

# Celerity: How (Well) Does the SYCL API Translate to Distributed Clusters?

Philip Salzmann<sup>1</sup>, Fabian Knorr<sup>1</sup>, Peter Thoman<sup>1</sup>, Biagio Cosenza<sup>2</sup>

<sup>1</sup>University of Innsbruck, Austria - [first.last@uibk.ac.at](mailto:first.last@uibk.ac.at)

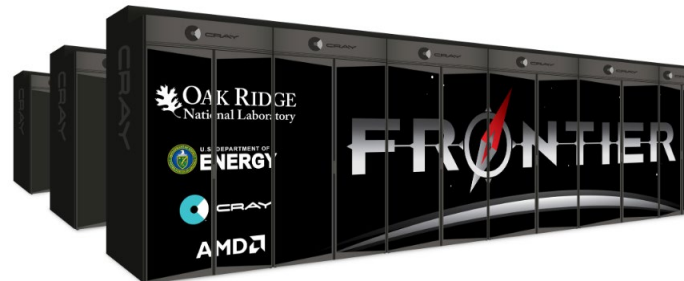
<sup>2</sup>University of Salerno, Italy – [bcosenza@unisa.it](mailto:bcosenza@unisa.it)

IWOCL & SYCLcon 2022



# SYCL in HPC

- SYCL is gaining traction in HPC
- Several **upcoming (pre-)exascale machines** look to support SYCL
- Well-known HPC applications such as **GROMACS** are adding SYCL support



# SYCL in HPC

- SYCL has potential to become dominant in this space as it strikes a good balance between **abstraction** and **expressiveness**
  - High-level enough to alleviate many of the common burdens
  - Allows to take **low-level control** where it is needed
    - Inside kernels
    - Optional APIs for explicit data movement, dependency management etc.
  - **Interop APIs** allow to interface with vendor libraries and to gradually convert legacy codes
- And of course: It is **vendor neutral!**

# MPI + X

- The traditional approach
- Will probably remain relevant for foreseeable future
- Low level: Requires **manual** handling of **work and data partitioning**
- Typical approach: Blocking send/receive at clearly defined points in time, **implicit synchronization** across nodes
- Advanced approaches: Using non-blocking operations or one-sided communication for **computation/communication overlap**
  - More difficult to implement
  - Hard to change afterwards; hampers flexibility in algorithmic experimentation

# MPI + SYCL

- It works - SYCL can be combined with MPI in same ways as MPI + CUDA or MPI + OpenCL
- However: SYCL operates on higher level of abstraction than CUDA/OpenCL
- We believe that SYCL's high-level, declarative dataflow APIs **can and should** be extended to distributed memory clusters...

# The Celerity Programming Model

- Goal: Extend SYCL to distributed clusters
- It is not a SYCL implementation
  - Abstraction layer on top of **MPI + SYCL**
  - Forwards kernel code to an underlying SYCL implementation
- Tries to stay as **close to SYCL API** as possible
  - Code should look very familiar
  - Neither a true subset nor superset of the SYCL API
- Currently being validated in two industry use cases on **Marconi-100** supercomputer at CINECA, Italy as part of the **LIGATE** project
  - Drug-discovery pipeline
  - ToF room response simulation



# SYCL Core Concepts: Queues

```
sycl::queue my_queue(sycl::gpu_selector_v);  
my_queue.submit(/*...*/);
```

- It's not a queue!
  - (Unless `property::queue::in_order` is provided)
- Builds a task graph
  - Either **implicitly** (accessors),
  - Or **explicitly** (events, `handler::depends_on`)
  - Enables **scheduling freedom** for SYCL runtime
- Associated with a single device
  - Can have multiple queues in a program

# MPI + SYCL: Queues

- Typically, multiple devices (e.g., 4) on a single node
  - Need to manually manage device selection on a single node
- Can either use a single device per rank, multiple ranks per node
- ...or multiple devices per rank, single rank per node
  - Potentially faster
  - Additional layer of complexity regarding work and data distribution
- Few opportunities to leverage out-of-order semantics in basic MPI + SYCL applications
  - Due to implicit synchronization on communication



# Celerity: Queues

- In Celerity there is only one *distributed* queue
  - (Also not a queue!)
  - Manages device<->rank assignments automatically
- Works mostly the same way as in SYCL
  - Builds implicit task graph (but **with finer granularity**)
  - Allows for **task splitting** across cluster nodes
- SPMD model: All ranks submit the same set of tasks (command groups)

