GPGPU Computing with SYCL

Advanced School on HPC Computing with GPU Accelerators

CINECA

Outline

- General Introduction to SYCL
- Crash Course in C++
- Basics of SYCL Programming

General Introduction to SYCL

- General Introduction to SYCL
- Crash Course in C++
- Basics of SYCL Programming

GPGPU Programming Models

NVIDIA CUDA

AMD ROCm

SYCL

OpenACC

More Science, Less Programming

OpenCL

RAJA

OpenMP

kokkos

Characterization of Programming Models

Source Model
- Separate
- Single

Base Language
- Pragmas
- Fortran
- C++
- C

Hardware Compatibility
- NVIDIA
- AMD
- Intel
- FPGAs
- ASICs

Key Features
- Portability
- Usability
- Perf.

The Usual Suspects

Model	Source model	Base language	Compatibility	Key features
CUDA	Single-source	C, C++, Fortran	NVIDIA	PERF
HIP	Single-source	C, C++	NVIDIA, AMD	PERF
OpenACC	Single-source	Compiler directives	NVIDIA, AMD	EASE, PERF
OpenMP	Single-source	Compiler directives	NVIDIA, AMD, Intel, …	EASE, PERF
OpenCL	Separate sources	C, C++	NVIDIA, AMD, Intel, …	PORT, PERF

How does SYCL fit in this table?

Introducing SYCL

SYCL (pronounced “sickle”) is a royalty-free, cross-platform abstraction C++ programming model for heterogeneous computing. SYCL builds on the underlying concepts, portability and efficiency of parallel API or standards like OpenCL while adding much of the ease of use and flexibility of single-source C++.

Developers using SYCL are able to write standard modern C++ code, with many of the techniques they are accustomed to, such as inheritance and templates. At the same time, developers have access to the full range of capabilities of the underlying implementation (such as OpenCL) both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly using the underneath implementation, via their APIs. […]

[- SYCL 2020 Specification (rev.8)]

What SYCL provides

- An <u>abstraction layer</u> for low-level parallel programming APIs
  - Access to full-range of capabilities of the low-level API
  - Interoperability with underlying low-level API
- <u>C++-first programming model</u>
  - Single C++ source file for host and device code
  - Integration with modern C++ features (templates, classes, lambdas, …)

What we get from SYCL

- Portability:
  - Code runs on hardware from different vendors (NVIDIA, AMD, Intel)
  - Code targets various accelerator types (CPUs, GPUs, FPGAs, …)
- Productivity:
  - Interact with accelerators via well-designed C++ interfaces
  - C++ features make code simpler to write and more expressive
- Performance:
  - Fine-tune code for top-level performance on specific architectures
  - Interact directly with low-level APIs when necessary

Standard vs. Implementations

SYCL is an open industry standard for programming a heterogeneous system. […]

[…] SYCL implementations can provide SYCL backends for various heterogeneous APIs, implementing the SYCL general specification on top of them. We refer to this heterogeneous API as the SYCL backend API. The SYCL general specification defines the behavior that all SYCL implementations must expose to SYCL users for a SYCL application to behave as expected.

- SYCL 2020 Specification (rev.8)

