The 11th International workshop on OpenCL and SYCL

IWOCL & SYCLcon 2023

A SYCL Extension for User-Driven Online Kernel Fusion

Víctor Pérez, Codeplay

Lukas Sommer, Victor Lomüller, Kumudha Narasimhan, and Mehdi Goli

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.

Supported Solutions

oneAPI™

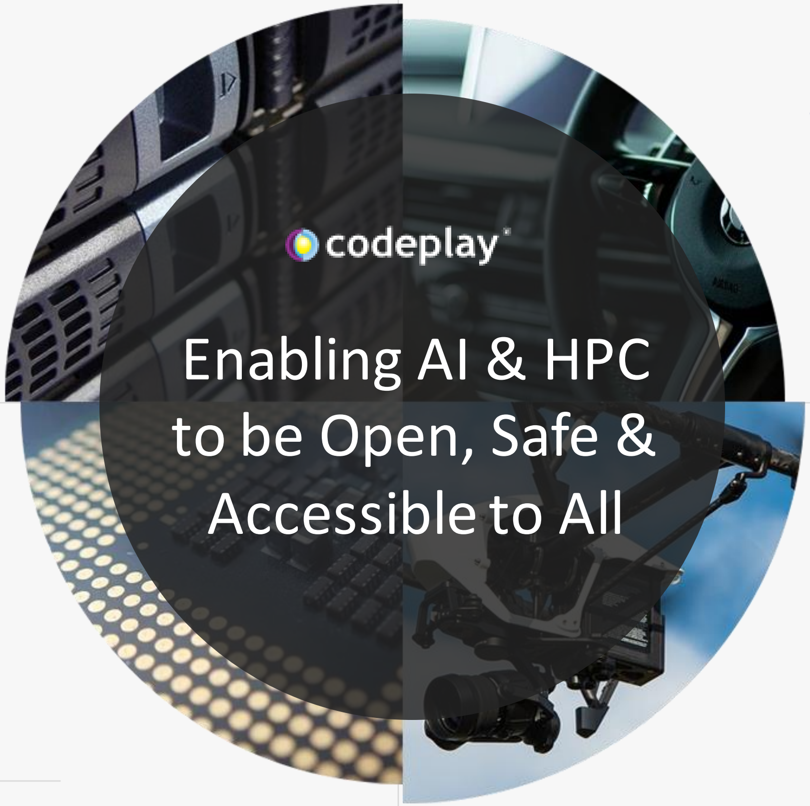
An open, cross-industry, SYCL based, unified, multiarchitecture, multi-vendor programming model that delivers a common developer experience across accelerator architectures

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™



codeplay®

Enabling AI & HPC to be Open, Safe & Accessible to All

Collaborations

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

And many more!

Markets

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

Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

Disclaimer

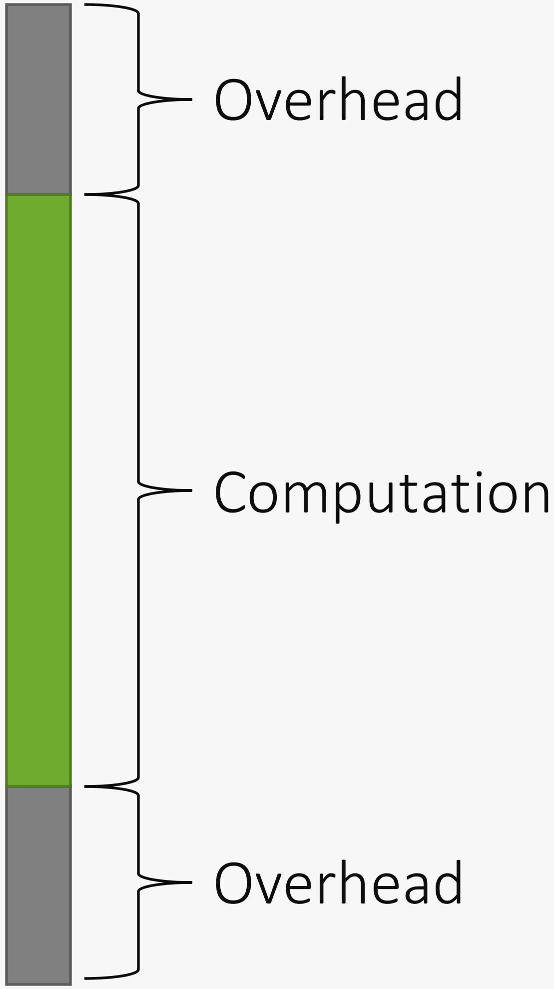
Agenda

- Motivation
- SYCL extension design
- Implementation
- Evaluation
- Conclusion

Motivation

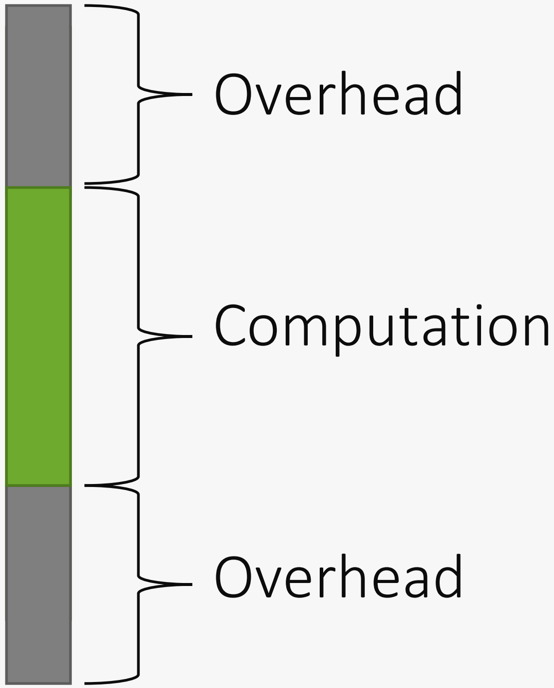
Motivation

- Every SYCL kernel launch carries cost



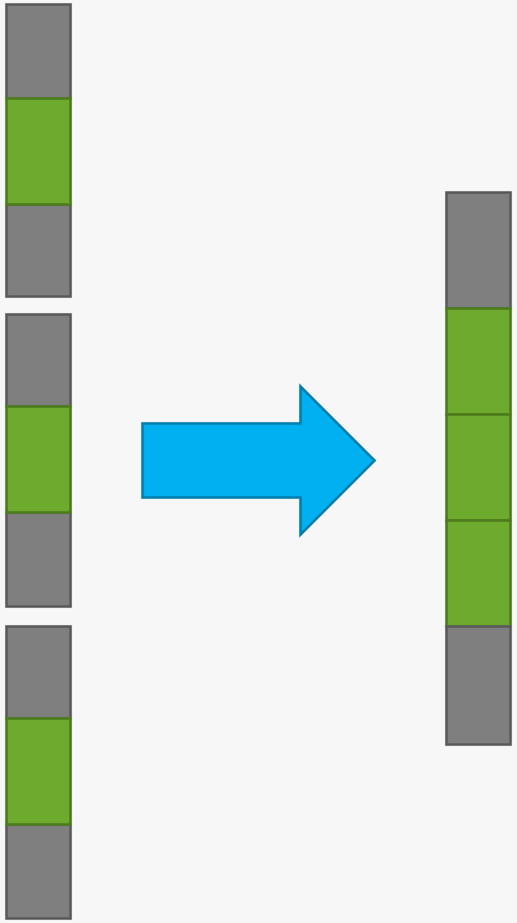
Motivation

- Every SYCL kernel launch carries cost
- Short running kernels problematic
    - High ratio of overhead to actual computation
- Can occur in different domains
    - Example: Machine learning
        - Each neural network operator maps to one kernel
        - Many operators in one network



