codeplay

Enable AI & HPC to be Open, Safe and Accessible to All

User-Driven Online Kernel Fusion for SYCL

Víctor Pérez, Lukas Sommer, Victor Lomüller, Kumudha Narasimhan, Mehdi Goli

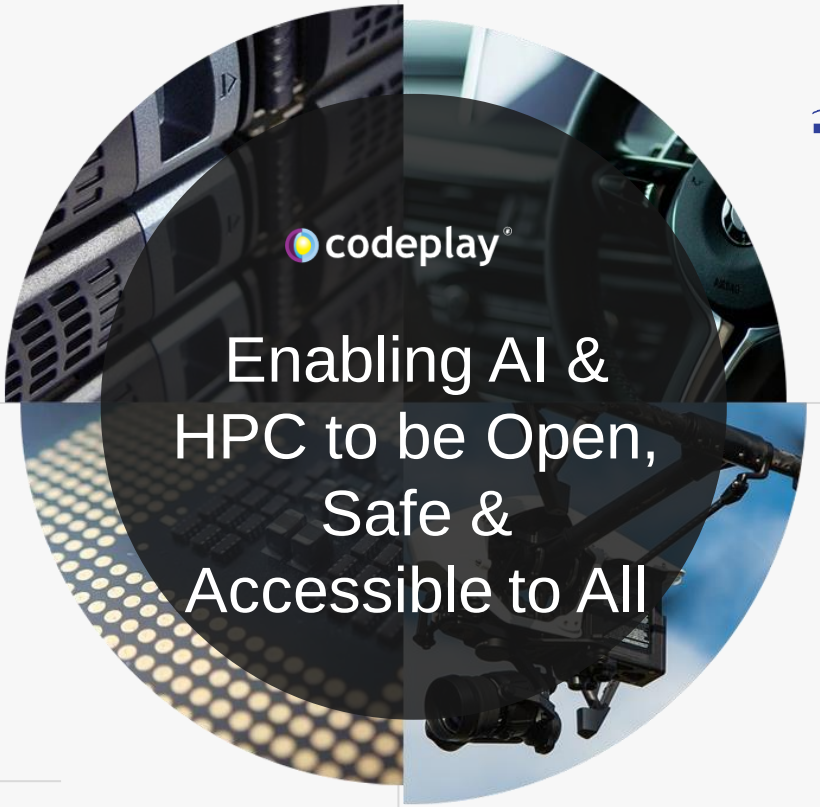
23rd August 2022

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, acquired by Intel in 2022 and now ~90 employees.



Partners

intel, BROADCOM, SYNOPSYS, CEVA, Imagination, RENESAS, KUC Kyoto Microcomputer Co., Ltd., NSI-TEXE, BERKELEY LAB, OAK RIDGE National Laboratory, Argonne NATIONAL LABORATORY

And many more!

Supported Solutions

oneAPI

An open, cross-industry, SYCL based, unified, multiarchitecture, multi-vendor programming model that delivers a common developer experience across accelerator architectures

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™

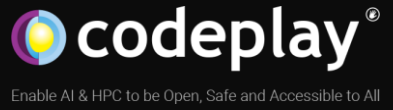
Markets

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

Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

Agenda

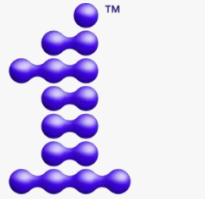
- What is SYCL?
- Introduction to kernel fusion
- Motivation
- SYCL extension design
- Implementation
- Case study 1: SYCL-DNN integration
- Case study 2: ONNX Runtime integration
- Conclusions



What is SYCL?

What is SYCL?

- An open standard heterogenous programming API introduced by Khronos
  - Using ISO standard C++ code
- Provides single-source programming model for accelerator processors
- Allow accessing both high-level and low-level code
- Multiple implementations
  - ComputeCPP
  - DPC++
  - hipSYCL



oneAPI

codeplay

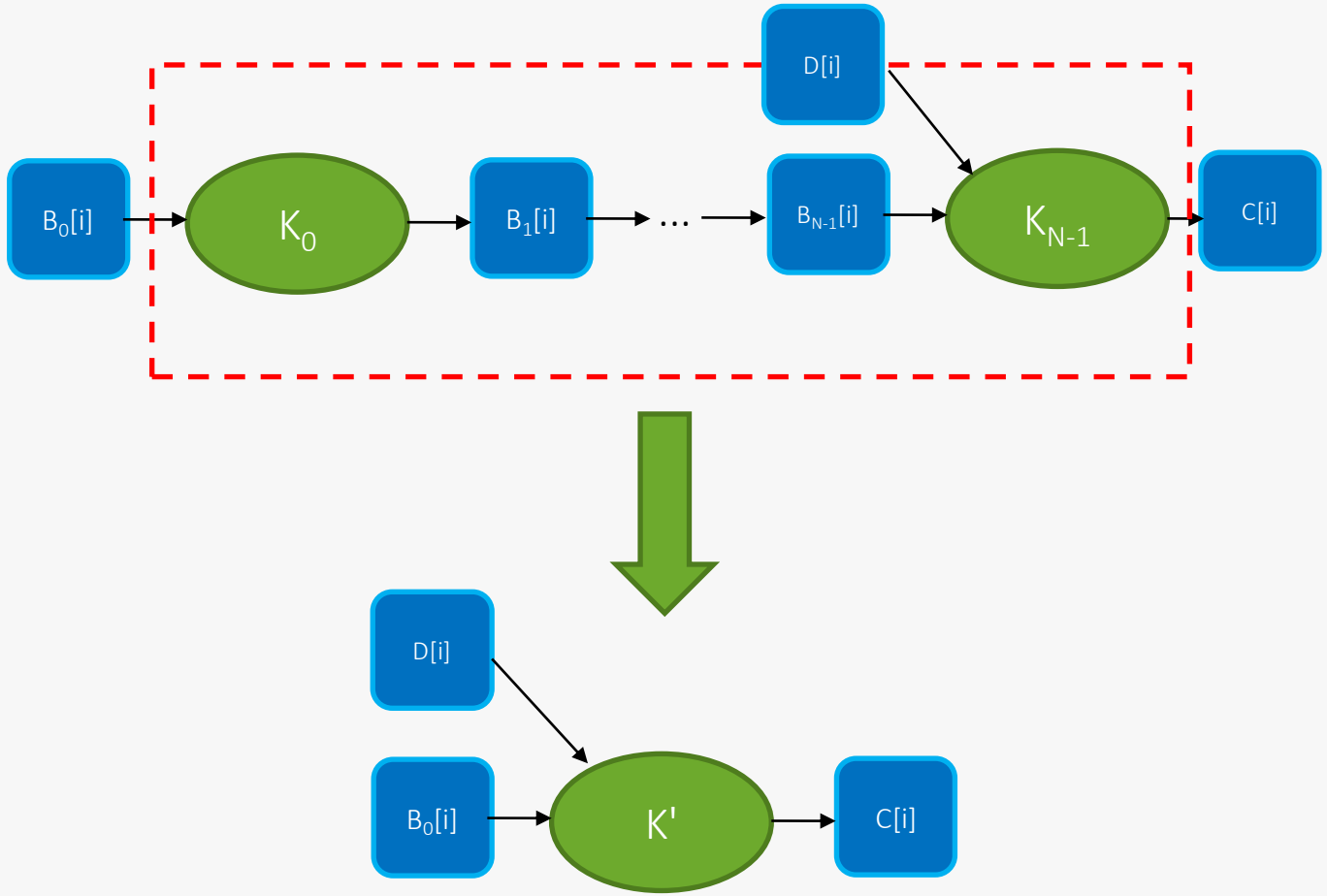
Enable AI & HPC to be Open, Safe and Accessible to All

Motivation

Motivation: Why use fusion?

