Specialization Opportunities in Graphical Workloads

Specialization Opportunities in Graphical Workloads

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

***Abstract*—Computer games are complex performance-critical graphical applications which require specialized GPU hardware. For this reason, GPU drivers often include many heuristics to help optimize throughput. Recently however, new APIs are emerging which sacrifice many heuristics for lower-level hardware control and more predictable driver behavior. This shifts the burden for many optimizations from GPU driver developers to game programmers, but also provides numerous opportunities to exploit application-specific knowledge.**

This paper examines different opportunities for specializing GPU code and reducing redundant data transfers. Static analysis of commercial games shows that 5-18% of GPU code is specializable by pruning dead data elements or moving portions to different graphics pipeline stages. In some games, up to 97% of the programs' data inputs of a particular type, namely uniform variables, are unused, as well as up to 62% of those in the GPU internal vertex-fragment interface. This shows potential for improving memory usage and communication overheads. In some test scenarios, removing dead uniform data can lead to 6x performance improvements.

We also explore the upper limits of specialization if all dynamic inputs are constant at run-time. For instance, if uniform inputs are constant, up to 44% of instructions can be eliminated in some games, with a further 14% becoming constant-foldable at compile time. Analysis of run-time traces, reveals that 48-91% of uniform inputs are constant in real games, so values close to the upper limit may be achieved in practice.

I. Introduction

Despite being a burgeoning industry projected to be worth $215 billion within 5 years [1], computer graphics receives minimal academic attention within the systems research community. Graphics papers typically aim to improve rendering algorithms or hardware, and many systems papers exist exploring architectural improvements or optimizations for general-purpose compute, but few focus on such systems-level optimizations for graphics workloads. This paper examines a variety of games, and explores the opportunities for shader specialization and data transfer optimizations they provide.

Graphical applications work similarly to flip-books – a series of still images (frames) are displayed quickly to give the illusion of motion, and the faster the frames are drawn (rendered), the smoother the motion appears. For PC games, rendering at least 60 frames-per-second (FPS) is standard, especially when quick reaction-times are required [2] [3], so games must update and render the world within 16ms.

The Graphics Processing Unit (GPU) is a specialized piece of highly parallel hardware designed to accelerate rendering and help games reach 60FPS. Games use pipelines of small single-program multiple-data (SPMD) graphics kernels called *shaders*. Thousands of instances of these SPMD shaders can run in parallel on different input data (e.g. one per pixel) before passing their output to the next round of shaders in the pipeline, and eventually outputting pixel colours to the screen. To set up these shader pipelines and control the GPU, games use a hardware-agnostic graphics API such as OpenGL [4]. Different hardware-specific implementations of the graphics API are supplied by GPU vendors within their drivers.

GPU drivers are filled with performance-boosting heuristics. GPU vendors often patch drivers shortly after major big-budget game releases with game-specific optimizations. One of the goals for newer lower-level graphics APIs like Vulkan [5] and DirectX 12 [6] is to provide developers with tools to create their own highly tuned application-specific optimizations.

Low-level APIs offer lower overheads, increased parallelism, more predictable cross-platform behavior, and finer-grained control at the expense of removing many driver heuristics. However, developers have access to application-specific knowledge that provides optimization opportunities beyond those possible within the GPU driver. This paper explores how game-specific knowledge can help drive specialization.

The GPU driver must compile shader code within tight time-constraints. However, developers may be able to perform more costly optimizations ahead of time during an offline build-process. Prior work shows that iterative compilation can give a speed-ups 4-13% on complex shaders [7]. This work focused only on compiling individual shaders, but real games often contain thousands. Furthermore, these techniques did not utilize data from the rest of the graphics stack.

As well as having more time to perform complex shader optimizations, game developers also have access to more context. It has been shown that over 99% rendering data is reused between frames [8], but is impractically large to keep in cache. Uniform variables are one type of data required by shaders. Uniforms supply the same value to all instances of an SPMD shader, and are frequently updated between frames.

In this paper we examine how much data requested by shaders is actually utilized, revealing that on average 41% (up to 97%) of uniform variables are unused in fragment shaders. Also, an average of 71% (up to 91%) of all uniform variables remain constant throughout the game's lifetime, not just between frames. 76% of attempts to update uniform values are redundant on average. This shows that we can potentially reduce the data transferred to the GPU, and can use the constant data to specialize shader code, reducing computation, memory footprint, and communication overhead.

This paper makes the following contributions

- The first static, oracle and dynamic analysis of specialization in existing graphics code

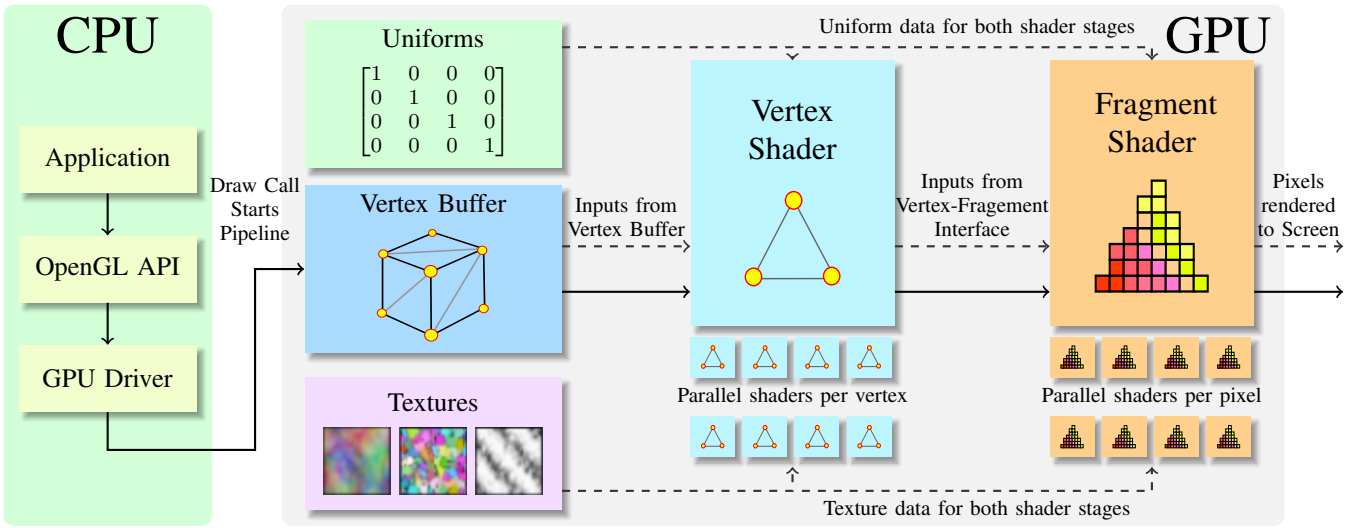


Fig. 1. Simplified graphics pipeline. The CPU submits draw calls through the OpenGL API to start the GPU rendering pipeline. Triangles are loaded from the 3D mesh in the vertex buffer. Vertex shaders process every triangle's vertices in parallel. The results are sent through the vertex-fragment interface, and get linearly interpolated across the triangle's surface. Using this, the fragment shaders calculate each pixel's colour in parallel, and send them to the screen.

- A large scale evaluation of over 12,000 shader programs in 8 real-world games
- Identification of significant specialization opportunities

II. Graphics Overview

Before exploring how we can specialize our graphics code, we first give a brief overview of how the graphics pipeline works, and introduce terms used in subsequent sections.

Consider the diagram in Figure 1. Here we can see that graphical applications have two parts - the CPU host code, and the GPU shader code. For games, CPU code is typically written in C/C++, and controls the GPU via a graphics API[1]

To display meshes on screen, OpenGL sends the GPU *draw calls*, which initialize state and start the shader pipeline as shown in Figure 1. Three types of data are input to the GPU: uniform, vertex buffer and textures which are accessed by the shader programs within the pipeline. On the GPU, shader programs are executed in a parallel pipeline, which usually consists of vertex and fragment shader stages. All 3D objects are represented as a series of triangles, which get stored within the vertex buffer. The first step in the pipeline is to load every vertex of every triangle from this buffer, and process them with vertex shaders running the same SPMD code in parallel (denoted by the multiple vertex boxes in Figure 1). After processing, the vertex shaders, write multiple outputs including the 3D vertex's position to appear on the 2D screen.