Motivation

- Applications often use multiple kernels
  - Leads to sequence of short-running kernels
- Fusing multiple kernels into one yields better overhead/computation ratio
- Manual fusion possible but...
  - Time-consuming for developers
  - Error-prone
  - Not possible for kernels coming from libraries



SYCL Extension

SYCL Extension

- Allow the user to instruct SYCL runtime to fuse multiple kernels
- Easy-to-use API
- User in the driver seat
  - Correctness and profitability hard to assess with compiler methodology
  - User triggers fusion and takes responsibility
- SYCL runtime automates creation of fused kernel
  - User does not need to manually implement fused version of the kernel

Extension API

- Create queue with fusion property

```
queue q{gpu_selector_v,
      property::queue::enable_fusion{}};
```

- Start fusion mode

```
q.start_fusion();
```

- Submit kernels
  - Not executed on the device right away
  - Stored in a list of kernels to fuse

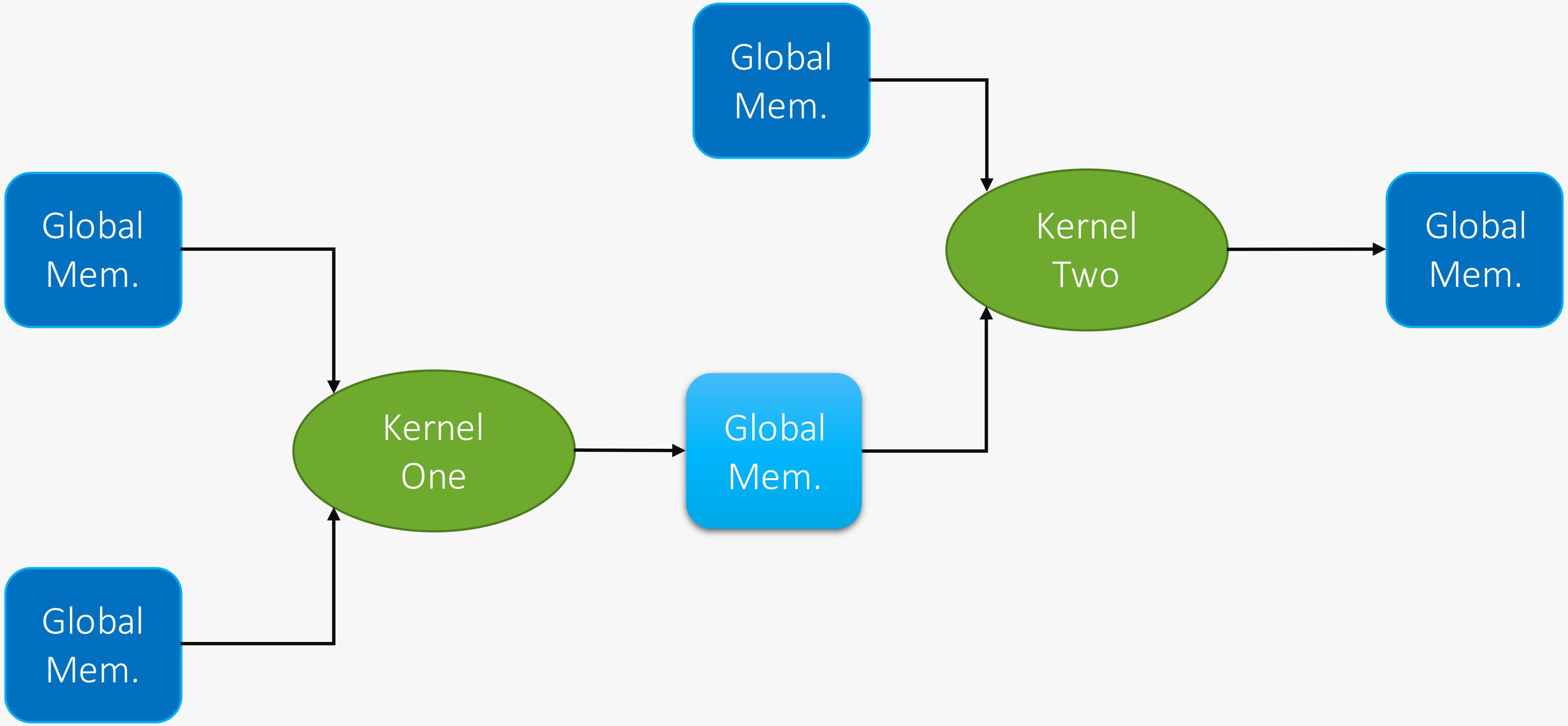
```
q.submit(…);

...

q.submit();
```

- Complete fusion
  - Leaves fusion mode
  - Creates fused kernel and schedules for execution

