ComputeAorta

A Toolkit for Implementing Heterogeneous Programming Models

Alastair Murray

Ewan Crawford

IWOCL – April 2020
Codeplay - Enabling AI to be Open, Safe and Accessible to all

**Products**

**Acoran**
Integrates all the industry standard technologies needed to support a very wide range of AI and HPC

**ComputeCpp**
C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

**ComputeAorta**
The heart of Codeplay's compute technology enabling OpenCL™, SPIR-V™, HSA™ and Vulkan™

**Addressable Markets**
High Performance Compute (HPC)
Automotive ADAS, IoT, Cloud Compute
Smartphones & Tablets
Medical & Industrial

**Technologies:** Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

**Company**
Leaders in enabling high-performance software solutions for new AI processing systems
Enabling the toughest processors with tools and middleware based on open standards
Established 2002 in Scotland with ~80 employees

**Customers**

- Broadcom
- Renesas
- Qualcomm
- CEVA
- Imagination
- ARM
- Synopsys
- Intel
Stacking Heterogeneous Standards

- **DSLs**
  - Tensorflow, Eigen, SYCL-BLAS, SYCL-DNN, OpenCV

- **Compute Cpp**
  - SYCL 1.2.1 implementation

- **Compute Aorta**
  - Enables OpenCL 1.2 and Vulkan 1.x Compute

- **Mux**
  - Abstraction layer to implement ComputeAorta on customer hardware
Introduction

• ComputeAorta’s role in the stack is:
  • To expose the full performance potential of heterogenous hardware
  • To provide standards-compliant interfaces for users of the hardware

• Requires a design that scales with minimum effort:
  • When supporting new heterogeneous programming standards
  • When supporting new customer hardware devices
  • Without comprising on performance at any stage

• This talk is about our solutions to these requirements via:
  • The intermediate abstraction layer “Mux”
  • The toolkit of components we provide to customer teams
How is ComputeAorta delivered?

Generic ComputeAorta
- Implementations of runtime APIs
- Software implementation of builtin functions
- Device agnostic optimizations
- Testing infrastructure

Customized ComputeAorta
- Generally by a project team within Codeplay
- Mux API Implementation for target hardware
- Writing device specific code/tests
- Optimizing for performance

Customer technology
- Compiler backend
- Hardware driver
- Simulator
- Any of these may be created by Codeplay
Requirements for delivery

• Absolutely no customer specific functionality in generic code.
  • ComputeAorta delivers source to multiple customers.

• Need to provide well-defined and documented interfaces between generic and custom code.
  • Expect customer teams to not modify generic ComputeAorta source.
  • But customer projects may need to extend behavior, so hooks required.

• Large amount of testing must be delivered.
  • Customer teams have confidence in correctness, so they can focus on performance.
  • Tests on external APIs are implementation independent and refactor agnostic.
  • Mux API has generic tests to validate customer specific implementation.

• ComputeAorta does modify Clang and LLVM directly.
  • Different customers use different versions, with their own modifications.
Heterogeneous standards
- OpenCL API
- Vulkan API

Toolkit of utilities
- SPIR-V Parser
- Elf Loader
- OpenCL Specific
- WFV
- DMA
- Work-item Scheduling

Testing
- UnitCL
- UnitVK
- Lit Tests
- UnitMux

Customer Device
- Host CPU

Mux
Mux is a Specification

- A specification like the OpenCL or Vulkan specifications
- Multiplexes from API to hardware device targets
- Build heterogeneous standards on top of the interface
- Implement the interface however best suits the hardware
- API is lower-level than OpenCL, similar level to Vulkan
Why a C specification?

