HOCHSCHULE RAVENSBURG-WEINGARTEN UNIVERSITY OF APPLIED SCIENCES

Simulating the GPU-based shaders in the graphics pipeline on a CPU-based language to allow code inspection at runtime

Masterthesis

at the University of applied science Ravensburg-Weingarten

by

Matthias Mettenleiter

September 2019

Student ID	29015
Supervisor	Daniel Scherzer
Secondary supervisor	Sebastian Mauser

Author's declaration

Hereby I solemnly declare:

1. that this Masterthesis, titled *Simulating the GPU-based shaders in the graphics pipeline on a CPU-based language to allow code inspection at runtime* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Masterthesis has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Masterthesis in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Weingarten, September 2019

Matthias Mettenleiter

Abstract

An abstract is a brief summary of a research article, thesis, review, conference proceeding or any in-depth analysis of a particular subject or discipline, and is often used to help the reader quickly ascertain the paper's purpose. When used, an abstract always appears at the beginning of a manuscript, acting as the point-of-entry for any given scientific paper or patent application. Abstracting and indexing services for various academic disciplines are aimed at compiling a body of literature for that particular subject.

The terms précis or synopsis are used in some publications to refer to the same thing that other publications might call an "abstract". In "management" reports, an executive summary usually contains more information (and often more sensitive information) than the abstract does.

Quelle: http://en.wikipedia.org/wiki/Abstract_(summary)

Contents

1 Introduction

Explanation of debugging "Debugging is the process of locating and removing faults in computer programs" according to [Collins 2014] . The steps that are part of the debugging process are reproducing the problem, identifying the source of the problem and fixing the problem. All of these steps can be done manually but there are ways to improve and accelerate this process.

To find problems there is the option of writing automated tests, inserting debug outputs on the console into the source code or writing states into log files. This enables the programmer to find anomalies before, while and after running the program.

When a way is found to reproduce the problem, to find the source of it the manual way is to increase the amount of debug outputs around the problematic part of the code and confine the point in the code at which the error occurs.

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" according to [Kernighan 1982] . As this states, debugging is a quite exhausting and time consuming task. For that reason for most programming languages there are tools to aid the programmer to narrow down the source of the bug with following methods:

- Enabling the user to set breakpoints at which the program pauses and he can inspect the values of the variables directly within the code. By continuing the program to move to the next breakpoint or by going forward through the code step by step the point where the error occurs can be found. It is also possible to be able to add conditions to the breakpoints describing a state that has to be fulfilled for the debugger to pause. [Undo 2019]
- Have the code throw an exception when unwanted behavior occurs and stop at this exception. By saving a stack of the calls which occurred before the exception was thrown or dumping the buffer, the programmer can retrace where the error may be found. [Jetbrains 2019]

- Reverse debugging records all program activities and thereby it is possible to move backwards in addition to forward stepping from a set breakpoint and see the changes in the variables and the calls in the code leading to the problem. [Undo 2019]

When the source of the problem is found the final step of fixing the problem is to correct the code.

Problem with debugging shaders in the graphics pipeline A shader is a program running on the GPU thereby mostly running as part of the graphics pipeline also known as rendering pipeline. [Khronos 2019a] The exception for this behavior is the compute shader which is independent from the graphics pipeline. [Khronos 2019b]

There are ways to debug shaders on the graphics pipeline, as shown in Section 2.1, but there are no general solutions for aiding the user in this debugging process. [Ciardi 2015]

While most CPUs are very broad in their functionality and support debugging by itself a GPU is more specialized in the way it functions and usually does not have the option to pause the code to enable inspections at runtime and does not even have access to a console or logger to write states into. [Fox 2017] As explained in more detail in Section 2.1 it is possible to enable debugging on the GPU with specialized drivers for specific hardware. [Nvidia 2019] [Microsoft 2016]

Existing approaches for compute shaders There are approaches which enable debugging of compute shader code by translating it from another language in more detail. The code is written in a debuggable language for the CPU, so it can be debugged running on the CPU. The code is then translated to the shader language and run as a compute shader on the GPU. See more in section 2.2.