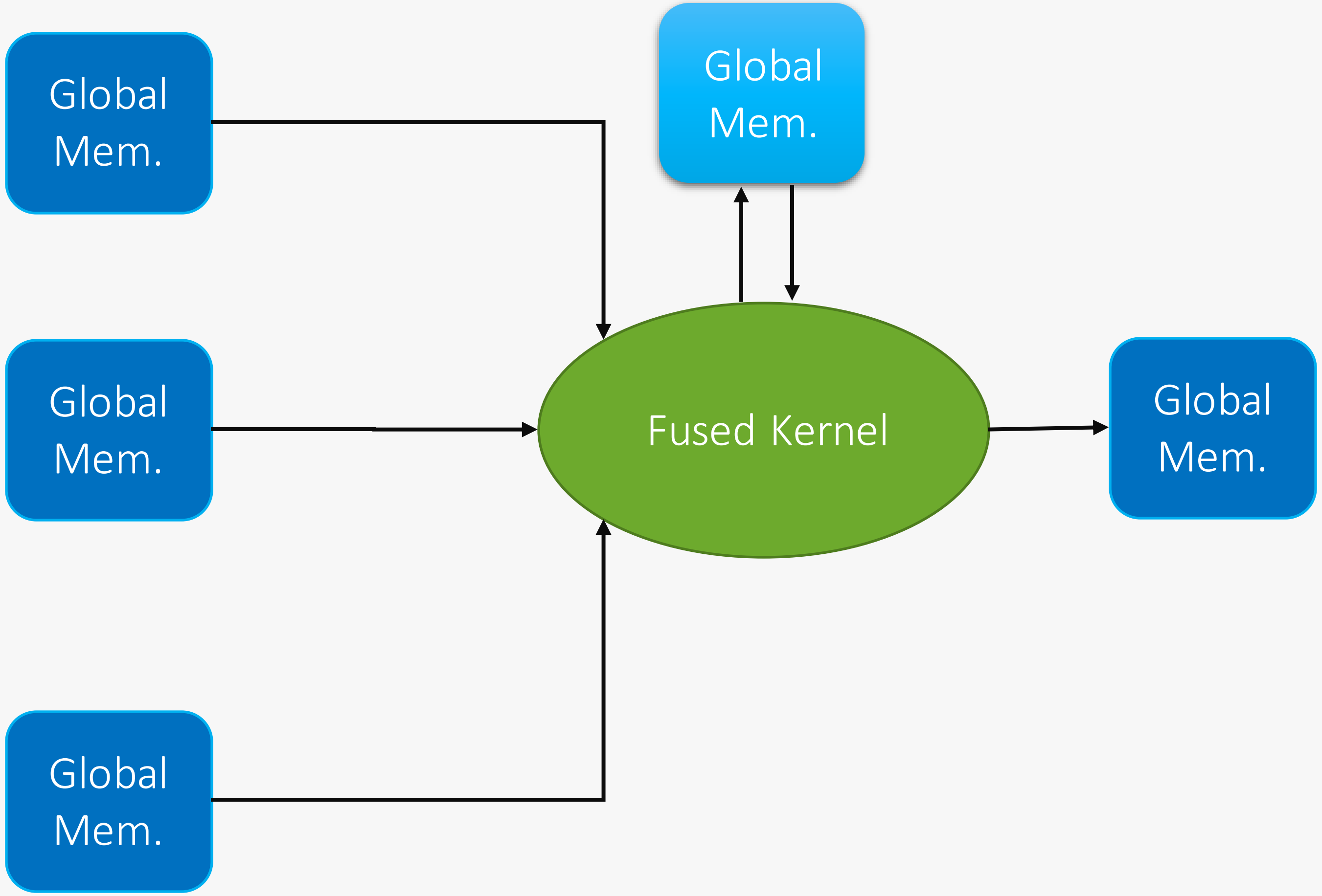
```
q.complete_fusion();
```

Dataflow Internalization

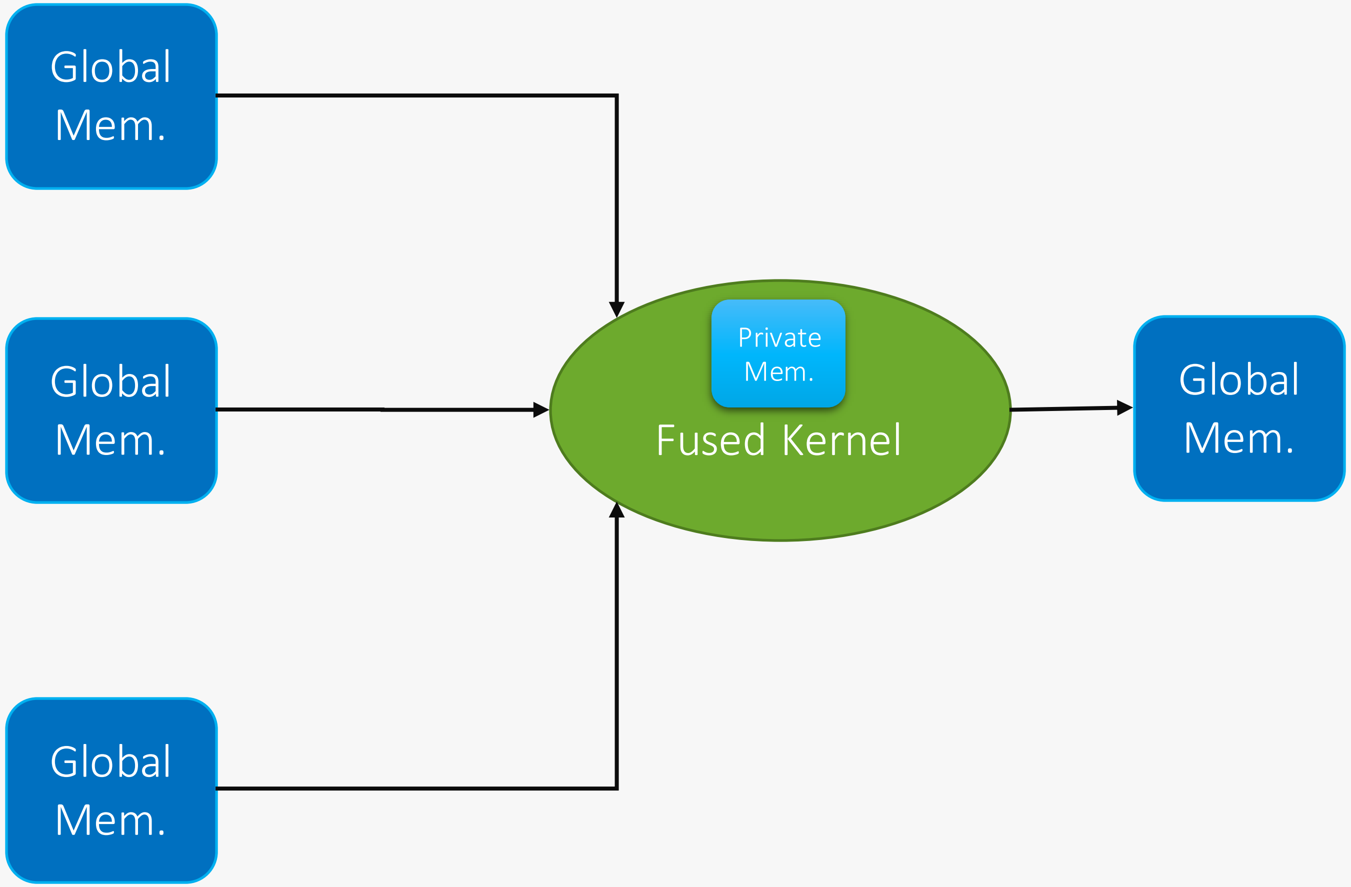
Unfused Kernels Dataflow



Fused Kernels Dataflow



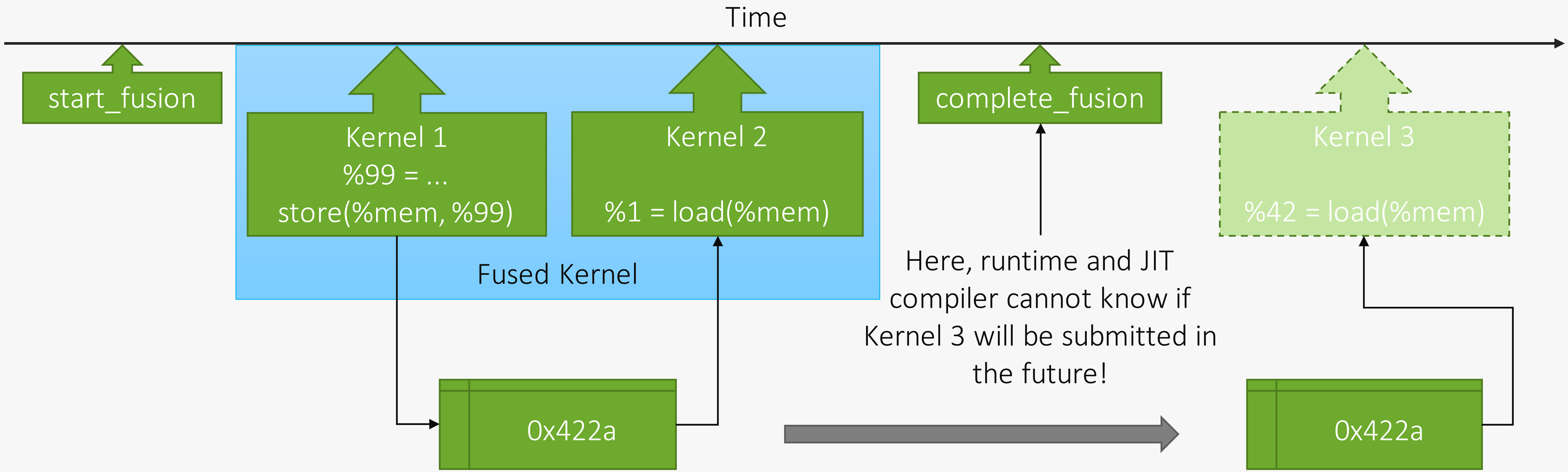
Internalized Dataflow



Internalization Extension API

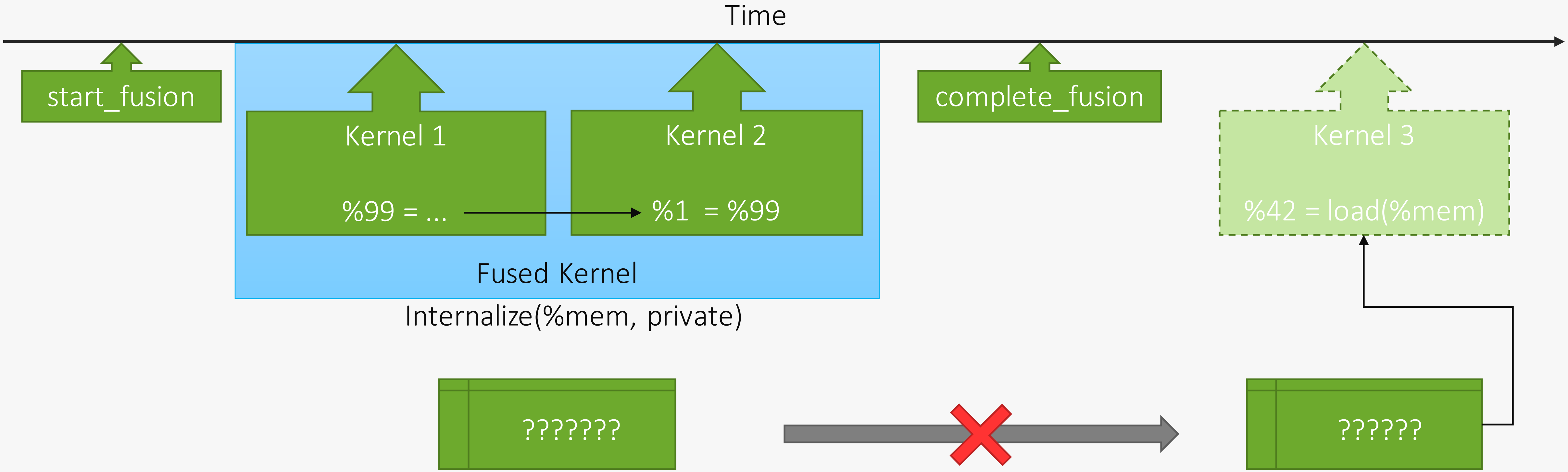
- Two properties:
  - `property::promote_local`
    - Promote to local memory