After vertex shading, the triangles are rasterized, meaning all their pixels get filled in. Vertex shader outputs get passed through the vertex-fragment interface, and are linearly interpolated across the surface of the triangle. Using these interpolated results, fragment shaders then run in parallel once for every pixel. They perform arbitrary calculations (e.g. lighting effects) before writing the pixel's final colour.

OpenGL shaders are written in GLSL (OpenGL Shader Language) [10], which has C-like syntax with extensions like vector and matrix primitives to simplify graphics calculations. Source-code for shaders gets submitted individually to the GPU driver's compiler, and then linked together into pipelines.

A. Data Types

Data can be passed to the GPU in several different ways:

Uniforms - Specified via the `uniform` keyword, these variables are passed from CPU to GPU. All parallel shader invocations receive the same value (i.e. it is uniform across all invocations). Uniforms are often small and frequently updated. They may contain values such as 4D transformation matrices specifying a mesh's position and orientation.

Inputs - Specified via the `in` keyword, these variables represent data from a prior pipeline stage. Vertex shader inputs come from the vertex buffer holding the 3D mesh's triangles. Each shader invocation receives a different vertex's data.

Textures - Textures are 2D images accessed using the `texture` function. This retrieve's the image's red, green, blue, and alpha channel values at a given 2D coordinate after performing hardware filtering and decompression.

III. Motivating Example

The example code in Figure 2 depicts a simple shader pipeline before and after specialization. The top two boxes `Vert` and `Frag` show the vertex and fragment shader code before we apply any specialization. The arrow between the boxes denotes that the variable `outUV` is an output from the vertex shader getting passed as an input to the fragment shader. The lower two boxes `Vert'` and `Frag'` refer to the code after specialization has taken place.

[1]On Microsoft platforms, DirectX 11 is most commonly used [9]. Cross-platform rendering applications use OpenGL or Vulkan. We focus on OpenGL in this paper, but the concepts are broadly similar across all APIs.

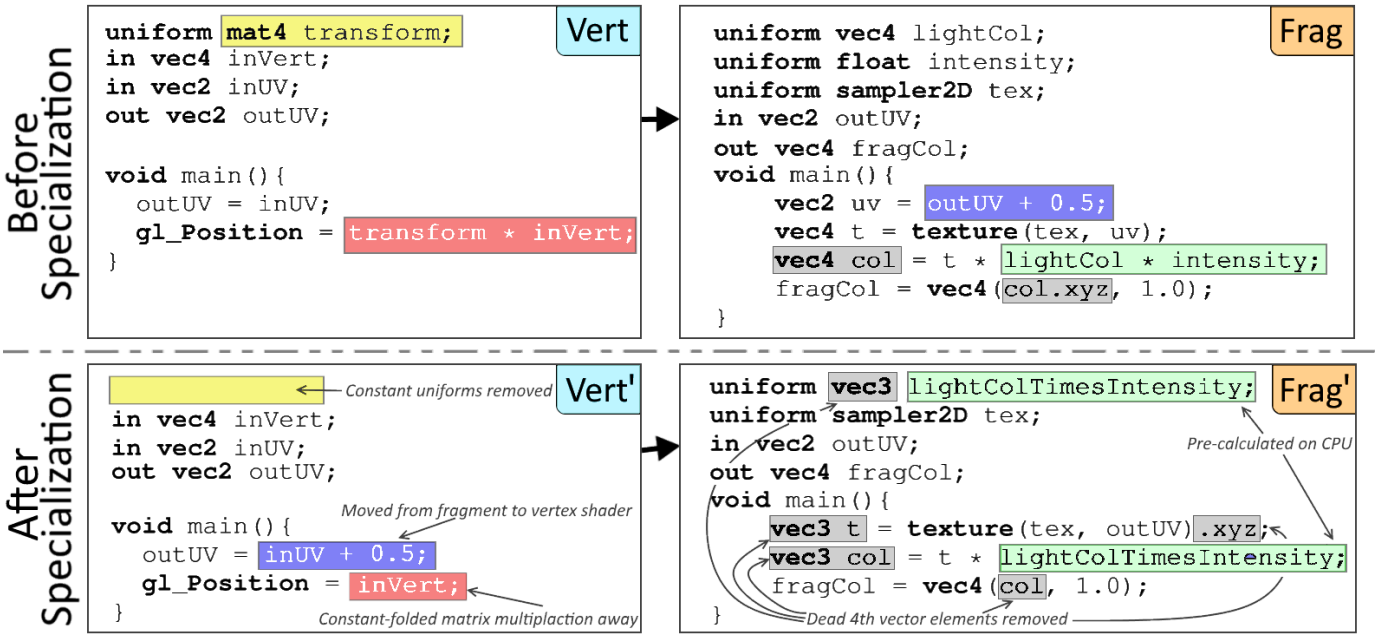


Fig. 2. Example vertex + fragment shader pipeline before and after code specialization. Coloured highlights in the "before" pipeline show specialization opportunities. After specialization, these highlight what has changed. We know that `transform` is the identity matrix through run-time profiling.
- constant - constant-foldable - movable to CPU - movable to vertex shader - contains dead elements

In `Vert` and `Frag`, we highlight a number of opportunities for specialization with colour-coded boxes. In `Vert'` and `Frag'` the highlights show the effects of the specialization. The specialization types examined are as follows:

Constant Uniforms. Consider the upper left hand box `Vert` and the declaration of `transform` highlighted in yellow. Run-time profiling has shown that `transform` is constant, and is known to be the identity matrix. This means that the declaration can be removed after specialization as shown in yellow in the lower program `Vert'`.

Constant-Folding - Knowing that `transform` is constant and the identity matrix lets us avoid the unnecessary matrix-vector multiplication `transform * inVert` highlighted in red in `Vert`. This multiplication is removed after specialization to give just `inVert` as highlighted in red in the vertex shader `Vert`'.

GPU-CPU Code Motion - Calculations using only uniforms can be pre-computed on CPU. The calculation `lightCol * intensity` highlighted in green in the `Frag` code of Figure 2 is an example of this. In the transformed code of in `Frag'`, this multiplication is performed on the CPU, and is passed in via the new uniform `lightColTimesIntensity`.

Fragment-Vertex Code Motion - Consider the calculation `outUV + 0.5` highlighted in blue in the fragment shader `Frag`. Its value is unaffected by linear interpolation, so can be moved to the vertex shader `Vert'` also shown in blue. This lets the fragment shader use the raw value of `outUV` for texture look-ups, potentially enabling better pre-fetching.

Removing Dead Vector Elements - If we again consider the code in `Frag`, only 3 components of `col` are used for the fragment shader's output, highlighted in grey. Knowing this, we see the 4th elements of `t` and `lightCol` are also unused. After specialization, we can change all these variables to `vec3`s instead, potentially saving registers, uniform buffer space, and texture unit bandwidth as shown in `Frag'`.

IV. Optimization Opportunities

This section provides more detail about the specialization opportunities demonstrated in Section III, and explains when they occur and how they might benefit performance. We start by defining some terms used throughout the paper:

Specialization Type: We focus on the following specialization opportunities: *Dead* code or data; *Movable* code; *Constant* data; *Constant-foldable* code *i.e.* compile-time computable.

Code Location: Code can be executed in the following locations: *CPU*, *Vertex Shader*; *Fragment Shader*.
Data Location: Data can come from the following locations: *Uniforms*; *Inputs* (from the vertex buffer or vertex-fragment interface); *Textures*.

Analysis Type: We perform the following types of analysis: *Static*; *Oracle* - an estimated upper bound on specializations *Dynamic* - using online profiling or offline trace analysis.