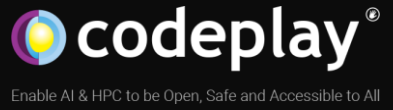
- **Short-running kernels** can affect performance.
- **Manual fusion** is already used in some domains.
  - Too much work.
  - Error-prone.
  - Domain-specific.
- Instead, **extend** the **SYCL API** to enable **user-driven, automatic, online kernel fusion.**

Kernel Fusion Example



- N-1 fewer kernels launches
- N-1 fewer buffer allocations
- 2(N-1) fewer memory accesses
- Further optimizations in the fused kernel, e.g., control flow optimization

  - Cache performance may vary
  - May run short of device resources



SYCL Extension Design

Extension Requirements

- **Accuracy**
- **Optimization**
- User-driven, but the SYCL runtime makes the final decision
- Use of runtime-available information for optimization

Extending `sycl::queue`

```
class property::queue::enable_fusion;
class property::promote_local;
class property::promote_private;
class property::no_barriers;

void queue::start_fusion();
void queue::cancel_fusion();
event queue::complete_fusion(const property_list &props = {});
```

Kernel Fusion: Minimal Overhead

```
queue q{gpu_selector{}, {property::queue::enable_fusion{}}};
{
  buffer<float> buffer1{data1, range};
  buffer<float> buffer2{data2, range,
    {property::promote_private{}}};
  q.start_fusion();
  q.submit(…);
  q.submit(…);
  …
  q.complete_fusion({property::no_barriers{}});
}
```

Kernel Fusion: Minimal Overhead

```
queue q{gpu_selector{}, {property::queue::enable_fusion{}}};
{
  buffer<float> buffer1{data1, range};
  buffer<float> buffer2{data2, range,
    {property::promote_private{}}};
  q.start_fusion();
  q.submit(…);
  q.submit(…);
  …
  q.complete_fusion({property::no_barriers{}});
}
```

Kernel Fusion: Minimal Overhead

```
queue q{gpu_selector{}, {property::queue::enable_fusion{}}};
{
  buffer<float> buffer1{data1, range};
  buffer<float> buffer2{data2, range,
    {property::promote_private{}}};
  q.start_fusion();
  q.submit(…);
  q.submit(…);
  …
  q.complete_fusion({property::no_barriers{}});
}
```

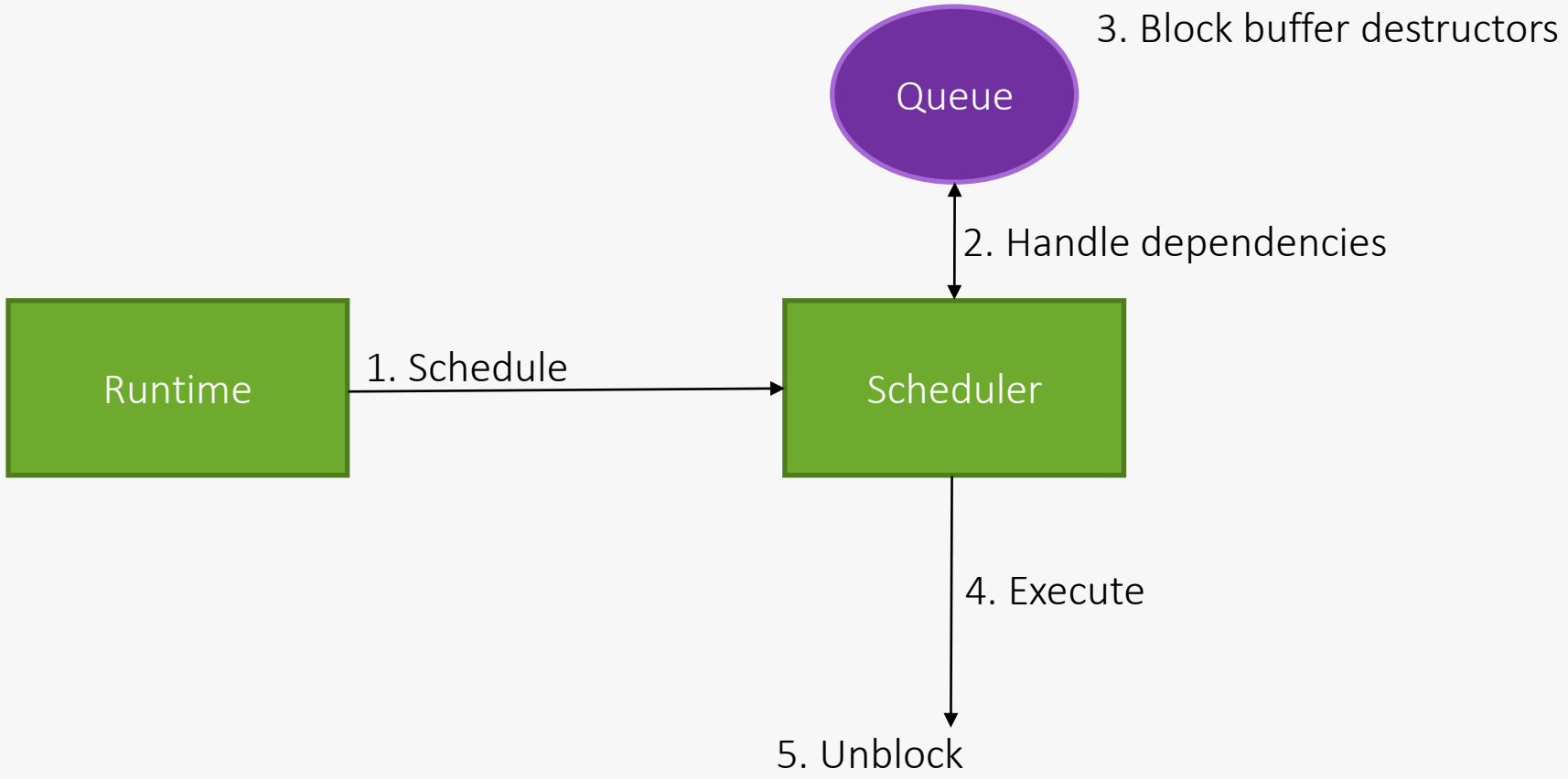
Kernel Fusion: Minimal Overhead

```
queue q{gpu_selector{}, {property::queue::enable_fusion{}}};
{
  buffer<float> buffer1{data1, range};
  buffer<float> buffer2{data2, range,
    {property::promote_private{}}};
  q.start_fusion();
  q.submit(…);
  q.submit(…);
  …
  q.complete_fusion({property::no_barriers{}});
}
```

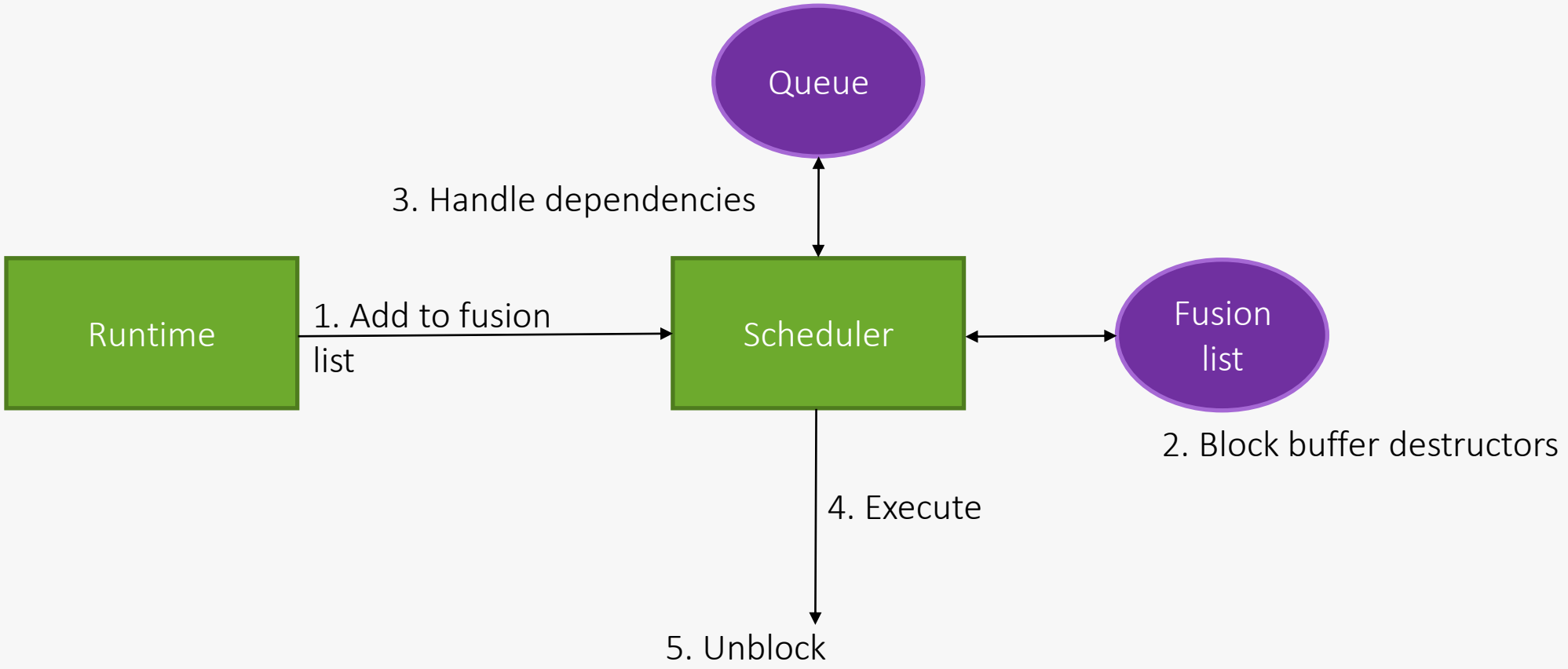
Kernel Fusion: Minimal Overhead

```
queue q{gpu_selector{}, {property::queue::enable_fusion{}}};
{
  buffer<float> buffer1{data1, range};
  buffer<float> buffer2{data2, range,
    {property::promote_private{}}};
  q.start_fusion();
  q.submit(…);
  q.submit(…);
  …
  q.complete_fusion({property::no_barriers{}});
}
```

