oneAPI

oneAPI and Data Parallel C++

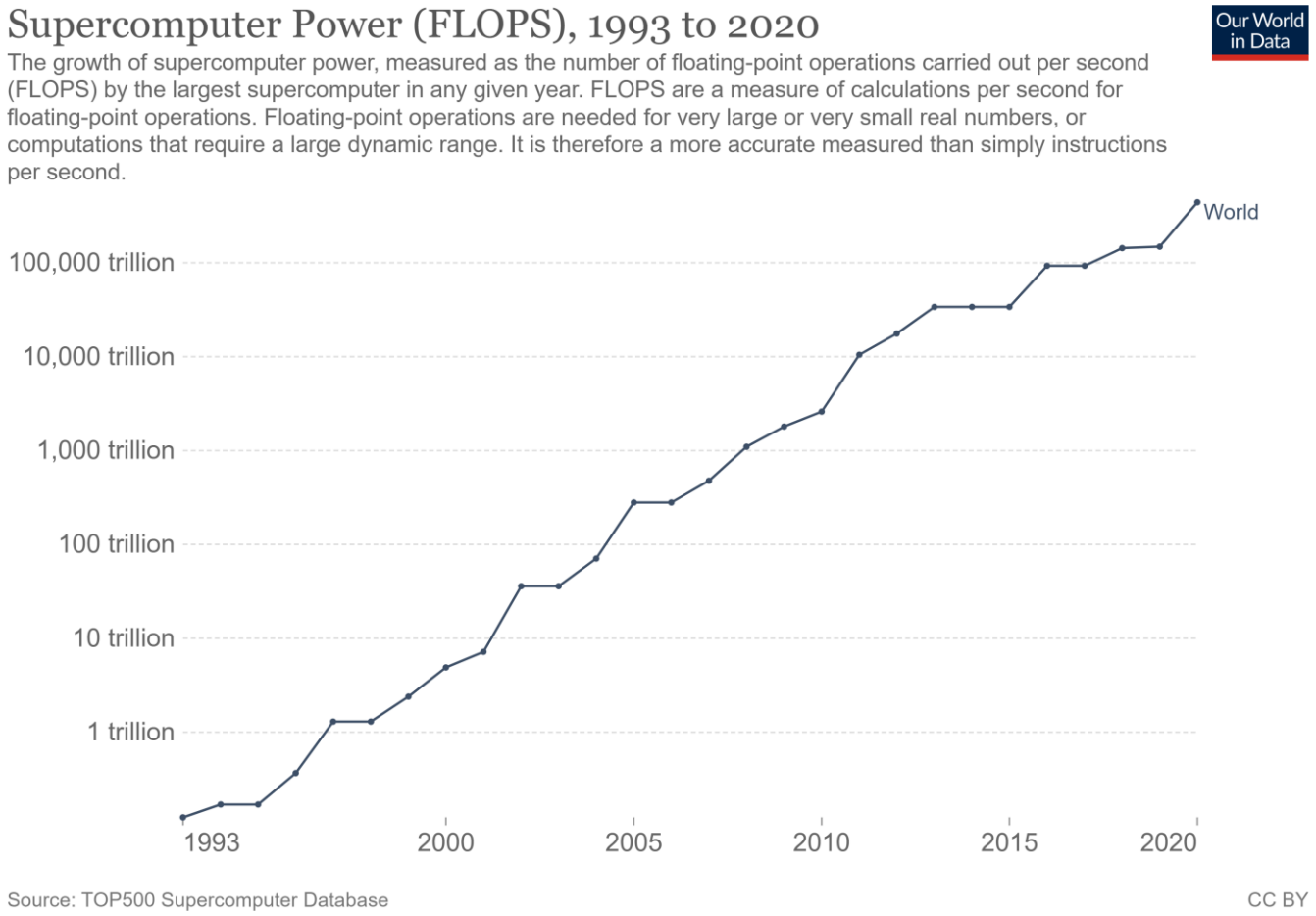
August 10, 2021

oneAPI Overview

Why oneAPI?

Heterogeneity is already here.