Specialization Granularity: We treat every element of vectors, matrices, and arrays individually. Specializations may be applied to variables either: *Fully* all elements can be

specialized; *Partially* only some elements can be specialized, but could be extracted.

We now provide more detail about the specialization opportunities explored, and their potential benefits. To provide continuity throughput the paper we use the same colour coding as used in Figure 2 to describe the various specializations:
constant constant-foldable movable to vertex shader
movable to CPU contains dead elements

A. *Dead*

Dead shader code and dead data elements in memory can be detected via static source-code analysis (see Subsection V-A), and provide the following optimization opportunities:

Dead Code: When some elements of an instruction's output have no impact on the final result. These may be pruned to reduce calculations, free up registers, or allow other non-dead elements to occupy their place.

Dead code can be eliminated using only static data. The driver's shader compiler may cull fully dead variables, but pruning partially dead ones is less common. Element-wise pruning may allow more compact storage e.g. combining 2 half-dead `vec4`s into a single live `vec4`. However, it must be applied carefully, as it may introduce unnecessary `mov` instructions or remove useful padding.

Dead Data: Elements in a shader's uniform, input, or texture interface that are either never loaded from, or the loaded result is never used. Removing dead data from a shader's interface may improve cache usage, and reduce the amount of memory, communication, interpolation slots, or texture look-ups required.

B. *Movable*

Static analysis can also detect which calculations may be transferred to different pipeline stages or to the CPU:

CPU-Movable: Some code can be moved from GPU shaders to the CPU. This can shrink the shader's uniform interface if the pre-computed results are smaller than the components of the calculation. Pre-computing values on the CPU may also improve performance if the GPU is the bottleneck. If branch conditions are evaluated on the CPU, it can select different specialized shader pipelines without GPU branches.

Vertex-Movable: If calculations are unaffected by linear interpolation, they can be transferred from the fragment to the vertex shader, which is typically invoked less often as objects usually have fewer vertices than pixels. This can reduce the total GPU calculations, use fewer vertex-fragment interpolation slots, and eliminate dynamically indexed texture look-ups. This code motion alters the vertex-fragment interface, and may be detrimental for meshes with many sub-pixel triangles.

Code motion requires knowledge of either the CPU-side code, or the shader pipeline linkage, these interfaces must be altered when code is moved. It will not always result in improved performance, so care should be taken that code movement only occurs where it will be beneficial. Using dynamic analysis increases the amount of code motion opportunities beyond those visible using only static data.

C. *Constant*

Detecting data that is constant at run-time enables further optimizations and specializations. In addition to the coloured squares used to label specialization opportunities, we use coloured circles to represent which type of data is constant:
Uniforms **Inputs** **Textures**

Constant Uniforms: Uniforms which always take the same value at run-time. Uniforms are stored on a per shader-pipeline basis, and are frequently updated, but are often set to the same value each frame.

Constant Inputs: Vertex shader inputs come from the vertex buffer. They are constant if they have the same value for all vertices in all buffers the pipeline uses (which is fairly rare). Fragment shader inputs come from vertex shader outputs, which may be constant at compile-time, or via propagating dynamic data detected at run-time.

Constant Textures: If a texture's colour channel is the same for every pixel, elements read from this channel are constant. This is common for fully opaque textures with constant alpha channels of `1.0`.

D. *Constant Foldable*

Constant-Foldable: Elements that are compile-time constant or known via prior run-time analysis can be folded in the compiler to reduce computation.

V. Methods

Here, we describe the analysis used to detect the specialization opportunities above, and the games used as benchmarks. The static specialization opportunities detected are described in Section VI, and the run-time trace analysis of constant uniform data is found in Section VII.

A. *Dead Code/Data Analysis*

To detect dead elements in the shader code (see Subsection IV-A), we use a backwards-propagating dataflow analysis. This starts with only final store calls live, and propagates this liveness into every other element used to calculate the existing live ones. After all the live elements were determined, all others were defined to be dead as can be seen in Algorithm 1.

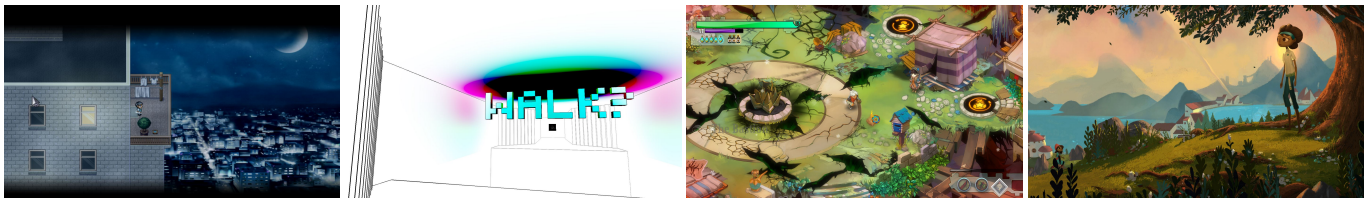
Dead uniform and input data is determined by examining which elements of `load` instructions are live, and merging results for any aliased loads. This let us see not only which variables were declared but never loaded from, but also which ones were loaded from but only partially used.

B. *Movable & Constant Code Detection*

To statically analyse which code elements were movable to the CPU or the vertex shader (see Subsection IV-B), we used a simple forward-propagating algorithm. This algorithm also tagged which sources different inputs were loaded from, enabling us to explore runtime-constant inputs (see Subsection IV-C) via an oracle study in Subsection VI-C.

We iterated through all the shader's instructions, and assigned every element of the return value the following values:[2]

[2]We use the same colour coding as before



(a) A Bird Story (b) Antichamber (c) Bastion (d) Broken Age

(e) Brütal Legend (f) Cities: Skylines (g) Costume Quest (h) Counter-Strike: Source

Fig. 3. Screenshots from the 8 Ubuntu-compatible games selected as benchmarks, covering a variety of different game-engines and art-styles.

Algorithm 1 Dead Element Detection

for all $Inst$ in $Instructions$ **do**
  if $Inst$ is a store, a terminator, or has side-effects **then**
    Set all elements of $Liveness_{Inst}$ to $Live$
    Add $Inst$ to $WorkList$
while $WorkList$ not empty **do**
  Pop $Inst$ from $WorkList$
  for all Op in operands of I **do**
    if $Inst \in$ { extract, insert, swizzle } **then**
      $Liveness_{Op} \longleftarrow$ permutation of $Liveness_{Inst}$
    if $Inst$ is an elementwise operation (e.g. +, *) **then**
      $Liveness_{Op} \longleftarrow Liveness_{Inst}$
    if $Inst$ uses all elements of Op (e.g. dot, cross) **then**
      $Liveness_{Op} \longleftarrow$ all elements $Live$
    if Any elements of $Liveness_{Op}$ changed **then**
      Add instruction defining Op to $WorkList$
All elements not set to $Live$ by now must be $Dead$

- **U** - Loaded from a Uniform
- **V** - Loaded as Input from a previous shader stage
- **T** - Loaded from a Texture
- **C** - Constant

We then used the following rules for tagging data as constant-foldable or movable:

- **CC** - Constant-foldable

$$\mathbf{CC} = f(\mathbf{C}, \mathbf{C})$$

where **CC** is the result of an arbitrary function $f()$ whose arguments are all constant.

- **UU** - Movable from GPU to CPU

$$\mathbf{UU} = f(\mathbf{U}, \{\mathbf{C}|\mathbf{U}|\mathbf{CC}|\mathbf{UU}\})$$

where **UU** is the result of a function $f()$ whose arguments consist of only uniform or constant values.

- **VV** - Movable from fragment to vertex shader

$$\begin{aligned}\mathbf{VV} = &\{\mathbf{V}|\mathbf{VV}\} \\ &*\{1|\mathbf{C}|\mathbf{U}|\mathbf{CC}|\mathbf{UU}\} \\ &+\{0|\mathbf{C}|\mathbf{U}|\mathbf{V}|\mathbf{CC}|\mathbf{UU}|\mathbf{VV}\}\end{aligned}$$

where the value of **VV** will be unaltered by linear interpolation between corners of the triangle, so must be a linear combination of a vertex input and either constants or uniforms.

