Towards a Vulkan Compute Target Platform for SYCL

Peter Thoman, Daniel Gogl and Thomas Fahringer

peter.thoman@uibk.ac.at
Background and Motivation
Accelerators are becoming increasingly common and heterogeneous.

- At the high end for **increased peak performance** levels (e.g. HPC, dedicated professional and enthusiast GPUs)
- At the lower end for **increased energy efficiency** (e.g. cell phones, embedded systems)

However, **programmability** is still far from optimal.

- Ideally needs *cross-vendor, broadly applicable* options
Cross-vendor API Choices

- **OpenCL™**
  - Most widespread *compute-specific* option
  - Assumes some familiarity with *low-level HW aspects*
  - Some implementation and maintenance overhead

- **SYCL™**
  - High-level *programmability-focused* C++ API
  - Starting with SYCL2020, officially supports *non-OpenCL backends*
  - However, often no current options for running it on some embedded devices

- **Vulkan®**
  - Primarily a graphics API, but also supports compute
  - Very *widespread support on consumer HW* due to importance for graphics
  - Similar low-level implementation details requirements as OpenCL
Sylkan

- Compiler and runtime system for executing **SYCL programs** on arbitrary **Vulkan devices**
- **Goal**: extend the ease of use and programmability of SYCL to even more target platforms
- In this work:
  1. An analysis of the **semantic and engineering gap** which needs to be bridged
  2. An overview of our **prototype implementation**
  3. An initial **qualitative and quantitative evaluation** of our implementation
Semantic Mapping
Mapping from SYCL to Vulkan Compute

- Three major categories:
  1. General Terminology and Object Mapping
  2. User Code Structure and Runtime System
  3. Kernel Code

Majority of complexity in terms of required transformations
### General Terminology and Object Mapping

<table>
<thead>
<tr>
<th>SYCL / OpenCL</th>
<th>Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Instance</td>
</tr>
<tr>
<td>Device</td>
<td>Physical Device</td>
</tr>
<tr>
<td>Context</td>
<td>Device</td>
</tr>
<tr>
<td>Queue</td>
<td>Queue</td>
</tr>
<tr>
<td>Buffer</td>
<td>Buffer</td>
</tr>
<tr>
<td>Program</td>
<td>Shader Module</td>
</tr>
<tr>
<td>Kernel</td>
<td>(Compute) Pipeline</td>
</tr>
<tr>
<td>Event</td>
<td>Timeline Semaphore</td>
</tr>
</tbody>
</table>

- **Straightforward 1:1 mapping**
- **Some more complexity, details later**

- Multiple potential options here *(Timeline Semaphores)* most convenient since they allow omnidirectional synchronization.
User Code Structure and Runtime System

- **Code Structure**
  - SYCL allows for a **single-source** application targeting heterogeneous devices
  - Vulkan follows a traditional **split host/device code** model
    → Sylkan compiler infrastructure needs to extract device parts

- **Runtime System**
  - Mostly straightforward implementation work, one **key difference**:
    - SYCL tracks dependencies and performs data transfers and synchronization
    - Vulkan requires explicit manual data transfers and synchronization
    → Sylkan RT needs to implement dependency tracking etc.

For both of these issues: DPC++ runtime used as a baseline proved extremely helpful
Kernel Code

- Looks simple from high-level perspective: both OpenCL and Vulkan consume kernel code in the SPIR-V intermediate language
- However, SPIR-V capabilities very different – “kernel” and “addresses” for OpenCL, “shader” for Vulkan
- 4 main categories of differences:
  1. Addressing Model
  2. Address Spaces
  3. Calling Conventions
  4. Structured Control Flow
Kernels vs. Shaders

• Kernels allow a **physical addressing model**, while shaders are restricted to **logical addressing** → Severely restricts use of pointers

• The *CrossWorkgroup* storage class is the default in OpenCL kernels for global memory, but is not available in Vulkan shaders

• Calling convention: kernels are functions with **parameters**, while shader parameters need to be wrapped as **global structures** and bound in descriptor sets on the host

• Shaders require **structured control flow**, which introduces a variety of constraints on the CFG structure, and requires **explicit merging** of branches and loops
Sylkan Prototype Implementation
Overview

- Implementation leverages existing open source components (primarily Clang/LLVM/DPC++)
- The majority of changes were necessary in the Clang Compiler Driver and LLVM-SPIRV translator
- The Sylkan runtime plugin is entirely new
Sylkan Runtime & Vulkan Plugin

- Generally, the Sylkan runtime creates and manages objects following the previously discussed **semantic mapping**
- Automatically uses **staging buffers** for good data transfer performance
- When direct memory re-use is requested, specific alignment guarantees need to be made → can not be generally guaranteed by Sylkan for user data, falls back to staging if mis-aligned
- Omitting some details e.g. regarding extension requirements, linking and kernel lookup here, more information in the paper
Device Code Compilation

- Not an exhaustive list, but the most important steps
Device Code Compilation

Address Space Mapping
• Replace incompatible mappings early on in the compilation process
• Modify accessor compilation to maintain type information and prevent address space casts

Kernel Interface Generation
• Requires generating structural global data and descriptor sets for all kernel parameters
• Currently implemented using individual buffers for all types of parameters
Structured Control Flow

- In order to restructure the control flow graph to fulfill structured CFG restrictions, we leveraged 2 existing LLVM passes: `structurizeCFG` and `loop-simplifyCFG`.
- One issue is that these passes may generate new blocks after their respective successor.
- This is not allowed in SPIR-V – we remedied it by creating a new pass `BBsort` which topologically sorts the strongly connected components of the CFG.
Device Code Compilation

Structured Control Flow
• Although CFG has correct structure, still need to generate explicit selection operands → uses dominator tree search

Addressing Model and Pointers
• Need to replace getelementptr with OpAccessChain
• Latter is more constrained, need to backtrack along possible getelementptr chains and accumulate indices
Prototype Performance Evaluation
Benchmarks

- 3 simple benchmarks, each implemented in SYCL as well as native Vulkan:
  - **Stencil**
    - Iterative 2D heat stencil
    - Many kernel calls stress RT performance
  - **MatMul**
    - Basic dense matrix-matrix multiplication
    - Check for fundamental codegen inefficiencies
  - **PrefixSum**
    - Illustrate behaviour of two distinct interacting kernels

Two platforms: Nvidia GTX 1070 and Intel HD Graphics 530; Same host system
Sylkan and Vulkan versions on Nvidia; Sylkan, Vulkan and DPC++ OpenCL 3.0 on Intel
5 measurements of each point, mean reported
Sylkan toolchain generates more concise SPIR-V kernel code in this case
Results – Intel

Relatively large initialization overhead (very small total time)
Conclusion & Outlook
Conclusion

- Mapping SYCL to Vulkan is not straightforward, but appears to be viable
  - Some specific/newer features may not be implementable on all platforms
  - Most significant constraints are in the supported SPIR-V capabilities

- Scientific prototype implementation induces a performance penalty over native Vulkan between 5% and 50% across three simple benchmarks on two platforms
  - Faster in some cases than OpenCL-based SYCL implementation
Outlook

- Current parameter passing scheme is quite inefficient
  - This is primarily an engineering resources bottleneck, needs specialized handling of various cases
- Most significant missing feature: **local memory support**
  - Most promising implementation path (with specialization constants) requires substantial redesign of toolchain
Thank you for your attention!

peter.thoman@uibk.ac.at

https://github.com/tadeaustria/llvm/tree/syclcon2021

Contributions to this research were partially funded by the FFG INPACT project, as well as the FWF as part of the Celerity project.