A Cross-platform Evaluation of Graphics Shader Compiler Optimization

A Cross-platform Evaluation of Graphics Shader Compiler Optimization

Lewis Crawford, Michael O'Boyle
School of Informatics, University of Edinburgh, UK

***Abstract*—For real-time graphics applications such as games and virtual reality, performance is crucial to provide a smooth user experience. Central to this is the performance of shader programs which render images on the GPU. The rise of low-level graphics APIs such as Vulkan means compilation tools play an increasingly important role in the graphics ecosystem. However, despite the importance of graphics, there is little published work on the impact of compiler optimization.**

This paper explores common features of graphics shaders, and examines the impact and applicability of common optimizations such as loop unrolling, and arithmetic reassociation. Combinations of optimizations are evaluated via exhaustive search across a wide set of shaders from the GFXBench 4.0 benchmark suite. Their impact is assessed across three desktop and two mobile GPUs from different vendors. We show that compiler optimization can have significant positive and negative impacts which vary across optimisations, benchmarks and platforms.

I. Introduction

High quality graphics are essential in modern user interfaces. A key issue is device performance which is critical to graphics quality especially in real-time applications such as computer games or augmented reality [1] [2]. Performance affects not only the visual fidelity of a game, but also how responsive it feels. To satisfy the ever-increasing demands of real-time graphics requires hardware acceleration using highly parallel graphics processing units (GPUs).

Graphics is an extremely important area of computing; the computer graphics market is projected to be worth $215 billion by 2024 [3]. Despite the importance of graphics, there are relatively few papers examining the impact of computer architecture or the system stack on performance. While there are many papers on modifying architecture or improving compiler optimization for compute workloads, the majority of graphics papers focus on algorithmic issues. This paper examines the impact of compiler optimizations on graphics workloads and shows they can significantly impact performance.

Modern computer games are large and complex pieces of software. Achieving good performance on any particular target platform requires expert knowledge. Games rely not only on the software written by a developer, but also content from 2D or 3D artists. They also depend on GPUs and drivers from different hardware vendors and third-party game-engines. The overall software stack is shown in Figure 1. This complex stack of interconnected software and hardware makes achieving good performance on one platform without negatively affecting any others, a challenging task.

The GPU is programmed using small graphics programs called "shaders", which run in parallel as stages of a programmable rendering pipeline (see Figure 2). The most computationally intensive part is often the fragment shader, which determines the RGBA colour of each pixel inside a given triangle. Little academic work has been done on optimizing shaders. This is partly due in part to the way that most of the graphics ecosystem is structured, which requires any transformations on shaders to occur at a source-to-source level.

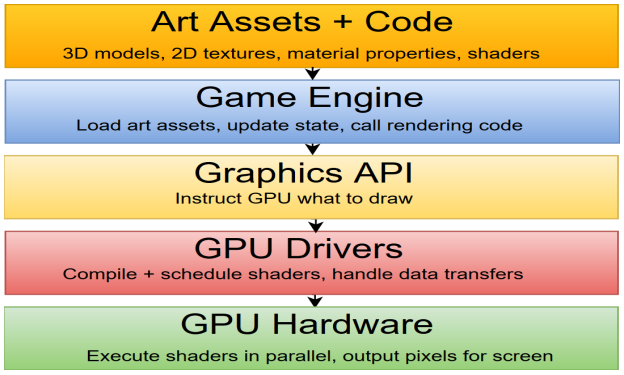


Fig. 1. Graphics Stack

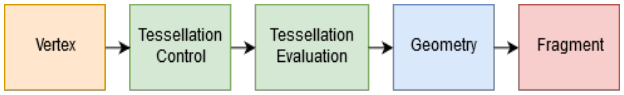


Fig. 2. Progammable Pipeline

Recent graphics technologies such as Vulkan [4] provide greater access to shader compiler infrastructure. However, due to its immaturity, there are few tools available. In this paper, we use LunarGlass [5], an existing source-to-source shader optimization tool for OpenGL [6]. We examine its impact on performance across different platforms, and extend it with new optimizations types, which are equally applicable for Vulkan.

The next section provides a motivating example illustrating the impact of transformations on different platforms. This is followed by description of the optimizations explored and a detailed description of the experimental setup. We characterise the benchmarks examined and then provide a detailed analysis of the impact of optimzations across platforms. This is followed by related work and some concluding remarks.

II. Motivating Example

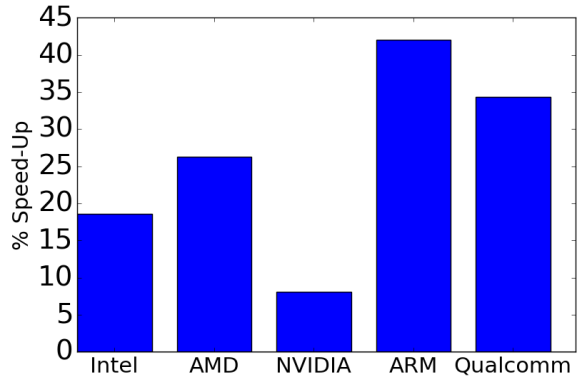
Listing 1 shows an example based on a GFXBench 4.0 shader [7] where offline compiler optimizations give performance improvements of up to 45% (see Figure 3). The

Listing 1. Before Optimization

```
out vec4 fragColor; in vec2 uv;
uniform sampler2D tex;
uniform vec4 ambient;
/*Main Function*/
//9 symmetric weights + sample offsets
const vec4[] weights = vec4[](
    vec4(0.01), ... , vec4(0.01));
const vec2[] offsets = vec2[](
    vec2(-0.0083), ... , vec2(0.0083));
float weightTotal = 0.0;
fragColor = vec4(0.0);
for(int i = 0; i < 9; i++){
  weightTotal += weights[i][0];
  fragColor += weights[i] *
   texture(tex,uv+offsets[i])*3.0*ambient;
}
fragColor /= weightTotal;
```

Listing 2. After Optimization

```
/*Main Function*/
fragColor = vec4(0.0);
vec4 fc1 = texture(tex,uv+vec2(-0.0083));
        ...
vec4 fc9 = texture(tex,uv+vec2(0.0083));
vec4 t0 = fc5 * vec4(1.83);
vec4 t1 = (fc4 + fc6) * vec4(0.63);
vec4 t2 = (fc3 + fc7) * vec4(0.42);
vec4 t3 = (fc2 + fc8) * vec4(0.15);
vec4 t4 = (fc1 + fc9) * vec4(0.03);
vec4 sum = t4 + (t3 + (t2 + (t0 + t1)));
vec4 fac = vec4(0.699301) * ambient;
fragColor = sum * fac;
```



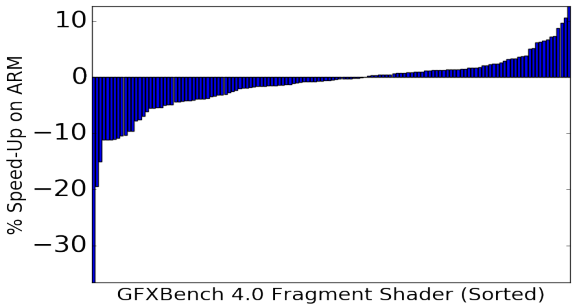


Fig. 3. Code before and after optimization, with the percentage performance gains on each platform. However, these optimizations are not universally positive, as seen in the distribution of percentage speed-ups across all GFX 4.0 benchmarks on the ARM Mali-T880 platform here.

example fragment shader repeatedly samples from a 2D image texture `tex` at various offsets from the coordinates `uv` (passed in from the vertex shader). The weighted sum of these samples is written to `fragColor` as the final pixel output.

