SYCL : Integrated compiler runtime for accelerated Deep Learning

Abhilash Majumder
(abhilash.majumder@intel.com)
AI Frameworks and Compiler Engineer

INTEL

ABSTRACT

LLMs and generative models have become the mainstream deep learning architectures for industries globally and with customized optimizations there is a lot of developments among deep learning compilers . However, majority of the frameworks supporting exa-scale model training/finetuning (such as Pytorch or Jax) has extensive device specific compiler runtime codes which are performant on a single specific hardware type. To democratize deep learning models and benchmark them across different runtime devices, there is a need to support a device agnostic compiler backend which can be run on Nvidia/AMD or Intel (other ISA's of x86 CPU or llvm/clang supported GPU). This talk focuses on how to create such backends using SYCL (originally from Khronos) and induce platform specific optimizations; also building abstractions on top of llvm/clang to suit SYCL runtime optimizations for GPUs /CPUs and FPGAs .The generalization of standard compiler runtime is enabling deep learning frameworks delegate device specific IR without having to write customized Api calls.

CONTENTS

- Scope of the presentation
- Brief Introduction to LLVM Compiler Backend
- Introduction to SYCL (SYCL program model, memory, parallel stl, adaptive cpp)
- Custom Kernels in SYCL (along with Pytorch case study)
- DPCT (DPC++ Toolkit, automigration samples)
- Performance Results
- References
- Conclusion

Code available at: https://github.com/abhilash1910/ISO-CPP-SYCL-Compiler-Conference

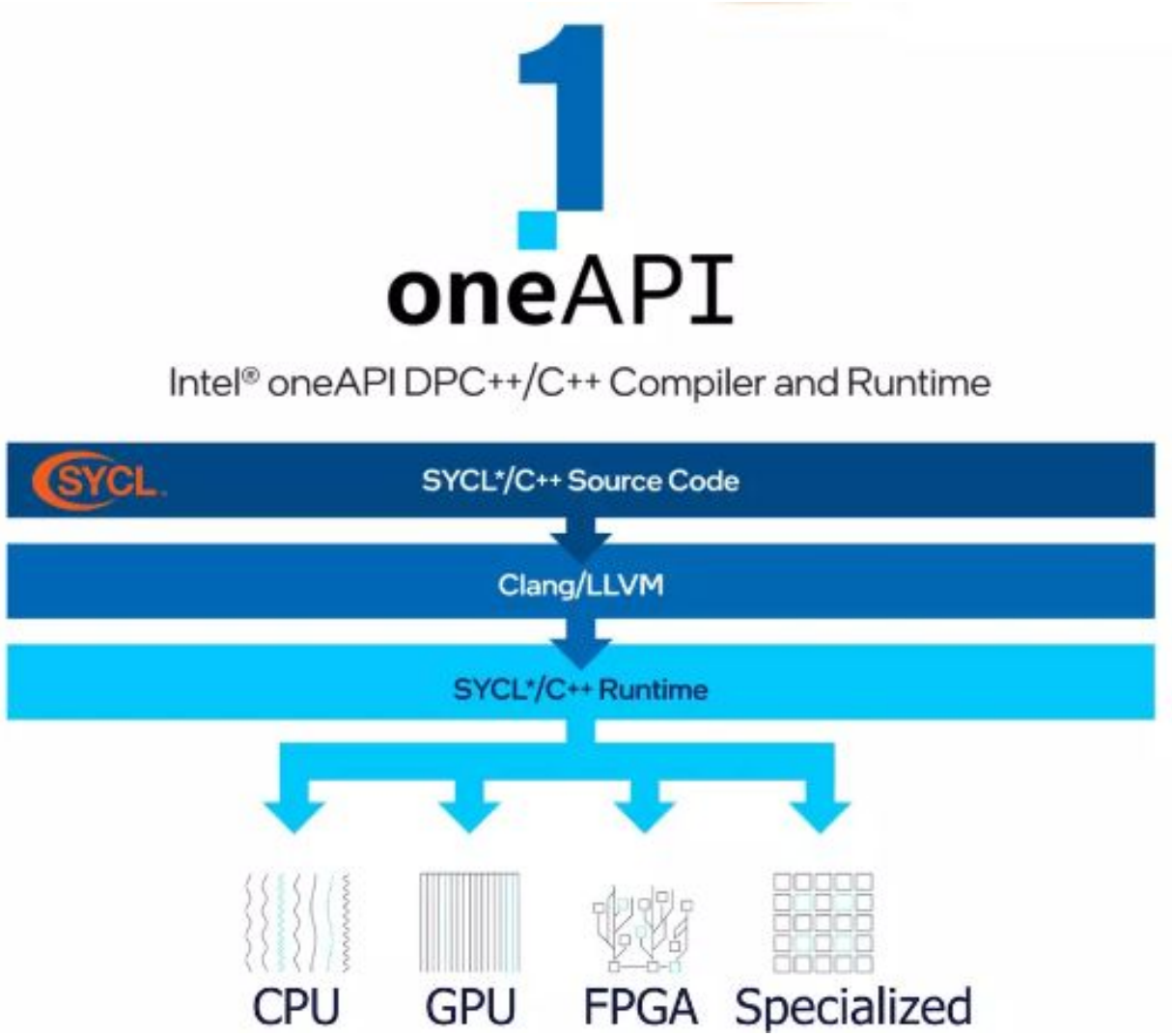
Scope

- What is and why do we need SYCL ? Code once and build anywhere
- Where does cross platform SYCL language come into picture
- Semantics of Device & Host Asynchronous Task Scheduling and parallel programming model of SYCL
- Effects on Deep Learning
- DPC++ Compiler for auto migration of CUDA to SYCL code

Introduction to LLVM Compiler Backend (Clang)

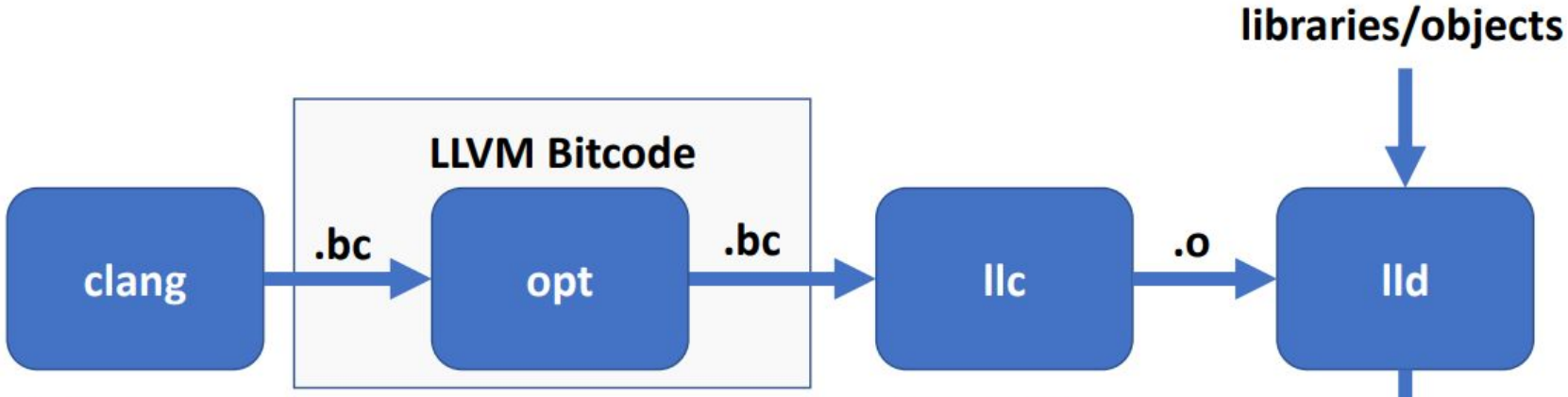
LLVM Compiler: Industrial strength toolchain of compiler technologies

- oneAPI's Data Parallel C++ (DPC++) is an Intel-led project that lets us write programs that execute across different computing systems without major, time-consuming code changes.
- The compiler contains three compiler drivers — icx, icpx, and dpcpp — to further simplify tailoring code for unique support requirements. These drivers are for compiling and linking C programs, C++ programs, and C++ programs with SYCL extensions, respectively.
- The icpx (SYCL) compiler is the heart of oneAPI and provides foundation of SYCL/C++ runtime across accelerators.



Introduction to LLVM Compiler Backend (Clang)

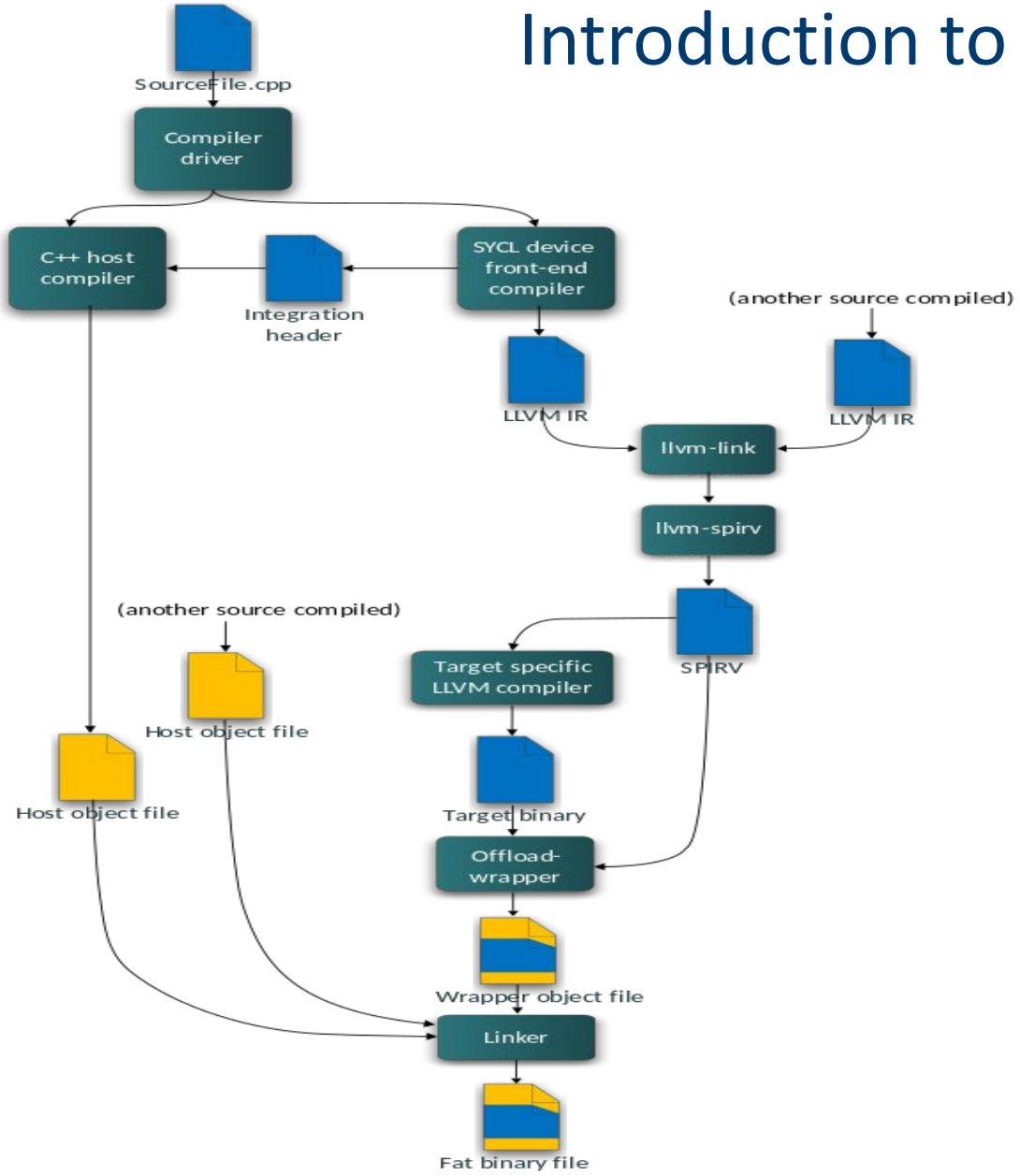
LLVM Optimizing Compiler



- Create a new directory outside the LLVM source directory for your build

```
cd directory-for-build
cmake path-to-llvm-sources
cmake --build .
```

Introduction to Intel's LLVM Compiler



Introduction to LLVM Compiler Backend (Clang)

Compiler Commands for SYCL code:

- Compiling with icx/icpx compiler is follows:

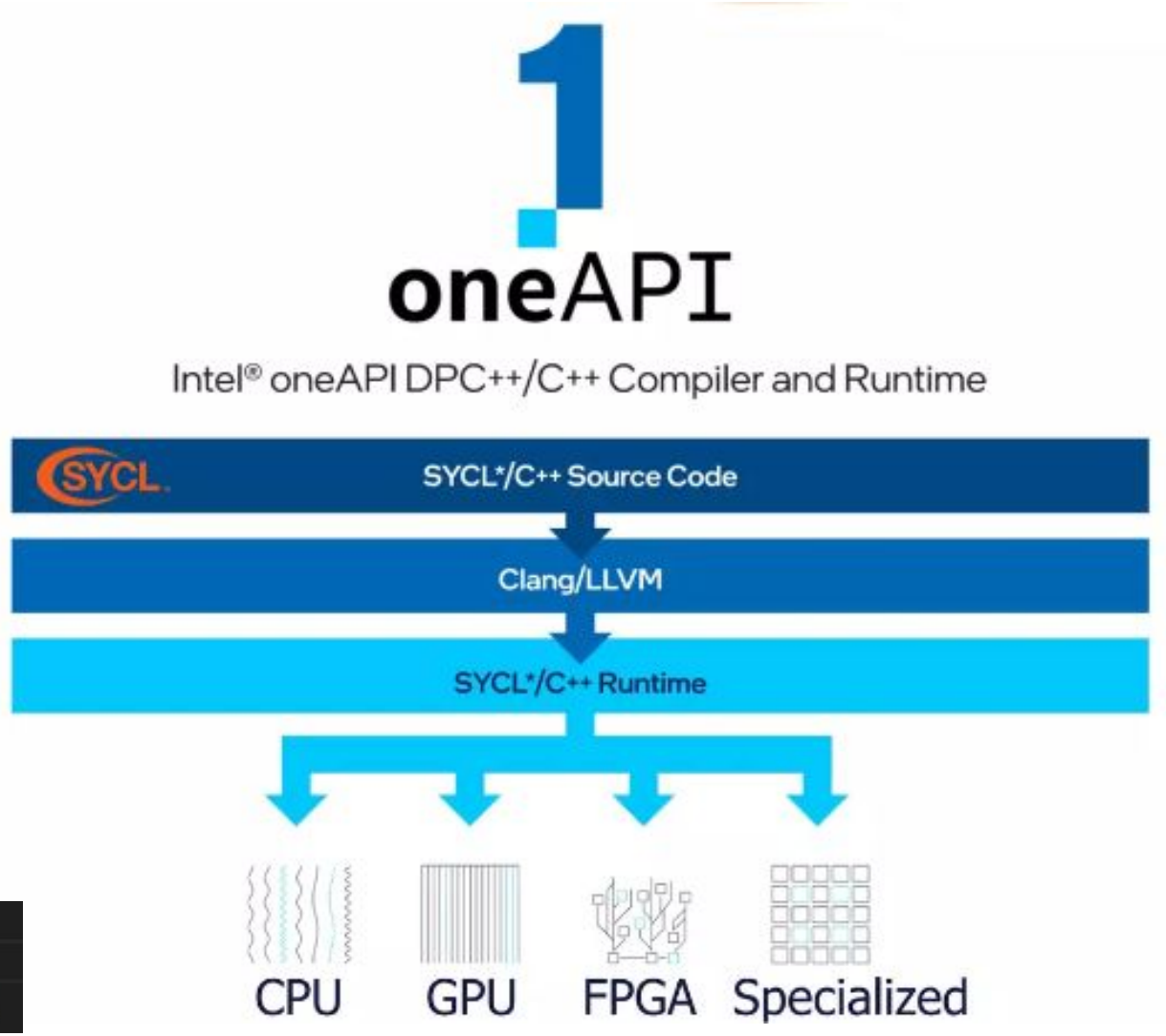
```
icpx -fsycl block_load_store.cpp -o block_load_store.o
```

- For spv (spirv) builds the command is:

```
icpx -fsycl-device-only -fno-sycl-use-bitcode block_reduce.cpp
-o block_reduce.spv
```

- For ptx device code generation, command is:

```
icpx -fsycl-device-only
-Xsycl-target-backend=nvptx64-nvidia-cuda
-emit-llvm  -c sycl_llvm_kernel.cpp -o sycl_llvm_kernel_ptx.ll
```



Introduction to LLVM Compiler Backend (Clang)

Compiler Commands for SYCL code:

- To use clang++ compiler, command is:

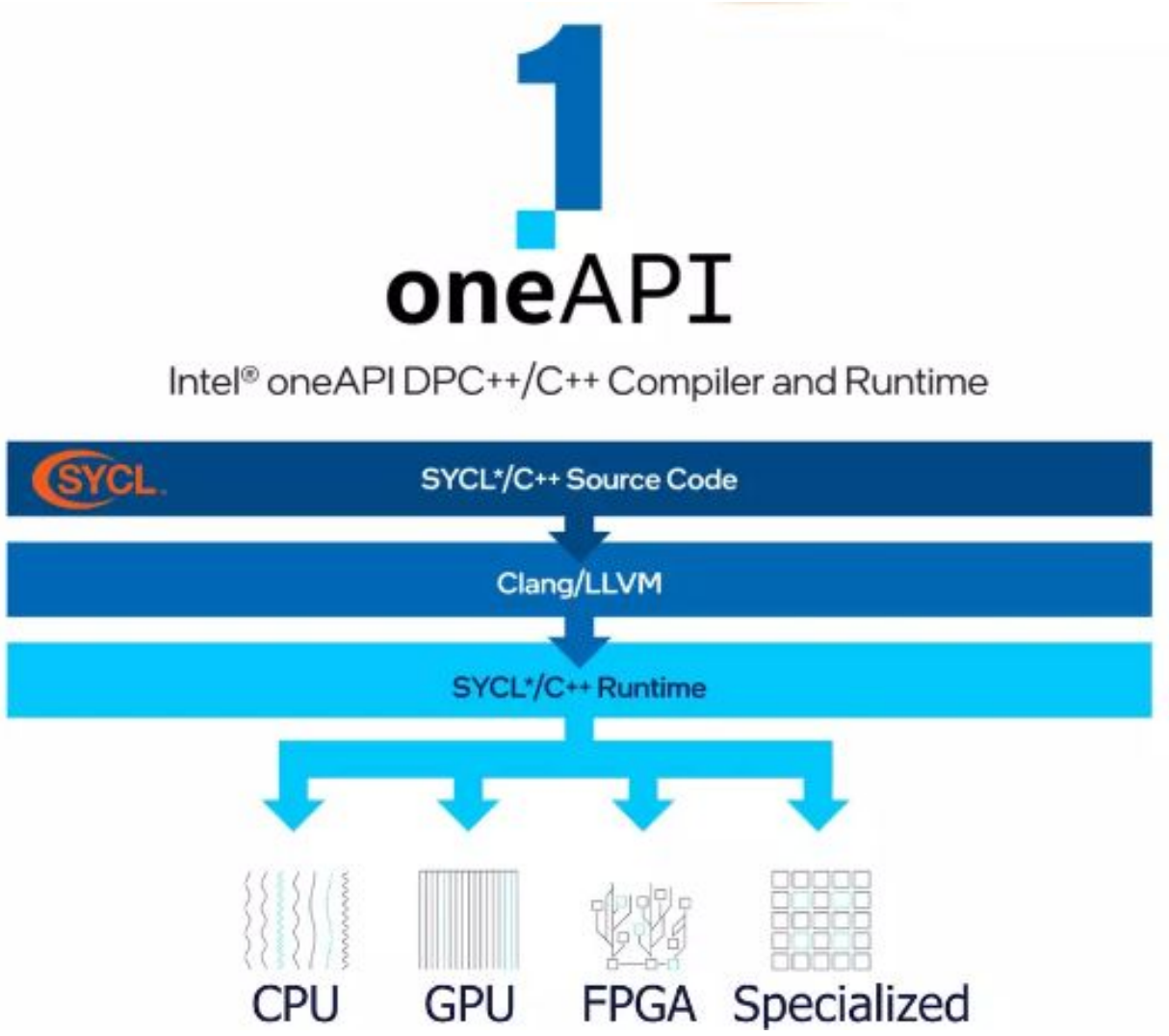
```
clang++ -fsycl-device-only -std=c++17
-fno-sycl-use-bitcode block_reduce.cpp
-o block_reduce.spv
```

- For disassembling bc code, we can do:

```
llvm-dis block_reduce.bc
```

- Alternate way to read ll code from codegen

```
clang++ -std=c++17 -fsycl -fsycl-device-only
-emit-llvm -S -c
block_reduce.cpp -o block_reduce.ll
```



Introduction to LLVM Compiler Backend (Clang)

LLVM : ICX/ICPX Compiler AOT

- AOT is useful feature which requires SYCL with L0 backend for device segregation

```
icpx -fsycl -fsycl-targets=spir64_x86_64 -Xsycl-target-backend  "-march=avx2"
main.cpp
```

- No additional compilation time is done when running your application.
- No just-in-time (JIT) bugs encountered due to compilation for the target. Any bugs should be found during AOT and resolved.
- Your final code, executing on the target device, can be tested as-is before you deliver it to end-users.
  - A program built with AOT compilation for specific target device(s) will not run on different device(s). You must detect the proper target device at runtime and report an error if the targeted device is not present. The use of exception handling with an asynchronous exception handler is recommended.

- `-fsycl-targets=spir64_x86_64`
- `-Xsycl-target-backend "-march=<arch>"`, where *<arch>* is one of the following:

Switch	Display Name
avx	Intel® Advanced Vector Extensions (Intel® AVX)
avx2	Intel® Advanced Vector Extensions 2 (Intel® AVX2)
avx512	Intel® Advanced Vector Extensions 512 (Intel® AVX-512)
sse4.2	Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2)

Introduction to LLVM Compiler Backend (Clang)

LLVM : ICX/ICPX Compiler AOT (contd)

- For linking SYCL kernel codes with non kernel codes for AOT, we can compile separately both of them and link them using a target device backend.

```
icpx -c main.cpp      // This creates the host object that is used below (no kernel code)
icpx -c -fsycl-targets=spir64_x86_64 -Xsycl-target-backend "-march=mavx2" block_reduce.cpp // kernel code
icpx -fsycl-targets=spir64_x86_64 -Xsycl-target-backend "-march=mavx2" block_reduce.o main.o //link kernel code
```

Introduction to SYCL

SYCL and DPC++

A complete program

Single source

Host code and heterogeneous accelerator kernels can be mixed in same source files

Familiar C++

Library constructs add functionality, such as:

Construct	Purpose
queue	Work targeting
Pointers and Buffers	Data management
parallel_for	Parallelism

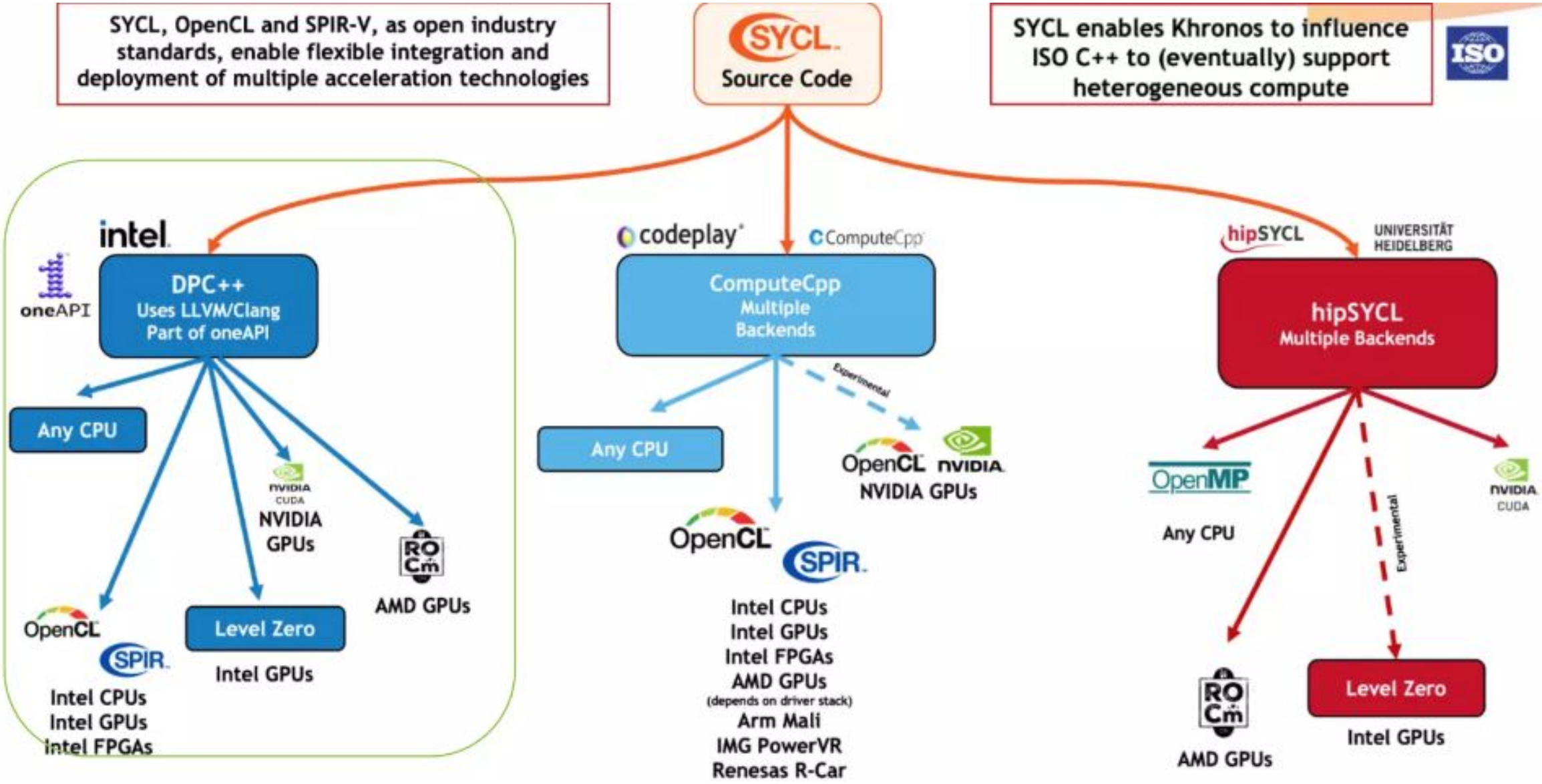
Host code

Accelerator device code

Host code

```
#include <sycl/sycl.hpp>
constexpr int num=16;
using namespace sycl;

int main() {
  queue Q; // use default device

  auto R = range<1>{ num };
  buffer<int> A{ R };
  int *B = malloc_shared<int>(R, Q);

  Q.submit([&](handler& h) {
    accessor out{A, h};
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = B[idx]; });
  });

  host_accessor result{A, read_only};
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "\n";
  return 0;
}
```