# SYCL Core Concepts: Buffers

```
sycl::buffer<double, 2> my_buf({512, 512});
```

- High level of abstraction
  - Multi-dimensional data access
  - Do not correspond to any single allocation
  - Transparently migrated between host and one or more devices as needed
- Safe
  - Ref-counted
  - Destructor will block until all operations on buffer have completed

# MPI + SYCL: Buffers

Manual work partitioning requires manual data partitioning.  
Some options:

- Buffer on each rank contains partial data
  - Standard MPI approach
  - Manual bookkeeping required: Which part of problem domain exists where at what point in time?
- Use single global buffer containing data for all ranks
  - Pro: Can use global indexing (with offsets) inside kernels
  - If not all data is needed everywhere, it's wasteful at best, infeasible at worst

# Celerity: Buffers

- Fully virtualized!
  - All ranks can use the same buffer
  - Only required parts are allocated on each rank
- Consequence: We need to somehow know which parts of virtualized buffer are required *where*

# SYCL Core Concepts: Accessors

```
sycl::accessor my_acc(my_buf, cgh, sycl::write_only);
```

- Core construct for declarative data access
- Communicate **ahead of time** how a buffer will be accessed (for reading, writing, or both)
- Available both on device and host
- Fine-grained control through ranged accessors (optimization opportunity)

# MPI + SYCL: Accessors

```
if(rank == 0) {  
    sycl::host_accessor my_acc(my_buf, sycl::read_only);  
    MPI_Send(my_acc.get_pointer(), my_acc.size(), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);  
} else {  
    sycl::host_accessor my_acc(my_buf, sycl::write_only, {sycl::no_init});  
    MPI_Recv(my_acc.get_pointer(), my_acc.size(), MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
}
```

- Use host accessor to get data from/to device before/after MPI transfer
  - However, if there is a direct GPU<->GPU interconnect (e.g., PCIe bus or GPUDirect RDMA) this incurs **unnecessary transfers**
- Host accessors are **implicit synchronization points**

# Celerity: Accessors

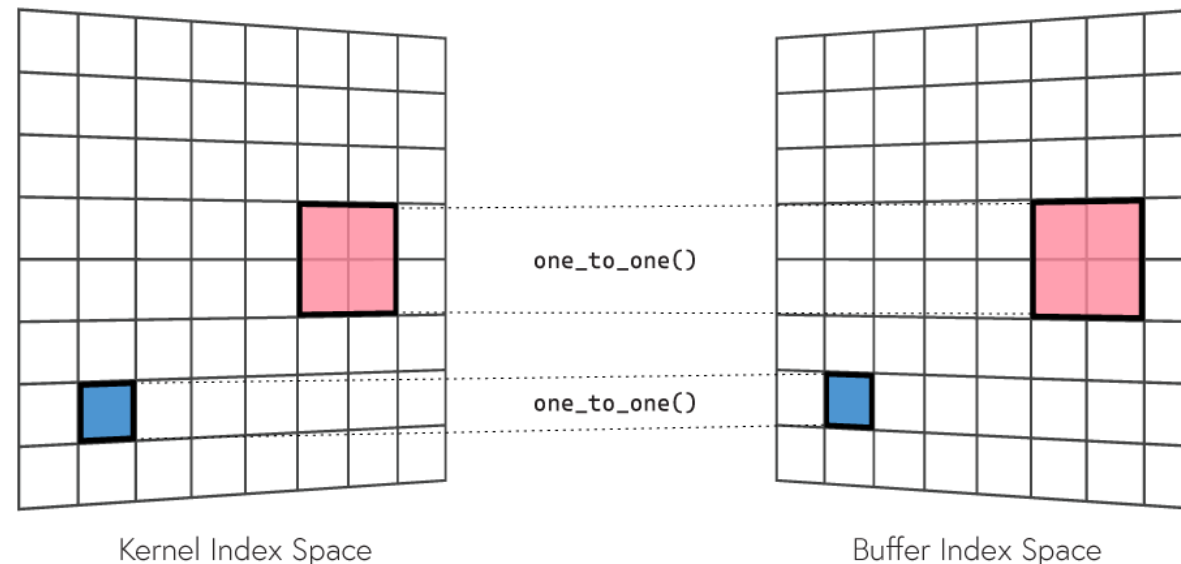
```
celerity::accessor my_acc{my_buf, cgh, celerity::access::slice<2>(1), celerity::read_only};
```