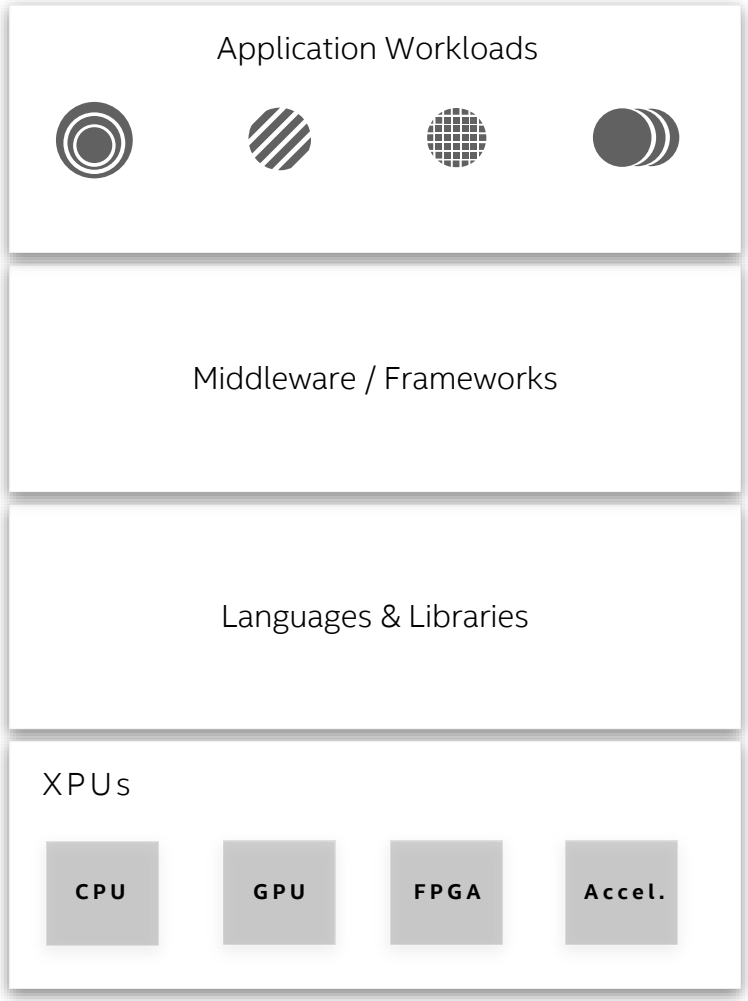
- Over half of the Top 10 in the TOP500 are heterogeneous machines.
- Domain specific accelerators on the rise
  - Energy efficiency a design concern



SPECIALIZED WORKLOADS REQUIRE SPECIALIZED HARDWARE

oneAPI

Growth in specialized workloads



No common programming language or APIs

Inconsistent tool support across platforms

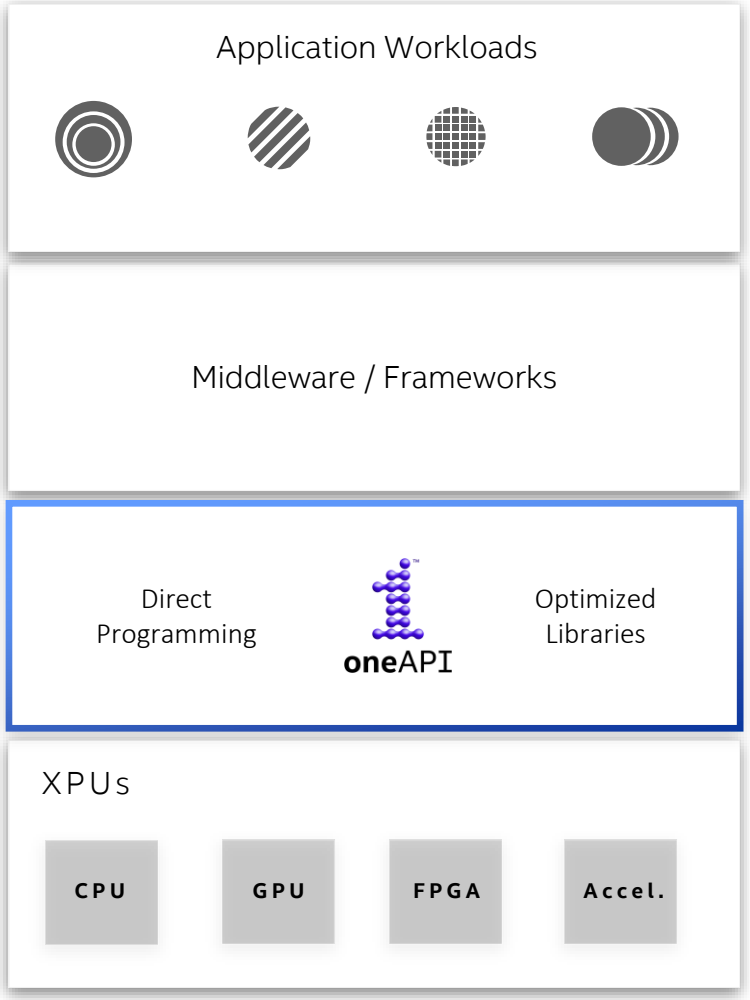
Each platform requires unique software investment

