A Vulkan Graphics Driver for RISC-V CPUs

Martin Troiber[1]*

[1]School of Computation, Information and Technology, Technical University of Munich

Abstract

This report presents the first open-source Vulkan graphics driver for RISC-V. We achieved this by porting SwiftShader - a graphics driver targeting CPUs - to RISC-V. For the evaluation of our port we utilized QEMU emulation and the Allwinner D1 RISC-V chip. On RISC-V chips without a GPU this is the first time rendering 3D graphics with Vulkan is possible. This is particularly important for server CPUs or low cost RISC-V chips which are often used on single board computers.

Introduction

Open-source CPU architecture has become increasingly popular due to the success of RISC-V. The inner workings of a GPU on the other hand are currently less accessible for researchers and the general public. Even more so creating a complete open-source SoC with graphics rendering capability is currently not possible.

Therefore we do resort to a graphics driver that targets CPU cores instead of GPU cores. For our project we want to utilize SwiftShader [1] which is currently mainly developed by Google. We picked SwiftShader after going through literature research and deciding that the competing implementations are either incomplete like Kazan [2] or tied to the large Mesa 3D graphic stack like LLVMpipe. Since the completion of our project a separate effort for adding RISC-V support to LLVMpipe has started [3].

SwiftShader Architecture

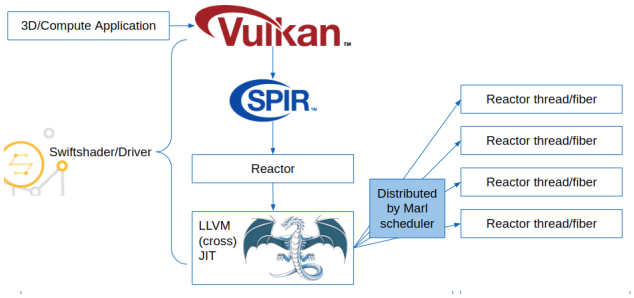

Figure 1: *SwiftShader architecture*

SwiftShader is composed of a layered architecture displayed in Figure 1. Vulkan and SPIR-V are exposed to the user. Vulkan is the low level graphics API that the user interacts with if he wants to run an application on a GPU. The actual programs that Vulkan executes on our RISC-V cores are called shaders. Before executing a Vulkan program we first have to pre-compile all shaders to SPIR-V.
This is then translated to Reactor which is SwiftShader's internal representation. The Reactor language makes use of run-time specialization which allows the generation of smaller binaries.

The compilation of the Reactor language to machine code then happens with the Just-In-Time compiler of LLVM (LLVM-JIT). [4] When LLVM-JIT gets invoked by SwiftShader it takes the representation of Reactor code from memory, compiles it to machine code for the target architecture and places the binary in another memory location. After compilation SwiftShader can use the binaries and tell the scheduler to distribute them among the CPU cores for execution.

SwiftShader uses the Marl Scheduler which allows the creation of either threads or fibers on the CPU cores. SwiftShader utilizes fibers as they are more light-weight and allow for more fine grained control over task switching.

SwiftShader Port for RISC-V

The Vulkan API, SPIR-V Shaders and Reactor did not require any RISC-V specific changes as these are high level interfaces and intermediate representations which do not directly interact with the hardware.

With our contributions it is now possible for SwiftShader to use the current development version LLVM 16 to utilize the latest JIT features for RISC-V [5].

The main reason for updating LLVM is that newer versions support more relocations in the code generated for RISC-V. Despite the significant number of relocations still missing we were able to utilize LLVM-JIT already in its current form.

Finally the linking layer within LLVM-JIT that SwiftShader uses does not support RISC-V. As a result we had to switch from the currently used RTDyldObjectLinkingLayer to the ObjectLinkingLayer which is newer and does support RISC-V [5].

*Corresponding author: m.troiber@tum.de

For Marl to be compatible with RISC-V we had to add RISC-V's register layout [5]. Marl needs to know about this information to save and switch the register content when a context switch happens.

Target Platforms

On our targets we use Debian as an operating system.

For the ease of development we started out on a RV64GC based QEMU [6] system. As a host system for QEMU we use a Ubuntu based system with an AMD Ryzen 7 PRO 4750U which features 8 cores clocked at 1.7Ghz - 4.1Ghz.

