Thomas Hinterkircher, BSc

Direct3D Shader Crosscompiler

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Institute for Computer Graphics and Knowledge Visualization
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Co-Supervisor
Dipl.-Ing. Michael Kenzel BSc

Graz, März 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

In recent developments, GPGPU technologies have become flexible enough to make GPU-based software renderers feasible. Popular computer graphics libraries and increasing demands of developers and artists have originally driven the move towards programmable GPUs, but do still rely on dedicated hardware for some of the functionality, such as rasterization, depth testing or blending. However, unlike GPGPU frameworks, a specialized shading language is used for programmable stages (shaders) in the rendering pipeline. Providing a compatible graphics API with an underlying software implementation requires means of executing those shaders.

In this master's thesis, a program to compile Direct3D shaders to CUDA device code has been developed. The input language is not HLSL, a shading language designed for Direct3D, but a driver-independent low-level intermediate binary format produced by Microsoft's HLSL compiler. This same form is also used in the Direct3D API, making it a requirement for sufficient compatibility. For each shader, a CUDA unit containing C++ or PTX code is produced, which can later be linked statically or at runtime with a software renderer.

Contents

List of Figures

1. Introduction

Modern graphics application programming interfaces (APIs) provide means to implement some parts of their rendering pipeline with small custom programs to enhance flexibility and allow for a wide range of visual effects. Programs which execute a programmable section of a rendering pipeline are commonly referred to as *shaders* and are usually written in a specialized *shading language*. Graphics processing units (GPUs) are capable of running many shader instances in parallel to accelerate the rendering process. To utilize those computational capabilities in applications outside of computer graphics, general-purpose GPU (GPGPU) frameworks have been developed, such as OpenCL or NVidia CUDA.

Using these technologies, a GPGPU-based software rendering pipeline is being developed in an ongoing research project. The goal is to evaluate new rendering algorithms on real application data that was acquired by recording all calls to the Direct3D API. The graphics pipeline used by Direct3D comprises a combination of fixed-function and programmable stages (Blythe, 2006). GPUs often contain dedicated hardware to implement some of the fixed-function stages efficiently. Application programmers have limited control over those parts of the pipeline. Replacing the fixed-function stages with GPU-side programs in CUDA is a major objective of the research. However, for the correct interpretation of input data, the evaluation of programmable stages has become essential for state-of-the-art real-time graphics. In order to make an implementation compatible, shaders of the same format as used by Direct3D need to be executed. Features for interoperability with graphics APIs exist for CUDA (NVIDIA Corporation, 2017a) and OpenCL (The Khronos Group, 2010; Houston and Cameron, 2010). However, such features are currently limited to sharing resources between graphics and GPGPU implementations, but no executable code in either source or compiled binary form. For these reasons, a program to

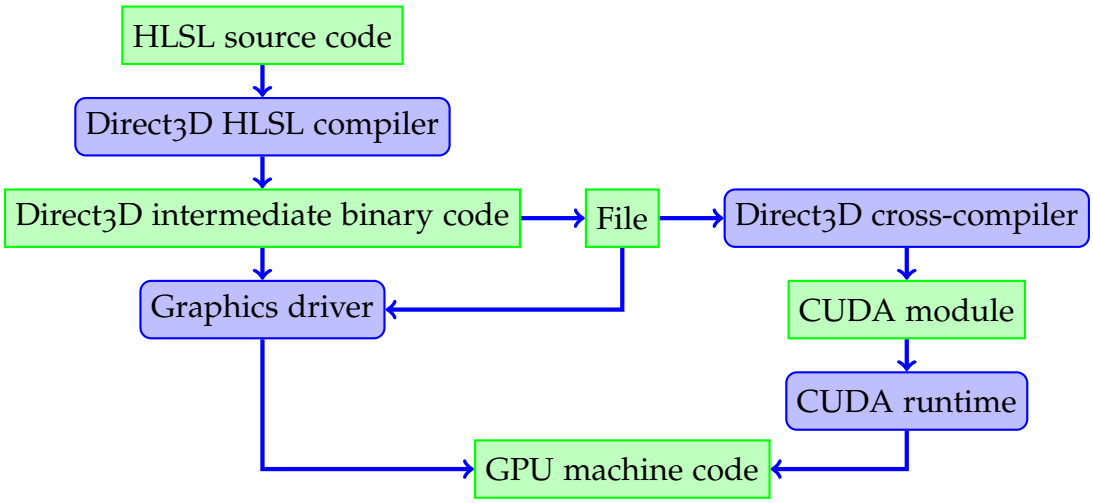


Figure 1.1.: Comparison of work flows for shader compilation

automatically compile Direct3D shaders to CUDA-compatible code has been developed in this master thesis. In the generated code, the functionality of a shader will be represented by a function performing the same task in CUDA [1]. As shown in figure 1.1, shaders are translated from an intermediate form instead of the original source code, which is not always included in applications using Direct3D.

1.1. Shaders in Computer Graphics

Frequently, controlling a rendering system through a set of configuration parameters is too restricting for a programmer or an artist. To overcome such limitations, the operation to be performed by the renderer may be specified using a programming language, instead of configuring fixed functionality. Depending on the context, different parts of the rendering process are feasible to be implemented using shaders. A very common application for shaders is to calculate the lighting or color of a point on a surface.

[1] An exception is the hull shader stage, where parts of the same shader may run in parallel

Shading languages often supply domain-specific features integrated into a general-purpose language syntax. For example, special data types for vectors and matrices are usually part of the language instead of being implemented as user-defined types. Texture samplers, images or atomic types are also often provided as built-in types. Syntactic constructs designed for interaction with fixed-function pipeline stages are common as well, such as syntax to specify the binary interface of a shader, or attributes containing meta-information that are interpreted by the renderer.

In the real-time computer graphics library OpenGL (Segal and Kurt Akeley, 2016), extensions supported vertex and fragment shaders written in an assembly language (Beretta, 2013; K. Akeley, 2002), as well as earlier and more limited forms of programmability, before the OpenGL Shading Language (GLSL; Kessenich, Baldwin, and Rost, (2016)) was introduced. Similarly, in Microsoft Direct3D shaders are programmed using the High Level Shading Language (HLSL). Cg (Mark, Glanville, et al., 2003) is another shading language intended for real-time rendering, developed by NVidia and Microsoft, which is intended to be compatible with both Direct3D and OpenGL. The Vulkan graphics API (The Khronos Vulkan Working Group, 2017) uses a lower-level intermediate representation for shaders instead of a specific programming language; shading languages are not part of its specification. This intermediate form, called Standard Portable Intermediate Representation (SPIR; Kessenich, Ourial, and Krisch, (2017)), is intended as a format for both shaders and compute kernels (see also 1.2). Other shading languages for non-real-time rendering exist as well, such as the Open Shading Language (Gritz, 2016) or the RenderMan Shading Language (Hanrahan and Lawson, 1990).

As of now, multiple types of shaders are used by the prominent real-time graphics libraries OpenGL and Direct3D, making a significant portion of their graphics pipelines programmable. Vertex shaders generally implement a transformative step for vertex data before it is assembled into primitives. Typical transformations include, for example, the conversion between coordinate systems, or a procedure required for skeletal animation called *skinning*.

Pixel shaders control the color of a rasterized fragment. However, pixel shader output is treated in an abstract manner and may, especially in the

case of multi-pass rendering, represent arbitrary data other than colors. Since the appearance of those two shader types, more recent developments have introduced additional types of shaders.

Geometry shaders are an optional pipeline stage that support programmatic access to geometric primitives, such as triangles or points, and can output one or more primitives per input primitive. The generation of new primitives has in many cases been superseded by the use of tessellation shaders.

Tessellation is programmable using two types of shaders, one to control how finely a patch should be subdivided along each edge and on the inside, and one to compute attributes of the vertices generated by the tessellator. In OpenGL, those are called *Tessellation Control Shaders* and *Tessellation Evaluation Shaders*, while in Direct3D the terms *Hull Shader* and *Domain Shader* are used, respectively.

Compute shaders are a step towards general-purpose computing on the GPU within a graphics API while not requiring an external library. The same shading language as for other shader types is used. However, there are some language extensions currently specific to compute shaders, such as shared memory and new synchronization primitives.

Although real-time graphics APIs intend shaders to be compiled for execution on a GPU, other implementations are conceivable. Reasons to have a shader implementation target the CPU, either by a compiler or interpreter, would be to remove the dependency on dedicated graphics hardware or to enhance the debugging experience.

While the OpenGL specification only defines the source language for shaders, Direct3D also specifies the intermediate representation HLSL programs will be compiled to, which graphics drivers then translate into executable code during runtime. The intermediate representation of Direct3D shaders is the input format for this master thesis' shader cross-compiler and will be described in more detail in chapter 3.

1.2. Shader Programming versus GPGPU Programming

By generalizing the programming model beyond the requirements of graphics programming, GPGPU libraries have introduced some differences to the conventional shader programming model, which will be investigated in this section. Terminology is different for GPGPU programming, since there is not necessarily a connection to computer graphics. Programs are consequently called *kernels* instead of *shaders*. Input and output data are accessed via arrays of arbitrary data types, residing in the main memory of a GPU. This stands in contrast to shaders that operate on data streams where elements represent vertices, control points, patches, primitives or fragments.

For shaders in a rendering pipeline, parallelism is implicit. Programmable stages are designed in a way that parallel execution is possible and graphics drivers will attempt to run multiple shaders instances in parallel, but the parallelization is not controlled by the programmer. Instead, drawing commands are issued by the user of a graphics API, and input data such as vertex attributes or interpolated fragment data is automatically assigned to shader instances. On the other hand, for compute kernels, a programmer explicitly invokes a number of instances (threads), which in case of CUDA or OpenCL kernels or compute shader instances are organized in a grid of up to 3 dimensions, that is, each thread has its own spatial index. Input and output data are passed from and to the kernels as an array containing all data, each thread then uses its coordinates within the grid to calculate the address for reading or writing data values.

Shading languages have been described in the previous sections. In case of OpenCL or CUDA, kernels are written in dialects of existing languages, such as C, C++ or FORTRAN. Compilation of kernels can occur ahead of time or at runtime when executing the host program (NVIDIA Corporation, 2017b; NVIDIA Corporation, 2017c). Additionally, CUDA makes use of a common intermediate language similar to assembly code, called "PTX" short for *parallel thread execution* (NVIDIA Corporation, 2017e). Both CUDA and OpenCL implementations provide a runtime library for compiling or loading kernels, managing memory and other objects. No dependency on

Method	Impl. Effort	Scalability	Ease of Use	Performance
Manual Translation	None	Bad	Bad	High
Graphics Interop.	Medium	Good	Medium	Low
GPU-based Interpreter	High	Good	Good	Low
Compile to CUDA	High	Good	Good	High

Table 1.1.: Overview of possible implementations

graphics libraries exist, although using both a graphics and a compute library in the same program is possible, with some interoperability features being provided by the compute API.

1.3. Possible Implementations

Several approaches to the portability problem have been considered and will be described in this section. It should also be discussed if the problem can be solved using libraries or other existing infrastructure. Graphics applications, especially game engines, frequently use a large number of different shaders or select between a variety of shading techniques in the same shader ("Uber-shaders"). Therefore, an important objective is that the solution scales with the number of shaders used, and also with increasing complexity of a single shader. Table 1.1 gives an overview of all considered approaches, as well as an assessment according to the applied selection criteria.

1.3.1. Manual Translation

For each Direct3D shader in use by the original application, a shading function is written by hand to reproduce the behavior of the shader. If

the HLSL source is not available, the shader has to be translated from its disassembly.

Effort Since no additional program or component has to be implemented, the up-front effort is low. However, it is proportional to the number of shaders required.

Advantages If the rendered scene is not complex, this approach is the most straight-forward.

Disadvantages The effort for rendering complex scenes is high. Additionally, the translation is prone to human errors.

Applicability Scalability and inflexibility are major drawbacks to this method. Other approaches are preferable for a general purpose software renderer.

1.3.2. CUDA Graphics Interoperability

CUDA is able to share resources such as buffers or textures with the graphics libraries OpenGL and Direct3D. However, shaders can still only be launched from Direct3D and use a different programming model, as described in section 1.2. In order to run a shader, a rendering call has to be issued. Direct3D only provides access to the output of shader stages through an optional stream-output stage or texture render targets (Microsoft Corporation, 2018). Pass-through shaders are required to obtain the results of other stages in the pipeline for use in CUDA.

Effort Routines to execute shader stages and obtain the results without other side-effects need to be created once. Pass-through shaders need to be written or automatically generated for every different shader signature (see also section 3.1).

Advantages Since Direct3D is used to run the shaders, correctness of shader execution is ensured by the graphics driver.

Disadvantages Some pipeline stages will be run with a pass-through configuration redundantly, possibly multiple times, to isolate the desired shader stage. Performance is likely to be negatively affected by this. Furthermore, a dependency on Direct3D is introduced by this method.

Applicability While the amount of manual work does not increase with complexity, the performance implications are a drawback. The Direct3D rendering pipeline is used in a way it is not intended for, leaving this approach to be considered more of a work-around than a solution.

1.3.3. GPU-side Interpreter

With the high degree of programming flexibility in modern GPUs, implementing an interpreter running as a compute kernel becomes possible. Some pre-processing might be required for the Direct3D intermediate binary format. Interpreting instructions introduces some additional data-dependent branching. However, threads in the same warp would interpret the same shader, making these conditional branches mostly uniform and not inherently introduce blocking times in addition to the interpreter overhead.

Effort The effort for implementation and testing is estimated to be about the same as implementing a compiler for the shaders.

Advantages With an interpreter, any shader can be used at runtime by the renderer. With a compiler, every shader that participates in producing a frame has to be identified and translated in preparation of the rendering process.

Disadvantages Despite most branching being uniform, the interpreter would cause considerable slowdown of shader execution.

Applicability This approach is feasible, but the practical advantages over using a compiler instead are unclear.

1.3.4. Shader to GPGPU Compiler

Shaders are automatically translated to one or more shading functions in a programming language usable within CUDA. In order to make the generated code compatible with C++, the most suitable target languages are C, C++ and PTX. Additionally, it is possible to generate code in the internal representation of NVidia's CUDA compiler, NVVM, which is derived from the intermediate representation of the LLVM compiler infrastructure

(NVIDIA Corporation, 2017d). A PTX back-end for NVVM code is available in the CUDA toolkit.

Effort Implementation and testing cause a considerable amount of effort up front, but allows the use of any shader in a CUDA-based renderer in an automated way afterwards.

Advantages The overhead compared to running the original shader is very low.

Disadvantages It is necessary to work around a few limitations of the CUDA API. Those limitations are described in more detail in chapter 5 and section 6.2.

Applicability Scalability and execution performance make this approach well suited for software rendering. Any of the mentioned target languages would suffice.

From the options presented, a compiler for automatic translation to CUDA has been chosen. Its strength lies in the run-time performance of ported shaders, since no inherent slow-down exists. This makes it possible to evaluate algorithms designed for interactive rendering speed. Taking the high scalability and simple workflow into account, it becomes the most viable option, despite the initial effort required for its implementation.

2. Related Work

Automatic code generation for shaders has been a subject of interest for some time, in order to make authoring of shaders easier and to make different shading languages or graphics libraries more interchangeable. This chapter will give an overview of other existing work on this subject. Furthermore, some related work has been done regarding the run-time analysis or emulation of shaders or compute kernels, with some similarities to this project.

2.1. Shader Compilers

In addition to the shading languages described in 1.1, several other GPU programming systems have been created to exploit the programmability of modern graphics processors. The Cg shading language (Mark, Glanville, et al., 2003) along with its host runtime library aims to unify GPU programming across different 3D graphics APIs and operating systems. Its implementation employs a compiler using assembly code as its target language, either off-line or during application runtime. While still targeting GPUs as a platform, the Cg compiler aims at different goals, and the generated code is similar to the Direct3D bytecode used as a source language here. Furthermore, Cg is no longer actively developed, and it is not possible for end users to extend the Cg compiler with new target languages.

In Mark and Proudfoot, (2001), a compiler for the Stanford Real-time Shading Language (Proudfoot et al., 2001), targeting low-level shader programming interfaces (the `NV_vertex_program` and `NV_register_combiners` OpenGL extensions) is described. Pipeline stages are not programmed directly, but instead vertex and fragment programs are derived from a single

shader that specifies the frequency at which values are computed. This master thesis, in comparison, intends to port shaders bound to a specific stage of the rendering pipeline to CUDA. Source and target languages are different as well.

Peercy et al., (2000) achieves programmable shading by translating shaders written in a newly developed shading language to a multi-pass rendering procedure in OpenGL, treating each pass as one or more primitive operations of the shader program. At the time of its publication, programmable shading was not widely available in real-time graphics APIs. While modern graphics processors offer a high degree of programming freedom that is exploited by HLSL and the Direct3D intermediate code, it is useful to keep in mind that shaders can also be mapped to other programming models, in case direct translation to GPU device code becomes difficult.

The PixelFlow system (Olano and Lastra, 1998) was one of the first programmable real-time renderers. Shaders are written in *pfman*, a programming language similar to the RenderMan shading language. It is implemented with an optimizing compiler generating code for a dedicated SIMD computer. The *pfman* compiler differs from this work in its specialization on the PixelFlow target platform and *pfman* shading language. Due to its publication time frame, rendering pipeline stages other than pixel shaders are not programmable.

An alternative shading language for the Direct3D API called Spark (Foley, 2012) aims at managing complex shaders through modularization. The per-stage programming model is abstracted so that effects involving multiple stages of the rendering pipeline can easily be added while keeping the implementation encapsulated and in one place. Spark source code is compiled to a set of HLSL shaders along with C++ classes for the client to configure the pipeline and initiate the rendering process.