This code has many optimization opportunities. Firstly, it contains an unrollable loop with 9 constant iterations. Once unrolled, the sum for `weightTotal` contains only constants, so can be completely evaluated. We can also invert the `weightTotal` before applying it, thereby changing 8 additions plus a division into a single multiplication. Each value summed for `fragColor` has a common multiple of `3.0*ambient` which can be factorised out, leaving 1 multiplication instead of 9. The constant 3.0 can then be folded into the `weightTotal` result. Because the weights are symmetric, pairs of texture samples will share weights as common multiples which can be factorised out too.

For this example, optimizations provide large performance impacts, with speed-ups of 7-28% on desktop, and 35-45% on mobile. This means GPU vendor's JIT compilers do not catch every optimization opportunity, and offline compilers can have a large impact. It also shows that performance impacts can vary drastically depending on which GPU is used.

Despite this large positive impact here, applying these optimizations to all fragment shaders in the GFX 4.0 benchmark, gives very variable performance results. In Figure 3, ARM's Mali-T880 gains up to 10% and loses of up to 30%. This shows us improvement is possible in real-world shaders, but a one-size-fits-all approach often does more harm than good. Smarter techniques to choose when and how to optimize each shader for each platform are necessary to reap the performance rewards but avoid the large performance pitfalls.

III. Optimizations

Here, we describe the source-to-source optimization techniques we explored, and the tools we used to do so.

A. LunarGlass Optimization Framework

To perform the source-to-source optimizations on GLSL [8] shader code, we used the LunarGlass framework from LunarG. This is a modified version of LLVM 3.4 [9], with several extensions for GLSL-specific intrinsic functions, optimization passes, and a GLSL front and back end. The default optimization passes which can be toggled via command-line flags are:

- **ADCE** - Aggressive dead code elimination.
- **Hoist** - Flatten conditionals by changing assignments inside "if" blocks into select "select" instructions.
- **Unroll** - Simple loop unrolling for constant loop indices.
- **Coalesce** - Change multiple individual vector element insertions into a single swizzled vector assignment.
- **GVN** - Global value numbering.
- **Reassociate** - Reorder integer arithmetic to simplify it (or some floating-point expressions like $f \times 0$).

Several other LLVM optimizations were included, such as constant folding, common sub-expression elimination, and redundant load-store elimination. However, we did not experiment with these passes as they were not exposed by default via command-line flags, and some were necessary passes to canonicalize instructions for future optimizations.

We took these pre-existing passes, added some extra ones to handle unsafe floating point arithmetic (see below), and then used iterative compilation to explore their impacts. Because only 8 flags were available, it was possible to exhaustively apply all 256 possible combinations of passes. Many of these resulted in duplicated source code (see Section V), so measuring the performance impact of all these generated outputs was tractable in this case. It may be possible to use results from this sort of exhaustive analysis to guide better flag selection heuristics or machine learning in future work.

B. Additional Unsafe Optimizations

In addition to LunarGlass's default passes, we added several extra unsafe floating point optimizations. Many of these mimicked parts of the integer reassociation pass to perform simple arithmetic simplifications such as:

$$ab + ac \longrightarrow a(b + c)$$
$$a + a + a \longrightarrow 3a$$
$$a + b - a \longrightarrow b$$

We also re-ordered arithmetic operations to group constants together (for better constant folding and propagation), and to group scalar operations before turning the results into vectors. This scalar reassociation was designed to minimize unnecessary registers slots holding temporary vector results, when single scalar registers would suffice:

$$f_1(f_2 v) \longrightarrow (f_1 f_2)v$$
$$c_1(c_2 v) \longrightarrow (c_1 c_2)v$$

Where f_1, f_2 are scalar floats, and c_1, c_2 are constants. This re-ordering also canonicalized the sequence in which the operands occurred, which could allow for greater common-sub-expression elimination opportunities in subsequent passes. Other identities such as multiplying by 1, or adding 0 were also optimized out, and division by constant operands was changed into multiplication by the operand's inverse (which could be determined at compile time).

The aim of these additional passes was to explore the impact of unsafe floating point optimizations which could not be implemented in a conformant GPU driver's compiler, but would fit well in an offline optimization framework where the developer can control when they are used.

C. Artefacts

Because OpenGL drivers only accepted shaders as GLSL source-code (until the recent SPIR-V extension was standardized in OpenGL 4.6), we had to use source-to-source optimizations. However, this lead to artefacts that would not occur in typical human-written GLSL code, and could sometimes negatively impact the code's performance. Such artefacts included:

a) Scalarized Matric Multiplications: GLSL has primitive types for both vectors and matrices, and humans may write code such as:

```
mat4 m1, m2; vec4 v;
mat4 m = m1 * m2; vec4 res = m * v;
```

When this is processed in LunarGlass, however, the matrices are divided up into their individual scalar components, and instead of 2 lines of matrix-vector calculations, tens of lines worth of scalarized calculations will be generated in LunarGlass's output GLSL.

b) Unnecessary Vectorization: In LLVM, operands for addition, multiplication etc. must be of the same type. This means when adding or multiplying a vector, the operands must both be vectors. In GLSL, the syntax allows you to multiply both vectors and matrices by scalars:

```
float f; vec4 v; vec4 res = v * f;
```

Since LunarGlass is based on LLVM, it has to vectorize these floating point values before multiplying them. This unnecessarily increases the number of vector constants and vectorization instructions, so may affect the amount of registers or constant storage memory for shaders.

c) Large Basic Blocks: The conditional flattening and loop unrolling passes result in very large basic blocks in the generated code. This can put pressure on the register allocators in the GPU vendor's compiler.

d) Mobile Shaders: In order to run the desktop OpenGL shaders on mobile devices (which use OpenGL ES), we first converted them to SPIR-V using glslang, and then used SPIR-V Cross to generate GLES compatible shaders. Having passed through so many compilation tools means the code picked up slight quirks and artefacts from each one in turn, and was often very different from the original desktop GLSL shader. As a result, some of the measurements on mobile may be impacted by artefacts that are not present on desktop.

