703308 VO High-Performance Computing WS2021/2022
SYCL, Celerity, and Accelerators

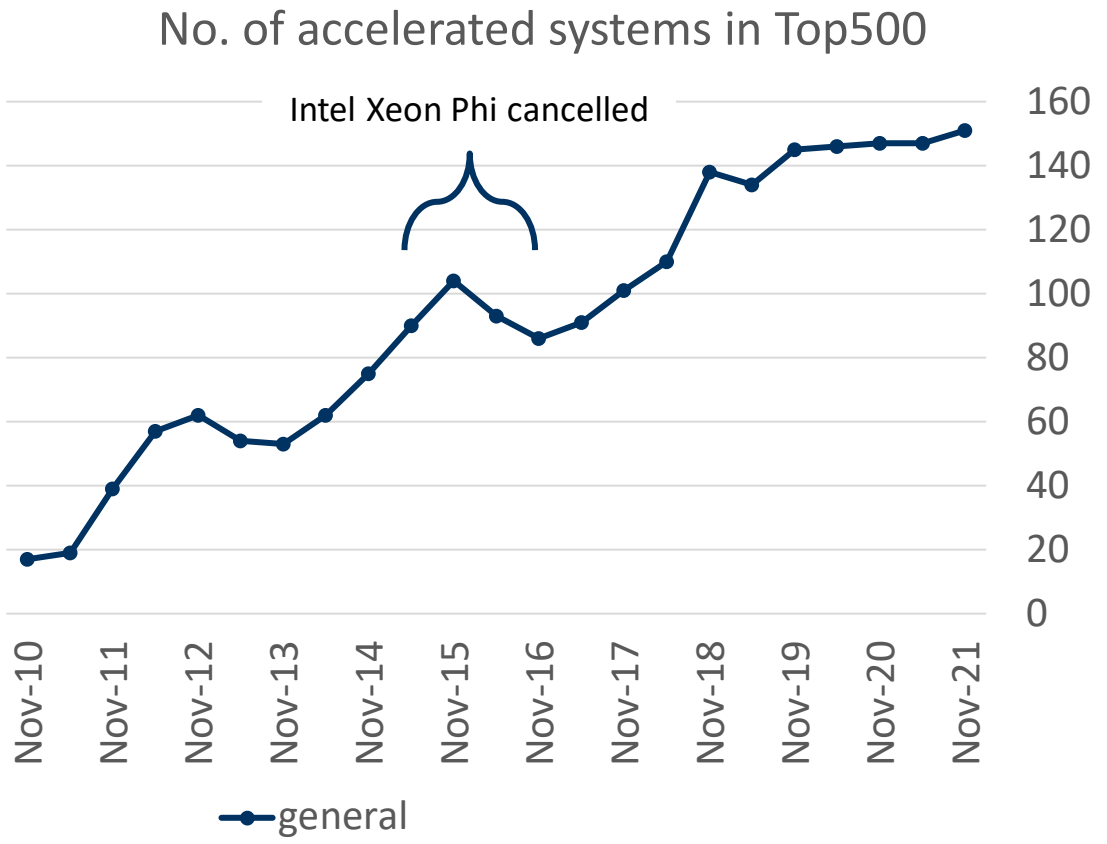
Philipp Gschwandtner

Overview

- Motivation: Why talk about accelerators
  - and then: why NOT about Nvidia/CUDA?
- SYCL
  - high-level C++-based programming model for accelerators
- Celerity
  - UIBK/DPS research, based on SYCL but with distributed memory support

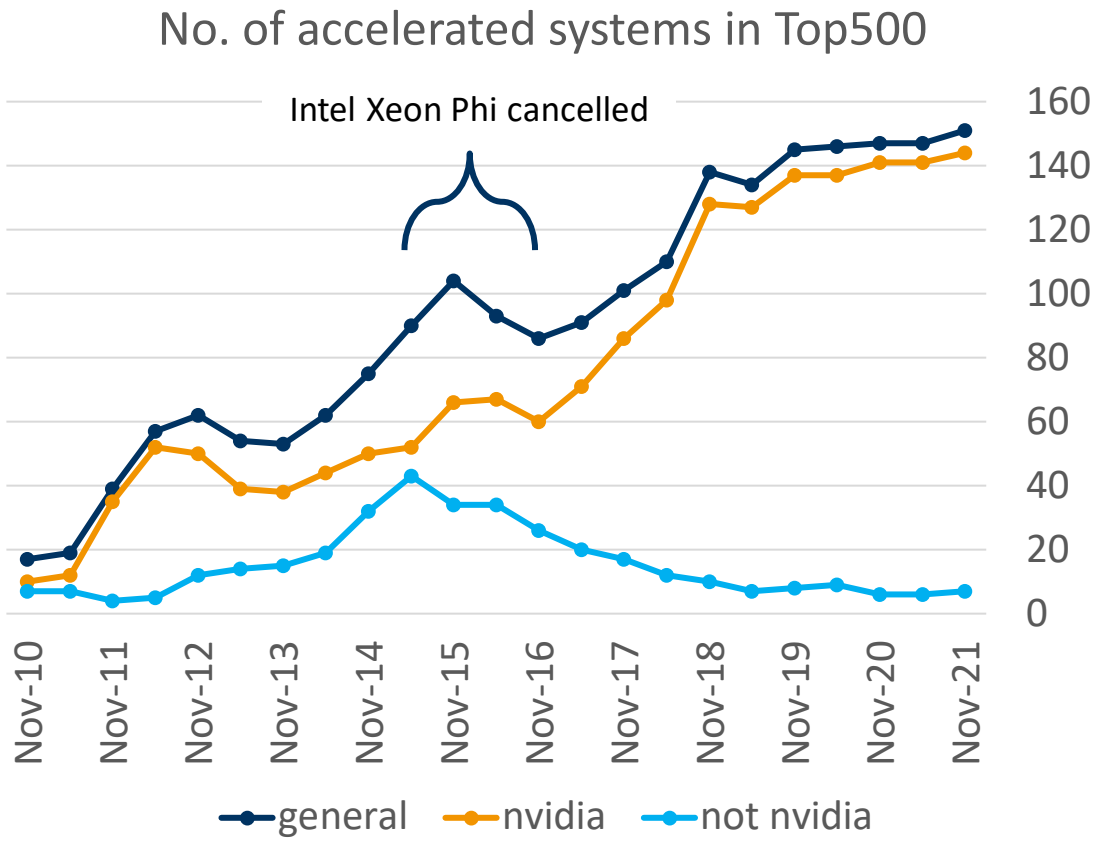
Motivation

- Accelerator market share in HPC has been steadily increasing and will likely continue to do so
  - currently 7 out of top 10 use accelerators (Nov. 2021)
  - 9 out of top 10 on Green500 list
- Application developers **need** to use accelerators to get high performance on modern systems