Chronology of the SYCL Standard

- SYCL 1.2 released in 2015

  Based on OpenCL as (only) backend and C++11

- Provisional SYCL 2.2 introduced in 2016

  Added support for OpenCL 2.2, never finalized

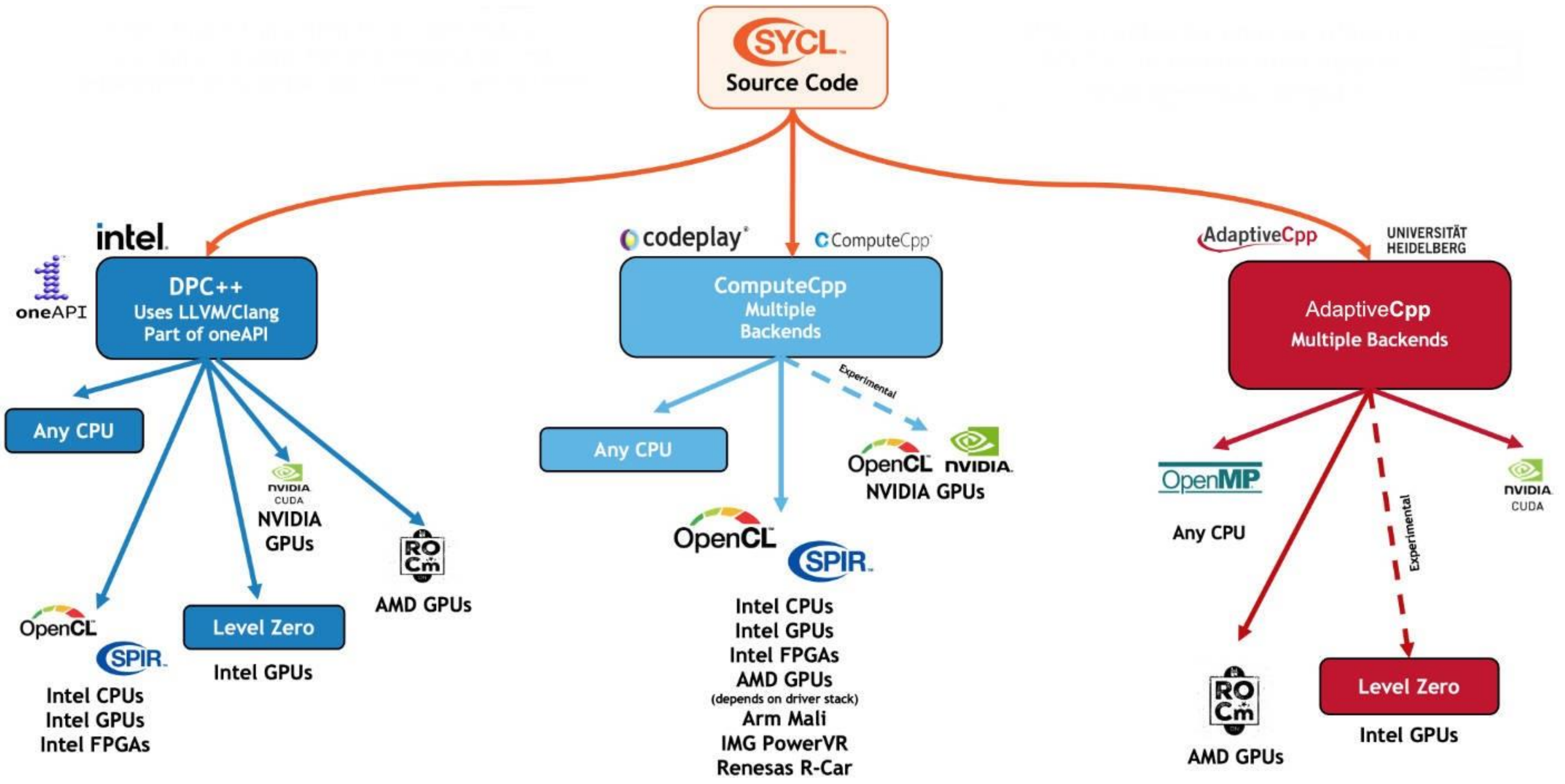
- SYCL 1.2.1 released in 2017 (revision 7 in April 2020)

  Introduces support for C++17 and parallel STL algorithms

- SYCL 2020 released in 2021 (revision 8 in September 2023)

  Based on C++17, generalized backend, much more …

SYCL Implementations



SYCL Implementation Strategies

- SMCP (*single-source, multiple compiler passes*)

  > […] there are separate host and device compilers. Each SYCL source file is compiled two times: once by the host compiler and once by the device compiler. […]

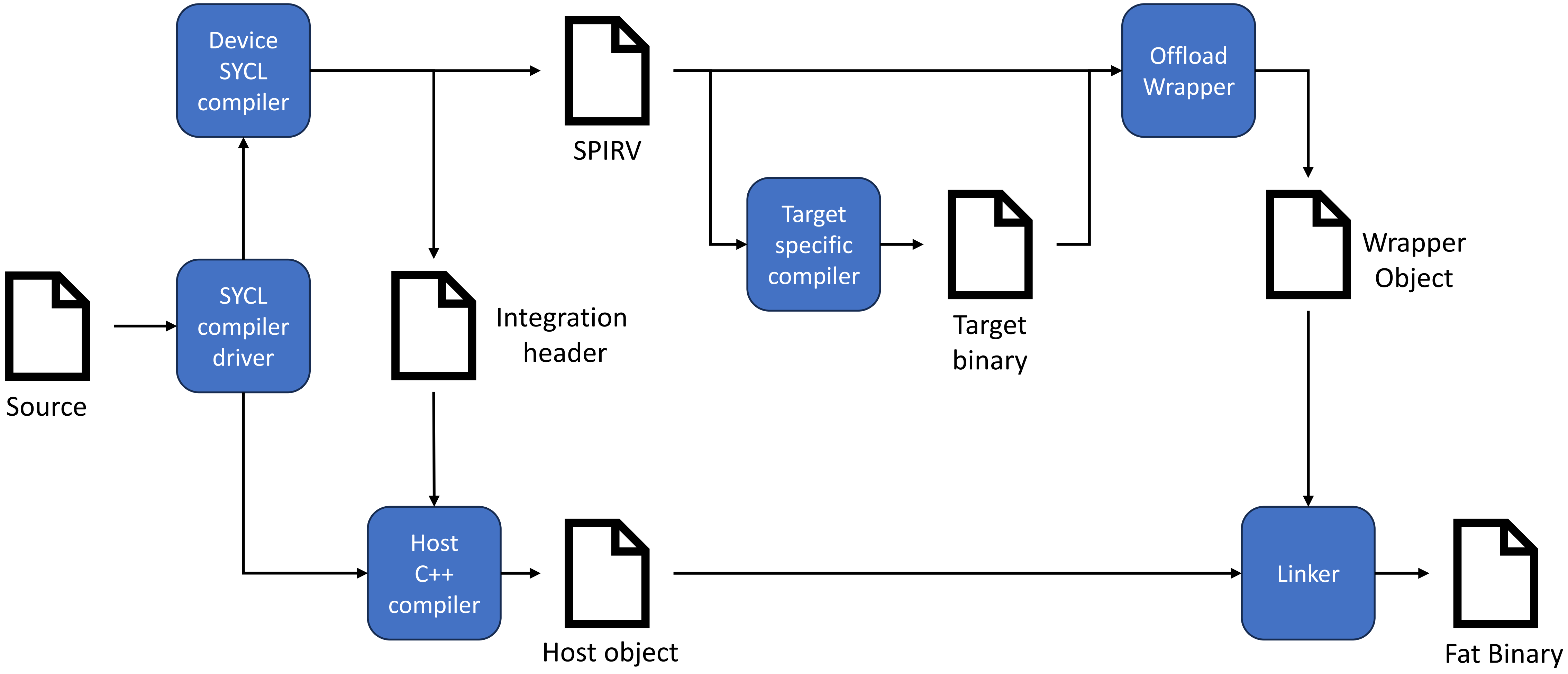
- SSCP (*single-source, single compiler pass)*

