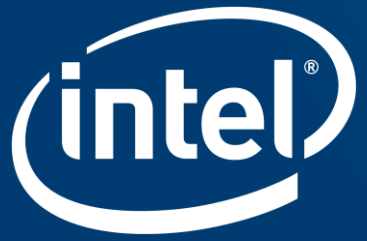


Using Intel® oneAPI Toolkits with FPGAs

introducing oneapi

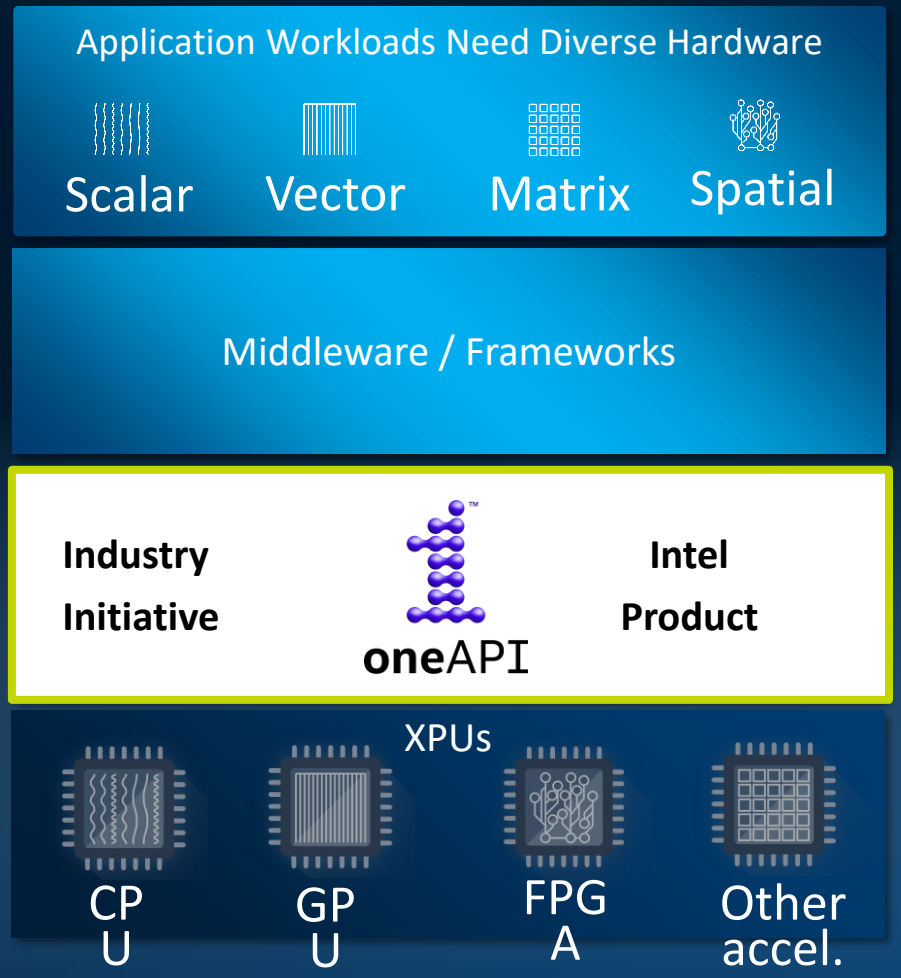
Unified programming model to simplify development across diverse architectures

Unified and simplified language and libraries for expressing parallelism

Uncompromised native high-level language performance

Based on industry standards and open specifications

Interoperable with existing HPC programming models

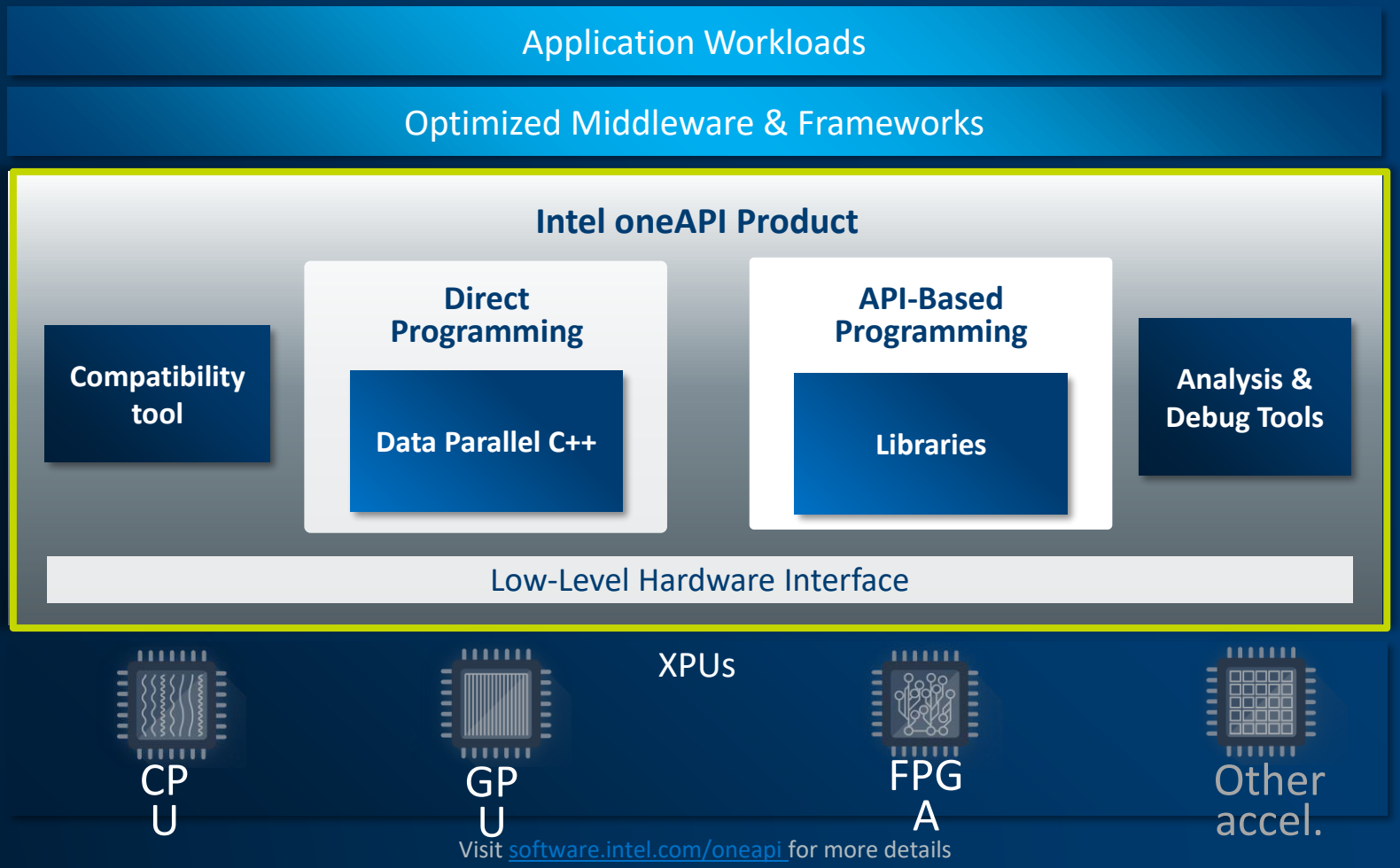


Intel® oneAPI products

Distributed through a core toolkit and a complementary set of add-on domain-specific toolkits

Includes DPC++ compatibility tool for code migration along with advanced performance analysis and debug tools

Beta Available Now



Visit software.intel.com/oneapi for more details

Some capabilities may differ per architecture and custom-tuning will still be required. Other accelerators to be supported in the future.

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

Intel® oneapi Data parallel C++ Compiler

Parallel programming productivity & performance

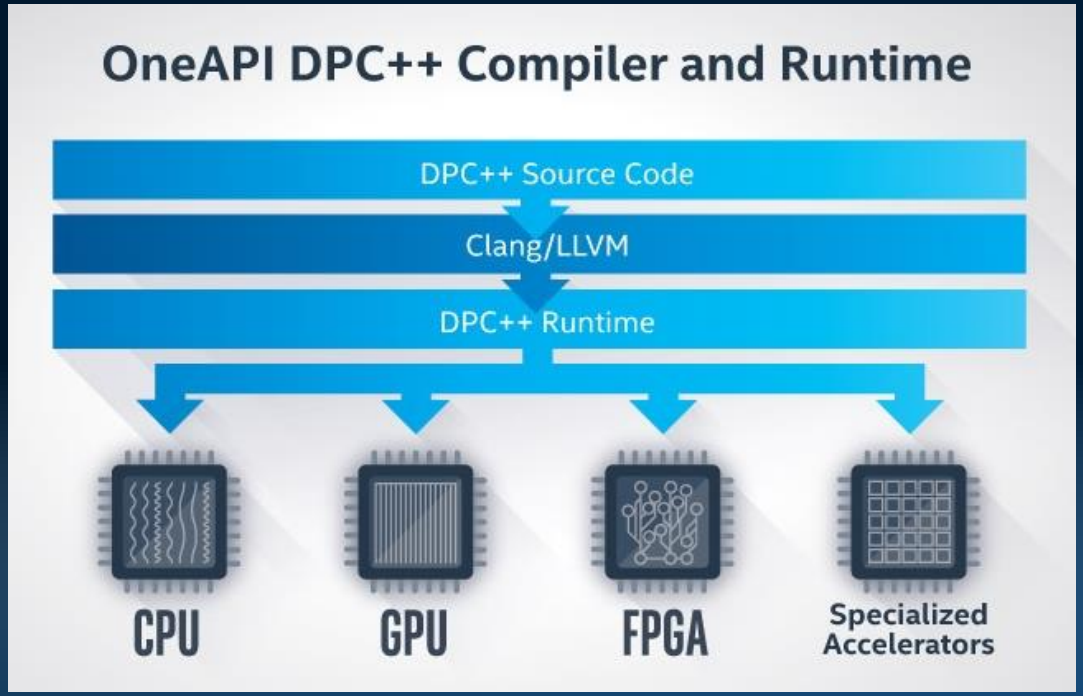
Compiler to deliver uncompromised parallel programming productivity and performance across CPUs and accelerators

- Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single architecture proprietary language

DPC++ is based on C++ and SYCL*

- Delivers C++ productivity benefits, using common and familiar C and C++ constructs
- Incorporates SYCL from The Khronos Group to support data parallelism and heterogeneous programming

Builds upon Intel's decades of experience in architecture and high performance compilers

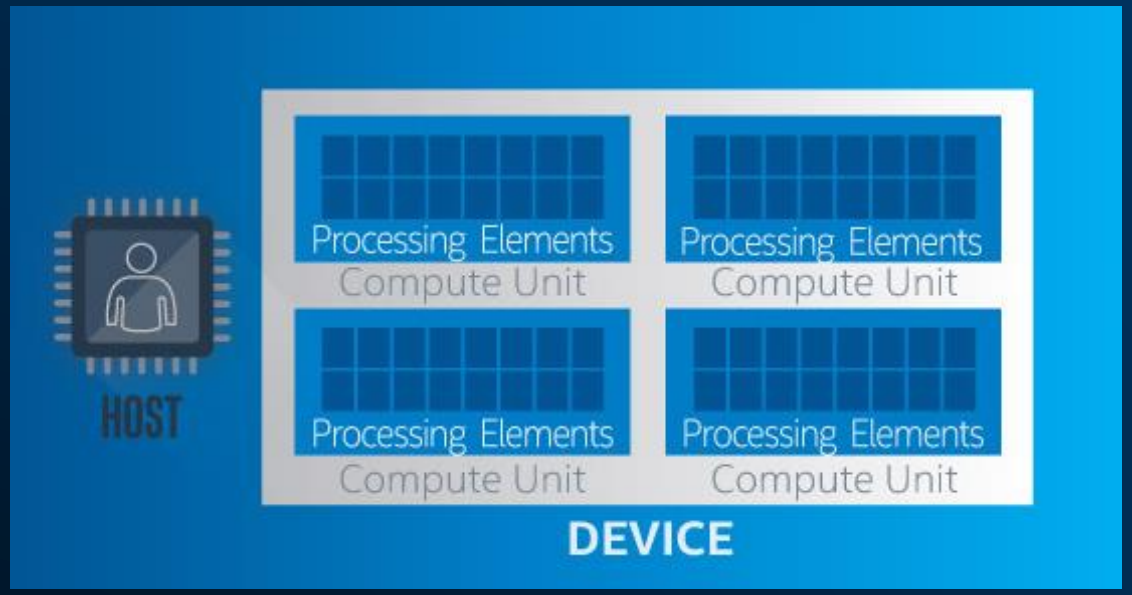


There will still be a need to tune for each architecture.