- Task splitting: Additionally specify *where* a buffer is being accessed
  - ***Range mappers*** are used to build fine-grained task graph

# Celerity: Accessors

```
celerity::accessor my_acc{my_buf, cgh, celerity::access::slice<2>(1), celerity::read_only};
```

- Task splitting: Additionally specify *where* a buffer is being accessed
  - **Range mappers** are used to build fine-grained task graph

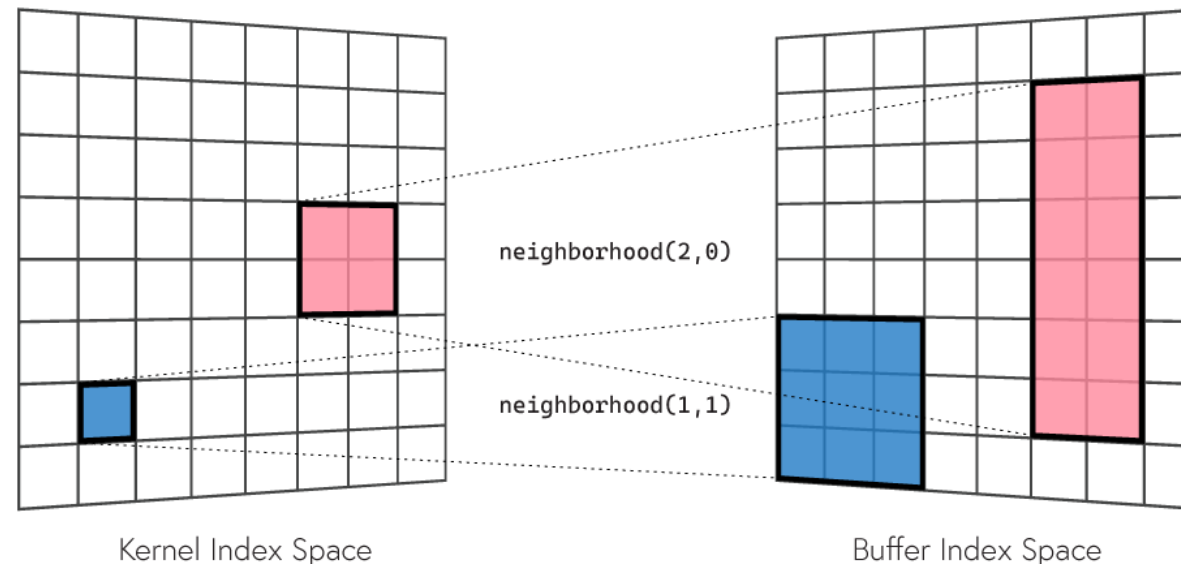




# Celerity: Accessors

```
celerity::accessor my_acc{my_buf, cgh, celerity::access::slice<2>(1), celerity::read_only};
```

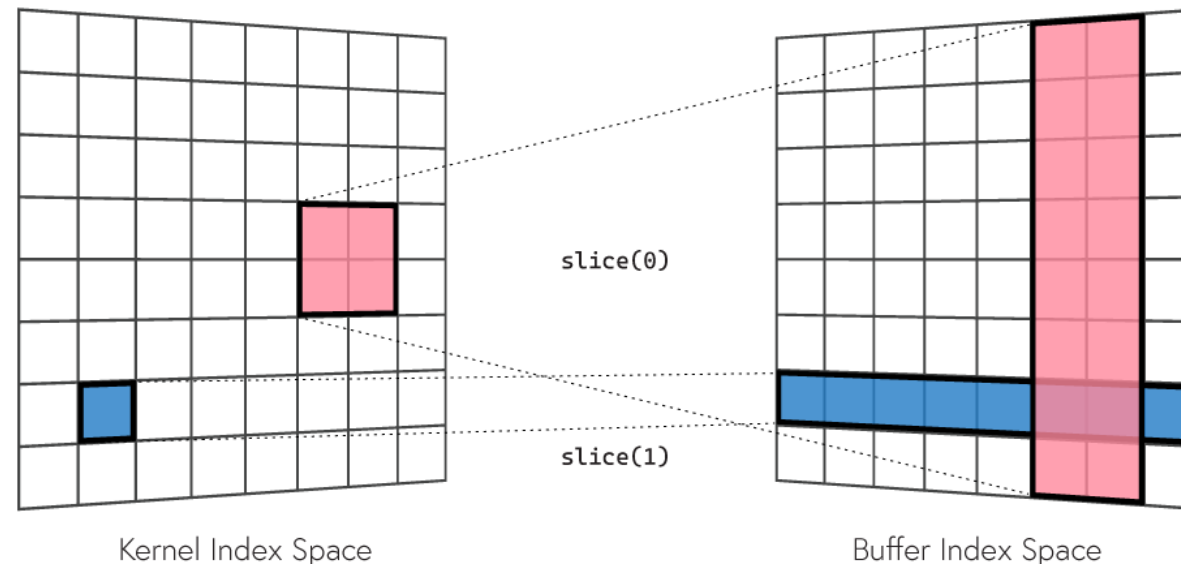
- Task splitting: Additionally specify *where* a buffer is being accessed
  - **Range mappers** are used to build fine-grained task graph