  > […] the vendor implements a custom compiler that reads each SYCL source file only once, and that compiler generates the host code as well as the device images for the SYCL kernel functions. […]

- Library-only

  > […] implement SYCL purely as a library, using an off-the-shelf host compiler with no special support for SYCL. In such an implementation, each kernel may run on the host system.

SMCP Compilation Workflow



The Usual Suspects + 1

Model	Platform support	Source model	Base language	Key features
CUDA	NVIDIA	Single-source	C, C++, Fortran	PERF
HIP	NVIDIA, AMD	Single-source	C, C++	PERF
OpenACC	NVIDIA, AMD	Single-source	Compiler directives	EASE, PERF
OpenMP	NVIDIA, AMD, Intel, …	Single-source	Compiler directives	EASE, PORT
OpenCL	NVIDIA, AMD, Intel, …	Separate sources	C, C++	PORT, PERF
SYCL	NVIDIA, AMD, Intel, …	Single-source	C++	ALL

Crash Course in C++ for C programmers

- General Introduction to SYCL
- Crash Course in C++
- Basics of SYCL Programming

Modern C++ for C Programmers

There are 5 main additions in C++

1. References
2. Classes
   - OOP Features (Access Control, Methods, Constructors, …)
   - RAII Paradigm
3. Templates
4. Exceptions
5. Lambdas (C++ >= 11)

References

```cpp
int x;
int& foo = x; // create reference to x

foo = 42; // access x through reference

std::cout << x << std::endl;
```

A *reference* allows us to declare an alias to another variable

- The reference can be used freely only as long as the original variable lives
- References are often used in function arguments to avoid copying large objects

Classes

```cpp
class Foo {
  int attribute {5};

public:
  int function(void) {
        return attribute;
  }
};

Foo foo {}; // create Foo instance
```

C++ classes are similar to C structs, but they can also contain functions

Access control

It is possible to specify three access policies for class members

- public

  Attribute or function is accessible by everyone.

- protected

  Attribute or function is accessible only from derived classes.

- private

  Attribute or function is accessible only within class, or *friends*.

The default access policy is private.

Constructors and Destructor

```
class Foo {
  size_t _size;
  int* _data;

public:
  Foo(size_t size) { // parametrized constructor
    _size = size;
    _data = new int[_size];
  }

  ~Foo() { delete[] _data; } // destructor
};
```

C++ has special functions designed to create and destroy class instances

Resource Acquisition Is Initialization (RAII)

When creating objects that require resources (memory, files, GPU, …)

1. Allocate resources during object initialization (within constructor)
2. Release resources at object destruction (within destructor)

Main advantages:

- No manual resource management
- Clear resource ownership
- Code encapsulation
- Code locality
- Allocation fails → initialization fails

Memory

- `std::unique_ptr`
- `std::shared_ptr`
- `std::string`
- `std::vector`
- `std::map`

Files

- `std::istream`
- `std::ostream`
- `std::ifstream`
- `std::ofstream`

Mutexes

- `std::lock_guard`
- `std::unique_lock`
- `std::shared_lock`

Templates

```cpp
template <typename T, size_t N> // type and non-type parameters
class Array { // templated class
  T _data[N];
  size_t _size{ N };
  // ...
};

template <typename T> // type parameter
T max(T a, T b) { // templated function
  return (a > b) ? a : b;
}
```

C++ templates allow to define functions and classes using generic argument types

Template Argument Types

Three types of template arguments:

- Type argument - `template<typename T>`
  - From C++20 also *concepts*
- Non-type argument - `template<<type> <name> = <default>>`
  - Type can be `int`, `pointer`, `enum`
  - From C++20 also `double`, `std::nullptr_t`
- Template arguments - `template<template<…>>`

Throwing Exceptions

```cpp
int foo(int arg1, arg2, arg3) {
  //check that third arg is nonzero
  if (arg == 0) {
    throw std::runtime_error("Invalid argument");
  }

  return arg1 * arg2 / arg3;
}
```

Errors in C++ are (usually) handled by throwing exceptions of some kind

- Standard exception types can be found in the C++ standard library
- Custom exception types can be created by deriving `std::exception`

Catching exceptions

```cpp
try {
  foo(15, 7, 0);
}
catch (const std::runtime_error& e) { // catch specific type
  std::cerr << e.what() << std::endl;
}
catch (const std::exception& e) { // catch generic exception
  std::cerr << e.what() << std::endl;
}
```

Thrown exceptions must be catched, otherwise the program fails

- Specific exceptions must be catched befor generic exception (base class)

Lambdas

```
int sum{ 100 };

auto add_to_sum_ref = [&](int num) {
  sum += num; // capture by ref can be modified
}

auto add_to_sum_val = [=](int num) -> int {
  return sum + num // capture by val is read-only
}

add_to_sum_ref(50);
sum = add_to_sum_val(70);
```

C++ lambdas are unnamed function objects capable of capturing variables in scope

Lambda Captures

Environment variables can be captured in two ways:

- by value `[=]`
  - good for trivially copiable types
  - captures are read-only
- by reference `[&]`
  - good for complex types
  - captures can be modified

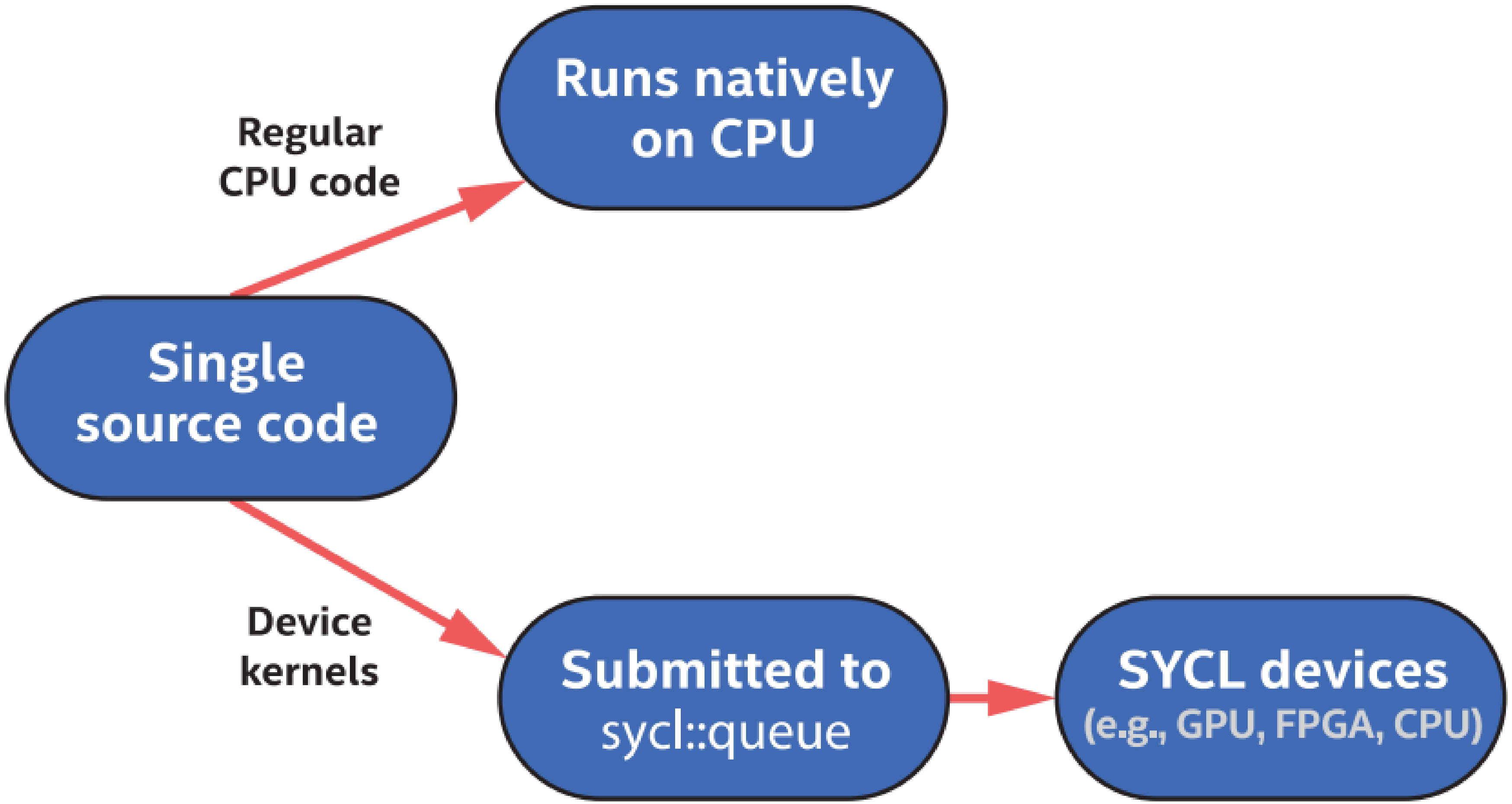
It is possible to choose specific capture type for each variable:

- If unspecified, default is to capture by value
- `[&var1, &var2] ->` captures var1 and var2 by ref, the rest by value

Basics of SYCL Programming

- General Introduction to SYCL
- Crash Course in C++
- Basics of SYCL Programming

SYCL Execution Model



Host and Device

Host Code	Device Code
• is executed synchronously • handles offload of computations to device • orchestrates data transfers to device • handles synchronization between host and device • can be single-threaded or multi threaded (with caution)	• is executed asynchronously • describes computations to be performed on device (kernels) • is identified with specific constructs • supports a restricted set of operations • should be adapted to run on the target architecture

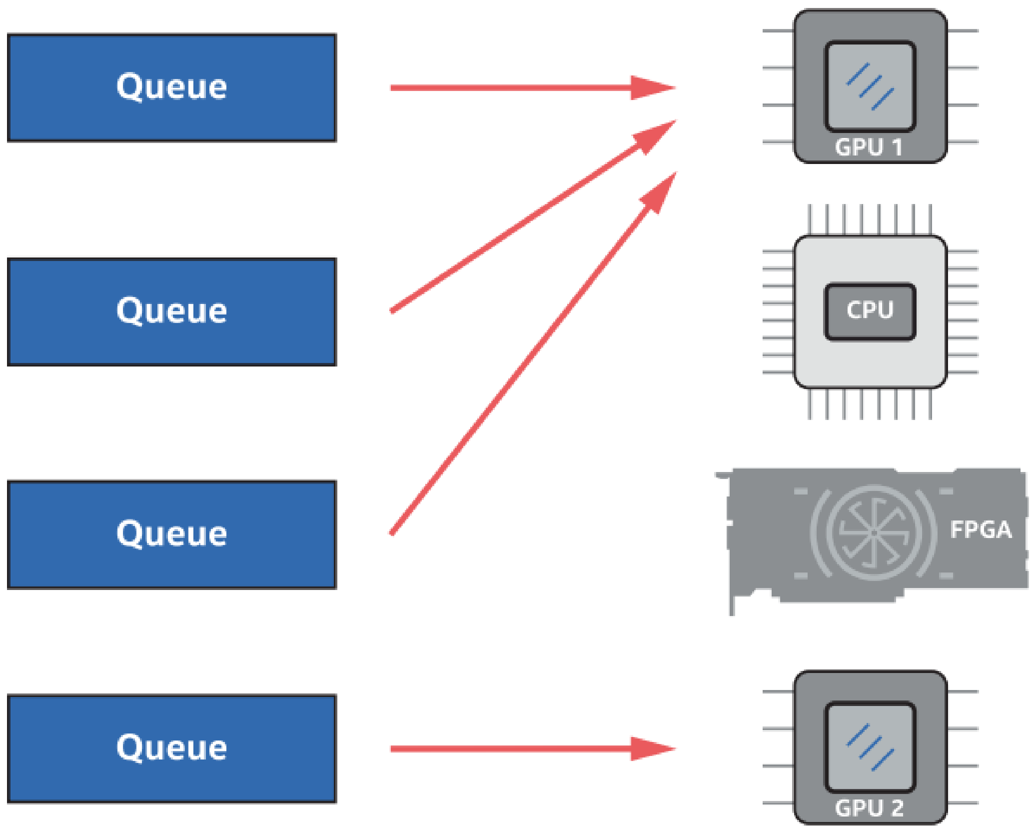
SYCL Queues

A queue is an interface to a device, and it is used to:

- query device properties
- submit kernels for execution on device
- transfer data to device
- synchronize execution

Important things:

- A queue is bound to a <u>single</u> device (selected during queue initialization)
- It is possible to have more than one queue associated to the same device



SYCL Queues

```
#include <iostream>
#include <sycl/sycl.hpp>

int main () {
    // create queue with specific device selector
    sycl::queue q { sycl::cpu_selector_v };

    // query device properties
    auto d_name { q.get_device().get_info<sycl::info::device::name>() };
    auto max_cu { q.get_device().get_info<sycl::info::device::max_compute_units>() };

    std::cout << "Device name        : " << name   << std::endl;
    std::cout << "Max compute units : " << max_cu << std::endl;
    return 0;
}
```