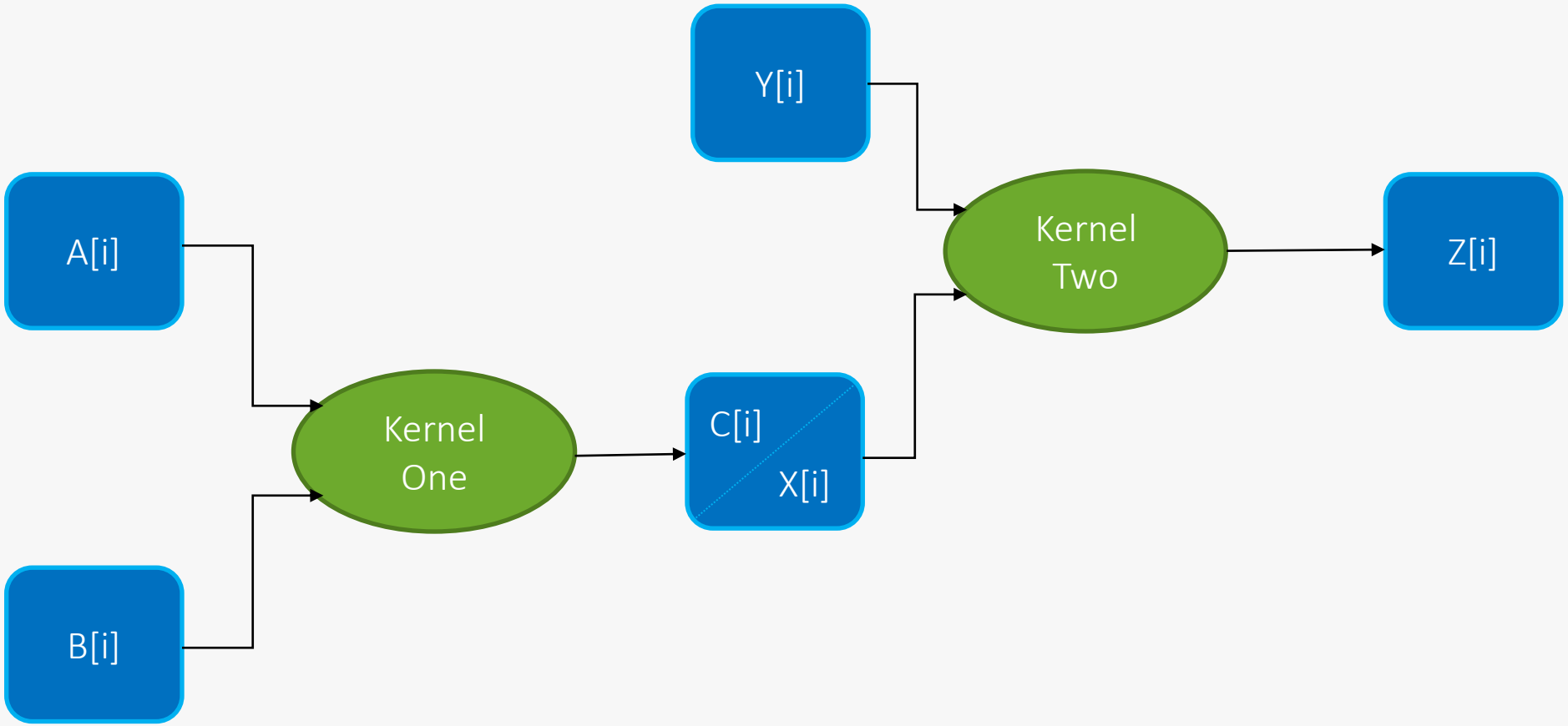
Kernel Scheduling



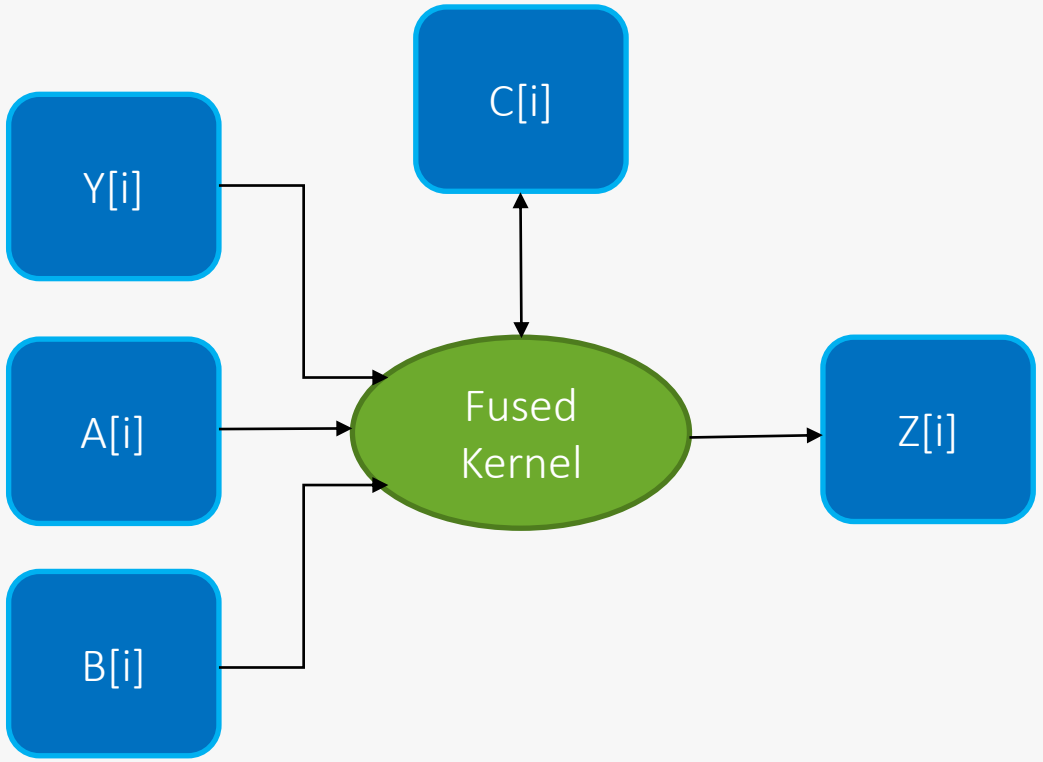
Kernel Fusion



Unfused Kernels Dataflow

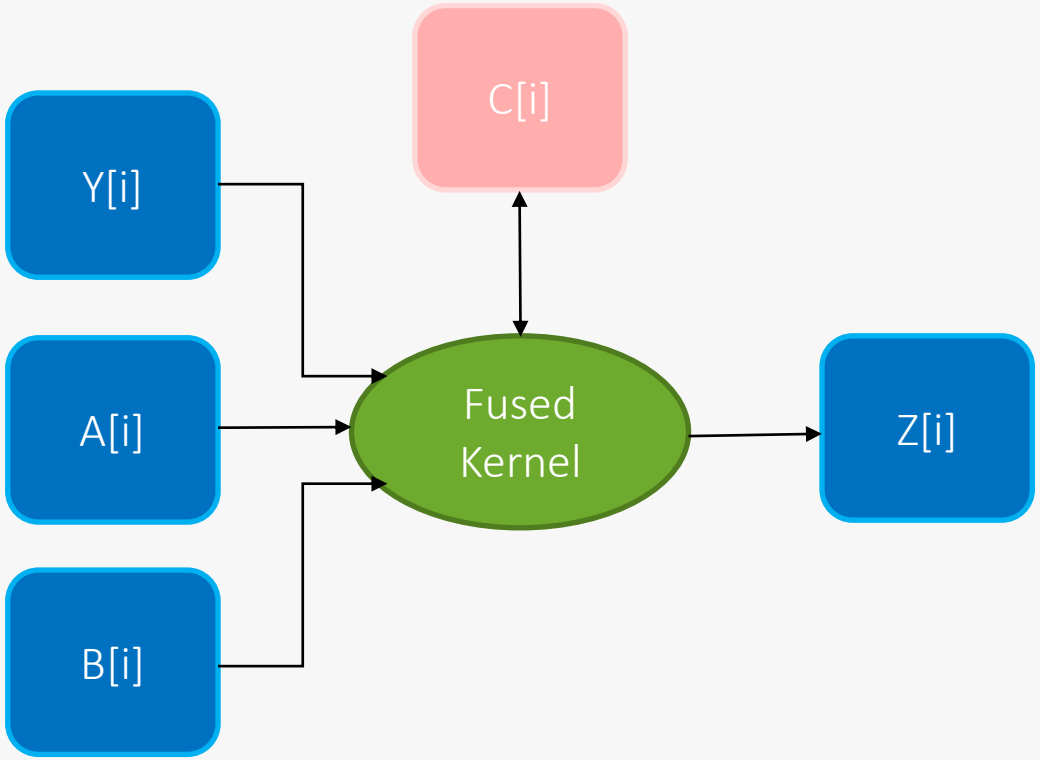


Fused Kernel Dataflow

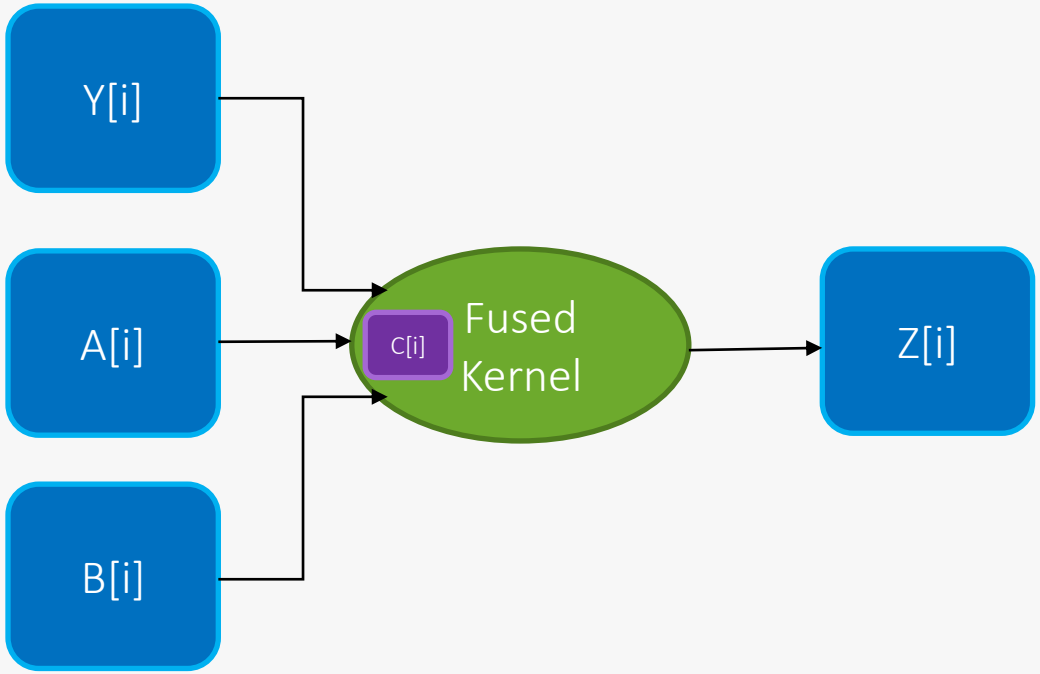


Fused Kernel Dataflow Using Internalization

Local Internalization
`sycl::property::promote_local`

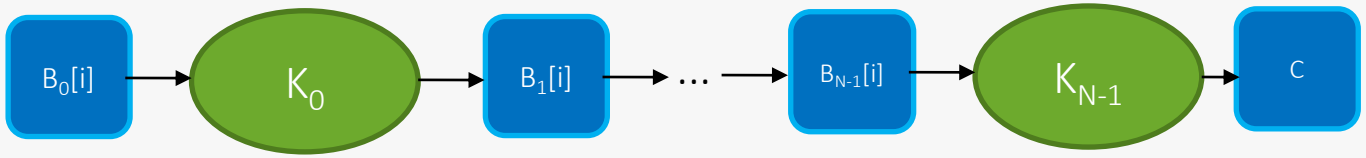


Private Internalization
`sycl::property::promote_private`

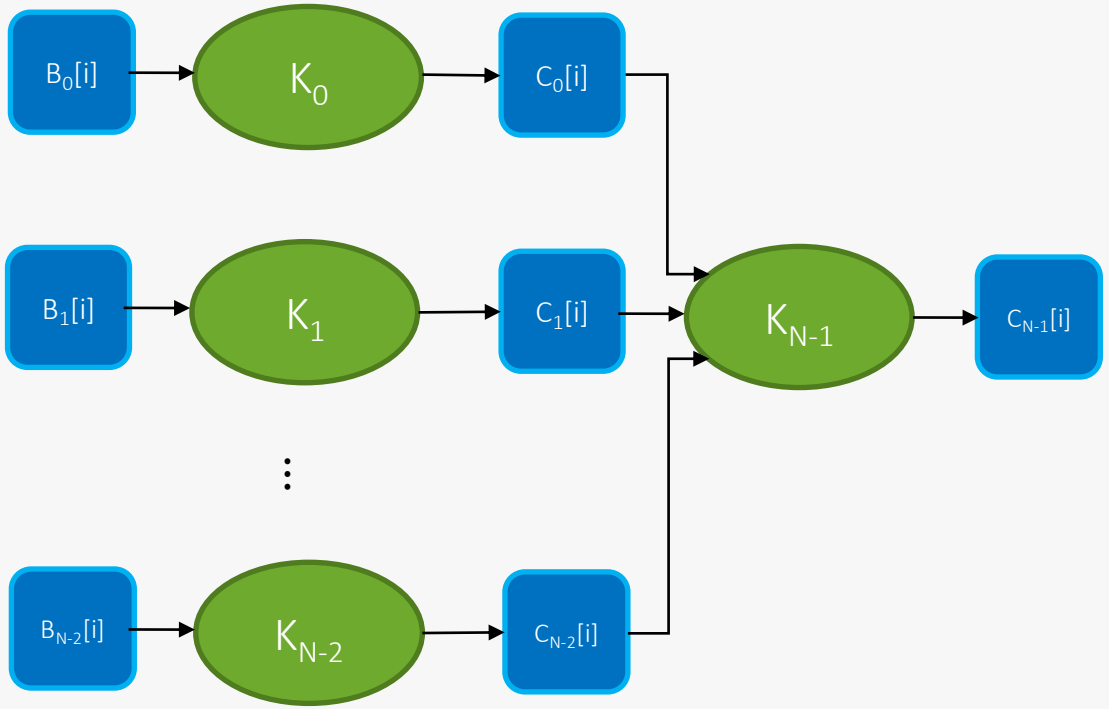