Objective of creating a general solution for debuging shaders in the graphics pipeline The objective of this work is to create a general solution to enable debugging of shaders within the graphics pipeline. The goals this solution should fulfill are the following:

- The different methods shown to assist the programmer in debugging mentioned in Section 1 are usable.
- The solution is not dependent on using specific graphics cards or drivers.
- It is possible to switch between a mode where debugging is enabled and a mode where the shaders run as usual, so the program can run with the full performance and without interference of the debugger.

- The resulting output per render iteration of the debugger is close to the output of the program with the undebugged shader. It is close enough that the programmer can see what the rendering result without the debugger would look like. Errors like those resulting from the use of float variables with their inaccuracies are tolerable because there are tolerances within human perception where minimal changes in position or color within a rendered result do not matter. [Franz 2006]

- Performance is not a major requirement while running the debugger. It is possible to see the output and the values of the shader within each frame and iterate through the frames. It is not necessary to view the result in the speed of the final application while debugging. The programmer uses time to inspect the values within the shader which would not be possible with changing variables at high speeds like the 60 frames per second a usual graphics application has. [Christensson 2015] The debug application should fulfil acceptable response times for user interfaces. "10 seconds is about the limit for keeping the user's attention focused on the dialogue." according to [Nielsen 1993] .

2 Related Work

2.1 Existing methods for debugging shaders in the graphics pipeline

For debugging a shader program within the graphics pipeline there is the option to use workarounds to get the values of the variables within the code or by using special drivers provided by the producers of the hardware to get the option to debug on this hardware.

Manual debugging The manual way of debugging a shader is by creating outputs of the values within the shader program to see anomalies in their values. This can't be done by writing these values on the console or in a log file like it would be done in a CPU based application because, as mentioned in section 1, there is no access to the console or a logger within a shader program. The workaround used here is to return the values projected on the rgba-color values on the resulting image of the program. In this way the programmer can see the rough area in which the values are located on the direct output. The image can also be saved and inspected closer to get the exact values within the pixels of the resulting image. [Ciardi 2015]

Debugging with special drivers on certain hardware It is possible to install special drivers for certain graphics cards provided by their producers to enable debugging of shaders running on this hardware within dedicated environments. The two big suppliers of graphics hardware Nvida and AMD provide these debugging environments in the form of *Nvidia Nsight* and *GPU PerfStudio*. These tools can be included into different IDEs or downloaded as standalone applications. There are debugging tools for GPU debugging provided by other sources than the producers of the hardware themselves but for gaining access to the values in the hardware pipeline the drivers of this hardware have to implement specific debugging interfaces. [Microsoft 2016] Applying these tools enables the use of breakpoints and the inspection of variables within the shader code at runtime [*GLSL-Debugger*] or dumping the buffer into a file. [Microsoft 2019] The disadvantage of this method is that not all graphics cards are supported with such drivers and tools by their producer.

2.2 Approaches for debugging compute shaders on the CPU

As [Jukka 2012] states "Relatively little has been published about debugging of GPU programs in practice. Most of the best practice guides and lessons learned papers discuss GPU programming rather than debugging. Although exceptions exist" Among these exceptions are approaches to write compute shaders in other programming languages so that the version of the shader in the other language can be run on the CPU where it can be debugged. This version of the shader will then be translated to the actual shader language so it can run on the GPU with the full performance advantage. This enables the use of all tools the chosen language supports to aid debugging without depending on specific hardware or drivers.

The advantage compute shaders have in comparison with other kinds of shader programs is that they run independent of the graphics pipeline which means they can be executed by setting an input and obtaining the output like any method or program runs in regular CPU based languages. The main difference which has to be taken into account is that setting the input and obtaining the output is not done by setting parameters and getting returned the output but by writing into and reading from distinct buffers.