<CL/sycl.hpp> -> SYCL 1.2.1
<sycl/sycl.hpp> -> SYCL 2020

Device Selection

It is possible to select a device in two ways:

- Using a device selector (implementation-independent)
  - Default selectors (deafult, cpu, gpu, accelerator)
  - Custom selectors (created ad-hoc to target desired device)
- Using external selection means (implementation-dependent)
  - Setting environment variables
  - Using compilation targets

Work Submission

```
q.submit( [&](sycl::handler& h) { // command group lambda
  // command group used to call action
  h.parallel_for(N, [=](sycl::id<1> idx) { // action lambda
    // device code
    C[idx] = A[idx] + B[idx];
  });
});
```

A bit of SYCL jargon:

- To send work to a device we call `.submit()` on the queue associated to the selected device
- `.submit()` takes as argument a C++ lambda known as the **command group**
- The argument of the command group lambda is known as the command group **handler**
- Inside the command group we use the handler to call an **action**

Command groups

A command group contains:

- Host code to set up "environment" for execution of action
- Device code to offload (action or memory operation)

The handler is an interface for:

- Executing actions (on device)
- Encoding memory requirements
- Setting dependencies on other kernels

```
[&](sycl::handler& h) { // command group

  // set up environment
  sycl::accessor A {bufA, h, sycl::read_only};
  sycl::accessor B {bufB, h, sycl::read_only};
  sycl::accessor C {bufC, h, sycl::write_only};

  // command group used to call action
  h.parallel_for(N, [=](sycl::id<1> idx) { // action
    // device code
    C[idx] = A[idx] + B[idx];
  });

} // end command group
```

Allowed Actions

There are only six possible actions that can be submitted in a queue, and they belong to two different cathegories:

DEVICE CODE EXECUTION	MEMORY OPERATIONS
• `single_task`	• `copy`
• `parallel_for`	• `fill`
• `parallel_for_work_group`	• `update_host`

Things to remember:

- Only one action is allowed per command group
- Inserting more than one action in same `.submit()` will result in a compilation error
- For GPU programming we use *`parallel_for`* 99% of the times (CUDA-like)

Structure of an Action

An action is composed of:

- An execution pattern
- The work-item (threads) distribution
- A C++ lambda containing the code to be executed by each work-item

```
h.parallel_for(
  N, // sycl::range<1>(N) → work-item distribution
  [=](sycl::id<1> idx) { // action lambda
    // device code
    C[idx] = A[idx] + B[idx];
  }
);
```

Important things:

- The lambda parameter represents a single work item, its type depends on work-item distribution
- The lambda must capture by value (always)

```
h.parallel_for(
  sycl::range {N, N}, //  work-item distribution
  [=](sycl::id<2> idx) { // action lambda
    int lin_idx = idx[0]*N + idx[1]; // linear index
    C[lin_idx] = A[lin_idx] + B[lin_idx];
  }
);
```

Work-Items Distribution

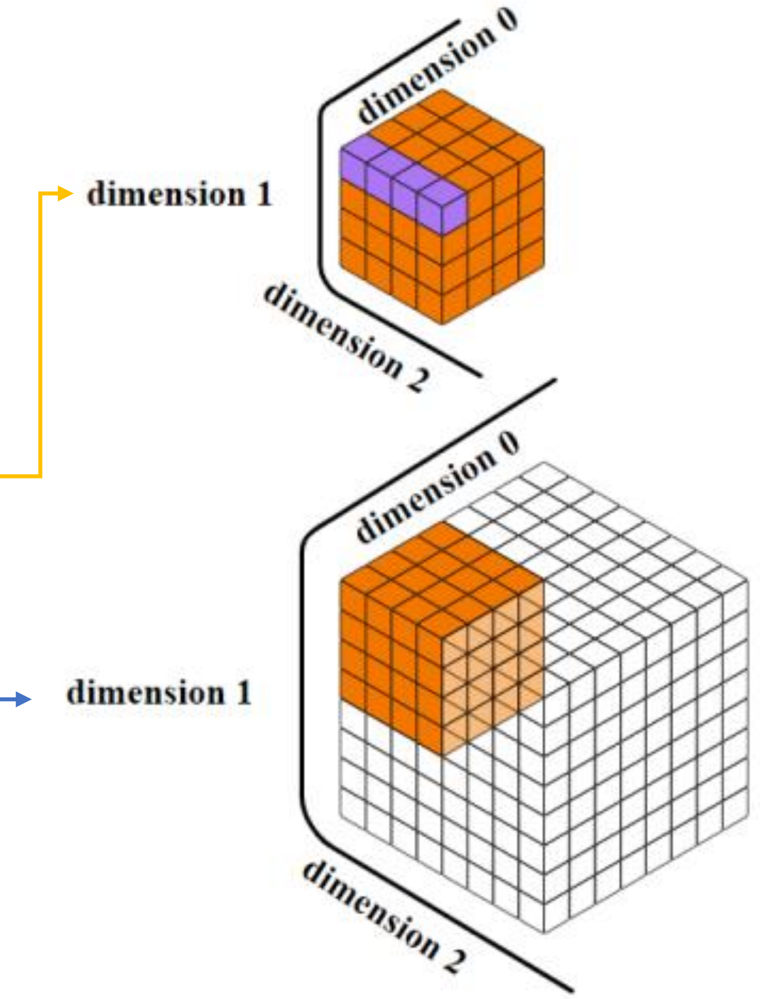
Work-item distributions is specified using `nd_ranges`