• Well defined interface between teams as well as code
  • Separation of Intellectual Property

• Why not define a C++ interface with abstract classes?
  • Easier for humans to reason about a written specification
  • Most types in Mux are opaque, customer teams can implement them as required

• Compared to POCL-like approach, gives each customer implementation complete flexibility
  • Use hardware driver code and intrinsics
  • Optimal work-item/work-group scheduling
  • Device debugger integration
Many aspects of the Vulkan API are attractive
  • Mux was strongly influenced by Vulkan as it provides precise control to the developer
  • But a current stage we would have to extend it a lot to use as a base API

Full control over API entry points we want
  • Important to minimize work as this is done per customer project
  • Avoid stubbing out of undesired features or having to meet conformance

Don’t need to preserve backwards compatibility
  • But do need to justify changes to customer teams
  • Quicker to iterate on based on what customers need now
Performance Philosophy

• Make it possible for programmers to write code that achieves near to 100% of absolute performance on customer hardware
  • Compiler optimizations if they will work on many cases
  • Extensions for specialist cases

• Customers are interested in performance even in early stages as "will it be fast enough" is the highest risk part of a project
  • Pre-written optimizations allows demonstrations of what will be possible

• Software emulate features if required
  • Describe the consequences in the optimization guide
  • In the future, OpenCL’s deployment flexibility should provide other paths for this, but being able to run software out-of-the-box is very useful
Building on top of Mux

VK -> CL -> Level Zero

Mux

CPU -> DSP -> GPU
Kernel Languages

• Vanilla upstream Clang front-end
  • Used for OpenCL-C `clCreateProgramWithSource`
  • Get new language extensions and support for any new kernel languages
  • Can contribute bug-fixes back upstream

• SPIR-V consumption
  • Enables Vulkan, Level Zero
  • OpenCL 1.2 `cl_khr_il_program` extension or core feature in 2.1+
  • Vital for SYCL and other technologies further up the stack

• Offline compiler
  • Device programs can be compiled offline to a binary format, then loaded at runtime using our dependency-less ELF loader module
  • ComputeAorta supports building a compiler-less driver with reduced C++ dependencies, where programs can only be created from loaded binaries
Customer Extensions

A Mux target can provide API extensions to expose target-specific features. There are two types of extensions which a target can expose:

• Runtime extensions - Modify the behaviour of the runtime API
• Compiler extensions - Modify the behaviour of the OpenCL-C/SPIR/SPIR-V compiler

Registered through CMake hooks we provide rather than Mux API
VK On OpenCL Builtins

• Maths library functions matching OpenCL precision requirement used for VK.
  • Overly accurate for GLSL SPIR-V extended instructions
  • Exact for OpenCL SPIR-V extended instructions

• VK uses mangled OpenCL work-item builtins for SPIR-V compute builtins
  • `GlobalInvocationId` maps to CL `get_global_id()`
  • `LocalInvocationIndex` implemented using CL `get_local_id()`
Abstraction Distance

Maps well: Mux descriptor type for specifying data for a given kernel parameter is an exact mapping to Vulkan. CL can use it to implement `clSetKernelArgs` efficiently but requires more code.

Maps awkwardly: Mux semaphore type for synchronisation can be used for OpenCL events, but Vulkan synchronisation is more fine-grained, and Level Zero events are even more complex.

Take the most constrained model in the case of subtly difference behaviours.

- OpenCL queues execute commands in-order by default
- Vulkan queues execute commands in arbitrary order
- Mux commands are thus executed as if in-order
Synchronization Primitives

- **VkFence**
  - One way coordination from device to host

- **VkSemaphore**
  - Device only synchronization

- **VkEvent**
  - Bi-directional synchronization between device and host

***Mux Semaphore***

- One way coordination from device to host
- Device side synchronization
Implementing Mux

VK

CL

Level Zero

Mux

CPU

DSP

GPU

© 2020 Codeplay Software Ltd.
“Host” Reference CPU Implementation

- Kernel and host code run under same process
- Guide to customer teams on implementing Mux
- CPU can be desktop CPU or housekeeping CPU on heterogenous device
- Test toolkit components
- Cross compile then run natively or on emulator
- Track performance
- Easy to debug
Command Batching

Problem On embedded devices building a command stream accounts for a significant expenditure of resources.

