neither the `if` intrinsic, nor an `amdgcn.else` or `amdgcn.loop` intrinsic, it must be uniform. At the moment, only ifs are instrumented after structurization, loops are ignored.

4.9 Uniform Variables

To track the uniformity of variables, we use simple counters that stay zero for a uniform variable or get incremented for a divergent variable. For the counters, we can reuse much of the code from basic block counters. We can emit the same intrinsics as `PGOInstrumentationGen` and the counter sections and more complex instructions will be handled by the `InstrProfiling` pass. We only have to take care that the counters for uniformity do not conflict with basic block counters. Thus, we assign the uniform counters to a new (nonexistent) function, with `-uniform` appended to the original function name.

For example, if the compiler wants to check if a variable `%0` is uniform, it emits the following code:

```
%1 = tail call i32 @llvm.amdgcn.readfirstlane(i32 %0)
; The 33 is the comparison code for non-equal
; Compare %0 != %1 and get the result for all lanes
%2 = tail call i64 @llvm.amdgcn.icmp.i32(i32 %0, i32 %1, i32 33)
%3 = icmp ne i64 %2, 0
%4 = zext i1 %3 to i64
; This intrinsic also takes metadata, like the function name
; Add %4 to the counter (0 if uniform, 1 if divergent)
call void @llvm.instrprof.increment.step(i8* getelementptr inbounds ([23 x i8], [23 x i8
    ]* @__profn__amdgpu_vs_main_uniform, i32 0, i32 0), i64 0, i32 18, i32 0, i64 %4)
```

Listing 8: *Check a variable for uniformity*

At the moment, we instrument conditions to find uniform branches and `LoadInstr`s. The interesting parts of a load instruction are the address — we want to know if we can use a scalar load — and the loaded memory value. This gives a coarse insight into how applications use memory loads and how diverse the content of their memory is.

The same analysis pass as for unused code creates statistics about the uniformity of variables The `DivergenceAnalysis` tells us if a variable is statically known to be uniform. If a variable is not statically uniform, it is dynamically uniform if the counter is zero or divergent if the counter is unequal to zero.

4.10 Analyses

Statistics are collected by a new LLVM pass called `PGOInstrumentationAnalysis`. This pass uses the information from annotations and writes the data to a log file. This log file is then processed by a Python script to aggregate the data for diagrams and tables.

One part of the collected statistic is the amount of unused basic blocks as described in section 4.7. These statistics lead to the evaluation in section 5.1. Other information which is collected through this pass is data about uniform variables. An instrumented variable can be either statically uniform, dynamically uniform or divergent. The most interesting cases are dynamically uniform variables: They are not detected by the compiler to be uniform, but at runtime, they turn out to be uniform.

There is one more part of the analysis which is not part of the LLVM pass. For debugging, the driver can dump all used pipelines to a folder. The dumps are in a human-readable form and contain the assembly code for the shaders and associated metadata. We use these files to get the number of registers used by the shaders.

4.11 Per Unit Counting

In the usual case, when simple atomic counters are used, one atomic operation per active lane is executed. This means each active lane will start an atomic add to the same memory location at the same time. These simultaneous atomics are quite slow and should be optimized to a single atomic operation per unit. When instrumenting before structurization, this could be accomplished by the atomic optimizer pass. Even if the atomic optimizer currently does not optimize the atomics for instrumentation, its purpose is to fuse atomic operations into a single operation per unit. After structurization, however, the atomic optimizer already ran, and we need another way. Also, counting per unit needs an atomic operation that gets executed on a single lane.

We implemented per unit counting in the `InstrProfiling` pass, which is responsible for lowering the profiling increment intrinsics.

An easy way to restrict the execution to a single lane is to set the EXEC mask to one. This means only the first lane will be active. The code in listing 9 accomplishes this with only three instructions overhead.

```
; %1 = EXEC
%1 = call i64 @llvm.read_register.i64(metadata !1) #1
; EXEC = 1
call void @llvm.write_register.i64(metadata !1, i64 1) #1
; Atomic increment
%2 = atomicrmw add i64* getelementptr inbounds ([3 x i64], [3 x i64]*
    @___profc___amdgpu_ps_main, i64 0, i64 1), i64 1 monotonic
; Restore the EXEC register from %1
call void @llvm.write_register.i64(metadata !1, i64 %1) #1

attributes #1 = { convergent }
!1 = !{!"exec"}
```

Listing 9: *Restricting execution to a single SIMD lane*

Unfortunately, these instructions do not mark any control-flow changes for LLVM so they can get reordered and mess up the rest of the code. After structurization, the code mostly stays in order, however, in some cases it still gets reordered, leading to graphical artifacts. To get stable code for incrementing counters on one lane only, we reuse code from the atomic optimizer. This code uses a few intrinsics to create a condition that will only be true for the first lane. Then it creates an if construct based on the condition where the if-block will only be run for a single lane. It expects however to be run before the structurization pass, which will transform the introduced branches. To be able to use this code in per unit counting, we adjust the emitted code in that case. If late instrumentation is active, the `InstrProfiling` pass directly creates code using the condition modifying intrinsics from structurization. I.e. it emulates the `StructurizeCFG` pass to run afterward.

The resulting code is more robust to LLVM reordering but also less efficient. Also it does not count blocks in the per unit mode when the EXEC mask is zero, so if no SIMD lane is active but the scalar unit may still compute something. It results in five instructions overhead when running before structurization and a few more when running afterward, also depending on the shader type (pixel shaders need special treatment when helper lanes are in use).

5 Evaluation

To evaluate our work, we run our analyses on various applications and also run benchmarks comparing the performance with and without profile-guided optimizations. We modified the Vulkan driver to enable PGO, so we need Vulkan applications for our benchmarks. We picked the following five games which have in common that they use the Vulkan API on Linux, and they have a benchmark which can be executed automatically and which collects performance data in a machine-readable form:

- Ashes of the Singularity, which is abbreviated as *Ashes* in the following
- Dota 2
- F1 2017
- Mad Max
- Warhammer 40,000: Dawn of War III, which we shorten to *Warhammer*

There are similar games, which we tried to integrate into our evaluation. We were not able to include the games F1 2018, Serious Sam Fusion 2017 and Total War Saga: Thrones of Britannia into our evaluation because of various problems. Some of them are not running on Linux and one is not even installable.

Another gaming focused application which we looked at is the Infiltrator Demo Project of the Unreal Engine [15]. This is not a playable game, but a demo scene which plays when the application starts and which showcases the game engine. Also, it does not measure the performance, so we did not use it to test the performance of PGO, but we measured the number of needed registers. Another non-game application which we used for evaluation purpose is a small switch-case bytecode interpreter that we describe further in section 5.2. We call this program *switch vm*.

For all these applications, we use a benchmark scene from the game. We use the same or similar options as Phoronix Test Suite [21], this means the in-game benchmarks for Ashes, F1 2017, Mad Max and Warhammer and a tournament scene for Dota 2. For Unreal Infiltrator and the switch vm, we use the application itself as the benchmark.

The hardware for our benchmarks consists of a Intel i7-5820K @ 3.30 GHz processor and an AMD Radeon RX 480 GPU with 8 GB of GPU memory. To make the benchmarks more reproducible, the GPU used the `profile_standard` performance mode, which means it runs at a fixed frequency. The CPU used the performance frequency governor and

turbo was switched off for benchmarks. The benchmarks are run in full HD resolution, so $1920 \times 1080 = 2\,073\,600$ pixel. In total, more than $6\frac{1}{2}$ h was spent benchmarking.

The performance numbers are expressed as frame times, i.e. how long does it take to compute a single frame. For example, a frame time of 16.6 ms corresponds to 60 frames per second. The lower the frame time is, the more pictures are shown per second and the better is the performance. The measured times are always given with a standard error using a confidence level of 68.3 % with the Student's t-distribution.

An application like a game uses a lot of shaders, Ashes has 250 different shaders, Dota 2 compiles 877 shaders for the benchmark scene and Mad Max 734 shaders. Warhammer uses the least amount with 225 shaders while F1 2017 needs the most with 3220. These numbers do not necessarily represent the total number of shaders in the games, but only the amount of shaders that are compiled for the benchmark scene. A difference here is that Ashes compiles most of its shaders at the start of the game while Dota compiles shaders lazily when they are needed, so we only see a fraction of the shaders that Dota contains, but we see a greater fraction of the shaders in Ashes. When evaluating basic block counters, unused code, and uniformity, we only consider shaders that were run at least once in the benchmark. For the register usage, we collect the statistics overall compiled shaders without taking into account if they were executed or not.

5.1 Basic Block Counters

Shaders are small programs compared to CPU programs, more like a single function. This means the number of basic blocks in a shader is low, usually around 20 or lower. After instrumenting shaders and running a benchmark, the compiler can associate a counter with each basic block.

A performance comparison using the default optimizations in LLVM that rely on PGO data is discussed in section 5.5. When instrumenting before the `StructurizeCFG` pass, an effect of PGO is that most pixel shaders are marked as hot functions while vertex shaders are often marked as unlikely. The reason is that a triangle contains three vertices but a lot more pixels. Hence a pixel shader gets executed a lot more often than a vertex shader and LLVM marks them accordingly.

To get an intuition for the collected counter values, we visualize them in a few diagrams. Figure 7 shows a histogram of the basic blocks, sorted by execution count. The total execution count is divided by the frame count of the benchmark and by the resolution of the screen. This means the x-axis of this diagram represents how often a basic block is executed on average per frame, per pixel. A bar for the execution counts 530–550, for example, represents all blocks which get run at least 530 times but less than 550 times. The height of the bar counts how many basic blocks fall into this range, in this example one basic block of all shaders in Dota 2 falls into the range. The lowest bucket of the diagram, the leftmost bar, contains most of the blocks. This bar is cut off, otherwise, the rest of the bars would not be visible. This means most basic blocks get executed less than 20 times per frame and pixel. A few blocks run more often, two of them more than 500 times, which means more than 10^{12} times in total over the duration of the benchmark. The blocks that are run most often are the inner parts of a hot loop.

Figure 8 gives a detailed view of the leftmost part of fig. 7. It only shows the range up to $5 \cdot 10^{-5}$ executions per frame and pixel. In this range, the blocks are distributed more equal, still, we see that most blocks are executed seldom while blocks with a higher counter value are rare.