Thise objects are composed of two ranges:

- the first is the global number of work-items in each dimension
- the second is the number of work-items in each dimension in a **workgroup**

Things to remember:

- A workgroup is a subset of the work-item distribution with added functionalities, such as synchronization (CUDA block)
- When using `nd_range` the argument of the action lambda must be a `nd_item` with the same number of dimensions



Memory handling

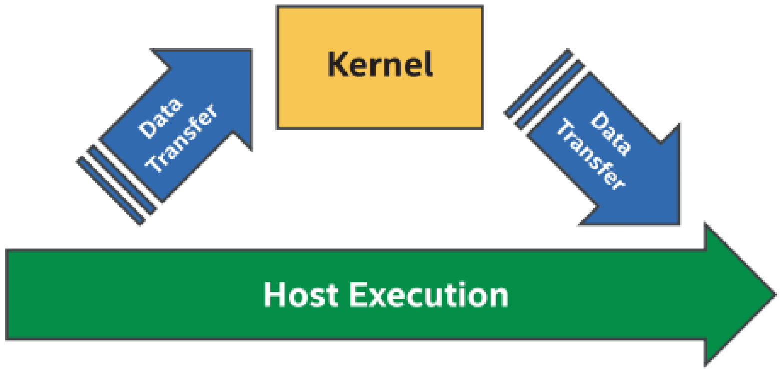
SYCL provides two distinct ways to handle host and device memory:

UNIFIED SHARED MEMORY

- Pointer based approach (CUDA-like)
- Easy to integrate with C/C++ code
- Data movements can be explicit or implicit

BUFFERS AND ACCESSORS

- Native SYCL approach
- Based on C++ abstractions (RAII)
- Data movements are only implicit



Data Movement

Data transfers between host and device can happen in two ways:

EXPLICIT DATA MOVEMENT

- Provides full control on where and when data are moved
- Allows overlapping data movements with computations for maximum performance
- It is error prone
- It is time consuming (for user)

IMPLICIT DATA MOVEMENT

- Provides no control, everything is handled by the runtime
- May result in sub-optimal performance
- Results in easier to write and debug code
- Requires less developer effort

USM – Implicit Data Movement

Characteristics:

- Pointer-based
- Data transfer is performed by SYCL runtime
- *cudaMallocShared()*-like

Best for:

- Integrating/porting existing C/C++ codes
- Not extreme-performance

```
// C++style API
int* A { sycl::malloc_shared<int>(N, q) };
int* B { sycl::malloc_shared<int>(N, q) };

// C-style API
int* C { (int*) sycl::malloc_shared(N*sizeof(int), q) };

// Device code for C[i] = A[i] + B[i]

sycl::free(A, q);
sycl::free(B, q);
sycl::free(C, q);
```

USM – Explicit Data Movement

Characteristics:

- Pointer-based
- Data transfer performed by user
- { *cudaMallocDevice()* + *cudaMemCpyAsync()* }-like

Best for:

- Integrating/porting existing C/C++ codes
- Maximum performance
- Fine tuning data movement

```
// C++ style API device allocation
int* devA { sycl::malloc_device<int>(N, q) };
int* devB { sycl::malloc_device<int>(N, q) };

// C-style API
int* devC { (int*) sycl::malloc_device(N*sizeof(int), q) };

// copy from host to device
q.copy<int>(hstA, devA, N); // C++ style
q.memcpy(devB, hstB, N*sizeof(int)); // C-style

// device code for devC[i] = devA[i] + devB[i]

// copy back from device
q.copy<int>(devC, hstC, N);
```

Asynchronous by default!!!