Diverse set of data-centric hardware required

oneAPI

Unified programming model to simplify development across diverse architectures

- Unified and simplified language and libraries for expressing parallelism
- Uncompromised performance
- Based on industry standards and open specifications
- Interoperable with existing programming models



oneAPI

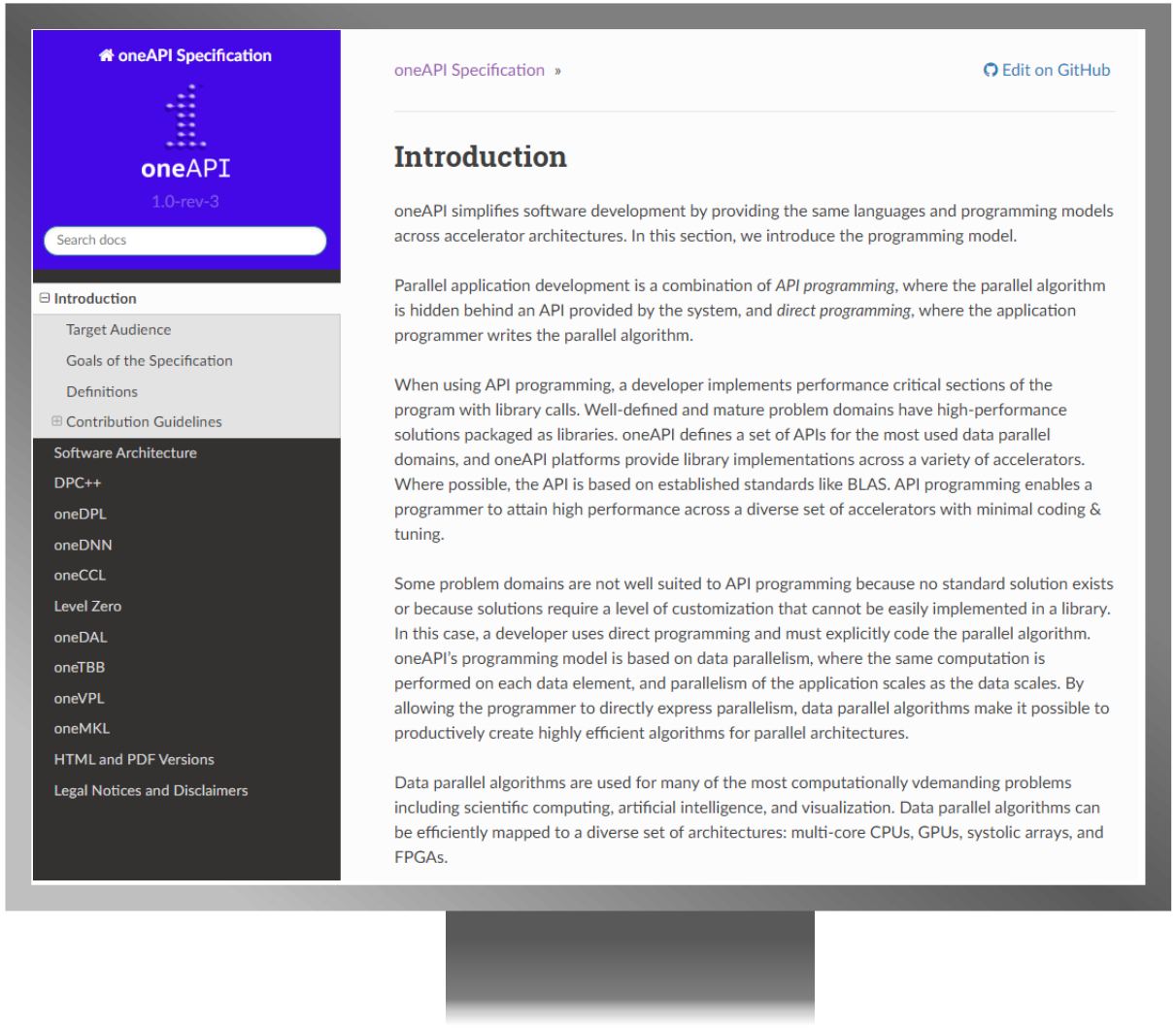
INDUSTRY SPEC

Rev. 1.1 provisional available now @ oneAPI.com

- License: Creative Commons

Drive the XPU programming standard through:

- Open call for community input
- Technical Advisory Boards comprised of 30+ leading industry experts



oneAPI

OPEN-SOURCE IMPLEMENTATIONS

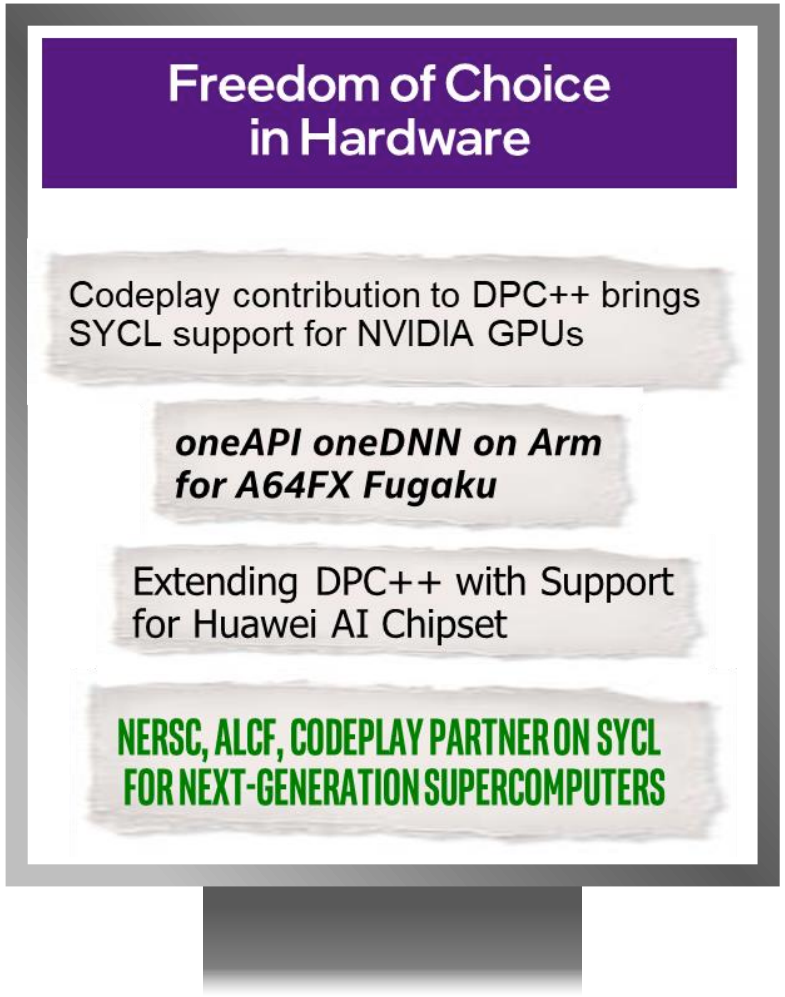
Establish a base oneAPI software stack to use immediately

- All 9 elements have open-source projects

Contributions follow well-known open-source methods & governance (GitHub)

Allows for port to new platforms with minimum uplift

- Already have contributions targeting Intel XPUs, NVIDIA GPUs, AMD GPUs, ARM CPUs, etc.