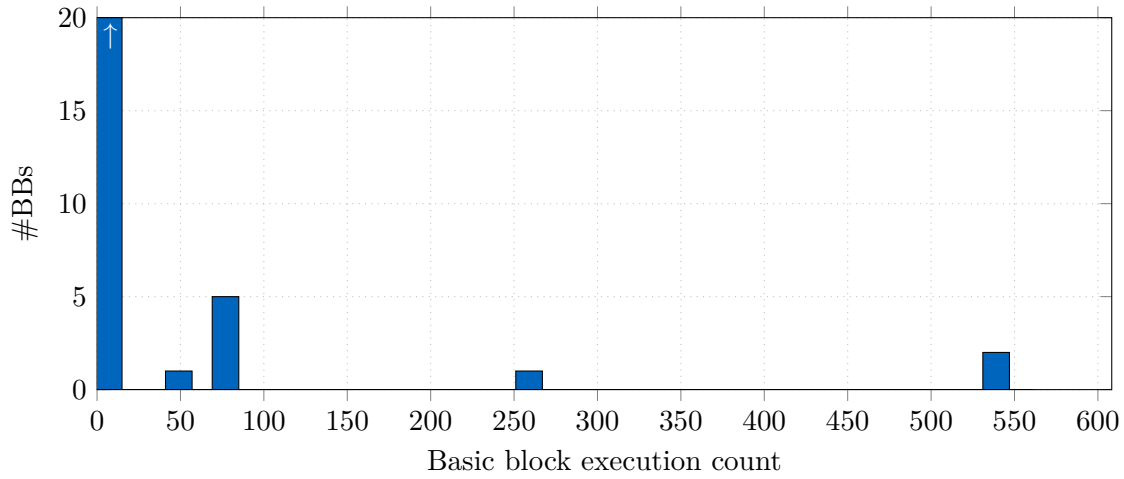


Figure 7: *The distribution of counter values on basic blocks of Dota 2*

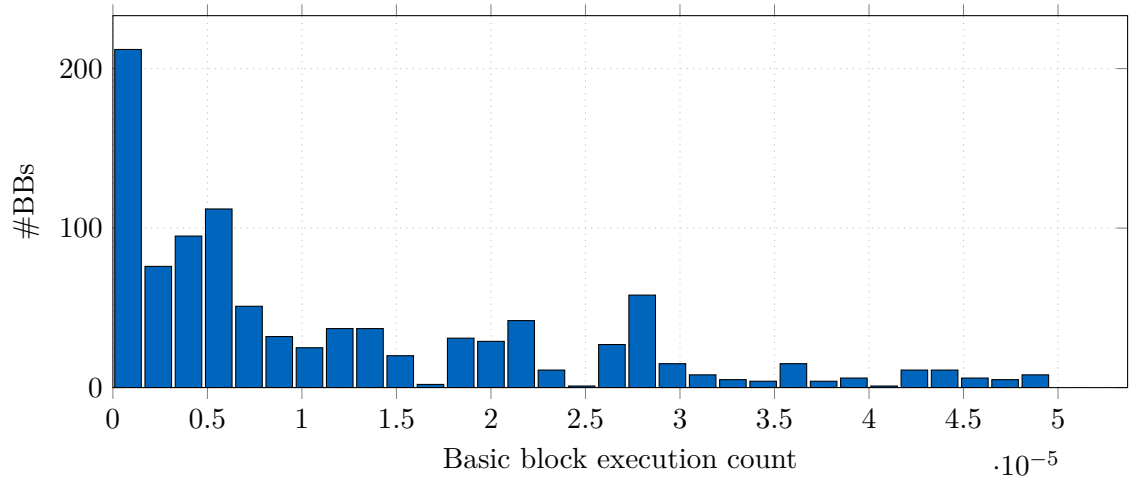


Figure 8: *The distribution of counter values on basic blocks of Dota 2 up to counters of* $5 \cdot 10^{-5}$

Ashes as of 2019-09-01 shows graphical artifacts when using PGO. This hints that there is a bug in LLVM that only gets triggered when profile-guided optimizations are enabled. So far, we did not find the cause for this behavior. In previous tests with PGO, Ashes did not show these artifacts, so maybe this bug is connected with a recent update of Ashes.

5.2 Unused code

Looking at basic blocks that are never executed, the compiler can remove them and study the changes in register usage and performance. The performance comparison can be found in section 5.5, this section shows statistics about the amount of unused code. The fraction of unused basic blocks ranges from 0.22 % in Mad Max up to 37.46 % unused blocks in Warhammer. Figure 9 shows a comparison of all analyzed games.

To gain more insight into how shaders and the unused code is distributed, fig. 10 and following diagrams show the unused code concerning the size of a shader in basic blocks. The x-axis represents the size of a shader, the number of blocks that a shader contains. The gray plot in the background displays the distribution of all shaders in an application.

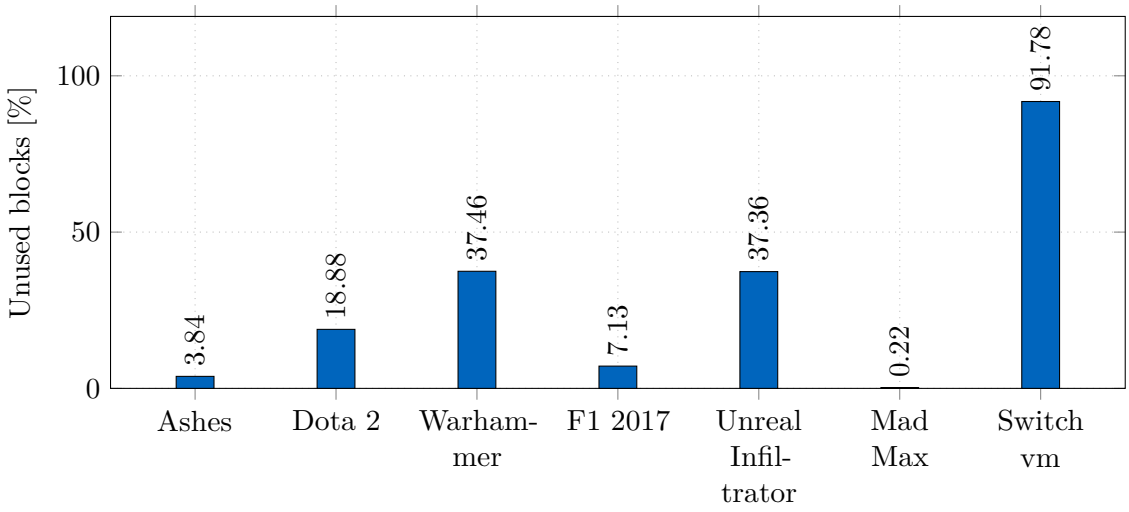


Figure 9: *The fraction of unused basic blocks per game*

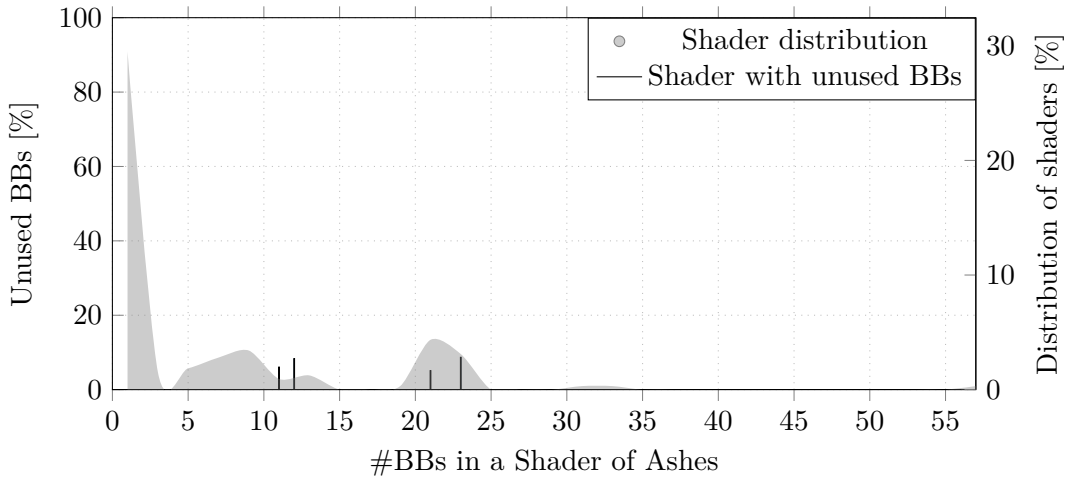


Figure 10: *The fraction of unused code by shader size in Ashes*

A height of 6 % at one point (the right y-axis is the crucial one here) means that about 6 % of all shaders in the game have this number of basic blocks. Each black line represents a shader with unused code. The left y-axis is connected to the percentage of unused code.

We test the effect of removing unused code on the games Ashes, Dota 2, Mad Max and on a small sample program. Our code to remove basic blocks is not robust enough to work for all tested games. Warhammer and F1 2017 crashed in this configuration. Ashes also did not work with our final technique that uses the LLVM unreachable instruction. Instead, we used a basic block that contained an endless loop and replaced unused basic blocks with this new block for Ashes. The Unreal Engine Infiltrator Demo contains some basic blocks that are executed seldom, so for this game, only blocks with a counter of zero are removed. This is not enough to get this game running, it crashes after about a quarter of the total time. Still, we analyze this game, but only using shaders that are compiled until the game crashes, which are 84.3 % of the 843 pipelines.

The small sample program which we tested in addition to the games runs a bytecode virtual machine in a shader using one large switch-case statement. Each case statement