As part of its Windows API compatibility layer, the WINE software package (Julliard, 2017) provides a compiler for Direct3D shaders that translates their binary form to GLSL. Some 3D graphics engines are capable of compiling shaders written in one shading language to another, in order to be compatible with many platforms. The Unreal Engine provides a library to translate HLSL shaders to GLSL (Epic Games, Inc., 2017). Unity can generate GLSL code from HLSL, Cg or Direct3D bytecode (Unity Technologies,

2017). While conceptually similar, those tools aim for compatibility with a different graphics API instead of porting shaders to a GPGPU platform. In Rhodin, 2010, a PTX back-end for an LLVM-based shader compiler for a custom shading language is developed. A shader compiler for functional programming was created by Elliott, (2004).

2.2. Code generators

Automatic generation of code for various shading languages has been employed frequently in tools that simplify the editing of shading functions. Various forms of visual shader editors exist already, commonly using different types of shade trees (Cook, 1984). For instance, Jensen et al., (2007) combines a shade tree editor with an optimizing Cg back-end. In McGuire et al., (2006), a program to create shade trees on a higher abstraction level is introduced. McCool et al. developed a C++ framework to perform algebraic operations on shaders, with the ability to generate shader code from recorded calls (M. McCool et al., 2004), based on the Sh shader metaprogramming language (M. D. McCool, Qin, and Popa, 2002). Similar to the program created for this master's thesis, those shader editing tools implement automatic generation of the shading code, but starting from a different shader representation. Before GPGPU frameworks were common, Brook for GPUs (Buck et al., 2004) provided a means to use a GPU for arbitrary parallel computations. It provides a compiler that translates kernels into shader programs and a runtime API to abstract the usage of a graphics library (Direct3D or OpenGL) to execute the shaders through a rendering call. An alternative CUDA module compiler targeting PTX assembly language was developed by Wu et al., (2016).

2.3. Debugging and Instrumentation Tools

For the purpose of debugging or software profiling, tools have been created which automatically manipulate shaders. Unlike this master's thesis, shaders are not translated to another target language, but the required parsing

and code generation poses some similarities. Strengert, Klein, and Ertl, (2007) developed a GLSL debugger that modifies shaders at source level, transparent to the user. The program "Total Recall" (Sharif and Lee, 2008) aids graphics debugging by instrumenting Direct3D shaders to record their behavior and subsequent emulation of specific shader execution instances on the CPU. In Duca et al., (2005), another debugger which uses automatic shader instrumentation has been created.

2.4. Software Renderering on GPUs

Existing software rendering pipelines are good examples of applications motivating the automatic translation of shaders. Implementing the entire renderer in software overcomes the limitations imposed on customizability by fixed-function parts of common real-time graphics libraries.

FreePipe (Liu et al., 2010) implements a fully programmable rendering pipeline in CUDA with a structure similar to OpenGL or Direct3D.

CUDARaster (Laine and Karras, 2011) implements a graphics pipeline in CUDA with emphasis on achieving high performance while following the same rules for rasterization and rendering order as fixed-function units of graphics processors.

Piko (Patney et al., 2015) is a framework to synthesize implementations of arbitrary graphics pipelines with programmable shading based on a description in a domain-specific programming language similar to C++. It uses LLVM to generate target code for CUDA GPUs or multi-core CPUs.

A complete implementation of a Direct3D driver needs to translate the shader binaries for its target architecture. This is true even if the API is implemented without a system-level component but entirely as a user-space library on top of CUDA. Such a system, intended to evaluate experimental rendering algorithms on frame data captured from real-world graphics applications, is the target platform of the translator developed in this master's thesis.

3. Direct3D Shader Format

Shaders for the Direct3D API are authored in HLSL and compiled to a hardware-abstracted low-level binary format by Microsoft's HLSL compiler. The result is a program in an assembly language for a virtual GPU, as described in section 3.2. In earlier versions of Direct3D, it was possible for programmers to write shaders in that assembly language, but that option has been removed for recent versions. Graphics drivers will perform the final translation step to machine code at application runtime. Since the binary format is hardware independent, and the behavior is defined explicitly, it is possible for applications to use only shader binaries during deployment, and not ship the HLSL source code. Consequently, it is necessary to use this binary format instead of HLSL as the input language for this work, in order to be compatible with all Direct3D shaders.

3.1. Structure

Direct3D shader binaries are organized into self-contained "chunks", each describing a set of information needed for loading the shader. One type of those chunks contains the encoded shader program, others are for example the signatures of the shader or debug information. Signatures tell a graphics driver how to link different shader stages together, and which input or output variables have *system value semantics*, that is they are generated or interpreted by fixed-function graphics pipeline stages. Another important chunk type describes the data layout and binding information for shader resources, such as constant buffers, textures, generic buffer types or unordered access views (UAVs). Figure 3.1 gives a schematic view of the structure of shader binaries, using a pixel shader as an example. The remainder of this section focuses mostly on the contents of the shader code chunk, since this

representation of the program serves as the compiler's source language. Information encoded by other chunks is used to derive the public interface to the generated code, for example the signature of the shading function.

3.2. Shader Models and Instruction Set

Direct3D defines *shader models* as an abstraction of graphics hardware to define required features of a GPU. They are used as a basis for the instruction set of Direct3D's intermediate code, an assembly language for a virtual graphics processor (Microsoft Corporation, 2017). The instruction set architecture can be classified as a load-store architecture with non-destructive operands. In general, instructions are orthogonal, except if the instruction operates on specific operand types, such as texturing instructions. One exception to this classification are constant buffer operands, which can be used as a source operand to almost every instruction.

Shader models use numbered registers as an abstraction of memory areas of real graphics hardware. Different types of operands, such as shader input and output or local variables are modeled as sets of registers, with each set using a separate numbering starting from 0. In most cases a register refers to a vector with 4 components, each 32 bits in size; however some special-purpose operand types are of different size. A register may also refer to an opaque type, such as textures, samplers, UAVs or shared memory buffers. All registers for opaque types have their own numbering as well. How the physical memory is allocated is not specified. Layout of the address space as well as ordering of concurrent memory accesses by shaders to shared resources is defined by the implementation.

Instruction opcodes and operands are encoded using any number of 32-bit tokens, therefore the size of each whole instruction is always a multiple of 4 bytes; however their length varies widely. Every instruction generally consist of an opcode comprising one or more tokens, in some cases encoding additional instruction-specific information in the form of attributes or flags. Such additional data encoded in the opcode is present mostly for a group of instructions that serve as variable or attribute declarations. The opcode is followed by any number of operands. Since operands can encode

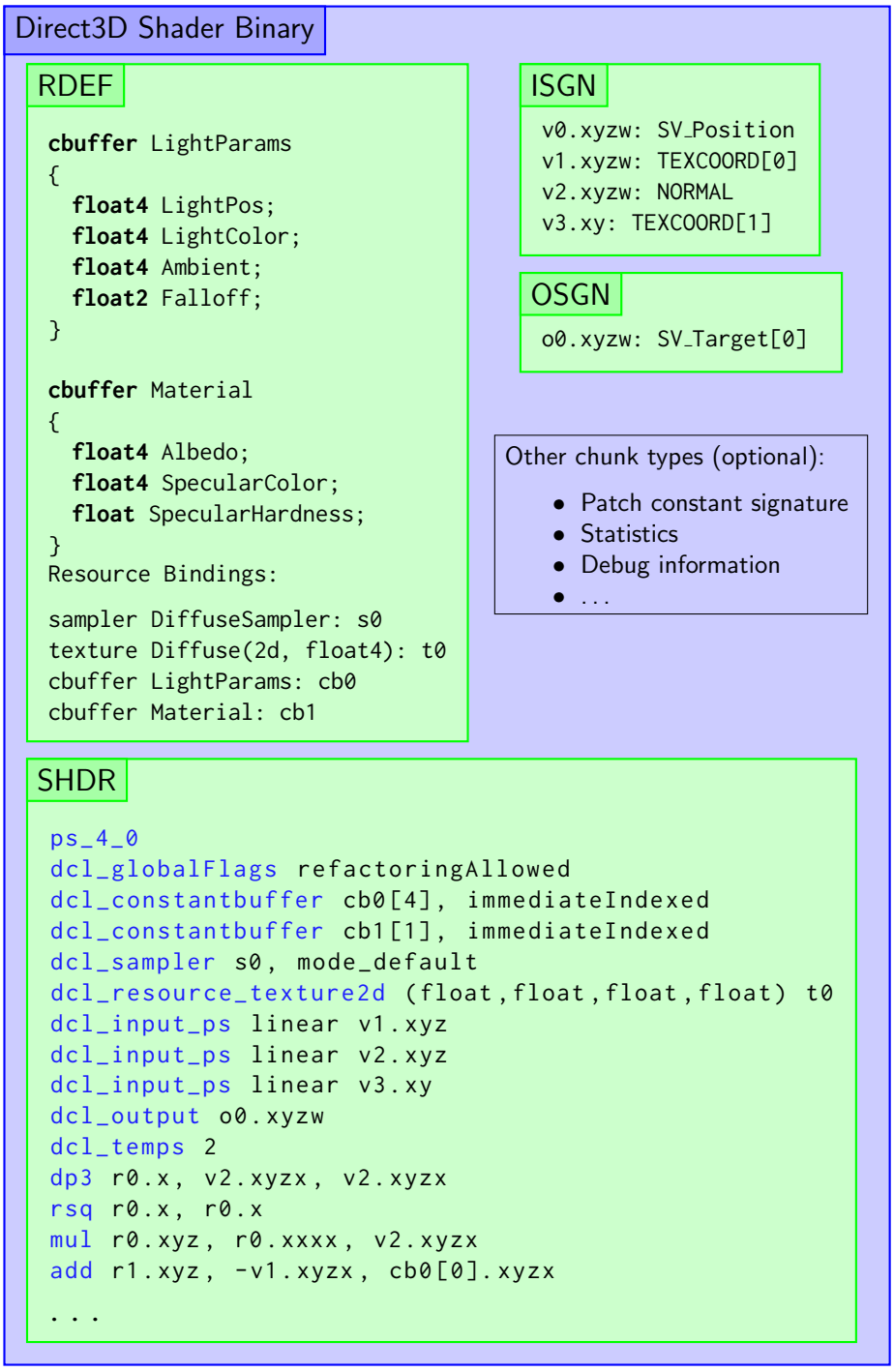


Figure 3.1.: Shader binary structure. Note that the code makes up only part of the file contents. Chunk types are use the same abbreviated names as in the binary file. RDEF: Resource definitions, ISGN: Input signature, OSGN: Output signature, SHDR: Shader code

dynamic indexing of arrays, nested operands are possible. In that case, the encoded dynamic index will come after the array operand in the shader binary. Operands can take many different forms in Direct3D shader binaries. Vector components can be write-masked for destination operands while for source operands, component swizzle or selection of a single component as a scalar operand is possible. Additionally, different array indexing modes are supported using either constant or variable offsets or a combination of the two. An optional modifier may be applied to the operand that calculate the absolute numeric value or invert its sign. Instead of registers, operands may also contain immediate data in the form of one or four 32-bit values. A visualization of the instruction encoding is given by figure 3.2, showcasing a multiplication instruction with a temporary register as destination, an input register and a constant buffer operand as sources. The use of an additional operand for dynamic indexing is also shown.

Most instructions can be categorized as arithmetic, declarations, memory accesses or data copy, control flow or application specific instructions. Typical integer and floating-point arithmetic instructions are provided, with an optional modifier to clamp the result to the range [0,1] existing for many floating-point operations. Control flow is implemented as structured programming constructs, unlike most assembly or machine languages using conditional and unconditional jumps. For example, instructions to mark the beginning and end of conditional, loop or switch statements exist. All data objects used by the program require a declaration instruction to be present at the start of the program. These declarations abstract the details of memory allocation and therefore have no similarity to non-virtual computer architectures. Memory that is shared between shader instances is accessed using memory load and store or texturing instructions, or atomic operations. Other application-specific instructions include thread and memory barriers, geometry shader operations to emit vertices and finish primitives, screen-space derivative approximation instructions, or discarding pixels.

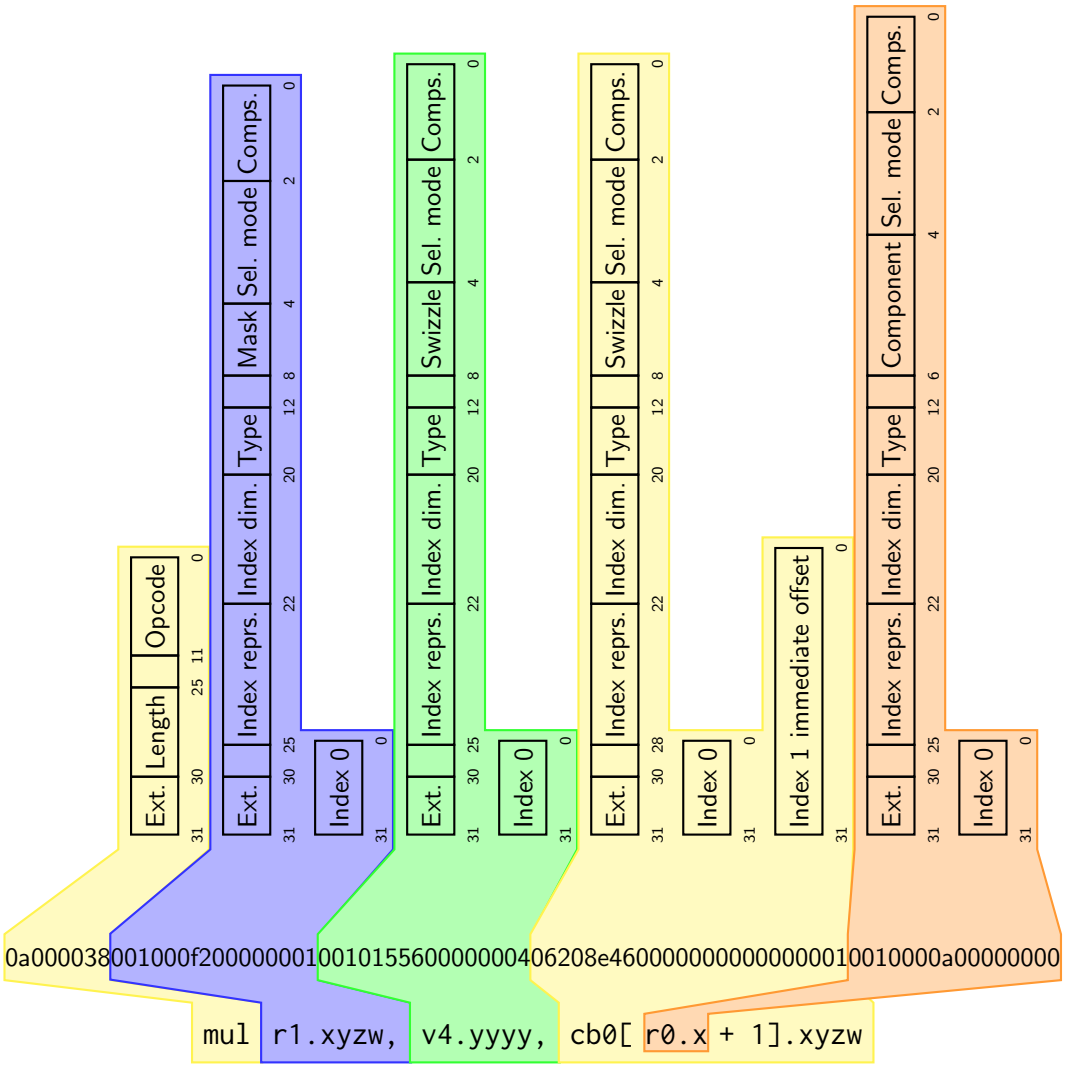


Figure 3.2.: Instruction encoding example. The instruction multiplies components of a dynamically indexed constant buffer vector with the y component of an input operand and stores the result in a temporary register.

3.3. Hull Shaders

Hull shaders are programmed as one pipeline stage, but independently compute output data for each control point as well as once per patch, called patch constant data. At minimum, the patch constant data contains all required tessellation factors for a patch. This computation of independent outputs allows for more parallelism than the amount exposed to shader programmers. Therefore, a hull shader contains multiple phases designed to possibly run with a different number of instances. A HLSL compiler can automatically generate the different phases from source code to maximize the amount of work that can be done in parallel (Ni et al., 2009, p. 24). Up to three different phases can exist in a single hull shader. The first phase is the control point phase, running once for each output control point. The other two phases are known as *fork phase* and *join phase*, and compute the patch constant data as well as tessellation factors along the edges and for the interior of a patch, respectively. All phases are contained in a single shader binary, with special instructions marking the start of each phase. In addition to phases, hull shaders have a declaration section for declarations common to phases, as well as setting up some configuration parameters for the tessellator.

4. System Design

Writing a compiler is generally a well understood task, for which many techniques have been thoroughly researched and are being applied in productive environments. In this case, the required source language and ensuing target languages somewhat reduce the implementation complexity. Most compilers are designed to accept a plain text source program written in a language described by a context-free or possibly even less restricted grammar. Working with a binary input format simplifies the parsing step, since no lexical analysis is required.

A three phase structure has been followed comprising a front end for parsing the shader binaries, back ends to generate PTX or C++ code or disassembly and a middle end operating on the internal representation of a shader. No optimization is performed when translating shaders, since the generated code will be subject to optimization done by the CUDA toolchain. In addition to translating the shader code, a C++ header declaring the interface to the shader is automatically derived for integrating the generated code in a CUDA application. Direct3D shader intermediate code does not carry explicit type information, but its instructions imply that their source and destination operands have a certain data type. Since both C++ and PTX as target language allow programming in a type-safe manner, type information is reconstructed based on the implied types of operands.

4.1. Program Structure

In the compiler's 3-phase implementation, the front-end is responsible for three related tasks. The chunks contained in a shader binary are parsed to obtain linkage and resource definition information, then the shader

program embedded inside a program is decoded according to the instruction encoding described in chapter 3. In the third place, while decoding the shader, a tree structure is created that serves as the internal representation of the input program. This tree form simplifies PTX code generation and construction of a control flow graph. Other than the shader code itself, important chunks are the shader signatures and resource definitions. The former define the binary layout of input and output variables as well as the per-patch data for tessellation shaders, while resource definition chunks describe the contents of constant buffers and other shader resources. In order to allocate those resources in generated code and to provide an interface to the graphics pipeline implementation, the binary parser needs to recognize the resource definitions. Similarly, shader signatures determine the set of input and output parameters that need to be generated for a function implementing a shader stage or hull shader phase.