IV. Experimental Setup

This section describes the shaders, execution framework, hardware, and timing techniques used to measure the performance impact of optimizations from Section III.

A. Benchmarks

For the timing experiments, we chose fragment shaders from GFXBench 4.0. This is a graphics hardware benchmarking suite from Kishonti [7], designed as standard way to compare the real-time rendering performance of GPUs from different vendors. The OpenGL version of GFXBench 4.0 contains several 3D animated scenes designed to use advanced and expensive rendering techniques to test the GPU's capability under heavy loads. Performance on these benchmarks is important to vendors because they are used to compare against GPUs from competitors.

We chose GFXBench 4.0 because it is a well-known, self-contained, cross-platform benchmark that covers a variety of different and potentially complex situations to test out shader compilation techniques. However, it is not open source, so shaders had to be extracted from the graphics driver at run-time. Because OpenGL's shaders are submitted to the GPU-vendor's compiler stack as GLSL source-code, they can be easily intercepted via the Linux's Mesa graphics drivers.

Many of the benchmark's shaders follow the "übershader" pattern, where a single file containing numerous graphics techniques is customised via preprocessor directives to enable or disable sections when generating shader instances. As such, some shaders are identical apart from preprocessor `#define` statements, forming families of similar shaders where some optimizations apply frequently because all include code segment, despite being specialized in different ways elsewhere.

B. Shader Execution Enviroment

To accurately time shaders, we executed them in an isolated context. Injecting them back into GFXBench would cause the performance impact of any single-shader optimizations to be lost in the noise of other shaders and CPU computations. As such, we built a custom measurement framework repeatedly rendered full-screen quads using the specified fragment shader, and timed the execution of each draw-call.

This work focuses on fragment shaders, as they are required in every graphics pipeline, are often more complex and varied than vertex shaders, and their performance can be more easily isolated from other pipeline stages. All the optimizations in Section III work for every shader type, but performance data was only measured for fragment shaders for this paper.

To reduce the overhead of non-fragment shader stages, we drew only full-screen triangles (clipped to 500*500 quads during rasterization), so only 3 vertex shader calls are required for every 250000 fragment shader invocations. Each frame, 1000 triangles (100 on mobile devices) were drawn front-to-back, and the draw calls were timed using queries to `GL_TIME_ELAPSED`. Although these queries can be noisy and introduce profiling overhead, and better vendor-specific instrumentation may be available, `GL_TIME_ELAPSED` provided a simple cross-vendor comparison metric that was accurate enough for basic performance results. The tests were run for 100 frames, and then repeated 5 times per shader variant. These large numbers of samples are used to reduce noise from environmental factors, profiling overhead, and measurement inaccuracies in the timer query API.

To run a fragment shader, we require a vertex shader with a matching interface. Instead of using GFXBench's vertex shaders, we automatically generate simplified ones based on the fragment shader's inputs. This reduces unnecessary overheads, provides flexibility to allow for shaders from other sources, and allows for a simple adjustment of the full-screen triangle's depth via a uniform variable in the vertex shader.

Some vendors also require all uniform variables and texture bindings to be initialised, so we used shader introspection to ascertain types and sizes for all uniform inputs. The framework then initialised them automatically to default values (e.g. 0.5 for floats, or a colourfully-patterned opaque power-of-two image for texture bindings). This is not representative of typical shader input, and may circumvent some data-dependent code paths. More complex techniques like input fuzzing, or extracting real-world inputs from GFXBench via instrumentation may provide better results, but experiments would take far longer to run due to a combinatorial explosion in input values. As such, we used the simple approach of using constant inputs, which still gives a broad overview of performance characteristics without the additional implementation complexity and run-time overhead.

C. Hardware

Timing experiments were run on 3 PCs and 2 mobile phones, each with a GPU from a different hardware vendor.

The desktop platforms were fitted with identical hardware apart from their GPUs. Each had 16GB of RAM, an i7-6700K CPU, and Ubuntu 16.10 installed. The GPUs and drivers chosen for each vendor were as follows:

- **NVIDIA** - GeForce GTX 1080, with OpenGL 4.5 and NVIDIA proprietry driver version 375.39
- **AMD** - RX 480 (8GB), with OpenGL 4.5 and Gallium 0.4 on AMD POLARIS10 (DRM 3.3.0 / 4.8.0-37-generic, LLVM 3.9.1) from Mesa 17.0.0-devel
- **Intel** - HD Graphics 530 (embedded on the i7-6700K), with OpenGL 4.5 and Mesa DRI Intel(R) HD Graphics 530 (Skylake GT2) from Mesa 17.0.0-devel

On Mobile, we used an HTC10 (with a Qualcomm GPU), and a Samsung Galaxy S7 (with an ARM GPU) to test on. Although GPU timer queries were integrated into desktop OpenGL 3.3 in 2010, they are only available on mobile via the EXT_disjoint_timer_query extension these phones were selected to support this. Both ran Android 7.0, and had the following GPUs and and CPUs:

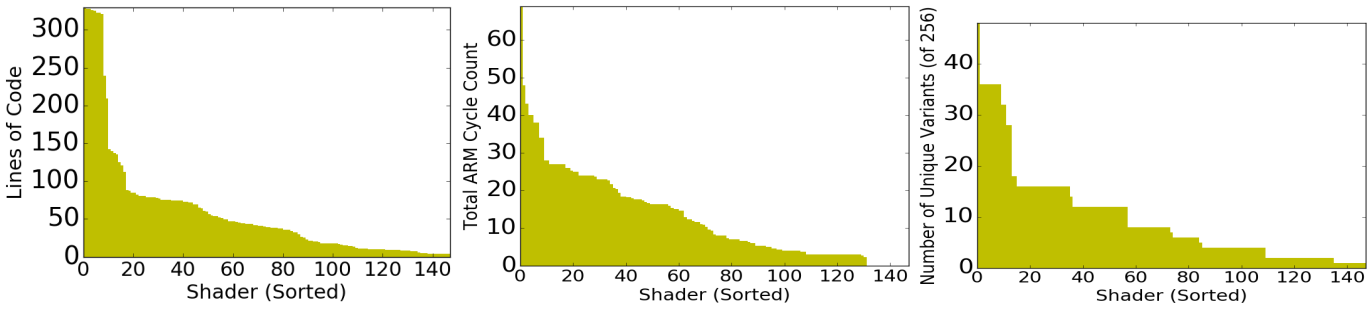
- **ARM** - Mali-T880 MP12 (on Exynos 8890 with quad-code Mongoose CPU and quad-core Cortex-A53 CPU)
- **Qualcomm** - Adreno 530 (on Snapdragon 820 with Kryo quad-core CPU)

V. Benchmarks

Graphics shaders are somewhat different in nature from typical CPU code or other forms of GPGPU code. Here, we examine the nature of the benchmark shaders, including their typical complexity, and their susceptibility to the optimization passes.