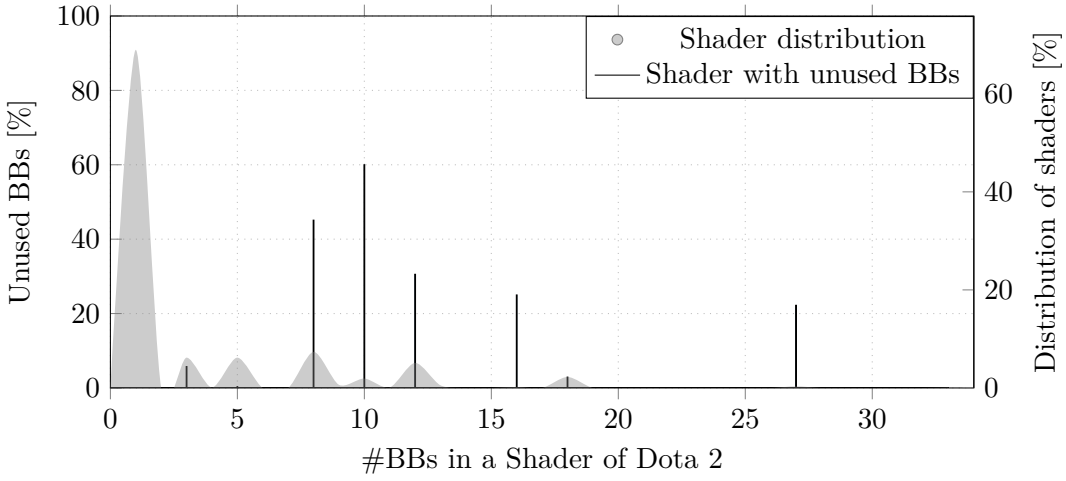


Figure 11: *The fraction of unused code by shader size in Dota 2*

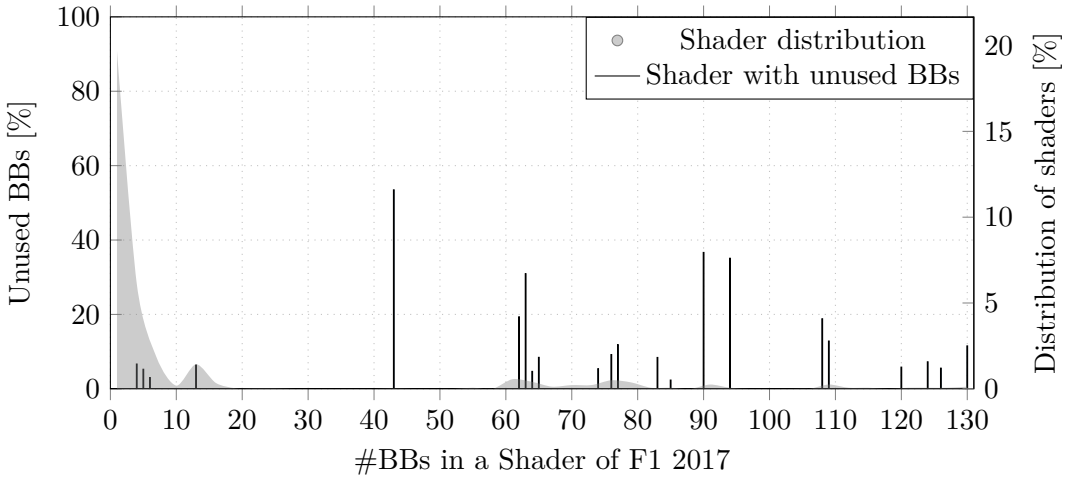


Figure 12: *The fraction of unused code by shader size in F1 2017*

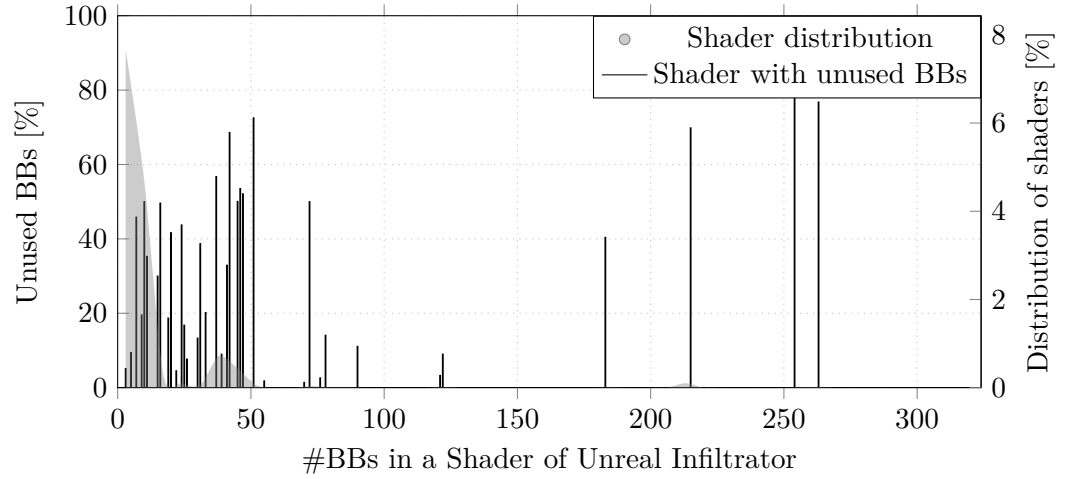


Figure 13: *The fraction of unused code by shader size in Unreal Infiltrator*

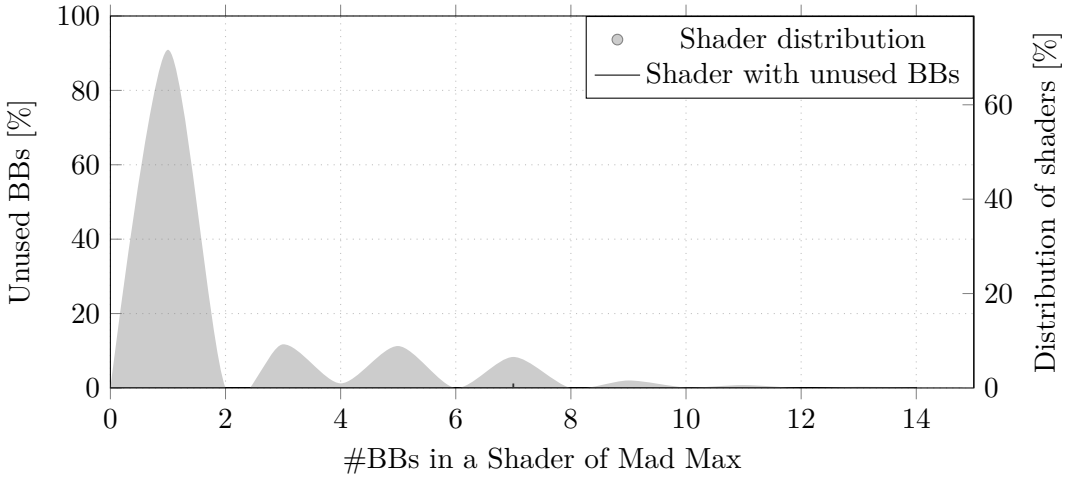


Figure 14: *The fraction of unused code by shader size in Mad Max*

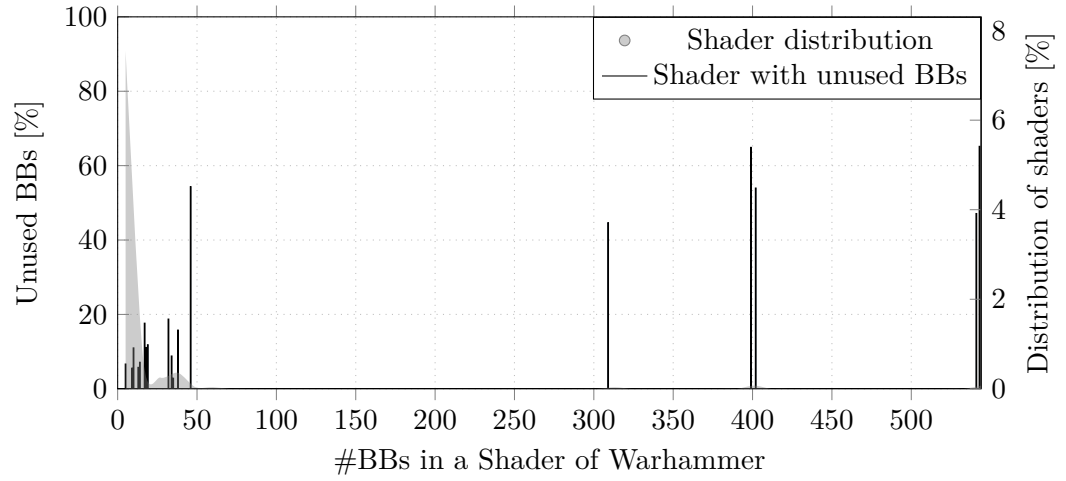


Figure 15: *The fraction of unused code by shader size in Warhammer*

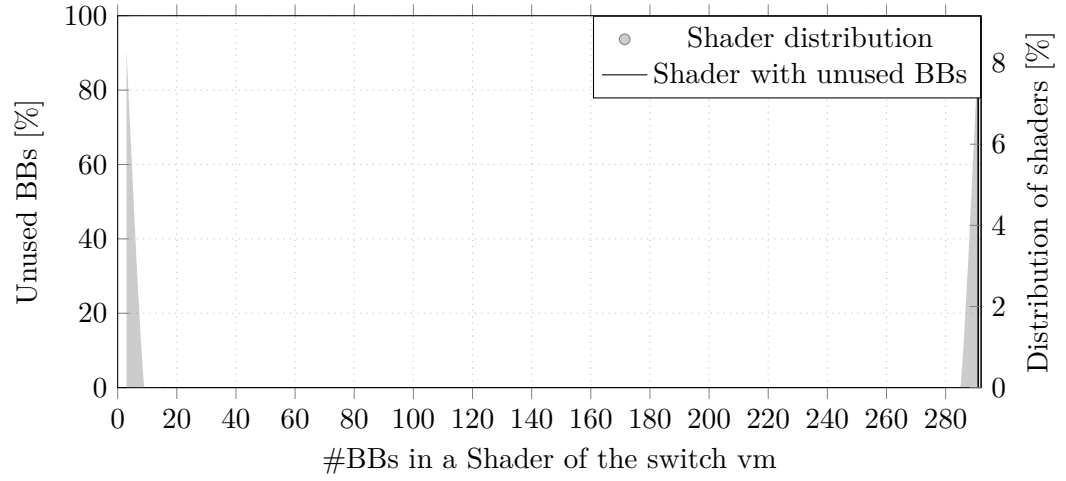


Figure 16: *The fraction of unused code by shader size in the switch vm*

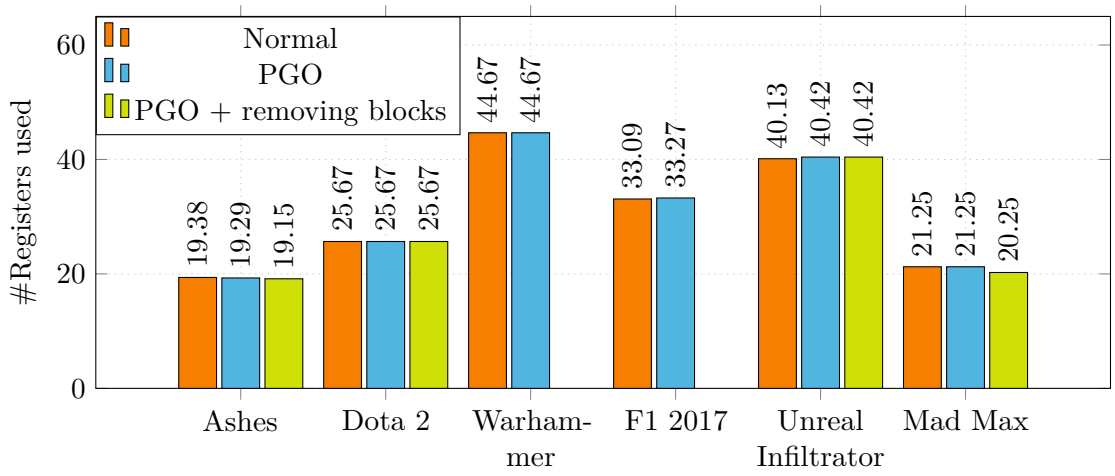


Figure 17: *Average vector register usage in compute shaders*

contains the code for one virtual instruction and the shader iterates through every instruction. The bytecode in our benchmark runs uses only a single instruction out of the 71 possible instructions. The code is transferred to the shader through a uniform buffer, consists of 128 instructions stored in 4 KiB of memory and is executed two times until the result is used as the final fragment color. LLVM compiles the switch statement into a binary jump tree, therefore in each iteration of the virtual machine, multiple jumps are executed until the right case for the current instruction is found.