    - Requires *work-group* exclusive access
  - `property::promote_private`
    - Promotes to private memory (registers)
    - Requires *work-item* exclusive access
- Property can be applied to buffers or accessors

Why Properties for Internalization?



Here, runtime and JIT compiler cannot know if Kernel 3 will be submitted in the future!

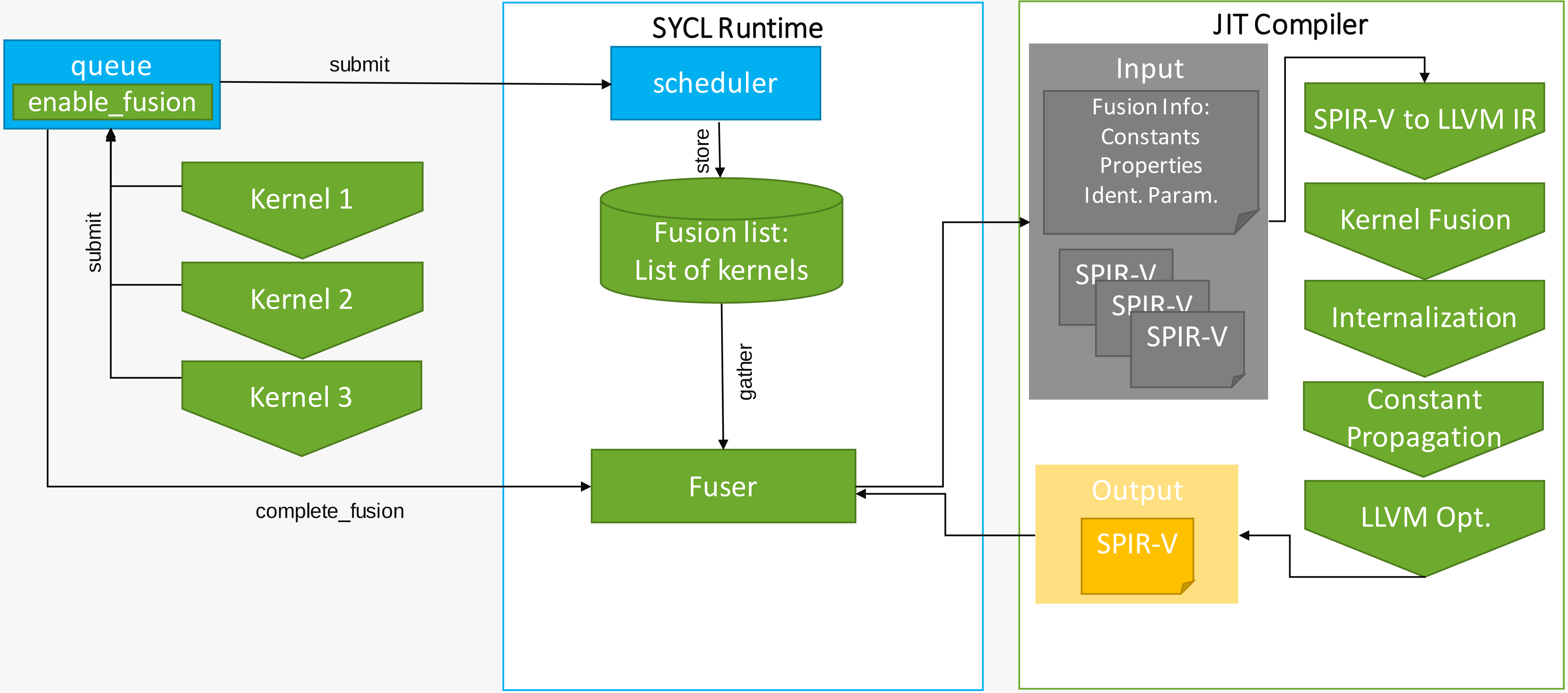
Why Properties for Internalization?