Two Kinds of Data Internalization

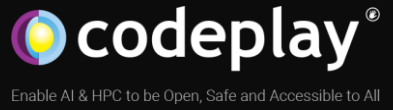
Vertical Internalization



Horizontal Internalization

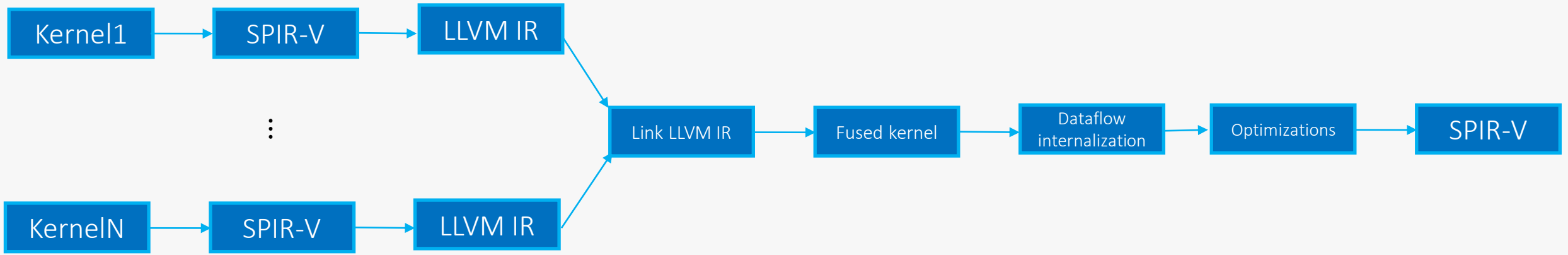


In horizontal internalization, more resources are needed, as results from different kernels must coexist.



Implementation

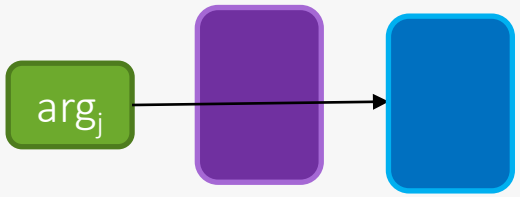
JIT Compilation



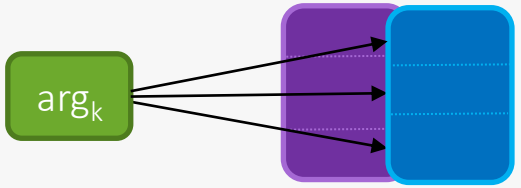
Dataflow Internalization: Overview

1.

2.

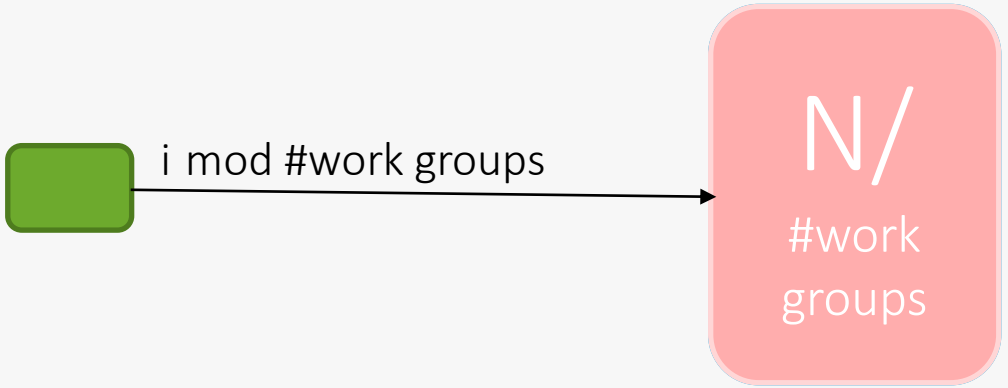


3.