Operations performed in the middle phase typically have no control dependencies between each other. Therefore, the middle phase is implemented as a series of passes sharing a common interface. Each pass generally carries out a code transformation or computation of additional information about the shader. The passes are optional and are enabled based on requirements of the selected back-end and program invocation parameters. Most configurations require a control-flow graph to be created. The middle phase is also capable of converting the source program into static single assignment (SSA) form. Reconstructing type information for local variables operates on the SSA representation of shaders.

Code generation back-ends have been implemented for C++ and PTX as target languages. The program is also capable of disassembling the shader binary without code generation. For generating code, a visitor pattern operating on the code tree produced by the parser is used. A more detailed description of the back-end implementation is given in chapter 5.

4.2. Phases

Different translation targets, the disassembler functionality and other user-specified options lead to a variety of required behaviors for the compiler.

The described structure of interchangeable modules is capable of adjusting to such variation. A driver class has been implemented that is responsible for instantiating a front-end and a back-end object and the required passes of the middle phase, and for guiding the compilation process.

4.2.1. Front End and Shader Code Representation

The frond-end for shader binaries has been separated into three components to break down the complexity of parsing the binary file format. First, low-level input methods are delegated to a separate class. Those input operations involve reading objects of primitive types, such as integers and character sequences, and following offsets for structures stored using pointer swizzling. The main parsing routines involve recognizing the different chunks of the program described earlier. Since the shader instruction encoding is essentially an additional binary format, a second class has been created for decoding the program. A hand-written recursive descent parser is used for both types of binary encodings found in shader files. Using a parser generator would be difficult with the use of pointer swizzling by the file format; furthermore, deriving the formal grammar of all chunk types and the instruction encoding would be a cumbersome task.

Both the instructions of a shader program and its tree structure are represented in the same class hierarchy. Most of those classes are an abstraction of a set of instructions which follow a common pattern, for example, a three operand instruction performing a binary operation or a texture or UAV memory access. Sequences of instructions without branching are contained by block objects, which share the same base class as the tree object hierarchy. The control flow instructions `if_z`, `if_nz`, `loop` and `switch` are implemented as subclasses that hold blocks of instructions the shader program may branch into, therefore forming a tree which resembles a syntax tree of a structured programming language. A visual example of such a tree can be found in figure 4.1. Since instruction operands have a complex structure, a self-contained operand class has been created instead of storing the encoding of operands directly within the instruction objects, therefore encapsulating the large number of possible operand forms and corresponding to the mostly orthogonal nature of the instruction set architecture.

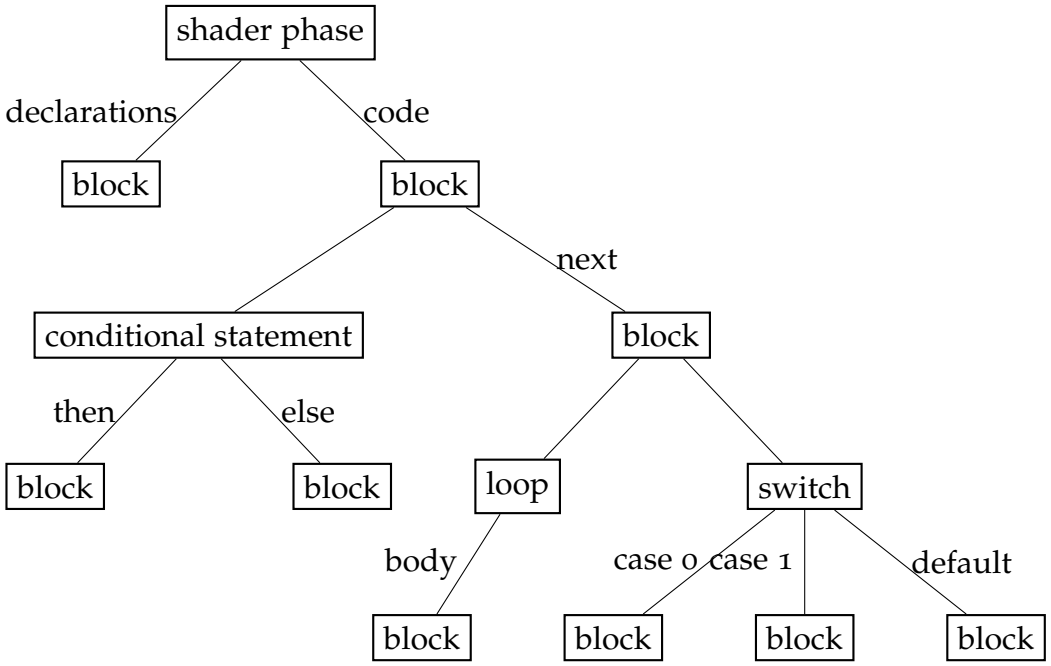


Figure 4.1.: An example tree created from shader code. Nodes labeled "block" are sequences of instruction objects. Non-branching instructions that may be part of blocks have been omitted.

4.2.2. Middle End

Converting the programs to SSA form is a central responsibility of the middle phase. For one of the approaches implemented to derive type information, SSA form is a prerequisite. Three transformation pass subclasses are implemented to build the control flow graph (CFG), convert the shader code to SSA form and finally SSA destruction. The term "SSA destruction" commonly refers to the operation that removes the Φ-functions and renames the arguments and return value of Φ-functions to a common variable name, which is required for code generation. Deriving type information from the shader binary also uses the same interface. During that pass, a table is generated to associate each component of a temporary register with a basic type, such as integers or floating-point numbers.

Depending on the selected texturing mode, an additional renaming step for texture and sampler objects may be required. CUDA supports two different modes of accessing textures. In what is called *independent* texturing mode, texture objects are separated from various other controls, such as filtering and addressing modes, which are held in a sampler object. Textures and samplers may then be combined arbitrarily for sampling textures. Direct3D and HLSL always use textures this way. The other texturing mode, used by default in CUDA, is called *unified* mode and causes each texture object to contain a single sampler object to use in sampling instructions. In order to generate code in unified texturing mode, the transformation pass needs to detect all texture and sampler uses and rename the objects to a common texture object also containing the corresponding sampler state.

4.2.3. Back End

Both code generation back-ends are designed in an almost identical manner, since the main difference lies in what code needs to be generated for each instruction. Consequently, it is possible to share parts of the implementation between the back-ends. Per target, a subclass of the back-end interface is responsible for generating file-level declarations and boilerplate code, as well as orchestrating code generation for each shader function and the hull shader phases. A header file is generated for the externally visible names

required to use the translated shader in a CUDA kernel. For generating code for a shader phase or function, both back-ends make use of a visitor class, which traverses the tree structure of the parsed code, and passes each instruction to a code generator class. A common base class is used for the code generators of both targets. An auxiliary class collects declaration instructions as they are visited, in order to simplify accessing the declaration object when referenced by an operand using its register number. The code generators each make use of another class that emits C++ statements or PTX assembly instructions, respectively. As a result, creating formatted textual output is separated from generating the declarations or statements.

4.3. Generating SSA Form

This section goes into detail about constructing a CFG for the shader and converting it to SSA. For the latter, a number of established algorithms exist that realize all necessary steps. Some considerations had to be made on how to integrate the changes from conversion to SSA into the existing shader code structures.

Some methods of generating the SSA form, including the chosen implementation, require a control flow graph to be created. The CFG is implemented as a class for storing per-node data, and another class to hold the set of node objects and implement the algorithms described in this section. A single CFG is generated per shader phase in hull shaders and for the main function otherwise. Since shader functions share registers with their caller and all call destinations can be statically determined, it is more sensible to insert edges to a function subgraph for calls than to create a different CFG for each function.

A rather simple algorithm to identify basic blocks and construct a CFG exists for code using only conditional and unconditional jump statements for control flow (Aho et al., 2007, pp. 525-516, 529). Such a format is common for intermediate code in compiler infrastructures. However, as mentioned previously, Direct3D shader code does not use jump instructions for control flow. To avoid creating a custom intermediate code format based on jumps, a different algorithm has been used to build the CFG.

The following control flow constructs are part of the shader intermediate code instruction set:

- Conditionals, using `if_z`, `if_nz`, `else` and `endif`
- Loops, using `loop` and `endloop`, as well as `break` and `continue` (all loops are terminated using `break`)
- Selection statements, using `switch`, `case` and `default`
- Function calls with late binding, using `call` and `fcall` (functions are denoted using a `label` instruction)

Constructing the CFG is essentially a depth-first search (DFS) on the code tree, with some additions to support the `break` and `continue` instructions present in some shaders. In order to set the correct nodes as successors, the

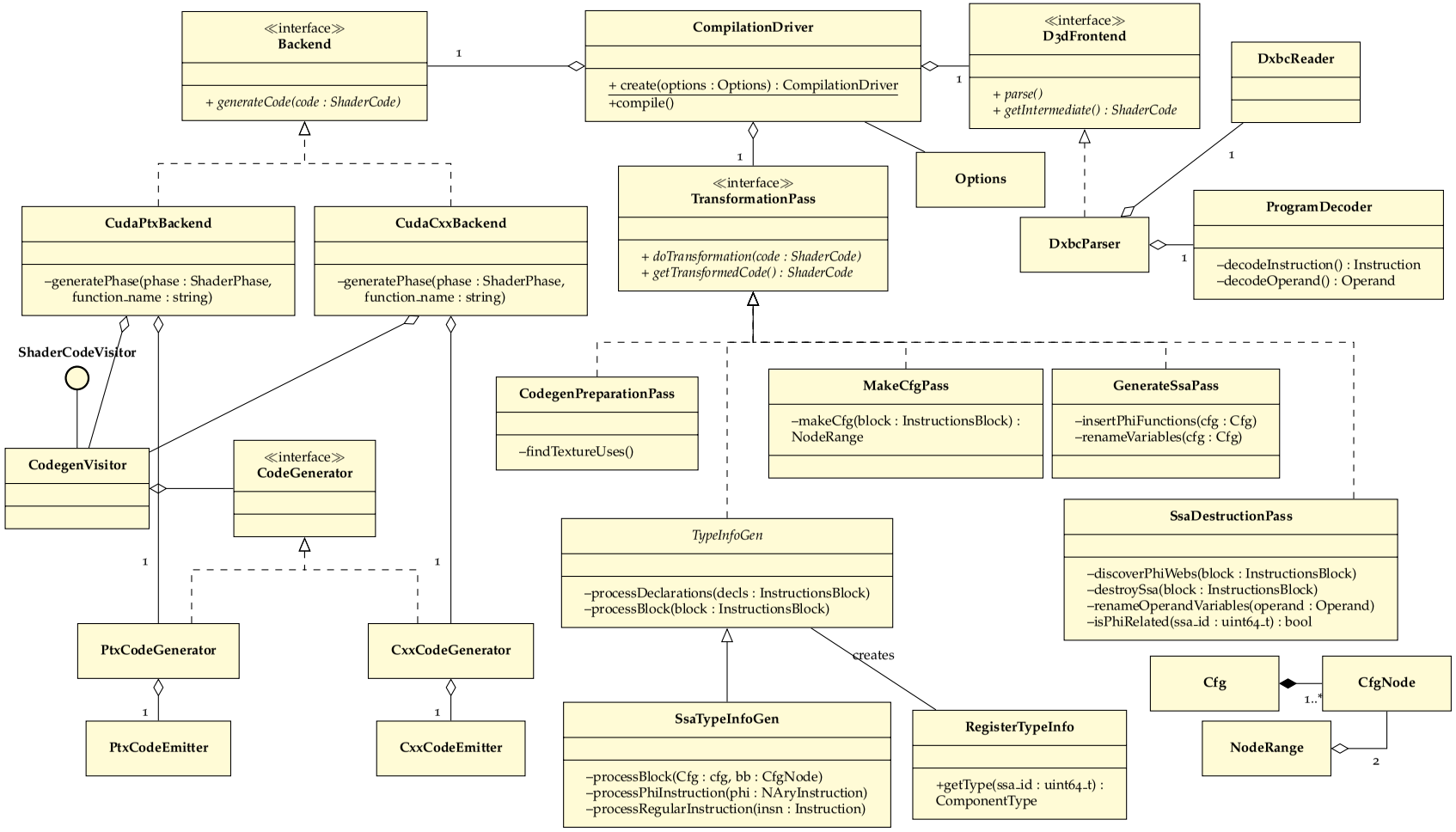


Figure 4.2.: Class diagram

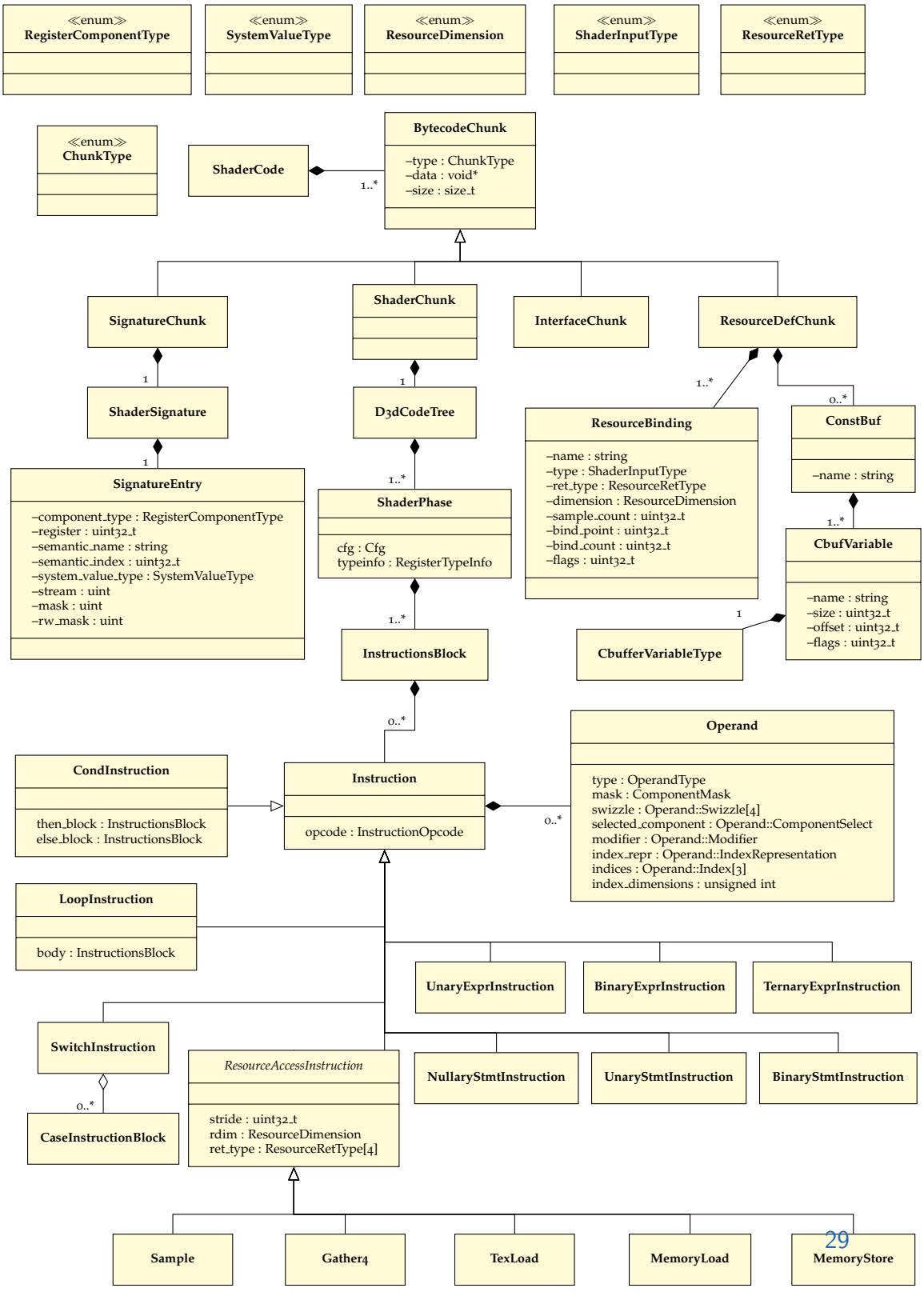


Figure 4.3.: Class diagram of data structures representing the shader code

current path of the DFS needs to be walked backwards to find the enclosing loop or switch block. The implementation uses a recursive function that generates the CFG for a branch of the program tree and, given as a block of instructions, returns a pair of references to the first and last node of the created CFG. After a recursive call for a branch, those nodes are required to correctly set successors of nodes depending on the control flow instruction that ends the given block. Figure 4.4 gives an example of this recursive construction leading up to the complete graph.

In the case of conditional statement instructions, the procedure is straightforward: the node created for the input block must have the first node of the *then* block as successor, and node for the block following the given one is set as successor of the *then* block's last node; the same is done for the *else* branch if it exists. For loop instructions, the first node of the loop body is a successor of the given block's node and also a successor of the loop body's last node. In a manner similar to conditionals, switch statement instructions add an edge from the current node to the first node for the block started by the case label, and another edge from its last block to the block after the given one. Loop bodies and case blocks may end early with a break instruction, creating another edge to the successor of the loop body's or case block's last node. Similarly, an edge to the loop body's first node must be inserted for blocks ending with continue instructions. Interface calls jump to a location only known at runtime. Since the caller and callee share the same set of temporary registers, each possible call target is added as a successor, and the return statements of those functions have the node of the caller's next block added as a successor. For the main routine, return instructions cause an exit node to be added as successor.

In the process of SSA generation, variables are given a unique name whenever appearing as a destination operand of an instruction. Moreover, Φ-functions are inserted for variables when a point is reached where multiple different definitions of that variable may have occurred based on the path taken at runtime. Unique variable names are usually created by adding a version number to each occurrence of a variable. For the operands of vectorized instructions, as used for shader binaries, each vector component of a register constitutes a variable name, operand objects containing one to four variable names as a result. A 64-bit integer is used to encode the register and component numbers with their version to form a unique name.