There are projects like [*ILGPU*] where a special syntax is created within the supported language to enable the translation of the written code as a compute shader to run on the GPU. With this it is possible to switch between running the code on the CPU where it can be debugged or running it on the GPU with optimal performance.

Other projects like [*Campy*] provide a compiler translating the full codebase to run as a compute shader on the GPU. Here it is possible to write the code in the familiar language without having to adapt special syntax rules. To debug the code, it is run within a regular compiler for the language while it will run on the GPU when it is executed with the special compiler.

2.3 Translating shaders from other languages

To run code on the GPU usually the programmer has to use one of multiple low level languages specialized in this task. To enable programmers to run code on the GPU while being able to stay at the language they are accustomed to there are multiple solutions to translate different languages into these low level languages.

There are basically three different types of programming languages. The ones that are directly compiled ahead of time (AOT), the ones that are compiled to an intermediate language which then runs in a JIT(Just in Time) compiler and the ones that are running in an interpreter. For these three types there are different approaches to translate them to a shader language. [Turton 2017]

For all of these kinds of languages, it is possible to write a compiler translating them directly to the shader language. This is what [*GLSLplusplus*] does with the usually AOT compiled language C++ when it is translated to the shader language GLSL to run on the GPU. In this project the possibility of using this to debug the shaders written in C++ is even mentioned. [*SharpShader*] is another example which uses the Language C# which is usually compiled to the intermediate language MSIL. In this example C# code is also directly compiled to a Shader language. In this case it is possible to have HLSL or GLSL as output language of the translation. In both cases the programmer can use his accustomed language with the limitation of having to write the code for the GPU within special syntax rules to be able to translate it to be translated into a shader for the GPU. An advantage that results of the direct translation is that variable names given in the written code can be retained in the shader code.

Another option that arises with the use of a language that compiles to an intermediate language is to take the resulting program in the intermediate language and further translate it into a shader language. Examples doing this are [*ILGPU*] and [*Campy*] where the shaders are written in C#. This is then compiled to the intermediate language MSIL which is then further translated to the Compute language CUDA which then runs on the GPU. Another example is [*SpirvNet*] which translates the Intermediate Language MSIL to Spirv which is a intermediate language for graphical shader code that can be interpreted as a shader on the GPU or translated further to other shader languages with tools such as for example [*SPIRV-cross*] . The translation in all of these cases is based on a uniform intermediate language which could be the result of multiple higher level languages. So by writing the translator for this intermediate language the option exists to write the shaders in different languages compilable to this intermediate language. The variable names are lost by compiling to the intermediate language which means that the variable names are not retainable in the shader code.

To get a code translated to a shader code able to run on the GPU it is also possible to chain multiple existing tools together to translate the language in which the shaders should be written in to the desired shader language. An example for this process can be seen in the way the Interpreted language R is translated through multiple steps to run on the

GPU in the example "Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation" [Juan Fumero 2017]

3 Contribution

The proposed solution of this work is inspired by the approaches for compute shaders described in section 2.2. The code for the shaders is written in a language which supports the different methods for debugging listed in 1.

To enable the functionality while debugging, the linking of the shaders and the steps in between the shaders on the graphics pipeline, usually already provided by the graphics hardware, will be simulated in the other language.

To run the shader as usual on the GPU the shaders written in the debuggable language is translated to the shader language and loaded on the hardware.

Steps for simulating the graphics pipeline

Steps for translating the shader code

4 Implementation

5 Conclusion

Fazit ziehen über das Projekt und die Arbeit. Welche Erkenntnisse wurden gewonnen? Was hat gut/schlecht funktioniert? Wurden die eigenen Erwartungen erfüllt oder nicht? War das Projekt erfolgreich?