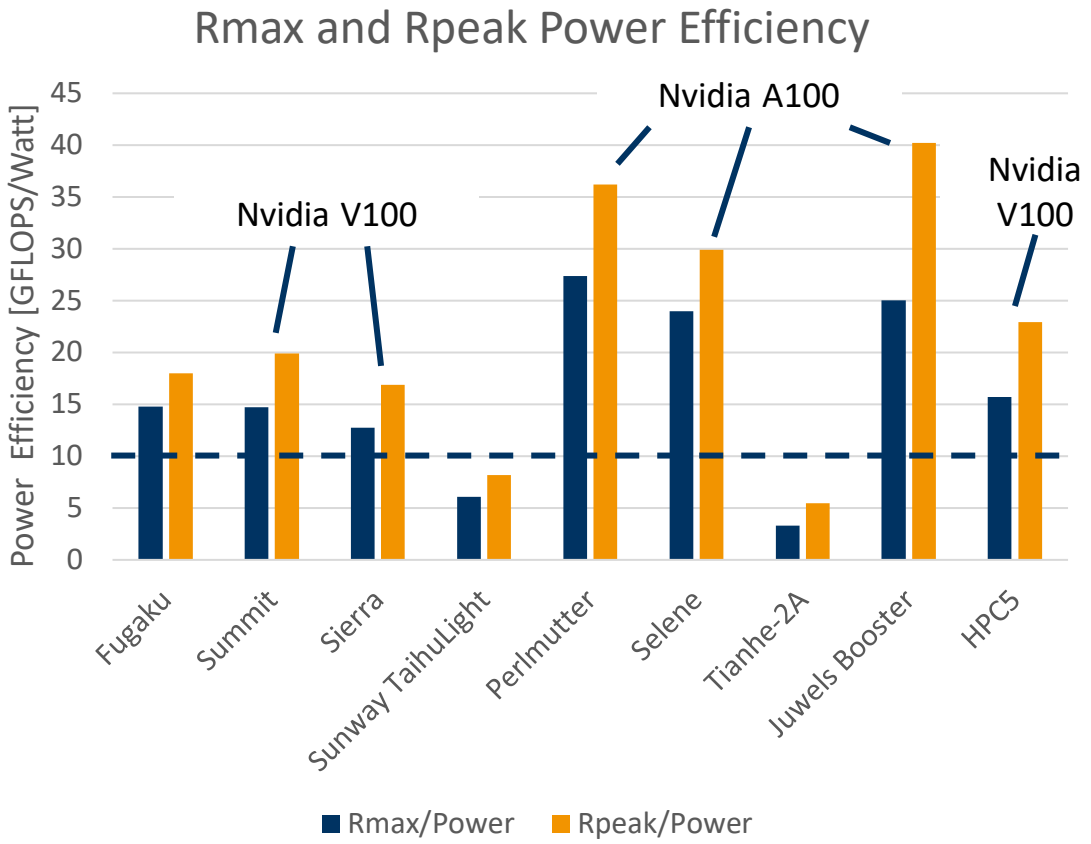
Motivation cont'd

- Problem: No market competition
  - Nvidia is predominant
  - originally also Cell processor (Playstation 3!) and Intel Xeon Phi
  - vastly disappeared since ~2015
- Requires using CUDA
  - vendor lock-in

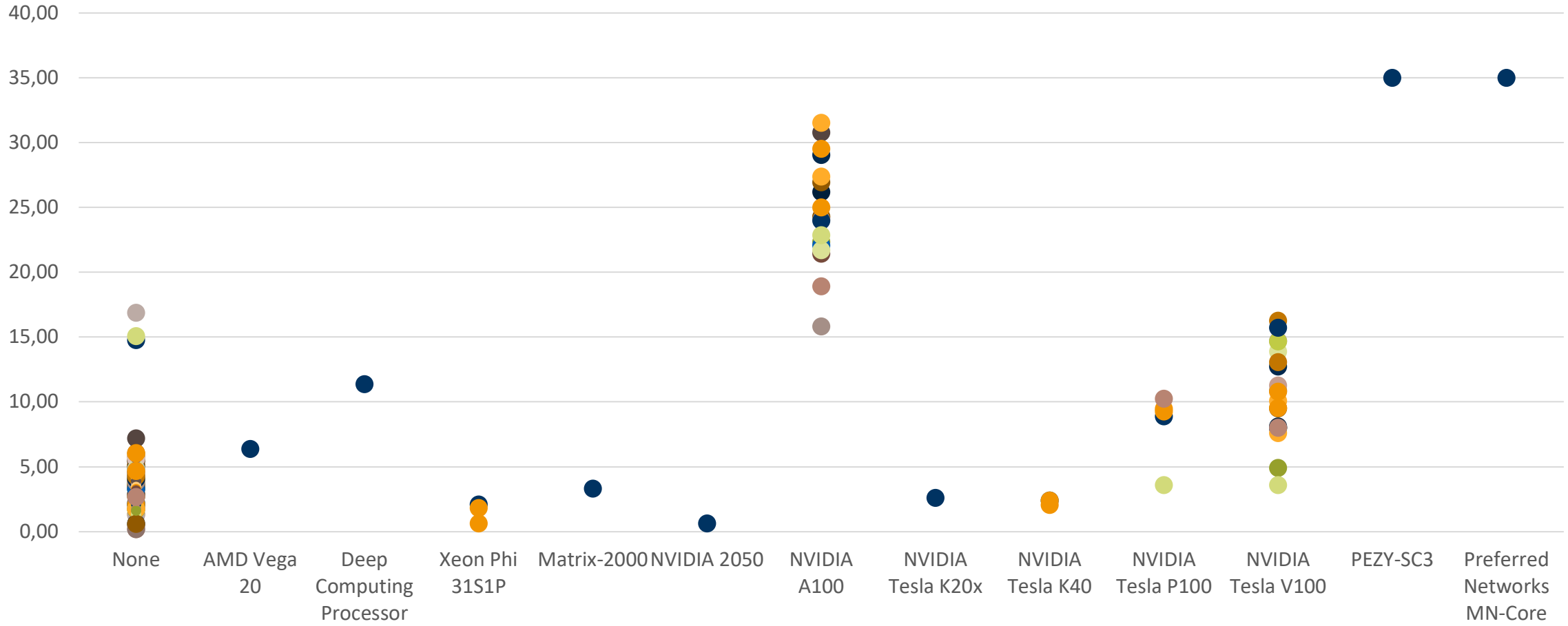


Why Use Accelerators?