Using PGO, the compiler detects that the instructions in all but one case statements are never executed. Thus, in the optimized version all these basic blocks are gone. This leaves us with the compare instructions from the original binary jump tree but the branch instructions are removed and the code has fewer basic blocks than before. This improves the performance as shown in section 5.5.

Apart from performance, we also want to analyze the register usage of shaders when removing unused blocks. This helps us to estimate gains of register spilling to reduce register pressure. When looking at registers, we have to differentiate between scalar and vector registers. Scalar registers are cheap in hardware compared to vector registers because only $\frac{1}{64}$ of them is needed. Hence, the hardware has more of them and the occupancy of shaders is mostly limited by vector registers. We also split our observations by shader type, we look at compute, vertex and pixel/fragment shaders separately. Each game is tested in three configurations: Running the game without any special options, using profile-guided optimizations and using PGO with removing unused basic blocks. The figures starting at fig. 17 show the register usage, averaged over all shaders that fall into the category, e.g. vector registers in compute shaders in the first diagram. The register usage for our switch vm is not shown because it stays unchanged, only one vector register less is used and the occupancy does not change.

The average register usage stays the same when using PGO. Ashes sometimes uses a bit fewer registers than normal, F1 2017 sometimes more but the differences are small. Ashes and Mad Max show the greatest effect when removing unused code, removing unused code sometimes reduces the register usage by one register averaged over all shaders. A more detailed analysis of the effect of removing unused blocks in fig. 23 shows the register amount for each shader before and after removing blocks. Every dot represents a shader, the x-coordinate is the number of used registers when compiling the shader with PGO, the

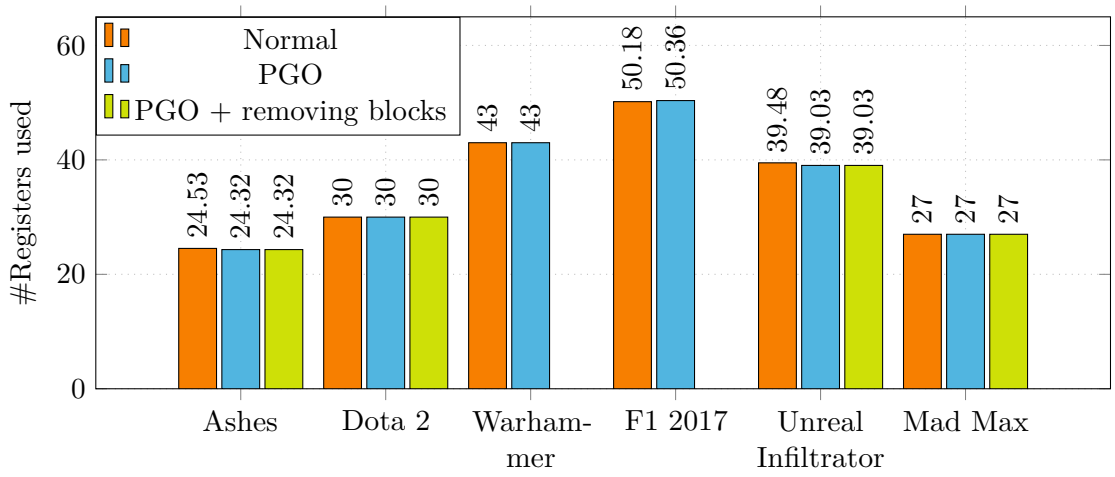


Figure 18: *Average scalar register usage in compute shaders*

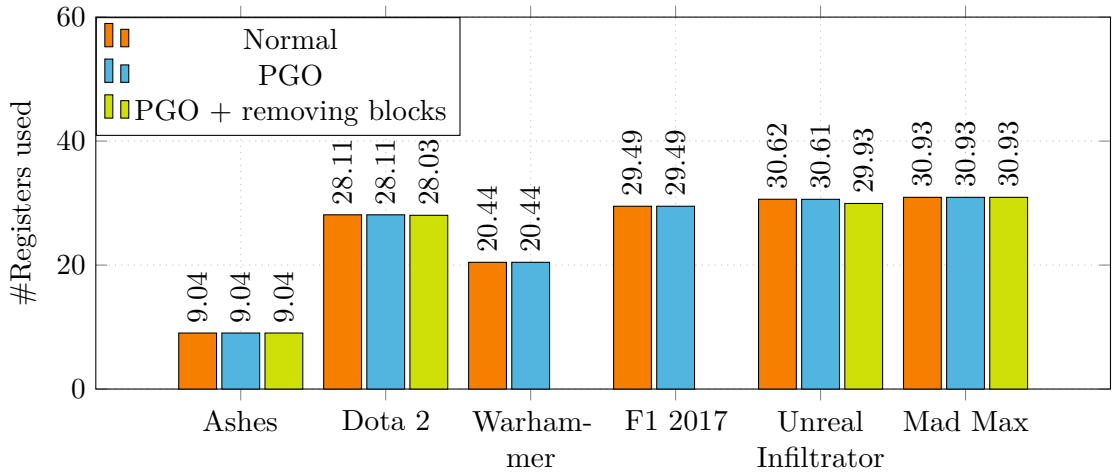


Figure 19: *Average vector register usage in vertex shaders*

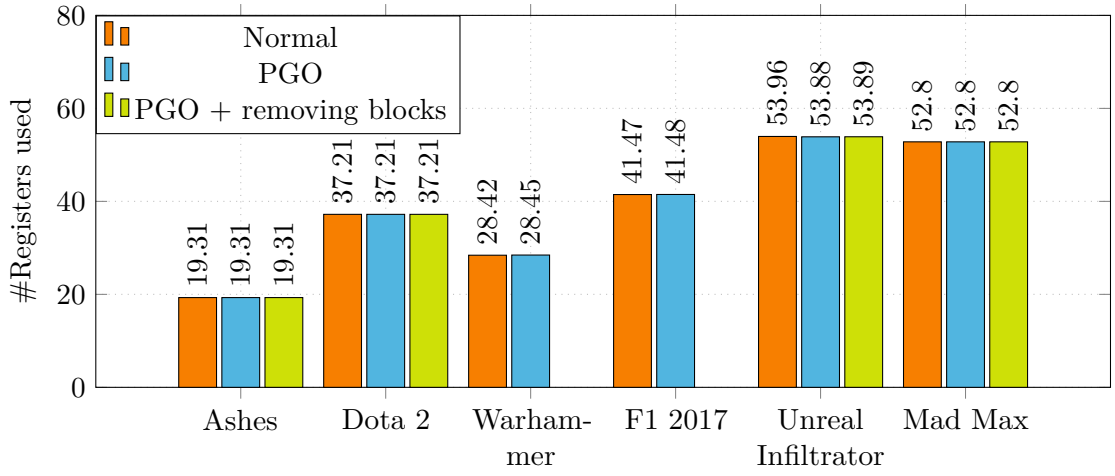


Figure 20: *Average scalar register usage in vertex shaders*

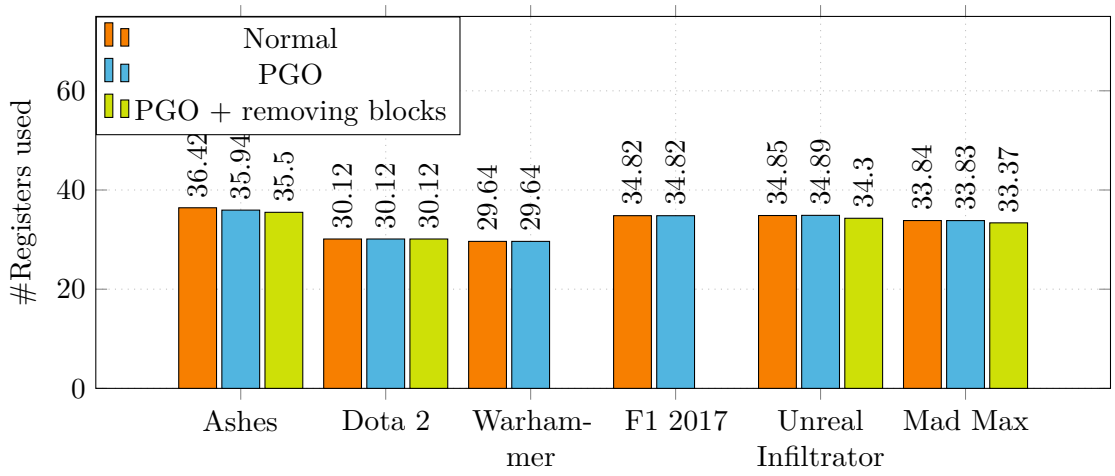


Figure 21: *Average vector register usage in pixel shaders*

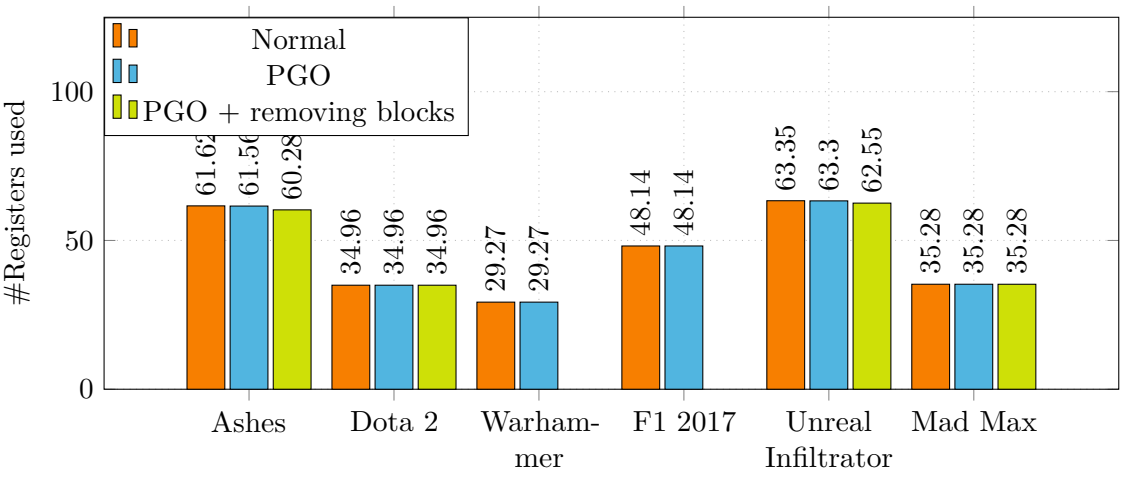


Figure 22: *Average scalar register usage in pixel shaders*

y-coordinate are the registers when compiling with PGO and additionally removing blocks which never get executed. Most shaders lie on the diagonal, which means the number of registers does not change. A few points are above the diagonal, so the compiler allocates more registers for them when unused code is removed. Some points are also below the diagonal, meaning they need fewer registers when blocks are removed.