Acronyms

CPU Central Processing Unit
GPU Graphics Processing Unit
IDE Integrated Development Environment
JIT Just in Time
AOT Ahead of Time
MSIL Microsoft Intermediate Language (also known as IL)

List of Figures

List of Tables

Bibliography

Campy. URL: http://campynet.com/ (visited on 08/24/2019).

Christensson, P. (2015). *FPS Definition*. URL: https://techterms.com/definition/fps# (visited on 08/22/2019).

Ciardi, Francesco Cifariello (2015). URL: https://computergraphics.stackexchange.com/a/101 (visited on 08/22/2019).

Collins, Harper (2014). *Collins English Dictionary – Complete and Unabridged, 12th Edition*. URL: https://www.collinsdictionary.com/dictionary/english/debugging (visited on 08/22/2019).

Fox, Alexander (2017). *MTE Explains: The Difference Between a CPU and a GPU*. URL: https://www.maketecheasier.com/difference-between-cpu-and-gpu/ (visited on 08/22/2019).

Franz, Dr. Maren (2006). *Die Subjektivität der Wahrnehmung*. URL: http://www.nlp-hh.de/img/Subjektivitaet_der_Wahrnehmung.pdf (visited on 08/22/2019).

GLSL-Debugger. *GLSL-Debugger*. URL: http://glsl-debugger.github.io/#features (visited on 08/23/2019).

GLSLplusplus. URL: https://github.com/AlexanderDzhoganov/GLSLplusplus (visited on 08/24/2019).

GPU PerfStudio. URL: https://gpuopen.com/archive/gpu-perfstudio/ (visited on 08/22/2019).

ILGPU. URL: http://www.ilgpu.net/ (visited on 08/24/2019).

Jetbrains (2019). *Debugging Exceptions*. URL: https://www.jetbrains.com/help/rider/Debugging_Exceptions.html (visited on 08/21/2019).

Juan Fumero Michael Stuewer, Lukas Stadler (2017). "Just-In-Time GPU Compilation forInterpreted Languages with Partial Evaluation". In: URL: https://michel.steuwer.info/files/publications/2017/VEE-2017.pdf (visited on 08/24/2019).

Jukka, Julku (2012). "Debugging GPU code". In: URL: https://wiki.aalto.fi/download/attachments/70779066/jjulku_debugging_gpu_code_final.pdf?version=1&modificationDate=1357205536000&api=v2 (visited on 08/24/2019).

Kernighan, Plauger (1982). *The Elements of Programming Style, 2nd edition*. McGraw-Hill. ISBN: 978-0070342071.

Khronos (2019a). *Shader*. URL: https://www.khronos.org/opengl/wiki/Shader (visited on 08/22/2019).

– (2019b). *Shader*. URL: https://www.khronos.org/opengl/wiki/Compute_Shader (visited on 08/22/2019).

Microsoft (2016). *Debugging GPU Code*. URL: https://docs.microsoft.com/de-de/visualstudio/debugger/debugging-gpu-code?view=vs-2019 (visited on 08/22/2019).

– (2019). *Using automatic shader PDB resolution in PIX*. URL: https://devblogs.microsoft.com/pix/using-automatic-shader-pdb-resolution-in-pix/ (visited on 08/23/2019).

Nielsen, Jakob (1993). *Usability Engineering*. Morgan Kaufmann. Chap. 5. ISBN: 978-0125184069.

Nvidia (2019). *Debugging Solutions*. URL: https://developer.nvidia.com/debugging-solutions (visited on 08/22/2019).

Nvidia Nsight. URL: https://developer.nvidia.com/tools-overview (visited on 08/22/2019).

SharpShader. URL: https://github.com/Syncaidius/SharpShader (visited on 08/24/2019).

SPIRV-cross. URL: https://github.com/KhronosGroup/SPIRV-Cross (visited on 08/24/2019).

SpirvNet. URL: https://github.com/Philip-Trettner/SpirvNet (visited on 08/24/2019).