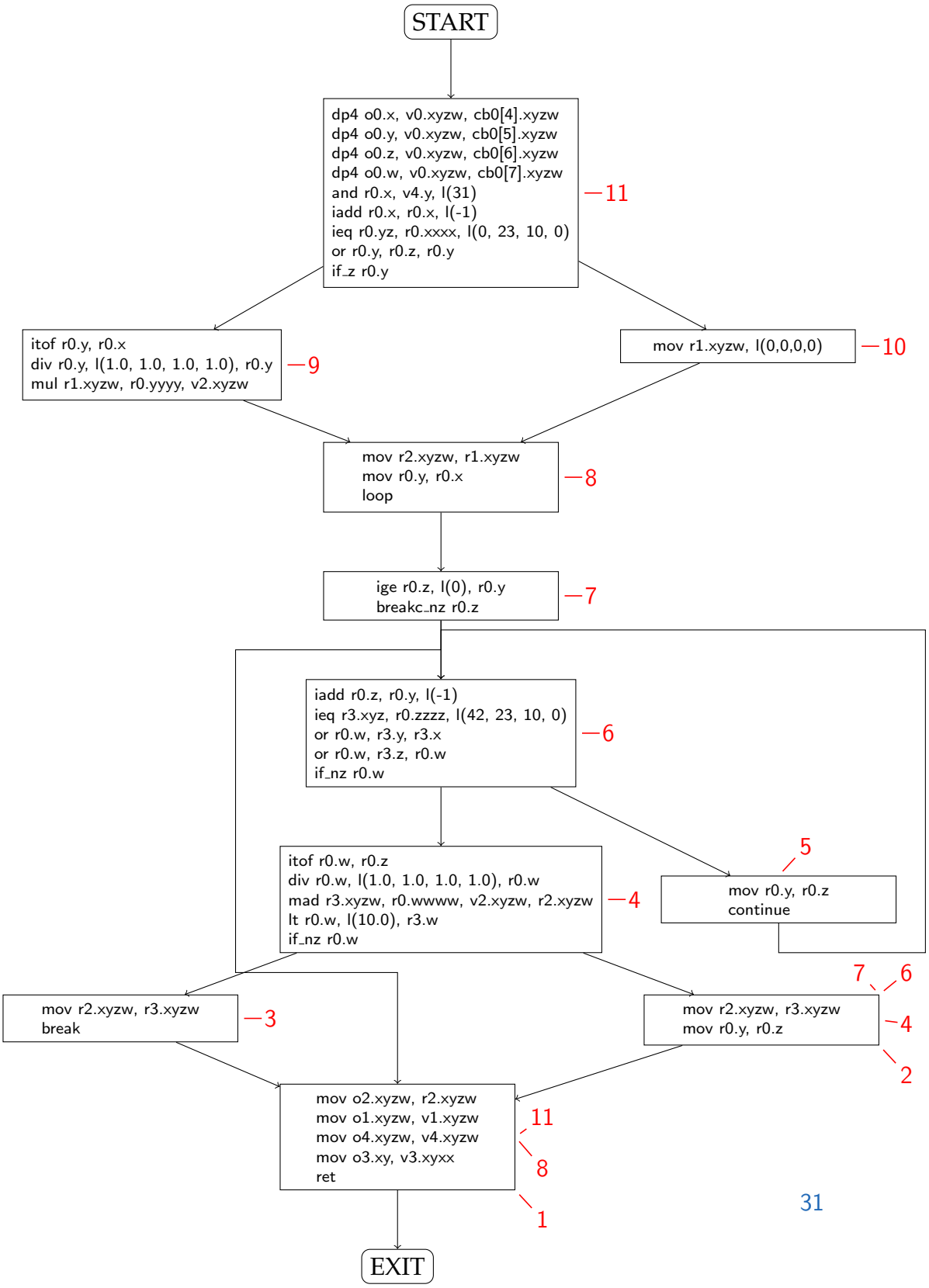


Figure 4.4.: A CFG created from one of the test cases, which uses a do-while loop and conditionals. The basic blocks are numbered to illustrate the intermediate results of the recursive calls.

Using a shader program from another test case, figure 4.5 illustrates the conversion to SSA form. A new instruction with an arbitrary number of operands is added to the shader instruction set to represent Φ-functions. Those instructions only exist during the middle phase of translation until they are removed by SSA destruction.

Standard algorithms have been used to implement the Φ-function insertion at the CFG's dominance frontiers and variable renaming, originally developed by Cytron et al., 1991, pp. 466,470,472. To compute the dominators of each CFG node, an efficient algorithm created by Cooper, Harvey, and Kennedy, 2001 is a common solution and was used in this work as well. As a somewhat special case, any dynamic array indices that may be present in an operand object need to be regarded as source operands as well during the renaming step. Since no optimizations were required, the conventional SSA form is retained throughout all phases. Therefore, the Φ-functions can be removed directly during SSA destruction.

4.4. Reconstructing Type Information

The PTX assembly language is an abstraction of graphics hardware that gives its users the ability to write programs in a type-safe way by declaring registers with a data type. This feature is optional; PTX also supports the declaration of untyped registers. C++ is naturally type-safe, a reinterpretation of variable contents as a different type has to be enforced using CUDA C++ language extensions. An attempt has been made to generate type-safe code for both languages, using typed variables and only forcing a conversion when the shader binaries reinterpret binary register contents through instructions expecting a certain type. This case occurs for example when a shader author uses the HLSL intrinsic functions `asfloat` or `asuint`, but is also sometimes generated by the HLSL compiler, as observed from manual inspection of shader disassembly.

Other type reconstruction tools have previously been developed to reverse engineer machine code with decompilers. Mycroft, (1999) converts target code in register-transfer (RTL) form to SSA and expresses the assignment of types to renamed variables as a solution to a system of type constraints

umin r0.xy$_{0,0}$, v4.yzyy, l(128,255)	$\{\texttt{r0.x}_0, \texttt{r0.y}_0\} \longleftarrow \min(\texttt{v4.xy}, \{128, 255\})$
and r0.z$_0$, v4.z, l(65535)	$\texttt{r0.z}_0 \longleftarrow \texttt{v4.z} \otimes 65535$
mov r1.xy$_{0,0}$, l(0,0,0,0)	$\{\texttt{r1.x}_0, \texttt{r1.y}_0\} \longleftarrow \{0, 0\}$
loop	loop 0:
___phi r1.y$_1$, r1.y$_0$, r1.y$_2$	$\texttt{r1.y}_1 \longleftarrow \Phi(\texttt{r1.y}_0, \texttt{r1.y}_2)$
___phi r1.x$_1$, r1.x$_0$, r1.x$_2$	$\texttt{r1.x}_1 \longleftarrow \Phi(\texttt{r1.x}_0, \texttt{r1.x}_2)$
___phi r0.x$_0$, r0.w$_0$, r0.x$_0$	$\texttt{r0.w}_0 \longleftarrow \Phi(\texttt{r0.w}_0, \texttt{r0.w}_1)$
uge r0.w$_1$, r1.y$_1$, r0.x$_0$	$\texttt{r0.w}_0 \longleftarrow \texttt{r1.y}_0 \geq \texttt{r0.x}_0$
breakc_nz r0.w$_1$	if $\texttt{r0.w}_0 \neq 0$ then exit loop 0
iadd r1.x$_2$, r0.z$_0$, r1.x$_1$	$\texttt{r1.x}_2 \longleftarrow \texttt{r0.z}_0 + \texttt{r1.x}_0$
iadd r1.y$_2$, r1.y$_1$, l(1)	$\texttt{r1.y}_2 \longleftarrow \texttt{r1.y}_0 + 1$
endloop	end loop 0
mov o4.y, r1.x$_1$	$\texttt{o4.y} \longleftarrow \texttt{r1.x}_1$
mov r1.xz$_{1,1}$, l(0,0,0,0)	$\{\texttt{r0.x}_1, \texttt{r0.z}_1\} \longleftarrow \{0, 0\}$
loop	loop 1:
___phi r0.w$_2$, r0.w$_1$, r0.w$_3$	$\texttt{r0.w}_2 \longleftarrow \Phi(\texttt{r0.w}_1, \texttt{r0.w}_3)$
___phi r0.z$_2$, r0.z$_1$, r0.z$_3$	$\texttt{r0.z}_2 \longleftarrow \Phi(\texttt{r0.z}_1, \texttt{r0.z}_3)$
___phi r0.x$_2$, r0.x$_1$, r0.x$_3$	$\texttt{r0.x}_2 \longleftarrow \Phi(\texttt{r0.x}_1, \texttt{r0.x}_3)$
uge r0.w$_3$, r0.z$_2$, r0.y$_0$	$\texttt{r0.w}_3 \longleftarrow \texttt{r0.z}_2 \geq \texttt{r0.y}_0$
breakc_nz r0.w$_3$	if $\texttt{r0.w}_3 \neq 0$ then exit loop 1
xor r0.x$_3$, r0.x$_2$, v4.w	$\texttt{r0.x}_3 \longleftarrow \texttt{r0.x}_2 \oplus \texttt{v4.w}$
iadd r0.z$_3$, r0.z$_2$, l(1)	$\texttt{r0.z}_3 \longleftarrow \texttt{r0.z}_2 + 1$
endloop	end loop 1
mov o4.z, r0.x$_2$	$\texttt{o4.z} \longleftarrow \texttt{r0.x}_2$
mov o0.xyzw, v0.xyzw	$\texttt{o0.xyzw} \longleftarrow \texttt{v0.xyzw}$
mov o1.xyzw, v1.xyzw	$\texttt{o1.xyzw} \longleftarrow \texttt{v1.xyzw}$
mov o2.xyzw, v2.xyzw	$\texttt{o2.xyzw} \longleftarrow \texttt{v2.xyzw}$
mov o4.xw, v4.xw	$\{\texttt{o4.x}, \texttt{o4.w}\} \longleftarrow \{\texttt{v4.x}, \texttt{v4.w}\}$
mov o3.xy, v3.xy	$\{\texttt{o3.x}, \texttt{o3.y}\} \longleftarrow \{\texttt{v3.x}, \texttt{v3.y}\}$
ret	return

Figure 4.5.: Left: Shader assembly code in SSA form. Versions of temporary variables (*single components* of registers) are annotated as subscripts in red. Right: Semantics of the shader code.

derived from the target code. Ambiguities are then resolved by user interaction. A similar system was developed by Robbins, King, and Schrijvers, (2016) to decompile target code for an abstracted version of x86 machine language to a type-safe dialect of the C programming language. It uses an automatic solver and a system of Horn clauses and constraint handling rules describing the relationship between source and target code to decompile programs.

Reconstructing type information for Direct3D shaders is simpler in comparison, as a result of limitations in HLSL. Its type system does not include pointer or reference types. User-defined structure types are possible if non-recursive, but will be split into variables of primitive types by the compiler when used outside constant buffers or UAVs. Memory accesses are not done directly, but through the special object types texture, constant buffer, UAV and shared memory, which view memory as arrays with one to four dimensions accessed by array indices or, in case of textures, a coordinate vector. It is easy to statically determine the types of texture elements and constant buffer variables. Contents of other memory objects are currently considered to have an unknown data type for simplicity. For these reasons, a simpler approach not involving automated theorem proving or constraint satisfaction is possible, with a fall-back to partially not type-safe code in case a conflict arises due to optimizations by the HLSL compiler.

From the typed instructions that Direct3D shader binaries are using, it is possible to derive the type of variables when the shader has been converted to SSA form. Each version of a temporary register, which has been assigned by SSA renaming, is associated with the type produced by the instruction that uses the particular version as a destination operand. In case of arithmetic instructions, this is a simple task, since those exist for all basic types and the type can be derived from the opcode as a result. The resulting type of instructions accessing resources is in some cases encoded within the instruction, but can also be detected using the resource definitions provided by the shader file. In 4.1, the generated type information is shown for some example instructions in SSA form.

Destination operand types of the data movement instructions `mov`, `movc` and `swapc` depend on the source operand. In simple cases, that type is already known when processing such an instruction. However, more complicated

Instruction	Type Information
`mov r0.xy`$_{0,0}$`, v1.xy`	*type*(`r0.x`$_0$) := *type of* `v1` *in input signature* *type*(`r0.y`$_0$) := *type of* `v1` *in input signature`
`iadd r1.w`$_1$`, r1.w`$_0$`, l(1)`	*type*(`r1.w`$_1$) := *int*
`sample r0.xyzw`$_{1,1,0,0}$`, r0.xyxx`$_{0,0,0,0}$`, t1.xyzw, s1`	*type*(`r0.x`$_1$) := T *type*(`r0.y`$_1$) := T *type*(`r0.z`$_0$) := T *type*(`r0.w`$_0$) := T

Table 4.1.: Type information derived from example instructions. T is the resource return type of `t1` (in shader model 4.0) or of the sample instruction (shader model 5.0+)

cases sometimes occur in shaders, for example a `mov` instruction using the result of a Φ-function which has been inserted at the start of a successor of a loop exit. Immediate-value operands have no known data type, instead a symbolic "immediate" type is propagated through all data movement instructions until the value is interpreted as a certain type by an instruction. When a Φ-instruction is encountered, the output operand's type must be able to hold all possible input operands. If all input types are equal, that type is used for the output operand, otherwise the type assigned to the Φ instruction's destination is a union. Immediate types encountered as Φ input types are interpreted as the common type of the other input operands, if it exists, otherwise default to unions. When a shader contains a loop, the type of an input operand to a Φ-instruction may depend on its output operand. In order to resolve potentially conflicting types as described, all input types to the Φ-function must be known.

Therefore, the result type of a Φ-function depends on all predecessors of the basic block being processed and possibly also their dominators. Finding an order to process basic blocks in so that the type of every operand can be determined when encountered is difficult, if not impossible for the general case. Instead, the order is not fixed, but basic blocks are processed in a queue. The queue is initialized with all basic blocks of the shader program, where the order can be arbitrary. If one or more operand types could not be

determined because of dependencies, the block is re-entered into the queue. In SSA form, a definition of a variable may not depend on itself, unless it is assigned the result of a Φ-function. This special case is handled, therefore the algorithm will always terminate.

During SSA destruction, a representative variable is used in place of a Φ-function so that it can be removed before code generation. Its destination operand and all source operands are replaced by the representative variable throughout the program. The type of this representative variable must be able to hold all the types of variables it replaces, which is solved by using the destination type of the Φ instruction.

5. Back-End Implementation

At the time of writing, CUDA does not support independent texturing mode, which would be equivalent to the use of texture objects in Direct3D. For this reason, code generators must be capable of generating code that uses unified texturing mode despite the convention of separating textures from sampler states. To accomplish this, a separate pass as has been implemented as described earlier, in which all uses of texture resources are detected and stored as a collection of texture-sampler pairs. For each unique pair of texture and sampler, a CUDA texture reference in unified mode is generated. Its name is derived from the texture and ampler registers used; for example, a unified texture `t0_s0` would be generated for a combination of texture and sampler using the respective registers `t0` and `s0`. Other shader resources translate to CUDA in very simple ways. Direct3D constant buffers are equivalent to constant memory in CUDA with an externally visible name. Buffer objects and read-write buffer objects, called buffer UAVs in shader assembly, are implemented as objects in the global state space. Texture UAVs, referred to as read-write textures in HLSL, are supported by CUDA using surfaces. Dynamic shader linking, using HLSL interfaces, is currently not implemented by the back-end. Each shader phase translates to a CUDA function defined as a static struct method. Input and output variables not used by the shader are included in the function parameter list regardless, so that the behavior of Direct3D shaders can be emulated, where shaders with matching signatures can always legally be combined in a pipeline.

5.1. Name mangling

In the generated PTX assembly code, the function names are created using *name mangling*, a process where symbol names are derived from the

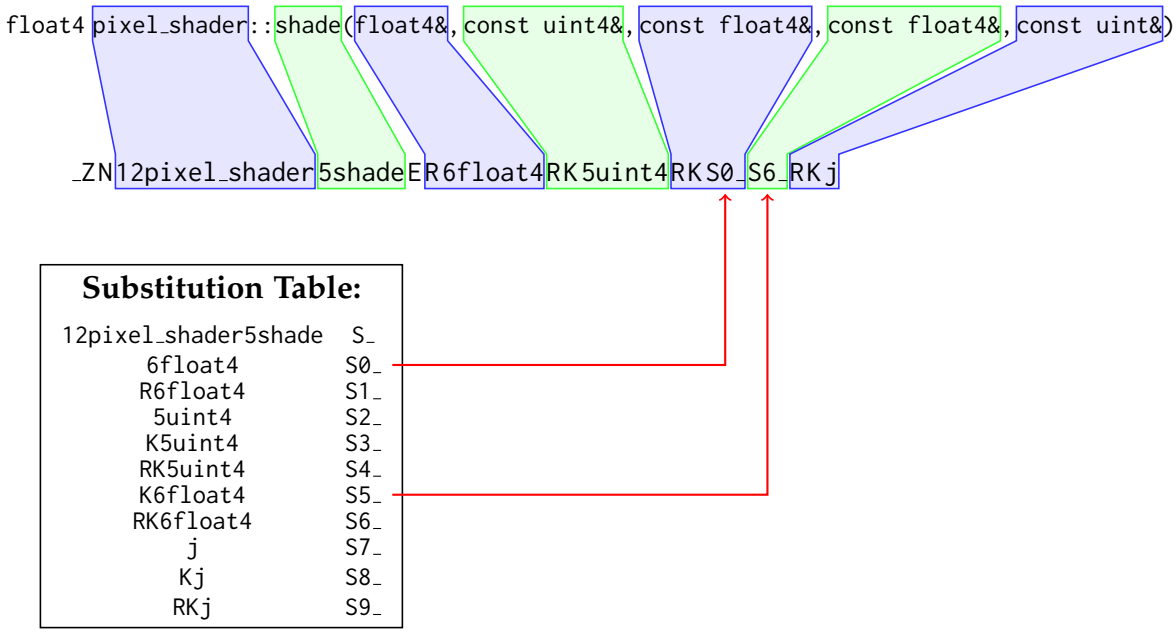


Figure 5.1.: Name mangling example

namespace, class name, method signature and template arguments of a C++ method or variable. C++ compilers use this process to generate unambiguous names for the linker, which has no understanding of context-dependent name lookups required by the C++ language. The PTX back-end has to implement name mangling as well, so that the generated code is compatible with the application binary interface (ABI) used by CUDA object files compiled from C++. In other words, it is necessary to use the same name mangling conventions as NVCC, which are specified by the Common Vendor ABI (CodeSourcery, 2017). Since the generated code does not make use of C++ templates or exception specifications, implementing a subset of the full name mangling scheme is sufficient.

