Universiteit Leiden
The Netherlands

Opleiding Informatica

Optimizing the Citra emulator

by decompiling shader machine code

Mees Delzenne

Supervisor:
Todor P. Stefanov

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

November 16, 2018

Abstract

In this thesis, we describe the work done to improve the running speed of the Citra emulator. This work entailed an optimization of the Citra emulator render pipeline. New code was written to allow a new part of the Citra emulator render pipeline, the vertex shading stage, to be emulated on a GPU as opposed to the CPU and thus improving performance. In order to allow the GPU to emulate the vertex shading stage we need to decompile 3DS's shader machine code to a higher level language which can run on a GPU. So a decompiler was written which allowed Citra to decompile 3DS shader machine code at runtime. Citra was then changed further to allow the decompiled shader machine code to run on a GPU, emulating the 3DS's GPU, bypassing the previous emulation on the CPU. For evaluation, several programs were written which test the performance of the graphical pipeline of Citra. The evaluation shows that the new additions to Citra improve performance in most cases.

Contents

Chapter 1

Introduction

In this chapter, we give some background information on emulation and the goal of this project.

1.1 Emulation

Emulation is the process of making a certain computer platform, hereafter referred to as the host platform, able to behave like another platform. Code for the emulated platform would then be able to run on the host platform as if it were running on the emulated platform. This can be done for various reasons: to test the effectiveness of a new hardware platform without actually needing to create the hardware or to be able to run software from platforms which no longer exist.

1.1.1 Emulation types

Emulation is traditionally done by simulating the hardware of a platform down to the individual registers or cycle-accurate emulation. This has the benefit of being highly accurate, a quality necessary if one wants to run all possible programs for an emulated platform or if one wants to test a platform. The first emulators written for game platforms did emulation via this method. These emulators were able to allow programs to be able to run at full speed since those emulators targeted an emulated platform which was generally far older and less advanced than the host platforms on which they would run. An example of such an emulator is jsspeccy [8] a javascript ZX Spectrum emulator. However, the emulation of the Nintendo N64 proved, at the time, to be difficult to accomplish by the traditional method. So, a new way of emulation was devised.

The authors of UltraHLE [17], a Nintendo N64 emulator, were the first to coin the term *High Level Emulation* or HLE. High level emulation abandons the simulation of individual registers in favor of less accurate but more performant solutions. Instead of simulating the graphics pipeline of the Nintendo N64 by fully emulating every part of the graphical hardware, UltraHLE chose to intercept the system calls which the emulated program

made and then writing their graphics routines implementing the emulated platforms system calls. This requires a lot less computational power but since the individual registers are no longer simulated the emulation is less accurate. Because this method was called High Level Emulation the more traditional method was called *Low Level Emulation*. Most modern game platform emulators today implement High Level Emulation.

1.2 Emulators vs Virtual Machines

Emulators and virtual machines can seem very similar. Especially High Level Emulators have a lot in common with process virtual machines like the Java Virtual Machine [15]. Both can have techniques like Just-In-Time compilation to speed-up performance. Both also provide an implementation of a machine, the only difference being that one is an abstract machine and the other corresponds to real hardware. When taking these factors in account it might be hard to distinguish the two.

The original definition of a virtual machine is defined by Popek and Goldberg as "An efficient, isolated duplicate of a real computer machine." [9]. Although this definition might be somewhat outdated, today's virtual machines do not always correspond to real hardware [10], it does provide a base definition of a virtual machine. If we look at how an high level emulate functions we can see that an emulator does not satisfy the efficiency requirement. By Popek and Globerg's definition efficiency is defined as the capability of the virtual machine to run a significant portion of the code without intervention of the virtual machine often allowing almost direct access to some hardware of the computer. Although the argument can be made that High Level Emulators are as efficient as a process virtual machine like the Java Virtual Machine in terms of executing code directly on the CPU when they implement a JIT compiler, similar to a process virtual machine, this does not extend to any other hardware. High Level Emulators can not provide access to the hardware of the host platform since they need to emulate the hardware of the emulated platform. So by this definition of requiring efficiency we can see that High Level Emulators are distinct from virtual machines.

1.3 Citra

Over the years, as computer platforms were succeeded by newer platforms, there have often been enthusiasts taking on a project to save a platform by creating an emulator which can run the code written for the platform and thus, as hardware begins to deteriorate, the programs, can still be run on newer hardware. Through this process various emulators have been written for platforms ranging from the ZX spectrum [8] to the Playstation 3 [3], which are no longer being produced since last year [7]. The Citra emulator [14] is one of these emulators, emulating the Nintendo 3DS and targeting x86 on Windows, Linux and MacOS. Since the goal of the Citra is to emulate the Nintendo 3DS and not to provide an isolated platform we can call Citra an emulator. Futhermore, Citra also does not satisfy the efficiency requirement of a virtual machine thus it is an emulator.

1.3.1 3DS

For Citra, the emulated platform is the Nintendo 3DS. The Nintendo 3DS, which will henceforth be referred to as the 3DS, is a hand-held game console released on the 25th of March 2011 in Europe. The 3DS is the successor of the Nintendo DS and has better hardware, a description of which can be seen in Table 1.1, and a screen which is capable of creating a 3D effect. Over the years various updates have been released to the 3DS improving hardware and adding additional control interfaces in some cases. However, all software released for the various version of the 3DS continue to be compatible with the original 3DS.

CPU	268 MHz quad-core ARM11 & 134 MHz single-core ARM9
GPU	268 MHz Digital Media Professionals PICA200
Memory	256 MB FCRAM @ 6.4GB/s

Table 1.1: 3DS Hardware

The 3DS has a quite capable graphics pipeline. The GPU's four cores can be programmed to perform various operations, for more information see Section 2.2. All four are capable of performing vertex processing with a programmable shader. Vertex processing is a stage in a rendering pipeline where operations are applied to a sequence of vertices often via shaders, see Section 2.1. One of the four cores, called the Primitive Engine, can be used as a geometry processing unit allowing the GPU to generate geometry from data. This stage is also programmable. This pipeline uses its own specialized instruction set for operations. For more information on the 3DS pipeline see Section 2.1.

1.4 Problem description

Citra consists of multiple different parts, emulating different parts of 3DS's hardware. The CPU is emulated using a JIT compiler which compiles the ARM instructions from the 3DS's platform to native x86 which can then directly be executed. This is one of the optimization which Citra makes to achieve better performance. By avoiding cycle-accurate emulation, Citra is able to run code directly on the target platform. The GPU is similarly implemented on top of the OpenGL library. This allows the emulator to potentially run graphical tasks on the GPU. However, in practice, this is only partially true. A large part of the 3DS's graphical pipeline, the geometry and vertex processing stage, was at the start of the project done on the host platform's CPU. Only rasterization is done on the host platform's GPU. This is primarily because Citra is still in active development and the programmable nature of the geometry and vertex processing pipeline make implementing this pipeline on a modern desktop not trivial. Allowing the vertex processing stage to run on the GPU is the primary focus of this thesis.

At the start of this project Citra was already capable of running a variety of different programs at a reasonable speed. However, it was not yet capable of achieving performance similar to the performance of the 3DS even on recent hardware. After running the emulator with the callgrind profiler [5], it turned out that Citra spends

more then half of its execution time emulating the GPU. The emulation of the emulated platform GPU was done on the host platform CPU. Although Citra did implement a JIT Compiler for the shader code, doing vertex transformation on the CPU remained a major bottleneck. If we were able to emulate the shaders of the 3DS on the host platform GPU it might be possible to achieve large improvements in performance.

In this thesis we want to improve the performance of Citra. We focus specifically on the GPU emulation of Citra as this part of Citra has the largest impact on performance. Before our work, Citra spent over half its execution time emulating the GPU. Our goal is to reduce that execution time.

1.5 Thesis Contribution

In this thesis, we create a decompiler which is capable of decompiling 3DS vertex shader machine code to OpenGL Shading Language or GLSL. See Section 2.3.1. We then made further changes to the Citra emulator, so it can use this decompiler to decompile 3DS vertex shader machine code it encounters to GLSL and subsequently use this code to emulate the vertex shader stage on the host platform GPU. After making these changes, we wrote several programs to test the performance of the modified Citra emulator.

1.6 Related work

In this section, we describe other related work relevant to our approach.

1.6.1 The Citra project

At the end of this project, the Citra project [14] itself has released its own decompilation optimization. Because the experiments presented in this thesis were already completed at that time, we could not include the Citra implementation in our performance evaluation. Whereas the decompiler proposed in the thesis relies on matching control flow structures, the implementation of the Citra project instead matches the code as if none of the control flow instructions can jump into a region of code which any other instruction jumps over. This is not necessarily correct for all possible shaders but works for a large part of the possible shaders. If the shader can not be decompiled because it does not adhere to this rule, the implementation falls back to the CPU implementation. Because of this, their decompiler is a lot less complex, but our method should be able to decompile more possible shader programs. However, their decompiler is more capable than ours in some way as it is able to both decompile the geometry shader machine code and the vertex shader machine code.