- All top 10 systems above 10 GFLOPS/Watt use accelerators
- Exceptions:
  - Fugaku: ARM-based, no accelerators
  - Tianhe-2A: Matrix 2000 accelerators (128 core RISC CPUs)



Power Efficiency of all Top 500 Systems



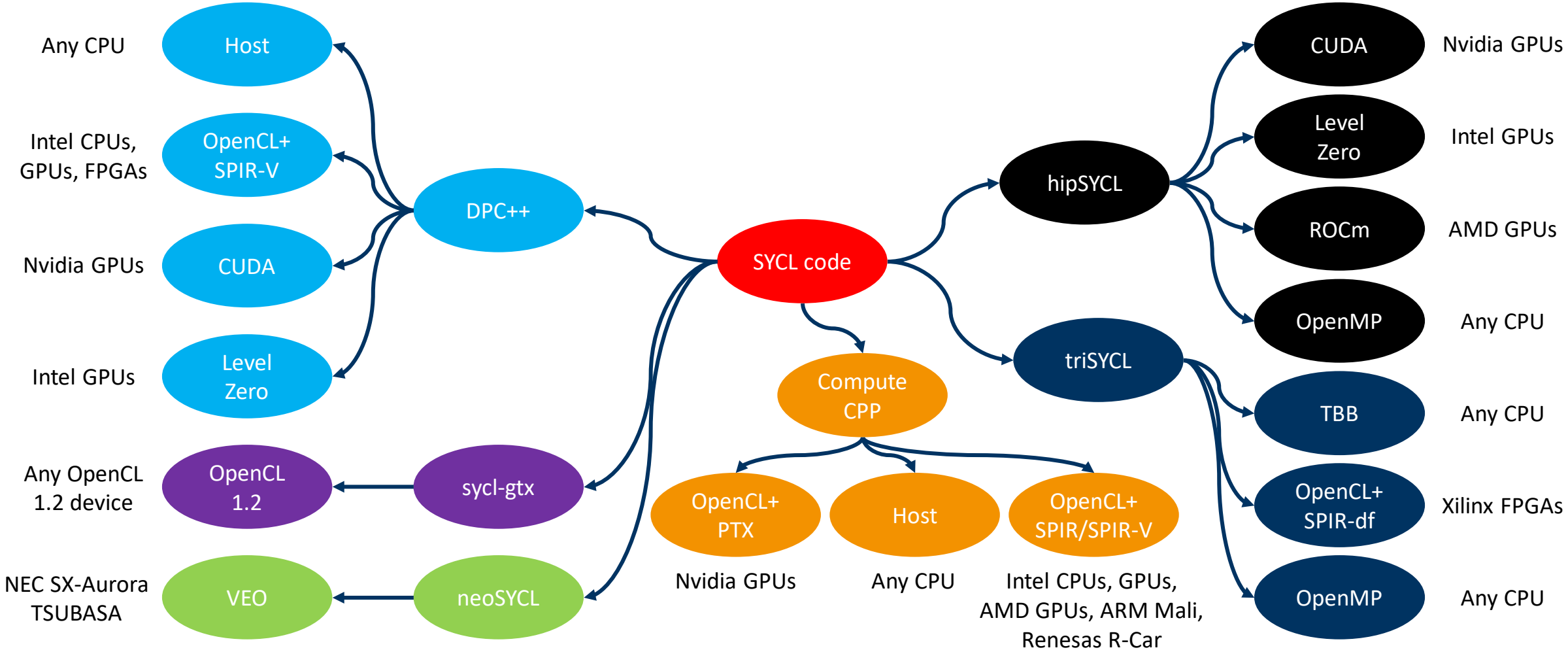
Motivation cont'd

- So what's the software stack?
  - CUDA: very mature but limited to Nvidia, language extension
- Any alternatives?
  - OpenCL: lots of boilerplate, often deprecated
  - ROCm/HIP: limited to AMD, language extension
  - C++ AMP: deprecated since VS 2022, language extension
  - etc…
- SYCL
  - pure C++ (14 to 20, depending on specification version)
  - single-source
  - high-level
  - vendor-independent
  - partial vendor support

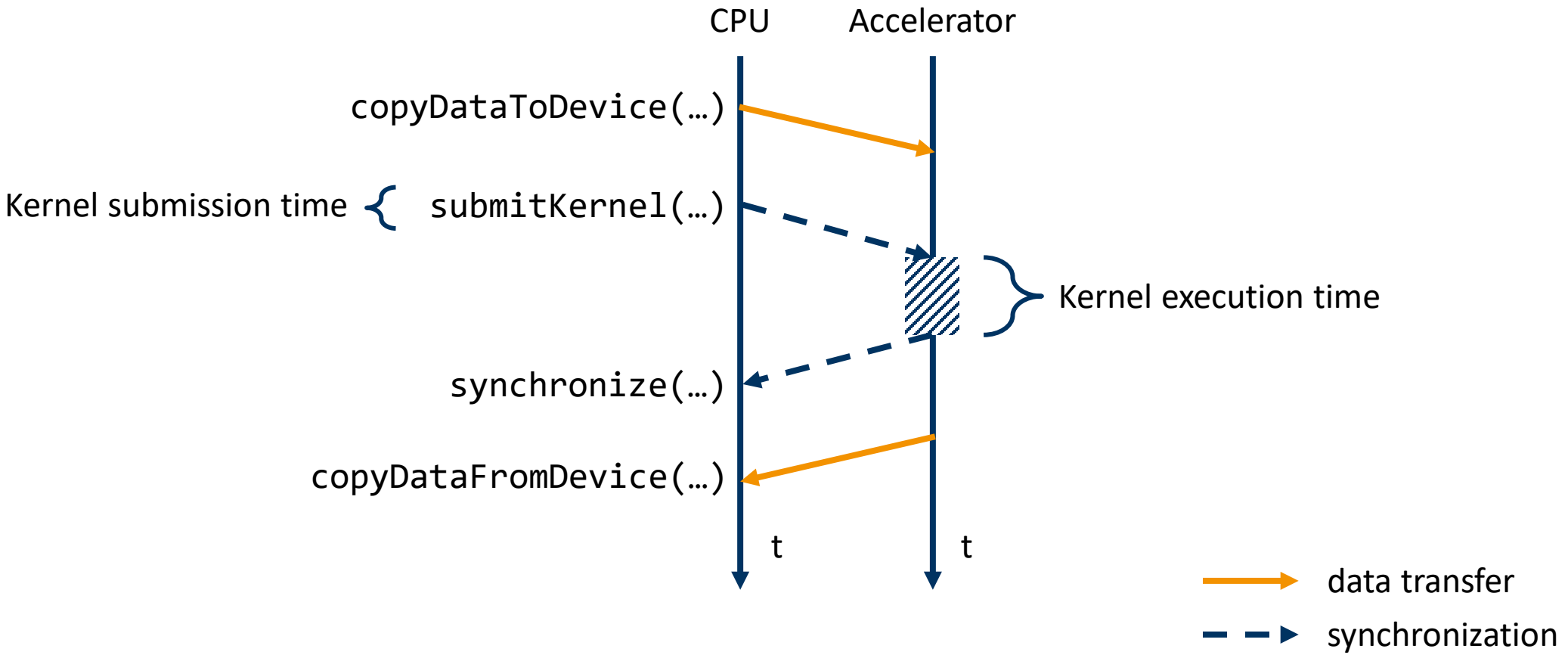
SYCL

- Khronos industry standard
  - successor of OpenCL
- is a programming model / specification, not an implementation
  - several implementations available, see next slide
- not limited to a specific vendor or even accelerator type
  - consider e.g. FPGAs
- single-source (!), high-level, C++-based, programming model specifically tailored towards accelerator computing
  - essentially an embedded DSL within C++
- Leverages C++ semantics to provide abstractions for boilerplate code
  - data migration from/to accelerator, kernel specification, etc.