Introduction to SYCL



Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming

- **Unified Shared Memory (USM)** enables code with pointers to work naturally without buffers or accessors
- **Parallel reductions** add a built-in reduction operation to avoid boilerplate code and achieve maximum performance on hardware with built-in reduction operation acceleration
- **Work group** and **subgroup** algorithms add efficient parallel operations between work items
- **Class template argument deduction (CTAD)** and template deduction guides simplify class template instantiation
- Simplified use of **Accessors** with a built-in reduction operation reduces boilerplate code and streamlines the use of C++ software design patterns
- Expanded interoperability enables efficient acceleration by diverse backend **acceleration APIs**
- SYCL atomic operations are now more closely aligned to **standard C++ atomics** to enhance parallel programming freedom

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Memory Management

Memory Management

Two views of memory

- Unified shared memory (USM)
  - Pointer-based programming
  - **Flexibility:** Scopes of sharing. Automatic or manual movement

```
int* shared_array = malloc_shared<int>(N, Q);

for (int i = 0; i < N; i++)
  shared_array[i] = i;

Q.submit([&](handler& h) {
  h.parallel_for(N, [=](id<1> i) {
    shared_array[i] = shared_array[i] + 2;
  });
});
Q.wait();
```

- Buffers
  - Flexible abstraction for dense arrays of data
  - Accessors inform the DPC++ RT about usage
  - Data movement automatic

```
auto R = range<1>{ num };
buffer<int> A{ R };

Q.submit([&](handler& h) {
    accessor out{A, h}; // read-write
    h.parallel_for(R, [=](id<1> idx) {
      out[idx]++; });
  });
```

Introduction to SYCL - USM

UNIFIED SHARED MEMORY – WHEN TO USE IT

Buffers are powerful and elegant

- Use if the abstraction applies cleanly in your application, and/or buffers aren't disruptive to your development

USM provides a familiar pointer-based C++ interface

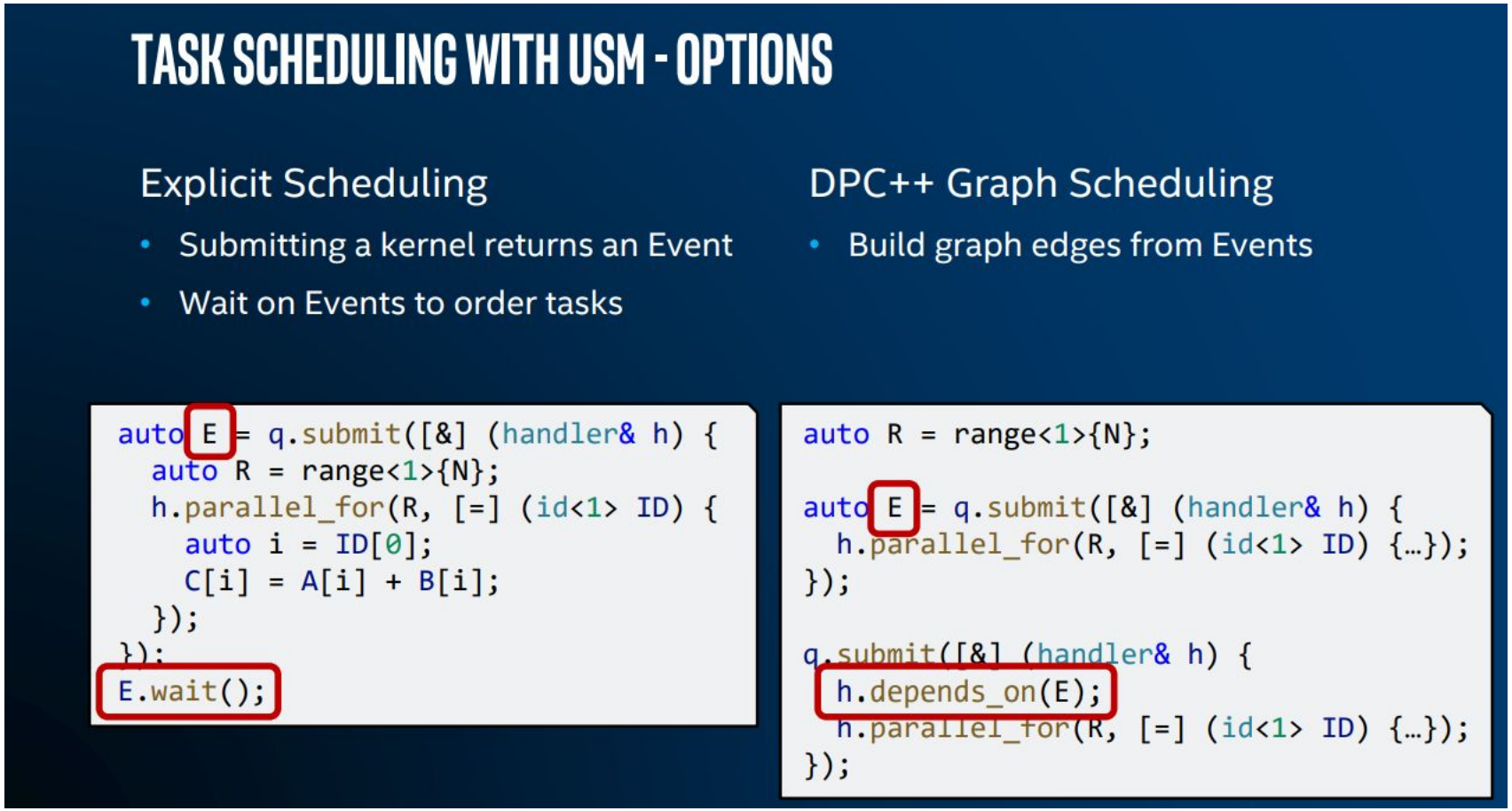
- Useful when porting C++ code to DPC++, by minimizing changes
- Use shared allocations when porting code, to get functional quickly

USM not intended to provide peak performance out of box

- Allows code to become functional quickly
- Use profiler to identify small % of code where you should tune

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – USM buffer Task Queues



Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – USM buffer Task Queues

Work is submitted to devices through queues.
A queue maps to ***one and only one*** device.
Multiple queues can map to the same device.

Queues are out-of-order by default.

- Work may not execute in the order in which it was submitted.
- Work can be ordered through events or accessors.

An in-order queue can be created by passing a property to the constructor.

- In-order queues just do one thing after another – easier to reason about!

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – USM

`aspect::usm_device_allocations`.
`aspect::usm_host_allocations`.

```
namespace sycl {
namespace usm {

enum class alloc : /* unspecified */ {
  host,
  device,
  shared,
  unknown
};

}
}
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Queues

ORDERED QUEUES

DPC++ Queues are Out-of-Order

- Allows expressing complex DAGs

Linear task chains are common

- DAGs unnecessary here and add verbosity

Simple things should be simple to express

- In-order semantics express the linear task pattern easily

```
// Without Ordered Queues
queue q;
auto R = range<1>{N};

auto E = q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

auto F = q.submit([&] (handler& h) {
  h.depends_on(E);
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.depends_on(F);
  h.parallel_for(R, [=] (id<1> ID) {…});
});
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Queues

ORDERED QUEUES

DPC++ Queues are Out-of-Order

- Allows expressing complex DAGs

Linear task chains are common

- DAGs unnecessary here and add verbosity

Simple things should be simple to express

- In-order semantics express the linear task pattern easily

```
// With Ordered Queues
ordered_queue q;
auto R = range<1>{N};

q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});
```