# Celerity: Accessors

```
celerity::accessor my_acc{my_buf, cgh, celerity::access::slice<2>(1), celerity::read_only};
```

- Task splitting: Additionally specify *where* a buffer is being accessed
  - **Range mappers** are used to build fine-grained task graph



# Celerity: Accessors

```
celerity::accessor my_acc{my_buf, cgh, celerity::access::slice<2>(1), celerity::read_only};
```

- Task splitting: Additionally specify *where* a buffer is being accessed
  - *Range mappers* are used to build fine-grained task graph
- Kernel code can in many cases be directly reused from SYCL
  - Pointer based access is a bit tricky due to virtualized buffers (different strides)

# Additional Features: USM

```
float* my_ptr = sycl::malloc_device<float>(1024, my_queue);
```

- SYCL
  - Allows for **low level control** over device memory
  - Enables **interop** with pointer-based APIs and legacy codes
  - Requires **manual dependency management** between kernels
- MPI + SYCL
  - Fully manual control over data movement
  - Allows to leverage **GPU-aware MPI**
- Celerity
  - **Impossible** to support (pure library implementation)

# Additional Features: Host Tasks

- SYCL
  - Allows to insert **host code** into **asynchronous execution flow**
  - Offers **interoperability** features to access native objects (e.g., CUDA, Level Zero, OpenCL) behind buffers, queues etc.
  - Requires care when interacting with objects from main thread
- MPI + SYCL
  - Enables **asynchronous communication** and **latency hiding** in combination with non-blocking routines
  - Interop presents way of leveraging **GPU-aware MPI with buffers**
- Celerity
  - Additionally supports **collective host tasks**, useful for bulk I/O and other collective operations

# Additional Features: Reductions

```
my_queue.submit([&](sycl::handler& cgh) {  
    sycl::accessor acc(my_buf, cgh, sycl::read_only);  
    auto sum_reducer = sycl::reduction(sum_buf, cgh, sycl::plus<>());  
    cgh.parallel_for(my_buf.get_range(), sum_reducer, [=](sycl::id<2> id, auto& sum) {  
        sum += acc[id];  
    });  
});
```

- SYCL
  - **Declarative API** similar to accessors
  - Currently only 0-dimensional (buffer) or 1-dimensional (span) reductions
- MPI + SYCL
  - May require additional **MPI reduction** for final result
- Celerity
  - Automatically takes care of inter-node reduction step

# MPI + SYCL / Celerity: Summary

- SYCL can be paired with MPI just like CUDA or OpenCL
  - Many different options with varying degrees of complexity and flexibility
    - Host accessors
    - Host tasks + interop
    - USM
    - ...
- SYCL **already has** information required to execute tasks **across multiple devices** or even **multiple nodes** in distributed cluster
  - *Which* buffers are being accessed, *when*, and *how* (reading/writing)
  - Requires one additional piece of information (*where*) to enable task splitting

# From SYCL to Celerity: Jacobi Stencil

```
sycl      ::      queue queue;

sycl      ::buffer<double, 2> in_buf({N, N});
sycl      ::buffer<double, 2> out_buf({N, N});

for(int i = 0; i < num_iterations; ++i) {
    queue.submit([&](sycl      ::handler& cgh) {

        sycl      ::accessor in{in_buf, cgh,      sycl      ::read_only};
        sycl      ::accessor out{out_buf, cgh,      sycl      ::read_write};

        cgh.parallel_for(out_buf.get_range(), [=](sycl      ::item<2> itm) {
            /* boundary handling omitted for brevity */
            const auto i = itm[0];
            const auto j = itm[1];
            out[itm] = (in[{i, j - 1}] + in[{i, j + 1}] + in[{i - 1, j}] + in[{i + 1, j}]) / 4.0;
        });
    });
    std::swap(in_buf, out_buf);
}
```