- Property specifies two things
  - JIT compiler should try to replace store & load with register/dataflow
  - **No future kernel will need access to results written by Kernel 1**
    - Kernel 3 can still use the same buffer, but cannot access results of Kernel 1

Implementation

Implementation overview



Fusion Steps – Defining fused kernel

```
void input_kernel1(accessor%1, scalar%2, accessor%3){...}


void input_kernel2(accessor%4, scalar%5, accessor%6){...}


void fused_kernel(accessor%1, scalar%2, accessor%3, accessor%4, scalar%5,
                      accessor%6){
  call input_kernel1(accessor%1, scalar%2, %accessor%3);
  work_group_barrier()
  call input_kernel2(accessor%4, scalar%5, accessor%6);
}
```

Fusion Steps – Omitting Barriers

```
void input_kernel1(accessor%1, scalar%2, accessor%3){...}


void input_kernel2(accessor%4, scalar%5, accessor%6){...}


void fused_kernel(accessor%1, scalar%2, accessor%3, accessor%4, scalar%5, accessor%6)
!2 {
  call input_kernel1(accessor%1, scalar%2, %accessor%3);
  work_group_barrier()
  call input_kernel2(accessor%4, scalar%5, accessor%6);
}


LLVM Metadata:
!2 property_no_barrier ← Specified by the user through no_barriers property
...
```

Fusion Steps – Omitting Barriers

```
void input_kernel1(accessor%1, scalar%2, accessor%3){...}


void input_kernel2(accessor%4, scalar%5, accessor%6){...}


void fused_kernel(accessor%1, scalar%2, accessor%3, accessor%4, scalar%5,
accessor%6){
  call input_kernel1(accessor%1, scalar%2, %accessor%3);
  call input_kernel2(accessor%4, scalar%5, accessor%6);
}


LLVM Metadata:
!2 property_no_barrier
...
```

Fusion Steps – Removing Identical Arguments

```
void input_kernel1(accessor%1, scalar%2, accessor%3){...}


void input_kernel2(accessor%4, scalar%5, accessor%6){...}


void fused_kernel(accessor%1, scalar%2, accessor%3, ~~accessor%4~~, scalar%5,
accessor%6) !1 {
  call input_kernel1(accessor%1, scalar%2, %accessor%3);
  call input_kernel2(accessor%4, scalar%5, accessor%6);
}
```

```
LLVM Metadata:
!1 identical(%accessor3, %accessor4)
...
```

← Can be determined by SYCL runtime

Fusion Steps – Removing Identical Arguments

```
void input_kernel1(accessor%1, scalar%2, accessor%3){...}

void input_kernel2(accessor%4, scalar%5, accessor%6){...}

void fused_kernel(accessor%1, scalar%2, accessor%3, scalar%5, accessor%6) !1 {
  call input_kernel1(accessor%1, scalar%2, %accessor%3);
  call input_kernel2(accessor%3, scalar%5, accessor%6);
}

LLVM Metadata:
!1 identical(%accessor3, %accessor4)
...
```

Fusion Steps – Constant Propagation

```
void input_kernel1(accessor%1, scalar%2, accessor%3){...}

void input_kernel2(accessor%4, scalar%5, accessor%6){...}

void fused_kernel(accessor%1, scalar%2, accessor%3, scalar%5, accessor%6) !3 {
  call input_kernel1(accessor%1, scalar%2, %accessor%3);
  call input_kernel2(accessor%3, scalar%5, accessor%6);
}

LLVM Metadata:
!3 list((scalar%2, value=5), (scalar%5, value=17))
...
```

← Can be determined by SYCL runtime

Fusion Steps – Constant Propagation

```
void input_kernel1(accessor%1, scalar%2, accessor%3){...}

void input_kernel2(accessor%4, scalar%5, accessor%6){...}

void fused_kernel(accessor%1, accessor%3, accessor%6) {
  call input_kernel1(accessor%1, 5, %accessor%3);
  call input_kernel2(accessor%3, 17, accessor%6);
}
```

Fusion Steps – Inline Fused Kernel Functions

```
void input_kernel1(accessor%1, scalar%2, accessor%3){ <body_kernel1> }


void input_kernel2(accessor%4, scalar%5, accessor%6){ <body_kernel2> }


void fused_kernel(accessor%1, accessor%3, accessor%6) {
  <body_kernel1>
  <body_kernel2>
}
```

Fusion Steps – Internalization

```
void fused_kernel(accessor%1, ~~accessor%3,~~ accessor%6) !4 {
  ...
  <kernel1>
    value%99 = mul ...
    ~~store value%99, accessor%3~~
  </kernel1>
  <kernel2>
    ~~value%1337 = load %accessor3~~
    value%1338 = add %value1337, ...
  </kernel2>
}
```

```
LLVM Metadata:
!4 internalize(accessor%3, private)
```

← Specified by the user through SYCL property

Fusion Steps – Internalization

```
void fused_kernel(accessor%1, accessor%6) !4 {
  ...
  <kernel1>
    value%99 = mul ...
  </kernel1>
  <kernel2>
    value%1338 = add %value99, ...
  </kernel2>
}
LLVM Metadata:
!4 internalize(accessor%3, private)
```

Evaluation

Evaluation Setup

Device type	Model	OpenCL driver Version	OS	SYCL Compiler Version
CPU	Intel i7-6700K	2022.13.3.0.16_160000	Ubuntu 18.04.6 Kernel 4.15.0	ComputeCpp-PE 2.10.0
GPU	Intel Gen9 HD Graphics NEO	21.38.21026		