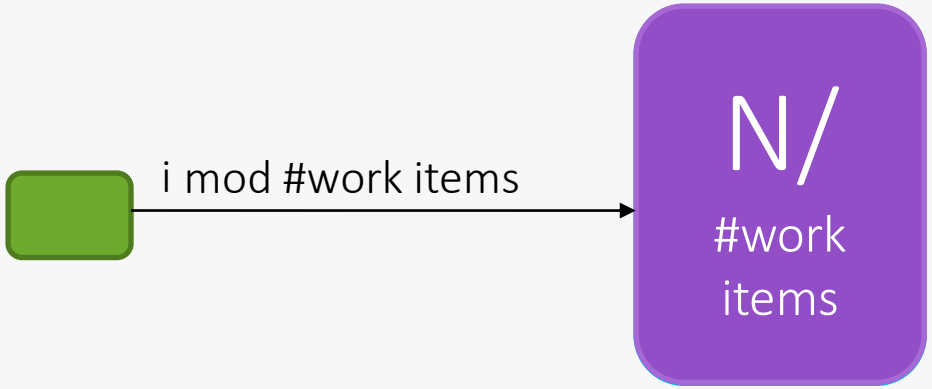


Dataflow Internalization: Remapping

Local Internalization

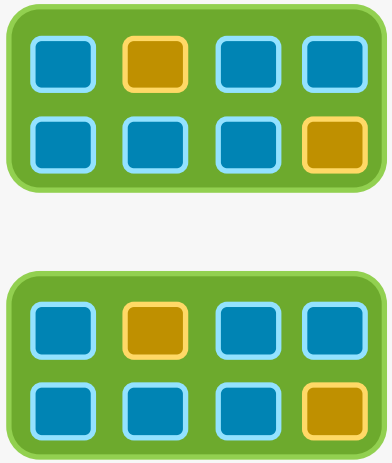


Private Internalization

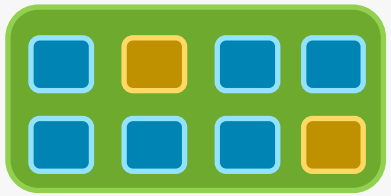


The Kernel Fusion Pipeline

0. Original kernels

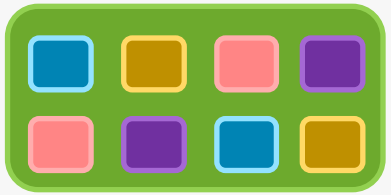


1. Fusion

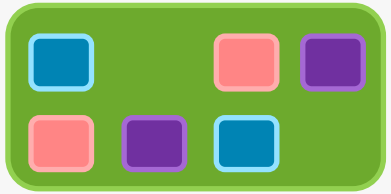


130 instructions,
11 blocks

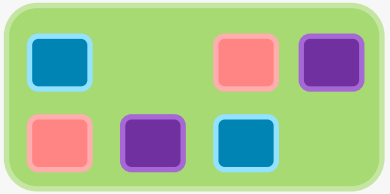
2. Dataflow internalization



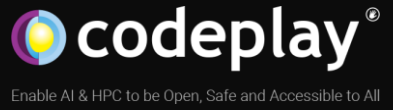
3. SYCL constants propagation



4. Other passes
(SROA, Mem2Reg, jump threading, etc.)



21 instructions,
1 block



Case study 1: SYCL-DNN Integration

SYCL-DNN

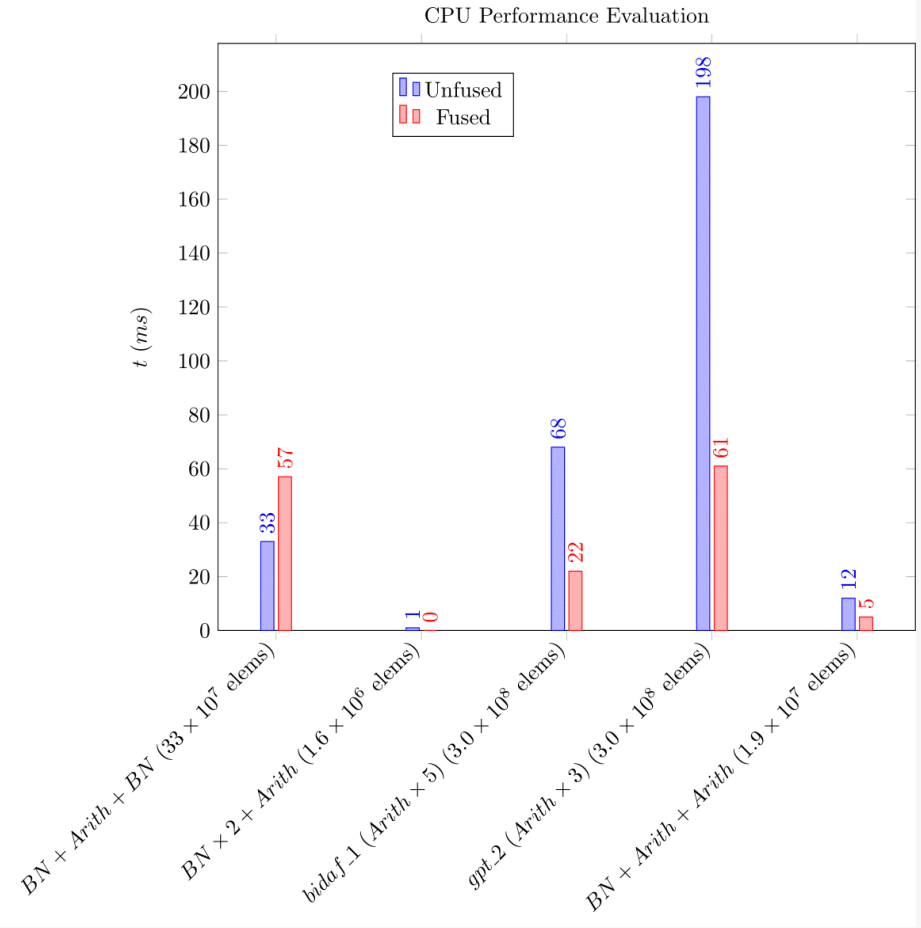
- Deep neural networks (DNN) represented as graphs.
- Enables running DNNs using SYCL.
- Lacks information for graph-based optimizations:
  - kernel fusion
  - graph partitioning
  - memory reusability

Evaluation setup

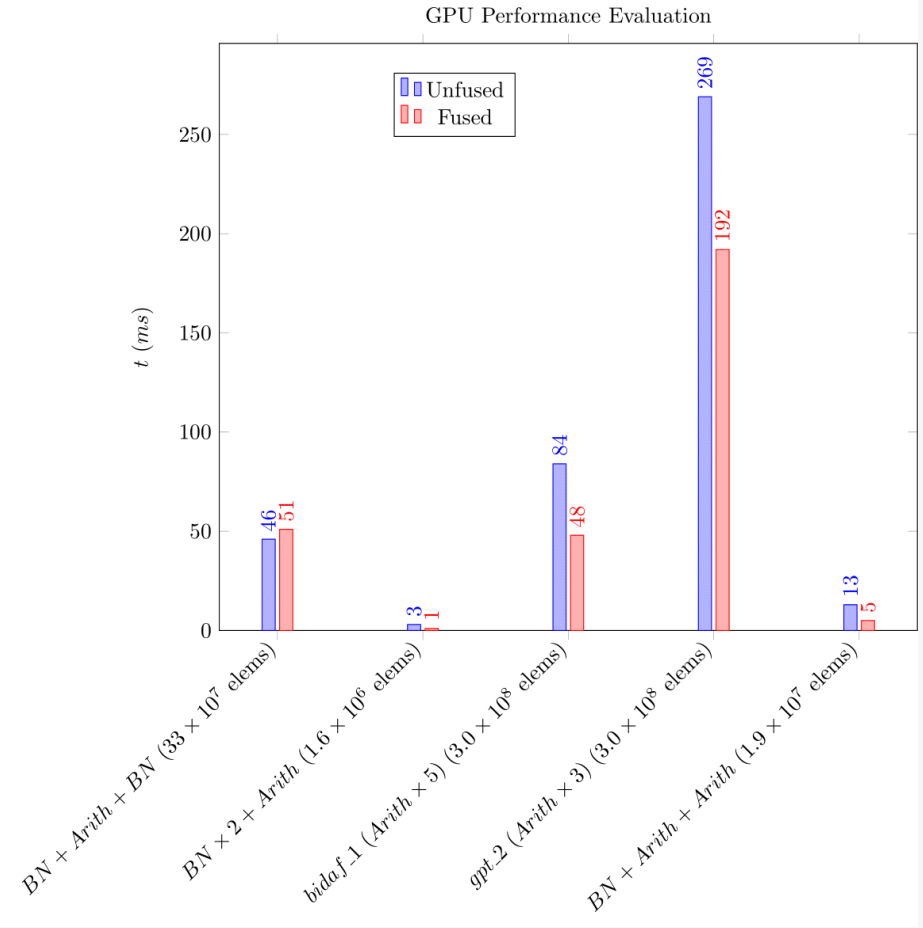
- Several microbenchmarks (arithmetic-heavy kernel fusion, BatchNormalization+Relu...).
- ResNet-50 and VGG16 complete runs (also run forcing a less efficient, more fusion friendly convolution algorithm).