All these tags are applied to scalar elements individually, and are permuted whenever a vector's elements are inserted, extracted, or shuffled. This gives a fine-level analysis, where the same vector may have different elements that are dead, movable, and constant. We implemented this analysis in LunarGlass, an LLVM-based compiler [11] which handles GLSL.

C. Dynamic Trace Analysis

To extract shaders and analyse the run-time behavior of our benchmarks, we used a modified version apitrace [12]. Widely used for debugging graphics drivers and games, apitrace is an open-source tool that injects itself into applications, and traces all OpenGL API calls. This trace can then be played back and inspected on different devices. Because OpenGL drivers require shader source code to be submitted for compilation, these shaders are recorded as arguments by apitrace. We extracted these shaders from the trace files, and used them for static analysis in Section VI. We also used apitrace to analyse how each shader's uniforms were updated by tracking every uniform update call.

D. Benchmark Games

We used a variety of 2D and 3D Linux-compatible games to gauge the behavior of typical graphics workloads (screenshots in Figure 3). The table in Figure 4 show the games used, and their total vertex and fragment shaders to give a rough idea of their complexity.

Antichamber uses the Unreal engine, and Cities: Skylines uses Unity, which are the most popular commercial game engines. Counter-Strike: Source uses Valve's famous Source engine, which is also available for studios to licence. Brütal Legend and Costume Quest use the in-house Budda engine,

Short Name	Full Name	2D/ 3D	No. Shaders		Avg. LOC	
			Vert	Frag	Vert	Frag
Bird	A Bird Story	2D	16	16	7	8
Anti	Antichamber	3D	1277	2833	105	70
Bast	Bastion	3D	7	22	11	17
BAge	Broken Age	2D	88	70	32	7
Brut	Brütal Legend	3D	262	1238	77	158
City	Cities: Skylines	3D	484	484	86	98
CQuest	Costume Quest	3D	150	888	70	150
CS:S	Counter-Strike:Source	3D	3025	503	72	42

Fig. 4. Benchmarks' abbreviations, numbers of shaders, and lines of code.

but have different art-styles. Including both allows us to see whether engines or art-styles have more impact on shaders data patterns. Bastion using another in-house engine, but its 3D graphics are much simpler, and many of the backgrounds are 2D. A Bird Story uses a modified version of the RPG-Maker, and its simple tile-based 2D graphics barely utilize shaders. Broken Age is a more complex 2D game in the open-source Moai engine, using skeletal animations, parallax layers, particles, and some 3D effects. These games span a wide variety of engines and art-styles of varying complexities, and are a good cross-section of games available on Linux.

VI. Shader Specialization Results

Here, we examine the results of our shader analysis described in Section V, and show how many of the specialization opportunities from Section IV occur in shaders from typical games. First, we use static analysis to determine that large percentages of code and data are dead (and in Section VIII, we see that removing this give up to 6x speed-ups). We then show how there are small statically available specialization opportunities in shader code, especially in hoisting conditionals from the GPU to the CPU to avoid branching.

In Subsection VI-C, we perform an oracle study to determine the upper bounds of code specializability if all uniform, input, or texture variables were known to be constant at compile time. The results indicate that large code reductions are possible, especially since many condition variables for branching or select statements become constant-foldable.

Based on the results of this oracle study, we examine in Section VII the actual dynamic values of uniforms and show that there is both a large amount of constant values and redundant update operations.

A. ▢ *Static Dead Code and Data*

Dead Code: Figure 5 shows that up to 13% of all scalar elements across all shaders in a game can be statically classified as dead code, with 6 / 9% dead on average for vertex/fragment shaders. This means most games have many shaders with a modest amount of dead code amenable to pruning. This modest amount is not surprising as all fully dead instructions are already stripped out by a previous compiler pass.

Dead uniform data: While there is little dead code, there is significant dead data. On average, 21 / 41% of vertex/fragment

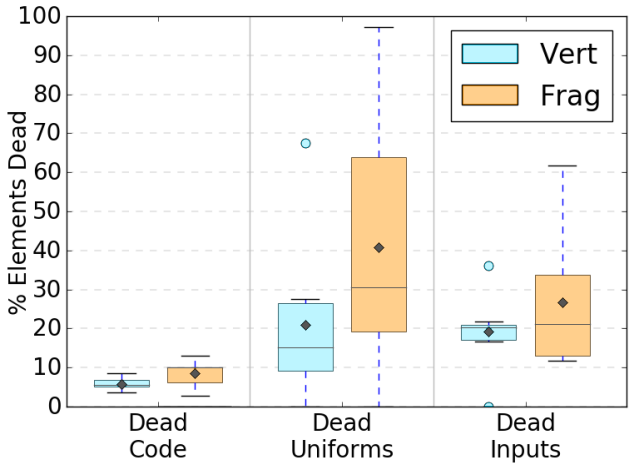


Fig. 5. Aggregated % dead code, and dead uniform and input data across all shaders for each game. Small portions of code elements, but sizeable amounts of uniform and input data are dead.

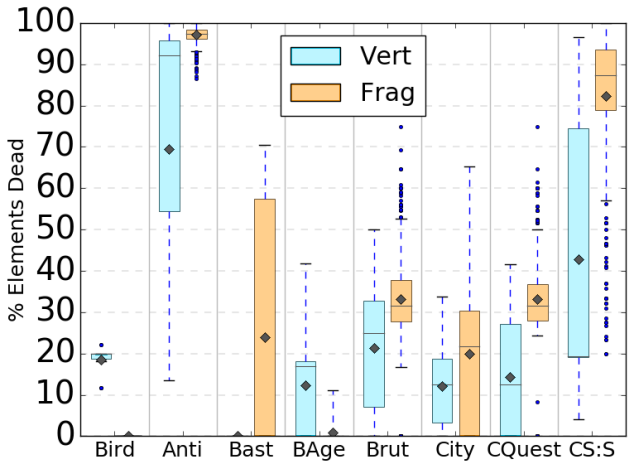


Fig. 6. % Dead uniform elements for all individual shaders in each game. There is wide variability between games, and for shaders within each game. Simple 2D games often have good uniform usage, but larger 3D games often have more dead elements, especially Antichamber and Counter Strike: Source.

uniforms are dead, as shown in Figure 5, but there is high variability among games. Some games' uniform interfaces are far larger than necessary. Complex 3D games with automatically generated shaders such as Antichamber, which uses the Unreal Engine [13] can produce pathological cases. Figure 6 provides more detail on a per game basis. Here we can see that 68 / 97% of Antichamber's declared vertex/fragment uniforms are statically dead. In contrast, simpler 2D games like A Bird Story utilize almost almost every uniform. Overall, fragment shader uniforms contain dead data more often than vertex shaders uniforms, and their usage varies more between games.

Dead input data: Input variables are also frequently dead. Figure 5 shows that on average, 19% of inputs from the vertex buffer and 27 % from the vertex-fragment interface are dead. There is again considerable variation across games as shown in Figure 7. Here 26% of Cities: Skylines vertex inputs and 62% of Antichamber fragment inputs are dead. This means

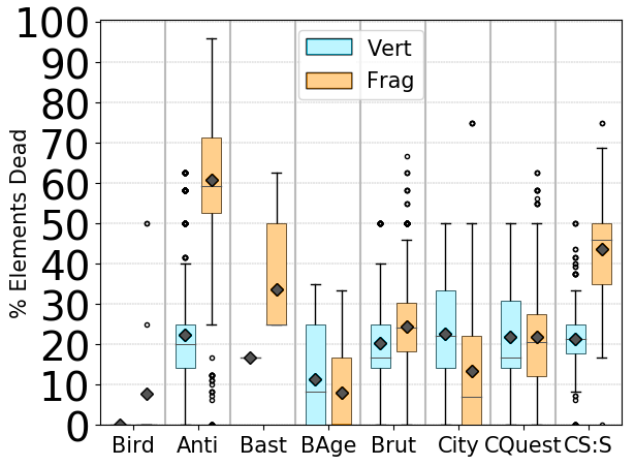


Fig. 7. % Dead input elements for all individual shaders in each game. Results are less variable than they are for uniforms. Most vertex shaders have around 20% dead inputs. Fragment shaders are more variable, with simple 2D games having better utilization than complex 3D games like Antichamber.

a significant portion of the vertex buffer could be reduced, lowering the mesh's memory footprint and speed up the vertex input assembly stage. If we back-propagated the dead data information across the vertex-fragment interface, then vertex computations could be significantly reduced. We could also shrink the size of the interface to lower the number of vertex-fragment interpolation slots and load instructions required.