Case Study – SYCL-DNN

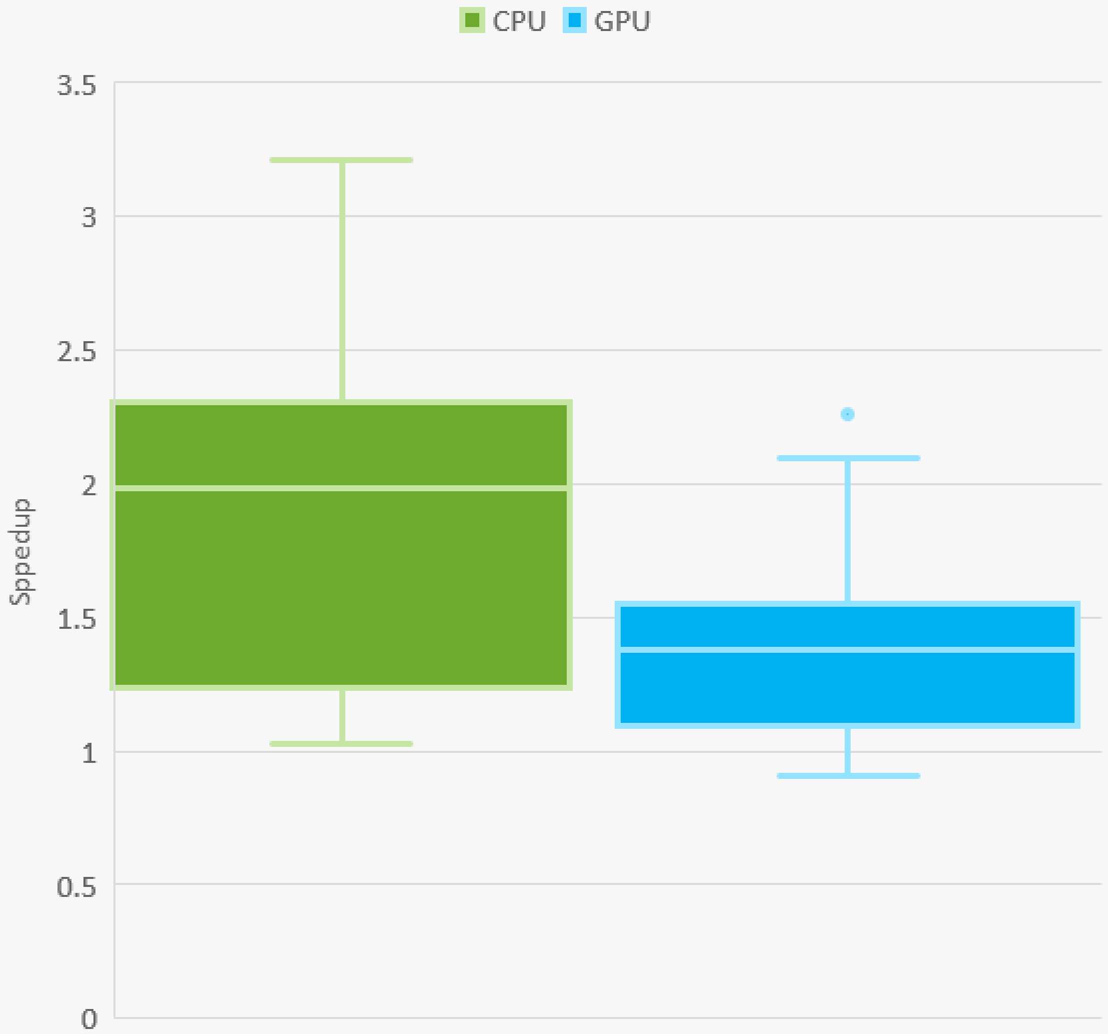
SYCL-DNN

- Open-source SYCL library for neural network operators
- Two types of workloads
  - Microbenchmarks
    - Sequences taken from actual neural network models (e.g. GPT-2)
    - Fusion of arithmetic-heavy kernels
    - Example: BatchNormalization + ReLu
  - End-to-end neural networks
    - VGG16 & ResNet-50
    - Alternative version using a more "fusion-friendly" convolution algorithm

Microbenchmarks Results - Distribution

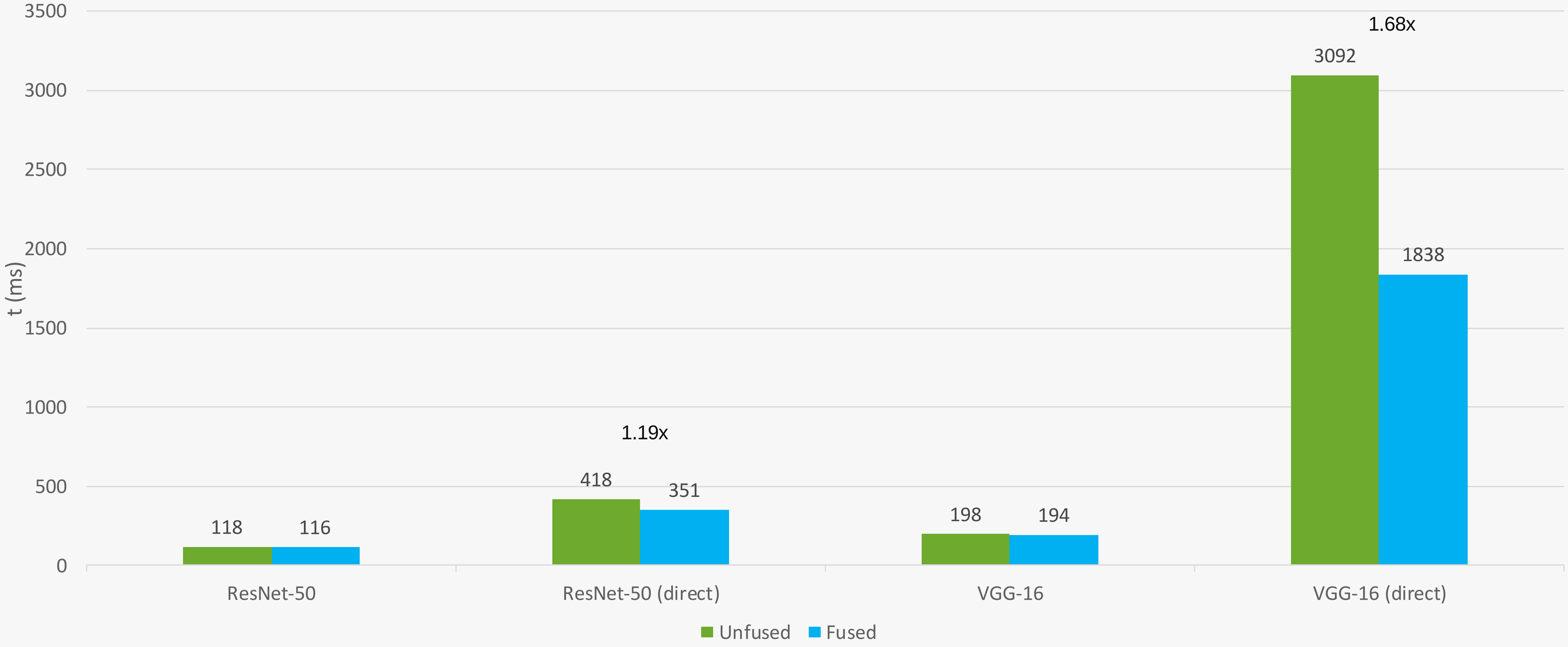
- CPU:
  - 100 %: >1x
  - 75 %: >1.25x
  - 50 %: >1.78x
  - Max: 3.21x