SYCL Software and Hardware Targets

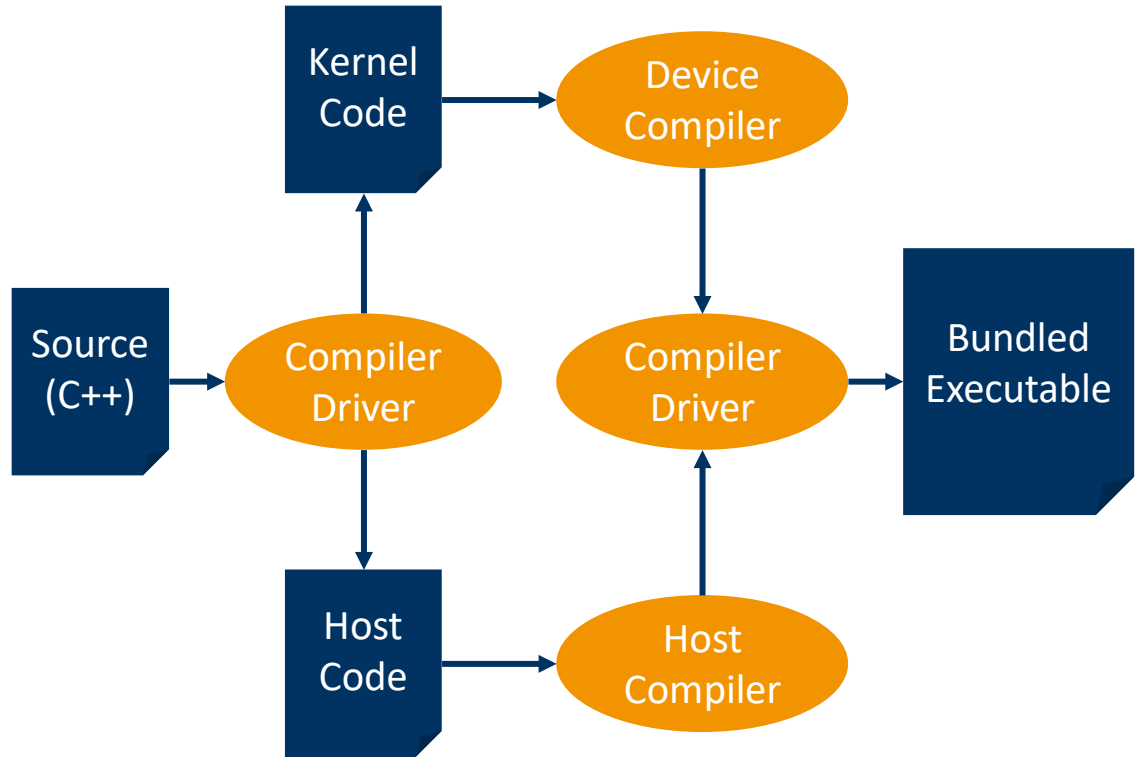


General Accelerator Execution Model



SYCL Compilation Flow (e.g. hipSYCL)

- Compiler driver invokes both host and device compiler respectively
  - can be architecture-specific, e.g. AMD HIP
  - allows compiling for multiple architectures simultaneously into a single executable
  - host compilation pass requires only C++
- Driver then merges compiled device and host binaries into single executable
  - kernels are mapped to the host code at the correct location via their names (C++ type names)



Basic SYCL Concepts

- SYCL buffers encapsulate 1D – 3D dense, typed data
  - are handles to data in device memory
  - data movements to/from device are handled via explicit *“accessors”* that specify read/write behaviour
- SYCL queue controls device activities
  - submission of command groups
  - synchronization with host code
- SYCL command groups
  - hold the actual kernel function, its range and accessors

Code Example: Vector Addition in SYCL

```
cl::sycl::queue queue;

cl::sycl::buffer<float, 2> buf_a(host_a.data(), cl::sycl::range<2>(512, 512));
cl::sycl::buffer<float, 2> buf_b(host_b.data(), cl::sycl::range<2>(512, 512));
cl::sycl::buffer<float, 2> buf_c(host_c.data(), cl::sycl::range<2>(512, 512));

queue.submit([=](cl::sycl::handler& cgh) {
  auto r_a = buf_a.get_access<cl::sycl::access::mode::read>(cgh);
  auto r_b = buf_b.get_access<cl::sycl::access::mode::read>(cgh);
  auto w_c = buf_c.get_access<cl::sycl::access::mode::write>(cgh);
  cgh.parallel_for<class KernelName>(cl::sycl::range<2>(512, 512),
    [=](cl::sycl::item<2> item) {
      w_c[item] = r_a[item] + r_b[item];
  });
});
```

`cl::sycl::queue`

- selects device
  - constructor can take device selector, e.g. `cl::sycl::default_selector`
  - can also enable profiling, asynchronous exception handling, etc.

- controls device by e.g.
  - submitting command groups with kernels, e.g. `cgh.parallel_for<class KernelName>(…)`
  - synchronizing for kernel completion, e.g. `queue.wait()`
  - C++ object lifetime semantics: queue destructor call triggers synchronization

cl::sycl::buffer

- handle to data in device memory
  - can map to host data upon construction via raw pointer
  - C++ object lifetime semantics: allows write-back to host memory upon destruction of the buffer object
- only indirect access via accessors
  - can specify access modes `read`, `write`, or `read_write`
  - access data by indexing into accessor via `cl::sycl::item`

```
// data in host memory
std::vector<float> host_a;

// map to device memory
cl::sycl::buffer<float, 2> buf_a(
  host_a.data(), cl::sycl::range<2>(512, 512));

// create accessors,
// triggers to-device data transfers
auto r_a = buf_a.get_access<cl::sycl::access::mode::
read>(cgh);
auto w_c = buf_c.get_access<cl::sycl::access::mode::
write>(cgh);

// access data on device (inside a kernel function)
cl::sycl::item<2> item = ...;
w_c[item] = r_a[item] + ...
```