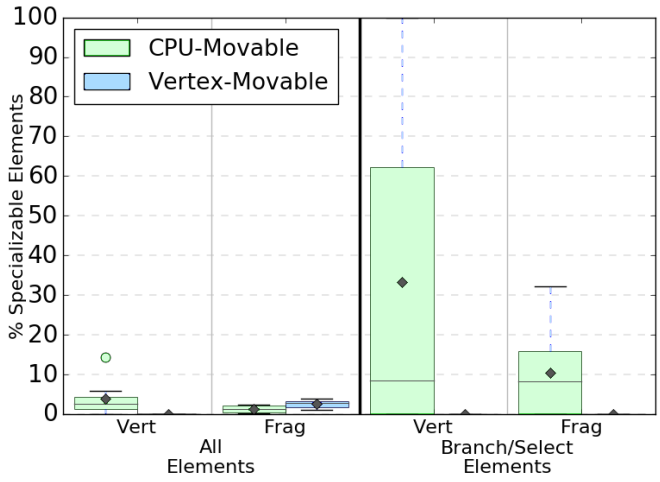


Fig. 8. Aggregated % statically movable code for across all vertex/fragment shaders for each game. Statically movable code makes up a small percentage of all instructions, but many branches could be hoisted to the CPU.

B. Statically Movable Code

As well as finding dead code elements, we can also statically determine how much code is movable to either the CPU, or the vertex shader (see Figure 8).

We can see that statically movable code makes up an even lower percentage of shaders than dead code. On average, 4 / 1% is movable to the CPU for vertex/fragment shaders, with up to 14% CPU-movable in A Bird Story's vertex shaders. An average of 3% (max 4%) is movable from fragment to vertex shaders too. This indicates that statically detecting movable code does little to reduce the overall percentage of instructions.

Branches: If we focus only on variables that are conditions for branching and select statements, we see that large portions of these are CPU-movable, as shown in the right-hand part of Figure 8. An average of 33 / 10% (up to 100 / 32%) of vertex/fragment shader conditionals can be hoisted to the CPU, indicating that expressions based on uniform variables are significantly over-represented in conditional statements.

All of the conditionals in Brütal Legend's vertex shaders, and 99% of those in Costume Quest's can be moved to the CPU. However, it should be noted that conditional statements are far less common in shaders than in typical CPU code. Moving the few existing branch conditions to the CPU might allow fully specialized pipelines to be selected, thereby avoiding GPU branching and removing large sections of shader code altogether. This can allow for better instruction pipelining, constant-folding, and instruction caching.

C. Oracle Constants

If we knew which shader inputs were constant at run-time, we could increase the amount of specialize the code. To quantify these effects, we perform an oracle study here to measure the upper limits of how specializable code becomes if we knew that 100% of each input type was constant. Based on this study, we explore the most promising candidates using dynamic analysis in Section VII. Figures 9 and 10 show the increase in all specializable instructions, and in specializable branch/select conditions. We now examine the effects of constant uniforms, inputs, and textures.

1) Oracle: Constant Uniforms: In the left hand column of Figure 9, we see that uniform load instructions make up an average of 23 / 17% of vertex/fragment shaders, so setting these as constants greatly reduces loads. There is also a small increase in constant-foldable code – 4 / 1% for average vertex/fragment shaders, and up to 14% in A Bird Story's vertex shaders.

Branches: The newly constant-foldable code has a large impact on branch and select instructions, as can be seen on the left hand column of Figure 10. Here, an average of 33 / 10% of conditions can now be statically determined, and up to 100 / 32% for some game's vertex/fragment shaders. This has large code reduction possibilities depending on which branch is statically selected. We can see in Subsection VII-A that large portions of a shader's uniforms are constant at run-time, so values close to these oracle results are likely to be achievable.

2) Oracle: Constant Inputs: From the middle column of Figure 9, we see that constant inputs are rarer than constant uniforms, but enable a wider range of specializations.

Vertex: If vertex shader inputs are constant, elements can be removed from the vertex buffer to reduce its memory footprint by an amount proportional to the number of triangles in the

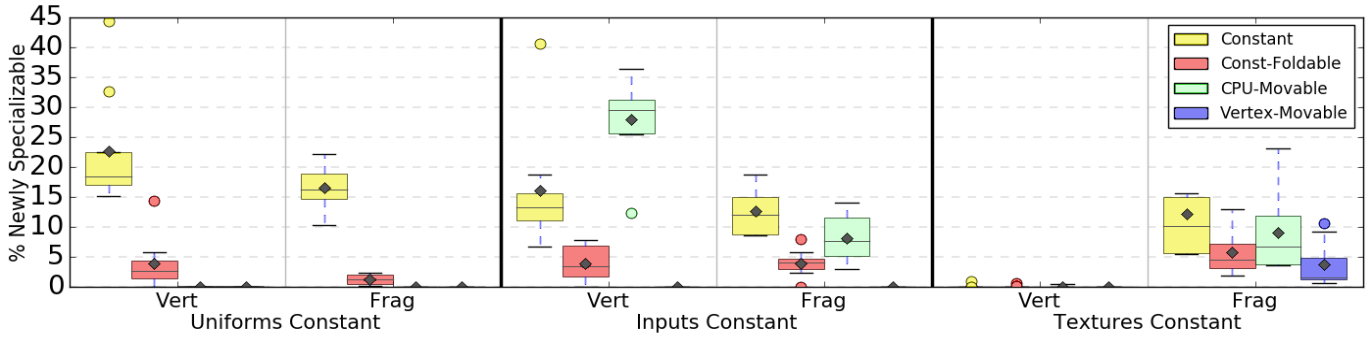


Fig. 9. % of shader elements that become specializable when setting all uniforms/inputs/textures to constants (aggregated across all shaders for each game). Load instructions significantly decrease for all types, and much of the vertex shader could be run on the CPU if vertex-buffer inputs were constant.

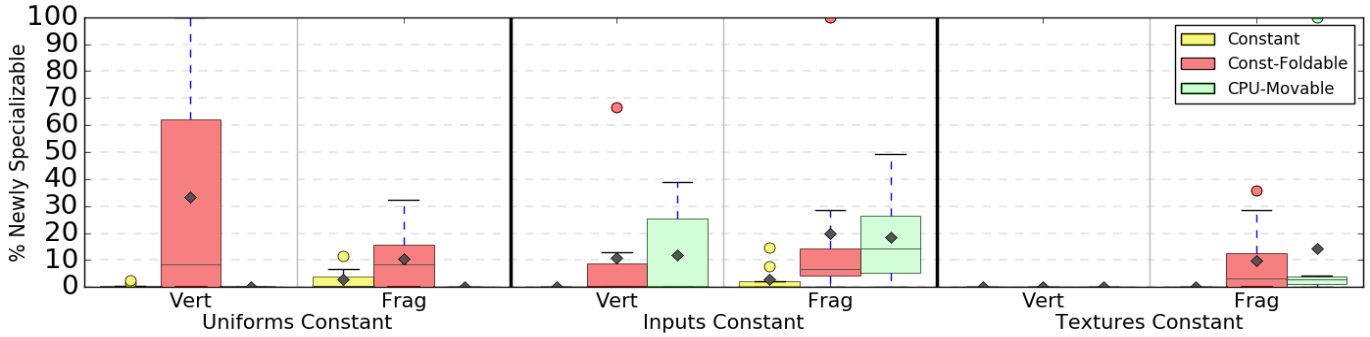


Fig. 10. % of branch/select conditions that become specializable when setting all uniform/input/texture resources to constants (aggregated across all shaders for each game). Many branches become constant-foldable or CPU-movable, with especially many branches becoming constant-foldable if uniforms are constant.

mesh (several thousand for complex models). On average, Vertex shaders' code is 16% input load instructions, which would all be specialized away if they were constant, and would render a further 4% constant-foldable. Much of vertex shader could also be extracted to the CPU, with 28% becoming CPU-movable on average.