- GPU:
  - 50 %: >1.35x
  - 25 %: >1.5x
  - Max: 2.36x



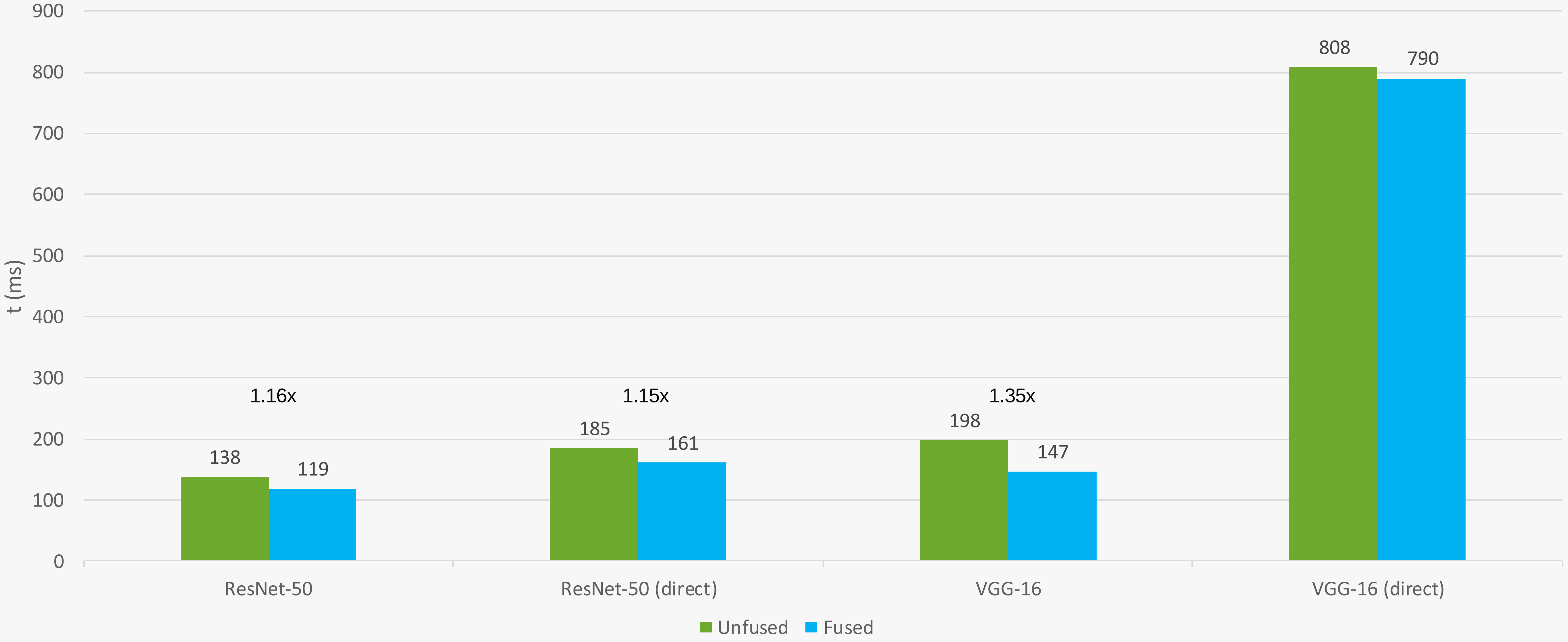
See backup for workloads and configurations. Results may vary. See TACO paper for more information: https://tinyurl.com/taco-paper

Full NN Results - CPU



See backup for workloads and configurations. Results may vary. See TACO paper for more information: https://tinyurl.com/taco-paper

Full NN Results - GPU



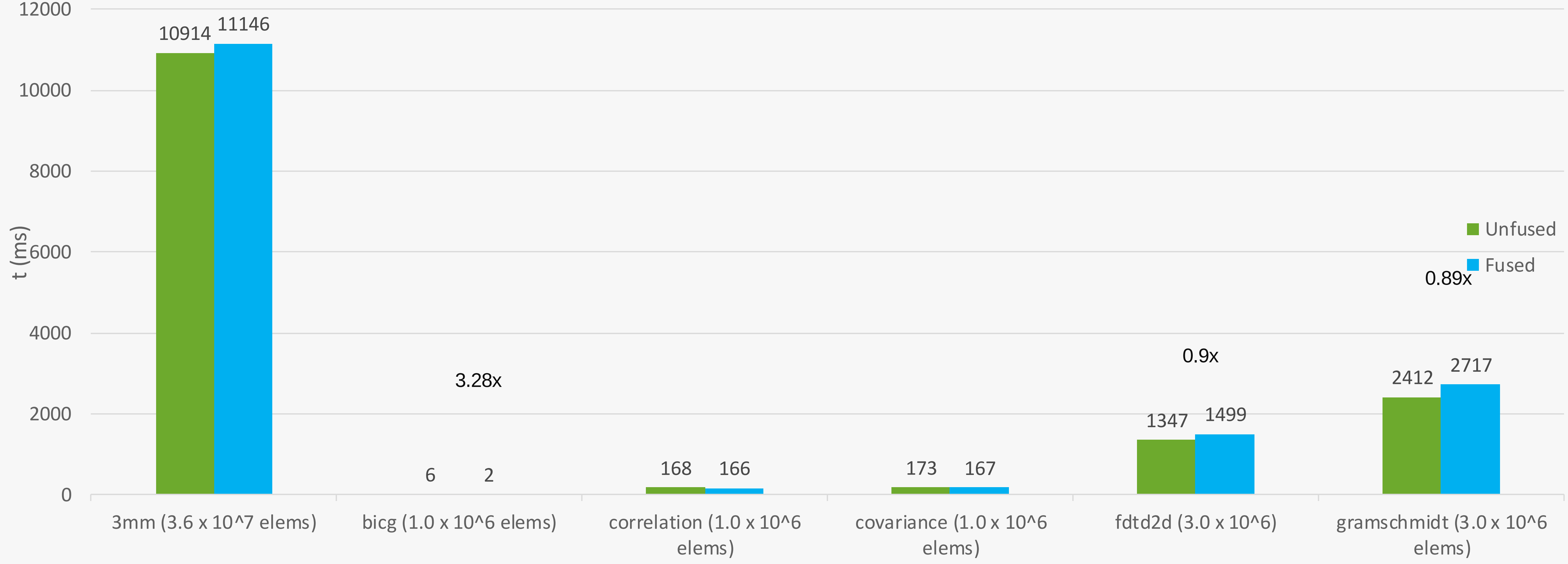
See backup for workloads and configurations. Results may vary. See TACO paper for more information: https://tinyurl.com/taco-paper