A. Static Code Size

Subfigure 4a, roughly illustrates the shaders' complexities using "lines of code" as a simple metric. This was measured after running the shaders through a preprocessor to get a more accurate idea of their complexity, because most of the raw GLSL shaders are generated from much larger base shaders that get split-up and recombined with GLSL preprocessor directives. This cuts down on the number of unused lines, but there are often still many unused function definitions left



(a) (b) (c)

Fig. 4. (a) Lines of code for GFXBench 4.0 fragments shaders (after preprocessing) (b) Sum of all cycles spent on Arithmetic, Load/Store, and Texture operations on the longest execution path (From ARM's offline shader compiler). (c) Number of unique shader variants generated from all possible combinations of LunarGlass and custom passes.

over from the larger base shader, that would be removed by dead-code elimination, but still contribute to the "lines of code" metric. Non-executable lines such as uniform and input parameter declaration, comments, white-space, lone brackets etc. are all ignored in when calculating these numbers too.

A shader's "lines of code" follows a power-law-like distribution, with very few lengthy shaders, and a numerous simpler shaders (many containing only a few lines). However, even the longest shaders are only around 300 lines. The majority of shaders are less than 50 lines (which may include unused function definitions too). This shows us that shaders are typically much smaller than an average piece of software. The shaders in the benchmark suite typically consist of long sequences of arithmetic, with only a small number of branches in their control flow. Loops are surprisingly uncommon in these shaders, and they mostly follow straight lines of execution with 1-3 branches, and large basic blocks.

B. Dynamic Cycle Count

Subfigure 4b uses ARM's static shader analyser to calculate how many cycles each shader takes on ARM's Mali GPU. Although this metric is platform-specific, it avoids problems like loops and unused function definitions that affect the "lines of code" metric. The ARM cycle count graph's power-law-like shape is less pronounced, but the distribution is quite similar.

Both graphs have long tails with a large number of low-complexity shaders. These simple shaders give less opportunities for compiler optimizations, as there are only a limited number of lines of code to deal with, and therefore a lower probability of finding instructions that can be optimized.

C. Uniqueness

Subfigure 4c shows how many unique variants LunarGlass generates for each shader. For the 8 possible flags, there are 256 potential combinations. In practice, however, most of the flags do not alter the source code, resulting in large numbers of duplicate shaders getting outputted. Even the shader with the most variants only has 48 distinct versions, with most of the others having less than 10.

Having so few unique variants makes it possible to exhaustively explore the search space, but there are very few data points for some of the optimization types. Since most shaders have < 50 lines of code, it is unsurprising that there are so few unique variants. Also, larger shaders with more interesting control flow are more likely to be affected by optimizations, but speed-ups here can sometimes be dwarfed by the long overall execution time.

VI. Results

This section examines the performance impact of the optimizations from Section III on the GFXBench 4.0 fragment shaders (see IV-A) across all target platforms (see IV-C). We discuss the effectiveness and applicability of the optimizations, and the performance variability across GPUs.

A. Overall Performance

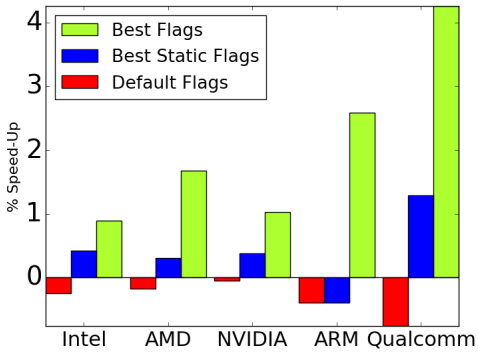


Fig. 5. Average percentage speed-ups across all shaders.

In Figure 5, our technique achieves average speed-ups of 1-4% across all shaders. In contrast, the default LunarGlass transformations give average slow-downs of 0-0.7%.

For some shaders, optimization leads to more substantial gains. The 30 most improved shaders on each platform (Figure 6), show average speed-ups of 4-13%. Some shaders experience gains as high as 25% (see Figure 7).

Table I. Best static flags for each platform: Flags that maximise the average speed-up across all the benchmark shaders.

	Flag							
Platform	ADCE	Coalesce	GVN	Reassociate	Unroll	Hoist	FP Reassociate	Div to Mul
Intel	-	✓	-	-	✓	-	✓	✓
AMD	-	✓	-	-	✓	-	✓	✓
NVIDIA	-	✓	-	-	✓	-	✓	-
ARM	-	✓	✓	✓	✓	✓	-	-
Qualcomm	-	✓	-	-	-	-	✓	✓
All	-	✓	-	-	✓	-	✓	✓

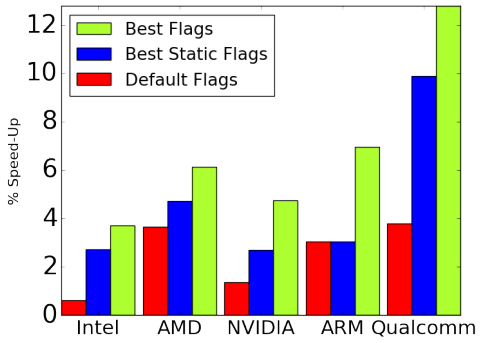


Fig. 6. Average speed-up for 30 shaders with the highest average per platform.

B. Best Static Flags

The "best static" flags in these diagrams are chosen by selecting the flag sequence with the highest average speedup per platform for all shaders. These flags are shown in Table I, and represent the optimal compilation settings to use if you cannot adapt on a per-shader basis. We can see that most platforms share similar flag preferences (ARM being the notable exception).

It is interesting note the best flags chosen experimentally are not the flags enabled by default (apart from for ARM). The default GVN, integer reassociation, and hoisting passes are detrimental on average despite being enabled by default. The ADCE pass never changes the output code, so can be safely omitted from the minimal optimal flag selection. Also, the new unsafe floating point passes generally have a positive enough impact to be included in the best static set of flags for all platforms (apart from ARM).

This similarity in optimal flags shows a surprising amount of agreement on which optimizations are beneficial to most vendors. However, in Subsection VI-D we can see that although vendors share preferences for the presence or absence of optimizations, the actual performance impact varies.