Fragment: Fragment shader inputs are only constant when data passed out of the vertex shader is constant, so this occurrence is also rarer than constant uniform data. If this data was constant, however, fragment shaders would benefit from removing the 13% of instruction that were input loads, increasing constant-foldability by by 4%, and moving 8% of code to the CPU on average. Removing constant inputs from the vertex-fragment interface would also reduce the amount of linear interpolation required between shader stages.

Branches: As can be seen in Figure 10, constant inputs have less of an impact on branch conditions than constant uniforms. However, Figure 8 shows that most vertex shader conditionals are already statically CPU-movable, so there is less room for improvement using dynamic data. If all vertex buffer inputs were constant, over 70% of conditionals could be constant-folded or moved to the CPU (100% for most vertex shaders).

Constant input data in fragment shaders also impacts branching. An average of 23% (max 100%) more conditions become constant or constant-foldable, and 18% become CPU-movable. This means around 50% of fragment shader conditions can be specialized, which is a lower percentage than that of vertex shaders, but may account for more shaders overall, as fragment contain branches more often.

3) ■ ● *Oracle Constant Textures:* Consider the final column of Figure 9. Here, texture look-ups account for 12% of fragment shader code on average, which can be omitted if their values are constant. Fragment shaders from simple 2D games like A Bird Story consist of 30% texture look-ups. Fully constant textures are rare, but constant colour channels are not uncommon, and can increase code specializability in every way. With constant textures, code becomes an average of 6% more constant-foldable, 9% more CPU-movable, and 4% more movable from fragment to vertex shaders.

Branches: Constant textures also improve branch specializability as can be seen in Figure 9. 10% more conditions become constant-foldable on average (max 36%), and 14% more become CPU-movable (with 100% for A Bird Story). This has the least impact on specializing conditionals compared to constant uniform and input data, but still offers significant potential for branch reduction.

Texture look-ups are seldom used in vertex shaders. Cities: Skylines is the only game we tested that did so, but even then they were so rare that they had a $\leq$1% impact on specializability. All significant specialization opportunities that constant textures provide are contained within fragment shaders.

4) Oracle Study Summary: Using dynamically constant data can improve code specializability far beyond what is

possible with only static data. Utilising constant uniform data provides the largest reduction in load instructions, and can cause large percentages of branch conditions to become constant-foldable. This is true for both vertex and fragment shaders, and we shall show in Section VII that constant uniforms are very common in practice.

VII. Dynamically Constant Uniforms

Based on the previous section' observations, we explore to what extent uniforms are actually constant in practice.

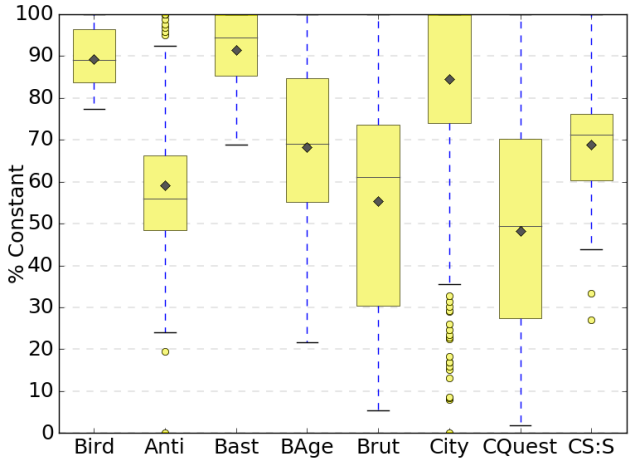


Fig. 11. % uniforms that are constant at run-time for all shader pipelines in each game. Despite high variability between shader pipelines, large portions of all uniform data is dynamically constant in every game.

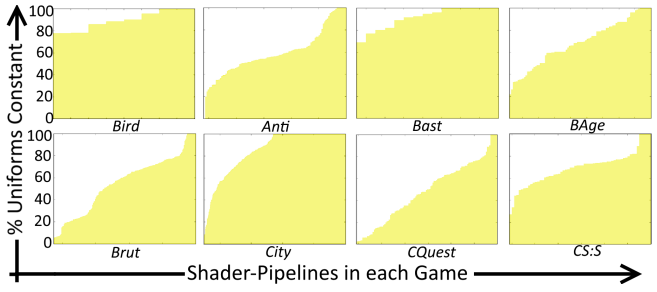


Fig. 12. % uniforms that are constant at run-time for all shader pipelines in each game. Games with fewer shaders like A Bird Story and Bastion have many constant uniforms for all pipelines. All games have significant portions of constant uniform data above 50% for most shaders apart from a few outliers.

A. *Constant Uniform Values*

In Figure 11, we see that an average of 60-90% of uniform variables in each shader pipeline are constant for almost every game tested. This means there is room in most games to achieve close to the oracle results for perfect specializability when setting all uniforms to constant. In Figure 12, we can see that most shaders within each game have at least 50% of their uniforms constant, so these optimization opportunities are almost ubiquitous among shaders.

This means that real games can achieve large reductions in uniform loading, and large increases in constant-foldable conditional statements, which allow many branches to be removed at compile-time if the dynamically constant uniform values are known.

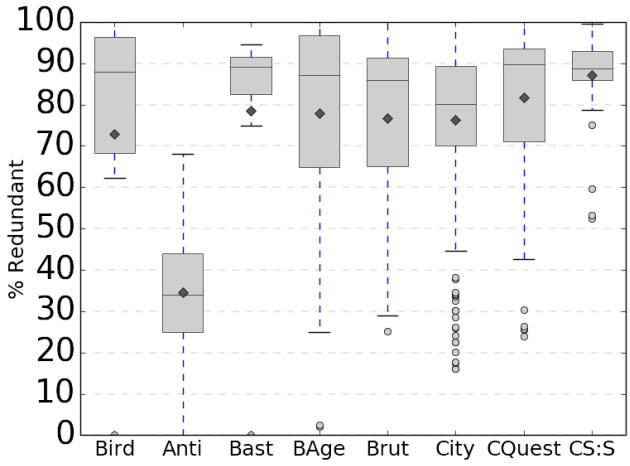


Fig. 13. % uniform element updates that are redundant, as they set uniforms to the same value they were previously. For almost every game, large percentages of uniform updates are unnecessary.