1.6.2 Decompilation of Binary Programs

In the paper Decompilation of Binary Programs [4] the authors describe their work creating a decompiler. The decompiler is a general case decompiler with replaceable back-end and front-ends allowing the decompiler to, in principle, decompile several different machine code formats to several different high level programming languages. So, the paper describes a decompiler for the Intel 80286 architecture which generates C programming language code. The decompiler itself is somewhat similar to our own decompiler. It uses the same technique of matching control flow structures on a control flow diagram. However, their decompiler described is a general decompiler which makes it some what more complex. Also, the decompiler is not written for our use case as it tries to generate readable code. Being able to output readable code is not a requirement for our decompiler.

1.6.3 No More Gotos

The paper No More Gotos [11] describes a decompiler which is capable of generating goto-free output. The decompiler is capable of recovering all control constructs in a binary program by using pattern-independent control-flow structuring. Their decompiler described in this paper is different compared to our decompiler which does use patterns to match control flow structures. Their decompiler is thus capable of decompiling all possible binary code while our decompiler cannot. The implementation of this decompiler is a lot more complex than our implementation. The decompiler described in the paper is also not build for decompiling the 3DS shader machine code to GLSL so it cannot be used for our use case.

1.6.4 Dynamic Binary Translation

The paper Dynamic Binary Translation [13] describes a program which translates x86 machine code to a proprietary VLIW machine code. It does this translation on the fly while the program is running. This use case is somewhat similar to our own decompiler which translates machine code to GLSL on the fly in order to emulate the 3DS's GPU on the host platform GPU. However the program described in the paper translates from machine code format to another machine code format. This avoids the problem of matching high level control flow structures on machine code which is one of our major problems.

1.7 Thesis Overview

This chapter contains the introduction; Chapter 2 contains background information necessary to understand the work done in this thesis; Chapter 3 discusses how our implementation works; Chapter 4 evaluates the implementation; Chapter 5 concludes the thesis.

This thesis is a bachelor thesis and is part of the computer science study at the Leiden Institute of Advanced Computer Science. This thesis is supervised by Todor P. Stefanov.

Chapter 2

Background

In order to emulate a certain task, we need an understanding of both the task to be emulated and the host platform on which we are going to emulate this task. The task that is considered in this thesis is the task executed by the 3DS's vertex stage. In this chapter, we detail how the vertex shader works on the 3DS and what we need to be able to run it on the host platform's GPU. The host platform which Citra targets, is a modern desktop computer with Windows, Linux, or MacOS. More specifically, we target a host platform which is capable of running the OpenGL API. This API will handle the differences between the different platforms for us and is thus our target. In Section 2.3, we give a general overview of how this API works and give further information necessary to understand the implementation.

The information cited about the 3DS's hardware is taken from an online reverse engineering wiki 3DBrew [1], maintained by volunteers, and the behavior of the already present code in the Citra emulator.

2.1 The PICA200

The GPU of the 3DS has the code name PICA200 and it has a graphics pipeline which consists of four stages: Primitive Assembly, Geometry Processing, Vertex Processing, and Rasterisation. When the PICA200 GPU renders, it handles the data given by the CPU of the 3DS in these four stages, each time applying operations to finally produce an image on the screen of the 3DS. Primitive Assembly is the stage where primitives are copied from memory and prepared to be sent to the GPU for processing. The Geometry Processing is the stage where data is transformed to generate additional geometry. This can be used for graphical operations like tessellation or mesh deformation. Implementing this on a GPU is often faster then doing this on the CPU. This stage is programmable using a specialized instruction set. A program which runs on a GPU is called a shader. In the case of the PICA200 such a shader consists of a set of instructions which are part of the specialized instruction set. The Vertex Processing stage is the stage where the geometry, which was sent to the GPU by the CPU or by the previous geometry processing stage, is transformed according to certain operations. This stage is also programmable with a specialized instruction set. This instruction set is mostly the same as the

instruction set for the geometry processing stage with the only difference being a few instructions which are absent as they are only allowed in the geometry stage. Rasterisation is the process of taking the data, received from the vertex stages and calculating the actual pixels. As mentioned previously, the Rasterisation stage of the pipeline is already implemented on the GPU in Citra. On the 3DS this is also the stage where lighting is done if used. A diagram of the pipeline can be seen in Figure 2.1.

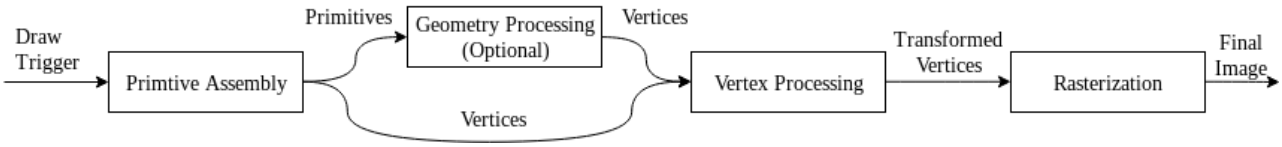

Figure 2.1: 3DS graphics pipeline

The PICA200 GPU is configurable by a number of registers which are accessible by the CPU of the 3DS. These registers allow a program to specify the data which the GPU needs to render a frame. A few registers related to this paper are: The *draw* register which causes the GPU to render after a write. We generally refer to the effect of writing to this register as a draw trigger. The *uniform* registers is a set of registers which contain values that can be accessed during shader execution. The *attribute* registers that specify the offset, size, and format of geometry data which the GPU will process. The *output map* registers which specify the semantics of the output registers of the shaders, see Section 2.2.1. Finally, the *code transfer* registers allow the 3DS's CPU to write the instructions for the PICA200 GPU's shader.

Programming the shaders for the PICA200 GPU is done in the devkitpro toolchain [6] using an assembly language and an assembler. This toolchain is made by volunteers for the creation of homebrew applications and is almost certainly not used in the creation of official games for the 3DS. Unfortunately we did not have access to an official development kit so we do not know how the shaders for official games are created.

2.2 PICA200 Shader Instruction set

In this section we detail the specialized instruction set which is used to program the two programmable stages of the 3DS's graphics pipeline. We have called this instruction set the PICA200 instruction set after the code name of the 3DS's GPU. This instruction set specifies various instructions which the programmable interface of the GPU can execute. Some of these instructions are linear algebra operations, like dot product and vector magnitude, as these operations are commonly used in rendering code. Furthermore, most instructions execute on a vector of four values instead of a singular scalar. An instruction generally has 1, 2 or 3 source registers and a single destination register. The instruction set does not specify any memory operations because the GPU is not capable of random memory access.

2.2.1 Shader registers

The PICA200 instruction set has access to a set of shader accessible registers: output registers, input registers, uniform registers, and temporary registers.

The output registers, which can contain a float vector of size 4, are write-only. These registers are used to retrieve the program's output. The semantics of these registers can be set by the program, each of the components of an output can have a special meaning: like vertex position or vertex color. These semantics are specified by the CPU accessible registers. This data is then used in later stages of the pipeline to paint a picture. There are seven output registers.

The input registers, of which there are 8, are a set of read-only registers which store the per-vertex data given by the CPU. These registers also contain a float vector of size 4.

The uniform registers, of which there are 108, are a set of registers which can be set by the CPU. These registers remain the same across a single shader invocation. 96 of the 108 registers are float vectors of size 4, 8 are boolean scalar values, and 4 are integer vectors of size 4. These registers are often used to contain items like view-matrices.

The temporary registers are only accessible by the shader and are used to temperarily store data between instructions. These registers also contain a float vector of size 4.

There is also a single address register which is used for the LOOP instructions and relative addressing. It has three components `a.x`, `a.y` and `aL` which are all integers. This register can be used to do relative addressing. It is possible to add the value of an address register to the address of a uniform register when using an instruction. For example, if we use relative addressing and have an instruction `MULL r0,c4,c8` and the values of **a.x** and **a.y** are correspondingly 3 and 5, the instruction will effectively be `MULL r0 c7 c13`. This relative addressing can only be done with uniform registers.

Finally, there is a register called **cmp** which will contain the result of a CMP comparison instruction. This register has two boolean components called `cmp.x` and `cmp.y`. This register can not be accessed directly. It can only be set by a CMP instruction and the result is used in conditional branching.

An overview of all registers is given in Table 2.1.

When an instruction accesses a register it can specify which components it wants to use. This is called swizzling and is commonly used in GPU operations. The instruction set specifies a way to select which components of the source registers will be used and in which order. Furthermore, the instruction set also allows one to specify which components of the destination registers are affected by the operation. Since the bits in the machine code of the instruction set are limited, certain instructions which reference two or more operands can only access the input and temporary registers in some of the operands.

Type	Name	Value	Use description
Input	i0-i7	float vector 4	Per-vertex data
Output	o0-o6	float vector 4	Program out data
Temporary	r0-r15	float vector 4	During program operation uses.
Uniform	c0-c95	float vector 4	Used for per invocation data
	b0-b7	boolean vector 4	
	i0-i3	integer vector 4	
Address register	a.x,a.y,aL	integer vector 3	Relative addressing and loop counter
Cmp	cmp.x,cmp.y	boolean vector 2	Storing result of comparison and condition branching

Table 2.1: Overview of the PICA200 registers

2.2.2 Shader instructions

The instruction set consists of three different instruction groups i.e: Arithmetic, Geometry, and Control Flow instructions. Arithmetic instructions have 1, 2, or 3 source registers and a destination register. Most of these arithmetic instructions use all components of the vector. An overview of all arithmetic instructions can be seen in Table 2.2.

