

# Using Intel<sup>®</sup> oneAPI Toolkits with FPGAs



Copyright © 2021 Intel Corporation.

This document is intended for personal use only.

Unauthorized distribution, modification, public performance,  
public display, or copying of this material via any medium is strictly prohibited.

# Course Objectives

- Understand the development flow for FPGAs with the Intel® oneAPI toolkits
- Gain an understanding of common optimization methods for FPGAs

# Course Agenda

- Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits
  - Introduction to oneAPI
  - Introduction to DPC++
  - What are FPGAs and Why Should I Care About Programming Them?
  - Development Flow for Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits
  - Lab: Practice the FPGA Development Flow
- Optimizing Your Code for FPGAs
  - Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits
  - Lab: Optimizing the Hough Transform Kernel

# Exact Timeline of This Class

Let's now look at the exact timeline for this class

(So, you can plan important things like lunch!)



# Section: Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits

## Sub-Topics:

- Introduction to oneAPI
- Introduction to DPC++
- What are FPGAs and Why Should I Care About Programming Them?
- Development Flow for Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits

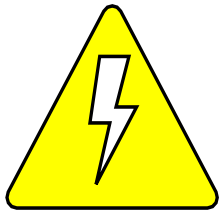
# Advantages of Heterogeneous Computing

## Multiple Architectures

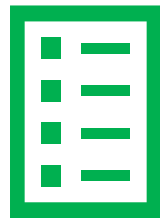
Developers can optimize specialized inline and offload workloads to meet business needs.

- Strengths of individual xPUs (CPU, GPU, FPGAs, etc.) can be combined for the benefit of the overall system.