Device type	Model	OpenCL driver Version	OS	SYCL Compiler Version
CPU	Intel i7-6700K	18.1.0.0920	Ubuntu 18.04.6 Kernel 4.1.50	ComputeCpp-PE 2.9.0
GPU	Intel Gen9 HD Graphics NEO	19.41.14441		

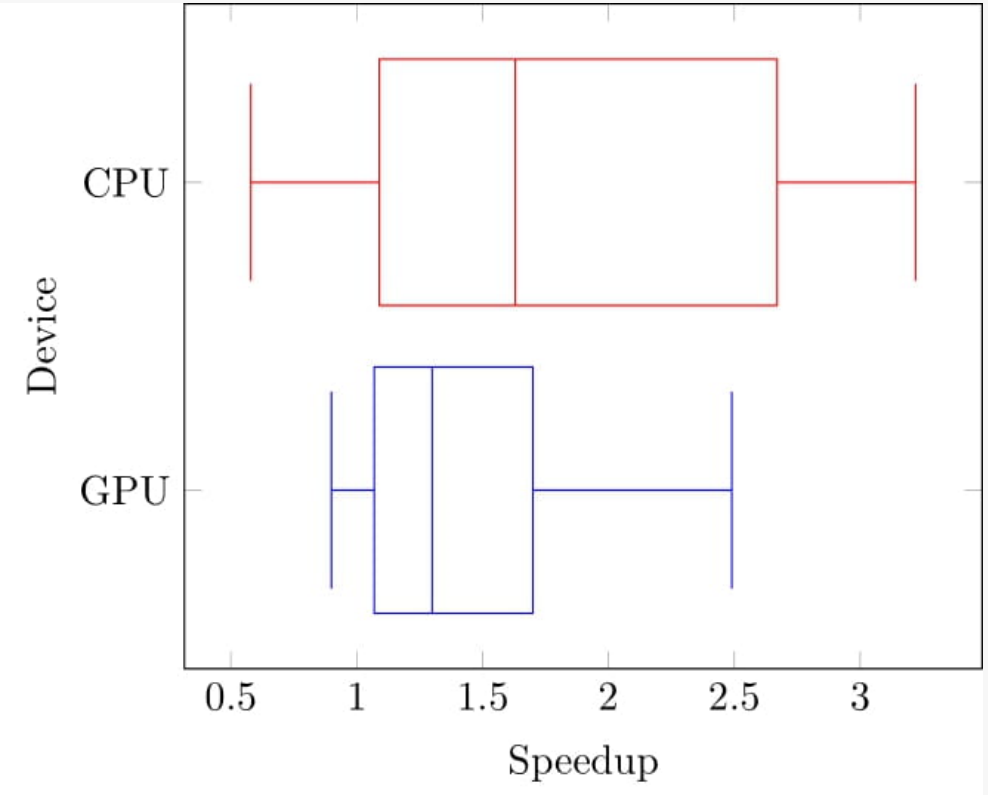
Microbenchmarks Results



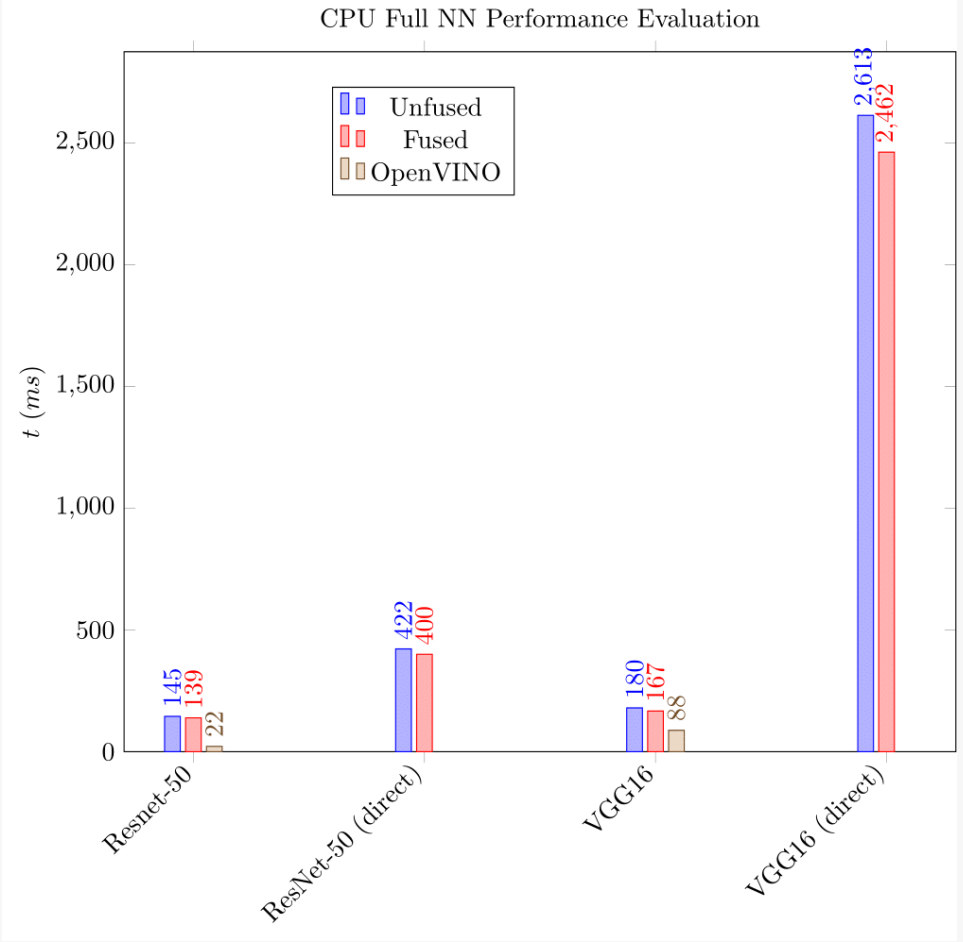
Microbenchmarks Results



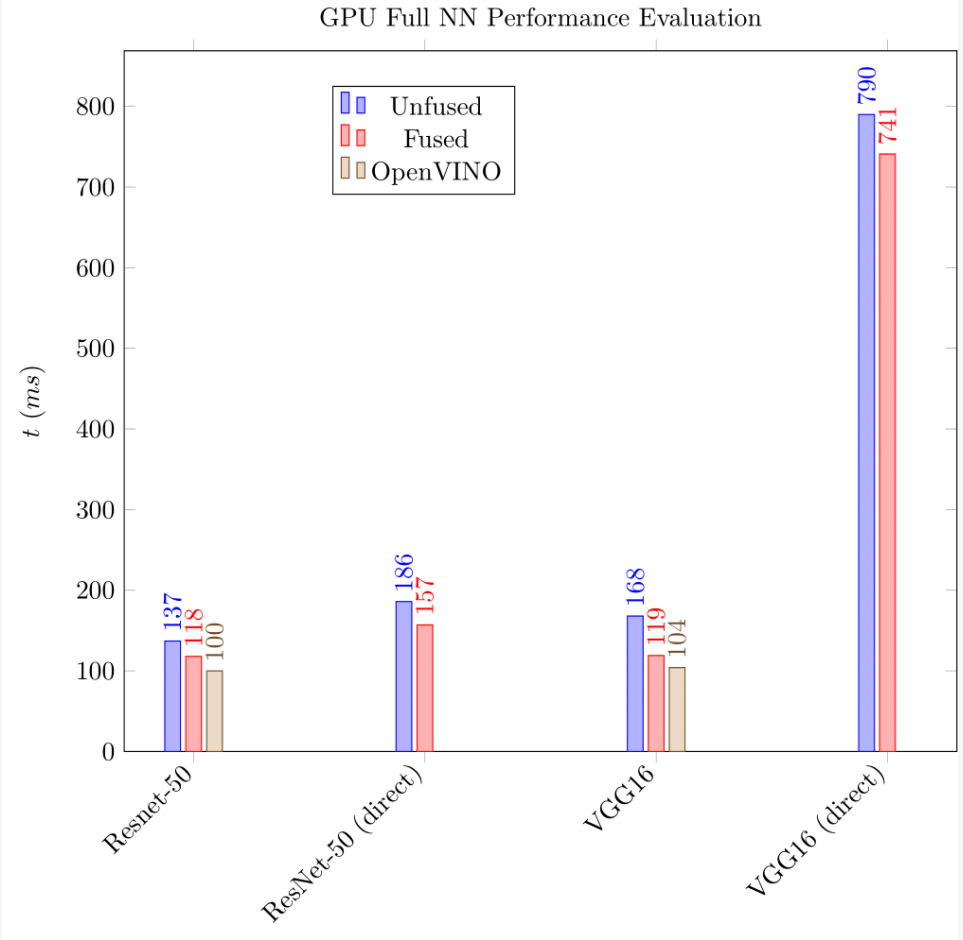
Microbenchmarks Results



Full NN Results



Full NN Results



codeplay

Enable AI & HPC to be Open, Safe and Accessible to All

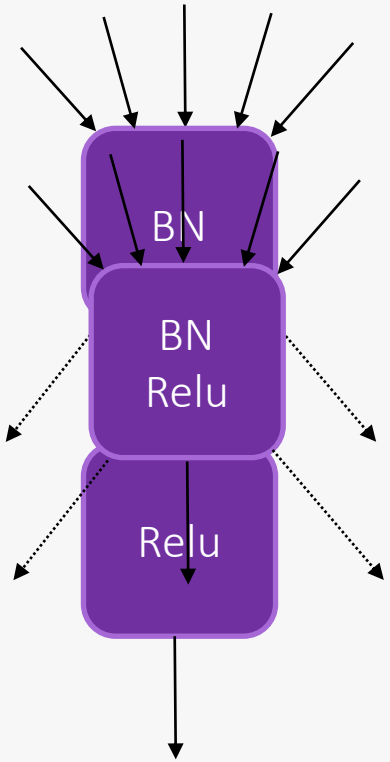
Case study 2: ONNX Runtime Integration

ONNX Runtime

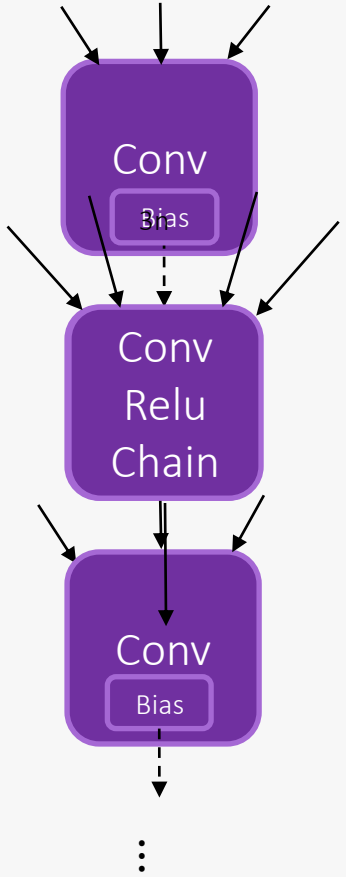
- DNN represented as graphs.
- Different execution providers:
  - CUDA
  - OpenVINO
  - SYCL
- Graph optimizations are applied before running the models.