In one case, Mad Max uses a pixel shader with a lot of computations, but these computations never get executed in the benchmarks. Thus, when removing unused basic blocks, only a simple shader that checks an always false condition remains and fewer vector registers are used. This shader can be seen in the lower right corner of fig. 23. Surprisingly, Dota 2 does not show a great effect when removing code, although nearly 20 % of the basic blocks are removed.

The important metric that builds upon the number of used registers is the occupancy. If fewer registers are used, the occupancy gets higher and the GPU can hide memory latency better. The optimal occupancy is circa at 10 parallel shader executions per SIMD unit [1], so we focus on shaders with low occupancy. The amdgpu register allocator in LLVM optimizes for occupancy, which means that no shader should get worse occupancy

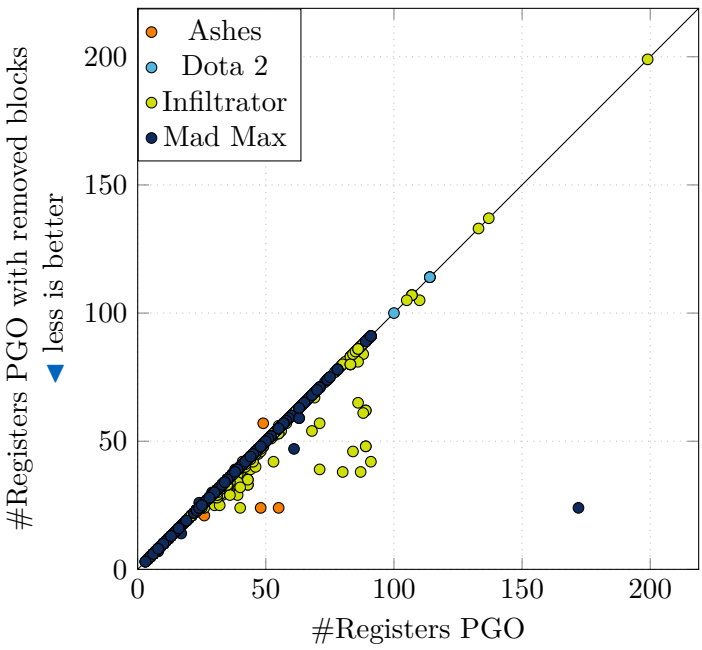


Figure 23: *Vector register usage comparison when removing blocks, each dot represents a shader*

when applying more optimizations. The occupancies of shaders in Ashes, Dota 2, Unreal Infiltrator and Mad Max are displayed in fig. 24. In this diagram, the occupancy is based only on vector registers and ignores scalar registers. On some hardware, scalar registers can also limit the occupancy but this depends on the hardware and is only seldom the case, thus we ignore scalar registers here. Most shaders are on the diagonal, so the occupancy does not change. Some are on the upper-left half, which means that removing unused basic blocks increases the occupancy. E.g. instead of four shaders, five shaders can be run in parallel. Finding shaders where unused code limits the occupancy is exactly the goal of this test. For a single shader, the compiler allocates more registers so that the occupancy is reduced.

In summary, removing unused code can greatly reduce the register usage for some shaders. Unused code, in general, does not seem to be a problem for games, at least not from the perspective of register usage.

5.3 Uniform Branches

Apart from basic block counters, we examined the uniformity of certain variables. One type of variables that we further looked at is branch conditions. If a condition is uniform, the branch which is guarded by this condition is taken uniformly by the SIMD unit, if the condition is divergent, the unit has to take both branches after each other.

At compile time, a variable can be classified as uniform or as divergent. If it is classified as uniform, the compiler knows that under all possible circumstances, the variable always contains the same value across all lanes. Usually, such variables are stored in scalar registers and computations on these variables use the scalar unit. If a variable is classified as divergent, the compiler is not able to prove that the variable always contains the same value on all lanes. It does not necessarily mean that the variable value differs on multiple lanes at runtime. Our instrumentation records the behavior at runtime and classifies a

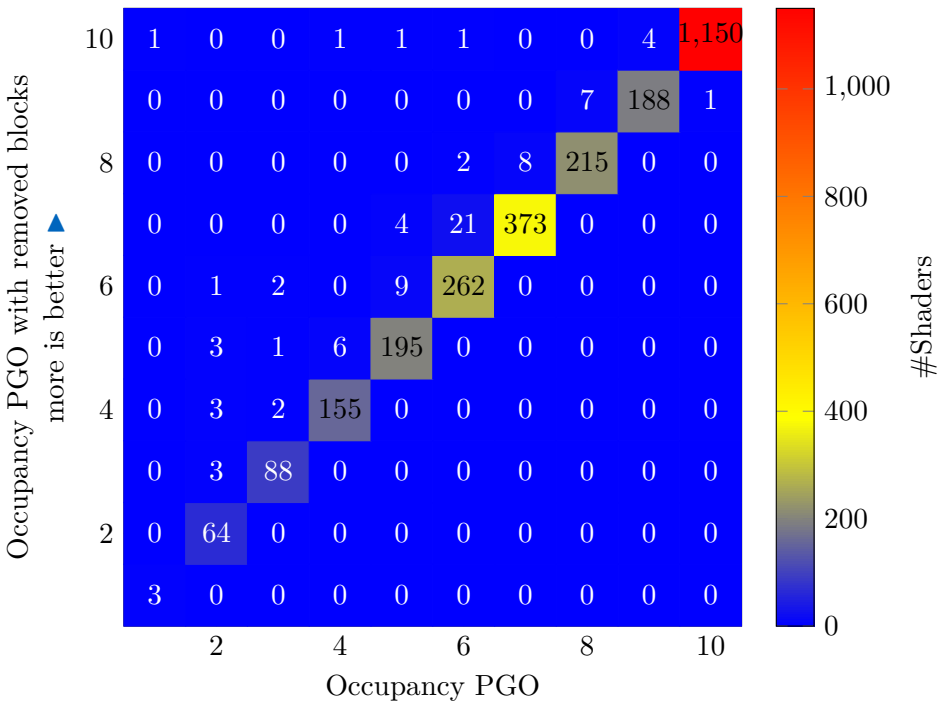


Figure 24: *Occupancy based on vector register usage, when removing blocks*

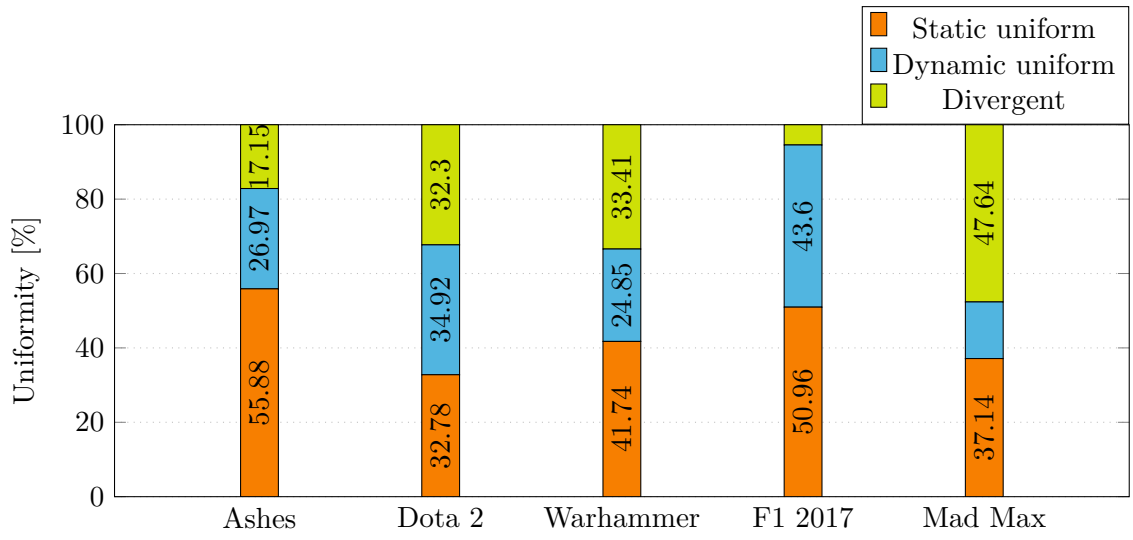


Figure 25: *Uniformity of branches*

variable as *dynamic uniform* if it always contains the same value on all lanes or as divergent if not. *Static uniform* variables are the ones where the compiler can prove that they are uniform.

Figure 25 shows how conditions and branches are divided into the three categories. The "best" category are static uniform branches. For these branches, the compiler can emit a simple branch instruction and the compiler knows that either all or none of the active lanes will take this branch. Conditions in the dynamic uniform category represent a branch where the compiler emitted additional code to support divergent branching. At runtime,

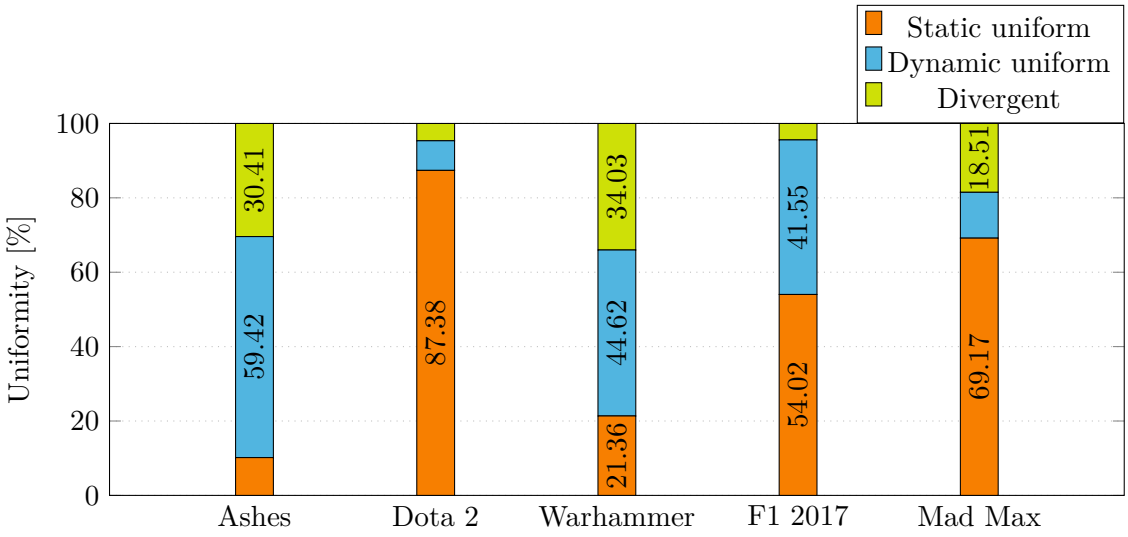


Figure 26: *Uniformity of load instructions*

this code is not used, the branch is always taken by all lanes uniformly. The performance wise worst case are divergent branches. In these cases, the compiler emits additional code like in the dynamic uniform case and at runtime, both branches are taken by different lanes, so the SIMD unit needs to execute both parts sequentially.