Introduction to SYCL

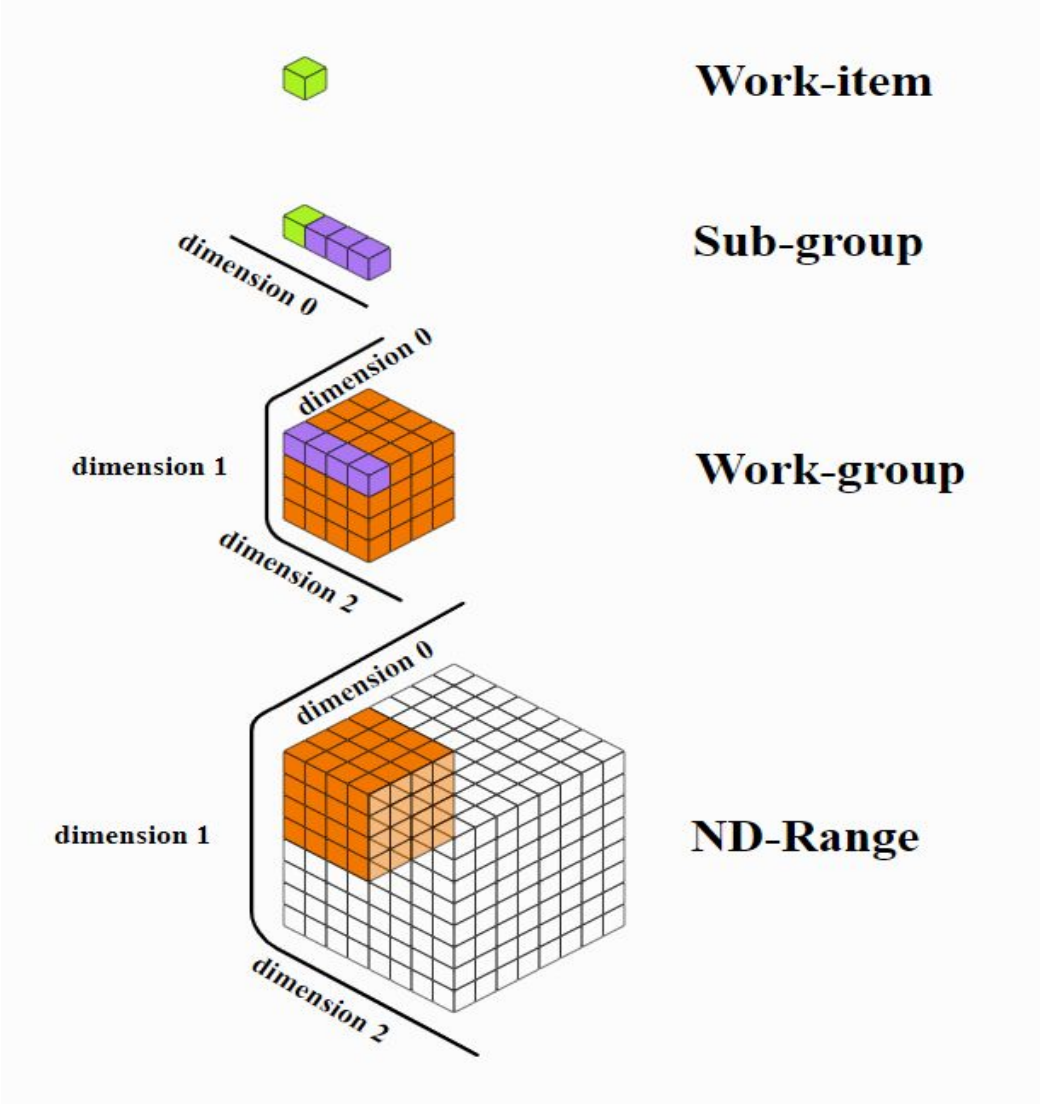
SYCL : Heterogeneous Parallel Device Programming – Queues in SYCL Kernels

```
// Launch an asynchronous kernel to initialize a
myQueue.submit([&](handler& cgh) {
  // The kernel writes a, so get a write accessor on it
  accessor A { a, cgh, write_only };

  // Enqueue a parallel kernel iterating on a N*M 2D iteration space
  cgh.parallel_for(range<2> { N, M },
                   [=](id<2> index) { A[index] = index[0] * 2 + index[1]; });
});
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Work and Sub Groups

- The index space of an ND-Range kernel is divided into work-groups, sub-groups, and work-items. A work-item is the basic unit. A collection of work-items form a sub-group, and a collection of sub-groups form a work-group.
- All the work-groups run concurrently but may be scheduled to run at different times depending on availability of resources. Work-group execution may or or may not be preempted depending on the capabilities of underlying hardware.
- A sub-group is a collection of contiguous work-items in the global index space that execute in the same



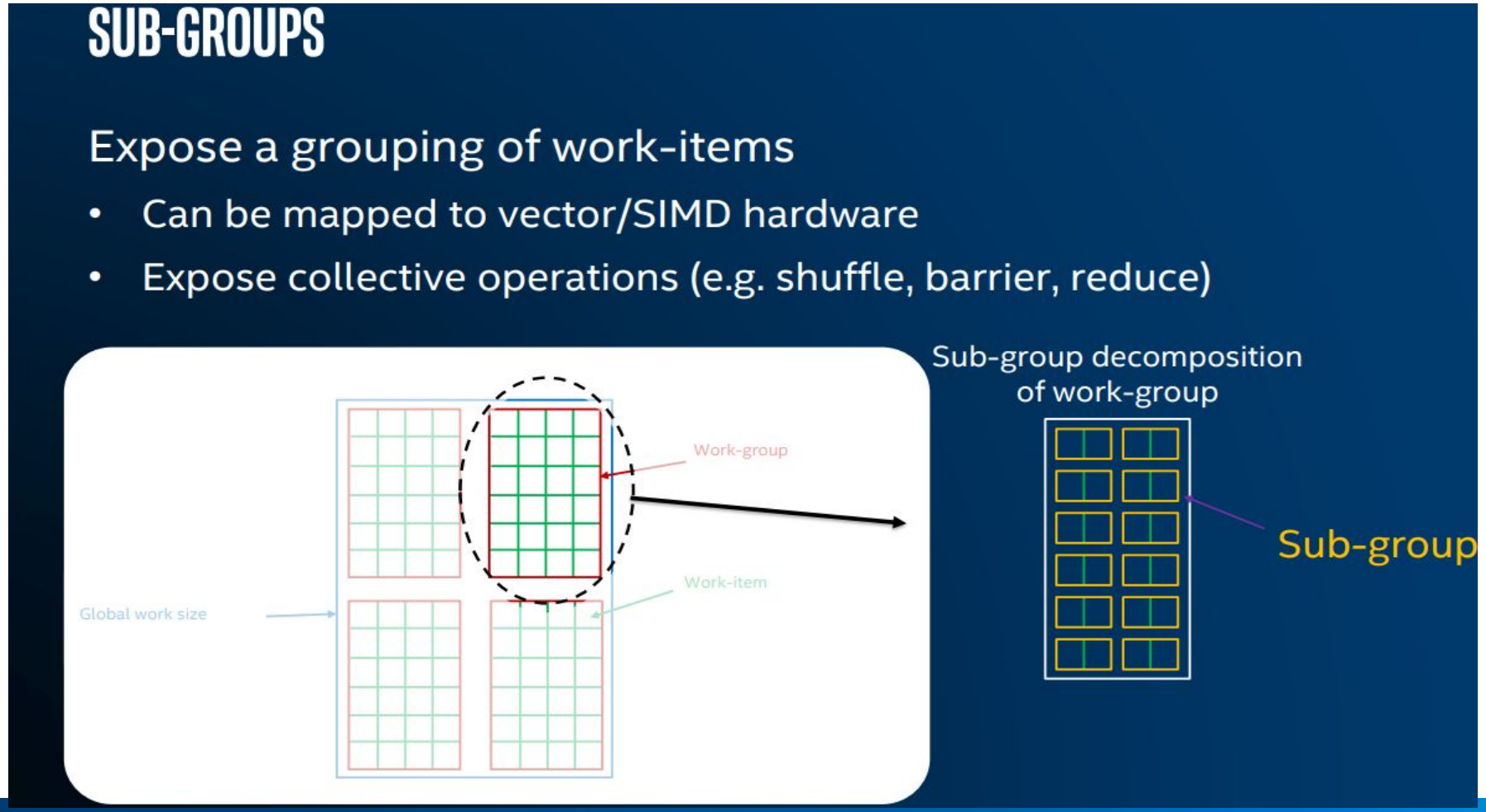
Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Memory Model

- Global-memory is accessible to all work-items in all work-groups. Work-items can read from or write to any element of a global memory object. Reads and writes to global memory may be cached depending on the capabilities of the device. Global memory is persistent across kernel invocations. Concurrent access to a location in an USM allocation by two or more executing kernels where at least one kernel modifies that location is a data race; there is no guarantee of correct results unless mem-fence and atomic operations are used.
- Local-memory is accessible to all work-items in a single work-group. Attempting to access local memory in one work-group from another work-group results in undefined behavior. This memory region can be used to allocate variables that are shared by all work-items in a work-group. Work-group-level visibility allows local memory to be implemented as dedicated regions of the device memory where this is appropriate.
- Private-memory is a region of memory private to a work-item. Attempting to access private memory in one work-item from another work-item results in undefined behavior.
- Generic-memory is a virtual address space which overlaps the global, local and private address spaces. Therefore, an object that resides in the global, local, or private address space can also be accessed through the generic address space

Introduction to SYCL

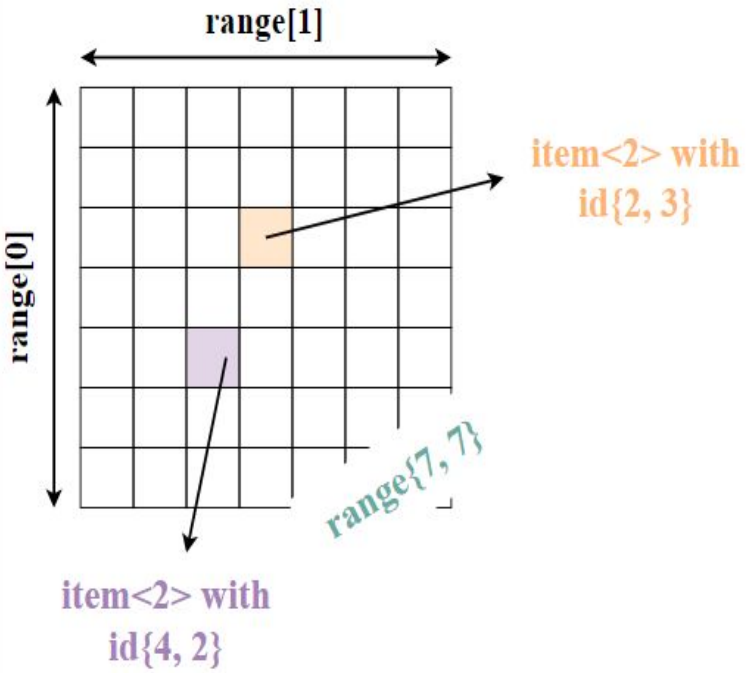
SYCL : Heterogeneous Parallel Device Programming – Work and Sub Groups



Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – ND Ranges

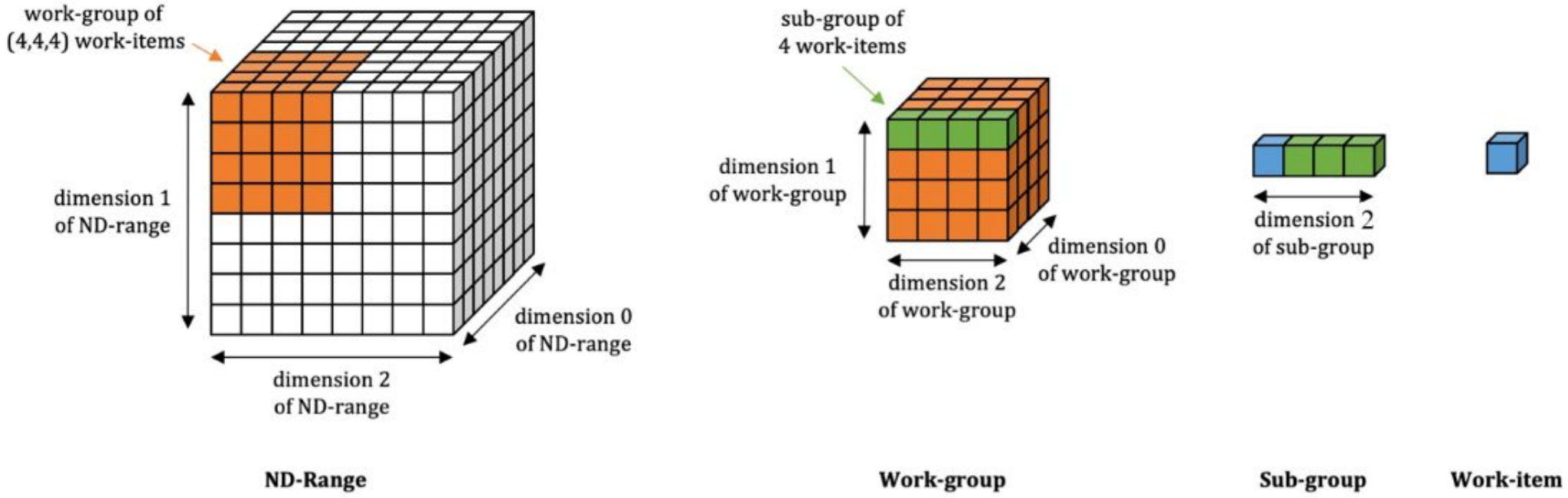
```
Q.submit([&](handler &cgh) {
  accessor acc { buf, cgh, write_only };

  cgh.parallel_for(range<2> { n_work_items }, [=](id<2> idx) {
    acc[idx] = 42.0;
  });
});
```



A `range<2>` object, representing a 2-dimensional execution range. Each element in the range is of type `item<2>` and is indexed by an object of type `id<2>`. Items are instances of the kernel. An N-dimensional range is in row-major order: dimension $N - 1$ is contiguous. Figure adapted from [RAB+21].

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Work and Sub Groups



> *Work group size = Threads x SIMD sub-group size*

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Memory Scope

```
namespace sycl {

enum class memory_scope : /* unspecified */ {
  work_item,
  sub_group,
  work_group,
  device,
  system
};

inline constexpr auto memory_scope_work_item = memory_scope::work_item;
inline constexpr auto memory_scope_sub_group = memory_scope::sub_group;
inline constexpr auto memory_scope_work_group = memory_scope::work_group;
inline constexpr auto memory_scope_device = memory_scope::device;
inline constexpr auto memory_scope_system = memory_scope::system;

} // namespace sycl
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Task & Heirarchical Data Parallelism

TP

HP

Sync

```
q.submit([&](handler &cgh) {
  cgh.parallel_for_work_group(
      range<1>{N / 2}, range<1>{2}, [=](group<1> myGroup) {
        auto j = myGroup.get_id(0);
        myGroup.parallel_for_work_item(
            [&](h_item<1> it) { A[(j * 2) + it.get_local_id(0)]++; });
      });
});

q.wait();
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Task & Heirarchical Data Parallelism

- get_ group_id: Id of workgroup
- get_local_id : id of work item in a group
- Get_local_range : dimension of work item
- get_group_range: number of subgroups in a workgroup
- get_max_local_range: maximum number of work items permitted in a workgroup
- get_group_linear_id: same as `get_group_id()[0]`.
- get_local_linear_id: same as `get_local_id()[0]`.
- get_group_linear_range: same as `get_group_range()[0]`.
- get_local_linear_range: same as `get_local_range()[0]`.
- leader: return leader of the work group

```
id<1> get_group_id() const;

id<1> get_local_id() const;

range<1> get_local_range() const;

range<1> get_group_range() const;

range<1> get_max_local_range() const;

uint32_t get_group_linear_id() const;

uint32_t get_local_linear_id() const;

uint32_t get_group_linear_range() const;

uint32_t get_local_linear_range() const;

bool leader() const;
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Kernel Definitions

Kernels as Function Objects

A kernel can be defined as a named function object type. These function objects provide the same functionality as any C++ function object, with the restriction that they need to follow SYCL rules to be device copyable. The operator() member function mus be const-qualified, and it may take different parameters depending on the data accesses defined for the specific kernel. If the operator() function writes to any of the member variables, the behavior is undefined.

```
class RandomFiller {
 public:
  RandomFiller(accessor<int> ptr)
      : ptr_ { ptr } {
    std::random_device hwRand;
    std::uniform_int_distribution<> r { 1, 100 };
    randomNum_ = r(hwRand);
  }
  void operator()(item<1> item) const { ptr_[item.get_id()] = get_random(); }
  int get_random() { return randomNum_; }

 private:
  accessor<int> ptr_;
  int randomNum_;
};

void workFunction(buffer<int, 1>& b, queue& q, const range<1> r) {
  myQueue.submit([&](handler& cgh) {
    accessor ptr { buf, cgh };
    RandomFiller filler { ptr };
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Kernel Definitions

Kernels as Lambdas

```
myQueue.submit([&](handler& h) {
  // Explicitly name kernel with previously forward declared type
  h.single_task<MyKernel>([=] {
    // [kernel code]
  });

  // Explicitly name kernel without forward declaring type at
  // namespace scope.  Must still be forward declarable at
  // namespace scope, even if not declared at that scope
  h.single_task<class MyOtherKernel>([=] {
    // [kernel code]
  });
});
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Kernel Definitions

Type trait – device copyable

```
namespace sycl {
    template<typename T>
    struct is_device_copyable;

    template<typename T>
    inline constexpr bool is_device_copyable_v = is_device_copyable<T>::value;
};
```

is_device_copyable

- is_device_copyable must meet the Cpp17UnaryTrait requirements.
- If is_device_copyable is specialized such that is_device_copyable_v<T> == true on a T that does not satisfy all the requirements of a device copyable type, the results are unspecified.