Turton, Lawrence (2017). *Interpreters, compilers, JIT & AOT compilation*. URL: https://medium.com/@avelx/interpreters-compilers-jit-aot-compilation-753fb2f9014e (visited on 08/24/2019).

Undo (2019). *What is Reverse Debugging, and why do we need it?* URL: https://undo.io/resources/reverse-debugging-whitepaper/ (visited on 08/21/2019).

Appendix

A. Screenshot NameNode Web-Interface

B. DVD Inhalt

C. DVD

A. Screenshot NameNode Web-Interface

Hadoop | Overview | Datanodes | Datanode Volume Failures | Snapshot | Startup Progress | Utilities

Overview 'localhost:9000' (active)

Started:	Fri Jul 10 00:23:31 CEST 2015
Version:	2.7.0, rd4c8d4d4d203c934e8074b31289a28724c0842cf
Compiled:	2015-04-10T18:40Z by jenkins from (detached from d4c8d4d)
Cluster ID:	CID-322169a1-9f18-4284-9cfa-490bd79c1dd4
Block Pool ID:	BP-1249407956-127.0.1.1-1436480592942

Summary

Security is off.

Safemode is off.

1 files and directories, 0 blocks = 1 total filesystem object(s).

Heap Memory used 26.65 MB of 50.49 MB Heap Memory. Max Heap Memory is 966.69 MB.

Non Heap Memory used 30.99 MB of 32.25 MB Commited Non Heap Memory. Max Non Heap Memory is 214 MB.

Configured Capacity:	18.58 GB
DFS Used:	24 KB (0%)
Non DFS Used:	2.85 GB
DFS Remaining:	15.73 GB (84.67%)
Block Pool Used:	24 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	10.7.2015, 00:23:31

NameNode Journal Status

Current transaction ID: 1

Journal Manager	State
FileJournalManager(root=/tmp/hadoop-root/dfs/name)	EditLogFileOutputStream(/tmp/hadoop-root/dfs/name/current/edits_inprogress_0000000000000000001)

NameNode Storage

Storage Directory	Type	State
/tmp/hadoop-root/dfs/name	IMAGE_AND_EDITS	Active

Hadoop, 2014.

C. DVD Inhalt

```
⊢ Anwendung/
|    – pom-xml                              ⇒ Maven POM Datei
|    ⊢ conf/                                ⇒ *.properties Dateien für Konfiguration
|    ⊢ src/                                 ⇒ Quellcode Dateien
|    ⊢ target/
|    |    – Logfileanalyzer-1.0-SNAPSHOT.jar ⇒ Ausführtbare JAR-Datei
|    |    ⊢ site/apidocs/                   ⇒ JavaDoc für Browser
|
⊢ Literatur/                                ⇒ PDF Literatur & E-Books
⊢ Praesentationen/
|    – Abschlusspraesentation.pptx          ⇒ Präsentation vom 21. August 2015
|    – Abschlusspraesentation.pdf
|    – Kickoffpraesentation.pptx            ⇒ Präsentation vom 03. Juni 2015
|    – Kickoffpraesentation.pdf
|
⊢ Sonstiges/
|    – LineareRegression.xlsx               ⇒ Berechnung der linearen Regression
|
⊢ Latex-Files/                              ⇒ Editierbare LaTeX Dateien der Arbeit
     – bibliographie.bib                    ⇒ Literaturverzeichnis
     – dokumentation.pdf                    ⇒ Bachelorarbeit als PDF
     – dokumentation.tex                    ⇒ Hauptdokument
     – einstellungen.tex                    ⇒ Einstellungen
     ⊢ ads/                                 ⇒ Header, Glosar, Abkürzungen, etc.
     ⊢ content/                             ⇒ Kapitel
     ⊢ images/                              ⇒ Bilder
     ⊢ lang/                                ⇒ Sprachdateien für LaTeX Template
```