DPC++ Overview

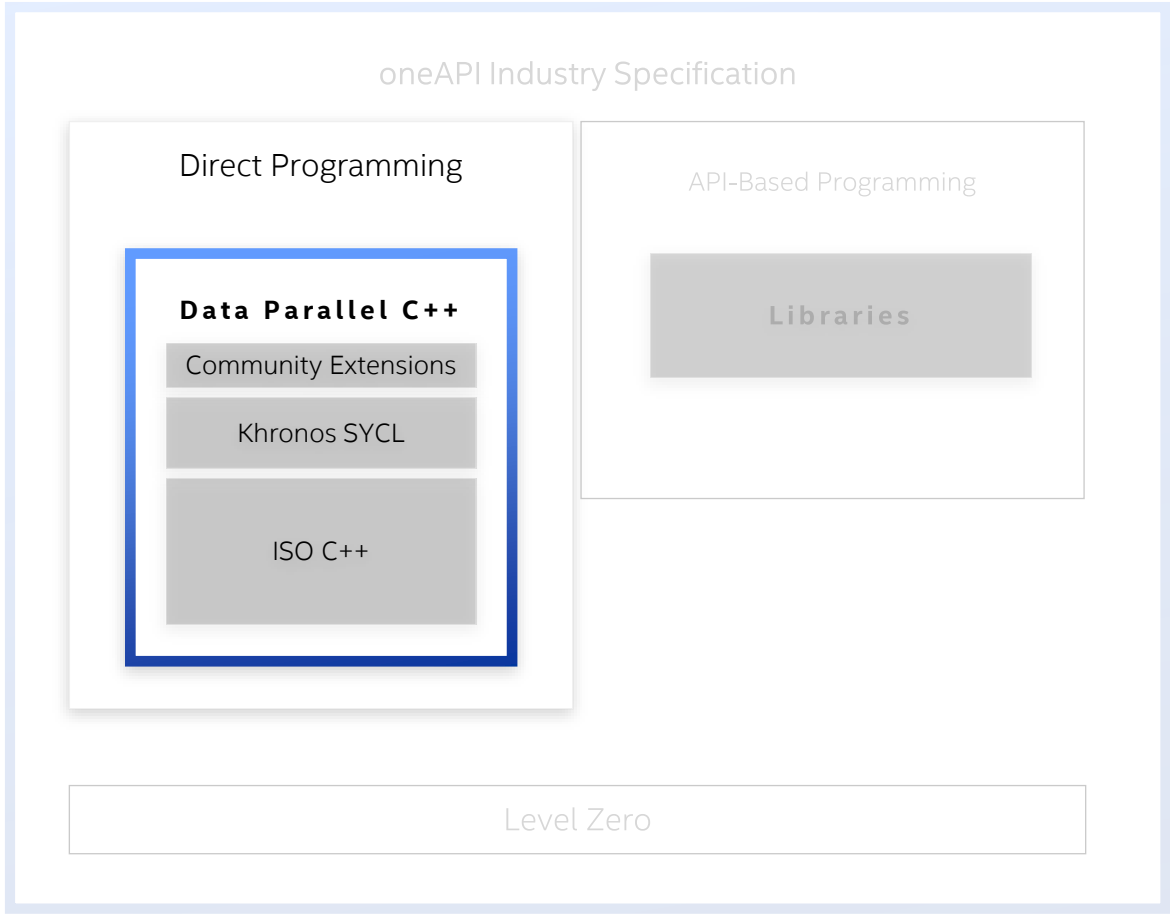
DATA PARALLEL C++

Standards-based, Cross-architecture Language

Language to deliver parallel programming productivity and uncompromised performance across CPUs and XPUs

- Based on modern C++
- Incorporates SYCL to support data parallelism and heterogeneous programming
- Includes open extensions to simplify data parallel programming