SYCL Command Group

- Encapsulates device kernel, data dependencies and setup information
  - forms a single object, i.e. a full recipe for kernel execution
  - prevents race conditions or resource leaks
  - allows for clean, modern programming style

```
queue.submit([=](cl::sycl::handler& cgh) {
  auto r_a = buf_a.get_access<acc::read>(cgh);
  auto r_b = buf_b.get_access<acc::read>(cgh);
  auto w_c = buf_c.get_access<acc::write>(cgh);
  cgh.parallel_for<class KernelName>(
    cl::sycl::range<2>(512, 512),
    [=](cl::sycl::item<2> item) {
      w_c[item] = r_a[item] + r_b[item];
  });
});
```

General Good Practice with Accelerators

- In order to obtain high performance, we need to live on the accelerator
  - avoid data transfers between device and host whenever possible
  - PCI Express is fast, but it's not **that** fast
    (e.g. RTX 3090 TI memory bandwidth ~1 TB/s, PCIe 4.0 16x bandwidth ~32 GB/s)
- Try to implement algorithms with structured, regular, localized patterns
  - similar to SIMD code, just on a much larger scale
  - use contiguous memory access whenever possible
  - GPUs are not designed for irregular codes
  - highly local data access patterns preferred (e.g. stencils) over global data access (e.g. reductions)
- Do not port every application to accelerators (e.g. GPUs)
  - at least consider others, e.g. FPGAs

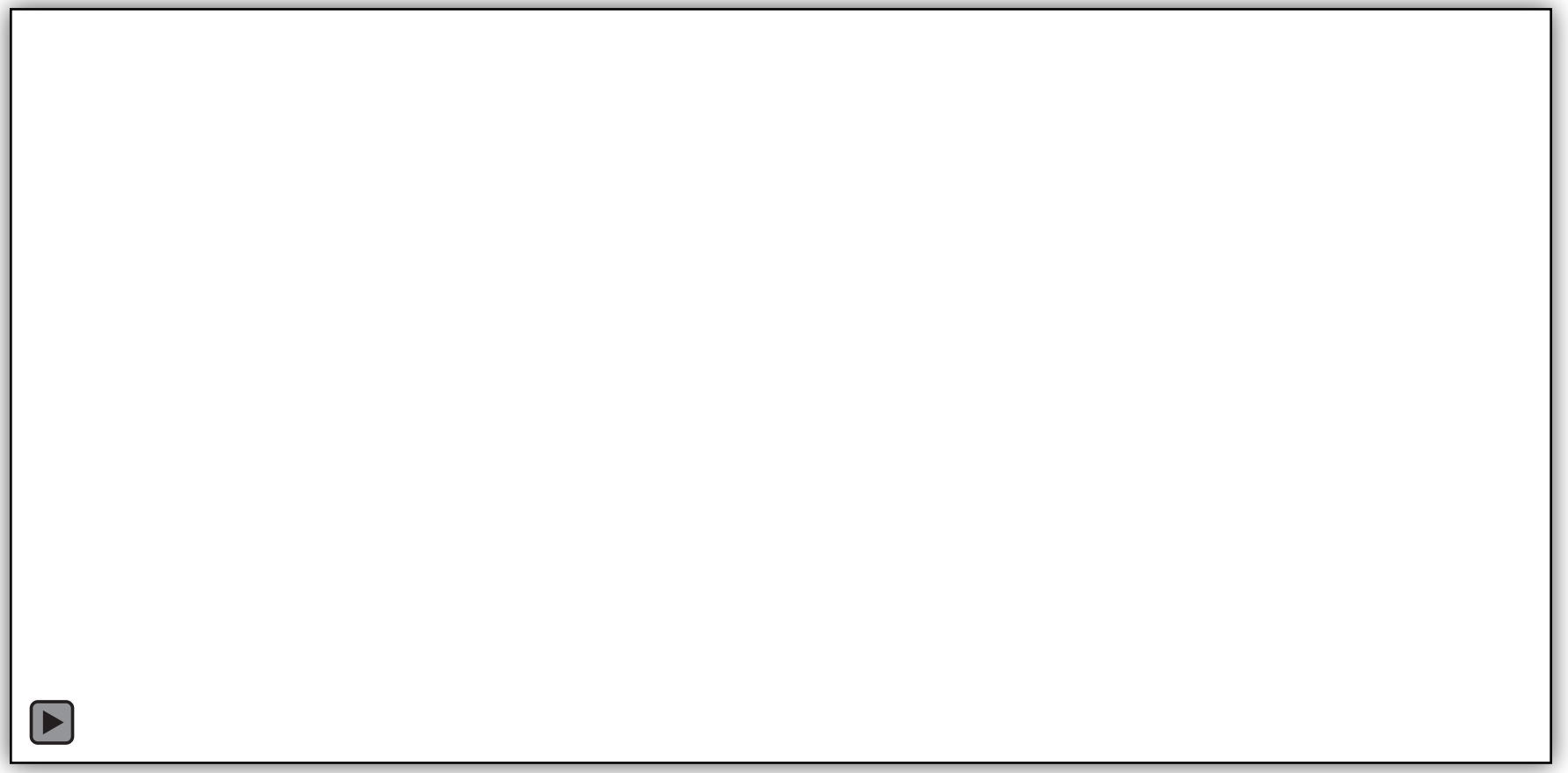
SYCL Limitations

- No automatic support for multi-device programs
  - multiple GPUs are supported but require explicit user code changes
- No support for distributed memory programs
  - must be combined with e.g. MPI
- Solution is often MPI+X, but is there something “better”?
  - Celerity to the rescue!

The Celerity Idea

- A **high-level API** designed from the ground up for accelerator clusters
  - allows to constrain data structures and processing patterns to ones efficient on accelerators ➔ less complex than fully general distributed memory programming
- Based on SYCL
  - adopts all advantages of SYCL and extends it for distributed memory
- Attempt to mitigate adoption issues faced by many frameworks
- The goal is **not** to beat the performance of year-long manually tuned applications

Celerity Overview Animation



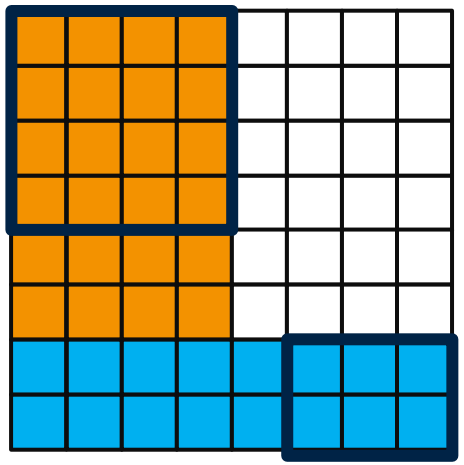
Code Example: Vector Addition in Celerity

```
celerity::distr_queue queue;

celerity::buffer<float, 2> buf_a(host_a.data(), sycl::range<2>(512, 512));
celerity::buffer<float, 2> buf_b(host_b.data(), sycl::range<2>(512, 512));
celerity::buffer<float, 2> buf_c(host_c.data(), sycl::range<2>(512, 512));

queue.submit([=](celerity::handler& cgh) {
  auto one_to_one = celerity::acccess::one_to_one<2>();
  auto r_a = buf_a.get_access<cl::sycl::access::mode::read>(cgh, one_to_one);
  auto r_b = buf_b.get_access<cl::sycl::access::mode::read>(cgh, one_to_one);
  auto w_c = buf_c.get_access<cl::sycl::access::mode::write>(cgh, one_to_one);
  cgh.parallel_for<class KernelName>(sycl::range<2>(512, 512),
    [=](sycl::item<2> itm) {
      w_c[itm] = r_a[itm] + r_b[itm];
  });
});
```