DPC++ Software Model

- Details four models to employ one or more devices as an accelerator.
- Platform model - Specifies the **host** and **device**(s).
    - Host: A CPU-based system that executes the application scope and command group scope.
    - Device: An accelerator or specialized component
        - Examples include CPU, FPGA, GPU.
- Execution model – Issues commands for execution on the device(s).
- Memory model - Defines how the host and devices interact with memory.
- Kernel model - Defines execution of code on the device(s).

Platform Model



- Platform – Host controlling one or more devices
- Device – accelerator that can execute kernels

Platform model enables:

- Enumeration of devices and device attributes
- Prioritized selection of devices for acceleration
- Fallback devices if higher priority devices fail execution

```
auto platforms = sycl::platform::get_platforms();

  for (auto &platform : platforms) {

    std::cout << "Platform: "
      << platform.get_info<sycl::info::platform::name>()
      << std::endl;


    auto devices = platform.get_devices();
    for (auto &device : devices ) {
      std::cout << "  Device: "
        << device.get_info<sycl::info::device::name>()
        << std::endl;
    }
  }
```

```
Platform: Intel(R) CPU Runtime for OpenCL(TM) Applications
  Device: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
  Device: Intel(R) FPGA Emulation Device
Platform: Intel(R) OpenCL HD Graphics
  Device: Intel(R) Gen9 HD Graphics NEO
```

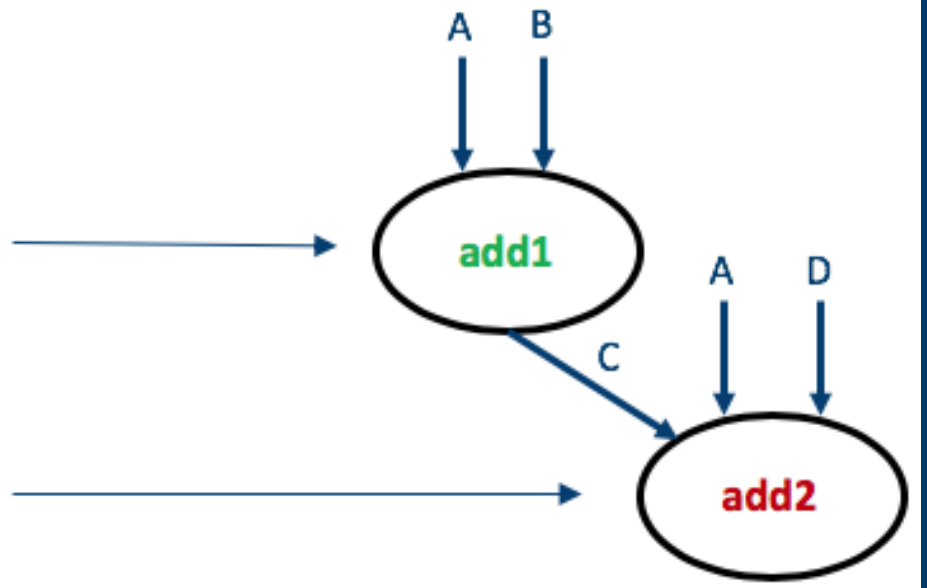
List platform and devices

Execution Model

Command group defines a set of constraints and a kernel. The handler associated with the command group is submitted to a command queue.

Queues are executed out-of-order enforcing the constraints. Constraints are communicated by employing accessors.

```
myQueue.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::read>(cgh);
      auto B = b.get_access<access::mode::read>(cgh);
      auto C = c.get_access<access::mode::discardwrite>(cgh);
      cgh.parallel_for<class add1>( range<2>{N, M},
          [=](id<2> index) { C[index] = A[index] + B[index]; });
});

myQueue.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::read>(cgh);
      auto C = c.get_access<access::mode::read>(cgh);
      auto D = d.get_access<access::mode::write>(cgh);
      cgh.parallel_for<class add2>( range<2>{P, Q},
          [=](id<2> index) { D[index] = A[index] + C[index]; });
});
```



Memory Model

Buffers and Accessors coordinate memory between host and devices. Ensure correctness and performance.

Buffer encapsulates a 1, 2, 3-dimensional array to share between host and devices.

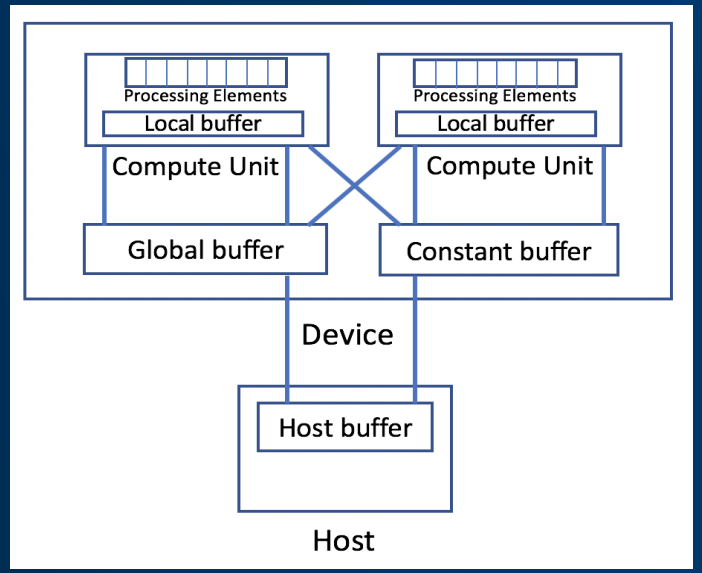
Member functions to obtain size, range, number of elements.

Access target specifies memory location requirement.

Private memory is determined by compiler or by employing private_memory class.

```
sycl::buffer<int>  a_device(a.data(), a_size);
sycl::buffer<int>  c_device(c.data(), a_size);

d_queue.submit([&](sycl::handler &cgh) {
  sycl::accessor<int, 1, sycl::access::mode::discard_write,
    sycl::access::target::global_buffer> c_res(c_device, cgh);
  sycl::accessor<int, 1, sycl::access::mode::read,
    sycl::access::target::constant_buffer> a_res(a_device, cgh);
```

Access Targets



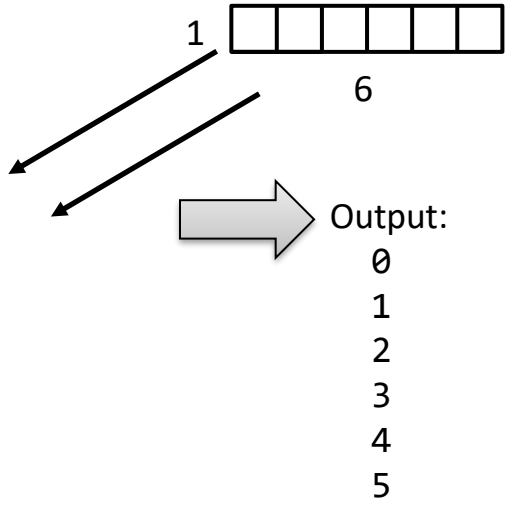
Memory Hierarchy

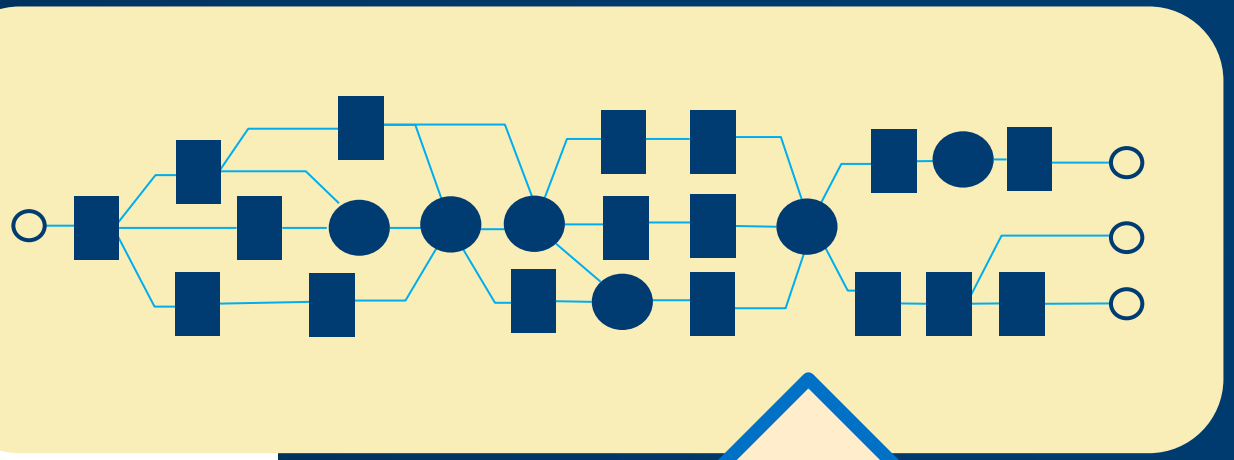
KERNEL Model

single_task()

- Similar to CPU code with an outer loop
- Allows many-staged custom hardware to be built in an FPGA

```
myQueue.submit([&](handler & cgh) {
    stream os(1024, 80, cgh);

    cgh.single_task<class myKernel>([=] () {
      for (int i=0;i<NUM_ELEMENTS;i++) {
        os << i << "\n";
      }
    });
});
```





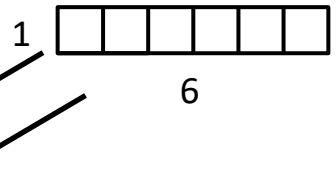
A custom hardware datapath can be generated in an FPGA for complex single_task kernels

KERNEL Model

parallel_for(num_work_items)