DPC++ = ISO C++ and Khronos* SYCL* and Extensions



DATA PARALLEL C++

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/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;
}
```

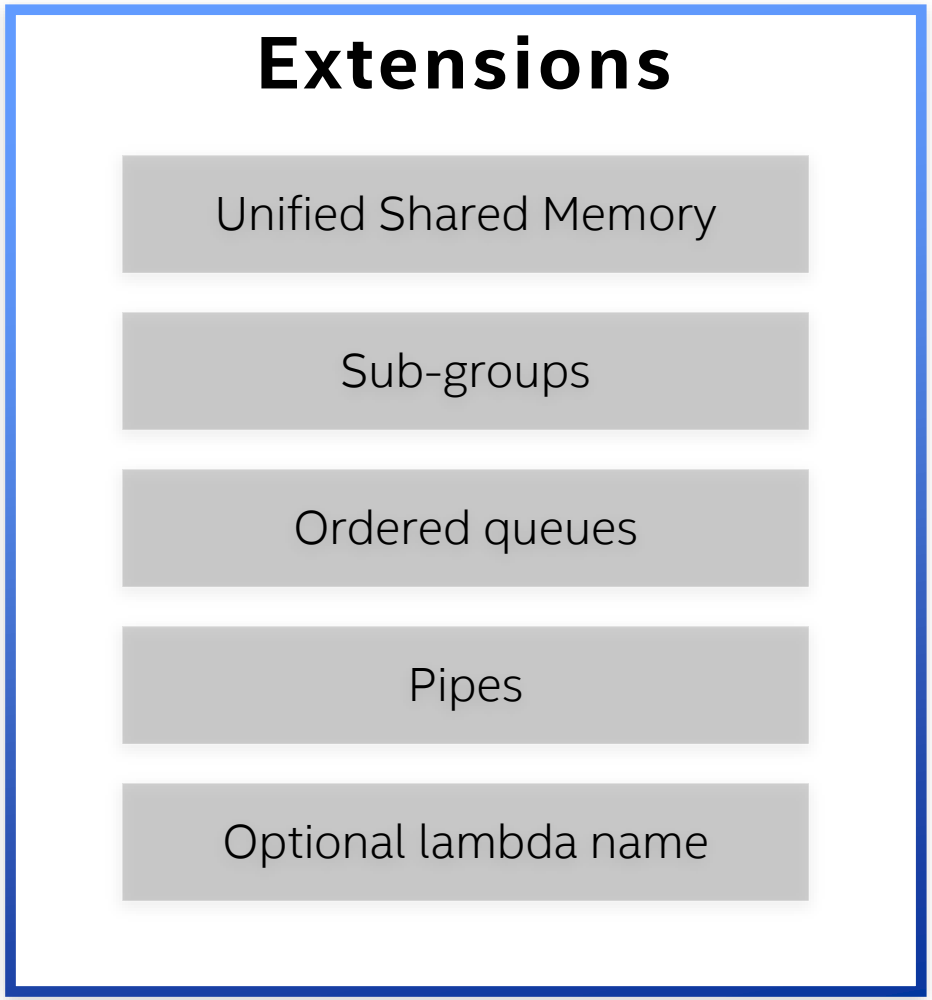
DATA PARALLEL C++

DPC++ = ISO C++ and Khronos* SYCL* and Extensions

Enhance productivity and performance

DPC++: Fast-moving open collaboration feeding into the SYCL standard

- Open source implementation with goal of upstream LLVM
- DPC++ extensions aim to become core SYCL, or Khronos extensions
  - SYCL 2020 is the newest version
  - 80% of SYCL 2020 features were DPC++ extensions



Performance Portability?

Does DPC++ claim to solve performance portability?

- No.

Normal DPC++ code will run well on different hardware

- Data parallel programming models map well to hardware

Peak performance may require hardware specific optimizations

- Different block sizes, optimized operations, etc.

Language Features

Buffers

Abstracts memory management and data movement

Unified Shared Memory

Pointer-based programming that enables desired level of control over data movement

Device Management

Manage contexts; Discover and select devices

Built-in Functions

Rich library of common math operators and atomic operations

Scheduler

Graph execution of kernels controlled by dependences

Data Parallelism and Tasks

Many ways to express tasks and data parallel computations

Low Level Interop

Interoperability with Level Zero, OpenCL, CUDA, etc objects

Data Types

Scalar and vector data types, swizzles

Error Handling

Detect and diagnose

Extensions

Device-specific extensions to enable best performance

Buffers

Buffers are one abstraction for data in DPC++/SYCL

- Multi-dimensional arrays
- Not suited for complex data structures that contain pointers

Data movement happens implicitly through data dependences

Declare C++ Arrays

Initialize Arrays

Declare Buffers

Declare Accessors

Use Accessors in Kernel

Arrays Updated

```
auto A = (int *) malloc(N * sizeof(int));
auto B = (int *) malloc(N * sizeof(int));
auto C = (int *) malloc(N * sizeof(int));

for (int i = 0; i < N; i++) {
  A[i] = i; B[i] = 2*i;
}

{
  buffer<int, 1> Ab(A, range<1>{N});
  buffer<int, 1> Bb(B, range<1>{N});
  buffer<int, 1> Cb(C, range<1>{N});

  q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = accessor(Ab, h, read_only);
    auto aB = accessor(Bb, h, read_only);
    auto aC = accessor(Cb, h, write_only);
    h.parallel_for(R, [=] (id<1> i) {
      aC[i] = aA[i] + aB[i];
    });
  });
  q.wait();
} // A,B,C updated
```

Unified Shared Memory

Unified Shared Memory (USM) is a new feature in DPC++ and SYCL 2020:

- Pointer-based approach for data management
- Complements buffers, not a replacement

Why USM?

- Simplify porting existing C++ codes
- Give desired level of control over data movement

USM Allocation Types

Three types of allocations:

- Device
  - Accessible on device, not on host
  - “Give me a pointer to an allocation in my GPU’s DRAM”
  - Use when you want full control over data movement
- Host
  - Accessible on host and device
  - Allocated in pinned Host memory, does not migrate to device
  - Useful for rarely-accessed or VERY large data sets
- Shared
  - Accessible on host and device – think CUDA “managed memory”
  - Can migrate between host and device
  - Use when you want things to “just work”

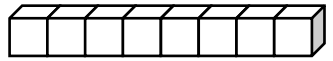
Parallelism within a Kernel: Simple

Application specifies size of the overall work

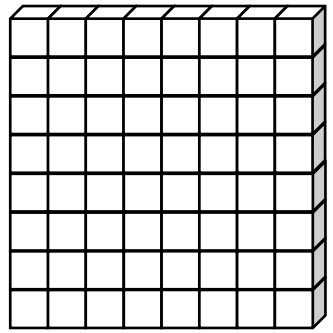
```
auto total = range{num};
buffer<int> a{total};