The control flow instructions determine how the program is run. These instructions have two values which change the operation of the instructions: DST and NUM. An overview of the branching instructions can be seen in Table 2.4. Most of these instructions branch the program based on a certain condition. An instruction can handle two different condition formats. The instructions suffixed with U are uniform conditional instructions. These instructions branch based on the value of a uniform register. The instruction suffixed with a C use the `cmp` variable. The instructions have two reference bits and a CONDOP operand, which describes how the instruction will evaluate the values in register `cmp`. It can check if the first and second components are both equal to the bits, if either of the components is equal to the bits, if the first component is equal to the first bit, or if the last component is equal to the last bit.

2.2.3 Floating-Point Behavior

The PICA200 uses 16 bit floating-point numbers. These floats are not IEEE compliant. For instance, the expression `Infinity * 0` does not return the special floating point value `NaN`, or Not a Number, as it should in IEEE compliant floats but returns `0`. This difference between the emulated platform and the host platform might cause inaccurate emulation. However, in general programs, relying on corner cases like these are uncommon. So the implementation, in the interest of time, ignores these differences.

2.3 OpenGL

The OpenGL API [12] is an open standard for graphics programming. Over the years various versions have been released each adding features to the standard. Citra targets the 3.2 version of OpenGL which is compatible

Instruction	Description
ADD	Adds to vectors component wise
MUL	Multiplies the 2 source vectors component wise
DP3	Computes the dot product of a 3 or 4 component vector
DP4	
DPH	Computes the dot product of a 3 component vector with 1.0 appended as the w component and a 4 component vector
DST	calculates new vector as following $DST = \{1, SRC1[1] * SRC2[1], SRC1[2], SRC2[3]\}$
EX2	Computes the exponent with base 2 of the first component assigns them to all components in destination
LG2	Computes the logarithm with base 2 of the first component assigns them to all components in destination
LITP	Computes a new vector with the component values between certain values
SGE	Compares the components of 2 vectors and set the output component to 1.0 if the first vectors component is greater then or equal to the second otherwise 0.0
SLT	Compares the components of 2 vectors and set the output component to 1.0 if the first vectors component is lest then the second otherwise 0.0
FLR	Rounds down a vector component wise
MAX	Calculates the maximum value of 2 vectors component wise
MIN	Calculates the minimum value of 2 vectors component wise
RCP	Calculates the reciprocal of the vectors first component
MIN	Calculates the reciprocal of the square root the first component
MOVA	Sets the value of the address register a.x and a.y to the first 2 components of a vector
MOV	Sets one vector to the value of the other vector
NOP	Does nothing
CMP	Compares 2 values x and y component separately can behave differently depending on operand see Table 2.3

Table 2.2: Arithmetic instruction

Operand	Description
EQ	Test if the first value component is equal to the other value component
NE	Test if the first value component is not equal to the other value component
LT	Test if the first value component is less then the other value component
LE	Test if the first value component is less or equal to the other value component
GT	Test if the first value component is greater then the other value component
GE	Test if the first value component is greater or equal to the other value component

Table 2.3: CMP operands

Instruction	Description
JMPC JMPU	If condition is true then the instruction will jump to DST
IFC IFU	If the condition is true will execute instructions until DST then jump to DST + NUM, otherwise will jump to dist. Only allowed to jump forward
LOOP	Takes a integer register INT and first sets address register aL to INT.y then loops over the code between itself and DST performing INT.x iterations adding INT.z to aL after each iteration. Only allowed to jump forward
BREAKC	Can only be used in a loop. If conditions is true will stop the loop.
CALLC CALLU	If the condition is true this will jump to dist and execute $NUM - 1$ instructions
CALL	Unconditional variant of CALLC and CALLU.
END	Ends program execution.

Table 2.4: Control Flow instructions

with newer versions. The API consists of various functions which allows a programmer to send data to the GPU and receive data from the GPU. The API also allows a measure of programmability of the GPU. The GPU can be programmed with shaders which are programs written in the GLSL language or OpenGL Shading Language. The shaders are similar to the programs which can run on the 3DS's GPU. There are a variety of different types of shaders in OpenGL, however, for this thesis, we only need to work with the Vertex shader which is the most similar to the vertex processing stage of the PICA200 GPU.

2.3.1 GLSL

GLSL is a C like language with some additional features for graphics operations. The languages has built-in support for vectors and matrices. Also, just like the PICA200 instruction set, GLSL supports swizzling, an example can be seen in Listing 2.1.

Listing 2.1: Swizzling in GLSL

```
1  vec4 foo = vec4(1.0,0.0,3.0,4.0);
2  vec3 bar = vec3(2.0,1.0,5.0);
3  vec3 faz.zyx = foo.wyz * bar.xyy;
4  // Faz is here {3.0, 0.0, 8.0}
```

This short code snippet creates two vectors, one of size 4 and the other of size 3 as denoted by `vec4` and `vec3`. These vectors are multiplied in Line 3 where the components used in the multiplication are selected with swizzling. The destination of the components is also swizzled as can be seen on Line 3 in the `faz.zyx` part.

Like most C-like languages GLSL supports the standard C control flow constructs; It has selection statements: (**if**, **else**, and **switch case**), iteration statements: (**for**, **while**, and **do while**), and a few jump statements, (**break**, **continue** and **return**). However, unlike C, it has no **goto** statement. One can also define functions in GLSL and just like C a function called main is used as the entry point into the program.

GLSL has various methods to define variables. This has to do with the various ways in which data can be handed to the shader. Just like PICA200, GLSL can define input variables which will contain per-vertex information. Moreover, there are also uniform variables which contain data which remains the same across a shader invocation, just like PICA200's uniform registers. Output is done via special variables and defined variables. For an example see Listing 2.2

Listing 2.2: Example vertex shader

```
1
2  // The vertex position
3  in vec3 position;
4
5  // A matrix representing transformation
6  uniform mat4 MVP;
7
8  void main(){
```

```
 9       //Set the output position of the vertex
10       gl_Postion = MVP * vec4(position,1.0);
11   }
```

In this example code the position of a vertex is handed to the shader via a attribute called `position` in the form of a three component vector. The vector is then transformed into a vector of size 4 by appending 1.0 as the fourth component. The resulting vector is then multiplied by a matrix and then assigned a special GLSL variable called `gl_Position`. `gl_Position` is a special value which is used to tell OpenGL what the final position is of a vertex for rendering.

2.4 GPU emulation in Citra

As mentioned in Section 2.1, the 3DS's GPU is configured via registers which are accessible to the CPU. This is also modeled in Citra. Each time the emulated CPU writes to a register a function is called to the GPU module of Citra, with the register written to and the data written as an argument. These writes cause the GPU module to change configurations to emulate the 3DS's GPU properly. For instance, when a code transfer register is written to the GPU module marks the current shader as dirty. Then when the emulated CPU writes to the draw register to trigger a draw, the shader is recompiled from the new state. This is how the GPU module generally functions. When the emulated CPU writes to a GPU register, the configuration which needs to be changed is marked as dirty and when the emulated CPU triggers a drawing the actual changes necessary are made before drawing.

When a draw is triggered, first the vertices are loaded from the memory individually. The memory locations for those vertices are calculated from the attribute registers. Each of these vertices is sent individually to the shader emulator which is, in the original implementation ,a x86 JIT-compiler which compiles the vertex shader machine code to x86 code which can run on the host platform. The machine code of the vertex shader to be emulated is written into a buffer by the emulated CPU by accessing CPU accessible GPU registers. In order to keep recompilation to a minimum the programs created by the JIT-compiler are stored in a hash map. Programs are stored with a key so that they can be retrieved later. This key consists of all values the GPU module keeps track of which might cause the shader to behave differently, like program instructions and the entry point of the shader. This way if the GPU is configured in a previously already encountered configuration, recompilation is not necessary. After a vertex is processed by the shader, the result is copied to a buffer which collects all the processed vertices. This buffer is then sent to the rasterizer of Citra which will draw the final image from the processed vertices. This rasterizer is already implemented on the GPU.

Chapter 3

Methods

In this chapter we describe our work that has been done in order to optimize Citra.

When we tested Citra we noticed that a larger part of the execution time was spent in the vertex shader emulator. In fact almost half of the entire execution time was spent in the vertex shader emulator. The time spent emulating the vertex shader made it so Citra could not run general applications at the speed which the 3DS would run the application. Considering the differences in hardware between a 3DS and a modern desktop, one would think that the emulator should be able to run the emulation faster. Inspection of the code did not yield any direct optimization which could be made. The shader emulator was a JIT compiler, compiling shader machine code to x86, so improving over this implementation on the CPU seemed unlikely. The only option which might lead to a significant performance improvement was moving the emulation vertex shading of the 3DS's GPU over to the host platform GPU. Our optimization, thus, focuses on allowing vertex shading emulation of Citra's GPU to be done on the host platform GPU.