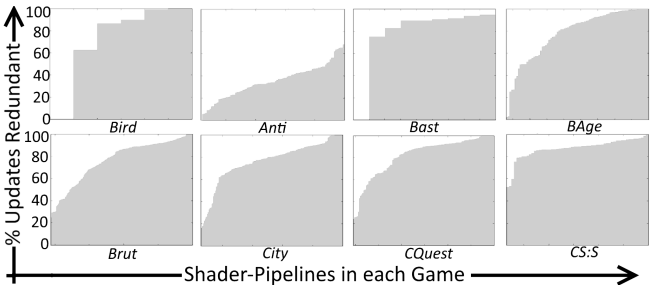
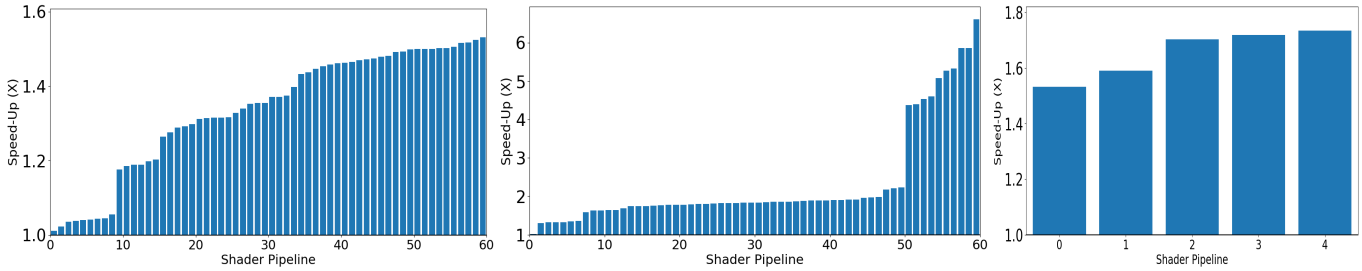


Fig. 14. % uniform element updates that are redundant for all shader pipelines in each game. In all games except Antichamber, over 75% of uniform updates to most pipeline set the uniform to the same value it was previously.

B. *Redundant Uniform Updates*

In figure Figure 11, we saw that large percentages of a game's uniform variables are constant at run time. In Figure 13, we can see that around 70-90% of all uniform updates are redundant, so the CPU-side code calling the update functions in the API is not making use of the constant nature of these values, and is unnecessarily updating them with the same values many times. From Figure 14, we can see that over 75% of uniform updates are redundant for almost every shader pipeline in each game. This means there are a large number of redundant OpenGL API calls, which might can have a significant performance impact in languages such as Java on Android devices, which incurs overhead for every JNI call. In the worst case, the driver does not catch this redundancy, and there is also unnecessary CPU-GPU communication.

Uniforms are updated very frequently, sometimes many times per frame, so there is room for optimization by reducing these frequent redundant update calls. It may be possible



(a) Antichamber (Ubuntu + GTX1080) (b) Antichamber (Windows + GTX970) (c) Counter-Strike (Windows + GTX970)

Fig. 15. Speed-up (X) in mean CPU time per frame when removing dead uniforms from several shader pipelines from Antichamber and Counter-Strike:Source. Timings were measured within an isolated timing harness performing thousands of draw-calls per frame with a single shader pipeline to reduce noise. On Windows 10 with a GTX 970, speed-ups reached up to 6.6x, with most around 2x. On Ubuntu with a GTX 1080, speed-ups were usually between 1.2-1.5x.

to fully specialize shaders such that all constant values are removed from the uniform interface, and the remaining dynamically updated uniforms are packed more tightly together to reduce communication bandwidth and the number of API calls required to update them. It may even be possible to use this sort of dynamic data to automatically group frequently-updated uniforms together so that seldom-updated values can be cached more effectively in a separate block.

VIII. Timing Tests

In order to determine whether some of the specialization opportunities detected would be able to provide concrete performance improvements, we built a timing tool to measure speed-ups on shaders before and after transformations. This timing harness was designed to run individual shader pipelines in an isolated environment to accurately more measure their performance characteristics.

In Figure 6, we can see that around 97% of Antichamber's fragment shader uniforms are unused, as well as a high percentage of its vertex shader uniforms. When examining these shaders' source-code, we noticed they all declare large arrays (224-256 elements) of `vec4` uniforms, but only tend to use the first 8 or so elements. This means we can remove large proportions of the dead uniform data by simply re-declaring these arrays using a smaller size (determined by the max array index used in the source code). Some vertex shaders contain an extra uniform array for bone transformations used in skeletal animations, which is indexed into via a variable rather than a constant, so we cannot statically determine which elements are dead, and would require run-time information on the array indices used in order to shrink these. However, typical uniform arrays are only ever accessed using constant indices, so it is simple to determine what the maximum index used is, and truncate the arrays accordingly.

We extracted the first 60 vertex/fragment shader pairs from a trace of Antichamber, and resized the uniform arrays to remove all trailing dead elements beyond the maximum index used. This typically meant that 2 arrays per shader pipeline with around 256 elements were shrunk to around 10 elements each.

We measured the performance impact of shrinking these arrays in our timing harness. This ran a simple rendering loop, iterating over 1024 models in a 32*32 grid, selecting the shader pipeline, vertex attribute bindings, textures, and uniform data to use for each model, and then rendering them to the screen. We recorded both the CPU and GPU time per frame over 100K frames, and took the mean of the median 80% of times to avoid outliers but still capture different phase change behaviors.

We can see the resulting speed-ups in Figure 15, which show significant performance improvements on two different PCs with Nvidia graphics cards. On a Windows 10 PC with a GTX 970, most shaders ran 2x faster, with several around 6x. The speed-ups on an Ubuntu machine with a GTX 1080 were more modest, ranging from 1.2-1.5x.

Counter-Strike:Source also has high percentages of dead uniforms (as seen in Figure 6) due to declaring large uniform arrays but only accessing a few elements. Unlike Antichamber, the indices used are sometimes spread out rather than clustered near 0, meaning we cannot just change the array's declared size, as we also need to compact it and alter the indices when accessing it. We performed this optimization manually on a several shaders from Counter-Strike:Source, and can see from Subfigure 15c that this also leads to significant speed-ups. The results are less extreme than Antichamber's, as the initial arrays are around 50 elements rather than 256, but the array compaction still leads to 1.5-1.7x improvements.

IX. Summary of Results

We performed static analysis of over 12,000 shaders extracted from 8 different games, as well as dynamic trace analysis of their run-time behavior. Through this, we found:

- Significant amounts of uniform and input data is dead.
- Culling dead uniforms sometimes provides 6x speed-ups.
- Many branch conditions can be hoisted to the CPU.
- If uniforms, inputs, or textures were constant, we could significantly reduce shaders' load instructions and increase their constant-foldable and CPU-movable code.
- If uniforms are constant, many branches can be determined at compile-time.
- Most uniforms are constant at run-time.
- Most updates to uniform values redundantly set them to the same value each time.

X. Related Work

The concept of specializing shaders has existed for decades [14]. However, shaders were quite different in this era, with abstractions like Cook's shader trees [15] and Perlin's image synthesizer [16] proposing flexible abstract ways of rendering 3D images. Much of the early graphics work was focused on ray-traced rendering for films, such as using Pixar's Renderman shading language [17], which later had custom SIMD hardware developed for it [18].

Early consumer GPUs for real-time rendering in games had fixed-function hardware, but researchers proposed ways to make it programmable [19]. Around 2001, GPUs began executing arbitrary vertex shaders [20], with fragment shaders and other user-programmable stages following in subsequent years. Researchers quickly exploited these capabilities for more than graphics computations [21], and the majority of GPU compiler and system optimization research today focuses on these general-purpose computations.

Much of graphics systems research is focused on developing novel pipeline abstractions such as RTSL [22], shader algebra [23], abstract shader trees [24], GRAMPS [25], SPARK [26], CoSMo [27], Spire [28], and Braid [29]. Research has also explored alternative shader programming languages and meta-programming systems like Sh [30], Cg [31], Renaissance [32], and Slang [33]. Most of these languages and pipeline abstractions focus primarily on allowing greater programmer productivity and expressiveness, rather than aiming to improve performance of existing systems.

One graphics research area focused on improving shading performance, is splitting large and complex shaders into multiple less expensive passes [34] [35] [36] [37] [38]. Other early work examines methods of optimizing resources for better shader throughput [39], or re-ordering shader pipeline calls to minimize state changes [40]. He et. al. have suggested using modular shader components [41] to aid both productivity and performance by allowing more opportunities for shader specialization and allowing better pipeline resource binding to avoid large amounts of unused shader parameters being defined (as this paper's results show is common). Research has also been done into code motion techniques to optimize power consumption rather than performance [42]. Other papers explore offline compiler optimizations for GLSL shaders [7], and analysing memory usage patterns [8], but little other work has examined the data redundancy and code specialization opportunities of real-world graphics workloads.