APIs ideally provide a way to reuse command streams

✓ Vulkan exposes command buffers
✓ Level Zero exposes command lists
• OpenCL has currently has no mechanism to achieve this
✓ Mux exposes command groups
Batching OpenCL Commands

Solution Build up Mux command groups by pushing OpenCL enqueue calls until a blocking event or flush, incurring a dispatch.

CL concept of pending dispatches, where additional commands can be pushed when a wait list contains compatible cl_events. A pending dispatch tracks:

- Mux command group
- Associated wait & signal Mux semaphores
- Associated wait & signal cl_events
- Callbacks to invoke upon completion
### Analysis of OpenCL wait list

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No wait events</td>
<td>Push command to the current command group or the last pending dispatch</td>
</tr>
<tr>
<td>Wait events associated with a single pending dispatch</td>
<td>Push command to the associated command group</td>
</tr>
<tr>
<td>Wait events associated with multiple pending dispatches</td>
<td>Get an unused command group</td>
</tr>
<tr>
<td>Wait events with no associated pending dispatches (already dispatched)</td>
<td>Get an unused command group</td>
</tr>
</tbody>
</table>
Toolkit Modules

- Whole Function Vectorizer
- STL alternative containers
- Maths Library
- ELF Loader
- SPIRV to LLVM IR translator
Maths Library

- C++ Maths Library
- Clang
- .bc Bitcode
- Embed in binary
- Shared Library
- Load from .data section at runtime
VECZ – SPMD Vectorizer

![VECZ Diagram](image-url)
VECZ – SPMD Vectorizer

• Computing multiple work-items in parallel does not depend on special patterns like loops which not all kernels contain

• Configurable by customer team for hardware traits
  • Vector width for SIMD packets
  • Optimizations, e.g. Branch On Superword Condition Code
  • Always enable, or enabled based on heuristic cost model

• See 2015 LLVM developers meeting talk “Creating an SPMD Vectorizer for OpenCL with LLVM”
Testing

- Automation
- Continuous integration
- Most testing runs on every branch before merge.
- Long-running tests run nightly.

- Khronos official
- Very strict on math precision
- Not strict on compiler accuracy
- No negative testing (i.e. doesn’t check for valid errors)

Jenkins

- Developed by Codeplay
- Checks error handling
- Primary regression test suite for ComputeAorta.

OpenCL CTS

- Like UnitCL, but for the Mux specification.
- ComputeAorta-specific
- Checks Mux implementations against the Mux specification.

UnitCL

- Highly targeted compiler tests.
- Used to ensure that compiler passes have desired effect.
- Used heavily to test that debug info is preserved.

UnitMux

- Khronos official

Lit

- Like UnitCL, but for the Vulkan specification.
- Covers SPIR-V extensively.

Vulkan CTS

- CLSmith etc.
- Tests the compiler very thoroughly
- Open source
- Creates random compiler tests

UnitVK

Fuzzing

- Open source
- Creates random compiler tests
Ecosystem

Arrayfire | Babelstream | Boost-compute | clBLAS | CLBlast | clGPU | clFFT
---|---|---|---|---|---|---
clRNG | Cloverleaf | clSPARSE | ComputeApps | ComputeCpp SDK | Eigen | Glow
Halide | opencl-book-samples | OpenCV | SYCL-DNN | piglit | Polybench | pyOpenCL

SYCL-BLAS | SYCL-DNN | Tensorflow | TVM | VexCL | ViennaCL
Conclusion

• Supporting multiple heterogeneous standards on a variety of devices requires making use of reusable components where possible, and allowing custom ones where it is not.

• Hard to predict in advance which parts will need to be customized, as exposing heterogeneous hardware capabilities is complicated. So follow an approach that allows flexibility.

• We do this primarily through the Mux API, allowing us to:
  ✓ Minimize effort to implement a heterogeneous API on a new device
  ✓ Enable high-performance programs on customer hardware
  ✓ Scale to supporting new standards