Our Subgraph Fusions

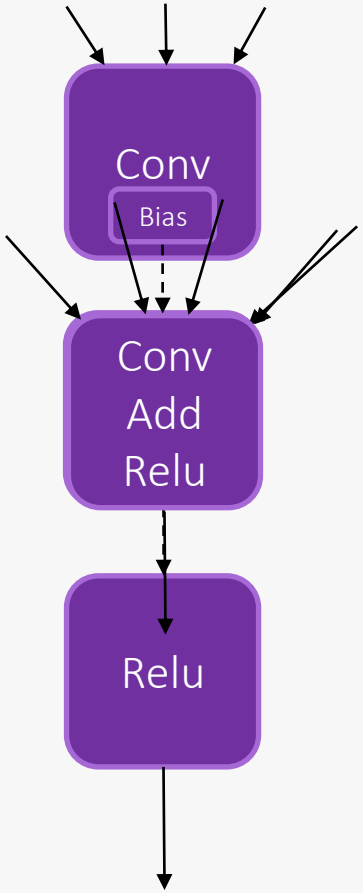
BatchNormalization Relu Fusion



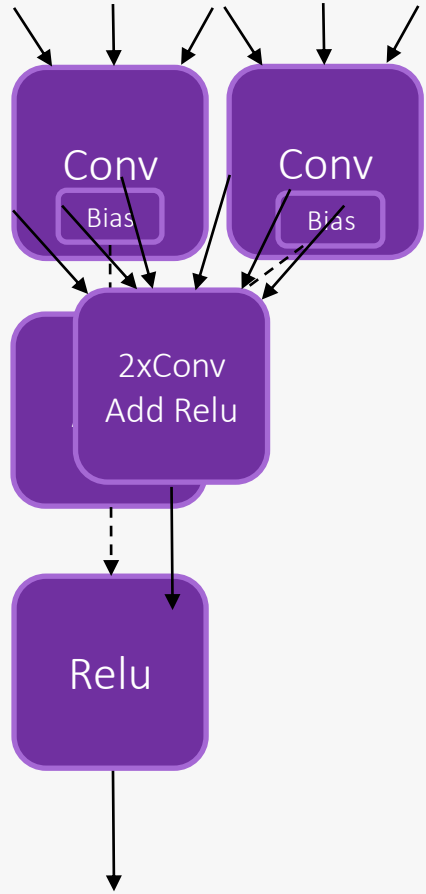
Conv Relu Chain Fusion



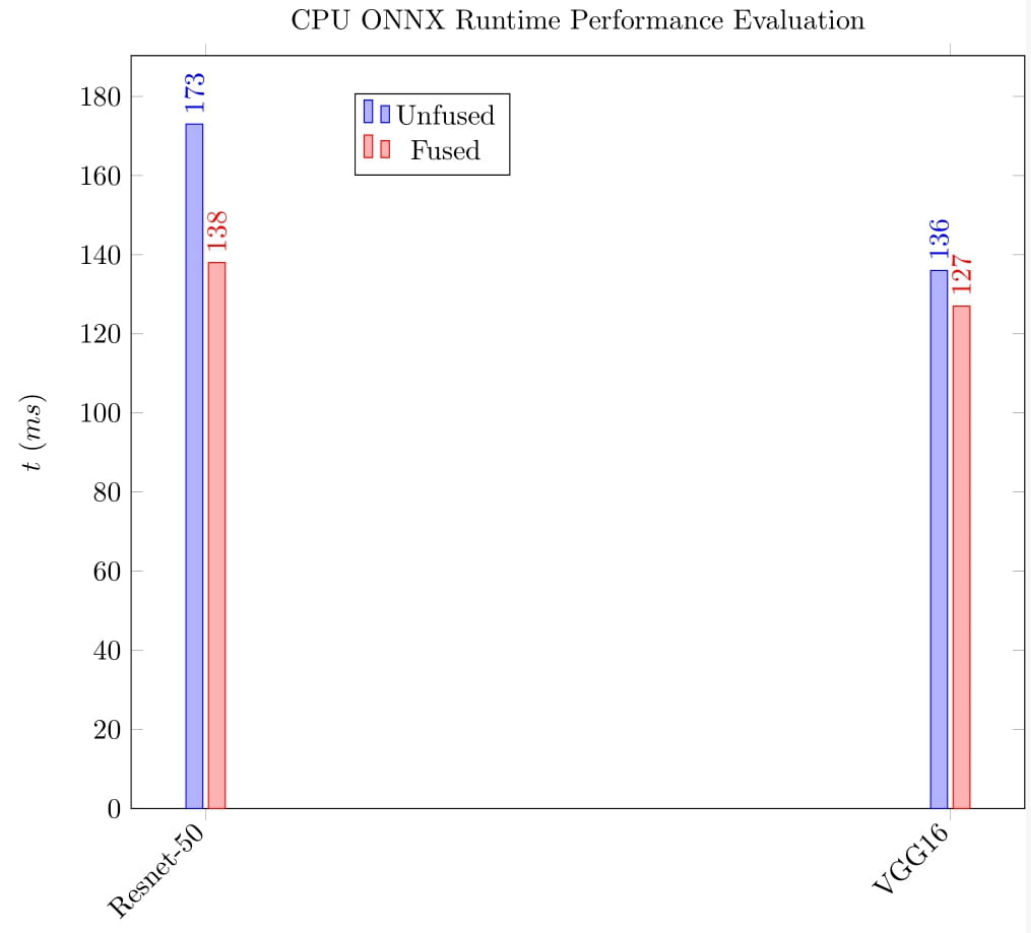
Conv Add Relu Fusion

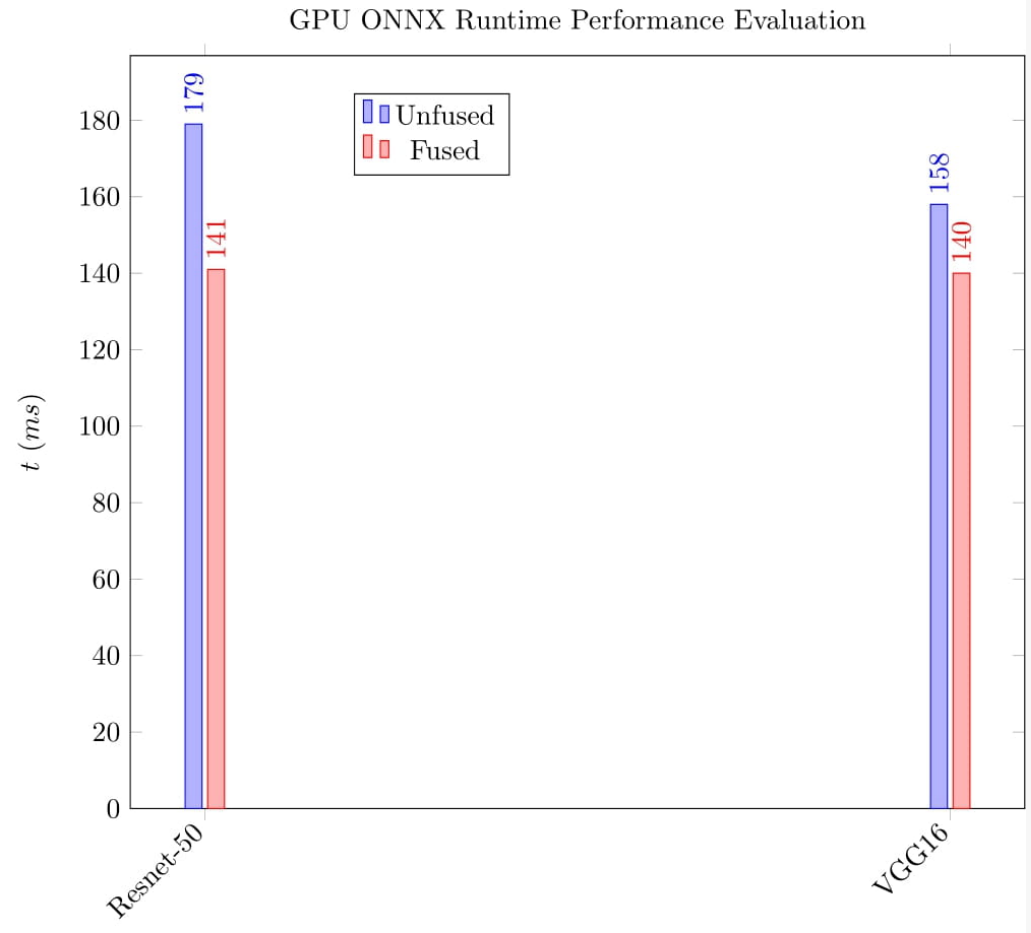


2xConv Add Relu Fusion



Results





codeplay

Enable AI & HPC to be Open, Safe and Accessible to All

Conclusions and Future Work

Conclusions

- SYCL-DNN:
  - GPU:
    - **VGG16:** x1.41
    - **Microbenchmarks:** >x2.00
  - CPU:
    - **Private internalization:** >x3.00
- ONNX runtime:
  - **ResNet-50:** Horizontal internalization

Future Work

- Make fusion applicability evident
- Single `property::internalize`
- Fusing kernels with different ND-ranges.
- Explore new applications

We're Hiring!

codeplay.com/careers/

codeplay

Enabling AI to be Open, Safe & Accessible to All

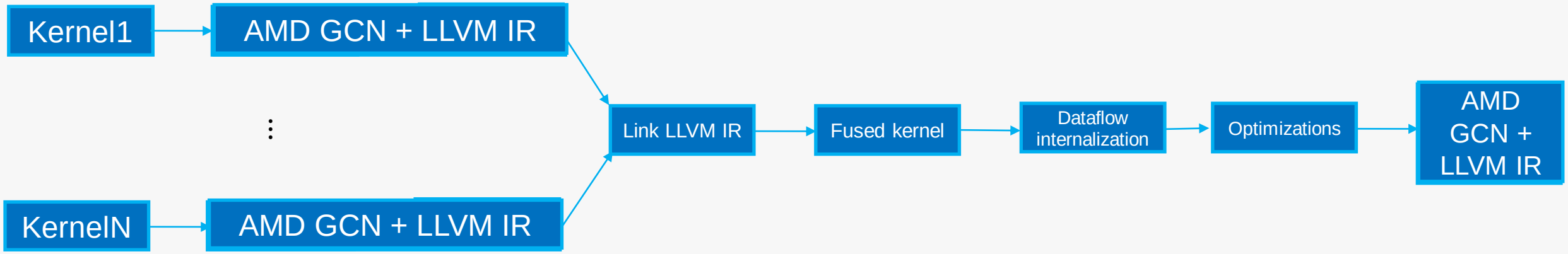
Any questions?

@codeplaysoft

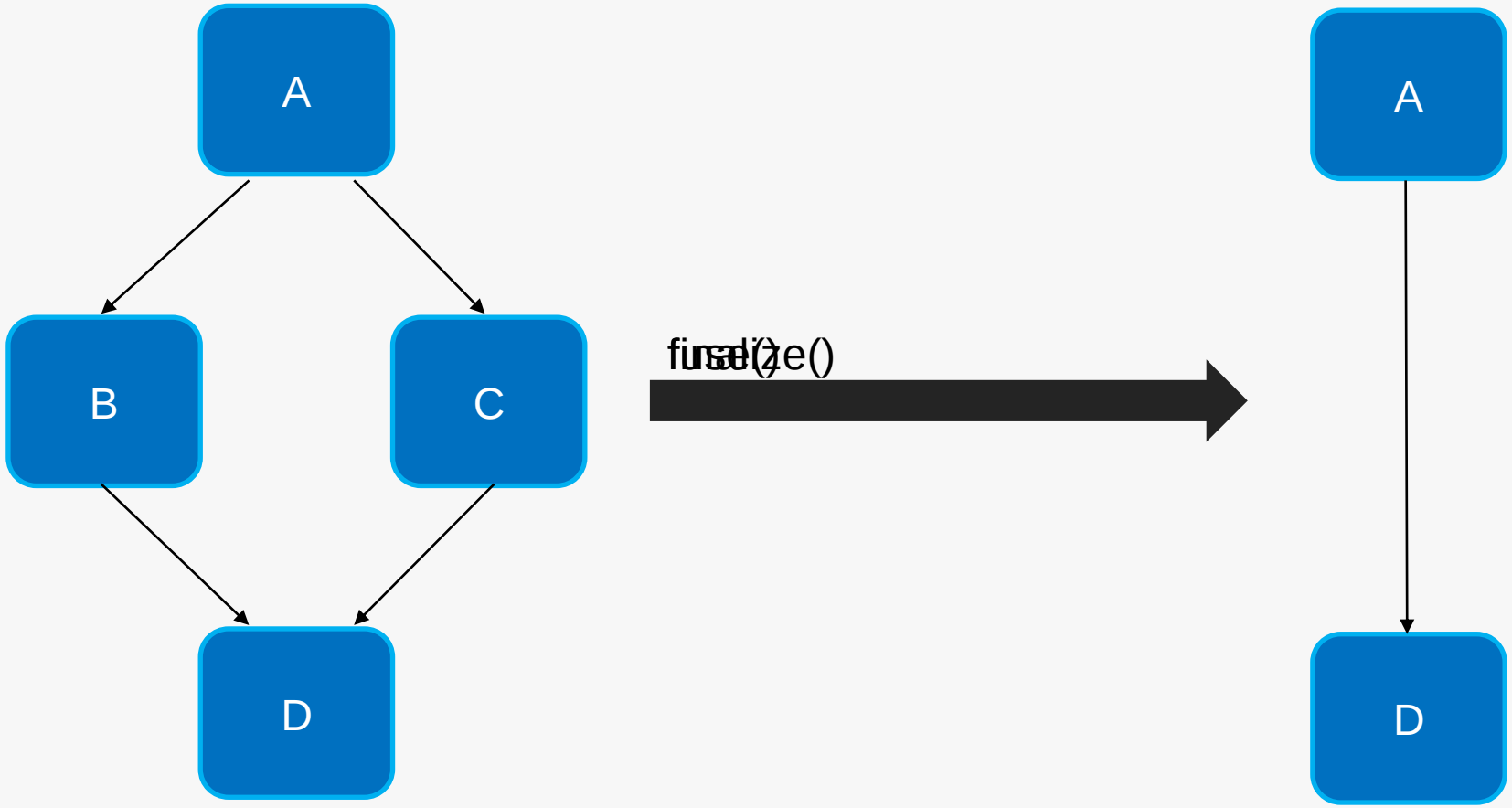
/codeplaysoft

codeplay.com

Adapting the JIT Compiler to DPC++



Application of Kernel Fusion to SYCL Graphs



Application of Kernel Fusion to SYCL Graphs