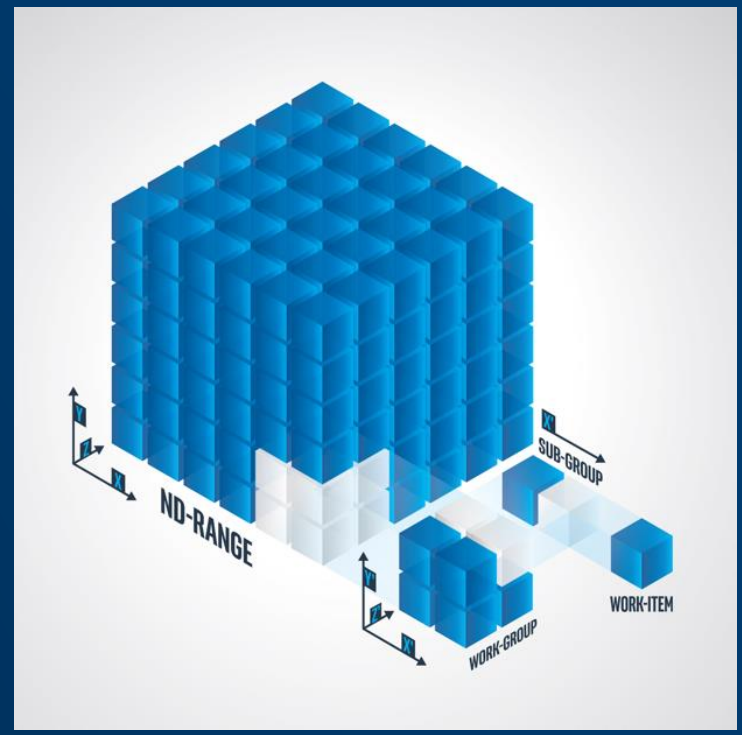
- Execute kernel in parallel over a 1, 2, or 3 dimensional index space
- Work-item can query ID and range of invocation (num_work_items)

```
myQueue.submit([&](handler & cgh) {
    stream os(1024, 80, cgh);

    cgh.parallel_for<class myKernel>(range<1>(6),
                                     [=] (id<1> index) {
        os << index << "\n";
    });
});
```



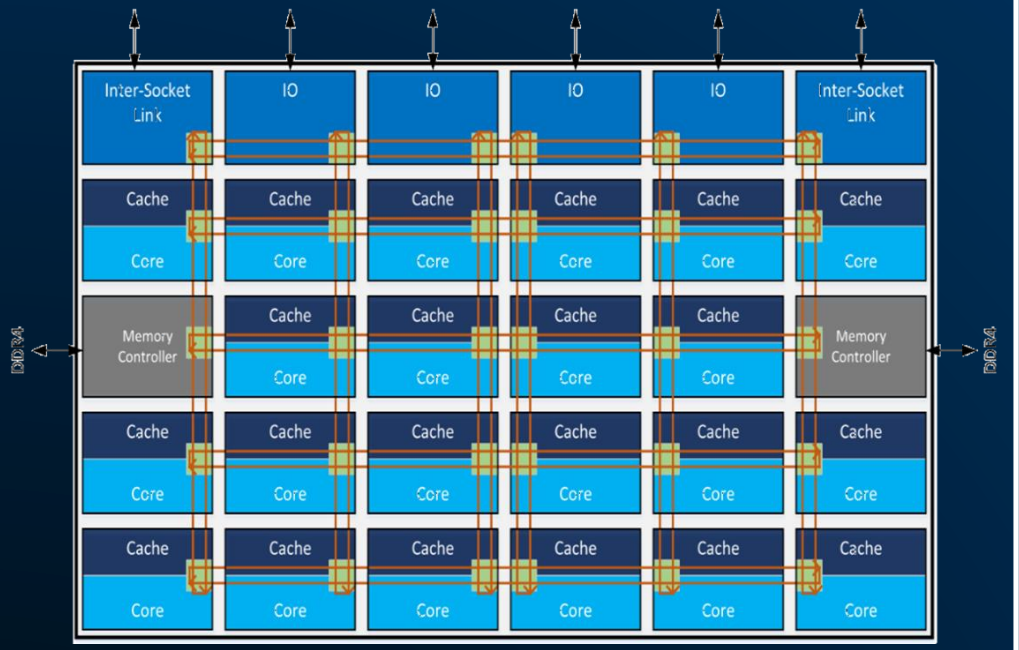
Output:

```
id<1>{ 0 }
id<1>{ 1 }
id<1>{ 2 }
id<1>{ 3 }
id<1>{ 4 }
id<1>{ 5 }
```



Can communicate execution across ND-Range
Sub-group is a DPC++ extension.

How it can be coded for CPU, GPU, FPGA

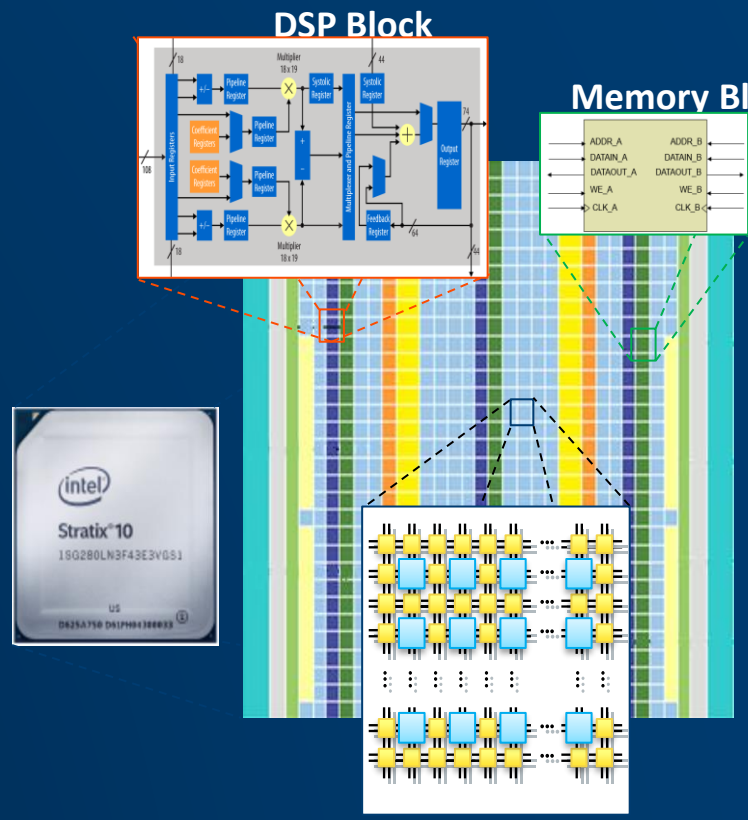


CPU

- MULTI-CORE
- MULTI-THREADED
- SIMD
- PIPELINED

GPU

- MULTI-CORE
- MULTI-THREADED
- SIMD
- PIPELINED



FPGA

- Custom Pipeline
- MULTI-CORE (pipeline)

SYCL Classes

DEVICE

- The **device** class represents the capabilities of the accelerators in a oneAPI system.
- The device class contains member functions for querying information about the device, which is useful for DPC++ programs where multiple devices are created.
- The function get_info gives information about the device:
  - Name, vendor, and version of the device
  - The local and global work item IDs
  - Width for built in types, clock frequency, cache width and sizes, online or offline

```
queue q;
device my_device = q.get_device();
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```

DEVICE SELECTOR

- The **device_selector** class enables the runtime selection of a particular device to execute kernels based upon user-provided heuristics.
- The following code sample shows use of the standard device selectors (default_selector, cpu_selector, gpu_selector…) and a derived **device_selector**

```
default_selector selector;
// host_selector selector;
// cpu_selector selector;
// intel::fpga_selector selector;
// gpu_selector selector;
queue q(selector);
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
```

queue

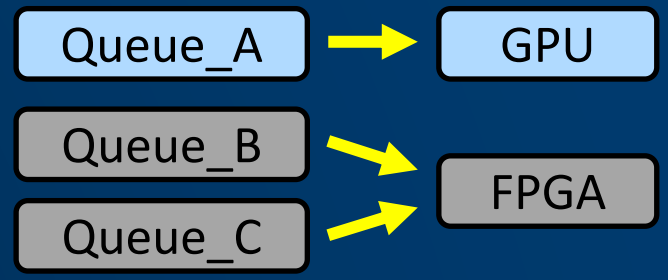
- A queue **submits command groups** to be executed by the SYCL runtime
- Queue is a mechanism where work is submitted to a device.
- A Queue maps to one device and multiple queues can be mapped to the same device.

```
queue q;

q.submit([&](handler& h) {
    // COMMAND GROUP CODE
});
```

Choosing where device kernels run

Work is submitted to queues

- Each queue is associated with exactly one device (e.g. a specific GPU or FPGA)
- You can:
  - Decide which device a queue is associated with (if you want)
  - Have as many queues as desired for dispatching work in heterogeneous systems



Create queue targeting any device:	queue();

Create queue targeting a pre-configured classes of devices:	queue(cpu_selector{});
	queue(gpu_selector{});
	queue(intel::fpga_selector{});
	queue(accelerator_selector{});
	queue(host_selector{}); **Always availab**

Create queue targeting specific device (custom criteria):	class custom_selector : public device_selector {
	int operator()(…… **// Any logic you want!**
	…
	queue(custom_selector{});

kernel

- The kernel class encapsulates methods and data for executing code on the device when a command group is instantiated
- Kernel object is not explicitly constructed by the user
- Kernel object is constructed when a kernel dispatch function, such as parallel_for or single_task, is called

```
q.submit([&](handler& h) {
  h.parallel_for(range<1>(N), [=](id<1> i) {
    A[i] = B[i] + C[i]);
  });
});
```

DPC++ language and runtime

- DPC++ language and runtime consists of a set of C++ classes, templates, and libraries
- **Application scope** and **command group scope** :
  - Code that executes on the host
  - The full capabilities of C++ are available at application and command group scope
- **Kernel scope**:
  - Code that executes on the device.
  - At kernel scope there are limitations in accepted C++

Single Task kernels

- Single-task kernels allow complex or lengthy datapaths to be built from custom hardware in FPGAs.
- Useful to offload code with dependencies that are difficult to execute in a data parallel fashion.
- Ideal for FPGAs

for-loop in CPU application

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

Offload to accelerator using **single_task**

```
h.single_task([=](){
    for (int i=0; i < 1024; i++) {
       A[i] =  B[i] + C[i];
    }
});
```

Parallel Kernels

- Parallel Kernels allow multiple instances of an operation to execute in parallel.
- Useful to offload parallel execution of a basic for-loop in which each iteration is completely independent and in any order.
- Parallel kernels are expressed using the parallel_for function

for-loop in CPU application

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

Offload to accelerator using **parallel_for**

```
h.parallel_for(range<1>(1024), [=](id<1> i){
    A[i] =  B[i] + C[i];
});
```

Basic Parallel Kernels

The functionality of basic parallel kernels is exposed via range, id and item classes

- range class is used to describe the iteration space of parallel execution
- id class is used to index an individual instance of a kernel in a parallel execution
- item class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