The problem, however, is that the 3DS's GPU has a different interface then the GPU on the host platform Citra is targeted. In order to be able to emulate the 3DS's GPU on the host platform GPU we need to translate how the 3DS interfaces with its GPU to something which is compatible to the interface of the host platform GPU. This is already partially done in Citra, the rasterization is already done on the GPU. Since translating, in the case of emulation in Citra, we go from a lower level programming construct to a higher level programming construct, i.e. , from machine code to a high level programming language, we call this translation decompiling.

3.1 Overview of changes to Citra

In Section 2.4, we described how Citra emulates the GPU before the implementation of decompilation. In order to allow Citra to do some decompilation, however, we needed to make some changes.

First, instead of going through CPU vertex shader Citra in our implementation first calls a new function which bypasses the CPU vertex shader and tries to draw directly from the vertices. Calling this function will cause

the program to check if the current state can be drawn directly from the vertices given by the emulated CPU. This is done by first checking if the geometry engine is enabled. Since our decompiling implementation cannot handle drawing when the geometry engine is enabled, the function will fail if the geometry engine is enabled. Next the program checks if there is already a decompiled shader for the current configuration of the program. Just like the previous implementation, already decompiled shaders are stored in a hash-map together with the relevant configuration state. If there is no decompiled shader found then decompilation of the shader is triggered. How the decompilation itself works is described in the next section, Section 3.2. It is possible that the decompilation fails. Then the result is stored in the hash-map and the called function also fails. If the program has a decompiled shader it hands the decompiled shader to OpenGL which will compile the shader into a OpenGL shader program which will be used during drawing. After the implementation has found or created a decompiled shader, the vertices are loaded from memory and stored in such a way that OpenGL can used these for drawing. This mostly entails copying data into a buffer at specific offsets. Then a draw on the host platform GPU is triggered and the function completes, bypassing the CPU vertex shader.

This implementation allows Citra to fallback to the CPU vertex shader if any of the parts of the GPU vertex shader fail.

3.2 Decompiling

Decompiling is the process of taking the result of a compiled program and try to reconstruct code in a higher-level language. Decompilation is mostly used in reverse-engineering. Since high-level code is more intuitive to understand compared to low level assembly, decompiling can help with understanding a program. Most literature about decompilation is about decompilation as a technique in reverse engineering. Our goal, however, is not the understanding of a program but to reconstruct a program which is semantically equivalent to the lower level code from which we are reconstructing.

Throughout this chapter, we use a single example code to demonstrate how the various steps in our decompiler work. We use an instruction format which has the opcode first then possible operands. The example code can be seen in Listing 3.1

Listing 3.1: Example shader assembly

```
1  MUL r0.xyzw, c7.wzwz , i0.xyzw; // Multiply
2  MOV o5.xz  , r0.xzzz           // Move
3  ADD o5.yw  , c90.zzzz, r0.yyww  // Add
4  IFU b1     , 14      , 0        // Conditional uniform if with DST=14 and NUM=0
5  MOV r4.xyzw, c90.wzyx           // Move
6  CMP EQ, EQ , c4.wzyx , r4.wyzw  // Comparison if 2 registers are equal
7  IFC X      , 9       , 4        // Conditional cmp if with DST=9 and NUM=4
8  MOV o6.x   , c95.wwww           // Move
9  IFC Y      , 11      , 1        // Conditional cmp if with DST=11 and NUM=1
10 END                             // Ending program
11 MOV        , o6.x    , c5.zzzz  // Move
```

```
12  NOP                             //
13  NOP                             //
14  MOV         , o4.yzw   , r4.yzww  // Move
15  END                             // Ending program
```

This is code written to demonstrate how the various parts of the decompiler work not necessarily to perform a meaningfull task. The code first does some basic operations: a multiplication in Line 1, a move in Line 2, 5, 8, 11, and 14, and addition in Line 3, and a comparison on Line 6. As can be seen in the code the operands have swizzle selectors, signifying which components are used in a multiplication. In line 4 one can see a branch instruction IFU. This instruction jumps conditionally based on a uniform register, in this case register b1. The IFU instruction functions like a if-else structure; executing the next instruction and then jumping over some instructions if the condition is true otherwise first jumping over some instructions and then executing a number of instructions.

However, in the case of Line 4 it functions like an normal if structure since the second operand, NUM, is 0. So instead Line 4 jumps to instruction 14 if the condition is false. In Line 7 and 9 the IFC instruction can be seen. This instruction's function is similar to IFU with the difference being that instead of the condition being a boolean register, it is the result of the CMP instruction. These instructions have a second operand, NUM, which is larger than zero so they function like an if-else structure. For example, the instruction on Line 7 jumps to instruction 9 if the condition is false, otherwise execution is continued as normal until instruction 9 after which the next four instructions are jumped over. For the precise function of basic and control-flow instructions see Table 2.2 and 2.4, respectively.

3.3 Decompiler Architecture

Decompilation is essentially reversing compilation. So, when we want to know how to decompile a program we can look at how a compiler operates. Most compilers can be divided into a few different parts [2]: The Lexical Analyzer, the Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Code Optimizer, and Code Generator. The general structure of a compiler can be seen in Figure 3.1. In order to determine what our decompiler needs we go backwards.

First, the Code Generator, this is where a compiler generates instructions from an intermediate format. This generation from an intermediate format is done mostly for portability. This allows the compiler to be able to generate different code depending on the platform with the same back-end. Since our decompiler only needs to be able to decompile the PICA200 machine code we can leave this step out.

The next step is the optimizer. Since we are decompiling code we can already assume that the code is optimized. Although it may be possible to optimize the code for GLSL, achieving, possibly, a higher speed, OpenGL itself will also optimize GLSL. Furthermore, the platform we will run our code on can reasonably be assumed to be faster than the platform for which the code to be decompiled was written, as most desktop GPU's are faster than the GPU which is part of the 3DS. Thus, implementing an optimizer does not seems to be necessary.

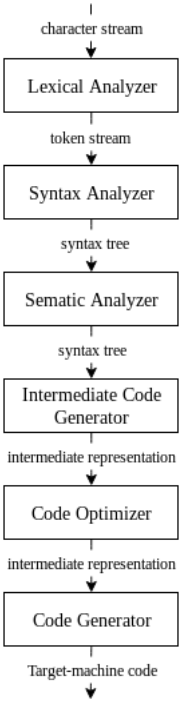

Figure 3.1: The steps of a compiler

The following step is Intermediate Code Generation. In this step, a syntax tree is converted into an intermediate representation. Also in this step, one often generates a control flow diagram. This step is required. If we want to be able to return to higher level code we first need to recreate the structure of the original program. Or at least structure it in such a way that it is possible to generate a syntax tree with higher level control flow structures. This step is called Structurizer in our decompiler and in order for this Structurizer to work we need an idea of how the program runs. This is why we generate a control flow diagram.

The next step is the semantic analysis. In this step, a compiler checks the program for semantic meaning and possible faults. In this step the types of various variables are assigned. As we are dealing with different types we also need to do this step. However, since we go backwards and we already know the types of the registers, constructing a type for a register is trivial.

Finally, the last steps are lexical analysis and syntax analysis. These steps are often separated in a compiler to make the compiler more manageable when dealing with faulty code. Since, we generate the code dealing with faulty code is not a concern so we can combine these steps. In general during lexical and syntax analysis the code is interpreted and transformed to an internal representation, often a syntax tree. So, in a decompiler this is the step where we take our syntax tree and generate high-level code. This is our code generator.

Thus, the layout of our decompiler is first to generate a control-flow diagram from a machine code, then to structurize that diagram to a syntax tree, and finally to generate code from that syntax tree. A diagram of this layout is shown in Figure 3.2

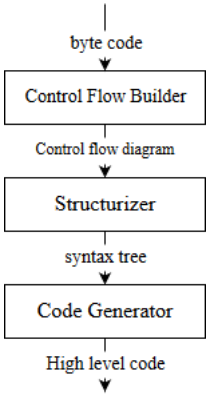

Figure 3.2: The steps of our decompiler

3.4 Design Decision

By looking at the branching instructions in Table 2.4, one might assume that translating the instructions set is a straightforward process because most instructions have a direct equivalent to a high-level control flow structure. The IF instructions can be translated to an if-else structure. The LOOP instruction is structurally equivalent to a for-loop structure. If this was the case for all instructions one would be able to translate all the instructions in a single pass, inserting the corresponding high-level control flow structures as their corresponding instructions are encountered. However, the JMP instruction prevents such simple implementation. JMP is allowed to jump almost anywhere in the program and can thus change the apparent high-level control flow structure. This is why we implemented a more complicated decompiler design with multiple passes.

3.5 Traversing PICO200 Shader Machine Code

Unlike a normal program, the machine code does not have a clear end after which we can stop. The buffer into which the machine code for the shader is stored can contain a maximum of 2048 instructions. Even tough, instructions might have been written to the entire buffer, the shader might end before reaching the end of the machine code buffer. Shaders are ended by the `END` instruction. However, this instruction only ends the program if it is encountered. A branching instruction might jump over an `END` instruction and continue executing.