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Kernel Definitions Parameter Rules

- The following SYCL types are legal parameter types:
  - `accessor` when templated with `target::device`;
  - `accessor` when templated with any of the deprecated parameters: `target::global_buffer`, `target::constant_buffer`, or `target::local`;
  - `local_accessor`;
  - `unsampled_image_accessor` when templated with `image_target::device`;
  - `sampled_image_accessor` when templated with `image_target::device`;
  - `stream`;
  - `id`;
  - `range`;
  - `marray<T, NumElements>` when `T` is device copyable;
  - `vec<T, NumElements>`.

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Kernel Definitions

Type trait – group functions

is_group<T>

```
namespace sycl {
template <class T> struct is_group;

template <class T> inline constexpr bool is_group_v = is_group<T>::value;
} // namespace sycl
```

•Use in barriers, broadcast , group algorithms (any_of, all_of, shift_left, shift_right, permute, reduce) etc. Ex: group _ barrier

```
template <typename Group>
void group_barrier(Group g,
                   memory_scope fence_scope = Group::fence_scope); // (1)
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Kernel Definitions

Address Space – Pointer Class

1. Accessors : Using multi_ptr

```
auto *d_A = dacc_A.get_multi_ptr<sycl::access::decorated::yes>().get();
```

2. Explicit pointer class: global_ptr

3. Generic Address Space: SYCL_EXTERNAL

```
SYCL_EXTERNAL void Foo();
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – SIMD Work and Sub Groups (CPY) 1024*1024 across 16 work items

```
constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
  h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                 [=](sycl::nd_item<1> it) {
                   int i = it.get_global_linear_id();
                   sycl::ext::oneapi::sub_group sg = it.get_sub_group();
                   int sgSize = sg.get_local_range()[0];
                   i = (i / sgSize) * sgSize * 16 + (i % sgSize);
                   for (int j = 0; j < sgSize * 16; j += sgSize) {
                     data[i + j] = data2[i + j];
                   }
                 });
});
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Exception Handling

queue::wait_and_throw()
queue::throw_asynchronous()
event::wait_and_throw()

```
void catch_invalid_errors(sycl::context const& ctx) {
  try {
    do_something_to_invoke_error(ctx);
  } catch (sycl::exception const& e) {
    if (e.code() == sycl::errc::invalid) {
      std::cerr << "Invalid error: " << e.what();
    } else {
      throw;
    }
  }
}
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Synchronization - Device Event

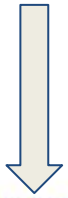
device_event (asynchronous) ⟹

```
namespace sycl {
class device_event {

device_event(__unspecified__);

 public:
  void wait() noexcept;
};
} // namespace sycl
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – Synchronization - Atomic ref

atomic_ref (asynchronous) provides ability to perform atomic operations in device code. Requires a Memory Ordering



```
// Partial specialization for integral types
template <memory_order DefaultOrder, memory_scope DefaultScope,
          access::address_space AddressSpace = access::address_space::generic_space>
class atomic_ref<Integral, DefaultOrder, DefaultScope, AddressSpace> {
```

Introduction to SYCL

Custom Kernels : SYCL Metaprogramming Methods

- SYCL allows us to easily leverage the power of C++ template metaprogramming in device code. Most valid compile-time constructs will work in a kernel. This means that we can define generic command groups, kernels, use functional programming concepts, and offload a lot of work to the compiler.

- . This is a sample case of a generic executable adder kernel which performs addition through operator overloading ().

- We can initiate a main and create a single buffer of inputs.It can be noticed that since our function object is now a full-fledged class, we do not have to make up an artificial template parameter, since the kernel name is known - it is exactly the name of the function object type

```
template<typename T, typename Acc, size_t N>
  class ConstantAdder {
  public:
    ConstantAdder(Acc accessor, T val)
      : accessor(accessor)
      , val(val) {}

    void operator() () {
      for (size_t i = 0; i < N; i++) {
         accessor[i] += val;
      }
    }

  private:
    Acc accessor;
    const T val;
  };
```

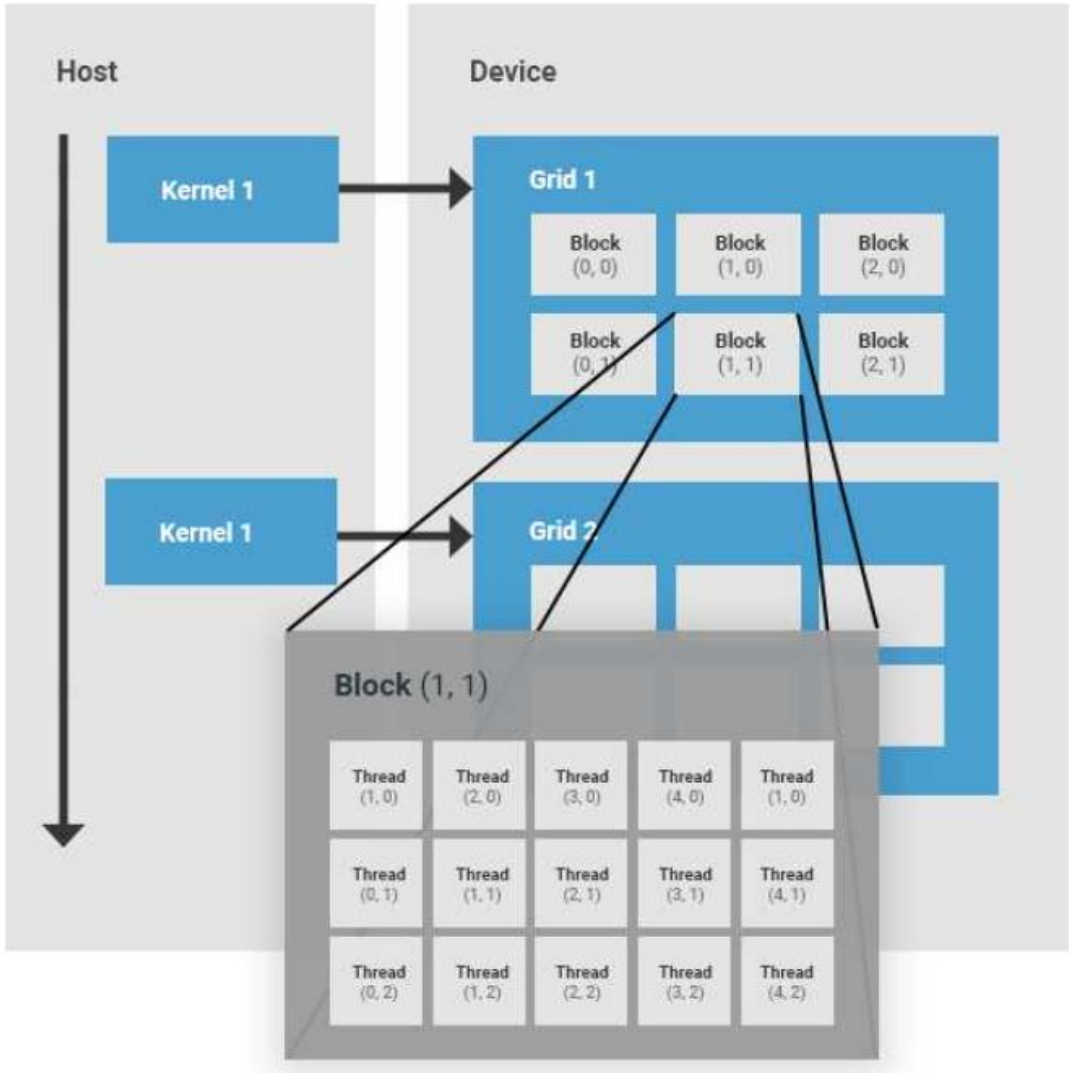
Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – CUDA SYCL

The CUDA thread hierarchy is composed of a grid of thread blocks.

•***Thread block*** : A thread block is a set of concurrently executing threads that reside on the same SM; share the resources of that SM, and cooperate among themselves using different hardware mechanisms. Each thread block has a block ID within its grid. A thread block can be one, two, or three dimensional.

•***Grid*** : A grid is an array of thread blocks launched by a kernel, that read inputs from global memory; write results to global memory, and synchronize dependency among nested kernel calls. A grid will be described by a user and can be one, two, or three dimensional.



Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – CUDA & SYCL memory

CUDA	SYCL
`gridDim.x/y/z`	`sycl::nd_item.get_group_range(2/1/0)`
`blockDim.x/y/z`	`sycl::nd_item.get_local _range().get(2/1/0)`
`blockIdx.x/y/z`	`sycl::nd_item.get_group(2/1/0)`
`threadIdx.x/y/z`	`sycl::nd_item.get_local_id(2/1/0)`
`warpSize`	`sycl::nd_item.` `.get_sub_group().get_local_range().get(0)`

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming –
CUDA kernel & SYCL command queues

CUDA	SYCL
`dim3`	`sycl::range<3>`

`Kernel<<<gridDim, blockDim>>>(...)`	1. Member function `parallel_for` of class `sycl::queue`:
	`sycl::queue q;`
	`q.parallel_for(sycl::nd_range<3>`
	`(girdDim * blockDim, blockDim),`
	`[=](sycl::nd_item<3> item){`
	`  kernel(...);`
	`});`

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – CUDA SYCL

•The DPC++ frontend pushes the SYCL code down several passes and then calls the PTX backend from LLVM to generate the PTX for the kernels in the SYCL application.

•The generated PTX ISA is usually comparable with the native NVCC compiler when using the same optimization flags.

•The other components are the runtime plugins, which enable the SYCL runtime to call native APIs on Nvidia and AMD platforms (CUDA Driver and HIP respectively). The runtime plugins are dynamic libraries that are called from the SYCL runtime when available. The DPC++ compiler automatically links against the SYCL runtime from the oneAPI distribution, and then if the Nvidia and/or AMD plugins are available, they can be selected at runtime for execution.

```
auto CUDASelector = [](sycl::device const &dev) {
  if (dev.get_platform().get_backend() == sycl::backend::ext_oneapi_cuda) {
    std::cout << " CUDA device found " << std::endl;
    return 1;
  } else {
    return -1;
  }
};
sycl::queue myQueue{CUDASelector};
```

Introduction to SYCL

SYCL : Heterogeneous Parallel Device Programming – CUDA SYCL

•The compiler driver patches enable the DPC++ frontend to build for Nvidia GPUs by identifying the target triple, and then triggering actions to build the device image using the existing CUDA compiler support from the LLVM project.

```
clang++ -fsycl -fsycl-targets=amdgcn-amd-amdhsa,nvptx64-nvidia-cuda,spir64 \
        -Xsycl-target-backend=amdgcn-amd-amdhsa --offload-arch=gfx1030 \
        -Xsycl-target-backend=nvptx64-nvidia-cuda --offload-arch=sm_80 \
        -o sycl-app sycl-app.cpp
```

Introduction to SYCL

SYCL : ParallelSTL
Intel oneDPL

Parallel API is an implementation of the C++ standard libraries algorithms and execution policies, as specified in the ISO/IEC 14882:2017 standard (commonly called C++17).

```
#include <iostream>
#include <vector>
#include


int main()
{
  std::vector<int> data {2, 2, 4, 1, 1};
  auto maxloc = max_element(


data.cbegin(),

data.cend());

  std::cout << "Maximum at element "
    << distance(data.cbegin(),
                maxloc)
    << std::endl;
}
```

```
#include <iostream>
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>

int main()
{
  std::vector<int> data{2, 2, 4, 1, 1};
  auto maxloc = oneapi::dpl::max_element(
          oneapi::dpl::execution::dpcpp_default,
          data.cbegin(),
          data.cend());

  std::cout << "Maximum at element "
       << oneapi::dpl::distance(data.cbegin(),

maxloc)
       << std::endl;
}
```

Introduction to SYCL

SYCL : ParallelSTL SYCL and ISO C++ differences

1. oneDPL execution policies only result in parallel execution if random access iterators are provided, the execution will remain serial for other iterator types.

2. Function objects passed in to algorithms executed with device policies must provide const-qualified operator(). The SYCL specification states that writing to such an object during a SYCL kernel is undefined behavior.

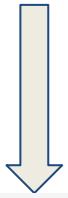
3. For the following algorithms, par_unseq and unseq policies do not result in vectorized execution: includes, inplace_merge, merge, set_difference, set_intersection, set_symmetric_difference, set_union, stable_partition, unique.

4. The following algorithms require additional O(n) memory space for parallel execution: copy_if, inplace_merge, partial_sort, partial_sort_copy, partition_copy, remove, remove_if, rotate, sort, stable_sort, unique, unique_copy.

Introduction to SYCL

SYCL : ParallelSTL execution policy vs std::execution::

Links with C++
std::execution::parallel_unsequenced_policy
Implies parallel execution on SIMD safe execution



```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <vector>

int main()
{
    std::vector<int> data( 1000 );
    std::fill(oneapi::dpl::execution::par_unseq, data.begin(), data.end(), 42);
    return 0;
}
```

Introduction to SYCL

SYCL : ParallelSTL ranges vs std::ranges::

Links with C++
std::ranges::views::all
(Range Adaptors)