queue{}.submit([&](handler& h) {
  accessor out{a, h};
  h.parallel_for(total,
      [=](auto idx) {
    out[idx] = idx[0];
  });
});
```

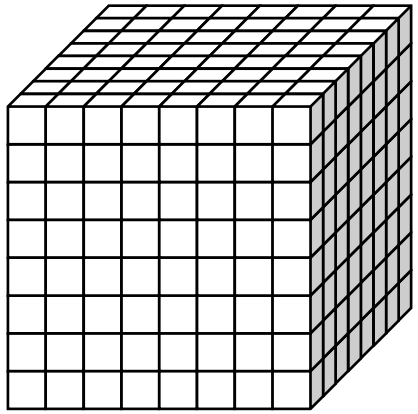
Runtime/HW decide how to apply kernel



1-D array



2-D array



3-D array

1 work-item
Kernel invoked once per work item

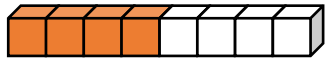
Parallelism within a Kernel: ND-Range

Application also specifies number of items to group into a “work-group”

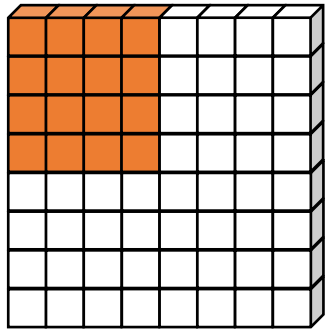
```
auto total = range{num};
auto wg = range{num/2};
buffer<int> a{total};