Mangled C++ names using that ABI are always prefixed by `_Z`, creating a *reserved* name, to prevent name collisions. Nested names (not in global scope) are surrounded by the characters `N` and `E`. The name itself is formed by concatenating all enclosing namespaces, outermost to innermost, with the C++ identifier. For function names, the encoded function's signature

Shader Phase/Type	Input	Output
Default(main)	Input signature	Output signature
Control point phase	Input signature	Output signature
Fork/Join phase	Input signature	Patch constant signature
Domain shader	Input signature and patch constant signature	Output signature

Table 5.1.: Shader signatures for shader types and hull shader phases

is appended to the name for overload resolution. Since return types are not part of a function overload, they are ignored in name mangling. User-defined identifiers are prefixed by their length as a decimal number, so that concatenated names are unambiguous. Built-in types, operator names and the `std` namespace use special names without the length prefix. For example, the type `int` would be encoded as `i`, while the CUDA type `int4` would be encoded as `4int4`. Additionally, names are compressed with a forward substitution scheme that all user-defined identifiers participate in. A concrete example of name mangling, including the substitution scheme, is shown in figure 5.1 on the basis of a typical shading function.

5.2. Code Generator

The back-end class begins code generation by instantiating a code generator for the target language and a code visitor. Global declarations of a shader, for example constant buffers or common declarations of a hull shader, are generated first by dispatching the block of declaration instructions to the visitor. The shading function parameter list is created by joining the input and output parameter list obtained from following the rules in table 5.1.

After the function signature has been generated, the function body is dispatched to the visitor. For each class of instruction encountered, a call to

the code generator is issued to produce the statements implementing that instruction.

The translation of most instructions is self-contained. However, PTX uses different constructs for control flow than the Direct3D shader code, in the form of jump instructions and labels as well as conditional execution of instructions using predicate registers. Control flow cannot be handled within a single callback method invoked by the visitor. The PTX code generator has to keep track of the control flow constructs used so that jump instructions and the corresponding can be emitted at the correct place. A stack can be used for that purpose; each entry stands for one nesting level and contains the type of block (e.g. "then", "else", "loop") and labels marking the start and end of the block.

A common technique for syntax-directed translation of control flow constructs is shown in Aho et al., (2007, p. 402). The code generator implements control flow in a very similar way. For conditionals and loops, figure 5.2 shows the generated PTX code layout.

The Direct3D instruction `if_z` conditionally executes a sequence of instructions if a given operand is equal to 0, and optionally executes the sequence following a corresponding `else` until `endif` otherwise. It is implemented in PTX by comparing the translated operand (see 5.3) to 0 and setting a local predicate register depending on the result. A label to mark the end of the conditionally executed sequence (leading up to the `else` or `endif`) is generated and remembered (`L0`), as well as another label (`L2`) marking the start of the alternative sequence (between `else` and `endif`) if one is present. Based on the predicate register, a conditional jump is emitted, branching either to the remembered end label, or the start label of optional the alternative block. Following the conditional jump, the first sequence is translated. In case an `else` instruction is used, it will translate to an unconditional jump to the end label (`L0`), followed by the label marking the start of the alternative sequence. Finally the end label is emitted to complete the *if* construct.

In the other example, a loop is shown. In Direct3D, loops do not have a condition but instead execute until they are terminated by a `break` or `breakc_{c|nz}` instruction. This translates to the basic PTX code layout of a start label (`L1`) followed by the loop body, an unconditional jump back to the start label, followed by the end label (`L0`). Conditional break (`breakc`)

if_z *cond*
  Instruction sequence 1
else
  Instruction sequence 2
endif
...
	`.reg .pred` tmpreg_*i*;
	`setp.eq.b32` tmpreg_*i*, *cond*, 0x0;
	`@`tmpreg_*i* `bra` L1;
	  Code for sequence 1
	  `bra` L0;
	L1:
	  Code for sequence 2
	L0:
	...

loop
  Instruction sequence 1
  breakc_nz *cond-1*
  Instruction sequence 2
  continuec_z *cond-2*
  Instruction sequence 3
endloop
...
	L1:
	  Code for sequence 1
	  `.reg .pred` tmpreg_*i*;
	  `setp.ne.b32` tmpreg_*i*, *cond-1*, 0;
	  `@`tmpreg_*i* `bra` L0;
	  Code for sequence 2
	  `.reg .pred` tmpreg_*j*;
	  `setp.eq.b32` tmpreg_*j*, *cond-2*, 0;
	  `@`tmpreg_*j* `bra` L1;
	  Code for sequence 3;
	  `bra` L1;
	L0:
	...

Figure 5.2.: Translation of control-flow structures. Direct3D shader code is shown on the left, with the corresponding PTX sequence on the right. Placeholders are printed in italic shape.

instructions are implemented again with a predicate register that is assigned the result of a comparison to 0, and a conditional jump to the end label if the register holds *true*. Conditional continue (`continuec`) instructions are similar but jump to the start label instead of ending the loop. Unconditional `break` and `continue` instructions simply map to an unconditional jump to the respective label in PTX.

The complex operand encoding of Direct3D shaders does not map directly to operands of the PTX instruction set architecture, which in most cases allow just registers and immediate values. Operands located in memory are accessed with load and store instructions. Furthermore, PTX instructions are not vectorized except for data movement and memory access instructions, which can copy up to 128 bits (4 32-bit words) at once. For these reasons, the code generator has to extract the scalar operations from instructions that use vector operands. However, the PTX code generator makes use of the vector load/store operations when possible. In the C++ back-end, vector instructions are unpacked as well for simplicity. Additional operand modifiers and array indexing can be translated to C++ expressions. When generating PTX code, it is necessary to break up complex operands into multiple instructions, and using a computed value in a register as the operand to the actual instruction. See 5.3 for details.

Some input operands are not part of the input signature and only defined with declaration instructions. Those operands are domain point coordinates and instance identifiers of hull shader phases, geometry or compute shaders. For compatibility, those input operands are not generated as shading function parameters, but use an external device-side helper function to obtain the variable. Additionally, some output operands defined with system-value semantics are interpreted by some fixed-function parts of the graphics pipeline, in addition to being copied or interpolated. These system values are fragment depth (`SV_Depth`), its clamped variants `SV_DepthLessEqual` and `SV_DepthGreaterEqual`, multi-sample anti-aliasing (MSAA) coverage masks (`SV_Coverage`) and shader-specified stencil reference values (`SV_StencilRef`). Callbacks are generated when those output variables are written so that the software pipeline is able to treat those values accordingly.

5.3. Example: Translation of a complex instruction

The code generation process is further illustrated here by showing the translation of a single instruction step-by-step. The Direct3D instruction is:

```
mad o5.xyz, r0.wwww, cb[0][r0.x + 6].xyzx, r1.xyzx
```

Semantically, it denotes a vectorized fused multiply-add between temporary variables and a variable from a constant buffer, using source operand swizzles as well as a write mask to keep the first 3 (x, y and z) components of the result. The destination operand is an output register, and therefore passed to the shading function by reference. Since the PTX instruction set uses a load-store architecture, this means a store operation (into the `.global` memory space) needs to be emitted after evaluating the multiply-add expression. Therefore, the generated scalar PTX multiply-adds will use temporary registers as destinations, since they cannot access memory directly. Before generating the actual PTX instructions, the operand names are generated for each component. A setup sequence is emitted for complex operands that cannot be represented directly in a PTX instruction. Component selection or swizzle are also handled at this point. The temporary registers used for the memory write serve as the component names of the destination operand `o5.xyz`, resulting in the following code pattern:

```
.reg .f32 tmpreg_10_x, tmpreg_10_y, tmpreg_10_z;
Evaluate multiply-add, store result in {tmpreg_10_x, tmpreg_10_y, tmpreg_10_z}
st.global.v2.f32 [o5_xyz + 0], {tmpreg_10_x, tmpreg_10_y};
st.global.f32 [o5_xyz + 8], tmpreg_10_z;
```

As seen in the code listing, memory stores are coalesced into vector store instructions to write 64 or 128 bits simultaneously when possible.

The operand component names for `r0.wwww` and `r1.xyzx` are simply derived from their SSA names. Statically or dynamically indexed operands, such as constant buffers, require an operand setup sequence, since only load instructions in PTX can reference memory addresses. The required address arithmetic encoded in the operand also needs to be broken down into separate instructions. As a result, the operand `cb[0][r0.x + 6].xyzx` requires a setup

sequence. First the base address of the constant buffer `cb0` is loaded into a temporary register. Then, the index expression `r0.x + 6` is evaluated by translating the dynamic index operand `r0.x` to its SSA name, and emitting an addition. Since Direct3D uses a 128 bit (4 32-bit components) granularity for constant buffers and PTX uses byte addressing, the index expression also needs to be scaled by 16, resulting in the following setup sequence:

```
.reg .u64 tmpreg_11;
.reg .u64 tmpreg_12;
cvt.u64.s32 tmpreg_12, r0_0_x;
.reg .u64 tmpreg_13;
add.u64 tmpreg_13, tmpreg_12, 0x6;
.reg .u64 tmpreg_14;
.reg .u64 tmpreg_15;
mov.u64 tmpreg_15, cb0;
mul.lo.u64 tmpreg_14, tmpreg_13, 0x10;
add.u64 tmpreg_11, tmpreg_15, tmpreg_14;
.reg .f32 tmpreg_16_x, tmpreg_16_y, tmpreg_16_z, tmpreg_16_w;
ld.const.v4.f32 {tmpreg_16_x, tmpreg_16_y, tmpreg_16_z,
    tmpreg_16_w}, [tmpreg_11 + 0];
```

Finally, the multiply-add instructions themselves are emitted. Since the write mask disables the w component, only 3 scalar operations are performed.

```
mad.rn.ftz.f32 tmpreg_10_x, r0_1_w, tmpreg_16_x, r1_6_x;
mad.rn.ftz.f32 tmpreg_10_y, r0_1_w, tmpreg_16_y, r1_6_y;
mad.rn.ftz.f32 tmpreg_10_z, r0_1_w, tmpreg_16_z, r1_6_z;
```

These steps form the following combined PTX sequence for the Direct3D instruction:

```
/* mad o5.xyz, r0.wwww, cb[0][r0.x + 6].xyzx, r1.xyzx */
.reg .f32 tmpreg_10_x, tmpreg_10_y, tmpreg_10_z;
.reg .u64 tmpreg_11;
.reg .u64 tmpreg_12;
cvt.u64.s32 tmpreg_12, r0_0_x;
.reg .u64 tmpreg_13;
add.u64 tmpreg_13, tmpreg_12, 0x6;
.reg .u64 tmpreg_14;
.reg .u64 tmpreg_15;
mov.u64 tmpreg_15, cb0;
mul.lo.u64 tmpreg_14, tmpreg_13, 0x10;
add.u64 tmpreg_11, tmpreg_15, tmpreg_14;
.reg .f32 tmpreg_16_x, tmpreg_16_y, tmpreg_16_z, tmpreg_16_w;
```

```
ld.const.v4.f32 {tmpreg_16_x, tmpreg_16_y, tmpreg_16_z,
    tmpreg_16_w}, [tmpreg_11 + 0];
mad.rn.ftz.f32 tmpreg_10_x, r0_1_w, tmpreg_16_x, r1_6_x;
mad.rn.ftz.f32 tmpreg_10_y, r0_1_w, tmpreg_16_y, r1_6_y;
mad.rn.ftz.f32 tmpreg_10_z, r0_1_w, tmpreg_16_z, r1_6_z;
st.global.v2.f32 [o5_xyz + 0], {tmpreg_10_x, tmpreg_10_y};
st.global.f32 [o5_xyz + 8], tmpreg_10_z;
```

6. Results

In the previous chapters, notable implementation details of the shader translator have been discussed. This chapter focuses verifying the correct translation of shaders to the CUDA platform. A testing environment has been developed containing an application to execute translated shaders and compare the result against a Direct3D-based reference implementation. It also includes a rudimentary software renderer in order to execute pixel shaders under same conditions as within Direct3D.

6.1. Test Environment

For shaders tested through execution, a reference platform is required. A simple application using Direct3D to issue rendering calls and save the shader output by using either a stream-output or render-to-texture is sufficient. The result is a binary file containing shaded vertex data or an image for each vertex or pixel shader, respectively. In the testbed, those generated files are compared against the result of running the cross-compiled versions of the same shader binaries using CUDA.

Automatic execution of translated shaders for efficient testing follows a set of static testcase descriptions. Each test case contains a graphics pipeline configuration together with the shaders used, as well as input vertex data and required shader resource data, and the reference output. A shader configuration referenced by one or more test cases also includes the shader's signature, which is required to set up the input parameters and run the shader. Since the generated shaders do not form complete CUDA modules, they are linked at runtime against a small compute kernel that retrieves the parameters, packs them into vector objects and calls the shading function,

then writes the results back to output buffer parameters. Parameter retrieval depends on the shader signature, therefore a different piece of device code is used for every unique combination of input and output signatures that is referenced by a shader. It is possible to generate that code from the shader signature descriptors for run-time compilation in case the number of different signatures increases significantly. Currently, code generation is not necessary here because the shader signatures are kept consistent across the different test cases.

However, not all shaders are easily testable without replicating large parts of the Direct3D graphics pipeline, which would go beyond the scope of this work. For that reason, shader types other than vertex and pixel shaders are currently not used for testing. In the unified shader models of newer versions of the Direct3D API, a large part of the available features can be tested using those types of shaders. The other types differ mostly in their interface to the graphics pipeline, and are compiled but not executed for testing.

A basic rasterizer has been implemented so that pixel shaders can be executed and to render simple images in software. It uses a tile-based approach with two levels, where the viewport is divided into 8 by 8 pixel tiles for a coarse-grained coverage test on the CPU. If a tile contains pixels covered by the triangle being rasterized, a CUDA kernel is launched with one thread for each pixel in the tile. On the GPU side, a per-pixel test for inclusion is performed using a top-left fill convention. The pixel shading function of the translated shader is then called for all pixels inside the triangle. For test configurations where no pixel shader is set, the vertex shader output is simply downloaded from the CUDA device and compared against the reference data. Otherwise, the rendered image is compared against the result of the reference platform, and optionally displayed in the output window.

6.2. Systematic Testing

The following functionality is tested by compiling and executing the shaders, since they are ubiquitous in shader-based computer graphics applications:

- Vector and matrix arithmetic
- If-else, while, do-while and switch statements, including nested variants
- Bit-field manipulations
- Binary re-interpretation using the `asuint` and `asfloat` HLSL intrinsic functions
- Half-precision variables
- Common mathematical functions (square root, exponentiation, logarithm, trigonometry) and checking for infinity/NaN values
- Texture sampling, texel fetching and texture gather operations

Additionally, the following test cases are checked for successful compilation:

- Read and write of system values which make use of call-back functions as described in chapter 5
- Use of buffer and texture UAVs
- Some tessellation and geometry shaders

Implementing and using the testing infrastructure revealed some discrepancies between the capabilities of Direct3D and CUDA. Despite being documented, independent texturing mode is not supported by the CUDA API at the time of writing. This issue is currently avoided by resorting to unified texturing mode, although it is not the exact equivalent of what Direct3D uses. Similarly, depth textures cannot be used with comparison filtering. A reference depth value can be specified in texturing instructions, but has no effect at runtime. CUDA surface objects are functionally almost equivalent to texture UAVs, but do not support all of their associated operations. Some of the atomic operations that Direct3D permits on texture UAVs are not accessible to programmers using CUDA surfaces and have therefore not been implemented (on buffer UAVs, all atomic operations are supported).

When executing tests for the features listed above, the produced results have been verified to be correct using the Direct3D-based reference application. In case of shaders that could not be executed in the current testing environment, translation was successful, and the generated code for the instructions tested has been checked manually. Two infrequently used features of HLSL, double-precision support and dynamic binding using interfaces, are not supported yet.

6.3. Test Renderings

In the previous section, an assortment of test cases have been used to verify the correct translation at instruction level. Running the generated code in an almost isolated way is rather unrelated to typical shader usage in graphics pipelines. For this reason, some renderings of models have been produced to test the integration of translated CUDA shaders in a rendering process. Based on the rasterizer described before, the testing environment has been extended with a simple software renderer. Using the existing facilities for loading and executing CUDA kernels, a basic rendering pipeline is set up, implementing programmable vertex and pixel stages. In combination with the developed cross-compiler, Direct3D vertex and pixel shaders can be used in the rendering process without the help of a graphics library. However, a separate Direct3D-based renderer has been used to produce reference images for comparing the obtained results.