Case Study – SYCL-Bench

SYCL-Bench

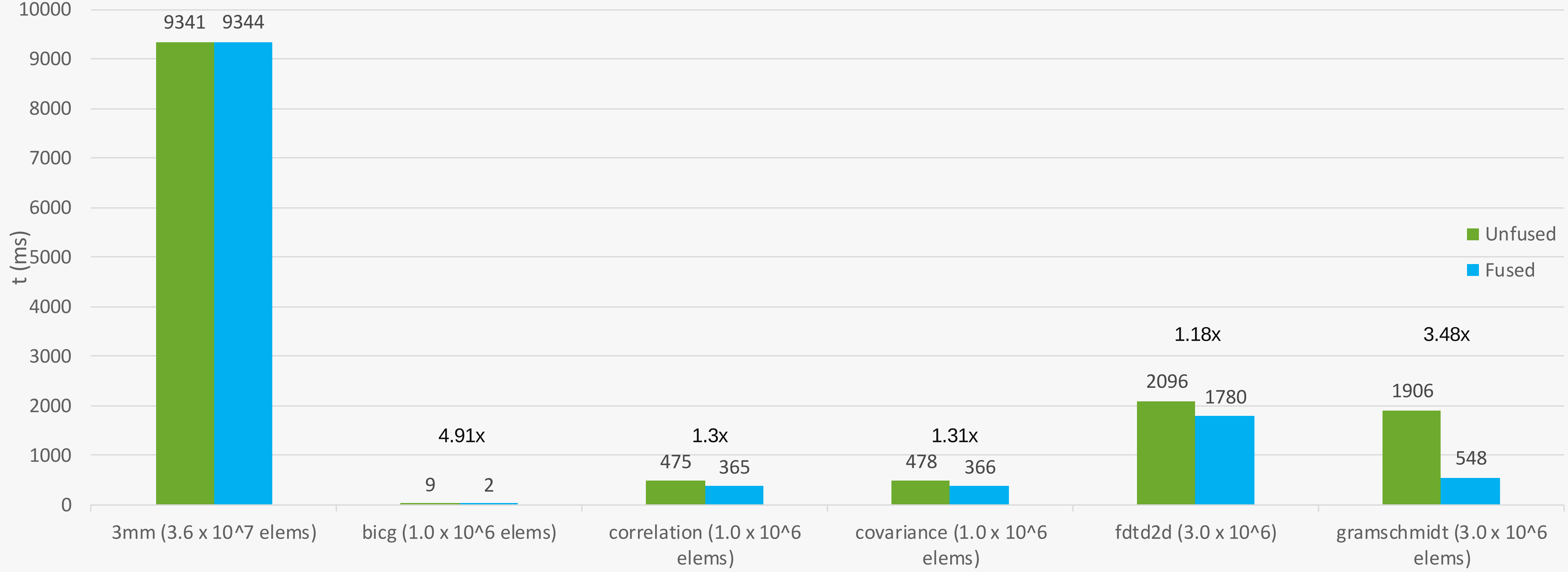
- SYCL benchmark-suite
- Focus on `polybench` benchmarks
  - arithmetic-heavy workloads
- Kernel fusion broadly applicable
  - 6 out of 9 polybench benchmarks
  - Internalization applicable to 2 out of 6
- https://github.com/bcosenza/sycl-bench

SYCL-Bench Results - CPU



See backup for workloads and configurations. Results may vary. See TACO paper for more information: https://tinyurl.com/taco-paper

SYCL-Bench Results - GPU



See backup for workloads and configurations. Results may vary. See TACO paper for more information: https://tinyurl.com/taco-paper

Extension Status Update

Current Extension

- Available in DPC++ with some modifications
- Only supporting SPIR-V targets for now
  - Working on PTX/HIP targets support
- Extension proposal: https://tinyurl.com/dpcpp-extension

Kernel Fusion + SYCL Graphs

- Use SYCL graphs to fuse several kernels at runtime
- WIP: *Graph Fusion* extension built on top of SYCL graphs
- Graph Fusion extension proposal:
  - https://tinyurl.com/graph-fusion
- SYCL graphs extension proposal:
  - https://tinyurl.com/sycl-graphs

Graphs Fusion Advantages

- Two different versatile APIs to construct fusion sequence: recording or explicit
- Fine-grained control over execution of JIT compilation through `command_graph::finalize()`
- Unify two similar APIs
- Reusability of fused kernels

Graph Fusion API

Kernel fusion

```
queue q{gpu_selector_v,
      property::queue::enable_fusion{}};
```

```
q.start_fusion();
```

```
q.submit(…);

...

q.submit();
```

```
q.complete_fusion();
```

Graph fusion extension proposal:
https://tinyurl.com/graph-fusion

Graph fusion

```
queue q{gpu_selector_v};
command_graph graph;
```

```
graph.add(…);

...

graph.add();
```

```
auto exec_graph = graph.finalize(q.get_context(),
        {property::perform_fusion});
q.ext_oneapi_graph(exec_graph);
```

codeplay

Conclusion

Conclusion

- Kernel fusion can improve SYCL performance significantly
  - Example: >3x for neural network operator sequence
- Extension allows user to instruct SYCL runtime to fuse
  - No need to manually write fused kernel
- Upstreamed to DPC++
  - Working on support for more targets (currently only SPIR-V based)
  - Align with SYCL graphs extension
- Links
  - TACO journal paper: https://tinyurl.com/taco-paper
  - DPC++ extension proposal: https://tinyurl.com/dpcpp-extension
  - SYCL graphs extension proposal: https://tinyurl.com/command-graphs
  - Graph fusion extension proposal: https://tinyurl.com/graph-fusion
  - Kernel fusion tutorial: https://tinyurl.com/kernel-fusion-blogpost

See backup for workloads and configurations. Results may vary. See TACO paper for more information: https://tinyurl.com/taco-paper

codeplay®

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

Ask me anything (about kernel fusion)!

@codeplaysoft

info@codeplay.com

codeplay.com

Workloads and Configuration

- System configuration
  - ComputeCpp Professional Edition (PE) 2.10.0
  - OS: Ubuntu 18.04.6 LTS, Kernel 4.15.0
  - CPU: Intel Core i7-6700K, OpenCL driver 2022.13.3.0.16_160000
  - GPU: Intel Gen9 HD Graphics NEO, OpenCL driver 21.38.21026
- Workloads
  - For workloads that measure time, measured time is time spent on the device and time used in the data transfer between host and device
  - Workloads selected include domains such as HPC and ML/DL
  - Each measurement is run 10 times, first measurement is discarded and then average is taken
- Results
  - In most cases, better performance was attributed to reduction in overhead through fusion and less reads and writes to global memory through internalization as part of fusion

Backup

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{}});
}
```

Fusion Steps

Adding Metadata

```
void input_kernel1(accessor%1, scalar%2, accessor%3){
<body kernel1>
}

void input_kernel2(accessor%4, scalar%5, accessor%6){
<body kernel2>
}

void fused_kernel() !0 !...

LLVM Metadata:
!0 list(input_kernel1, input_kernel2)
...
```