C. Per-shader Results

Figure 7 shows the performance distributions across all the individual shaders. All graphs have peaks and troughs on either end of a large near-zero mid-section. This demonstrates that frequently, optimization has little effect on shaders, but there are large performance peaks to strive for, and large performance troughs to avoid.

These graphs have by the large near-zero tails (particularly NVIDIA and Intel), where optimizations have little impact. This is due the relative simplicity of many shaders in the benchmark suite (see Section V), the low applicability of many optimizations (see Figure 8).

There are also cases where all optimizations cause slowdowns due source-to-source compilation artefacts (see III-C), or instances where loop unrolling and conditional flattening cause huge basic blocks which can strain register allocation code in the GPU vendor's compiler.

Despite these negative and near-zero cases, where the optimal strategy is leaving shaders untouched, there are still non-negligible performance gains available for around 25%.

On AMD, the biggest gains are available from some of the default passes like loop unrolling, so the default LunarGlass results are quite close to the optimal speed-ups.

On platforms like Qualcomm and Intel, much of the performance boost comes from the new unsafe floating point reassociation passes, and the default LunarGlass flags are closer to zero in these situations. This results in a larger blue area on the graph, because the main performance gains are from enabling these optimizations for all shaders, so there is less requirement to iteratively tune them.

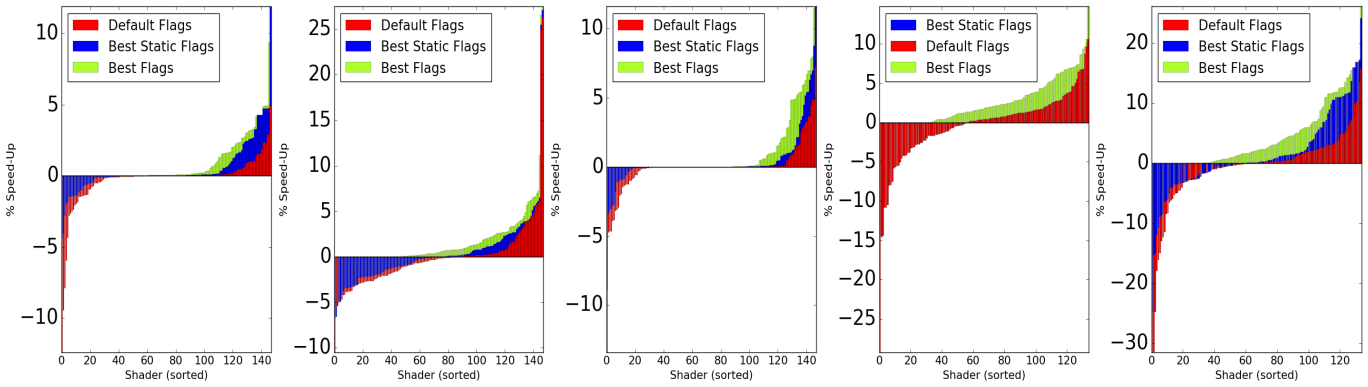
On ARM and NVIDIA, there are large green areas on the graphs, and small blue ones (the best-static and default LunarGlass settings are the same on ARM). This indicates that there is more to be gained from better flag selection heuristics on these platforms, as a single static set of flags does not guarantee significant performance improvement here.

All the graphs in Figure 7 and demonstrate that there are both large performance gains and performance pitfalls of between 10-30%. In many cases, the combination of boosting the maximum performance with the new custom passes, and eliminating poor optimizations means we are able to significantly improve over the default LunarGlass results.

D. Per-Flag Results

Here, we examine each flag's individual applicability and performance impact for each platform. Figure 8 show how frequently each flag applies to a shader, and how often using that flag results in optimal code. Green means it has a positive impact and denotes the number of times where the flag is frequently in the best performing codes, red means it has changed the output code while blue denotes the amount of code unaffected by the transformation

Figure 9 shows the performance impacts of each flag when used in isolation. Due to LunarGlass's compilation artefacts (see III-C), we use a baseline of LunarGlass running with all optimizations disabled here, rather than an unaltered shader, to



(a) Intel (b) AMD (c) NVIDIA (d) ARM (e) Qualcomm

Fig. 7. Percentage speed-up per shader for each platform. Green show the best possible performance, red is for default LunarGlass settings, and blue is for the best static set of flags from Table I

ensures we actually measure the optimization pass's impact, rather than the effect of the code generation artefacts. All the performance violins are centred close to zero due to all the low complexity shaders where the flags either do not affect the code (see Subfigure 4c), or change the source but do not impact the execution speed. As such, the extents and general shape of the violins are more interesting to observe than the mean values.

1) Aggressive Dead Code Elimination (ADCE): As can be seen from Subfigure 8h ADCE in practise never changes the source output. There is no green region showing it has a positive impact or a red region showing it has any impact. It should result in exactly zero speed up in the absence of noise, and can likely be safely omitted for most real-world shaders. However, this does not imply that dead code elimination itself has no impact. Trivially dead instructions get removed regardless of which flags are set, so ADCE's lack of effect simply means LLVM's `isTriviallyDead` function (plus extensions for GLSL-specific commands like `discard`) are sufficient to remove all the dead code.

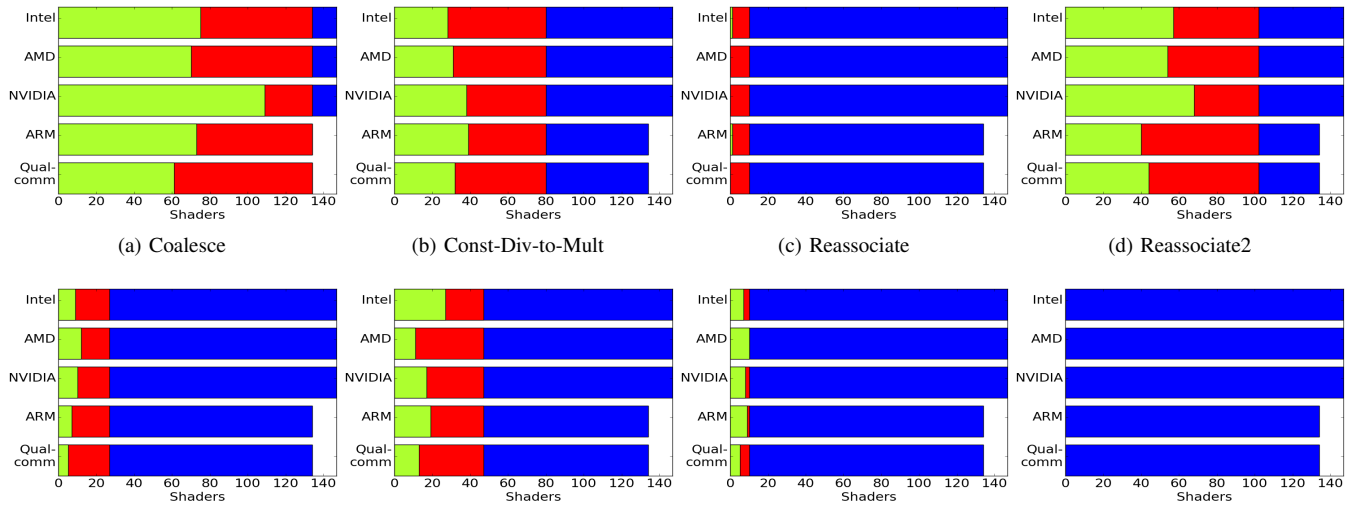
2) Global Value Numbering (GVN): GVN applies mainly to the few more complex shaders with many unique optimization variants, and generally has negative but near-zero impact. On Intel, it's results are very small, but generally negative. On NVIDIA, it's effects are centred around zero, but with one one example of 5% slowdown dragging its average impact down. Qualcomm, on the other hand, experiences gains of around 15% in some cases of using this flag, resulting on its average speedup being positive. Across all platforms, GVN is in optimal set for less than 50% of the shaders it applies to, so seldom improves code, even in the few cases where it applies.