An adjacent field of study aiming to automatically speed up shader code, is to create simplified versions of shaders. When objects are far away, it is common for simplified visual representations to be used, with an increasing level of detail (LOD) used for objects closer to the camera. Olano et. al. exploit this idea to generate simplified shader code for different LODs [43] [44] [45] by repeatedly applying lossy code transformation rules. Others have extended these simplification techniques using pattern matching [46], genetic programming [47], and surface-signal approximations [48]. Searching for simplified shader variants with a good balance between visual error and performance could take hours, but He et. al. [49] introduce search heuristics to reduce this to minutes. Recently, Yuan et. al. [50] introduce techniques to perform simplification-space searches in milliseconds, allowing real-time simplification of the most expensive few shaders, and the ability to exploit run-time context when determining the simplification's visual error. NVidia have also implemented run-time shader LOD simplifications based on their material definition language [51]. Although this paper focuses mainly on lossless optimization opportunities, the patterns in dead and constant data we have show here may be also exploitable in lossy shader simplification techniques like those above, especially since some are now capable of running in real-time.

XI. Conclusion

Given full knowledge of a graphics system, it is possible to heavily specialize shaders for large potential performance gains and memory bandwidth savings. As industry is moving to a state where such specializations might be done by game developers rather than driver engineers, such domain knowledge is exploitable.

Large amounts of uniform data is dynamically constant in typical games, so is ripe for shader specialization. There are also many other potential specialization opportunities for removing redundant buffer data and re-packing it, or transferring code from GPU to CPU or between shader stages.

This paper acts as a motivation for cross-stack graphics optimization tools by pinpointing some widespread opportunities for optimization. Future work will explore these opportunities.

References

[1] 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.

[2] K. T. Claypool and M. Claypool, "On frame rate and player performance in first person shooter games," *Multimedia systems*, vol. 13, no. 1, pp. 3–17, 2007.

[3] B. F. Janzen and R. J. Teather, "Is 60 fps better than 30?: the impact of frame rate and latency on moving target selection," in *CHI'14 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2014, pp. 1477–1482.

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

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

[6] Microsoft, "Direct3d 12 programming guide," https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx, 2016.

[7] L. Crawford and M. O'Boyle, "A cross-platform evaluation of graphics shader compiler optimization," in *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 219–228.

[8] G. Ceballos, A. Sembrant, T. E. Carlson, and D. Black-Schaffer, "Behind the scenes: Memory analysis of graphical workloads on tile-based gpus," in *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 1–11.

[9] A. Valdetaro, G. Nunes, A. Raposo, B. Feijó, and R. De Toledo, "Understanding shader model 5.0 with directx11," in *IX Brazilian symposium on computer games and digital entertainment*, vol. 1, no. 2, 2010.

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

[11] 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.

[12] J. Fonseca, "apitrace graphics tracing tool," http://apitrace.github.io/, 2008.

[13] "Unreal engine - a 3d game engine and development environment," https://www.unrealengine.com, 2017.

[14] B. Guenter, T. B. Knoblock, and E. Ruf, "Specializing shaders," in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. ACM, 1995, pp. 343–350.

[15] R. L. Cook, "Shade trees," *ACM Siggraph Computer Graphics*, vol. 18, no. 3, pp. 223–231, 1984.

[16] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[17] P. Hanrahan and J. Lawson, "A language for shading and lighting calculations," in *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4. ACM, 1990, pp. 289–298.

[18] M. Olano and A. Lastra, "A shading language on graphics hardware: The pixelflow shading system," in *SIGGRAPH*, vol. 98, 1998, pp. 159–168.

[19] 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*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 425–432.

[20] 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*. ACM, 2001, pp. 149–158.

[21] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM transactions on graphics (TOG)*, vol. 23, no. 3, pp. 777–786, 2004.

[22] 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*. ACM, 2001, pp. 159–170.

[23] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 787–795, 2004.

[24] M. McGuire, G. Stathis, H. Pfister, and S. Krishnamurthi, "Abstract shade trees," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 79–86.

[25] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "Gramps: A programming model for graphics pipelines," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 1, p. 4, 2009.

[26] T. Foley and P. Hanrahan, *Spark: modular, composable shaders for graphics hardware*. ACM, 2011, vol. 30, no. 4.

[27] G. Haaser, H. Steinlechner, M. May, M. Schwärzler, S. Maierhofer, and R. Tobler, "Cosmo: Intent-based composition of shader modules," in *Computer Graphics Theory and Applications (GRAPP), 2014 International Conference on*. IEEE, 2014, pp. 1–11.

[28] Y. He, T. Foley, and K. Fatahalian, "A system for rapid exploration of shader optimization choices," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, p. 112, 2016.

[29] A. Sampson, K. S. McKinley, and T. Mytkowicz, "Static stages for heterogeneous programming," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 71, 2017.

[30] M. D. McCool, Z. Qin, and T. S. Popa, "Shader metaprogramming," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 2002, pp. 57–68.

[31] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a c-like language," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 896–907, 2003.

[32] C. A. Austin and D. Reiners, "Renaissance: A functional shading language," Ph.D. dissertation, Iowa State University, 2005.

[33] Y. He, K. Fatahalian, and T. Foley, "Slang: language mechanisms for extensible real-time shading systems," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, p. 141, 2018.

[34] E. Chan, R. Ng, P. Sen, K. Proudfoot, and P. Hanrahan, "Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 2002, pp. 69–78.

[35] T. Foley, M. Houston, and P. Hanrahan, "Efficient partitioning of fragment shaders for multiple-output hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2004, pp. 45–53.

[36] A. Riffel, A. E. Lefohn, K. Vidimce, M. Leone, and J. D. Owens, "Mio: Fast multipass partitioning via priority-based instruction scheduling," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2004, pp. 35–44.

[37] A. Heirich, "Optimal automatic multi-pass shader partitioning by dynamic programming," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 2005, pp. 91–98.

[38] B. Silpa, K. S. Vemuri, and P. R. Panda, "Adaptive partitioning of vertex shader for low power high performance geometry engine," in *International Symposium on Visual Computing*. Springer, 2009, pp. 111–124.

[39] P. Lalonde and E. Schenk, "Shader-driven compilation of rendering assets," in *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 713–720.

[40] J. Krokowski, H. Räcke, C. Sohler, and M. Westermann, "Reducing state changes with a pipeline buffer." in *VMV*, 2004, p. 217.

[41] 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.

[42] Y.-P. You and S.-H. Wang, "Energy-aware code motion for gpu shader processors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3, p. 49, 2013.

[43] M. Olano, B. Kuehne, and M. Simmons, "Automatic shader level of detail," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 2003, pp. 7–14.

[44] M. Olano and B. Kuehne, "Sgi opengl shader level-of-detail shader white paper," SGI, Tech. Rep., 2002.

[45] M. Simmons and D. Shreiner, "Per-pixel smooth shader level of detail," in *ACM SIGGRAPH 2003 Sketches & Applications*. ACM, 2003, pp. 1–1.

[46] F. Pellacini, "User-configurable automatic shader simplification," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3. ACM, 2005, pp. 445–452.

[47] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic programming for shader simplification," in *ACM Transactions on Graphics (TOG)*, vol. 30, no. 6. ACM, 2011, p. 152.

[48] R. Wang, X. Yang, Y. Yuan, W. Chen, K. Bala, and H. Bao, "Automatic shader simplification using surface signal approximation," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 6, p. 226, 2014.

[49] Y. He, T. Foley, N. Tatarchuk, and K. Fatahalian, "A system for rapid, automatic shader level-of-detail," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 6, p. 187, 2015.

[50] Y. Yuan, R. Wang, T. Hu, and H. Bao, "Runtime shader simplification via instant search in reduced optimization space," in *Computer Graphics Forum*, vol. 37, no. 4. Wiley Online Library, 2018, pp. 143–154.

[51] L. Kettner, "Fast automatic level of detail for physically-based materials," in *ACM SIGGRAPH 2017 Talks*. ACM, 2017, p. 39.