```
sycl::buffer<int> buf(data, sycl::range<1>(10));
auto range_1 = iota_view(0, 10) | views::reverse();
auto range_2 = all_view(buf) | views::reverse();
```

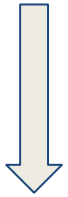
Links with C++
std::ranges::views::reverse
(Range Adaptors)

```
using namespace oneapi::dpl::experimental::ranges;

{
    sycl::buffer<int> A(data, sycl::range<1>(max_n));
    sycl::buffer<int> B(data2, sycl::range<1>(max_n));

    auto view = all_view(A) | views::reverse();
    auto range_res = all_view<int, sycl::access::mode::write>(B);

    copy(oneapi::dpl::execution::dpcpp_default, view, range_res);
}
```

Introduction to SYCL

SYCL : ParallelSTL miscellaneous vs std::

Links with C++ **std::iterator**



```
auto sum = std::reduce(dpl::execution::dpcpp_default,
                       count_a, count_b, init); // sum is (0 + 0 + 1 + ... + 9) = 45
```

Links with C++ **std::swap**

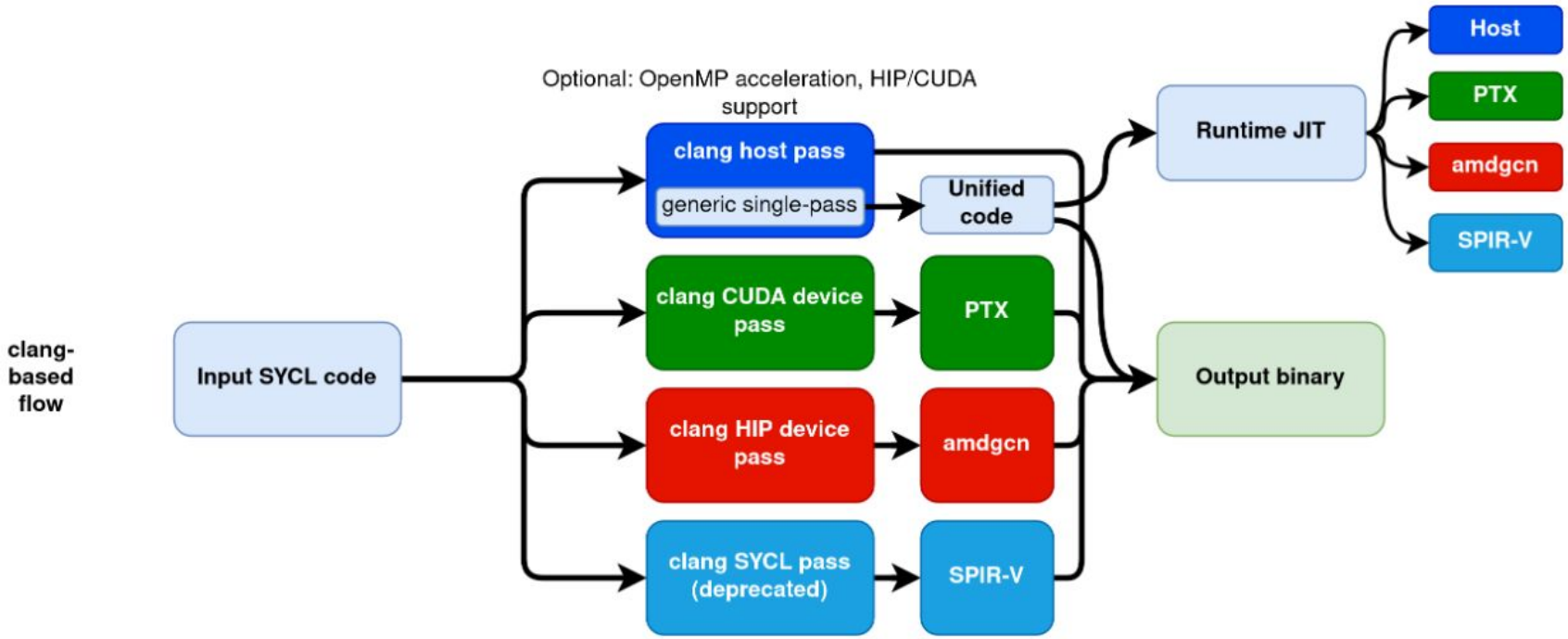
```
deviceQueue.submit([&](sycl::handler &cgh) {
auto swap_accessor = swap_buffer.get_access<sycl_read_write>(cgh)
cgh.single_task<class KernelSwap>([=]() {
    int & num1 = swap_accessor[0];
    int & num2 = swap_accessor[1];
    oneapi::dpl::swap(num1, num2);
    });
});
}
```

Introduction to SYCL

SYCL : Adaptive CPP /hipSYCL

SSCP – single pass compilation `--acpp-targets=generic`

SMCP – Multi pass compilation



Introduction to SYCL

SYCL : Adaptive CPP /hipSYCL (std::par)

Links with C++ **std::transform**

```
auto ret = std::transform(std::execution::par_unseq, data.begin(), data.end(),
                          device_out.begin(), transformation);
auto host_ret = std::transform(data.begin(), data.end(), host_out.begin(),
                               transformation);
```

Links with C++ **std::replace_copy**

```
auto ret = std::replace_copy(std::execution::par_unseq, data.begin(),
                             data.end(), dest.begin(), old_val, new_val);
std::replace_copy(data.begin(), data.end(), host_dest.begin(), old_val,
                  new_val);
```

Links with C++ **std::for_each_n**

```
auto res = std::for_each_n(std::execution::par_unseq, data.begin(),
                           data.size(), [=](auto &x) { x *= 2; });
```

Custom Kernels in SYCL

Custom Kernels : Blocked CUDA Reduce

https://godbolt.org/z/nPfrGGvd4

```
__global__ void SumKernel(int* data) {
  typedef cub::BlockReduce<int, 4> BlockReduce;

  __shared__ typename BlockReduce::TempStorage temp1;

  int threadid = threadIdx.x;

  int input = data[threadid];
  int output = 0;
  output = BlockReduce(temp1).Sum(input);
  data[threadid] = output;
}
```

```
void SumKernel(int* data,
  const sycl::nd_item<3> &item_ct1) {

  int threadid = item_ct1.get_local_id(2);

  int input = data[threadid];
  int output = 0;
  output = sycl::reduce_over_group(item_ct1.get_group(), input, sycl::plus<>());
  data[threadid] = output;
}
```

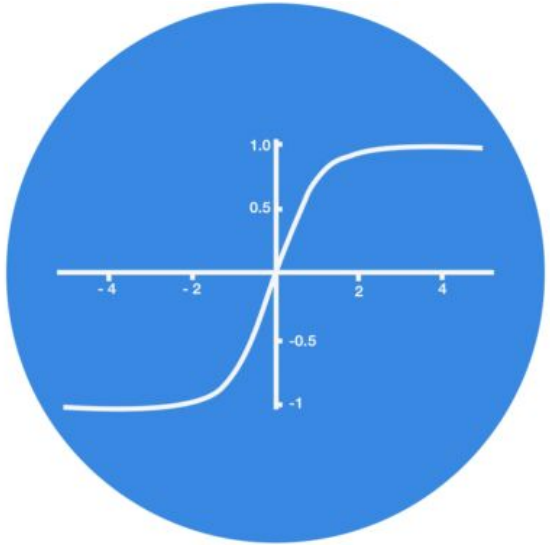
Custom Kernels in SYCL

Custom Kernels : Softmax Activation

The softmax activation function transforms the raw outputs of the neural network into a vector of probabilities, essentially a probability distribution over the input classes.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

5
0.1
-0.5



Custom Kernels in SYCL

Custom Kernels : Softmax Activation Kernel (CUDA)

```
...<input memcpy/malloc>
cudnnSoftmaxForward(handle, CUDNN_SOFTMAX_ACCURATE, CUDNN_SOFTMAX_MODE_CHANNEL,
&alpha, dataTensor, data, &beta, outTensor, out);
cudaMemcpy(host_out.data(), out, ele_num * sizeof(HT), cudaMemcpyDeviceToHost);
alpha = 2.f, beta = 0.f;
cudaDeviceSynchronize();
auto s = cudnnSoftmaxBackward(handle, CUDNN_SOFTMAX_ACCURATE, CUDNN_SOFTMAX_MODE_CHANNEL, &alpha, outTensor, out,
diffoutTensor, diffout, &beta, diffdataTensor, diffdata);
cudaDeviceSynchronize();

cudaMemcpy(host_diffdata.data(), diffdata, ele_num * sizeof(HT), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();

cudnnDestroy(handle);
cudaFree(data);
...<main>
```

Custom Kernels in SYCL

Custom Kernels : Softmax Activation Kernel (SYCL)

```
...<input memcpy>
data = (HT *)sycl::malloc_device(ele_num * sizeof(HT), *stream1);
out = (HT *)sycl::malloc_device(ele_num * sizeof(HT), *stream1);

stream1->memcpy(data, host_data.data(), ele_num * sizeof(HT)).wait();
stream1->memcpy(out, host_out.data(), ele_num * sizeof(HT)).wait();

float alpha = 2.f, beta = 1.5f;

auto s = (handle.async_softmax_forward(dpct::dnnl::softmax_algorithm::normal,
                                       dpct::dnnl::softmax_mode::channel, alpha,
                                       dataTensor, data, beta, outTensor, out),
          0);
dev_ct1.queues_wait_and_throw();
stream1->memcpy(host_out.data(), out, ele_num * sizeof(HT)).wait();
....<main>
```

DPC++ Toolkit

DPC++ Toolkit

WHAT IS DATA PARALLEL C++?

Data Parallel C++

= C++ and SYCL* standard and extensions

Based on modern C++

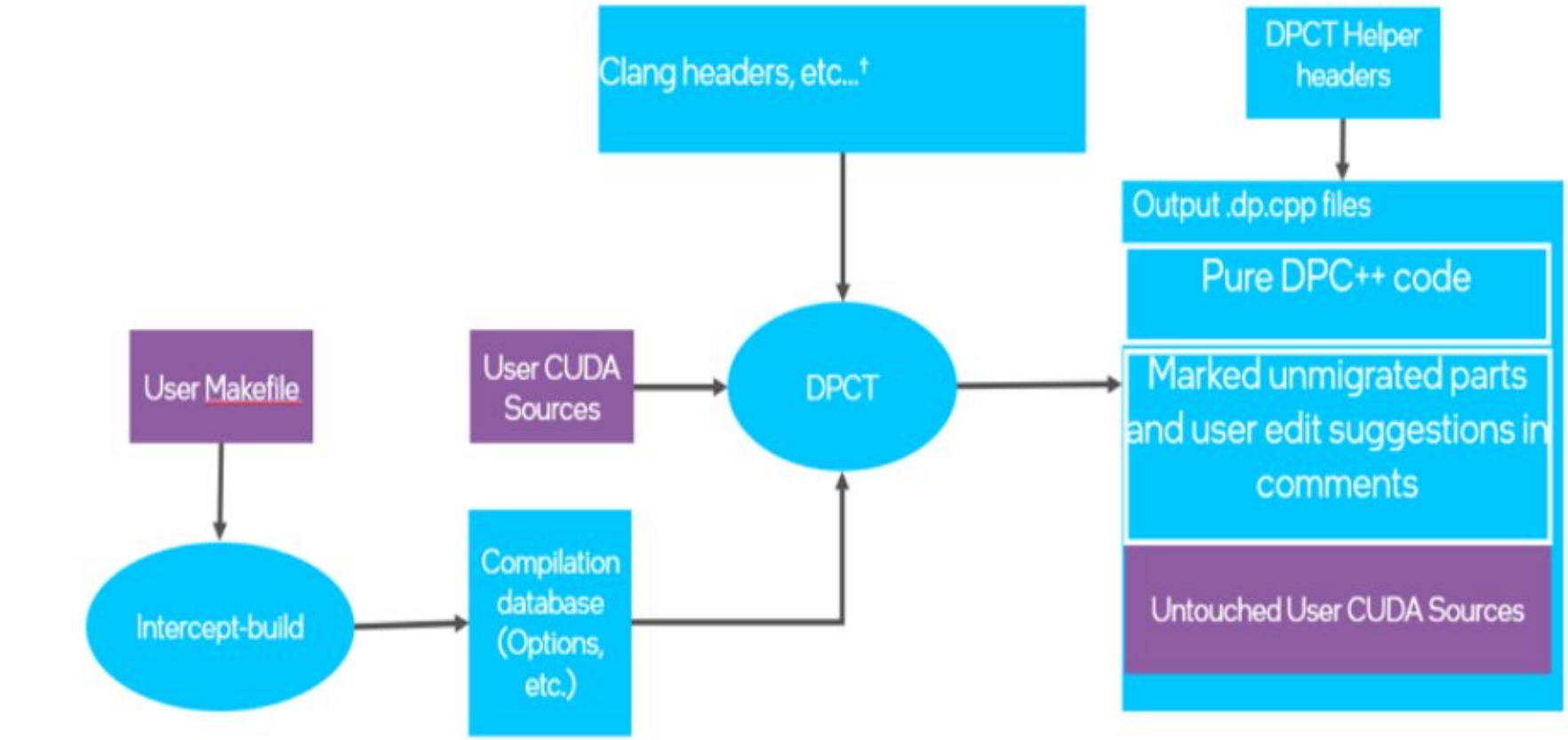
- C++ productivity benefits and familiar constructs

Standards-based, cross-architecture

- Incorporates the SYCL standard for data parallelism and heterogeneous programming

Data Parallel C++ ⇔ DPC++

DPC++ Toolkit



Command line example:

```
intercept-build make
```

Command line example:

```
cd rodinia_3.1/cuda/nw
dpct -p . --in-root=./ --out-root=dpct_output --keep-original-code needle.cu
```

† Certain CUDA header files may need to be available

DPC++ Toolkit

Steps to migrate sample project

- Git clone Syclomatic repository (https://github.com/oneapi-src/SYCLomatic)

```
git clone https://github.com/oneapi-src/SYCLomatic.git
```

- Build SYCLomatic for Linux

```
cd $SYCLOMATIC_HOME
mkdir build
cd build
cmake -G Ninja -DCMAKE_INSTALL_PREFIX=$PATH_TO_C2S_INSTALL_FOLDER
 -DCMAKE_BUILD_TYPE=Release
 -DLLVM_ENABLE_PROJECTS="clang"
 -DLLVM_TARGETS_TO_BUILD="X86;NVPTX" ../SYCLomatic/llvm
ninja install-c2s
```

- Source oneAPI

```
source $PATH_TO_C2S_INSTALL_FOLDER/setvars.sh
```

- Use “c2s” or “dpct” command to migrate cuda project or files (Use “c2s --help” for options). The most common option is :
-

```
dpct --cuda-include-path=<PATH_TO_CUDA_HEADERS>
--in-root=<PATH_TO_YOUR_CUDA_PROJECT>
--out-root=<PATH_FOR_GENERATED_SYCL_CODE>
```

- The source CUDA folders (containing .cuh or .cu) will be migrated to named sycl destination (or “dpct_output” if –out-root is not specified). .cuh is transformed to .dp.hpp and .cu to .dp.cpp
- Incremental migration is enabled by default but is switchable with the “—no-incremental-migration” option.
- Has options to add experimental features to pick up salient headers inside sycl.

DPC++ Toolkit

User defined Migration

- **Default migration rules.** A set of built-in migration rules used by the tool for all migrations.
- **Optional predefined migration rules.** A set of predefined migration rules that can optionally be used for migration. Available predefined migration rules are in the *extensions/opt_rules* folder on the installation path of the tool.
- **User-defined migration rules.** Custom migration rules defined by the user. User-defined migration rules extend the migration capability of Intel® DPC++ Compatibility Tool and can be used to target the migration of specific CUDA syntax to specific SYCL syntax.


```
- Rule: rule_forceinline          # Rule to migrate "__forceinline__" to "inline"
  Kind: Macro                     # Rule type
  Priority: Takeover              # Rule priority
  In: __forceinline__             # Target macro name in the input source code
  Out: inline                     # Name of migrated macro in the output source code
```

```
dpct sample.cu  --rule-file=rule_file1.YAML --rule-file=rule_file2.YAML
```

DPC++ Toolkit

COMPILING A DPC++ PROGRAM

Use the Intel DPC++ compiler!

- dpcpp -fsycl-unnamed-lambda my_source.cpp –o executable

Finding the compiler:

Using Intel's oneAPI Beta

Test code and workloads across a range of Intel® data-centric architectures at

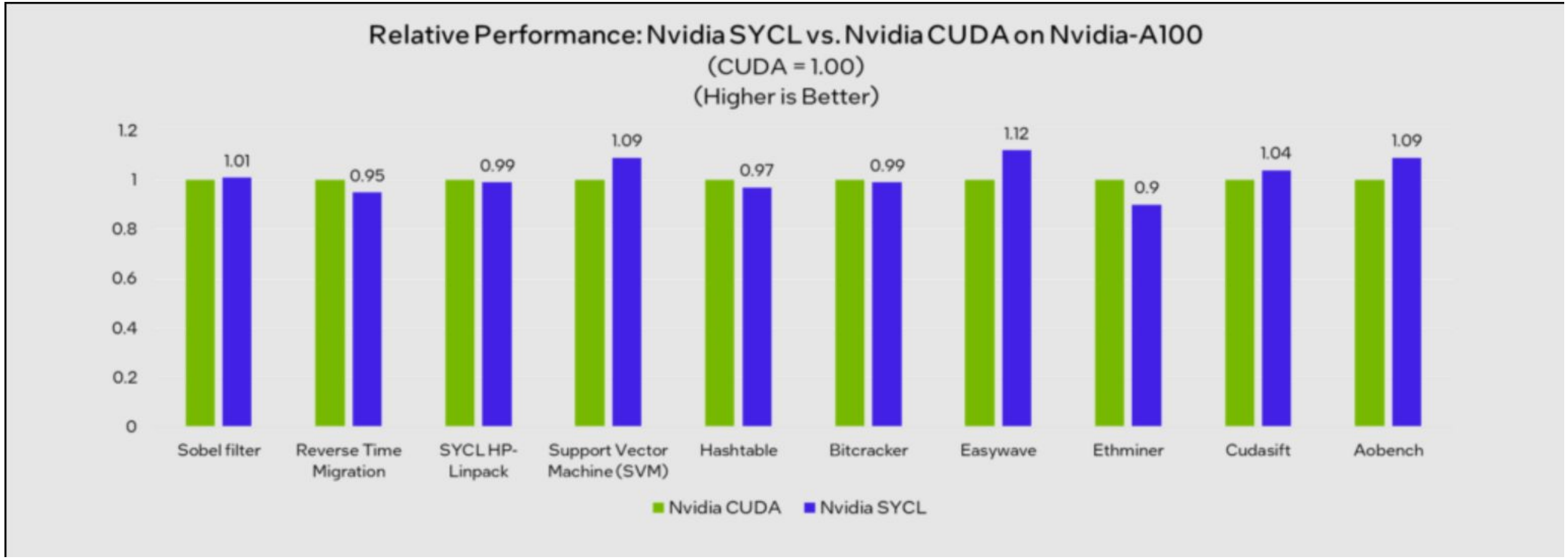
Intel® DevCloud for oneAPI

software.intel.com/devcloud/oneAPI

Learn more and download the **beta toolkits** at

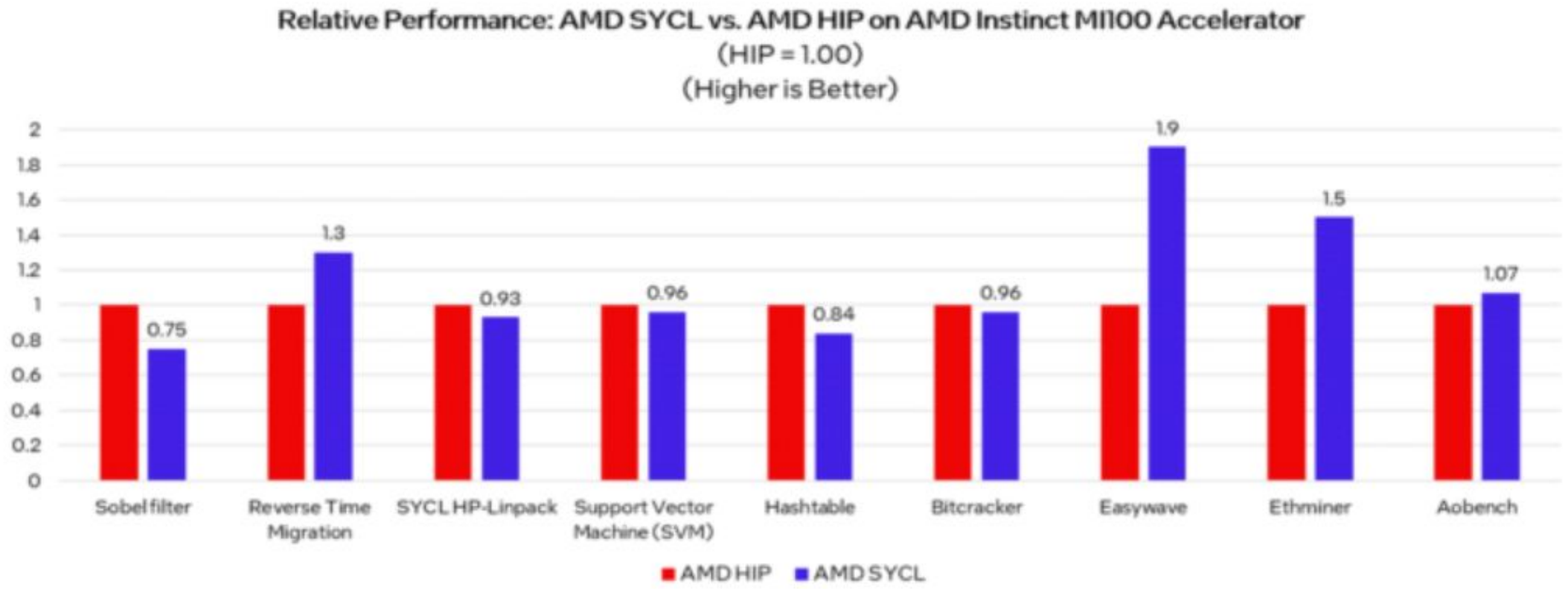
software.intel.com/oneapi

Performance Benchmarking



Performance Benchmarking

Relative Performance: AMD SYCL vs. AMD HIP on AMD GPU



Performance Benchmarking

Llama 3 (Meta) Benchmark on ARC 770

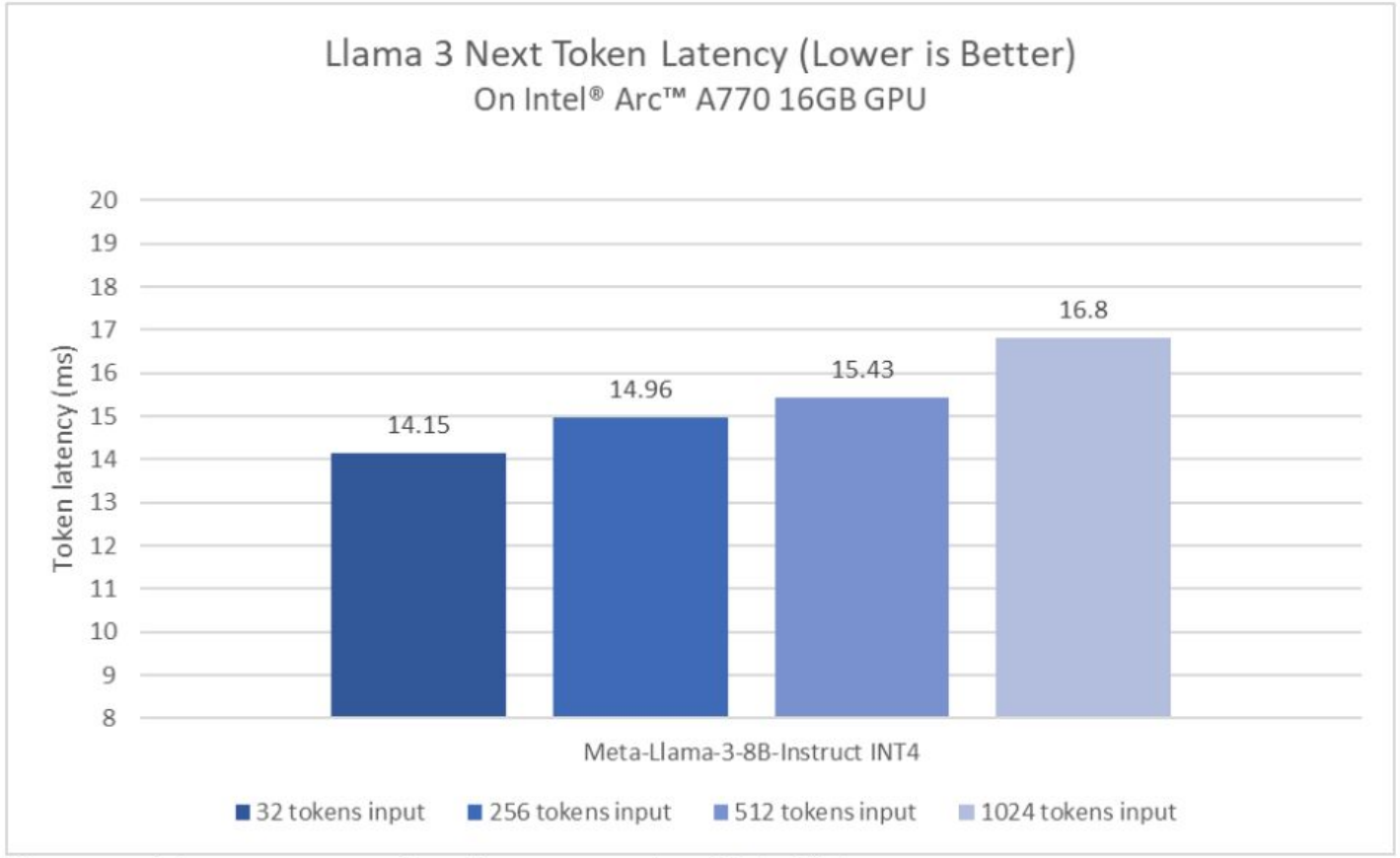


Figure 3. Llama 3 next token latency on Intel® Arc™ A770

References

Code For the Talk: (OS)

- https://github.com/abhilash1910/ISO-CPP-SYCL-Compiler-Conference/ (GH: abhilash1910)
- https://github.com/oneapi-src/oneAPI-samples
- https://oneapi-src.github.io/oneDPL
- https://github.com/AdaptiveCpp/AdaptiveCpp
- https://github.com/intel/llvm
- https://github.com/triSYCL

Some resource for oneAPI/SYCL

- https://intel.github.io/llvm-docs/
- https://www.intel.com/content/www/us/en/docs/oneapi/code-samples-dpcpp/2023-1.html
- https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu
- https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference
- https://www.intel.com/content/www/us/en/developer/articles/technical
- https://www.intel.com/content/www/us/en/developer/articles/technical/transfer-learning-with-tensorflow-on-arc-gpus.html

Conclusion

SYCL and DPC++ provide tools to write high-performance parallel programs using familiar C++ concepts

- Same language and tools to target CPU, GPU, FPGA
- Understanding how SYCL objects fit together helps to avoid bugs
- DPC++ provides flexible methods to select the right device
- Use lower-level tracing and profiling to debug and optimize programs

SYCL : Integrated compiler runtime for accelerated Deep Learning

Abhilash Majumder
AI Frameworks and Compiler Engineer

Abhilash.Majumder@intel.com

GH: https://github.com/abhilash1910

Custom Kernels in SYCL

Custom Kernels : Convolution Kernel (SYCL)