3) Reassociate: The integer reassociation pass (Subfigure 8c) is rarely applicable, because integers occur very rarely in GLSL shaders. Most cases where it has any impact are actually removing unnecessary additions of zero in floating point calculations, rather than optimizing integer calculations. This pass near-zero impact in most cases, and makes things worse in a few, especially on NVIDIA where performance dips by 6% in one case. Integer reassociation almost never occurs in a shader's optimal set of flags, largely because its main use cases are eclipsed by the floating point reassociation pass instead. The low applicability, and chances for slow-down makes this flag is one of the least beneficial in LunarGlass.

4) Floating Point Reassociate: By contrast, the floating point reassociation pass applies to $> 50\%$ of shaders, and frequently occurs in their optimal set of flags (see Subfigure 8d). This high applicability gives this pass a wider spread of values. All platforms except ARM agree on its average positive impact, with peaks of around 5% improvement on desktop platforms, and peaks around 25% on Qualcomm. However, its results are not universally positive, and despite being Qualcomm's highest performance peak, it is also its lowest trough at -15%. On ARM, one 20% slow-down drags the average low enough omit it from ARM's best static flags. The wide spread of results, and the fact this flag appears shaders' optimal flag sets around 50% of the time, may indicate that although this pass's core ideas result in speed-ups, further refinement to reduce slow-down cases. Dividing this pass into smaller components and using better heuristics may achieve the performance gains without all the pitfalls.

5) Loop Unrolling: Loop unrolling (Subfigure 8g) is seldom applicable (as few shaders contain loops), but is almost universally positive. On AMD, loop unrolling always improves performance, and can result in 35% gains. On ARM, despite some slow-downs, it reaches a peak of 25%, making it the best flag on ARM as well. On Intel, it's effect is near-zero, with a slightly larger slowdown than speed-up. On NVIDIA, is also near-zero, but with a peak of 5% improvement. Qualcomm is the only platform that unrolling is not included in its best static flags (see Table I), and the 8% drop shown in Subfigure 9e may indicate why (although it also achieves gains in some cases too, and hovers near-zero for the most part). For most shaders, unrolling is one of the optimal flags on every vendor, and is a high-impact, low applicability transformation.

6) Hoist: The hoist flag (Subfigure 8f) applies to around 25% of shaders, but is in the optimal set for less than half



(a) Coalesce (b) Const-Div-to-Mult (c) Reassociate (d) Reassociate2

(e) GVN (f) Hoist (g) Unroll (h) ADCE

Fig. 8. Fractions of shaders where optimization passes apply, and have positive impacts. Blue is the total number of shaders. Red is how many the flag has any impact on the output code for. Green is number where the flag is included for at least half of the optimal 10% of variants for that shader.

of them. On most platforms, a single pathologically bad case drags the average down massively. On Intel, it drops 11%, on AMD 7%, on NVIDIA 5%, and on ARM it reaches a massive 35% slowdown. Hoisting sometimes improves all platforms, but these steep pitfalls indicate it should be used with caution, and good heuristics for when to apply it would be valuable.

7) Constant Division to Multiplication: Changing constant division to multiplication (see Subfigure 8b) is possible in $>$50% of shaders, but is only in the optimal set for around half of these. This is likely an optimization many vendors perform already, so this flag may have little impact. It might be a coin-toss whether results are negative or positive (hinted at by its symmetrical results in Figure 9), and could occur in optimal sets because it has a near-zero impact, so can be toggled on or off safely without slowing down shaders. On Intel (which has the least measurement noise), its impact is almost zero in all cases. The results for NVIDA, AMD, and ARM are symmetrical and centred around zero, with ranges around 4%, 10%, and 10% respectively. However, Qualcomm's results range from $+25\%$ to -13%. This pass's wide applicability makes it difficult to tell whether the graphs show genuine improvements, or merely each platform's measurement noise (which the symmetrical results might back-up).

8) Coalesce: The coalesce flag applies to almost every shader (see Subfigure 8a) because they frequently insert elements into vectors. Its results span a wide range, with most averages near-zero, or slightly negative impact (see Qualcomm). However, this is at odds with its inclusion in the best static flags in Table I, so the largely symmetrical spread of results may be due measurement noise again. Occurring in optimal sets for so many shaders shows it is frequently favourable to include, although different platforms have slightly different preferences. NVIDIA prefers it almost always enabled, but on Qualcomm it is optimal for around 50%, so is less critical.

E. Summary

Only floating point reassociation, loop unrolling, and hoisting have sufficiently large impacts to affect shaders in the absence of other passes. Also, as there are few distinct variants for each shader (see Subfigure 4c), the optimal 10% of variants is often only a single shader, so the optimality in Figure 8 may also be somewhat fickle. However, the larger visible performance trends, and the number of shaders each type of optimization pass applies to, gives some interesting insight into the nature of graphics shaders in general, and how frequently different optimization opportunities arise.