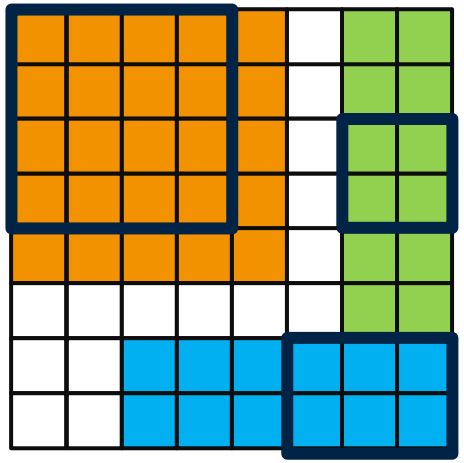
Key API Difference: Range Mappers

- Arbitrary functors mapping from kernel execution range *chunk* to buffer *subrange*: `(celerity::chunk<KD>) -> celerity::subrange<BD>`

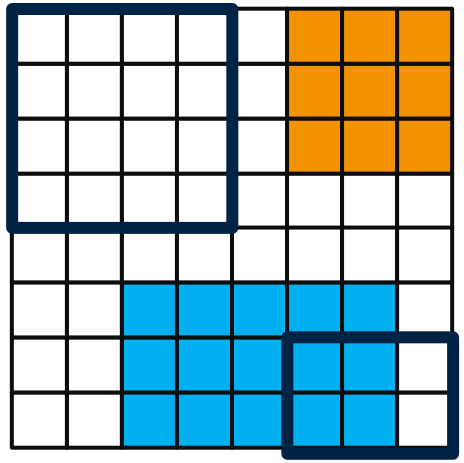
- Common cases with pre-built abstractions in the Celerity API:



`slice<KD>`



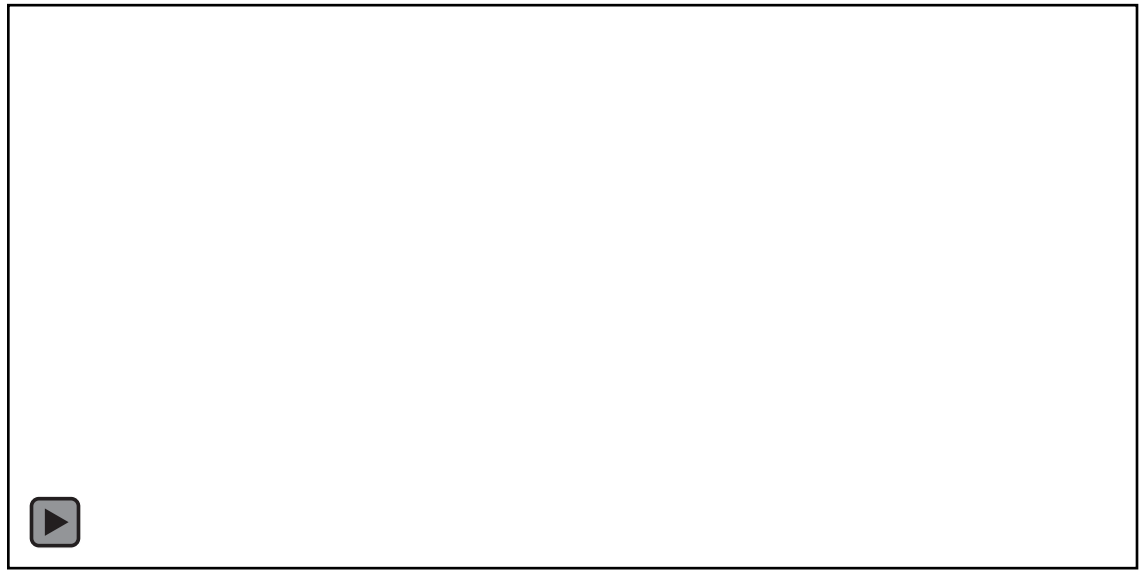
`neighborhood<KD>`



`fixed<KD, BD>`

A Real-world Application: Cronos

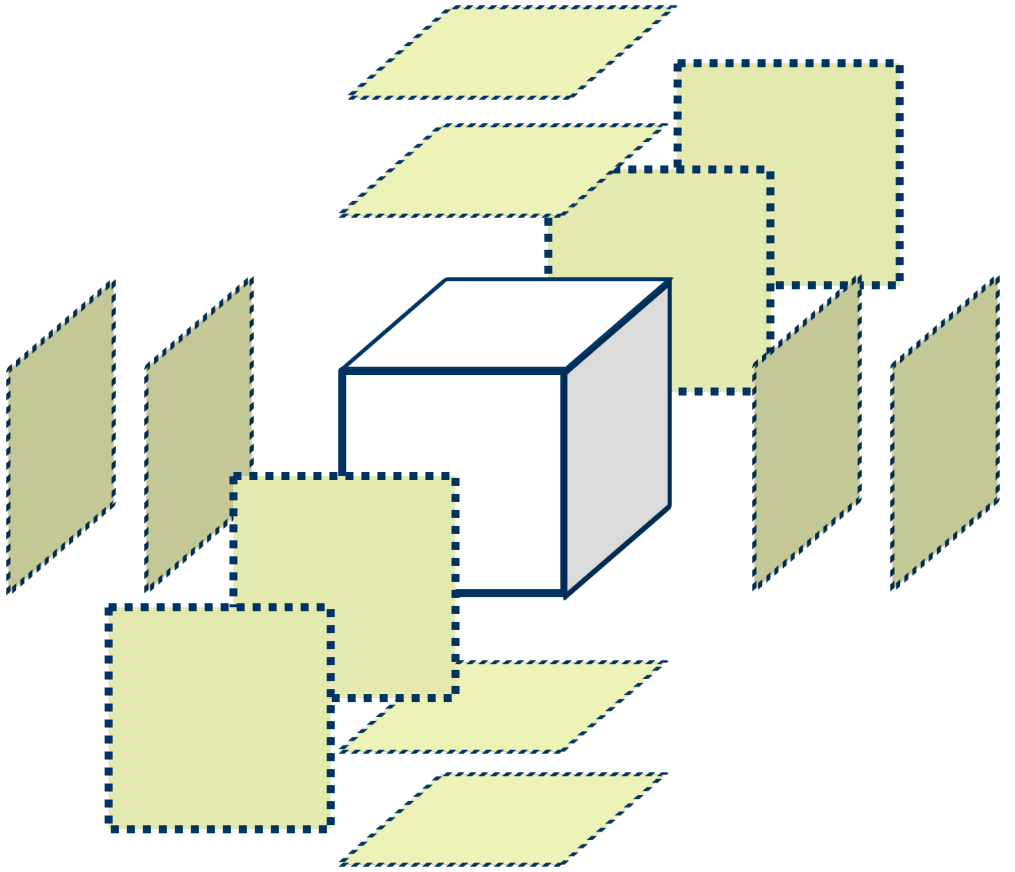
- Structured grid simulation (finite volume) in astrophysics
  - gamma ray emissions of binary star system LS 5039
- Self-written C++14 code
  - Astro- and Particle Physics @ UIBK (Ralf Kissmann, David Huber et al.)
  - MPI-only (no GPUs!)