```
h.parallel_for(range<1>(1024), [=](id<1> idx){
    // CODE THAT RUNS ON DEVICE
});
```

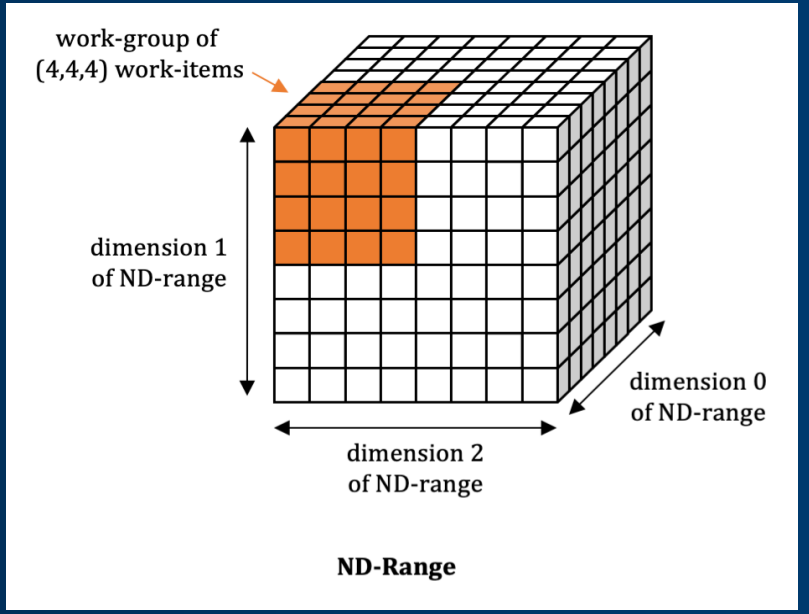
```
h.parallel_for(range<1>(1024), [=](item<1> item){
    auto idx = item.get_id();
    auto R = item.get_range();
    // CODE THAT RUNS ON DEVICE
});
```

ND_RANGE Kernels

Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level.

ND-Range kernel is another way to expresses parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units on hardware.

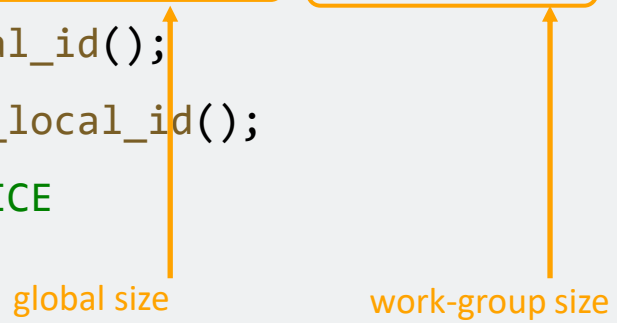
- The entire iteration space is divided into smaller groups called work-groups, work-items within a work-group are scheduled on a single compute unit on hardware.
- The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution.



ND_RANGE Kernels

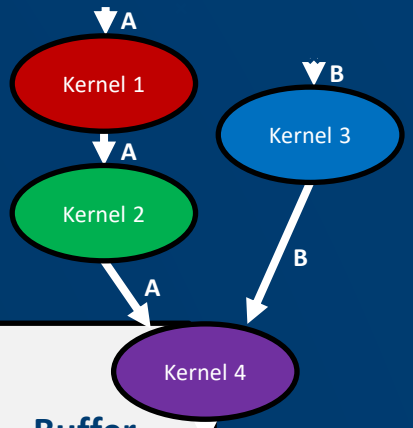
The functionality of nd_range kernels is exposed via nd_range and nd_item classes

```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)), [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // CODE THAT RUNS ON DEVICE
});
```



- nd_range class represents a grouped execution range using global execution range and the local execution range of each work-group.
- nd_item class represents an individual instance of a kernel function and allows to query for work-group range and index.

The Buffer Model



Buffers: Encapsulate data in a SYCL application

- Across both devices and host!

Accessors: Mechanism to access buffer data

- Create data dependencies in the SYCL graph that order kernel executions

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    auto out =
        A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) {
    auto out =
        A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });
…
```

Buffer

Accessor to buffer

DPC++ code anatomy

- oneAPI programs require the include of cl/sycl.hpp.
- Programs targeting FPGAs require the include of cl/sycl/intel/fpga_extensions.hpp.
- It is recommended to employ the namespace statement to save typing repeated references into the sycl namespace

```
#include <CL/sycl.hpp>
#include <CL/sycl/intel/fpga_extensions.hpp>
using namespace sycl;
```

DPC++ code anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue
(developer can specify a device type via device selector or use default selector)

DPC++ code anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

Step 2: create buffers (represent both host and device memory)

DPC++ code anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

Step 2: create buffers (represent both host and device memory)

Step 3: submit a command for (asynchronous) execution

DPC++ code anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

Step 2: create buffers (represent both host and device memory)

Step 3: submit a command for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

DPC++ code anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

Step 2: create buffers (represent both host and device memory)

Step 3: submit a command for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

Step 5: send a kernel (lambda) for execution

DPC++ code anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Kernel invocations are executed in parallel

Kernel is invoked for each element of the range

Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

Step 2: create buffers (represent both host and device memory)

Step 3: submit a command for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

Step 5: send a kernel (lambda) for execution

Step 6: write a kernel

Done!
The results are copied to vector `c` at `buf_c` buffer destruction

Asynchronous Execution

Think of a SYCL application as two parts:

1. Host code
2. The graph of kernel executions

These execute independently, except at synchronizing operations

- The host code submits work to build the graph (and can do compute work itself)
- The graph of kernel executions and data movements executes asynchronously from host code, managed by the SYCL runtime

Asynchronous Execution (cont'd)

Host

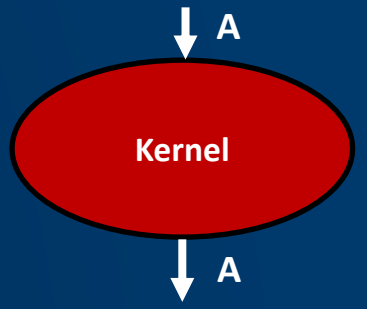
Host code execution

Enqueues kernel to graph, and keeps going

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

int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R };

  queue{}.submit([&](handler& h) {
    auto out = A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  auto result = A.get_access<access::mode::read>();
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "\n";

  return 0;
}
```

Graph

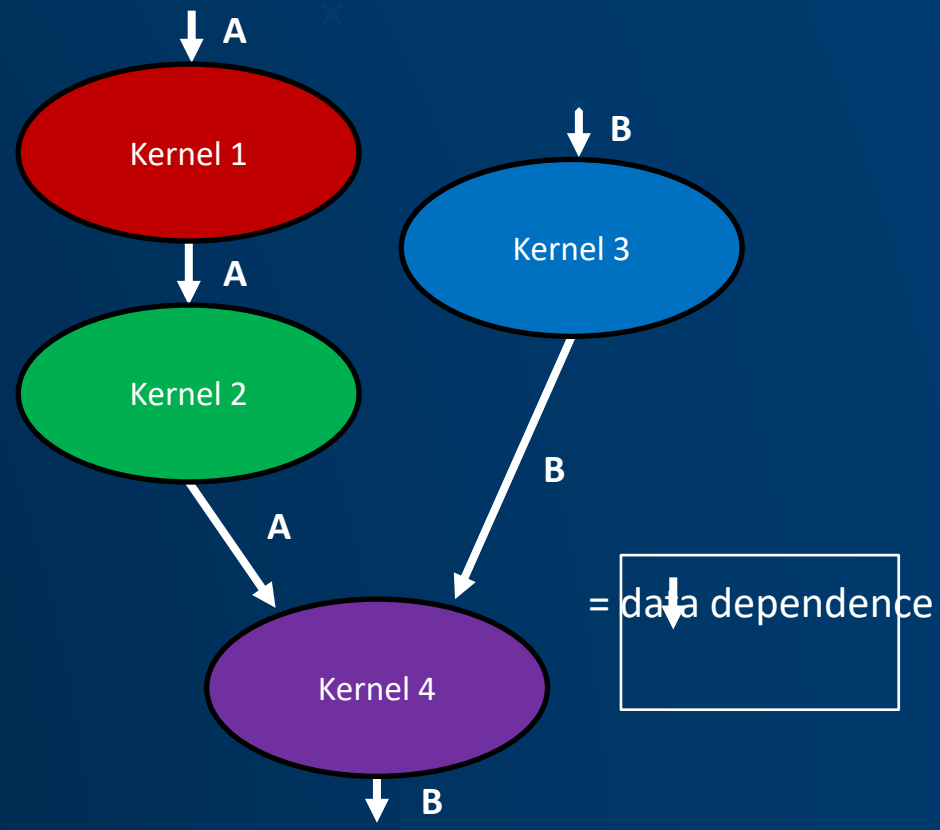
Graph executes asynchronously to host program



Graph of Kernel Executions

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                    } Kernel 1

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                    } Kernel 2

  Q.submit([&](handler& h) {
    auto out = B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                    } Kernel 3

  Q.submit([&](handler& h) {
  auto in = A.get_access<access::mode::read>(h);
  auto inout =
    B.get_access<access::mode::read_write>(h);
  h.parallel_for(R, [=](id<1> idx) {
    inout[idx] *= in[idx]; }); });                  } Kernel 4
}
```

Automatic data and control dependence resolution!



What is an FPGA?

First, let's define the acronym. It's a Field Programmable Gate Array.

“Field Programmable Gate Array” (FPGA)

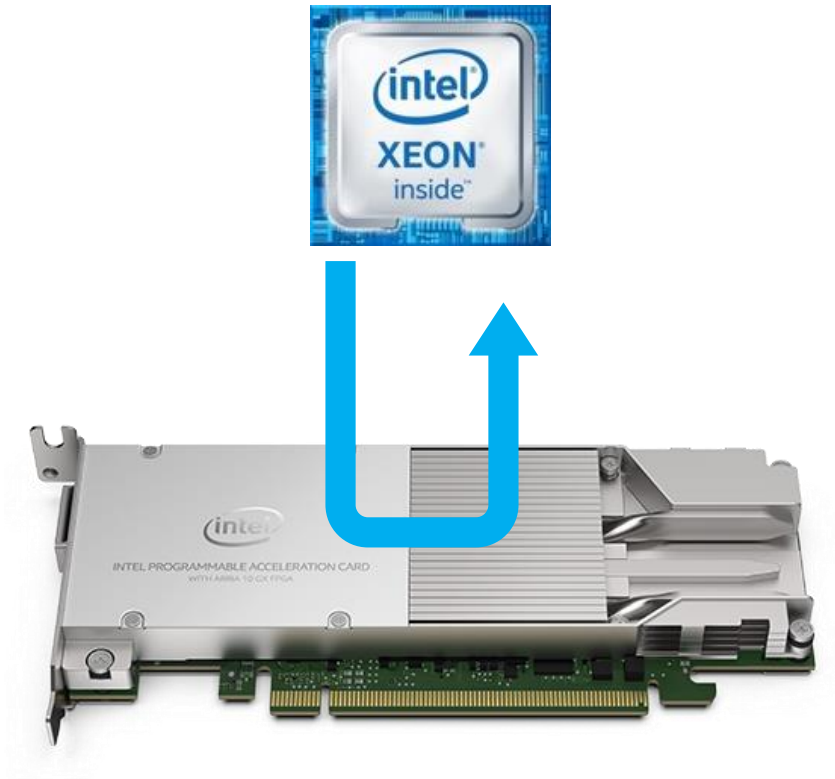
- “Gates” refers to transistors
  - These are the tiny pieces of hardware on a chip that make up the design
- “Array” means there are many of them manufactured on the chip
  - Many = Billions
  - They are arranged into larger structures as we will see
- “Field Programmable” means the connections between the internal components are programmable after deployment

FPGA = Programmable Hardware

How an FPGA Becomes What You Want It To Be

The FPGA is made up of small building blocks of logic and other functions

What About Connecting to the Host?