In order to avoid possible unused instructions which might interfere with code generation, we first need a method to traverse the shader machine code in such a way that only the instructions which might be used are encountered.

This is why we wrote a code visitor which iterates over the instructions only returning the instructions which might be used. It is implemented by keeping track of a stack of the current scopes in which a program might run. When the visitor starts, the entire possible program, from instruction 1 to 2048, is the current scope. When an instruction is requested by the visitor, it removes one instruction from the top scope. If it is a normal arithmetic instruction it just returns the instruction. However, if it is a branching instruction, the

visitor removes the scope at the top of the stack and creates two new scopes corresponding to the two possible ways the program can now run. The branching instruction is then returned. If a scope no longer contains any instructions or an END instruction is encountered we pop the current scope from the stack. All instructions are iterated if we no longer have any scope on the stack.

3.6 Generating a control flow diagram

A control flow diagram represents the different ways a program can run. We need this diagram in order to generate higher level control flow structures.

3.6.1 Control flow diagram definition

First, we need a definition of a control flow diagram. The diagram is a directed graph, the nodes of this graph are represented by basic blocks. A basic block is a set of sequential instructions which will always execute one after another. That is to say, no instruction can jump to an instruction in a code block which is not the start of the block. Furthermore, no instruction but the last instruction can jump to an instruction other then the next sequential instruction. An edge of the graph represents a connection of a basic block to another basic block via a jump. If a jump is a conditional jump the edge also has data on what the conditions need to be for that jump to be taken.

In our decompiler we handle CALL instructions a little different from other control flow instructions. Although this instruction is a control flow instruction and does jump we just handle the instruction as if it does not. Instead, when we encounter a CALL we handle the region to which the instruction jumps as if it is a new program, decompiling the region as if it was a separate program. By ignoring CALL instructions, we can simplify structuring.

3.6.2 Generation of a control flow diagram

Now that we have a definition for the control flow diagram we can start generating a control flow diagram. In our decompiler the control flow is generated in two steps: first we create the different blocks, then we generate the edges. By separating the generation into two steps, we simplify this process because after the blocks are generated we can be sure that no new nodes will be added.

Blocks are generated by splitting existing blocks when encountering a branching instruction. At the start of the generation, the whole program, from instruction 1 to 2048, is considered a single code block. As we traverse the machine code, we split this block after each branching instruction and before the possible targets of that branching instruction.

Then after the blocks have been generated, we connect the blocks by traversing the machine code once again. When we encounter a branching instruction, we generate a connection between its block and its targets blocks. If we encounter an END instruction, we generate a connection to a block which is designated as an exit block. This simplifies structuring later. In this step, we also generate the control flow diagram for the CALL instructions. As we explained earlier in Section 3.6.1, we do not handle CALL instructions as control flow. Instead, we generate a control flow diagram for the region to which the CALL instruction jumps. This is later used to generate functions. A control flow diagram of the example code in Listing 3.1 can be seen in Figure 3.3

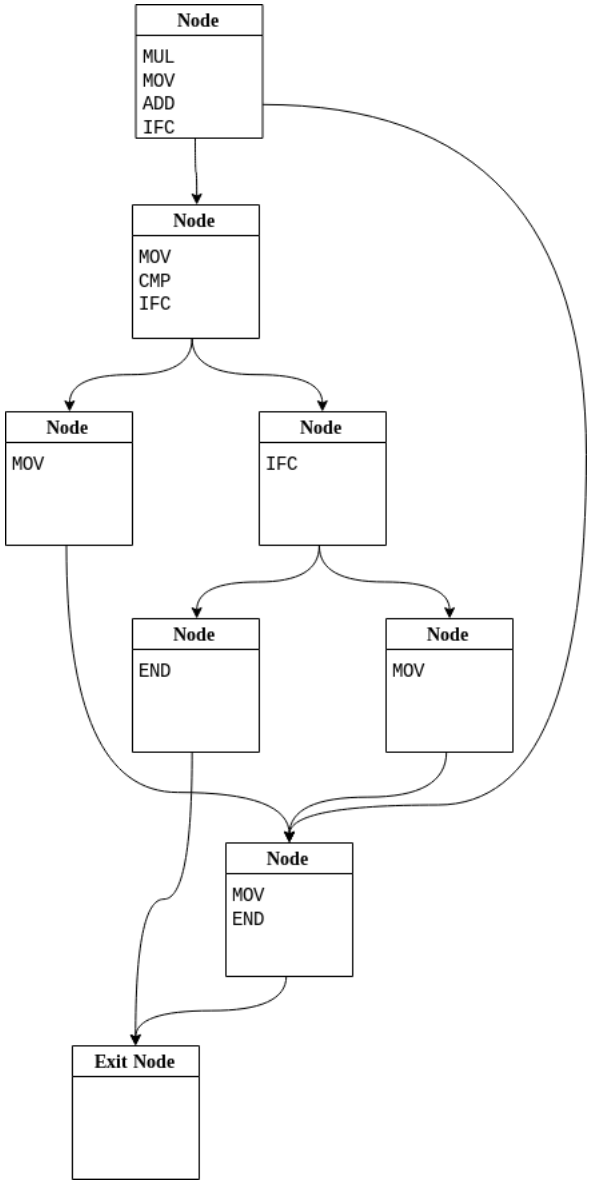

Figure 3.3: Control flow diagram

In this diagram, one can see the various code blocks with the instructions belonging to the code block written in the block. The edges between the nodes represent possible paths the code can take in a program. The exit of the program is also represented by a node, called ”Exit Node”, with no instructions.

3.7 Structurizer

Now that we have a control flow diagram we can start structuring the program into high-level control flow structures. In this step, we create a simplified syntax tree. This tree consists of nodes which denote a scope as one would see in high-level control flow structures. The basic idea of structuring is to look for simple patterns which are semantically equivalent to a high-level control flow structure and match those patterns.

In our implementation of the structuring set, the tree is traversed in post-order. This way, the inner most nodes of a nested pattern are matched first. When the structurizer comes across a pattern it recognizes, it replaces the nodes of the control flow with a single node which denotes the pattern. The new node will then be made to have the same incoming and outgoing connections as the nodes of the replaced pattern. This way, if we have, for example, a control flow diagram which is equivalent to two nested if-structures, with the nested if-structure being a simple if statement with no other control flow within the if. The Structurizer will first recognize and replace the nested if-structure. The outer if will then have a single node in its body and will once again look like a simple if statement which can be matched. Our Structurizer matches if, while, do-while, for, if-else, and sequential statements. When a control flow structure is matched we also store the information of when the control flow jump needs to be taken, retrieved from the edges of the control flow diagram. The Structurizer will keep traversing the control flow repeatedly until it could no longer match a single structure. An example of how this algorithm works can be seen in Figure 3.4

In the first step, we see the graph as it is before structurizing. As we are traversing the graph in post-order the first node we can match is the node with a single `IFC` instruction. In Step 2 we can see the result of the node being matched. The node is transformed into a node signifying a if-else structure. The node has two children nodes attached. The two children nodes represent the instructions which are part of the body of the if-else structure. As can be seen, the nodes matched are removed from the flow of the program, only the top node of the matched structure is kept in the flow. In Step 3, another node is matched. A part of the matched node was a previously matched node which is no part of the children node of the newly matched. In Step 4, there is another node matched. Finally, in Step 5, we match the last two sequential nodes as a `SEQ` node. Unlike other nodes the `SEQ` node does not have instructions itself. After Step 5 there are no more nodes which can be matched.

This structuring of the control flow diagram is done for the main program and all the function regions individually.

3.8 Code Generator

After we have generated the syntax tree we can start to generate code. The resulting code can be divided into three regions. The first region consists of the forward declarations. In this region we declare any functions and/or variables that we might have used in the code. By declaring all functions and variables in advance, we do not have to worry about where and when a variable or functions definition is needed. Functions are named