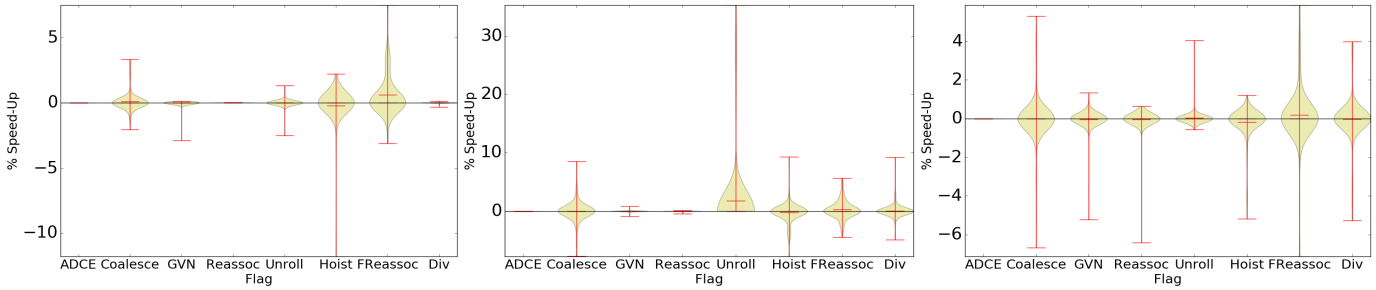
VII. Related Work

Few recent shader optimization papers exist. Most GPU papers cover general-purpose computation, whereas graphics papers typically cover algorithms rather than compilation. Shader optimization work is primarily industry-driven, as GLSL compilers are usually found in propriety GPU drivers. However, ecosystem changes such as Vulkan's SPIR-V compilation stack may allow researchers more access to this topic.

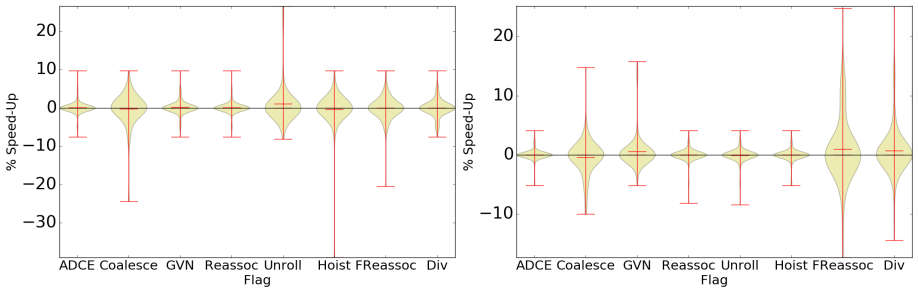
A. Shader Compilation and Pipeline Abstractions

Previous shader compiler work focuses on graphics pipeline abstractions rather than performance. This dates back to Cook suggesting composable "shade trees" in 1984 [10] . In 1985, Perlin [11] proposed per-pixel calculations (similar to modern-day fragment shaders), combining multiple passes for effects.

OpenGL originally exposed GPU hardware via a fixed-function pipeline. In Quake 3, idTech's shader language [12] abstracted this, and allowed multi-pass lighting effects. Similar multi-pass abstractions were also developed by Peercy et. al. [13], who used single SIMD instructions, and Proudfoot et.al. [14] who had multiple instructions per pass. NVIDIA's vertex unit became fully programmable in 2001 [15], and this



(a) Intel (b) AMD (c) NVIDIA



(d) ARM (e) Qualcomm

Fig. 9. Percentage speed-up from individual flags for each platform

trend continued, so modern APIs now expose 5 programmable pipeline stages to run arbitrary shader programs.

More recent pipelines includes GRAMPS [16] and Piko [17], which allow for novel topologies. SPARK [18] makes shaders modular and reusable in libraries. In SPIRE [19], calculations are performed at arbitrary rates, and an offline optimizer balances performance and visual quality when selecting which rates and algorithms to use. He et al. [20] then improve Vulkan performance via its parameter interface.

Another branch of shader compilation research is generating simplified code approximations which produce visually similar results. Olano et. al. [21] use lossy simplifications in a level-of-detail system where distant objects use faster drawing approximations. Pellacini [22] uses pattern matching rules to incrementally simplify shaders. Other techniques include genetic algorithms [23], moving code between pipeline stages [24], or interpolating texture samples with splines. [25]

B. Compilation for General-purpose GPU Computation

Many GPU papers focus on compilation for general-purpose computations in languages like OpenCL [26]. This often covers parallelising or scheduling algorithms, which is less relevant to the well-established graphics pipeline stages. However, some topics are relevant for both graphics and compute.

Jang et. al. [27] identify optimizations for AMD GPUs to improve ALU, texture unit, and thread-pool utilization, and also explore the effects of loop unrolling. Han and Abdelrahman [28] focus on reducing branch divergence, and use branch distribution to factor out common code from conditional expressions. Other work explores the importance scalars on GPUs [29]. Techniques such as scalar waving [30] pack individual scalar calculations into vector instructions. Our arithmetic reassociation technique to group scalars together (see Subsection III-B) would synergize well with this.

C. Existing Shader Tools

Many shaders profiling and compilation tools exist. Khronos's reference glslang [31] compiler turns Vulkan GLSL into SPIR-V [32], but contains few optimizations. SPIRV-Tools [33] includes a basic optimizer which unifies constants, inlines functions etc. but lacks the capabilities of larger frameworks like LLVM [9]. Google's shaderc [34] provides preprocessor directives and build-system integration around these projects.

SPIRV-Cross [35] is another tool for cross-compiling shader languages between one another, and performing reflection on shaders. It currently supports translating SPIR-V to GLSL for Vulkan, OpenGL, and GLES, HLSL [36] for DirectX, MSL for Metal [37], and C++ (to help with debugging). Another cross-compilation project is SPIRV-LLVM [38], but this currently only supports OpenCL SPIR-V, rather than Vulkan SPIR-V.

The Mesa project [39] provides Linux OpenGL drivers and includes a GLSL compiler. The glsl-optimizer [40] project from Unity [41] (a well-known game engine and IDE) modifies Mesa's shader compiler to optimize shaders for mobile devices. The Glassy Mesa project implements a variant of LunarGlass [5] for the Mesa drivers.

VIII. Conclusion

This paper introduced the graphics system stack and the programmable shader pipeline. We assessed the performance impact of common compiler optimizations on the fragment shaders of GFXBench 4.0 across 3 desktop and 2 mobile

GPUs. Although shaders undergo vendor-specific compilation, offline optimizations can still have significant positive and negative impacts, which vary across optimizations, benchmarks and platforms. Future graphics compiler technology may benefit from sophisticated profitability analysis, and automated machine-learning based techniques are likely to be attractive.

REFERENCES

[1] M. Claypool and K. Claypool, "Perspectives, frame rates and resolutions: it's all in the game," in *Proceedings of the 4th International Conference on Foundations of Digital Games*. ACM, 2009, pp. 42–49.

[2] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui, "Mobile augmented reality survey: From where we are to where we go," *IEEE Access*, 2017.