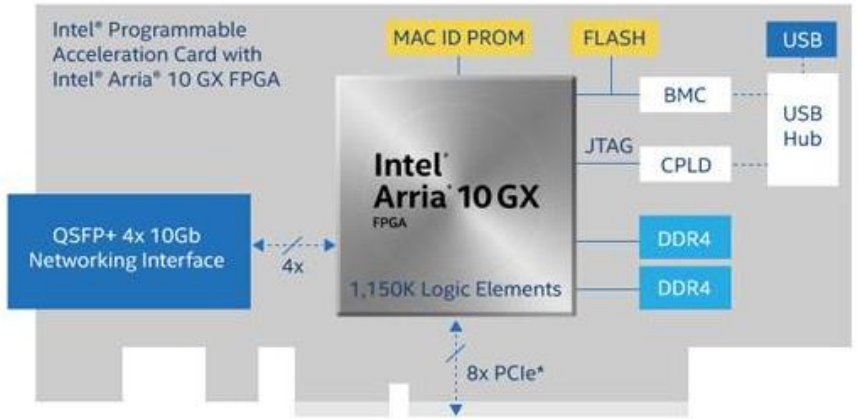


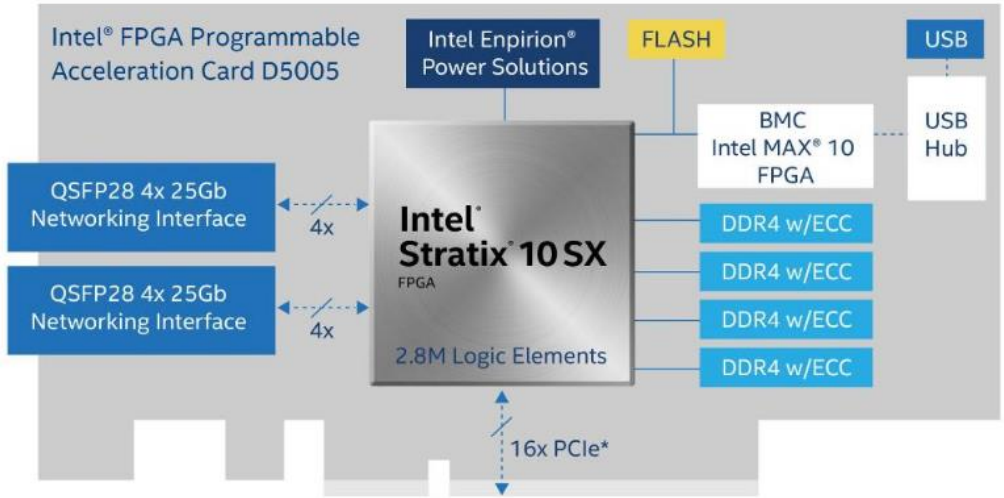
Accelerated functions run on a PCIe attached FPGA card

The host interface is also “baked in” to the FPGA design.

This portion of the design is pre-built and not dependent on your source code.

Intel® FPGAs Available





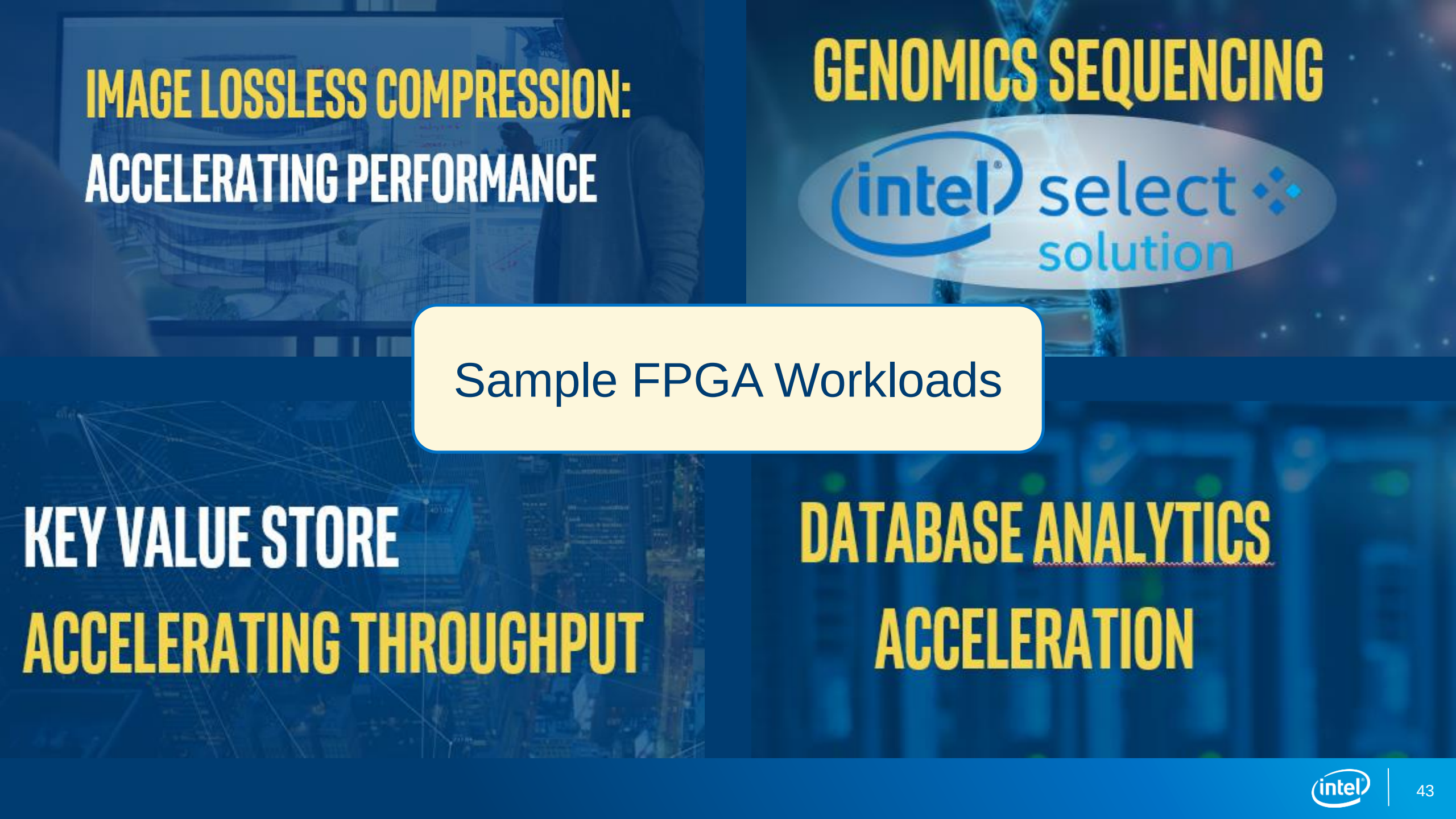


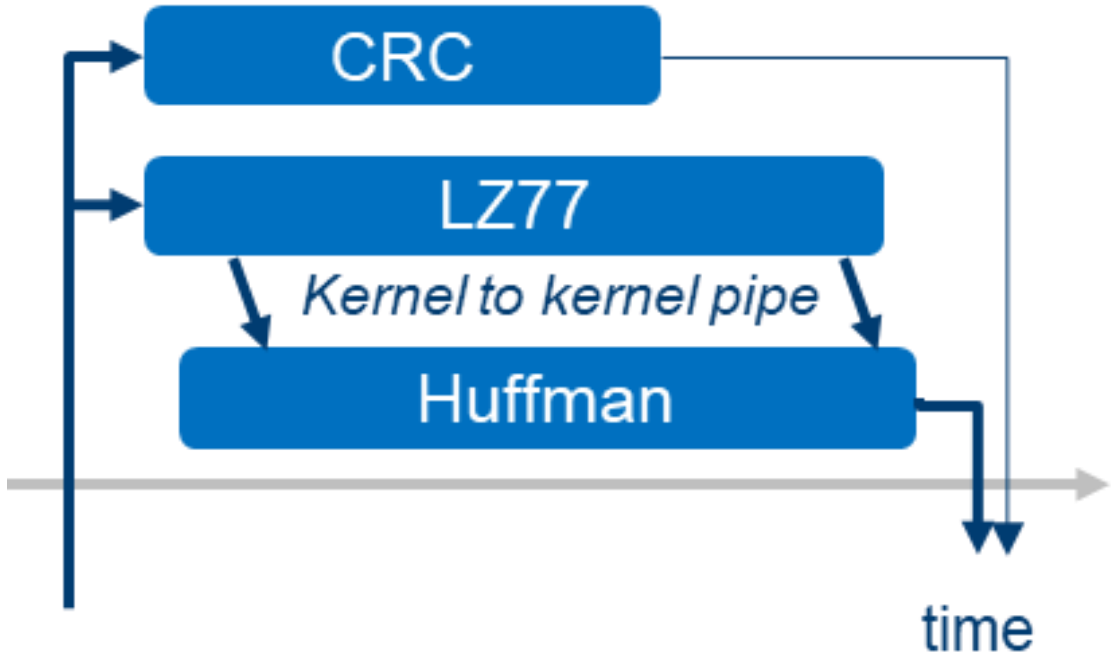
IMAGE LOSSLESS COMPRESSION:
ACCELERATING PERFORMANCE
GENOMICS SEQUENCING
intel select solution
Sample FPGA Workloads
KEY VALUE STORE
ACCELERATING THROUGHPUT
DATABASE ANALYTICS
ACCELERATION