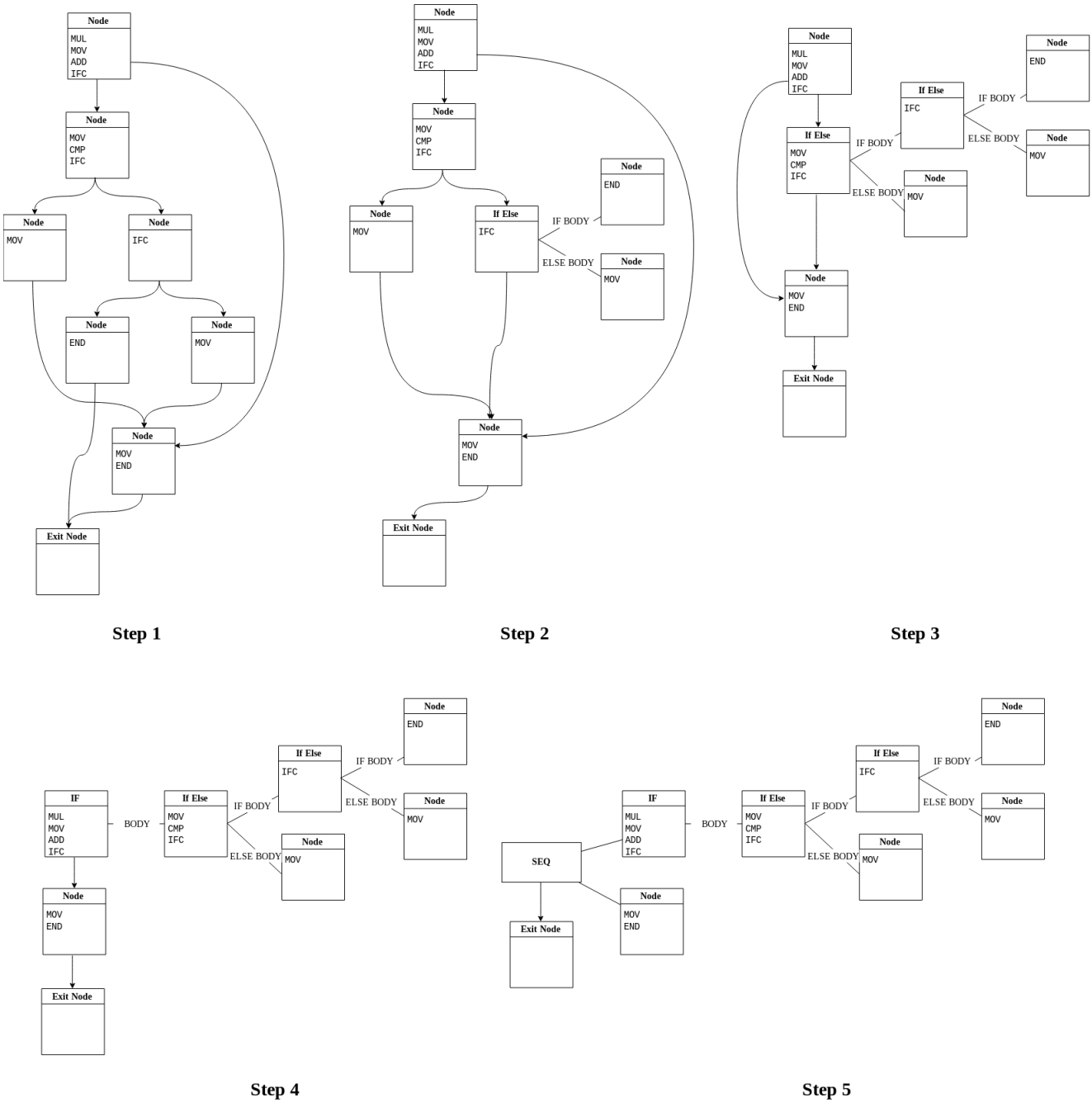

Figure 3.4: Structurizer steps.

after the region where the CALL instruction from which they originally jumped to. For example, if we find a CALL instruction with $DST = 67$ and $NUM = 13$ then the function is named `func_67_80()` Since function calls cannot return a value in the PICA200 instruction set, all functions in GLSL are defined as void. Variables are named after the register which they represent. Just like in the PICA200 instruction set the uniform registers are implemented as uniform variables in GLSL. They are defined as arrays of same size as the number of registers that are present in the instruction set. Temporary variables are declared globally as they are used in the instructions and are initialized to $0.0, 0.0, 0.0, 1.0$ just like in the instruction set.

The second region is the function definition. After we have declared all the functions used in the code we define the functions. This is done by going through all the function region control flow diagrams, which where structured, and generating the code for the region in the same way as the main region of code.

Finally, the last region is the main function definition. In this part, we write the code for the main function. Just like the previous section where we generated the definition of all the functions, we generate code from the syntax tree we have created.

The process of generating code is done by traversing the syntax tree in pre-order while translating the instructions which are part of the nodes of the tree. When we come across a node that represents a control flow construct, such as an if-node, we generate the corresponding control flow statement with the condition stored in the node. Then recursively code is generated for its children nodes, closing the scope for the statement after we are done generating code for its children.

3.8.1 Translating instructions

In general, translating the instructions is straightforward. Most operations can be directly substituted with an operator from GLSL which is semantically equivalent. For instance, if the decompiler encounters an ADD instruction as following `ADD r1, r2, r3` the resulting OpenGL code will be `reg_temp1 = reg_temp2 + reg_temp3`. Since GLSL supports swizzling, we can also translate swizzling on operands pretty easily. A source operand like `i0.xxzw` can be translated to `vs_in_reg0.xxzw`. For the destination operand we need to do some extra work. GLSL does not allow assignment of vectors of different sizes. So, we need to translate an instruction like `MOV r1.xy r2.zzzz` to `reg_temp1.xy = (reg_temp1.zzzz).xy`. Conditional instructions are not translated from the instructions themselves, but from the structure of the syntax tree. If the program encounters a function call it inserts call to the function generated from the region. For example, if we encounter the instruction `CALL 7,5` the program inserts `func_7_13();`.

After the decompiler has generated the main code, it still needs to generate some extra boilerplate code. First, it declares input variables which represent the input registers of the PICA200 shader instruction set. Then, it generates the output variables. These variables do not represent the output registers but instead represent the possible semantics of these registers. The variables which represent the actual registers are defined after the output registers. Then, the decompiler declares a struct representing the uniform registers of the PICA200 shader instruction set. Finally, it declares the internal registers of the PICA200 shader instruction set like the

temporary registers, and the `COND` and address registers. Then, the function with the generated shader code is written and finally we generate the entry point for the GLSL shader, the `main` function. In this function the decompiler first generates a call to the function with the generated shader code and then the decompiler generates some code which maps the variables representing the output registers to the variables representing the semantics of the output registers.

The final code decompiled from the example binary code in Listing 3.1 can be seen in Listing 3.2.

Listing 3.2: Final code

```
1  #version 330 core
2
3  layout (location = 0) in vec4 vs_input_reg_0;
4
5  out vec4 primary_color;
6  out vec2 texcoord[3];
7  out float texcoord0_w;
8  out vec4 normquat;
9  out vec3 view;
10
11 vec4 vs_out_reg[7];
12
13 layout (std140) uniform vs_regs{
14     ivec4 i[4];
15     vec4 f[96];
16     bool b[8];
17 }
18
19 bvec2 COND = bvec2(false,false);
20 ivec3 addres_register = vec3(0.0,0.0,0.0);
21
22 vec4 reg_temp_0 = vec4(0.0,0.0,0.0,1.0);
23 vec4 reg_temp_4 = vec4(0.0,0.0,0.0,1.0);
24
25 void run_shader(){
26     // 1: mull
27     vs_reg_temp_0.xyzw = vs_regs.f[7].wzwz * vs_input_reg_0.xyzw;
28     // 2: mov
29     vs_out_reg[5].xz = (vs_reg_temp_0.xzzz).xz;
30     // 3: add
31     vs_out_reg[5].yw = (vs_regs.f[90].zzzz + -vs_reg_temp_0.yyww).yw;
32     // 4: ifu
33     if((vs_regs.b[1])){
34         // 5: mov
35         vs_reg_temp_4.xyzw = vs_regs.f[90].wzyx;
36         // 6: cmp
37         COND = equal(vec2(vs_regs.f[4].wzyx), vec2(vs_reg_temp_4.xyzw));
38         // 7: ifc
39         if((COND.x)){
```

```
40              // 8: mov
41              vs_out_reg[6].x = (vs_regs.f[95].zzzz).w;
42          }else{
43              // 9: ifc
44              if((!COND.y)){
45                  // 10: end
46                  return;
47              }else{
48                  // 11: mov
49                  vs_out_reg[6].x = (vs_regs.f[5].zzzz).w;
50              }
51              // 12: nop
52          }
53          // 13: nop
54      }
55      //14: mov
56      vs_out_reg[4].yzx = (vs_reg_temp_4.yzww).yzx;
57
58      // 15: end
59      return;
60  }
61
62  void main(){
63      run_shader();
64      gl_Position.x = vs_out_reg[5].x;
65      gl_Position.y = vs_out_reg[5].y;
66      gl_Position.z = vs_out_reg[5].z;
67      gl_Position.w = vs_out_reg[5].w;
68      primary_color.x = vs_out_reg[4].x
69      primary_color.y = vs_out_reg[4].y
70      primary_color.z = vs_out_reg[4].z
71  }
```

In this code, we can see the declaration of variables in Line 3 to Line 23. Line 3 is the declaration of a single input registers. Line 5 to 9 declare the possible semantics of the output registers. Line 19,20, 22 and 23 contain declarations of the variables representing the internal registers of the PICA200 shader instruction set. From Line 25 to Line 60 we can see the function which represents the instructions in the shader which is decompiled. The high-level control flow structures, which were matched in the structurizing step of the decompiler, can be seen in this function. Before each Line in this function one can see a comment instruction which indicates what the next line of code is meant to represent. These comments are also generated in the actual decompiled code for debugging purposes. In Line 62 to 71 we can see the declaration of the GLSL shader entry point, the `main` function. In this main function the function representing the shader is called first in Line 63. Then in Line 64 to Line 70 the variables representing the output register are mapped to the variables representing the semantics of the output registers. Although in the code used as an example the output of a single register is mapped to a single semantic meaning. A single register having a single meaning seems to be common in implementations. However, it is possible for each of the four components of the output registers to have a

different semantic meaning. This is why each component of the variables representing the output registers are assigned individually to a components of the variables representing a sematic meaning. In this case it seems the output register 5 represents the vertex position and output registers 4 represents the color of the vertex.