6.3.1. Detailed Example

The complete process from translation to a rendered image is shown here at the hands of a diffuse reflectance pixel shader. Listing 6.1 shows the HLSL source code; the vertex shader has been omitted for brevity throughout this example.

```
float4 main(PsInput input)
{
    float4 output;
    float4 albedo = DiffuseTexture.Sample(DiffuseSampler, input.texcoord);
    float3 lightdir = LightPos.xyz - input.cameraSpacePosition.xyz;
    float n_dot_l = dot(normalize(lightdir), normalize(input.normal.xyz));
    output = albedo * saturate(n_dot_l) * LightColor;
    output.xyz = saturate(output.xyz);
    output.w = 1.0f;
    return output;
}
```

Listing 6.1: HLSL pixel shader

Translation by the Direct3D HLSL compiler yields the following intermediate code, in disassembly view:

```
ps_5_0
dcl_globalFlags refactoringAllowed
dcl_constantbuffer CB0[2], immediateIndexed
dcl_sampler s0, mode_default
dcl_resource_texture2d (float,float,float,float) t0
dcl_input_ps linear v1.xyz
dcl_input_ps linear v2.xyz
dcl_input_ps linear v3.xy
dcl_output o0.xyzw
dcl_temps 2
add r0.xyz, -v1.xyzx, cb0[0].xyzx
dp3 r0.w, r0.xyzx, r0.xyzx
rsq r0.w, r0.w
mul r0.xyz, r0.wwww, r0.xyzx
dp3 r0.w, v2.xyzx, v2.xyzx
rsq r0.w, r0.w
mul r1.xyz, r0.wwww, v2.xyzx
dp3_sat r0.x, r0.xyzx, r1.xyzx
sample_indexable(texture2d)(float,float,float,float) r0.yzw, v3.xyxx, t0.wxyz, s0
mul r0.xyz, r0.xxxx, r0.yzwy
mul_sat o0.xyz, r0.xyzx, cb0[1].xyzx
mov o0.w, l(1.000000)
ret
```

Listing 6.2: Pixel shader assembly

From the intermediate code, a shading function in PTX assembly is generated for rendering using CUDA. The pixel shader code is shown in abbreviated form in listing 6.3, while the complete code can be found in appendix C. In the pixel shader listing, usage of Direct3D textures and samplers has been converted to use CUDA unified texturing mode in order to circumvent the problems described in 4.2.2 and 5. Effectively, a texture image and sampler parameters are set for a single CUDA texture object which is then bound to a unified texture generated for that combination of texture and sampler.

```
/* dcl_constantBuffer cb[0][2].xyzw, immediateIndexed */
.const .v4.b32 cb0[2];
/* dcl_resource t0, Texture2D, (float,float,float,float); dcl_sampler, s0, default (unified) */
.global .texref t0_s0;

.visible.func (.align 16 .param.b8 retval[16])
  ↪ _ZN12pixel_shader5shadeER6float4RKS0_S3_S3_RK6float2 (.param.b64 o0_SV_TARGET0, .param.b64
  ↪ v0_SV_POSITION0, .param.b64 v1_TEXCOORD0, .param.b64 v2_NORMAL0, .param.b64 v3_TEXCOORD1)
{
  /* (...) */
  /* add r0.xyz, -v1.xyzx, cb[0][0].xyzx */
  .reg .u64 tmpreg_7, tmpreg_8, tmpreg_9;
  mov.u64 tmpreg_9, cb0;
  mul.lo.u64 tmpreg_8, 0x0, 0x10;
  add.u64 tmpreg_7, tmpreg_9, tmpreg_8;
```

```
  .reg .f32 tmpreg_10_x, tmpreg_10_y, tmpreg_10_z, tmpreg_10_w;
  ld.const.v4.f32 {tmpreg_10_x, tmpreg_10_y, tmpreg_10_z, tmpreg_10_w}, [tmpreg_7 + 0];
  sub.ftz.f32 r0_0_x, tmpreg_10_x, v1_x;
  sub.ftz.f32 r0_0_y, tmpreg_10_y, v1_y;
  sub.ftz.f32 r0_0_z, tmpreg_10_z, v1_z;
  /* (...) */
  /* sample r0.yzw, v3.xyxx, t0.wxyz, s0 */
  .reg .f32 tmpreg_11_x, tmpreg_11_y, tmpreg_11_z, tmpreg_11_w;
  call.uni (tmpreg_11_x), d3dxc_dFdx_fine, (v3_x);
  call.uni (tmpreg_11_y), d3dxc_dFdx_fine, (v3_y);
  .reg .f32 tmpreg_12_x, tmpreg_12_y, tmpreg_12_z, tmpreg_12_w;
  call.uni (tmpreg_12_x), d3dxc_dFdy_fine, (v3_x);
  call.uni (tmpreg_12_y), d3dxc_dFdy_fine, (v3_y);
  .reg .f32 tmpreg_13_x, tmpreg_13_y, tmpreg_13_z, tmpreg_13_w;
  tex.grad.2d.v4.f32.f32 {tmpreg_13_x, tmpreg_13_y, tmpreg_13_z, tmpreg_13_w}, [t0_s0, {v3_x,
    ↪ v3_y}], {tmpreg_11_x, tmpreg_11_y}, {tmpreg_12_x, tmpreg_12_y};
  mov.f32 r0_2_y, tmpreg_13_x;
  mov.f32 r0_2_z, tmpreg_13_y;
  mov.f32 r0_4_w, tmpreg_13_z;
  /* mul r0.xyz, r0.xxxx, r0.yzwy */
  mul.ftz.f32 r0_3_x, r0_2_x, r0_2_y;
  mul.ftz.f32 r0_3_y, r0_2_x, r0_2_z;
  mul.ftz.f32 r0_3_z, r0_2_x, r0_4_w;
  /* (...) */
  ret;
}
```

Listing 6.3: Pixel shader PTX code (excerpt)

For interfacing with the other rendering code, a C++ header as shown in listing 6.4 is generated together with the PTX code.

```
#include "shader_interface.h"

extern "C" {
  static const uint32_t global_flags = GLOBAL_FLAG_REFACTORING_ALLOWED;
  __constant__ unsigned char cb0[32];
  extern texture<float4, cudaTextureType2D, cudaReadModeElementType> t0_s0;
  static const unsigned int num_clip_distances = 0u;
  static const unsigned int num_clip_distances = 0u;
}

struct pixel_shader
{
  __device__ static float4 shade(float4&, const float4&, const float4&, const float4&, const
    ↪ float2&);
};
```

Listing 6.4: C++ header for pixel shader

Using the generated PTX shader, the image in figure 6.1 has been rendered.

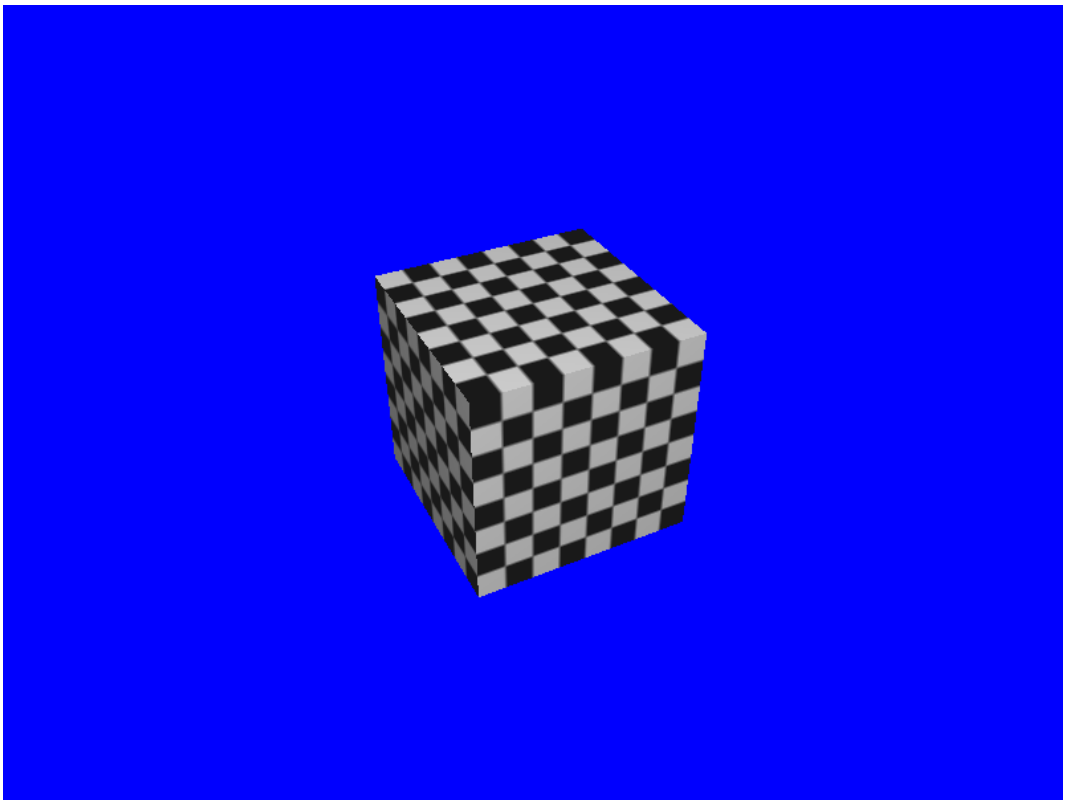


Figure 6.1.: An example using texturing and diffuse lighting

6.3.2. Other Renderings

A few other renderings have been produced to show practical usage of the developed cross-compiler. Due to the simplified rendering pipeline of the test environment, rather basic graphical effects had to be used.

The same test models have also been rendered using Direct3D with the same settings and the source shaders of the examples. A 800 by 600 viewport has been used in both cases. A visual comparison between the result of the testing environment's software renderer and the Direct3D implementation is shown in figures 6.5, 6.6, 6.7 and 6.8.

Table 6.1 gives a summary of a comparison with the reference implementation. The relative error is computed per pixel and color channel as the

Example	Different pixels	Total difference	Total relative error	Avg. relative error
Teapot (specular)	10	15	0.219742	0.0219742
Bunny (diffuse)	17	51	0.398571	0.0234454
Cube (texture)	9	27	0.328611	0.0365123
Rock (normal map)	9424	18963	259.232	0.0275076

Table 6.1.: Summary of differences compared to the Direct3D-based reference implementation. Columns from left to right: Name, number of different pixels, sum of intensity differences of all channels, sum of relative errors (difference divided by pixel intensity) over all channels, relative error averaged over different pixels.

absolute value of the difference divided by the intensity of the reference output. For the average relative error, that number is divided by the number of different pixels. Total difference is simply the sum of all absolute differences over all channels and pixels. The remaining deviations are believed to be due to numeric differences in the interpolation of pixel shader input parameters between the software rasterizer and Direct3D.

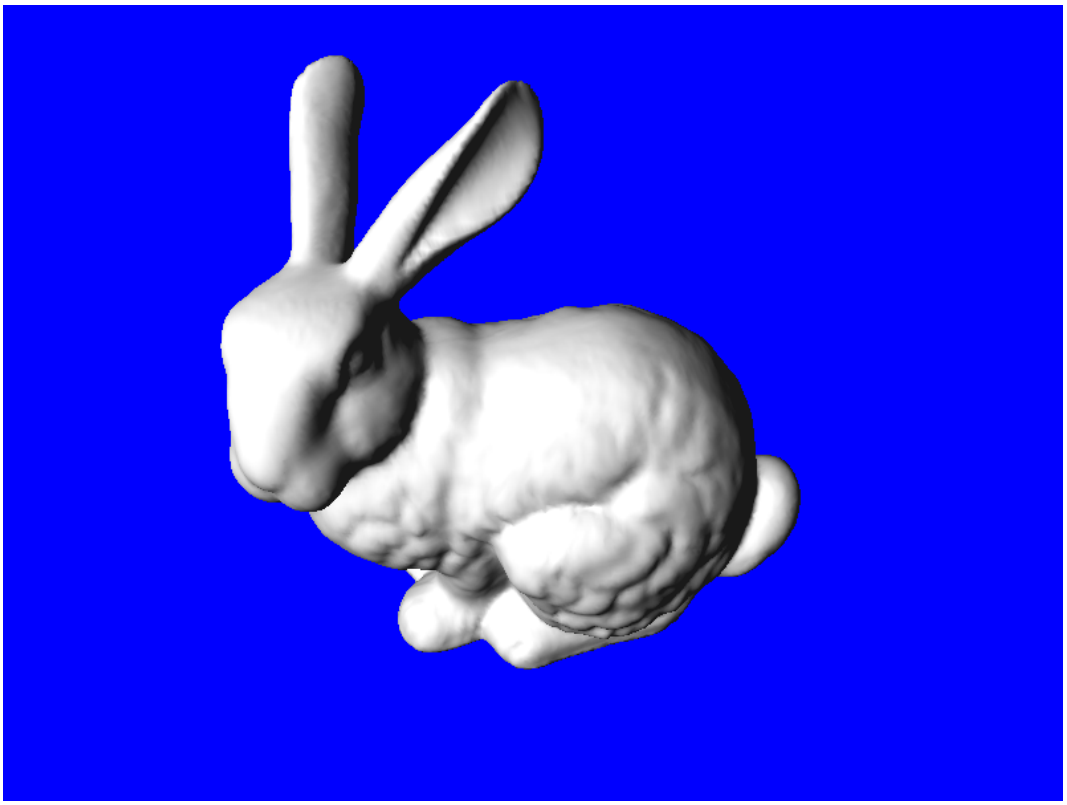


Figure 6.2.: A diffuse lighting example. Test model by Batty, (2001), a watertight version of the "Stanford Bunny".

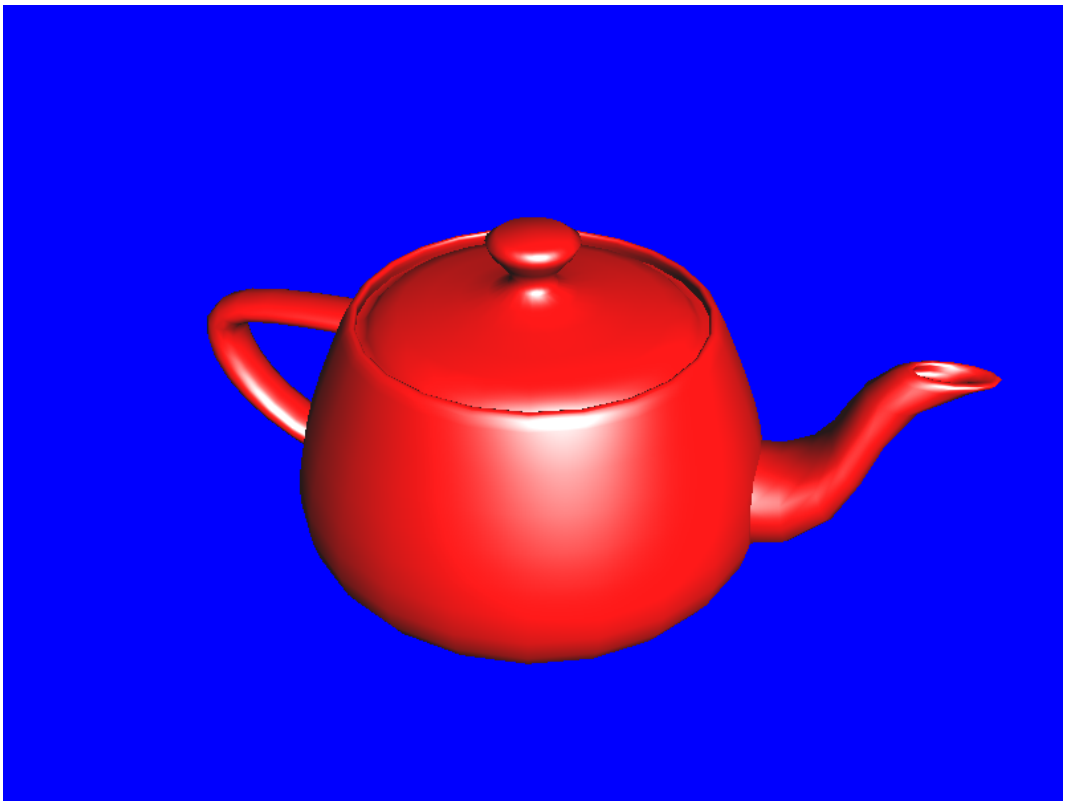


Figure 6.3.: A specular lighting example. The version of the teapot model used was created by Knowles, (2017).