Density in a binary star system

Performance Properties

- Structured-grid, time-dependent problem
  - local information propagation only
  - essentially a 13-point stencil in 3D
- Only 2 occurrences of global communication
  - reduction to compute dynamic dt dependent on the CFL condition (approx. once per second)
  - broadcast of error stopping condition (unnecessary, computation could be replicated on every node)



Porting Considerations for Celerity and GPUs

✔ structured grid algorithm with minimal global communication
- ✓ excellent scalability on up to 1000 cores (>95% parallel efficiency)
- ✓ used in an ongoing PRACE access project on 12288 cores (~50-60% efficiency)

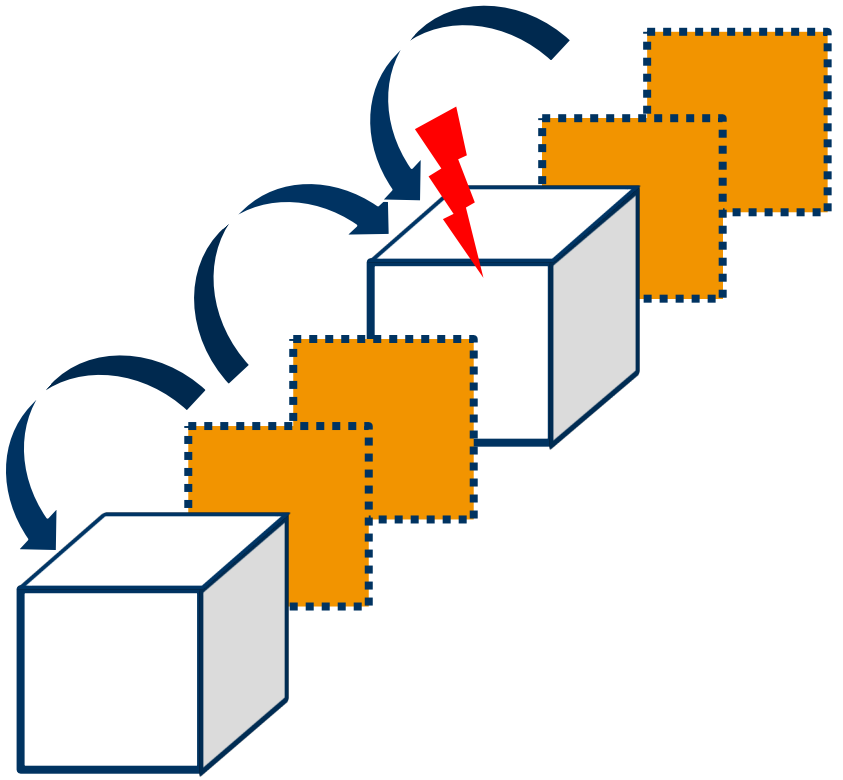
✔ Few external dependencies
- ✓ C++-14-compliant compiler, MPI, HDF5, GSL

✘ 40k lines of C++98-style code
- ✘ a ton of manual memory management (`new` & `delete`)
- ✘ a ton of virtual function calls
- ✘ custom vector and matrix classes with runtime-defined index ranges, e.g.

```
matrix.setLow(-5);
matrix.setHigh(5);
matrix[-3][2] = ...;
```

Algorithmic Changes

- Original code was input-driven
  - compute flux data for 2 faces
  - apply the corresponding change to both neighboring cells
- Forms a race condition
  - a single cell might be written to by two threads simulatenously
- Changed to an output-parallel approach
  - downside: doubles the amount of flux computation



Memory Management

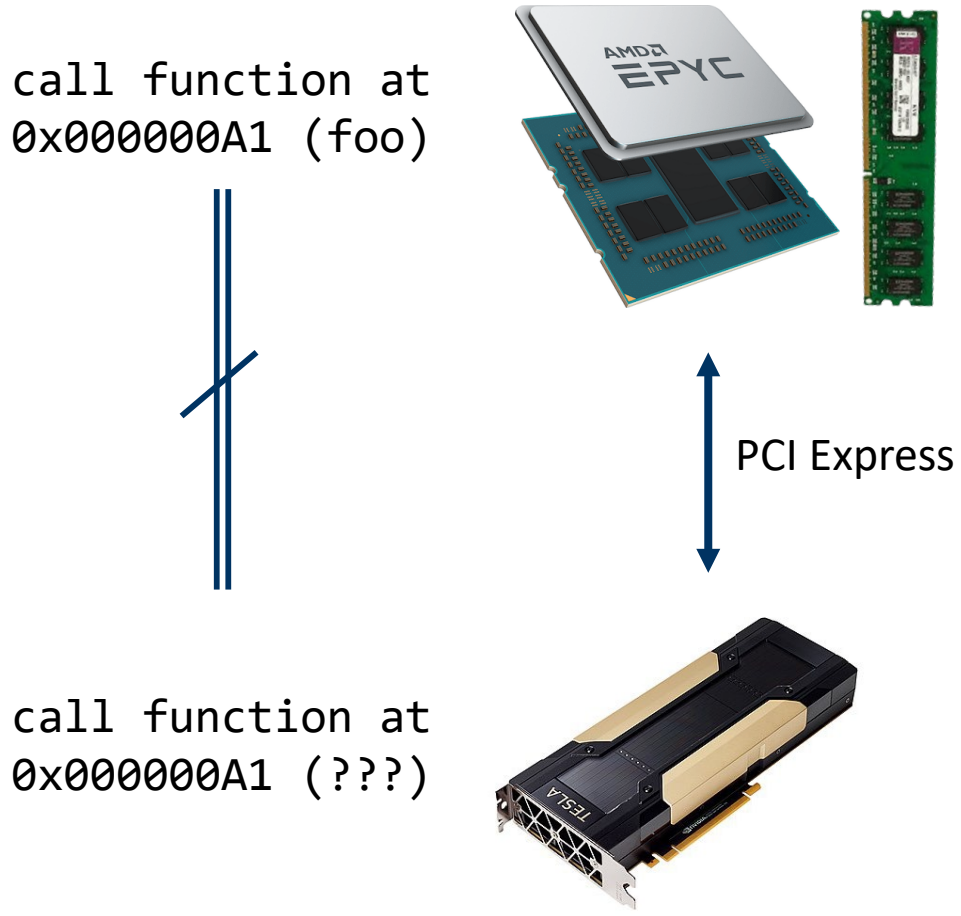
- Cronos contains a lot of manual memory management
  - raw pointers instead of `std::unique_ptr` or `std::shared_ptr`
  - a lot of calls to `new` and `delete`