3.9 Limitations

Although we made the decompiler capable of decompiling and running as many different configurations of the emulated GPU as possible it is not capable of running all the different configurations. In the cases where our emulation on the host platform GPU can not run it will fall back to emulation on the host platform CPU.

First, since our decompiler implementation has no support for geometry shading our implementation can not run if geometry shading is enabled. It might be possible to add this capability in the future, but for this thesis we chose not to implement it. We chose not to implement this feature because we were not very familiar with the OpenGL geometry shading capabilities and the documentation on the geometry stage of the 3DS, that we could find, was rather lacking. Since we could test our implementation without implementing geometry shading we chose to not implement this stage in order to make the scope of this project smaller.

Second, it is possible for a shader program to contain control flow which does not correspond to any higher level control flow structure. An example of such a control flow can be seen in Figure 3.5.

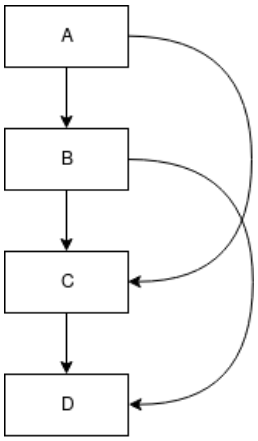

Figure 3.5: Unmatchable control flow structure

In the figure can be seen that nodes A, B and C might correspond to an if-structure if the jump out of node B to the node D did not exist. Node B,C and D would also be an if-structure if the jump from A into node C did not exist, so these nodes can not be matched. Because of the `JMP` instruction in the PICA200 shader instruction-set this control flow is possible. If such a control flow structure is encountered the implementation will have to fall back to emulation on the host platform CPU.

Chapter 4

Evaluation

In order to evaluate the performance of the Citra emulator with our optimization which decompiles shader machine code, we have written three different test programs. These programs are adapted from a test suite program written for Citra itself by wwylele [16]. The original program was written in order to test the emulation of the lightning capabilities of the 3DS. It displays a single cube which is rotating with some lighting applied. This program was selected because it had decent shader complexity which both applies a texture and applies lighting. From our experience a shader in a game commonly consists of applying some textures and then some lighting possibly followed by some other effects. We believe that this is probably also the case for games build for a 3DS. That is why this test seems, in terms of shader complexity, representative for a common workload.

However, the single cube does not seem to fit with a common game. A game generally has more than one object rendered and the objects often have more polygons than a single cube has. That is why we adapted the program in three ways, thereby creating three test programs.

The first test program allows one to render more or less cubes by pressing a button. Each of these cubes is rendered by changing the uniform data and then triggering a draw. This way of rendering is not a very good case for the emulator with GPU decompilation because each time rendering is triggered the emulator needs to do some preparation before it can render the frame.

The second test program is built to be an even worse case for the emulator with GPU decompilation. Instead of rendering blocks the test program only renders a single triangle per draw call. This test program minimizes the amount of work the vertex shader of the emulator needs to do. Because our changed version of the emulator is created to do vertex shader emulation faster, minimizing the actual emulation should result in a worse relative performance to the original version of the emulator. Also since the test program keeps the amount of time the emulator needs to prepare for drawing the same and our changed version of the emulator has more work to do while prepare for drawing, the test program might perform worse on our changed version of the emulator compared to the original version of the emulator.

The third test program also renders multiple cubes, the amount of cubes can be changed by pressing a button. However, instead of triggering a draw for each of the different cubes to be rendered, we update the buffer containing the vertices to include the vertices for another cube. This way, we can render extra cubes but render them all in a single draw trigger. This is a best case for the GPU vertex shader because it only needs to do the setup for rendering once and then the GPU can do the rest. Although the amount of work the shader does is the same on both the first, second an third test, the third test should be significantly faster for our changed version of the emulator since the GPU is better for this task.

Each of these test programs was run and then the average frame time was measured over 5000 frames and tested for both Citra with our decompiling method and the Citra without our decompiling method. We measure multiple times for different amounts of cubes that were rendered. For the specs of the PC on which these tests were done see Table 4.1.

CPU	AMD Ryzen 7 1700x @ 3,6 GHz
GPU	Nvidia GeForce GTX 1080
Memory	16G DDR4 @ 2133MHz
OS	Linux 4.18.5 zen
Distribution	Arch Linux
Driver	Nvidia Proprietary driver version 396.54

Table 4.1: Evaluation PC specs

4.1 Performance of the first test program

The performance result for the first test program where each of the cubes is rendered via a separate render trigger is shown Figure 4.1.

As can be seen from the graph, our version of Citra with decompilation is generally faster then Citra without decompilation. Before this test, we assumed that the GPU vertex shader would probably be slower, because using the GPU vertex shader requires some setup, which costs time. This setup might negate the advantage given by using the master platforms GPU instead of its CPU. Our initial assumption does not seem to be correct as this test case shows. However, if we look at the start of the graph, we can see that the test starts with twenty blocks. If we extrapolate the graph, we see that the emulator with decompilation might perform worse compared to the emulator without decompilation if less cubes are drawn.

4.2 Performance of the second test program

For the second test program we purposely tried to make the test as bad as possible for our version of the emulator with decompilation. The test only renders a single triangle per draw triggered instead of an entire cube. Thus this program requires less work from the vertex shader while keeping the amount of time the

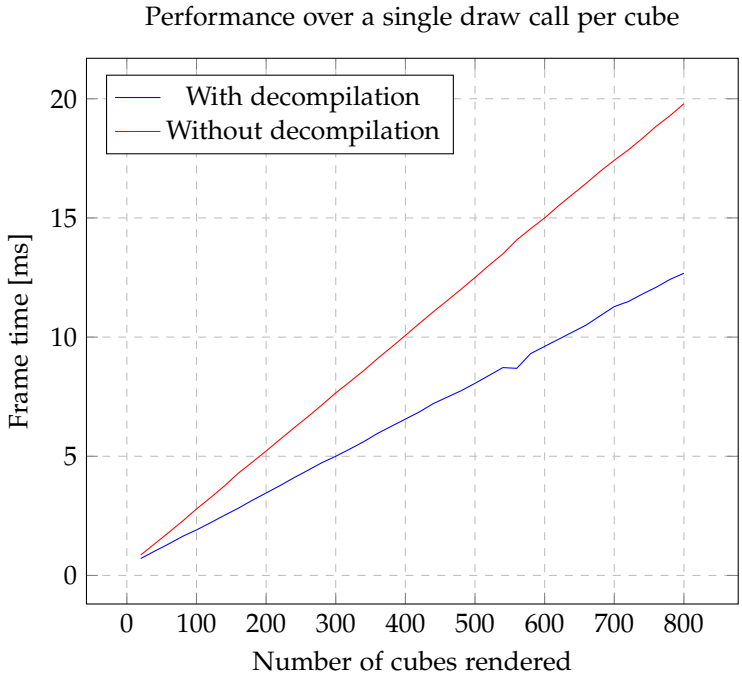

Figure 4.1: Performance of the first test program

emulator needs to prepare the same. So, the time the emulator needs for preparation should be a larger part of the overall frame time. The results of this test can be seen in Figure 4.2

The results does show that the GPU vertex shader performs less well than the CPU vertex shader when only drawing 1 to 7 triangles, while triggering a new draw for each of these triangles. After 7 triangles the GPU vertex shader performs similar or better than the CPU vertex shader. Even when the GPU vertex shader is performing worse than the CPU vertex shader it is only by less than a tenth of a millisecond. This is a very good result for our implementation.

4.3 Performance of the third test program

Here, we tested our best case. In this test program only one draw is trigger per frame while we can still increase the amount of cubes drawn. The results of this test can be seen in Figure 4.3.

As can be seen in Figure 4.3 when many vertices need to be drawn and this can be done in a single draw the GPU performs a lot better than the CPU. With 800 cubes, the emulator with decompilation is 11.5 times faster than the emulator without decompilation. This result shows that the emulator with decompilation can perform a lot better than the first test implied, when a lot of vertices are drawn with a single draw trigger.