Gzip Compression

FPGA Example included with the Intel® oneAPI Base Toolkit

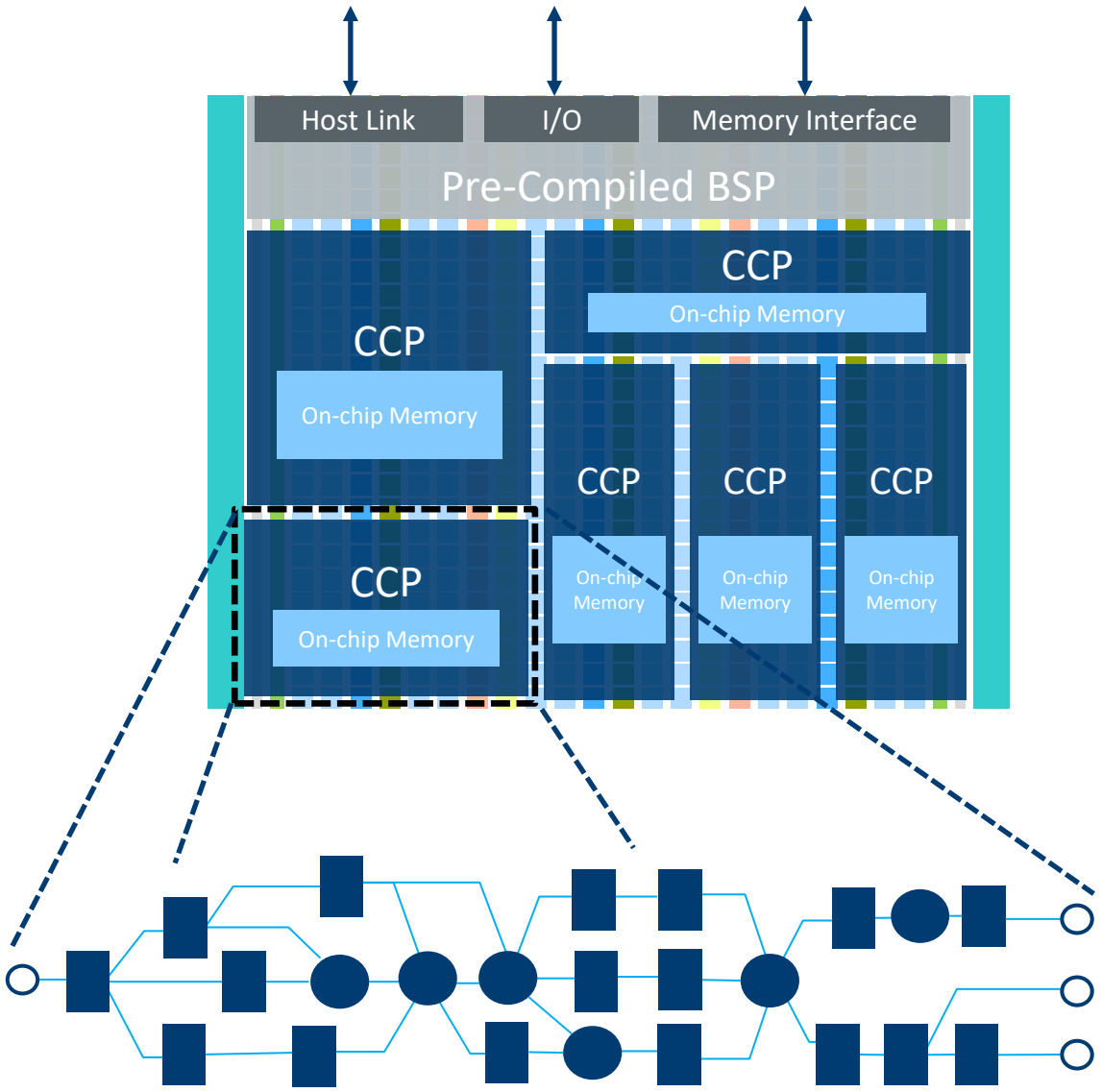
Concurrent kernels for LZ77, Huffman encoding and CRC

You are encouraged to try it for yourself!



Intel® FPGAs

Implementing Optimized **Custom Compute Pipelines (CCPs)** synthesized from compiled code

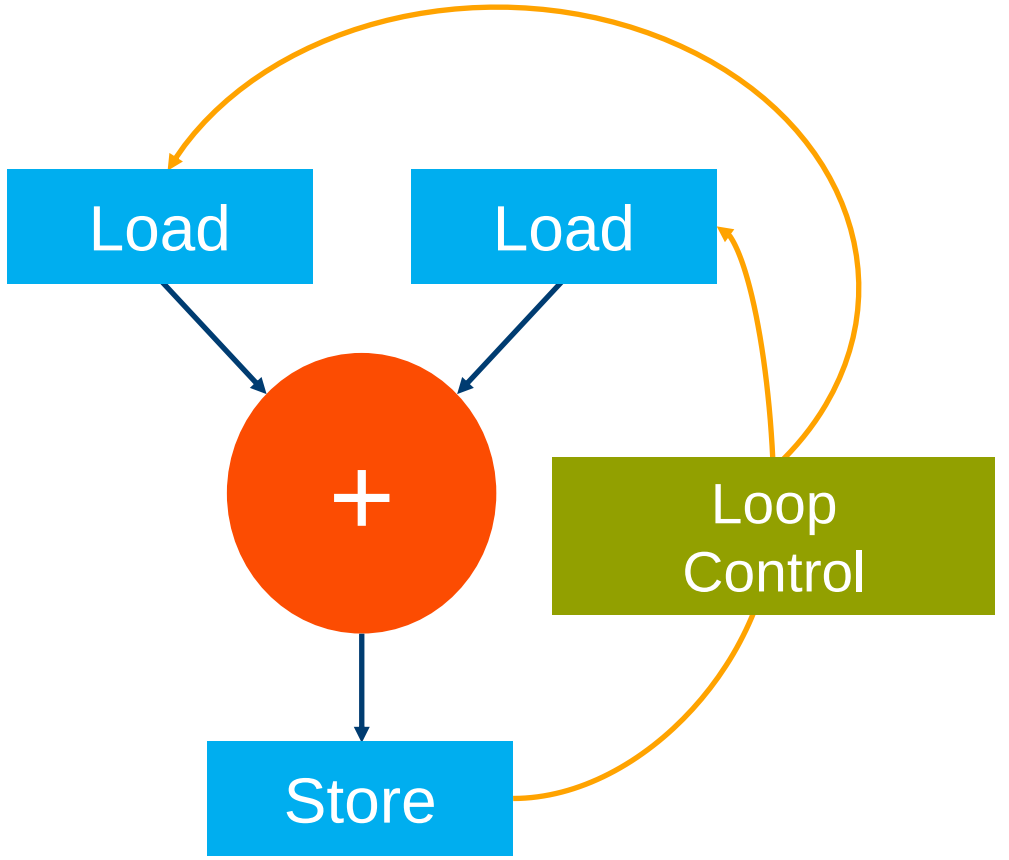


Custom Compute Pipeline

How Is a Pipeline Built?

Hardware is added for

- Computation
- Memory Loads and Stores
- Control and scheduling
  - Loops & Conditionals

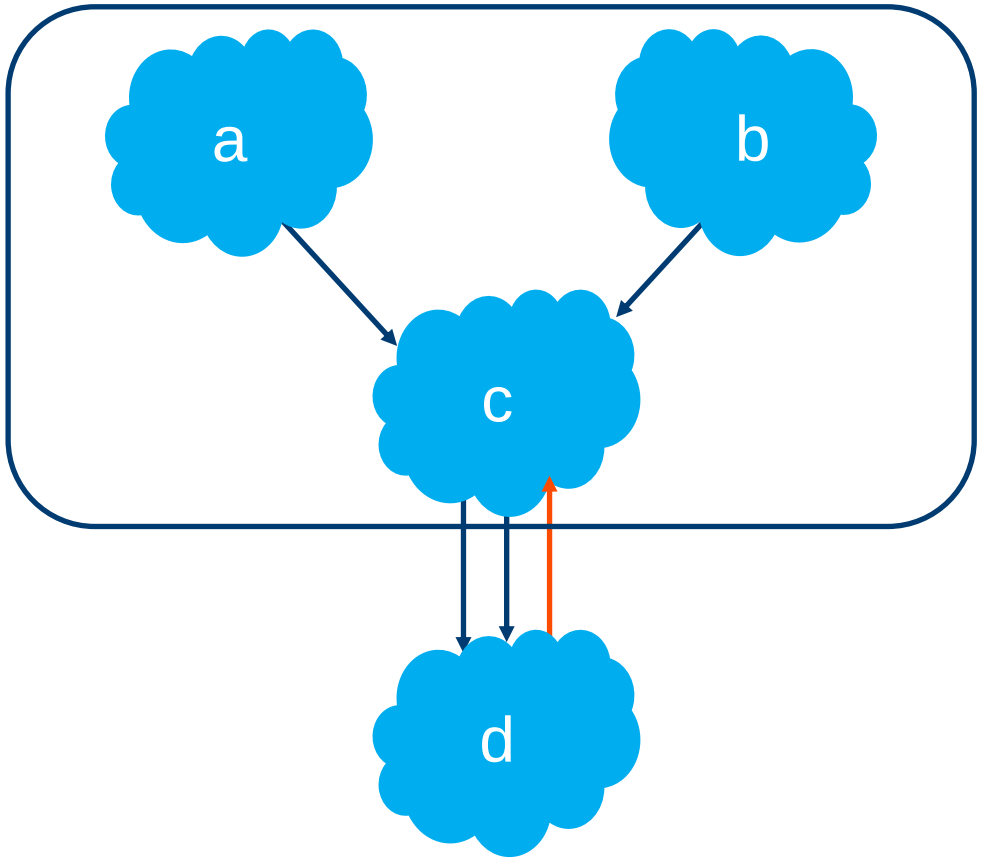



```
for (int i=0; i<LIMIT; i++) {
  c[i] = a[i] + b[i];
}
```

Data Path
Control Path

Connecting the Pipeline Together

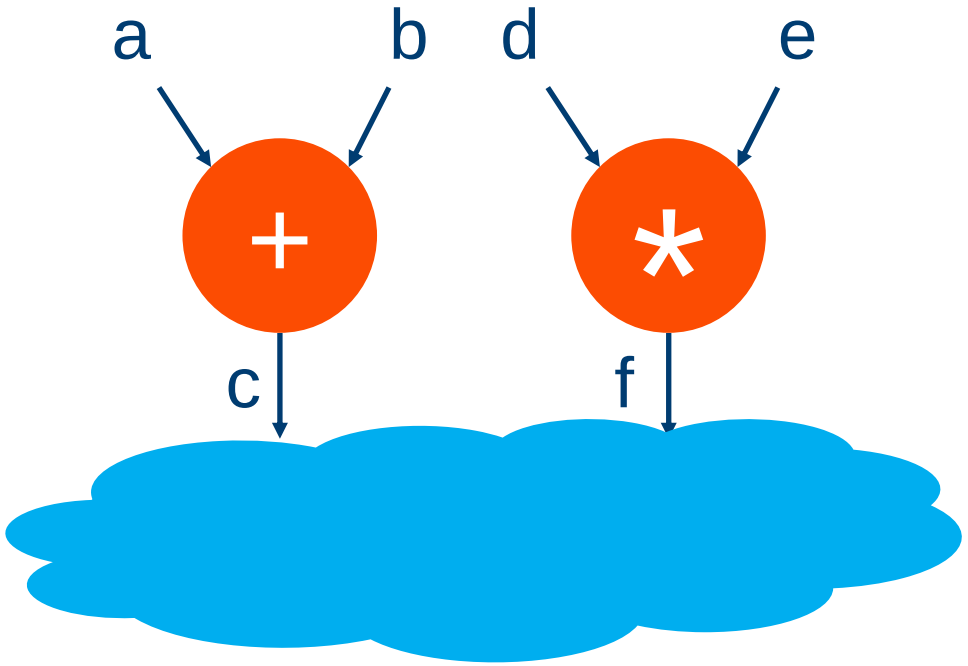
- Handshaking signals for variable latency paths
- Operations with a fixed latency are clustered together
- Fixed latency operations improve
  - Area: no handshaking signals required
  - Performance: no potential stalling due to variable latencies



Simultaneous Independent Operations

- The compiler automatically identifies independent operations
- Simultaneous hardware is built to increase performance
- This achieves data parallelism in a manner similar to a superscalar processor
- Number of independent operations only bounded by the amount of hardware