queue{}.submit([&](handler& h) {
  accessor out{a, h};
  h.parallel_for(nd_range{total, wg},
      [=](auto idx) {
    // kernel
  });
});
```

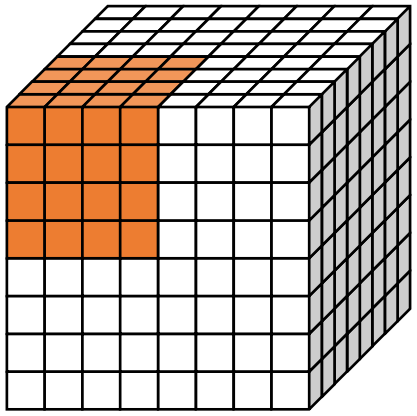
Gives application more control over how to map parallelism to hardware.



1-D array



2-D array



3-D array

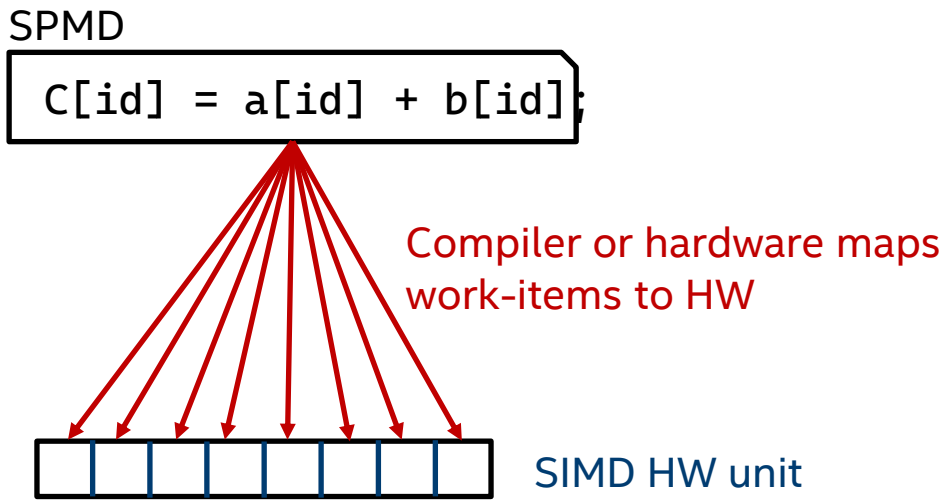
ND-range parallelism exposes hardware common on many accelerators, including new memory scope

Real applications are not so simple

Abstractions are great – until they're not

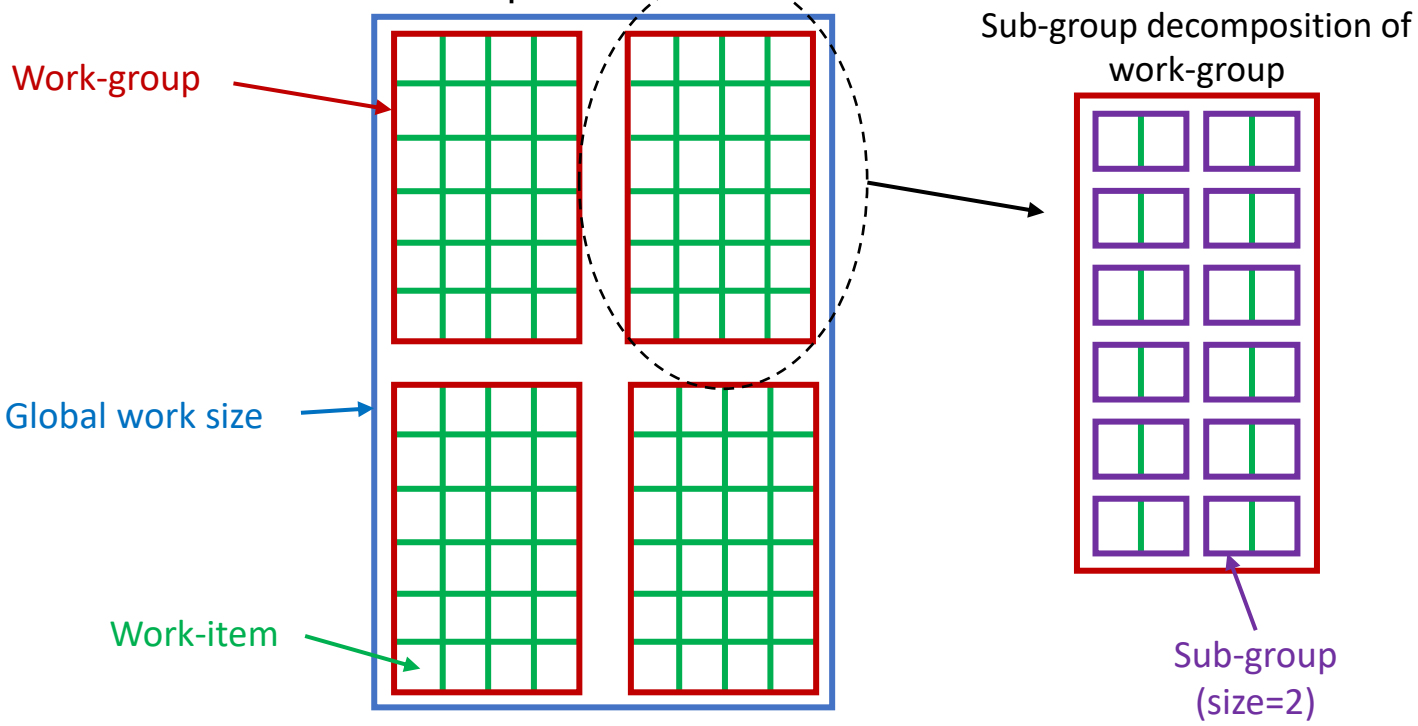
- Sometimes need to step outside the abstraction in small % of code to get best performance

Achieving close to peak performance can require more information in the compiler



Sub-groups

- Additional level of execution model hierarchy in DPC++
  - Formalizes mapping to what **can** be SIMD hardware – enables cross-lane collective operations



SYCL 2020 sub-group algorithms

```
sycl::sub_group sg = it.get_sub_group();

auto a = sycl::shift_group_left(sg, x, 1);
auto b = sycl::shift_group_right(sg, x, 1);
auto c = sycl::select_from_group(sg, x, id);
auto d = sycl::permute_group_by_xor(sg, x, mask);
```

e.g. Shuffles as free functions. Names aligned with C++

Reductions

Reductions are an important parallel pattern

- Language support enables:
  - Efficient implementations in the runtime
  - Programmer convenience by not reinventing the wheel

```
myQueue.submit([&](handler& h) {

  // Input values to reductions are standard accessors (or USM pointers)
  auto inputValues = accessor(valuesBuf, h, read_only);

  // Create temporary objects describing variables with reduction semantics
  auto sumReduction = reduction(sumBuf, h, plus<>());
  auto maxReduction = reduction(maxBuf, h, maximum<>());

  // parallel_for performs two reduction operations
  h.parallel_for(range<1>{1024}, sumReduction, maxReduction,
    [=](id<1> idx, auto& sum, auto& max) {
      sum += inputValues[idx];
      max.combine(inputValues[idx]);
  }); });
```

Want to learn more about DPC++?

Many ways to check it out:

- Download a oneAPI Toolkit from Intel
- Build the open source compiler from Github
- Try the Intel DevCloud

Want to learn DPC++?

- Great Jupyter notebook-based learning modules on DevCloud
- Read the free book
- Code samples: https://github.com/Apress/data-parallel-CPP