With the grading of the last paragraph, Ashes and F1 2017 are performing best because most of their branches are statically uniform and only a small fraction is divergent. We need to take into account though that this is a very coarse look at these games, we do not know the size and performance impact of the divergent branches, also sometimes divergent branches are inevitable.

In general, static uniform branches form the biggest group, followed by dynamic uniform and divergent branches. It depends on the game if the statically divergent branches are mostly uniform or divergent at runtime. In F1 2017 most are uniform and the minority is divergent while in Mad Max it is the other way around.

5.4 Uniform Loads

Similar to the analysis of uniform conditions in the last section, we analyzed `LoadInstr` instructions in LLVM. The uniformity of loaded memory values is shown in fig. 26. It is important to know that the analyzed load instructions do not include buffer loads and image loads which are more complicated to analyze because they are indexed with multiple dimensions, can return multiple dimensions, are connected with a sampler, use mipmapping, etc.

For comparison, fig. 27 shows the uniformity of addresses used in load instructions. This yields largely the same results as the uniformity of loaded values, sometimes it differs a bit. For example, Ashes has more divergent addresses than load values. This means that some loads with different addresses return a uniform value. There are cases, where using the same address returns different values across lanes. This happens when the address lies in scratch memory, which is used e.g. for spilled registers.

From the instrumented loads, divergent loads are the smallest category, staying under 35 % in every game. For Ashes and Warhammer, dynamic uniform loads are the common

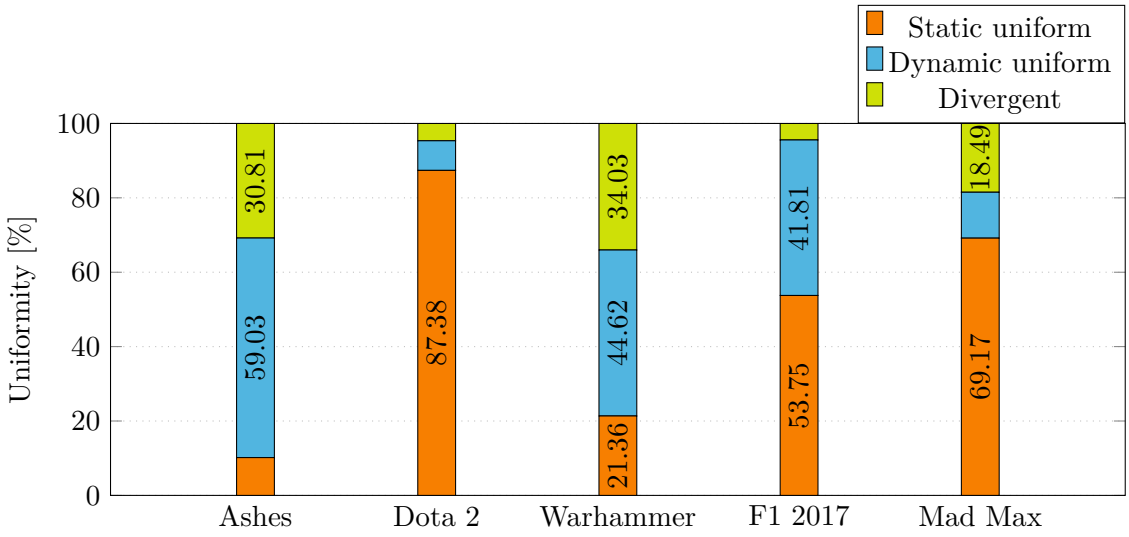


Figure 27: *Uniformity of addresses in load instructions*

case. For the rest of the games, static uniform loads make up more than half of the loads, leading to different distributions of load uniformity in every game.

For this work, we analyzed the uniformity of branches and loads, but we did not measure the performance changes when dynamic uniform values are known to the compiler. Using this information for further optimizations is left for future work.

5.5 Performance

In addition to the previous analyses, an important part is the performance change when using profile-guided optimizations. To evaluate the performance of shaders, we run the games in three different configurations: Without any special options, using profile-guided optimizations and using PGO with removing unused basic blocks. As before, removing blocks is only done for Ashes, Dota 2, Mad Max and the switch vm, the other games are not working with the current implementation. All games were run three times in each configuration.

The result is displayed in fig. 28 and table 1. In general, the optimizations did not change the performance of games by a great amount. On Ashes and F1 2017 it even had a negative effect, these games got a little slower. Dota and Mad Max did not show a significant difference. Warhammer reduced the time per frame from (65.98 ± 0.18) ms to (65.31 ± 0.13) ms with PGO, which is an improvement of (1.0 ± 0.4) %. The switch vm shows big improvements. When using PGO, it gets faster by (7.34 ± 0.10) %. When additionally removing unused blocks, it gets even faster by (21.06 ± 0.10) %.

The order of basic blocks and other optimizations based on basic block counter seem to have a minor effect on the performance of games. Removing unused blocks did not affect the tested games. On the switch vm however, PGO as well as removing basic blocks had a remarkable effect.

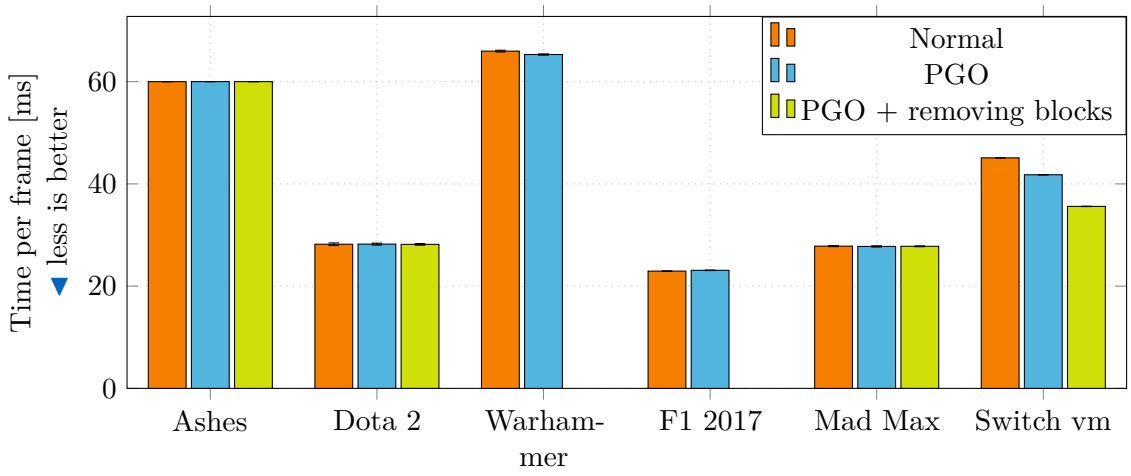


Figure 28: *Performance of games with PGO*

Game	Config	Time per frame	Difference
Ashes	Normal	(60.0034 ± 0.0022) ms	
	PGO	(60.0118 ± 0.0019) ms	(0.014 ± 0.005) %
	PGO + removing blocks	(60.006 ± 0.006) ms	(0.004 ± 0.010) %
Dota 2	Normal	(28.20 ± 0.26) ms	
	PGO	(28.22 ± 0.19) ms	(0.1 ± 1.2) %
	PGO + removing blocks	(28.17 ± 0.17) ms	(-0.1 ± 1.1) %
Warhammer	Normal	(65.98 ± 0.18) ms	
	PGO	(65.31 ± 0.13) ms	(-1.0 ± 0.4) %
F1 2017	Normal	(22.94 ± 0.05) ms	
	PGO	(23.10 ± 0.05) ms	(0.68 ± 0.29) %
Mad Max	Normal	(27.82 ± 0.11) ms	
	PGO	(27.77 ± 0.12) ms	(-0.2 ± 0.6) %
	PGO + removing blocks	(27.79 ± 0.09) ms	(-0.1 ± 0.5) %
Switch vm	Normal	(45.10 ± 0.04) ms	
	PGO	(41.79 ± 0.04) ms	(-7.34 ± 0.10) %
	PGO + removing blocks	(35.60 ± 0.04) ms	(-21.06 ± 0.10) %

Table 1: *Performance of games with PGO*

5.6 Overhead

To find hot paths, the instrumentation inserts counters into some basic blocks of a program. The counters introduce an overhead, compared to a non-instrumented version of the code. In the case of counting the frequency of basic block executions, the counters themselves are not sensitive to timing and thus not directly influenced by this overhead. But the code will run slower. In the case of a game, there will be fewer frames per second than usual. If the benchmark, that is run with PGO instrumentation, runs for a fixed time, the basic block frequencies will be lower. Therefore, the measured counters may show fewer executions than the actual frequencies that should be obtained. This effect should not have an impact on the optimization results because the ratio of frequencies stays the same, no matter how fast or slow a game runs.

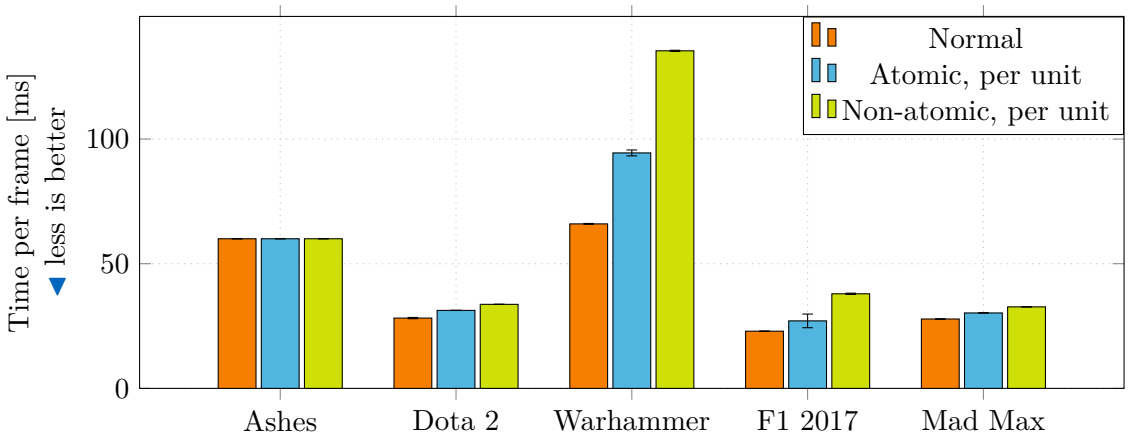


Figure 29: *Overhead of BB counters*

Game	Config	Time per frame	Overhead
Ashes	Normal	(60.0034 ± 0.0022) ms	
	Atomic, per unit	(60.010 ± 0.008) ms	(0.010 ± 0.013) %
	Non-atomic, per unit	(60.0080 ± 0.0022) ms	(0.008 ± 0.006) %
Dota 2	Normal	(28.20 ± 0.26) ms	
	Atomic, per unit	(31.28 ± 0.05) ms	(10.9 ± 1.1) %
	Non-atomic, per unit	(33.71 ± 0.06) ms	(19.5 ± 1.1) %
Warhammer	Normal	(65.98 ± 0.18) ms	
	Atomic, per unit	(94.4 ± 1.2) ms	(43.1 ± 1.9) %
	Non-atomic, per unit	(135.39 ± 0.23) ms	(105.2 ± 0.7) %
F1 2017	Normal	(22.94 ± 0.05) ms	
	Atomic, per unit	(27.1 ± 2.8) ms	(18 ± 12) %
	Non-atomic, per unit	(37.94 ± 0.26) ms	(65.4 ± 1.2) %
Mad Max	Normal	(27.82 ± 0.11) ms	
	Atomic, per unit	(30.25 ± 0.06) ms	(8.7 ± 0.5) %
	Non-atomic, per unit	(32.69 ± 0.06) ms	(17.5 ± 0.5) %