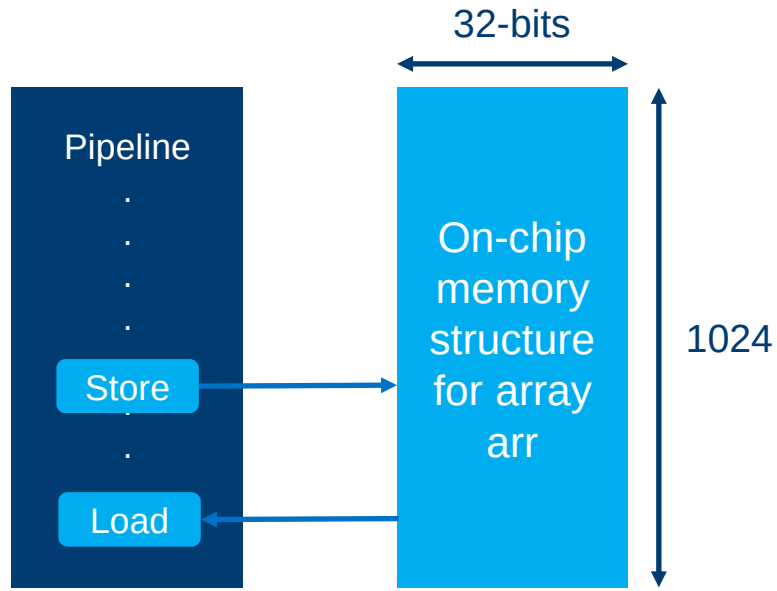
```
c = a + b;
f = d * e;
```



On-Chip Memories Built for Kernel Scope Variables

- Custom on-chip memory structures are built for the variables declared with the kernel scope
- Or, for memory accessors with a target of local
- Load and store units to the on-chip memory will be built within the pipeline

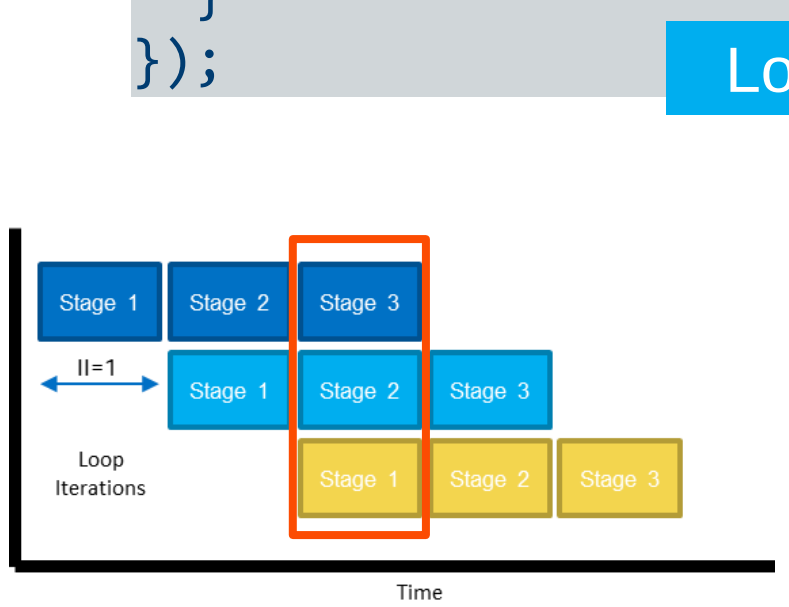
```
//kernel scope
cgh.single_task<>([=]() {
  int arr[1024];
  …
  arr[i] = …; //store to memory
  …
  … = arr[j] //load from memory
  …
} //end kernel scope
```

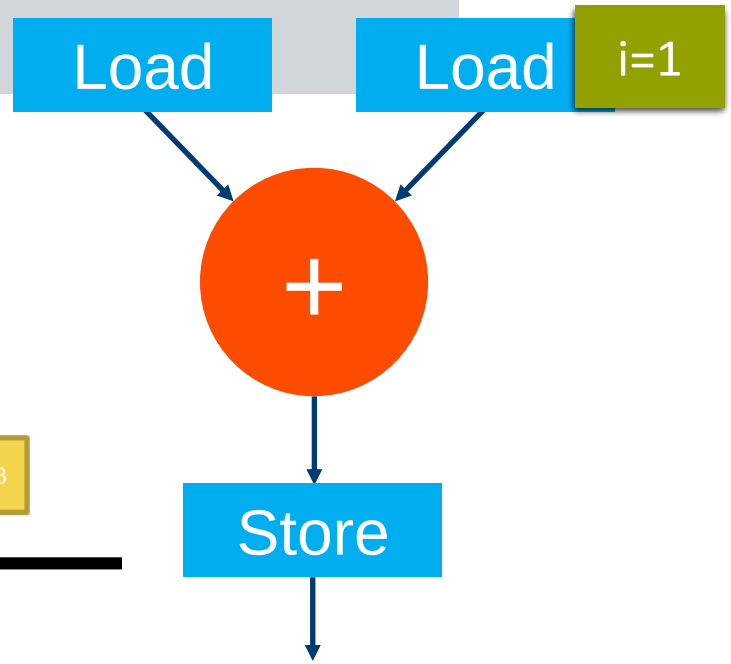


Pipeline Parallelism for Single Work-Item Kernels

- Single work-item kernels almost always contain an outer loop
- Work executing in multiple stages of the pipeline is called “pipeline parallelism”
- Pipelines from real-world code are normally hundreds of stages long
- **Your job is to keep the data flowing efficiently**

```
handle.single_task<>([=]() {
  … //accessor setup
  for (int i=0; i<LIMIT; i++) {
    c[i] += a[i] + b[i];
  }
});
```



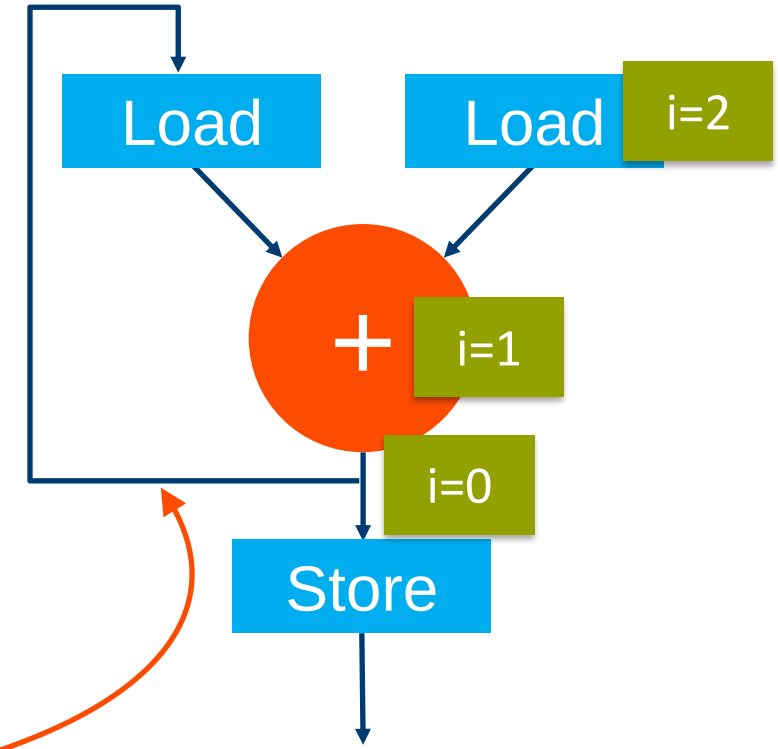


Dependencies Within the Single Work-Item Kernel

Key Concept
Custom built-in dependencies make FPGAs powerful for many algorithms

When a dependency in a single work-item kernel can be resolved by creating a path within the pipeline, the compiler will build that in.

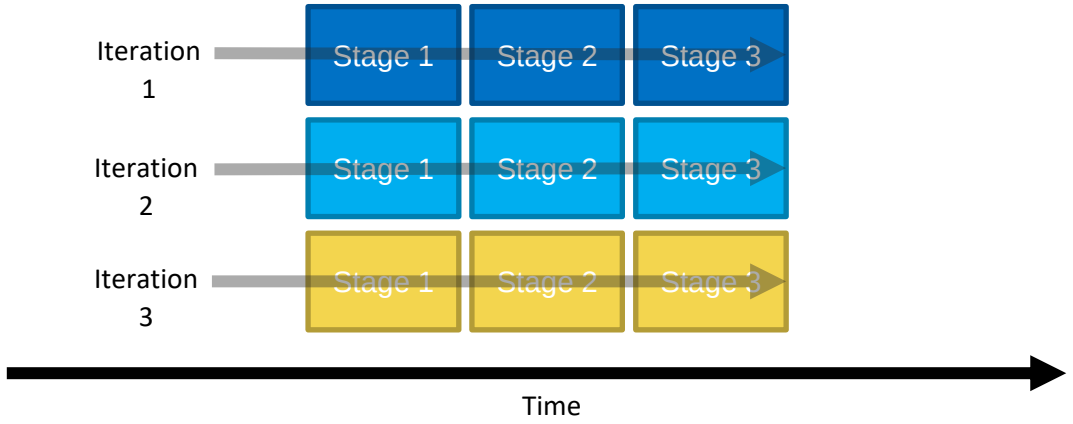
```
handle.single_task<>([=]() {
  int b = 0;
  for (int i=0; i<LIMIT; i++) {
    b += a[i];
  }
});
```



How Do I Use Tasks and Still Get Data Parallelism?

The most common technique is to unroll your loops

```
handle.single_task<>([=]() {
  … //accessor setup
  #pragma unroll
  for (int i=1; i<3; i++) {
    c[i] += a[i] + b[i];
  }
});
```