[3] Research Nester, "Computer graphics market : Global demand analysis & opportunity outlook 2024," https://www.researchnester.com/reports/computer-graphics-market-global-demand-analysis-opportunity-outlook-2024/354.

[4] The Khronos Vulkan Working Group, "Vulkan 1.0 core api specification," https://www.khronos.org/registry/vulkan/specs/1.0/xhtml/vkspec.html, 2016.

[5] LunarG, "Lunarglass compiler stack," https://lunarg.com/shader-compiler-technologies/lunarglass/, 2011.

[6] M. Segal and K. Akeley, "Opengl 4.5 core api specification," https://www.opengl.org/registry/doc/glspec45.core.pdf, 2016.

[7] Kishonti, "Gfxbench 4.0 - a bencmarking suite for opengl shaders," https://gfxbench.com.

[8] J. Kessenichm, "Opengl shading language 4.50 specification," https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf, 2016.

[9] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[10] R. L. Cook, "Shade Trees," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '84. New York, NY, USA: ACM, 1984, pp. 223–231. [Online]. Available: http://doi.acm.org/10.1145/800031.808602

[11] K. Perlin, "An Image Synthesizer," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '85. New York, NY, USA: ACM, 1985, pp. 287–296. [Online]. Available: http://doi.acm.org/10.1145/325334.325247

[12] P. Jaquays and B. Hook, "Quake 3: Arena shader manual, revision 10," in *Game Developers Conference Hardcore Technical Seminar Notes*. Miller Freeman Game Group, 1999.

[13] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar, "Interactive Multi-pass Programmable Shading," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 425–432. [Online]. Available: http://dx.doi.org/10.1145/344779.344976

[14] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan, "A Real-time Procedural Shading System for Programmable Graphics Hardware," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 159–170, 00233. [Online]. Available: http://doi.acm.org/10.1145/383259.383275

[15] E. Lindholm, M. J. Kilgard, and H. Moreton, "A User-programmable Vertex Engine," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 149–158. [Online]. Available: http://doi.acm.org/10.1145/383259.383274

[16] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A Programming Model for Graphics Pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, pp. 4:1–4:11, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1477926.1477930

[17] A. Patney, S. Tzeng, K. A. Seitz, Jr., and J. D. Owens, "Piko: A Framework for Authoring Programmable Graphics Pipelines," *ACM Trans. Graph.*, vol. 34, no. 4, pp. 147:1–147:13, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2766973

[18] T. Foley and P. Hanrahan, "Spark: Modular, Composable Shaders for Graphics Hardware," in *ACM SIGGRAPH 2011 Papers*, ser. SIGGRAPH '11. New York, NY, USA: ACM, 2011, pp. 107:1–107:12, 00023. [Online]. Available: http://doi.acm.org/10.1145/1964921.1965002

[19] Y. He, T. Foley, and K. Fatahalian, "A System for Rapid Exploration of Shader Optimization Choices," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 112:1–112:12, Jul. 2016. [Online]. Available: http://doi.acm.org/10.1145/2897824.2925923

[20] Y. He, T. Foley, T. Hofstee, H. Long, and K. Fatahalian, "Shader components: modular and high performance shader development," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, p. 100, 2017.

[21] M. Olano, B. Kuehne, and M. Simmons, "Automatic Shader Level of Detail," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 7–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=844174.844176

[22] F. Pellacini, "User-configurable Automatic Shader Simplification," in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 445–452. [Online]. Available: http://doi.acm.org/10.1145/1186822.1073212

[23] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic Programming for Shader Simplification," in *Proceedings of the 2011 SIGGRAPH Asia Conference*, ser. SA '11. New York, NY, USA: ACM, 2011, pp. 152:1–152:12. [Online]. Available: http://doi.acm.org/10.1145/2024156.2024186

[24] Y. He, T. Foley, N. Tatarchuk, and K. Fatahalian, "A System for Rapid, Automatic Shader Level-of-detail," *ACM Trans. Graph.*, vol. 34, no. 6, pp. 187:1–187:12, Oct. 2015. [Online]. Available: http://doi.acm.org/10.1145/2816795.2818104

[25] R. Wang, X. Yang, Y. Yuan, W. Chen, K. Bala, and H. Bao, "Automatic Shader Simplification Using Surface Signal Approximation," *ACM Trans. Graph.*, vol. 33, no. 6, pp. 226:1–226:11, Nov. 2014. [Online]. Available: http://doi.acm.org/10.1145/2661229.2661276

[26] A. Bourd, "The opencl specification version 2.2 decument revision 06," https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf, 2016.

[27] B. Jang, S. Do, H. Pien, and D. Kaeli, "Architecture-aware Optimization Targeting Multithreaded Stream Computing," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 62–70. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513903

[28] T. D. Han and T. S. Abdelrahman, "Reducing Branch Divergence in GPU Programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 3:1–3:8. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964184

[29] Z. Chen, D. Kaeli, and N. Rubin, "Characterizing scalar opportunities in gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 225–234.

[30] A. Yilmazer, Z. Chen, and D. Kaeli, "Scalar waving: Improving the efficiency of simd execution on gpus," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 103–112.

[31] Khronos Group, "glslang - khronos reference front-end for glsl and essl, and sample spir-v generator," https://github.com/KhronosGroup/glslang.

[32] J. Kessenich, B. Ouriel, and R. Krisch, "Spir-v specification provisional version 1.1 revision 4," https://www.khronos.org/registry/spir-v/specs/1.1/SPIRV.html, 2016.

[33] Khronos Group, "Spir-v tools - api and commands for processing spir-v modules," https://github.com/KhronosGroup/SPIRV-Tools.

[34] Google, "shaderc - tools and libraries for shader compilation from google," https://github.com/google/shaderc.

[35] Khronos Group, "Spirv-cross shader reflection and disassembly tool," https://github.com/KhronosGroup/SPIRV-Cross.

[36] M. Oneppo, "HLSL Shader Model 4.0," in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07. New York, NY, USA: ACM, 2007, pp. 112–152, 00000. [Online]. Available: http://doi.acm.org/10.1145/1281500.1281576

[37] Apple, "Metal shading language specification version 1.2," https://developer.apple.com/metal/metal-shading-language-specification.pdf, 2016.

[38] SPIRV-LLVM, "Spirv-llvm bidirectional conversion tool," https://github.com/KhronosGroup/SPIRV-LLVM.

[39] "The mesa 3d graphics library," https://www.mesa3d.org/, 2017.

[40] A. Pranckeviius, "Glsl optimizer based on mesa's glsl compiler," https://github.com/aras-p/glsl-optimizer.

[41] "Unity - a 3d game engine and development environment," https://unity3d.com/, 2017.