Table 2: *Overhead of BB counters*

We measured the overhead with two configurations. The baseline in fig. 29 and table 2 is a normal run of the game with no instrumentation. The first tested configuration uses atomic add instructions to increment the counters. The second variant uses normal add instructions, i.e. no atomics. All instrumentation was inserted after structurizing the CFG and increments the counter by one for the whole unit.

The benchmarks show that an atomic add causes less overhead than the non-atomic version. The reason is that the non-atomic increment needs a load and a store operation where the atomic version only needs a single memory transaction. The actual addition is computed inside the L2 cache for atomics. Similar to the PGO benchmarks, Ashes shows the smallest difference. The overhead of atomic counter instrumentation ranges from (0.010 ± 0.013) % for Ashes to (43.1 ± 1.9) % for Warhammer. The average overhead for the five games is at 16 %, the average for the non-atomic version at 42 %.

In some cases, it may be necessary to decrease the overhead of basic block counters. For example to use the instrumentation in production environments where high overheads are

not acceptable. Sometimes, the overhead is high enough to trigger timeouts in the GPU driver, leading to game crashes. This is the case for the Infiltrator demo project of the Unreal Engine when using atomic counters with one atomic operation per lane.

A simple way to decrease the overhead is to aggregate the counter addition to be executed once per SIMD unit instead of once per lane. I.e. for per unit counting, all lanes but one are turned off when the counter is incremented. For per lane counting, the same happens but the number of previously active lanes is added to the counter. This is enough to let e.g. the Infiltrator demo start. All our benchmarks used this technique.

Another overhead reduction can be achieved by skipping counting on most SIMD units and activate it e.g. only on 5 % of the units. This means most executions will skip the increments, having less overhead while the rest of the executions give accurate statistics. A problem with this approach is that (e.g. vertex-) shaders that get executed only a few times, might not land on any of SIMD units where counting is activated and no statistics at all are collected for them. This should only be a small problem because these shaders probably do not account for much of the computation time (as they are executed only a few times) and optimizing them cannot yield big benefits anyway.

A more sophisticated technique that can speed up basic block counting tries to reduce the memory pressure by atomics. As we know, many atomic operations will simultaneously try to access the same memory location. We can reduce this pressure if multiple memory locations in different cache lanes are used for the same counter and they spread the counting over these locations. As fewer atomic increments access the same memory, they do not have to wait as long as before and the program execution can continue faster. In the end, when shutting down the application, the driver has to add up all duplicated counters. This happens only once in the end for most applications so it adds not much overhead.

6 Discussion

Before we describe related work and opportunities for future research, we discuss the advantages and disadvantages of instrumenting before or after CFG structurization. To conclude this thesis, we summarize our work and results.

6.1 Instrumentation before and after Structurizing the CFG

The `StructurizeCFG` pass in the LLVM compiler pipeline is a turning point. Before, the perspective is one of a single thread (or lane). After this pass, the control-flow graph contains the per SIMD unit view, where an if-else was before, both branches can now be executed after each other. It is important to note that the per unit perspective only applies to the CFG after this pass. All instructions apart from branch instructions stay the same and exhibit the per lane view.

For profile-guided optimizations, the compiler can instrument the CFG before or after structurization. In the case of basic block counters, this will lead to different results. Before structurization, it is impossible to run through both branches of an if-else construct. After the CFG is structurized, it is still impossible *per lane*, but it is possible from the *per unit* perspective. If executions are counted per lane, instrumenting before or after structurization does not make a semantic difference. After `StructurizeCFG`, more counters are needed, because there are more basic blocks and LLVM does not know anymore that only one block of an if-else construct will be entered. On the other hand, if executions are counted per unit, it does make a difference. More importantly, counting executions per unit before the CFG gets structurized leads to wrong results as LLVM expects that an else-block will not be entered if the if-block is executed. And this assumption is false for the per unit point of view.

Summarizing, there are two different possibilities for instrumentation. First, counting per lane, where the instrumentation should be done before structurization because it leads to fewer counters and makes no difference otherwise. And second, counting per SIMD unit after structurization. These two opportunities represent the per lane and per unit view on a GPU. The decision for the instrumentation is the same as for actually using collected data. If counters are collected before the `StructurizeCFG` pass, the compiler must annotate the results at the same point in the pipeline. LLVM will propagate them when transforming the CFG in the structurization. The same applies to counting after structurization: The compiler must also annotate the counters after structurization. This

means the collected data is only available to the compiler after the `StructurizeCFG` pass. Looking at CPUs, LLVM applies instrumentation and usage of counters at the end of the pipeline, after most optimizations. This makes sense because the counters are used for linearization and getting data about an unoptimized CFG is not useful if the final CFG which needs to be linearized looks completely different.

The advantage of a per lane instrumentation is simplicity. The `PassManagerBuilder` can insert the needed passes easily (it runs before structurization) and we do not need to adjust any instrumentation code because the LLVM IR exhibits the per lane view.

To use per unit instrumentation, we need to add passes manually after structurization and cannot use the `PassManagerBuilder`. Additionally, the PGO instrumentation pass needs to be changed such that the increment instruction is only run on one lane and not on all lanes. An advantage of the per unit view is that it is more similar to the code that gets executed on the hardware in the end. For example, compare two basic blocks: One always gets run by the whole unit, for the other one only a single lane is active, e.g. to aggregate a result over the SIMD unit. When using per lane counting, the first basic block will be weighted a lot more important than the second one. Per unit counting provides a more accurate view, both blocks are executed the same amount of times, so they are equally important. The amount of active lanes does not matter.

6.2 Related Work

This work focuses on the profiling of shaders to get information for optimizations. There are many GPU profiling tools [33] that can provide information to optimize a graphics application. Some of them are integrated into a development environment [29] or part of a game engine that allows writing custom shaders [36, 37]. Others again are developed by hardware manufacturers [31, 32, 3].

These tools can show performance statistics like occupancy, used memory bandwidth and compute power. They also show the time that is needed per shader or draw call. But there are only a few tools which can show detailed information that looks inside shaders. The information which cannot be collected by most tools is e.g. how often each instruction is executed or how much time is spent for single instructions.

One of the tools that can collect metrics like counters is GPUOcelot [16, 19]. It can trace single instructions in shaders when emulating CUDA applications. Pyramid is another emulator and shader analyzer for various architectures, including the AMD GCN architecture that we examined in detail in this work [5]. While emulation can provide accurate information, an emulated application will run a lot slower as the CPU has less computational power. This makes it harder to find a suited benchmark for collecting runtime information. A developer always has to take into account that it will take a multiple of the time to emulate the benchmark on the CPU than to run it on the GPU. The collected information, however, have high quality and once an emulator is working it is easy to collect all kinds of information.

Tools that collect detailed information about the execution of shaders without emulation exist for example for Arm GPUs [6]. In the high-performance computing space, Lynx [14] is used to instrument CUDA applications and automatically measure performance statistics [13]. An interesting work by Stephenson et al. [35] makes it possible to insert instrumentation at any point into shader code. They use this to insert counters, analyze memory access patterns and even for value profiling. The instrumentation part is similar to this work, a difference is that their work did not include using this information

for automatic compiler optimizations.

The GPU space developed a few sets of standard benchmarks that can be used to create performance comparisons. For example Parboil, Rodinia, SHOC and Tensor are sets of benchmarks that are regularly used. In this work, we focused on the AMDVLK Vulkan driver for AMD GPUs but these benchmarks are not yet usable for Vulkan. Thus we are not able to use them for performance tests. There is also a Vulkan benchmark set called VComputeBench, which claims to be a general, publicly available benchmark for GPGPU [26]. However, we were unable to find these benchmarks online and the author did not respond to our questions. Therefore, we cannot include it in our benchmarks. The vkmark Vulkan benchmarks [2] are available publicly though this tool does not focus on shaders but the rest of the graphics pipeline. Therefore, the shaders are simple and do not provide possibilities for PGO. E.g. using the default LLVM PGO on the shaders did not change a single instruction. Thus, we decided to not include it in our benchmark list.

6.3 Future Work

This work does provide a good step into PGO on GPUs but it does not exhaustively implement and evaluate PGO. This section will explain possible directions for future work in this field.

When analyzing uniformity, the divergence analysis provides information about uniform or divergent variables. As mentioned before, this analysis does not yield perfectly accurate results at all steps in the compilation pipeline. An opportunity for future work is to look into these problems, fix them where possible and find the best point to run this analysis.

The uniformity analysis does only analyze conditions and values from memory at the moment. It would be interesting to expand this to image and buffer loads. They make up a considerable part of the memory accesses in a shader, but analyzing them is more difficult and did not fit into the time frame of this thesis.

In addition to analyzing if a variable is uniform or not, we are interested in statistics about how often a variable is uniform. If the divergent cases only make up a low fraction of the executions, it can still be beneficial to optimize for the common case. At the moment, we do not know how often a variable is uniform or not.

Based on the basic block counters, the compiler removed basic blocks that never get executed. This worked good for three games but did not work for others. As removing basic blocks was meant as a simple and dirty test, this is not too surprising. However, it would be nice to have this test working for more games and collect more information about the effects.

A related bug in LLVM is the artifacts that show up in Ashes when using PGO. We found one bug caused by PGO on GPUs in the `ControlHeightReduction` pass, we did not find the cause for the bug with Ashes though. Before PGO can be used in production environments, this bug should be fixed and more games should be tested with these optimizations.