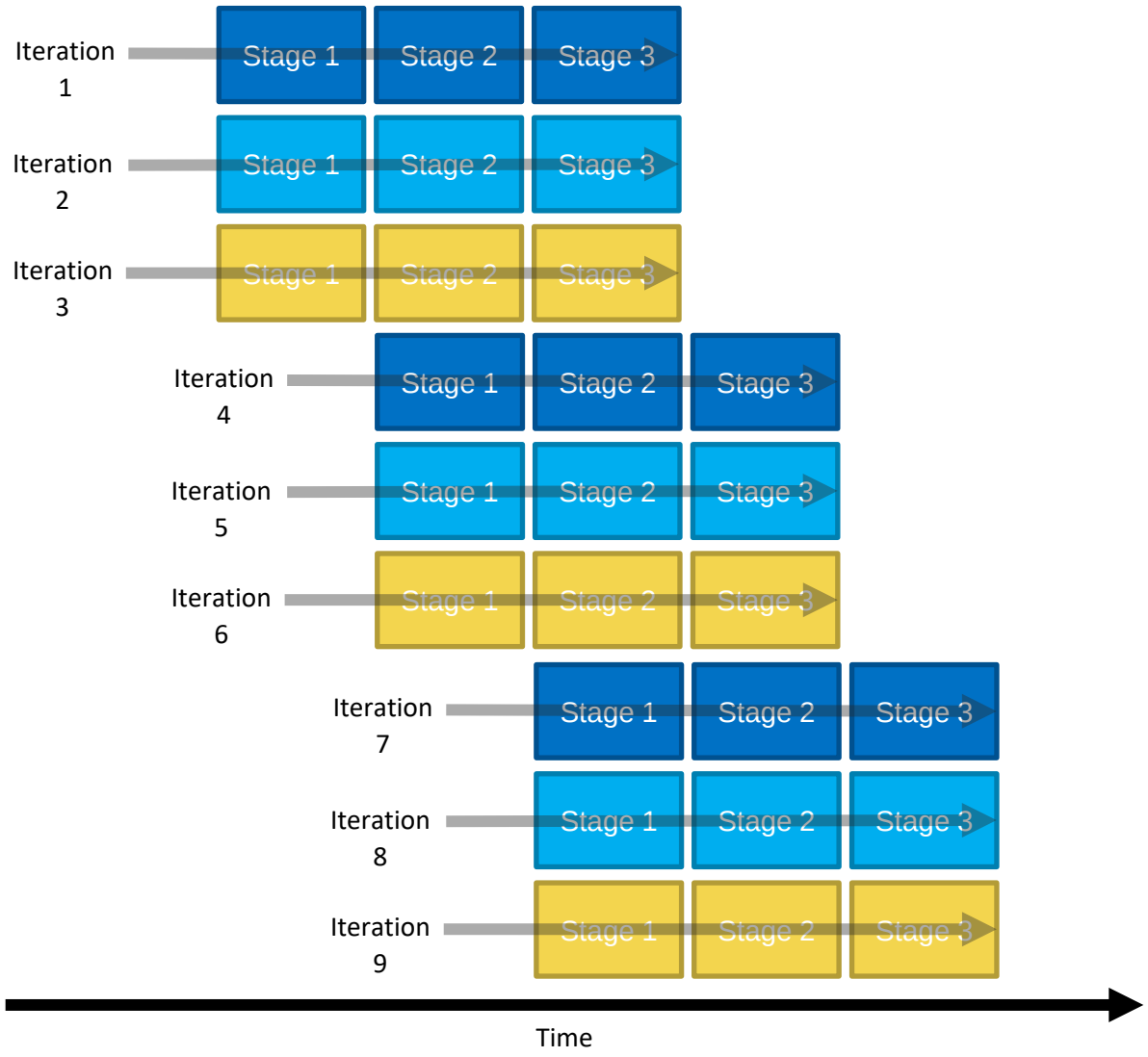


Unrolled Loops Still Get Pipelined

The compiler will still pipeline an unrolled loop, combining the two techniques

- A fully unrolled loop will not be pipelined since all iterations will kick off at once

```
handle.single_task<>([=]() {
  … //accessor setup
  #pragma unroll 3
  for (int i=1; i<9; i++) {
    c[i] += a[i] + b[i];
  }
});
```



What About Task Parallelism?

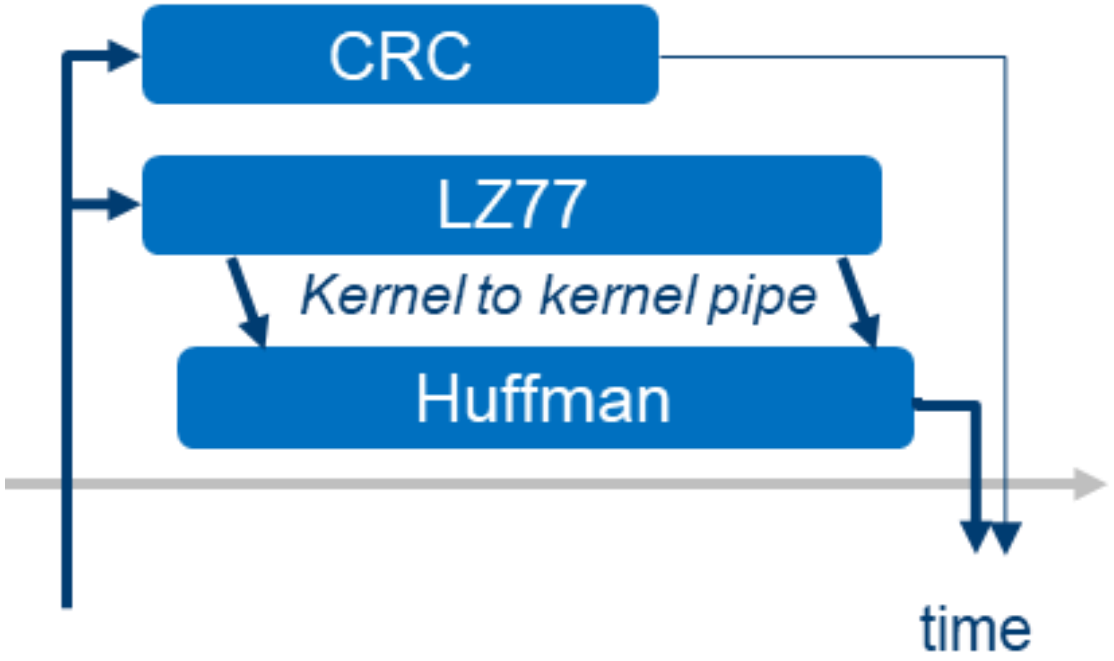
FPGAs can run more than one kernel at a time

- The limit to how many independent kernels can run is the amount of resources available to build the kernels

Data can be passed between kernels using pipes

- Another great FPGA feature explained in the Intel® oneAPI DPC++ FPGA Optimization Guide

Representation of Gzip FPGA example included with the Intel oneAPI Base Toolkit



So, Can We Build These? Parallel Kernels

- Kernels launched parallel_for() or parallel_for_work_group()

```
…//application scope

queue.submit([&](handler &cgh) {
  auto A = A_buf.get_access<access::mode::read>(cgh);
  auto B = B_buf.get_access<access::mode::read>(cgh);
  auto C = C_buf.get_access<access::mode::write>(cgh);

  cgh.parallel_for<class VectorAdd>(num_items, [=](id<1> wiID) {
      c[wiID] = a[wiID] + b[wiID];
   });

});

…//application scope
```

Yes, no problem, and you will learn to code them!

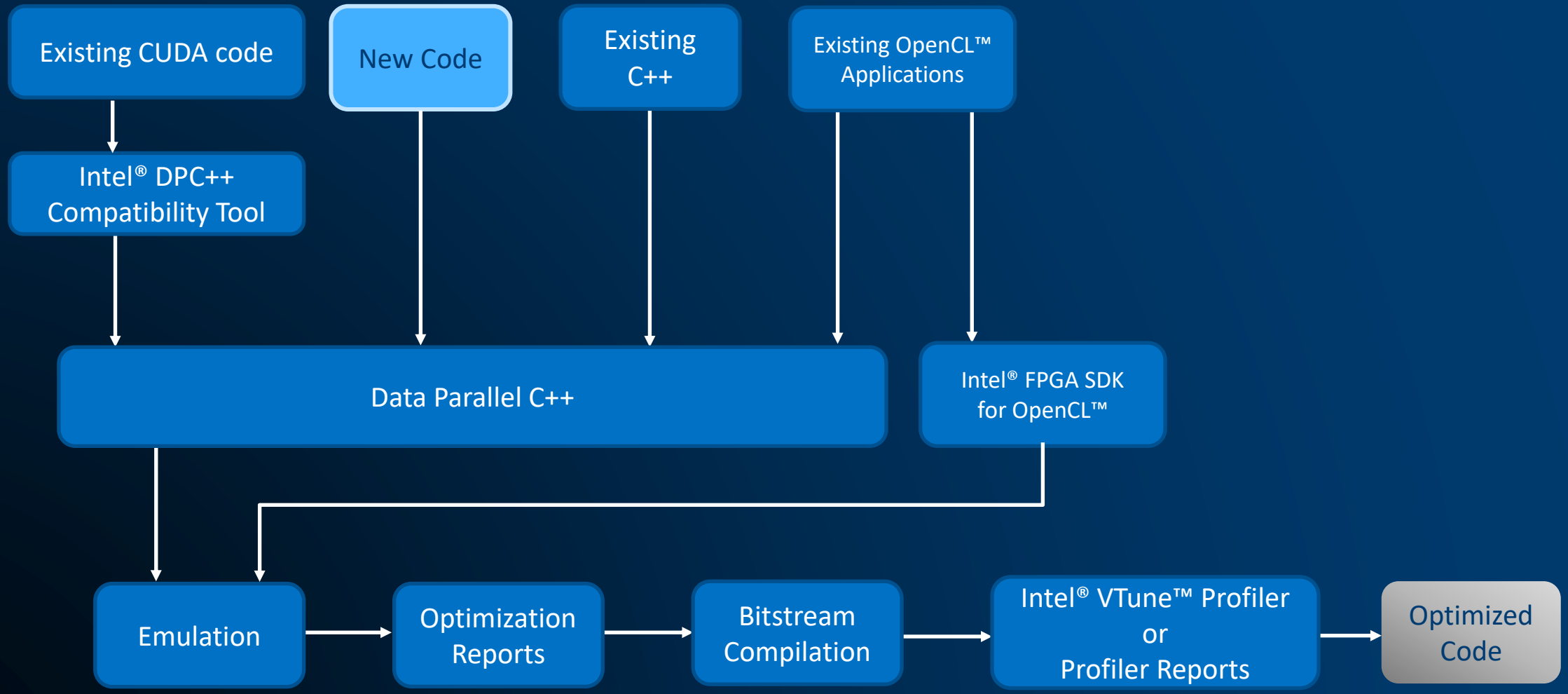
But, **tasks** usually imply more optimal pipeline structures.

The loop optimizations are limited for parallel kernels.

Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

oneAPI Single Node Workflow

I want to Accelerate direct programming on an FPGA…



Installing oneAPI

- Get started by visiting the Intel® Software Developer Zone page for the Intel® oneAPI Toolkits
  - https://software.intel.com/en-us/oneapi
- Get the Intel® oneAPI Base Toolkit for Linux*
  - Supports compiles for emulation and the optimization report
- Install the Intel FPGA Add-on for oneAPI Base Toolkit
  - Needed for compiles to FPGA hardware
  - Contains Intel® Quartus® Prime software “under the hood,” be sure to comply to required versions of operating system

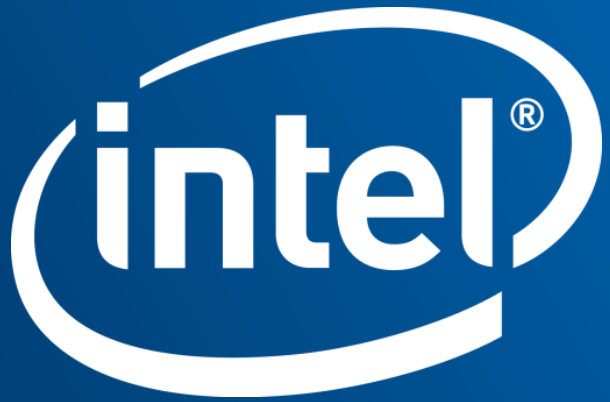
Or, Skip the Setup and Use the Intel DevCloud!

Sign up here:

- https://software.intel.com/devcloud/
  - Nodes with cards installed in the group fpga_runtime
  - Nodes with extra memory for full FPGA compiles in the group fpga_compile

References and Resources

- Website hub for using FPGAs with oneAPI
  - https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/fpga.html
- Intel® oneAPI Programming Guide
  - https://software.intel.com/content/www/us/en/develop/download/intel-oneapi-programming-guide.html
- Intel® oneAPI DPC++ FPGA Optimization Guide
  - https://software.intel.com/content/www/us/en/develop/download/oneapi-fpga-optimization-guide.html
- FPGA Tutorials GitHub
  - https://github.com/intel/BaseKit-code-samples/tree/master/FPGATutorials


intel