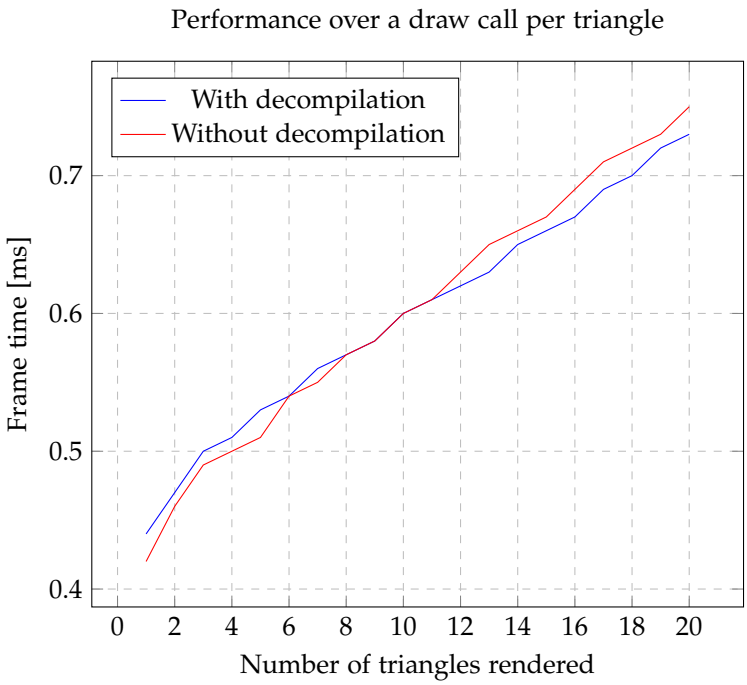

Figure 4.2: Performance of the second evaluation

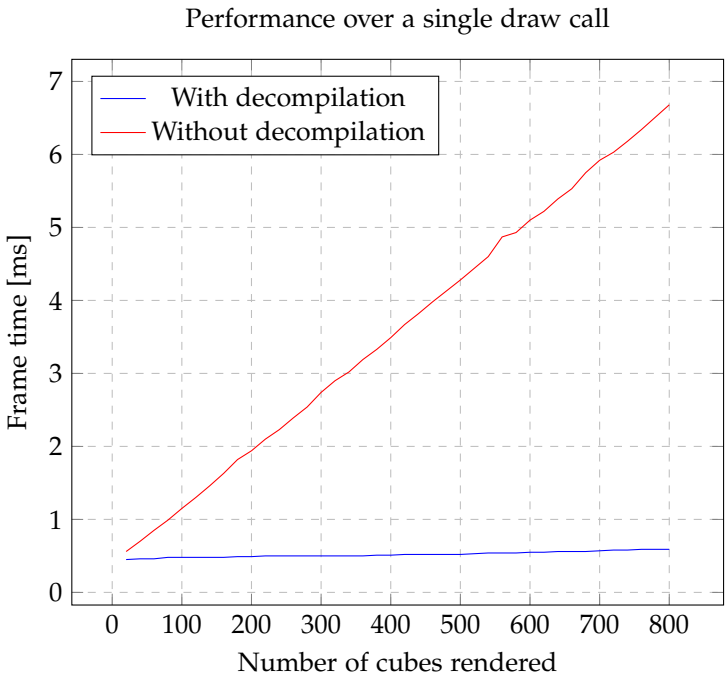

Figure 4.3: Performance of the third evaluation

Chapter 5

Conclusions

In this thesis, we tried to optimize the Citra emulator. In order to make this optimization possible we tried to transfer the emulation of the vertex shading stage of the 3DS's GPU to the host platform GPU. In order to make this possible, we wrote a decompiler which can generally decompile vertex shaders written for the 3DS's GPU. Furthermore, we made further changes to the Citra emulator to allow the emulator to use those decompiled shaders to emulated the 3DS's vertex shading stage on the host platform GPU.

We wrote three test programs to test the performance of the Citra emulator with our optimization. All test results are in favor of the new implementation showing performance improvement across all tests. One can conclude that decompiling the shader machine code to allow GPU emulation of vertex shaders on the host platform GPU leads to a performance increase. The difference in performance between CPU emulation and GPU emulation of vertex shading allows us also to conclude that implementing this emulation on the host platform GPU leads to a significant enough difference that the extra work that this emulation requires is worthwhile.

Although the implementation which was created as part of this project proved that decompiling shader machine code is a worthwhile optimization. The implementation itself is already succeeded by the developers of the Citra project. Their implementation is also capable of decompiling shader machine code of both vertex shaders as well as geometry shaders, see Section 1.6.1. So unfortunately our implementation will thus not be contributed upstream.

Appendix A

How to run

A.1 Running the emulator

This appendix describes how to build and run the emulator on Linux. Although, the code for the emulator might compile and run on other operating systems like Windows or MacOS has not been tested on these platforms. So we recommend to build the emulator on Linux.

A version of the emulator code can be obtained from the following link: *https://github.com/DelSkayn/citra*.

Note that the code has several dependencies to external repositories which will need to be downloaded in order to build the library. The easiest way to obtain all the code necessary for building is by cloning the repository via git with the following commands:

```
1 git clone --recursive https://github.com/DelSkayn/citra
2 cd citra
```

A.1.1 Dependencies

In order to run the Citra emulator one needs a **64-bit OS** and a graphics chip which supports **OpenGL 3.3** or higher. A reasonably performant CPU and GPU are also required if one wants to run 3DS programs at reasonable speed.

In order to build the emulator one needs the following dependencies:

- **SDL2** development libraries
- **Qt** development libraries
- **GCC 7** or higher

- **CMake 3.8** or higher
- (Optional) **boost** version 1.66 or higher, if not installed a version is provided with the code

A.1.2 Building

In order to build the code run the following commands from the top directory of the code directories.

```
1 mkdir build && cd build
2 cmake .. -DCMAKE_BUILD_TYPE=Release
3 make
```

After the build has finished the binaries can be found in the *src* sub-directory in the created *build* directory in the *citra* or *citra-qt* directory. The *citra* binary is a command line binary which needs to be called from the command line to run. The *citra-qt* binary provides a GUI and can be somewhat easier to use.

This build of the program is both capable of running with decompilation and without decompilation. In the *citra-qt* binary this can be changed in the configuration menu under the graphics tab. The option is called ”Enable GPU vertex shading” and it should be enabled by default.

A.2 Running the test programs

The code for the test programs can be obtained from the following link *https://github.com/DelSkayn/citra-test*.

A.2.1 Dependencies

The test programs have some less commonly used dependencies than the emulator itself.

In order to build the test program one needs to install the *devkitpro* [6] build environment. A tutorial for installing this build environment can be found on the following link *https://devkitpro.org/wiki/Getting_Started*. This environment has a dedicated package manager from which one needs to install the following packages:

- **3ds-dev**
- **citro3d**
- **picasso**

A.2.2 Building

Building is done by running make in each of the three top directories for the different programs. After compilation has finished a new file should have been created in the top directories with the `.3dfx` extension,

this file is an compiled 3DS binary. These binaries can be run with the emulator by calling the *citra* binary from the command line with the path of one of these binaries as an argument, or by using the open file dialog from the *citra-qt* binary.

Bibliography

[1] The 3DS homebrew wiki. https://www.3dbrew.org/wiki/Main_Page. Accessed: 23-07-2018.

[2] V. Aho Alfred, S. Lam Monica, Sethi Ravi, and D. Ullman Jeffrey. *Compilers, Principles, Techniques, Tools*. Pearson Eduction, 2 edition, 2007.

[3] Nekotekine at al. RPCS3 Open-source Playstation 3 Emulator. https://rpcs3.net. Accessed: 11-9-2018.

[4] Cifuentes Cristine and Gough K., John. Decompilation of binary programs. *Software, Practice and Experience*, 25:811–829, 1995.

[5] Valgrind Developrs. Callgrind: a call-graph generating cache and branch prediction profiler. http://valgrind.org/docs/manual/cl-manual.html. Accessed: 10-10-2018.

[6] The devkitpro team. Devkitpro build environment. https://devkitpro.org. Accessed: 23-9-2018.

[7] Sony entertainment. Japanese playstation site. http://www.jp.playstation.com/ps3/hardware/cech4300c.html. Accessed: 19-07-2018.

[8] Gasman. Javascript ZX Spectrum emulator. https://github.com/gasman/jsspeccy2. Accessed: 11-9-2018.

[9] Popek Gerald and Goldberg Robert. Formal requrements for virtualizable third generation architectures. *Communications of the ACM*, 17:412–421, 1974.

[10] Smith James and Nair Ravi. The architecture of virtual machines. *Computer*, 38:32–38, 2005.

[11] Yakdan Khaled, Eschweiler Sebastian, and Smith Elmar, Gerhards-Padilla adn Matthew. No more gotos: Decompilation using patter-independent control-flow structuring and smenatic-preserving transformations. Technical report, Internet Society, 2015.

[12] The khronos group. The OpenGL API. https://www.opengl.org. Accessed: 10-10-2018.

[13] Mark Probst. Dynamic binary translation. https://www.complang.tuwien.ac.at/cd/papers/bintrans.pdf. CD laboratory for Compilation Techniques.

[14] The Citra project. The Citra emulator. https://citra-emu.org/. Accessed: 11-9-2018.

[15] Lindholm Tim, Yellin Frank, Bracha Gilad, and Buckley Alex. The java virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/index.html. Accessed: 11-10-2018.

[16] wwylele. Citra hardware test programs. https://github.com/wwylele/ctrhwtest. Accessed: 11-9-2018.

[17] Zophar. UltraHLE build archive. https://www.zophar.net/n64/UltraHLE.html. Accessed: 11-9-2018.