- error-prone programming style
  - should be replaced with modern programming patterns
  - should use RAII (resource acquisition is initialization)

```
void foo() {
  Bar* bar = new Bar();
  // ...
  delete Bar;
}

void foo() {
  std::unique_ptr<Bar> bar =
    std::make_unique<Bar>();
  // ...
} // automatic "delete bar"
```

Virtual Function Calls in SYCL (and CUDA, …)

- CPU RAM and GPU VRAM are distinct memory address spaces
- Virtual function calls are implemented via pointers in C++
  - become meaningless when transferred between RAM and VRAM
  - requires some form of mitigation
  - C++ template patterns (CRTP), pseudo-virtual function calls using enumerations, etc…
- Only affects code running on GPUs
  - host-only code can use virtual function calls without issues

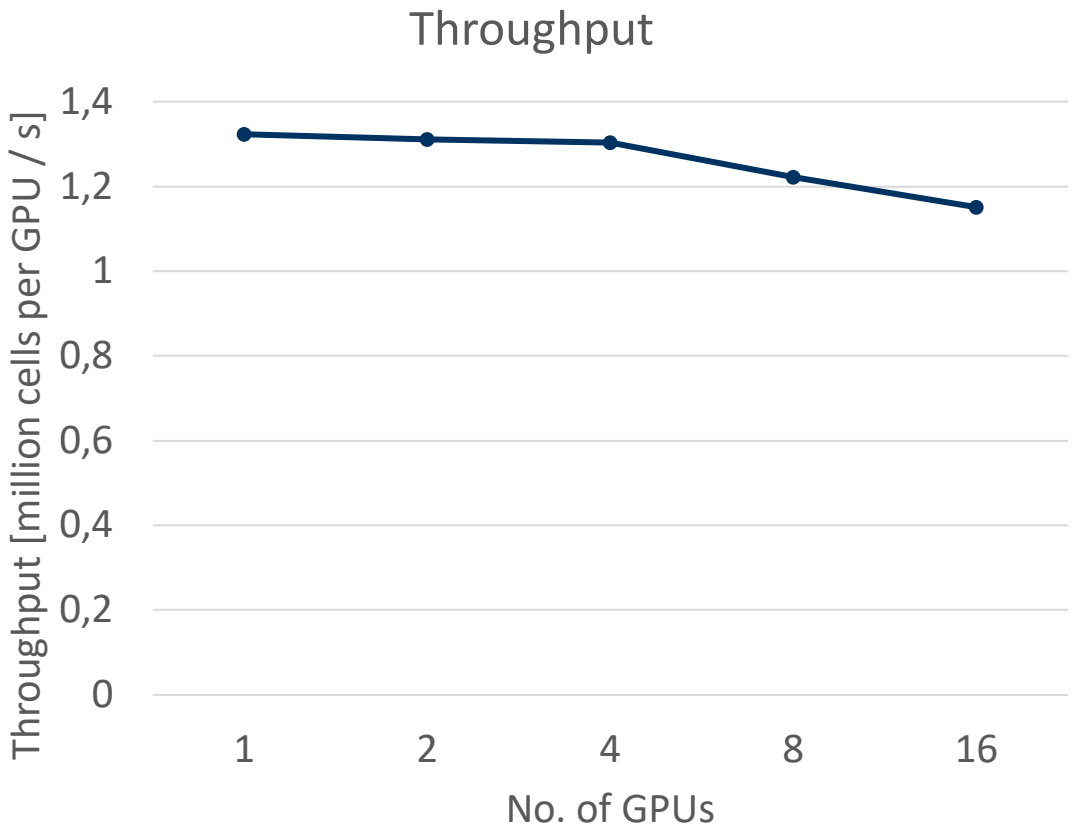


Main Code

```
queue.submit([=](celerity::handler& cgh) {

  auto srcAcc = src.get_access<mode::read>(cgh,
    celerity::access::neighborhood<3>(2,2,2));
  auto dstAcc = dst.nom.get_access<mode::write>(cgh,
    celerity::access::one_to_one<3>());

  Problem problem = {...};

  cgh.parallel_for<class Compute>(range,
    sycl::id<3>{3,3,3}, [=](sycl::id<3> item) {
    int z = item.get(0);
    int y = item.get(1);
    int x = item.get(2);

    numValsType numVals[DirMax], numValsX[DirMax];
    numValsType numValsY[DirMax], numValsZ[DirMax];

    compute(srcAcc, numVals, problem, x, y, z);

    // x direction
    compute(srcAcc, numValsX, problem, x + 1, y, z);
    getChanges(dstAcc, problem, numVals, numValsX,
      x, y, z, DirX);

    // y direction
    compute(srcAcc, numValsY, problem, x, y + 1, z);
    getChanges(dstAcc, problem, numVals, numValsY,
      x, y, z, DirY);

    // z direction
    compute(srcAcc, numValsZ, problem, x, y, z + 1);
    getChanges(dstAcc, problem, numVals, numValsZ,
      x, y, z, DirZ);
  });
});
```

Scalability

No. of GPUs	Wall Time [s]	Parallel Efficiency
1	$24.77 \pm 1.55 \times 10^{-3}$	1.00
2	$12.49 \pm 6.91 \times 10^{-4}$	0.99
4	$6.28 \pm 8.12 \times 10^{-4}$	0.99
8	$3.35 \pm 1.08 \times 10^{-3}$	0.93
16	$1.78 \pm 3.77 \times 10^{-4}$	0.87



Lessons Learned & Good Practice

- Use (deterministic) integration tests
  - pre-compute reference output and compare after every porting step (e.g. use CI)
- Replace old memory allocation code
  - greatly facilitates debugging
- SYCL helps in implementing and debugging
  - can run on GPUs and CPUs, on Windows and Linux
  - enables use of many existing tools
- Use a modern build system such as CMake when targeting multiple platforms

Summary

- Accelerators will continue to be on the rise
  - but will never replace CPUs
- SYCL facilitates modern accelerator programming
- Celerity extends SYCL to distributed memory
  - developed in-house @ UIBK, let me know if you want to work on/with it!

Image Sources

- AMD CPU: https://www.computerbase.de/2019-08/amd-epyc-7xx2-rome-zen-2-specs-release/
- Nvidia Tesla V100: https://wiki.vsc.ac.at/doku.php?id=pandoc:introduction-to-vsc:09_special_hardware:accelerators