Now, that collecting runtime data from shaders works, this data can be used in the compiler to improve the optimization of shaders. This includes many of the optimizations which we suggested in section 3.2. Hoisting buffer and texture loads out of conditional blocks, for example, is a promising candidate for rewarding optimizations. Also, the compiler is now able to collect data about the uniformity of variables but this information is not used for optimizations so far.

The existing optimizations were applied to five different games. The collected performance

data give an impression of what we can expect of PGO on graphics cards. There exist many more games with different engines and they all use the GPU in a slightly different way. Thus, it would be interesting to compare the performance of PGO of more games and other Vulkan applications.

6.4 Conclusion

To our knowledge, this is the first work which leverages profile-guided optimizations on GPUs. We started this thesis with an introduction to the topic of PGO and GPUs and continued with our design and implementation to enable profile-guided optimizations on graphics cards. Benchmarks of five different games and statistics about collected metrics give insights into potential performance gains and conclude the topic.

The profile-guided optimizations use the basic block counting functionality of LLVM. We adjusted the driver and parts of LLVM to get working counters on GPUs. These counters give information about unused basic blocks. We use this information to create statistics about the number of unused blocks and to remove unused basic blocks. This leads to three different configurations for running games. The first is to run them without any changes in the compiler pipeline, another one is to enable the standard PGO options of LLVM using basic block counters, and finally, the compiler can additionally remove code which is never executed.

For these three modes, we analyze the performance of five different games and a small test program. In the game Dota 2, nearly 20 % of the shader code is removed, in Warhammer even 37 %. The other games have a lower fraction of unused blocks. The performance of the games mostly stays the same, some games are running a little faster when switching on PGO, some are slower. Removing unused blocks has no measurable effect on the performance of the tested games. Our small test application benefits most from deleting unused code. Removing 92 % of the basic blocks leads to a speed improvement of (21.06 ± 0.10) %.

The overhead of instrumenting all shaders with basic block counters ranges from 0 % to 43 %, with an average of 16 % overhead.

An important property of shaders is the number of registers they use. The register count limits the number of shaders that can run in parallel on one SIMD unit and thus it is important for the overall performance. Looking at the register usage of games in the same three configurations shows that it is mostly unchanged, even if a large fraction of blocks is removed for some games. Mad Max contains a single shader, where removing unused code eliminates a large amount of code, leading to a lot lower register usage.

In addition to counting basic blocks, this work contributes an instrumentation for LLVM which measures the uniformity of single variables. One use case for this instrumentation is to test conditions. The uniformity of a condition tells us if the corresponding branch is taken uniformly or not. Averaging over all shaders and conditions, we find that for most games, the biggest category is static uniform branches. This means the compiler can prove that these branches are always taken uniformly. The other branches are either always uniform at runtime or divergent. The fractions of these two categories vary between games, there is no clear majority.

Apart from conditions, we reuse this instrumentation to analyze the values of memory loads. The fractions of the three categories for static uniform, dynamic uniform and divergent values vary a lot between games. Divergent loads form the smallest category with under 35 % for every game. For Ashes and Warhammer, dynamic uniform loads are the common case. For the rest of the games, static uniform loads make up more than half

of the loads.

When writing passes in LLVM, we have to decide where in the compilation pipeline this pass should be inserted. In the case of counting basic blocks, there are two main possibilities. The default position when activating PGO is after most optimizations but before the CFG gets structurized. An alternative position is after structurization. As the CFG after structurization is closer to the actual control-flow on the hardware, we conclude that the basic block counting instrumentation is better suited after structurization and that the counters should be incremented once per SIMD unit instead of counting all active lanes.

References

[1] Sebastian Aaltonen. *Optimizing GPU occupancy and resource usage with large thread groups*. Visited on 2019-08-28. May 2017. URL: https://gpuopen.com/optimizing-gpu-occupancy-resource-usage-large-thread-groups/.

[2] afrantzis. *vkmark*. Visited on 2019-09-04. URL: https://github.com/vkmark/vkmark.

[3] AMD. *GPU ShaderAnalyzer*. Visited on 2019-03-06. Apr. 2012. URL: https://gpuopen.com/archive/gpu-shaderanalyzer/.

[4] Denis Bakhvalov. *Machine code layout optimizations*. Visited on 2019-07-10. Mar. 2019. URL: https://easyperf.net/blog/2019/03/27/Machine-code-layout-optimizatoins.

[5] Joshua Barczak. *Pyramid*. Visited on 2019-07-02. URL: https://github.com/jbarczak/Pyramid.

[6] Stephen Barton. *Analyzing Performance of Mobile Games*. Apr. 2013. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/graphics-and-multimedia/Analyzing-Performance-of-Mobile-Games.pdf.

[7] Eli Bendersky. *Load-time relocation of shared libraries*. Visited on 2019-04-21. Aug. 2011. URL: https://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries/.

[8] Eli Bendersky. *Position Independent Code (PIC) in shared libraries*. Visited on 2019-04-21. Nov. 2011. URL: https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/.

[9] Brad Calder, Peter Feller, and Alan Eustace. "Value Profiling". In: *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE. 1997, pp. 259–269. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.1771&rep=rep1&type=pdf.

[10] Matthäus Chajdas. *Introduction to compute shaders*. Visited on 2019-03-01. 2018. URL: https://anteru.net/blog/2018/intro-to-compute-shaders/.

[11] Zhongliang Chen. *Scalar-Vector GPU Architectures*. 2016.

[12] Clang. *Clang Compiler User's Manual*. Visited on 2019-03-05. URL: https://clang.llvm.org/docs/UsersManual.html.

[13] Naila Farooqui et al. "Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications". In: *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*. 2014.

[14] Naila Farooqui et al. *Lynx: A Dynamic Instrumentation System for GPGPU Computing*. Visited on 2019-07-02. Feb. 2013. URL: http://casl.gatech.edu/research/lynx-a-dynamic-instrumentation-system-for-gpgpu-computing/.

[15] Epic Games. *Infiltrator Demo*. Visited on 2019-09-02. Aug. 2015. URL: https://www.unrealengine.com/marketplace/en-US/slug/infiltrator-demo.

[16] Georgia Institute of Technology. *GPU Ocelot*. Visited on 2019-07-02. URL: http://gpuocelot.gatech.edu/.

[17] Baldur Karlsson. *RenderDoc*. Visited on 2019-03-06. URL: https://github.com/baldurk/renderdoc.

[18] Donald E Knuth and Francis R Stevenson. “Optimal Measurement Points for Program Frequency Counts”. In: *BIT Numerical Mathematics* 13.3 (1973), pp. 313–322.

[19] Nagesh B Lakshminarayana and Hyesoon Kim. “Effect of Instruction Fetch and Memory Scheduling on GPU Performance”. In: *Workshop on Language, Compiler, and Architecture Support for GPGPU*. Vol. 88. Citeseer. 2010.

[20] Michael Larabel. *A Fresh Look At The PGO Performance With GCC 8*. Visited on 2019-03-27. July 2018. URL: https://www.phoronix.com/vr.php?view=26589.

[21] Michael Larabel. *Phoronix Test Suite*. Visited on 2019-09-13. URL: https://www.phoronix-test-suite.com.

[22] Tor Lillqvist. *The Future of OpenCL in LibreOffice*. 2016. URL: https://libocon.org/assets/Conference/Brno/libocon-2016-opencl.pdf.

[23] Linux. *Perf profiler*. Visited on 2019-03-06. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[24] LLVM. *Basic Block Linearization*. Visitied on 2019-03-13. URL: https://github.com/llvm/llvm-project/blob/192df587d1996c53060bfa9f63514af445de3cca/llvm/lib/CodeGen/MachineBlockPlacement.cpp#L1321.

[25] LLVM. *The LLVM Target-Independent Code Generator*. Visited on 2019-03-27. Mar. 2019. URL: https://llvm.org/docs/CodeGenerator.html.

[26] Nadjib Mammeri and Ben Juurlink. “VComputeBench: A Vulkan Benchmark Suite for GPGPU on Mobile and Embedded GPUs”. In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 25–35.

[27] Chris McClanahan. *History and Evolution of GPU Architecture*. 2010. URL: https://pdfs.semanticscholar.org/2479/80e834f1c8f684d85067402f950930e6af91.pdf.

[28] Sina Meraji and Berni Schiefer. *DB2 BLU w/GPU Demo – Concurrent execution of an analytical workload on a POWER8 server with K40 GPUs*. Mar. 2015. URL: https://openpowerfoundation.org/db2-blu-wgpu-demo-concurrent-execution-of-an-analytical-workload-on-a-power8-server-with-k40-gpus/.

[29] Microsoft. *GPU usage*. Visited on 2019-07-02. URL: https://docs.microsoft.com/en-us/visualstudio/profiling/gpu-usage?view=vs-2019.

[30] Microsoft. *Profile-Guided Optimizations*. Visited on 2019-03-051. Mar. 2018. URL: https://docs.microsoft.com/en-us/cpp/build/profile-guided-optimizations?view=vs-2019.

[31] NVIDIA. *Nsight Graphics*. Visited on 2019-07-02. URL: https://developer.nvidia.com/nsight-graphics.

[32] NVIDIA. *ShaderPerf*. Visited on 2019-03-06. URL: https://developer.nvidia.com/nvidia-shaderperf.

[33] PGI. *PGI Profiler User Guide*. 2014. URL: https://www.pgroup.com/doc/pgprof14ug.pdf.

[34] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Übersetzerbau 3: Analyse und Transformation*. Springer, 2010. DOI: 10.1007/978-3-642-03331-5.

[35] Mark Stephenson et al. “Flexible Software Profiling of GPU Architectures”. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 185–197.

[36] Unity. *GPU Profiler*. Visited on 2019-07-02. URL: https://docs.unity3d.com/Manual/ProfilerGPU.html.

[37] Unreal. *GPU Profiling*. Visited on 2019-07-02. URL: https://docs.unrealengine.com/en-US/Engine/Performance/GPU/index.html.

[38] Hiroshi Yamauchi. *Control Height Reduction*. Visited on 2019-06-26. Aug. 2018. URL: https://reviews.llvm.org/D50591?id=162410.

Acronyms

API application programming interface.

BB basic block.

CFG control-flow graph.

CPU central processing unit.

CU Compute Unit.

ELF executable and linkable format.

GCN Graphics Core Next.

GLSL OpenGL shading language.

GOT global object table.

GPGPU General Purpose GPU.

GPU graphics processing unit.

IR intermediate representation.

ISA Instruction Set Architecture.

JIT just-in-time compiler.

KMD kernel-mode driver.

LDS local data share.

LLPC llvm pipeline compiler.

PAL Platform Abstraction Library.

PCIe bus Peripheral Component Interconnect Express bus.

PGO profile-guided optimizations.

PIC Position Independent Code.

RDNA Radeon DNA.

SIMD single instruction, multiple data.

SIMT single instruction, multiple threads.

SMT simultaneous multithreading.