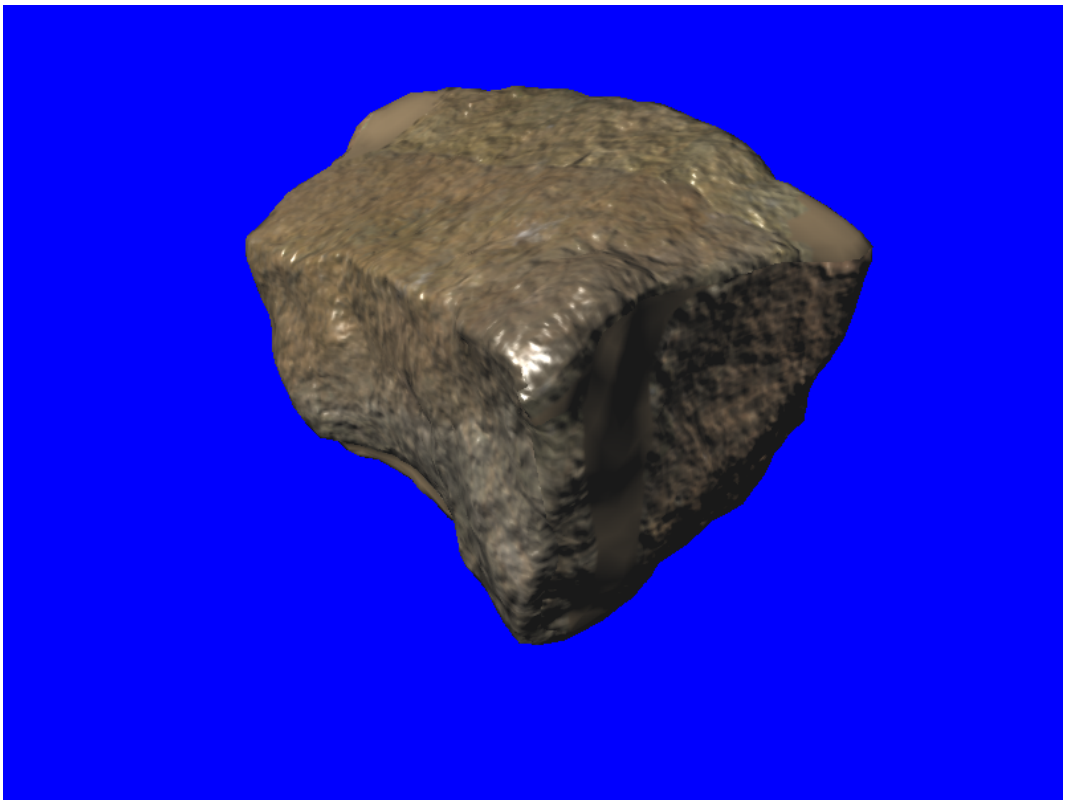
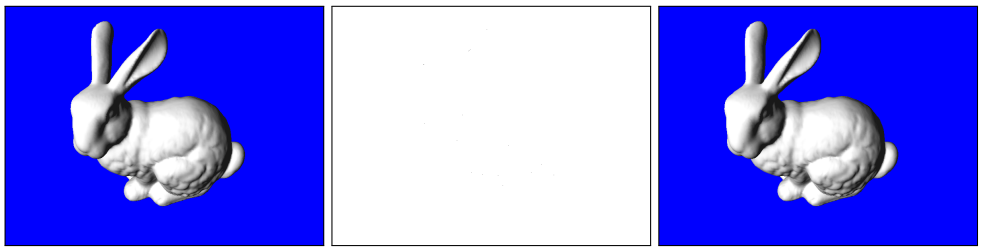


Figure 6.4.: Example using diffuse texture and normal map

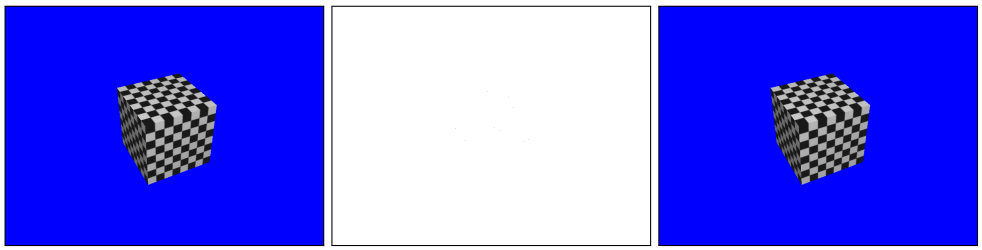
(a) Result (b) Difference (c) Reference

Figure 6.5.: Comparison of specular lighting example with reference implementation



(a) Result (b) Difference (c) Reference

Figure 6.6.: Comparison of diffuse lighting example with reference implementation



(a) Result (b) Difference (c) Reference

Figure 6.7.: Comparison of texturing example with reference implementation

(a) Result (b) Difference (c) Reference

Figure 6.8.: Comparison of texturing example with reference implementation

7. Conclusion

A compiler for Direct3D intermediate code binaries to CUDA objects that can be used in GPGPU-based software renderers has been developed. While not all shader instructions are currently supported, it is possible to translate shaders based on commonly used constructs. Using the generated code for shading in a renderer has been shown to be feasible. Nonetheless, integration of the work in concrete renderer implementations is required to use some of the shader features, such as screen-space derivatives and other interactions with fixed-function parts of the graphics pipeline. This chapter will point out some possibilities for future work and improvements to the existing tool.

By adding additional back-ends for other target languages, it would be possible to extend the compiler for other GPGPU technologies or to enable CPU-side software renderer support. Another possibility is re-purposing the program as a shader compatibility tool by adding back-ends for shading languages such as GLSL or SPIR-V. Extensions for other source languages are in general outside the scope of this work. The internal code representation mostly reflects the instruction set of Direct3D shader binaries, making the suitability as a generic intermediate format uncertain. However, support for shader models older than 4.0 is very likely representable using the current instruction set, and possibly viable if backwards compatibility to earlier versions of the Direct3D API becomes a concern.

The back-end could be enhanced with code instrumentation functionality as well. This way, insertion of additional statements could form a foundation for debugging and performance profiling tools for the software renderer target platform. Debugging information is optionally supplied in shader binaries. Using that data, generated code could be annotated to associate its instructions or statements with the original HLSL source code.

Currently, the code generators try to improve readability of the output by using strongly typed declarations for temporary variables whenever possible. For the C++ back-end, constant buffers are declared as variables or data structures instead of a block of uninterpreted data, and generated operands refer to constant buffers using those names instead of the offsets used by the shader code. In case this effort is pursued further, a mechanism to recognize common patterns in the binary code could be implemented to make the result appear more similar to higher-level code written by humans. For example, common vector and matrix operations usually generate the same sequences of instructions, and would then be replaced with a more familiar arithmetic expression form in C++.

Automatic simplification of shaders has been a research interest in the past. A shader compiler that generates instances of the same shader at different levels of detail has been developed by Olano, Kuehne, and Simmons, (2003). Simplification based on transformation rules for an AST of a shader has been implemented by Pellacini, (2005). Genetic programming also has been successfully employed to simplify shaders by Sitthi-Amorn et al., (2011), using an AST-based program representation as well. A similar system could be implemented for Direct3D shader binaries, allowing simplifications to be transparently applied to shaders of existing graphics applications as well. Instead of the plain form, the already generated SSA representation of shaders is likely a suitable basis for simplification algorithms.

In order to accelerate shading in a real-time rendering environment, caching of partial evaluations of pixel shaders has been proposed (Guenter, Knoblock, and Ruf, 1995). Another technique based on caching is reprojection (Nehab et al., 2007; Sitthi-amorn et al., 2008), which make use of frame-to-frame coherency to re-use results or partial results of earlier pixel shader evaluations. Future work could involve an extension to the existing program which modifies shaders to store and load cache-able expression values. Again, compiled HLSL shaders are suitable for this process even if the shader source code is not available. The SSA representation is believed to be helpful for identifying cache-able partial results, as each sub-expression is assigned a unique name. Data dependencies can be detected using SSA as well, making it easy to identify parts of the shader program which are affected by caching values.

Using such data dependency information, another useful optimization can be performed automatically, where shaders are separated into partial programs according to the frequency at which each part has to be computed. For example, a pixel shader typically consists of both view-dependent and view-independent parts that could be computed separately at different frequencies. A similar optimization was done by Ragan-Kelley et al., (2007), extracting the part of a surface shader independent of lighting to implement a fast cache-based preview system for editing the lighting of a scene.

Appendix

Appendix A.

List of Abbreviations

ABI Application binary interface
API Application programming interface
AST Abstract syntax tree
CFG Control flow graph
DFS Depth-first search
GLSL OpenGL shading language
GPU Graphics processing unit
GPGPU General-purpose graphics processing unit
HLSL High-level shading language
MSSA Multisample anti-aliasing
PTX Portable thread execution
SPIR Standard portable intermediate representation
SSA Static single assignment
UAV Unordered access view

Appendix B.

Direct3D Shader Instruction Set

B.1. Declarations

dcl_constantBuffer
dcl_function_body
dcl_function_table
dcl_globalFlags
dcl_gs_instance_count
dcl_hs_fork_phase_instance_count
dcl_hs_join_phase_instance_count
dcl_hs_max_tess_factor
dcl_immediateConstantBuffer
dcl_indexableTemp
dcl_indexRange
dcl_input
dcl_input_control_point_count
dcl_input_sgv[1]
dcl_input_siv[1]
dcl_input_ps[2]
dcl_input_ps_sgv[2]
dcl_input_ps_siv[2]
dcl_inputPrimitive
dcl_interface
dcl_interface_dynamicIndexed
dcl_maxOutputVertexCount
dcl_output
dcl_output_control_point_count
dcl_output_sgv
dcl_output_siv
dcl_outputTopology
dcl_resource
dcl_resource_raw
dcl_resource_structured
dcl_sampler
dcl_stream
dcl_temps
dcl_tessellator_domain
dcl_tessellator_output_primitive
dcl_tessellator_partitioning
dcl_tgsm_raw
dcl_tgsm_structured
dcl_thread_group
dcl_uav_raw
dcl_uav_raw_glc
dcl_uav_structured
dcl_uav_structured_glc
dcl_uav_typed
dcl_uav_typed_glc

[1] Documented as dcl_input_sv
[2] Undocumented

B.2. Arithmetic and Bit Operations

Single Precision

exp
frc
log
rcp
rsq
sqrt
add
div
dp2
dp3
dp4
eq
ge
lt
max
min
mul
ne
mad
sincos

Double Precision

dadd
deq
dge
dlt
dmax
dmin
dmul
dne
dfma

Integer Arithmetic

iadd
ieq
ige
ilt
imad
imax
imin
imul
ine
ineg
uaddc
udiv
uge
ult
umad
umax
umin
umul
usubss

Bit Operations

bfrev
bfi
countbits
firstbit_lo
firstbit_hi
firstbit_shi
not
and
ishl
ishr
or
ubfe
ushr
xor

Atomic Operations

atomic_and
atomic_cmp_store
atomic_iadd
atomic_imax
atomic_imin
atomic_or
atomic_umax
atomic_umin
atomic_xor
imm_atomic_alloc
imm_atomic_and
imm_atomic_cmp_exch
imm_atomic_consume
imm_atomic_exch
imm_atomic_iadd
imm_atomic_imax
imm_atomic_imin
imm_atomic_or
imm_atomic_umax
imm_atomic_umin
imm_atomic_xor

B.3. Control Flow

break	discard	loop
breakc	else	switch
call	endif	ret
callc	endloop	retc
case	endswitch	nop
continue	fcall	
continuec	if	
default	label	

B.4. Data Move and Conversion

dmov[3]	utof	round_pi
mov	itod[23]	round_z
f16tof32	utod[23]	dmovc[3]
f32tof16	dtoi[23]	movc
ftod[3]	dtou[23]	swapc
ftoi	dtof[3]	
ftou	round_ne	
itof	round_ni	

B.5. Memory Access

ld	store_structured	sample
ld_raw	store_uav_typed	sample_b
ld_structured	gather4	sample_c
ld_uav_typed	gather4_c	sample_c_lz
ld_2d_ms	gather4_po	sample_d
store_raw	gather4_po_c	sample_l

B.6. Domain-Specific and Miscellaneous

[3]Unimplemented

Appendix B. Direct3D Shader Instruction Set

deriv_rtx
deriv_rty
deriv_rtx_coarse
deriv_rty_coarse
deriv_rtx_fine
deriv_rty_fine
emit
emit_stream
emit_then_cut
emit_then_cut_stream
lod
resinfo
sampleinfo
samplepos
eval_snapped[23]
eval_at_sample_index[23]
eval_at_centroid[23]
abort[2]
debug_break[2]
sync
hs_control_point_phase
hs_decls
hs_fork_phase
hs_join_phase
gather4_feedback[23]
gather4_c_feedback[23]
gather4_po_feedback[23]
gather4_po_c_feedback[23]
ld_feedback[23]
ld_ms_feedback[23]
ld_uav_typed_feedback[23]
ld_raw_feedback[23]
ld_structured_feedback[23]
sample_l_feedback[23]
sample_c_lz_feedback[23]
sample_clamp_feedback[23]
sample_b_clamp_feedback[23]
sample_d_clamp_feedback[23]
sample_c_clamp_feedback[23]
check_access_fully_mapped[23]

Appendix C.

Shader Translation Example

An example for translated code is shown in listing C.3. The HLSL source code, as given by listing C.1, is the shader used for the specular lighting example (figure 6.3). Listing C.2 shows the shader assembly code (signatures and resource definitions omitted) generated by the HLSL compiler.

```
PsOut main(PsInput input)
{
    PsOut output;
    float4 albedo = DiffuseTexture.Sample(DiffuseSampler, input.texcoord);
    float3 light_dir = LightPos.xyz - input.cameraSpacePosition.xyz;
    float n_dot_l = dot(normalize( light_dir ), normalize(input.normal.xyz));
    output.color = albedo * saturate( n_dot_l ) * LightColor;
    output.color.xyz = saturate(output.color.xyz);
    output.color.w = 1.0f;
    return output;
}
```

Listing C.1: A test shader written in HLSL

```
ps_5_0
dcl_globalFlags refactoringAllowed
dcl_constantbuffer CB0[2], immediateIndexed
dcl_sampler s0, mode_default
dcl_resource_texture2d (float,float,float,float) t0
dcl_input_ps linear v1.xyz
dcl_input_ps linear v2.xyz
dcl_input_ps linear v3.xy
```

```
dcl_output o0.xyzw
dcl_temps 2
add r0.xyz, -v1.xyzx, cb0[0].xyzx
dp3 r0.w, r0.xyzx, r0.xyzx
rsq r0.w, r0.w
mul r0.xyz, r0.wwww, r0.xyzx
dp3 r0.w, v2.xyzx, v2.xyzx
rsq r0.w, r0.w
mul r1.xyz, r0.wwww, v2.xyzx
dp3_sat r0.x, r0.xyzx, r1.xyzx
sample_indexable(texture2d)(float,float,float,float) r0.yzw,
↪ v3.xyxx, t0.wxyz, s0
mul r0.xyz, r0.xxxx, r0.yzwy
mul_sat o0.xyz, r0.xyzx, cb0[1].xyzx
mov o0.w, l(1.000000)
ret
```

Listing C.2: Direct3D assembly code generated by the HLSL compiler

```
.version 5.0
.target sm_50, texmode_unified
.address_size 64

/* dcl_constantBuffer cb[0][2].xyzw, immediateIndexed */
.const .v4.b32 cb0[2];

/* dcl_sampler, s0, default */
.const .u32 s0_cmpFunc;

/* dcl_resource t0, Texture2D, (float,float,float,float); dcl_sampler, s0, default (unified) */
.global .texref t0_s0;

.visible.func (.align 16 .param.b8 retval[16])
  ↪ _ZN12pixel_shader5shadeER6float4RKS0_S3_S3_RK6float2 (.param.b64 o0_SV_TARGET0, .param.b64
  ↪ v0_SV_POSITION0, .param.b64 v1_TEXCOORD0, .param.b64 v2_NORMAL0, .param.b64 v3_TEXCOORD1)
{
  /* (...) */

  /* add r0.xyz, -v1.xyzx, cb[0][0].xyzx */
  .reg .u64 tmpreg_7, tmpreg_8, tmpreg_9;
  mov.u64 tmpreg_9, cb0;
  mul.lo.u64 tmpreg_8, 0x0, 0x10;
  add.u64 tmpreg_7, tmpreg_9, tmpreg_8;
  .reg .f32 tmpreg_10_x, tmpreg_10_y, tmpreg_10_z, tmpreg_10_w;
  ld.const.v4.f32 {tmpreg_10_x, tmpreg_10_y, tmpreg_10_z, tmpreg_10_w}, [tmpreg_7 + 0];
  sub.ftz.f32 r0_0_x, tmpreg_10_x, v1_x;
  sub.ftz.f32 r0_0_y, tmpreg_10_y, v1_y;
  sub.ftz.f32 r0_0_z, tmpreg_10_z, v1_z;
  /* dp3 r0.w, r0.xyzx, r0.xyzx */
```

```
mul.ftz.f32 r0_0_w, r0_0_x, r0_0_x;
mad.rn.ftz.f32 r0_0_w, r0_0_y, r0_0_y, r0_0_w;
mad.rn.ftz.f32 r0_0_w, r0_0_z, r0_0_z, r0_0_w;
/* rsq r0.w, r0.w */
rsqrt.approx.ftz.f32 r0_1_w, r0_0_w;
/* mul r0.xyz, r0.wwww, r0.xyzx */
mul.ftz.f32 r0_1_x, r0_1_w, r0_0_x;
mul.ftz.f32 r0_1_y, r0_1_w, r0_0_y;
mul.ftz.f32 r0_1_z, r0_1_w, r0_0_z;
/* dp3 r0.w, v2.xyzx, v2.xyzx */
mul.ftz.f32 r0_2_w, v2_x, v2_x;
mad.rn.ftz.f32 r0_2_w, v2_y, v2_y, r0_2_w;
mad.rn.ftz.f32 r0_2_w, v2_z, v2_z, r0_2_w;
/* rsq r0.w, r0.w */
rsqrt.approx.ftz.f32 r0_3_w, r0_2_w;
/* mul r1.xyz, r0.wwww, v2.xyzx */
mul.ftz.f32 r1_0_x, r0_3_w, v2_x;
mul.ftz.f32 r1_0_y, r0_3_w, v2_y;
mul.ftz.f32 r1_0_z, r0_3_w, v2_z;
/* dp3_sat r0.x, r0.xyzx, r1.xyzx */
mul.ftz.f32 r0_2_x, r0_1_x, r1_0_x;
mad.rn.ftz.f32 r0_2_x, r0_1_y, r1_0_y, r0_2_x;
mad.rn.ftz.sat.f32 r0_2_x, r0_1_z, r1_0_z, r0_2_x;
/* sample r0.yzw, v3.xyxx, t0.wxyz, s0 */
.reg .f32 tmpreg_11_x, tmpreg_11_y, tmpreg_11_z, tmpreg_11_w;
call.uni (tmpreg_11_x), d3dxc_dFdx_fine, (v3_x);
call.uni (tmpreg_11_y), d3dxc_dFdx_fine, (v3_y);
.reg .f32 tmpreg_12_x, tmpreg_12_y, tmpreg_12_z, tmpreg_12_w;
call.uni (tmpreg_12_x), d3dxc_dFdy_fine, (v3_x);
call.uni (tmpreg_12_y), d3dxc_dFdy_fine, (v3_y);
.reg .f32 tmpreg_13_x, tmpreg_13_y, tmpreg_13_z, tmpreg_13_w;
tex.grad.2d.v4.f32.f32 {tmpreg_13_x, tmpreg_13_y, tmpreg_13_z, tmpreg_13_w}, [t0_s0, {v3_x,
  ↪ v3_y}], {tmpreg_11_x, tmpreg_11_y}, {tmpreg_12_x, tmpreg_12_y};
mov.f32 r0_2_y, tmpreg_13_x;
mov.f32 r0_2_z, tmpreg_13_y;
mov.f32 r0_4_w, tmpreg_13_z;
/* mul r0.xyz, r0.xxxx, r0.yzwy */
mul.ftz.f32 r0_3_x, r0_2_x, r0_2_y;
mul.ftz.f32 r0_3_y, r0_2_x, r0_2_z;
mul.ftz.f32 r0_3_z, r0_2_x, r0_4_w;
/* mul_sat o0.xyz, r0.xyzx, cb[0][1].xyzx */
.reg .f32 tmpreg_14_x, tmpreg_14_y, tmpreg_14_z;
.reg .u64 tmpreg_15, tmpreg_16, tmpreg_17;
mov.u64 tmpreg_17, cb0;
mul.lo.u64 tmpreg_16, 0x1, 0x10;
add.u64 tmpreg_15, tmpreg_17, tmpreg_16;
.reg .f32 tmpreg_18_x, tmpreg_18_y, tmpreg_18_z, tmpreg_18_w;
ld.const.v4.f32 {tmpreg_18_x, tmpreg_18_y, tmpreg_18_z, tmpreg_18_w}, [tmpreg_15 + 0];
mul.ftz.sat.f32 tmpreg_14_x, r0_3_x, tmpreg_18_x;
mul.ftz.sat.f32 tmpreg_14_y, r0_3_y, tmpreg_18_y;
mul.ftz.sat.f32 tmpreg_14_z, r0_3_z, tmpreg_18_z;
st.global.v2.f32 [o0_xyzw + 0], {tmpreg_14_x, tmpreg_14_y};
st.global.f32 [o0_xyzw + 8], tmpreg_14_z;
/* mov o0.w, l(0x3f800000) */
.reg .f32 tmpreg_19_w;
```

```
  mov.f32 tmpreg_19_w, 1.00000;
  st.global.f32 [o0_xyzw + 12], tmpreg_19_w;
  /* ret */
  .reg .f32 tmpreg_20_x, tmpreg_20_y, tmpreg_20_z, tmpreg_20_w;
  ld.global.v4.f32 {tmpreg_20_x, tmpreg_20_y, tmpreg_20_z, tmpreg_20_w}, [o0_xyzw + 0];
  st.param.v4.f32 [retval + 0], {tmpreg_20_x, tmpreg_20_y, tmpreg_20_z, tmpreg_20_w};
  ret;
}
```