Buffers

SYCL Buffers:

- Provide an abstract view of memory
- Can be accessed both from host and device
- Migrate data automatically and maintain coherency
- Can wrap other C++ objects
  - Arrays / Pointers
  - C++ containers
- Should be accessed exclusively through **accessors**

```cpp
// create buffer directly
sycl::buffer<int> bufA { sycl::range {N} };

// create buffer from dynamic array
int* arrB { new int[N] };
sycl::buffer<int> bufB { arrB, N };

// create buffer from container
std::vector<int> vecC(N);   // creates vector with size N
sycl::buffer bufC { vecC };
```

Accessors

SYCL Accessors:

- Allow to safely access buffer memory
- Describe what we intend to do with memory (r/w/rw)
- Express data dependencies in kernels
- Allow SYCL runtime to define execution order

```
{ // create host accessors to init buffer memory
  sycl::host_accessor hstA { bufA, sycl::write_only };
  sycl::host_accessor hstB { bufB, sycl::write_only };
  // ...
} // host accessors killed at end of scope

q.submit([&](sycl::handler& h) {
  // accessors on device
  sycl::accessor devA { bufA, h, sycl::read_only };
  sycl::accessor devB { bufB, h, sycl::read_only };
  sycl::accessor devC { bufC, h, sycl::write_only };
  // ...
});
```

Things to Remember:

- SYCL Buffers must not be accessed directly, because this would invalidate memory coherency between host and device. Using accessors guarantees a safe access.

Scheduling kernels

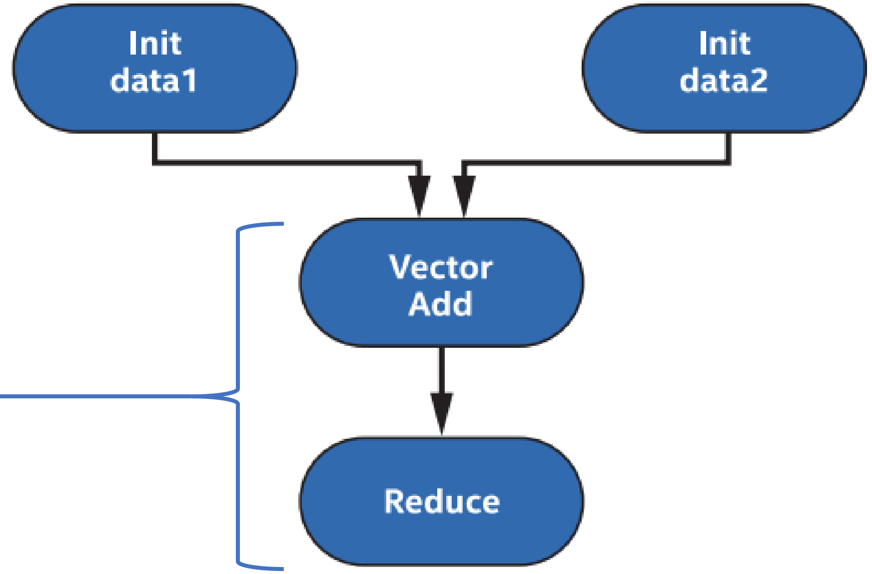
Kernels usually have constraints on their execution:

- The data they require need to be moved to device before execution
- They must follow a specific order to yield correct results

Execution order can be controlled in 3 ways:

- Forcing synchronization using `.wait()`
- Via impicit dependencies (accessors)
- Via explicit dependencies (events)

… or combining all these methods together



Events

How to use events:

- Create events (from *`.submit()`* calls)
- Set dependency on a specific event using command group handler

Things to Remember:

- Use *`.wait()`* on the last event or *`.submit()`* call to ensure completion of all scheduled kernels

```
sycl::event e1 = q.fill<int>(devA, 3, N); // fill devA
sycl::event e2 = q.fill<int>(devB, 2, N); // fill devB

sycl::event e3 = q.submit([&](sycl::handler& h) {
  h.depends_on(e1); // set dependency on event
  h.depends_on(e2); // -
  // ... devA = devA + devB
});

sycl::event e4 = q.submit([&](sycl::handler& h) {
  h.depends_on(e3); // -
  // ... reduction on devA
});
```

SYCL Error Handling

SYCL has two types of errors:

- Synchronous exceptions
  - Thrown as soon as invalid operation happens
  - Handled as simple C++ exception
  - Typical case for host code
- Asynchronous exceptions
  - Thrown (much) after invalid operation happend
  - Catched by SYCL runtime, thrown at C++ exceptions later
  - Typical case for device code

```cpp
try { // synchronous error
  sycl::buffer<int> B1{ sycl::range{ N } };
  sycl::buffer<int> B2{ B1, 4, sycl::range{ N } };
}
catch (const sycl::exception& e) {
  std::cout << e.what() << std::endl;
}

try { // asynchronous error
  sycl::event e = q.submit([&](sycl::handler& h) {
    // empty command group is an error
  });
  // e.wait_and_throw(); // throw for single kernel
  q.wait_and_throw();    // throw for all queued
  // q.throw_asynchronous();
}
catch( ... ) { ... }
```

Useful Resources

- SYCL Reference Book

  Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL | SpringerLink

- CodePlay's SYCL Academy Repository

  codeplaysoftware/syclacademy: SYCL Academy, a set of learning materials for SYCL heterogeneous programming (github.com)

- SYCL 2020 Specification

  SYCL™ 2020 Specification (revision 8) (khronos.org)

- SYCL 2020 Reference Guide

  sycl-2020-reference-guide.pdf (khronos.org)