As our Single Board Computer (SBC) we picked the Sipeed Nezha which features the Allwinner D1 SoC. The chip provides a single RV64GC core clocked at 1.0Ghz. Allwinner licensed the XuanTie C906 from T-Head Semiconductor as a CPU for this SoC. The development board also offers 1 GB DDR3 memory as well as an HDMI output. It only features a Tensilica HiFi4 Digital Signal Processor (DSP) for 2D graphics applications but no 3D acceleration.

Evaluation

With respect to functional evaluation we were able to pass a majority of the test cases that SwiftShader includes [5].

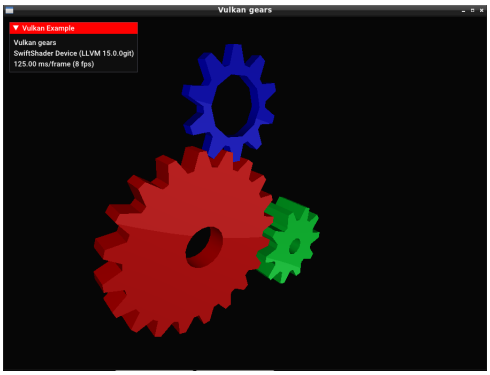

Figure 2: *Gears demo scene*

For a performance comparison we utilized the popular Vulkan demo scenes by Sascha Willems [7]. In particular we picked the 'gears' example displayed in Figure 2 as it features simple geometry and textures. In emulation and on the Nezha board we can successfully render this scene with our port of SwiftShader.

The data in Table 1 shows us that the performance of QEMU and Nezha is in the same order of magnitude. We can also see that by utilizing emulation in our use case the performance is about two orders of magnitude less than if we execute SwiftShader on the native X86 platform. We attribute this to emulation loss and the lack of SIMD support for our RISC-V setup. Similarly

Table 1: *Performance metrics on our target platforms. Gears scene rendered at 1080p resolution. Average values after a minute of execution.*

Target	Frame rate	RAM	Compute
QEMU[a]	1-4fps	120MB	80%-90%
Nezha[b]	2fps	77MB	80%
X86[a]	180fps	33MB	48%
GPU[c]	2600fps	60MB	56%

[a]AMD Ryzen 7 PRO 4750U with 8x1.7-4.1GHz.
[b]Allwinner D1 with 1x1.0GHz.
[c]AMD Radeon RX Vega 7 with 1.6GHz.

it shows that the architecture on the Allwinner D1 chip is significantly more simple than our X86 CPU. Finally we can also see that our GPU outperforms our CPU based rendering approach by an order of magnitude as well.

Table 2: *SwiftShader scaling by number of QEMU cores. Gears scene rendered at 1080p resolution. Average values after a minute of execution.*

Cores[a]	Frame rate	RAM usage	CPU usage
1	1fps	120MB	90%
2	2fps	120MB	90%
4	3fps	120MB	85%
6	4fps	120MB	83%
8	4fps	120MB	80%

[a]AMD Ryzen 7 PRO 4750U with 8x1.7-4.1GHz.

Additionally in Table 2 we analyzed how the number of QEMU cores affects the frame rate, memory usage and CPU usage. While the memory consumption of SwiftShader stays the same the frame rate does scale with the number of cores, albeit sublinearly.

References

[1] Nicolas Capens. *SwiftShader*. `https://swiftshader.googlesource.com/SwiftShader`. 2022.

[2] Jacob Lifshay. *Kazan*. `https://salsa.debian.org/Kazan-team/kazan`. 2020.

[3] Alex Fan. *Mesa merge requests: Add RISC-V support to LLVMpipe*. `https://gitlab.freedesktop.org/mesa/mesa/-/merge_requests/17801`. 2022.

[4] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[5] Martin Troiber. *Swiftshader issue tracker: Port Swiftshader to RISC-V*. `https://issuetracker.google.com/issues/217573066`. 2022.

[6] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. California, USA. 2005, p. 46.

[7] Sascha Willems. *Vulkan C++ examples and demos*. `https://github.com/SaschaWillems/Vulkan`. 2015.