```
...<input memcpy>
q_ct1.memcpy(data, host_data.data(), sizeof(float) * in * ic * ih * iw)
    .wait();
q_ct1.memcpy(out, host_out.data(), sizeof(float) * on * oc * oh * ow)
    .wait();
q_ct1.memcpy(filter, host_filter.data(), sizeof(float) * fk * fc * fh * fw)
    .wait();

dpct::dnnl::convolution_desc covdes;
                covdes.set(0, 0, 1, 1, 1, 1);

size_t size;
void *workspacesize;
size = 0;
workspacesize = (void *)sycl::malloc_device(size, q_ct1);

float alpha = 2.5f, beta = 1.5f;
handle.async_convolution_forward(covdes, dnnl::algorithm::convolution_auto, alpha,
                                 dataTensor, data, filterTensor, filter, beta,
                                 outTensor, out);
dev_ct1.queues_wait_and_throw();
q_ct1.memcpy(host_out.data(), out, sizeof(float) * on * oc * oh * ow)
    .wait();

....<main>
```

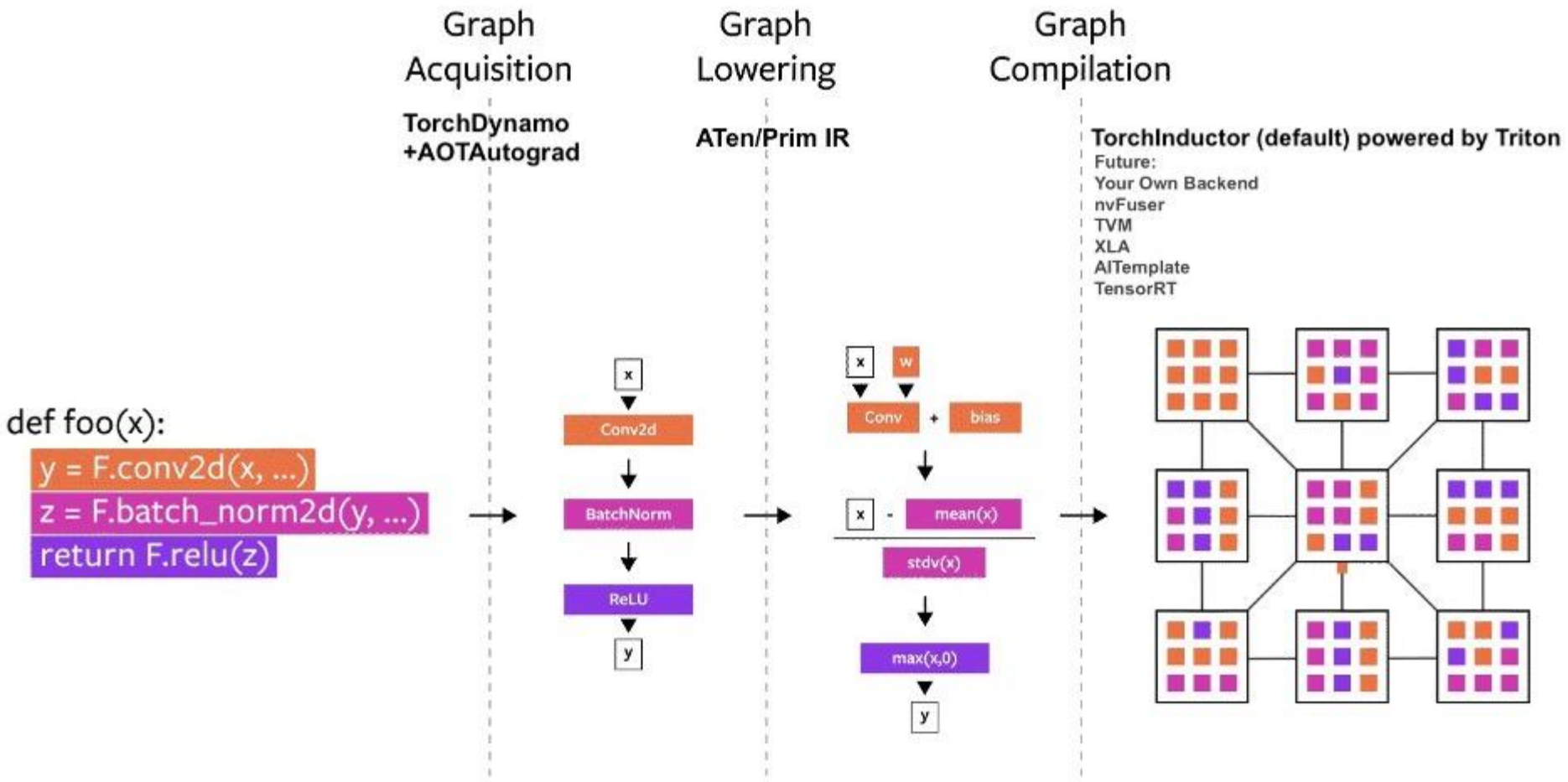
Custom Kernels in SYCL

Custom Kernels : Convolution Kernel (CUDNN)

```
...<input>
cudaMalloc(&workspacesize, size);

int dimo[4];
cudnnGetConvolutionNdForwardOutputDim(covdes, dataTensor, filterTensor, 4, dimo);

float alpha = 1.0f, beta = 0.0f;
cudnnConvolutionForward(handle, &alpha, dataTensor, data, filterTensor, filter, covdes, CUDNN_CONVOLUTION_FWD_ALGO_DIRECT, workspacesize,
size, &beta, outTensor, out);
cudnnConvolutionForward(handle, (void *)&alpha, dataTensor, data, filterTensor, filter, covdes,
CUDNN_CONVOLUTION_FWD_ALGO_DIRECT, workspacesize, size, (void *)&beta, outTensor, out);
cudnnConvolutionForward(handle, (float *)&alpha, dataTensor, data, filterTensor, filter, covdes,
CUDNN_CONVOLUTION_FWD_ALGO_DIRECT, workspacesize, size, (float *)&beta, outTensor, out);

cudaDeviceSynchronize();
cudaMemcpy(host_bias.data(), bias, sizeof(float) * on * oc * oh * ow, cudaMemcpyDeviceToHost);
cudaMemcpy(host_out.data(), out, sizeof(float) * on * oc * oh * ow, cudaMemcpyDeviceToHost);

cudnnDestroy(handle);
cudaFree(data);
...<main>
```

Custom Kernels in SYCL

Pytorch



Custom Kernels in SYCL

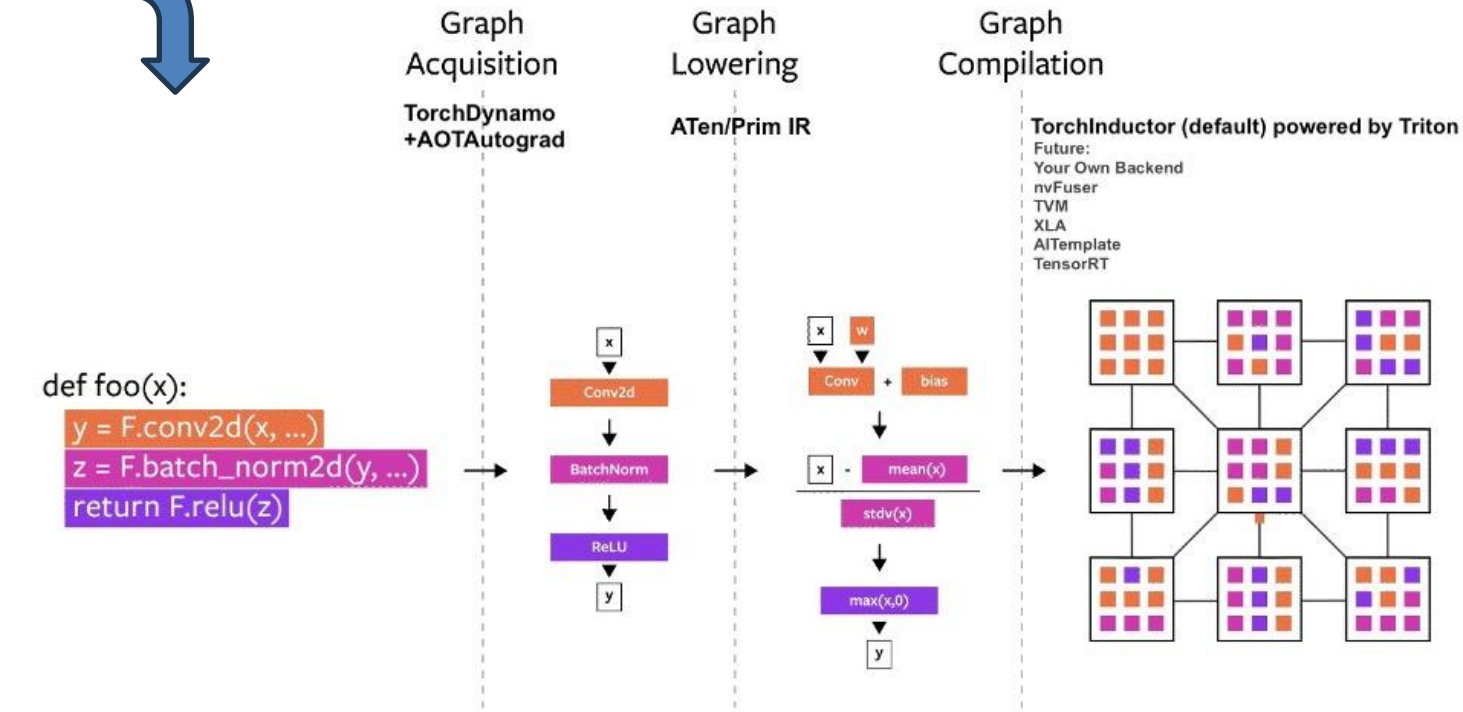
Pytorch

```
cxx_flags = [
        '-fsycl', '-O3', '-std=c++20', '-w', '-fPIC', '-DMKL_ILP64',
        '-Wno-narrowing'
        ]
    extra_ldflags = [
        '-fsycl', '-fPIC', '-Wl,-export-dynamic'

        ]

    include_headers=["kernels/utils.h", "kernels/reduction_utils.h"]

    sycl_source_files = [
                "kernels/sycl_activation_kernels.cpp",
                "kernels/sycl_layernorm_kernel.cpp",
                "kernels/sycl_positional_embedding_kernels.cpp",
                "kernels/sycl_self_attention.cpp"

            ]

    extensions = [
        DPCPPExtension(name="sycl_pytorch",
                       sources=sycl_source_files,
                       include_dirs=include_headers,
                       extra_compile_args={'cxx': cxx_flags},
                       extra_link_args=extra_ldflags)
```

Cmake

```
#include <sycl/sycl.hpp>
constexpr int num=16;
using namespace sycl;

int main() {
  queue Q; // use default device

  auto R = range<1>{ num };
  buffer<int> A{ R };
  int *B = malloc_shared<int>(R, Q);

  Q.submit([&](handler& h) {
    accessor out{A, h};
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = B[idx]; });
  });

  host_accessor result{A, read_only};
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "\n";
  return 0;
}
```

SYCL



Pytorch runtime

Custom Kernels in SYCL

Transformers

Transformers architecture is used in creation of almost all large language models, vision or multimodal models in deep learning. And in majority of the cases, python is used to implement the logic of transformers.

It consists of 4 fundamental layers:

- Positional Encoding Embeddings
- Self Attention (Multi head self attention)
- Layer Normalization
- Feed Forward Layers with Activation (NL)

We will design the transformer architecture using SYCL kernels for each individual component and link with pytorch runtime for ease of frontend usability. (https://github.com/abhilash1910/ISO-CPP-SYCL-Compiler-Conference/tree/main/sycl_with_pytorch)