Listing C.3: Generated PTX code. Declarations of temporary registers and parameter retrieval code have been omitted.

```
#include "shader_interface.h"

extern "C" {
  static const uint32_t global_flags =
      GLOBAL_FLAG_REFACTORING_ALLOWED;
  __constant__ unsigned char cb0[32];
  extern texture<float4, cudaTextureType2D,
      cudaReadModeElementType> t0_s0;
  static const unsigned int num_clip_distances = 0u;
  static const unsigned int num_clip_distances = 0u;
}

struct pixel_shader
{
  __device__ static float4 shade(float4&, const float4&,
      const float4&, const float4&, const float2&);
};
```

Listing C.4: Generated C++ interface for the PTX code

Bibliography

Aho, Alfred V. et al. (2007). *Compilers: Principles, Techniques and Tools*. 2nd edition. Addison-Wesley. ISBN: 0-321-48681-1 (cit. on pp. 27, 40).

Akeley, K. et al. (2002). *ARB_vertex_program*. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB%5C_vertex%5C_program.txt (visited on 04/24/2017) (cit. on p. 3).

Batty, Christopher (2001). URL: https://cs.uwaterloo.ca/~c2batty/bunny_watertight.obj (visited on 09/17/2017) (cit. on p. 55).

Beretta, B. et al. (2013). *ARB_fragment_program*. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB%5C_fragment%5C_program.txt (visited on 04/24/2017) (cit. on p. 3).

Blythe, David (2006). "The Direct3D 10 System." In: *ACM SIGGRAPH 2006 Papers*. SIGGRAPH '06. Boston, Massachusetts: ACM, pp. 724–734. ISBN: 1-59593-364-6. DOI: 10.1145/1179352.1141947. URL: http://doi.acm.org/10.1145/1179352.1141947 (cit. on p. 1).

Buck, Ian et al. (2004). "Brook for GPUs: Stream Computing on Graphics Hardware." In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. Los Angeles, California: ACM, pp. 777–786. DOI: 10.1145/1186562.1015800. URL: http://doi.acm.org/10.1145/1186562.1015800 (cit. on p. 13).

CodeSourcery et al. (2017). *Itanium C++ ABI (Revision: 1.86)*. URL: http://refspecs.linux-foundation.org/cxxabi-1.86.html (visited on 04/23/2017) (cit. on p. 38).

Cook, Robert L. (1984). "Shade Trees." In: *SIGGRAPH Comput. Graph.* 18.3, pp. 223–231. ISSN: 0097-8930. DOI: 10.1145/964965.808602. URL: http://doi.acm.org/10.1145/964965.808602 (cit. on p. 13).

Cooper, Keith D., Timothy J. Harvey, and Ken Kennedy (2001). "A Simple, Fast Dominance Algorithm." In: *Software: Practice and Experience* 4, pp. 1–10. URL: http://www.hipersoft.rice.edu/grads/publications/dom14.pdf (cit. on p. 32).

Cytron, Ron et al. (1991). "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *ACM Trans. Program. Lang. Syst.* 13.4, pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320 (cit. on p. 32).

Duca, Nathaniel et al. (2005). "A Relational Debugging Engine for the Graphics Pipeline." In: *ACM Trans. Graph.* 24.3, pp. 453–463. ISSN: 0730-0301. DOI: 10.1145/1073204.1073213. URL: http://doi.acm.org/10.1145/1073204.1073213 (cit. on p. 14).

Elliott, Conal (2004). "Programming Graphics Processors Functionally." In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: ACM, pp. 45–56. ISBN: 1-58113-850-4. DOI: 10.1145/1017472.1017482. URL: http://doi.acm.org/10.1145/1017472.1017482 (cit. on p. 13).

Epic Games, Inc. (2017). *HLSL Cross Compiler — Unreal Engine*. URL: https://docs.unrealengine.com/latest/INT/Programming/Rendering/ShaderDevelopment/HLSLCrossCompiler/ (visited on 05/07/2017) (cit. on p. 12).

Foley, Timothy John (2012). "Spark: Modular, Composable Shaders for Graphics Hardware." PhD thesis. Stanford University. URL: http://purl.stanford.edu/wz483vv5440 (cit. on p. 12).

Gritz, Larry (2016). *Open Shading Language 1.7 Language Specification*. URL: https://github.com/imageworks/OpenShadingLanguage/raw/master/src/doc/osl-languagespec.pdf (visited on 06/05/2017) (cit. on p. 3).

Guenter, Brian, Todd B. Knoblock, and Erik Ruf (1995). "Specializing Shaders." In: *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '95. New York, NY, USA: ACM, pp. 343–350. ISBN: 0-89791-701-4. DOI: 10.1145/218380.218470. URL: http://doi.acm.org/10.1145/218380.218470 (cit. on p. 60).

Hanrahan, Pat and Jim Lawson (1990). "A Language for Shading and Lighting Calculations." In: *SIGGRAPH Comput. Graph.* 24.4, pp. 289–298. ISSN: 0097-8930. DOI: 10.1145/97880.97911. URL: http://doi.acm.org/10.1145/97880.97911 (cit. on p. 3).

Houston, Mike and Christopher Cameron (2010). *cl_khr_d3d10_sharing*. URL: https://www.khronos.org/registry/OpenCL/extensions/khr/cl_khr_d3d10_sharing.txt (visited on 05/22/2017) (cit. on p. 1).

Jensen, Peter Dahl Ejby et al. (2007). "Interactive Shader Development." In: *Proceedings of the 2007 ACM SIGGRAPH Symposium on Video Games*.

Sandbox '07. San Diego, California: ACM, pp. 89–95. ISBN: 978-1-59593-749-0. DOI: 10.1145/1274940.1274959. URL: http://doi.acm.org/10.1145/1274940.1274959 (cit. on p. 13).

Julliard, Alexandre et al. (2017). *WINE*. Version 2.0.1. URL: https://www.winehq.org/ (cit. on p. 12).

Kessenich, John, Dave Baldwin, and Randi Rost (2016). *The OpenGL Shading Language*. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf (visited on 05/30/2017) (cit. on p. 3).

Kessenich, John, Boaz Ourial, and Raun Krisch (2017). *SPIR-V Specification*. URL: https://www.khronos.org/registry/spir-v/specs/1.2/SPIRV.pdf (visited on 05/30/2017) (cit. on p. 3).

Knowles, Pyarelal (2017). URL: http://goanna.cs.rmit.edu.au/~pknowles/models/wt_teapot.obj (visited on 09/17/2017) (cit. on p. 56).

Laine, Samuli and Tero Karras (2011). "High-performance Software Rasterization on GPUs." In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: ACM, pp. 79–88. ISBN: 978-1-4503-0896-0. DOI: 10.1145/2018323.2018337. URL: http://doi.acm.org/10.1145/2018323.2018337 (cit. on p. 14).

Liu, Fang et al. (2010). "FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-fragment Effects." In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: ACM, pp. 75–82. ISBN: 978-1-60558-939-8. DOI: 10.1145/1730804.1730817. URL: http://doi.acm.org/10.1145/1730804.1730817 (cit. on p. 14).

Mark, William R., R. Steven Glanville, et al. (2003). "Cg: A System for Programming Graphics Hardware in a C-like Language." In: *ACM Trans. Graph.* 22.3, pp. 896–907. ISSN: 0730-0301. DOI: 10.1145/882262.882362. URL: http://doi.acm.org/10.1145/882262.882362 (cit. on pp. 3, 11).

Mark, William R. and Kekoa Proudfoot (2001). "Compiling to a VLIW Fragment Pipeline." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '01. Los Angeles, California, USA: ACM, pp. 47–56. ISBN: 1-58113-407-X. DOI: 10.1145/383507.383526. URL: http://doi.acm.org/10.1145/383507.383526 (cit. on p. 11).

McCool, Michael D., Zheng Qin, and Tiberiu S. Popa (2002). "Shader Metaprogramming." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '02. Saarbrucken, Germany:

Eurographics Association, pp. 57–68. ISBN: 1-58113-580-7. URL: http://dl.acm.org/citation.cfm?id=569046.569055 (cit. on p. 13).

McCool, Michael et al. (2004). "Shader Algebra." In: *ACM Trans. Graph.* 23.3, pp. 787–795. ISSN: 0730-0301. DOI: 10.1145/1015706.1015801. URL: http://doi.acm.org/10.1145/1015706.1015801 (cit. on p. 13).

McGuire, Morgan et al. (2006). "Abstract Shade Trees." In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. Redwood City, California: ACM, pp. 79–86. ISBN: 1-59593-295-X. DOI: 10.1145/1111411.1111425. URL: http://doi.acm.org/10.1145/1111411.1111425 (cit. on p. 13).

Microsoft Corporation (2017). *Shader Model 5 Assembly*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/hh447232.aspx (visited on 05/30/2017) (cit. on p. 16).

Microsoft Corporation (2018). *Graphics Pipeline (Windows)*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx (visited on 01/18/2018) (cit. on p. 7).

Mycroft, Alan (1999). "Type-Based Decompilation (or Program Reconstruction via Type Reconstruction)." In: *Proceedings of the 8th European Symposium on Programming Languages and Systems*. ESOP '99. London, UK, UK: Springer-Verlag, pp. 208–223. ISBN: 3-540-65699-5. URL: http://dl.acm.org/citation.cfm?id=645393.651886 (cit. on p. 32).

Nehab, Diego et al. (2007). "Accelerating Real-time Shading with Reverse Reprojection Caching." In: *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. GH '07. San Diego, California: Eurographics Association, pp. 25–35. ISBN: 978-1-59593-625-7. URL: http://dl.acm.org/citation.cfm?id=1280094.1280098 (cit. on p. 60).

Ni, Tianyun et al. (2009). "Efficient Substitutes for Subdivision Surfaces." In: *ACM SIGGRAPH 2009 Courses*. SIGGRAPH '09. New Orleans, Louisiana: ACM, 13:1–13:107. DOI: 10.1145/1667239.1667252. URL: http://doi.acm.org/10.1145/1667239.1667252 (cit. on p. 20).

NVIDIA Corporation (2017a). *CUDA Driver API :: CUDA Toolkit Documentation*. URL: http://docs.nvidia.com/cuda/cuda-driver-api/group%5C_%5C_CUDA%5C_%5C_GRAPHICS.html#group%5C_%5C_CUDA%5C_%5C_GRAPHICS (visited on 05/30/2017) (cit. on p. 1).

NVIDIA Corporation (2017b). *NVCC :: CUDA Toolkit Documentation*. URL: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.

html%5C#cuda-compilation-trajectory (visited on 05/30/2017) (cit. on p. 5).

NVIDIA Corporation (2017c). *NVRTC (Runtime Compilation) :: CUDA Toolkit Documentation*. URL: http://docs.nvidia.com/cuda/nvrtc/index.html (visited on 05/30/2017) (cit. on p. 5).

NVIDIA Corporation (2017d). *NVVM IR :: CUDA Toolkit Documentation*. URL: http://docs.nvidia.com/cuda/nvvm-ir-spec/index.html (visited on 05/30/2017) (cit. on p. 9).

NVIDIA Corporation (2017e). *PTX ISA :: CUDA Toolkit Documentation*. URL: http://docs.nvidia.com/cuda/parallel-thread-execution/ (visited on 01/22/2018) (cit. on p. 5).

Olano, Marc, Bob Kuehne, and Maryann Simmons (2003). "Automatic Shader Level of Detail." In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '03. San Diego, California: Eurographics Association, pp. 7–14. ISBN: 1-58113-739-7. URL: http://dl.acm.org/citation.cfm?id=844174.844176 (cit. on p. 60).

Olano, Marc and Anselmo Lastra (1998). "A Shading Language on Graphics Hardware: The Pixelflow Shading System." In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, pp. 159–168. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280857. URL: http://doi.acm.org/10.1145/280814.280857 (cit. on p. 12).

Patney, Anjul et al. (2015). "Piko: A Framework for Authoring Programmable Graphics Pipelines." In: *ACM Trans. Graph.* 34.4, 147:1–147:13. ISSN: 0730-0301. DOI: 10.1145/2766973. URL: http://doi.acm.org/10.1145/2766973 (cit. on p. 14).

Peercy, Mark S. et al. (2000). "Interactive Multi-pass Programmable Shading." In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., pp. 425–432. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344976. URL: http://dx.doi.org/10.1145/344779.344976 (cit. on p. 12).

Pellacini, Fabio (2005). "User-configurable Automatic Shader Simplification." In: *ACM Trans. Graph.* 24.3, pp. 445–452. ISSN: 0730-0301. DOI: 10.1145/1073204.1073212. URL: http://doi.acm.org/10.1145/1073204.1073212 (cit. on p. 60).

Proudfoot, Kekoa et al. (2001). "A Real-time Procedural Shading System for Programmable Graphics Hardware." In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, pp. 159–170. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383275. URL: http://doi.acm.org/10.1145/383259.383275 (cit. on p. 11).

Ragan-Kelley, Jonathan et al. (2007). "The Lightspeed Automatic Interactive Lighting Preview System." In: *ACM SIGGRAPH 2007 Papers*. SIGGRAPH '07. San Diego, California: ACM. DOI: 10.1145/1275808.1276409. URL: http://doi.acm.org/10.1145/1275808.1276409 (cit. on p. 61).

Rhodin, Helge (2010). "A PTX Code Generator for LLVM." Bachelor's Thesis. Saarland University (cit. on p. 13).

Robbins, Ed, Andy King, and Tom Schrijvers (2016). "From MinX to MinC: Semantics-driven Decompilation of Recursive Datatypes." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, pp. 191–203. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837633. URL: http://doi.acm.org/10.1145/2837614.2837633 (cit. on p. 34).

Segal, Mark and Kurt Akeley (2016). *The OpenGL Graphics System: A Specification*. URL: https://khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf (visited on 05/30/2017) (cit. on p. 3).

Sharif, Ahmad and Hsien-Hsin S. Lee (2008). "Total Recall: A Debugging Framework for GPUs." In: *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. GH '08. Sarajevo, Bosnia and Herzegovina: Eurographics Association, pp. 13–20. ISBN: 978-3-905674-09-5. URL: http://dl.acm.org/citation.cfm?id=1413957.1413960 (cit. on p. 14).

Sitthi-Amorn, Pitchaya et al. (2011). "Genetic Programming for Shader Simplification." In: *ACM Trans. Graph.* 30.6, 152:1–152:12. ISSN: 0730-0301. DOI: 10.1145/2070781.2024186. URL: http://doi.acm.org/10.1145/2070781.2024186 (cit. on p. 60).

Sitthi-amorn, Pitchaya et al. (2008). "Automated Reprojection-based Pixel Shader Optimization." In: *ACM SIGGRAPH Asia 2008 Papers*. SIGGRAPH Asia '08. Singapore: ACM, 127:1–127:11. ISBN: 978-1-4503-1831-0. DOI: 10.1145/1457515.1409080. URL: http://doi.acm.org/10.1145/1457515.1409080 (cit. on p. 60).

Strengert, Magnus, Thomas Klein, and Thomas Ertl (2007). "A Hardware-aware Debugger for the OpenGL Shading Language." In: *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. GH '07. San Diego, California: Eurographics Association, pp. 81–88. ISBN: 978-1-59593-625-7. URL: http://dl.acm.org/citation.cfm?id=1280094.1280108 (cit. on p. 14).

The Khronos Group (2010). *cl_khr_gl_sharing*. URL: https://www.khronos.org/registry/OpenCL/extensions/khr/cl_khr_gl_sharing.txt (visited on 05/22/2017) (cit. on p. 1).

The Khronos Vulkan Working Group (2017). *Vulkan 1.0.50 - A Specification*. URL: https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf (visited on 05/30/2017) (cit. on p. 3).

Unity Technologies (2017). *Unity - Manual: Shading Language used in Unity*. URL: https://docs.unity3d.com/Manual/SL-ShadingLanguage.html (visited on 05/07/2017) (cit. on p. 12).

Wu, Jingyue et al. (2016). "Gpucc: An Open-source GPGPU Compiler." In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO 2016. Barcelona, Spain: ACM, pp. 105–116. ISBN: 978-1-4503-3778-6. DOI: 10.1145/2854038.2854041. URL: http://doi.acm.org/10.1145/2854038.2854041 (cit. on p. 13).