# From SYCL to Celerity: Jacobi Stencil

```
celerity::distr_queue queue;

celerity::buffer<double, 2> in_buf({N, N});
celerity::buffer<double, 2> out_buf({N, N});

for(int i = 0; i < num_iterations; ++i) {
    queue.submit(=[](celerity::handler& cgh) {
        auto nbr = celerity::access::neighborhood<2>{1, 1};
        auto o2o = celerity::access::one_to_one{};
        celerity::accessor in{in_buf, cgh, nbr, celerity::read_only};
        celerity::accessor out{out_buf, cgh, o2o, celerity::read_write};

        cgh.parallel_for(out_buf.get_range(), [=(celerity::item<2> itm) {
            /* boundary handling omitted for brevity */
            const auto i = itm[0];
            const auto j = itm[1];
            out[itm] = (in[{i, j - 1}] + in[{i, j + 1}] + in[{i - 1, j}] + in[{i + 1, j}]) / 4.0;
        }));
    });
    std::swap(in_buf, out_buf);
}
```

# From SYCL to Celerity: Jacobi Stencil

```

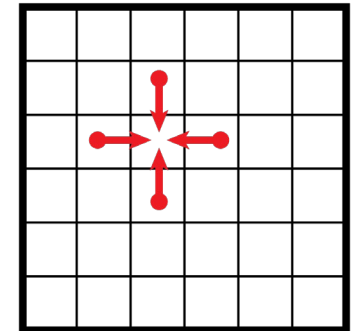
celerity::distr_queue queue;

celerity::buffer<double, 2> in_buf({N, N});
celerity::buffer<double, 2> out_buf({N, N});

for(int i = 0; i < num_iterations; ++i) {
    queue.submit([=](celerity::handler& cgh) {
        auto nbr = celerity::access::neighborhood<2>{1, 1};
        auto o2o = celerity::access::one_to_one{};
        celerity::accessor in{in_buf, cgh, nbr, celerity::read_only};
        celerity::accessor out{out_buf, cgh, o2o, celerity::read_write};

        cgh.parallel_for(out_buf.get_range(), [=](celerity::item<2> itm) {
            /* boundary handling omitted for brevity */
            const auto i = itm[0];
            const auto j = itm[1];
            out[itm] = (in[{i, j - 1}] + in[{i, j + 1}] + in[{i - 1, j}] + in[{i + 1, j}]) / 4.0;
        });
    });
    std::swap(in_buf, out_buf);
}
  
```

Read 4 neighboring elements  
along main axes



[\[Wikipedia\]](#)

# From SYCL to Celerity: Jacobi Stencil

```

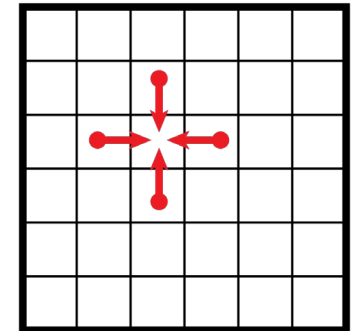
celerity::distr_queue queue;

celerity::buffer<double, 2> in_buf({N, N});
celerity::buffer<double, 2> out_buf({N, N});

for(int i = 0; i < num_iterations; ++i) {
    queue.submit(=](celerity::handler& cgh) {
        auto nbr = celerity::access::neighborhood<2>{1, 1};
        auto o2o = celerity::access::one_to_one{};
        celerity::accessor in{in_buf, cgh, nbr, celerity::read_only};
        celerity::accessor out{out_buf, cgh, o2o, celerity::read_write};

        cgh.parallel_for(out_buf.get_range(), [=](celerity::item<2> itm) {
            /* boundary handling omitted for brevity */
            const auto i = itm[0];
            const auto j = itm[1];
            out[itm] = (in[{i, j - 1}] + in[{i, j + 1}] + in[{i - 1, j}] + in[{i + 1, j}]) / 4.0;
        });
    });
    std::swap(in_buf, out_buf);
}
  
```

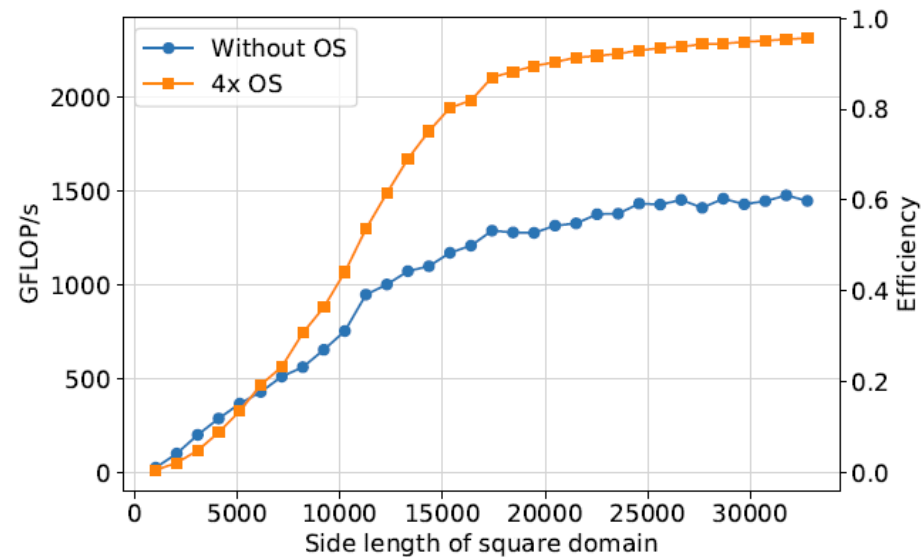
Write single element into  
buffer at thread index



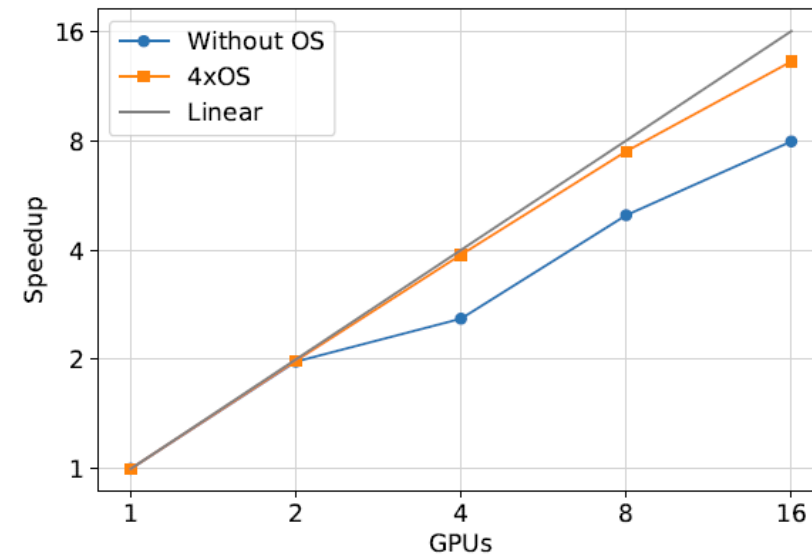
[\[Wikipedia\]](#)

# Performance

- Running fluid dynamics stencil code on consumer-grade (RTX 2070) cluster with up to 16 GPUs
  - Simple split vs 4x oversubscribed (overlapping boundary exchange with computation)
  - Caveat: Uses some unreleased features that will be upstreamed soon



(a) Varying simulated domain size on 16 GPUs.



(b) Strong scaling with domain size 16384.

# Celerity Under The Hood

- Celerity itself uses SYCL in an unusual manner
  - It has *more* information about task relationships than SYCL
  - Manages host-side memory on its own
  - Takes care of all data movement explicitly
- Kernels are submitted in a busy loop, checked for completion using event status queries
  - Want precise control over when kernels are launched
- USM would be a good fit (no implicit DAG)
  - However currently lacking 2D/3D rectangular copy operations

# Outlook / Wishlist

- Improved rectangular copy API, including for USM
- Multi-dimensional array reductions
  - Support for declarative prefix sums
- More precise control over when a kernel is launched
  - For example through a `queue::flush` API



# Wrapping Up

- Visit the Celerity website
  - <https://celerity.github.io>
- Follow the development on GitHub
  - <https://github.com/celerity/celerity-runtime>

This project has received funding from the  
European High Performance Computing Joint Undertaking (JU)  
under grant agreement **No 956137**  
as well from the Austrian Research Promotion Agency  
under grant agreement **No 879201**.