Performance/Watt



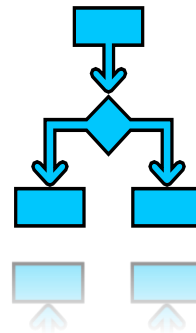
Throughput



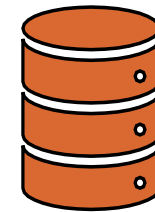
Latency



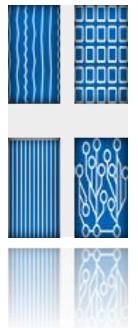
IO Flexibility



Memory Bandwidth

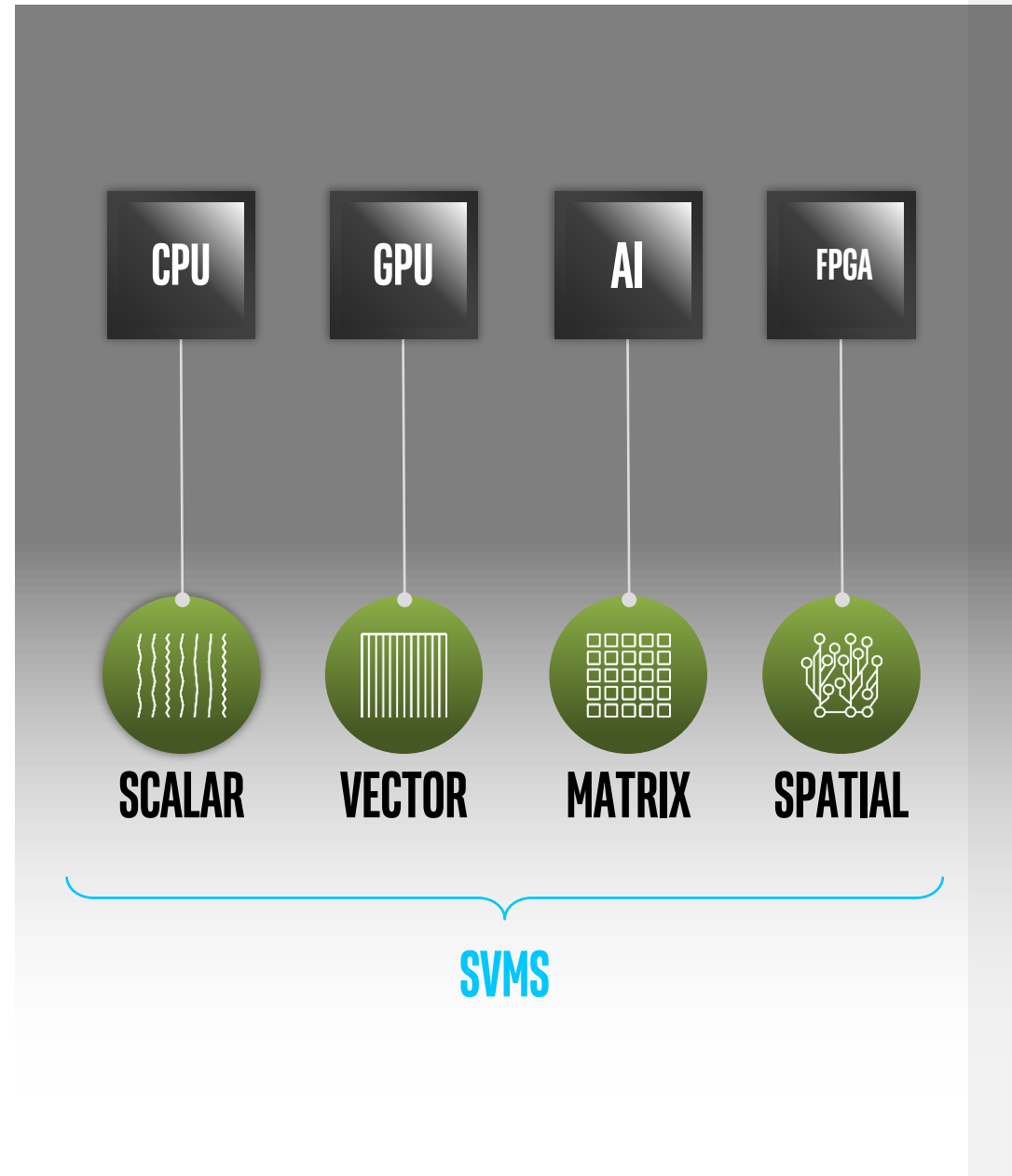


Architecture



# Diverse Workloads Require Diverse Architectures

**The future** is a **diverse** mix of scalar, vector, matrix, and spatial **architectures** deployed in CPU, GPU, AI, FPGA and other accelerators



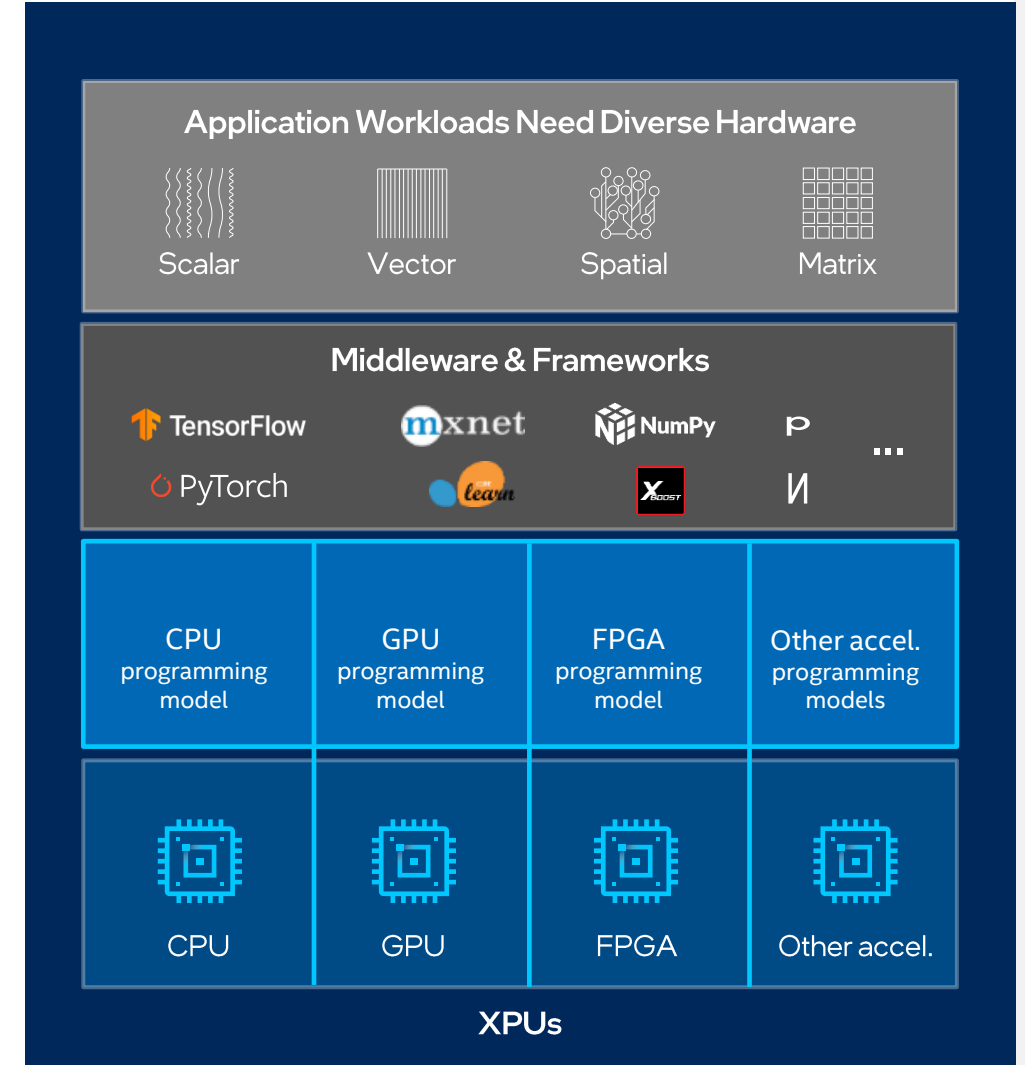
# Programming Challenges for Multiple Architectures

Separate programming models and toolchains for each architecture.

- Required **training** and **licensing** – compiler, IDE, debugger, analytics/monitoring tool, deployment tool, et al. – per architecture.
- Challenging experience in **debug**, **monitoring**, and **maintenance** of a cross-architectural source code.
- Difficult integration across proprietary IPs and architectures and **no code re-use**.

Software development complexity **limits** freedom of architectural choice.

- Isolated investments required for technical expertise to overcome the **barrier-to-entry**.





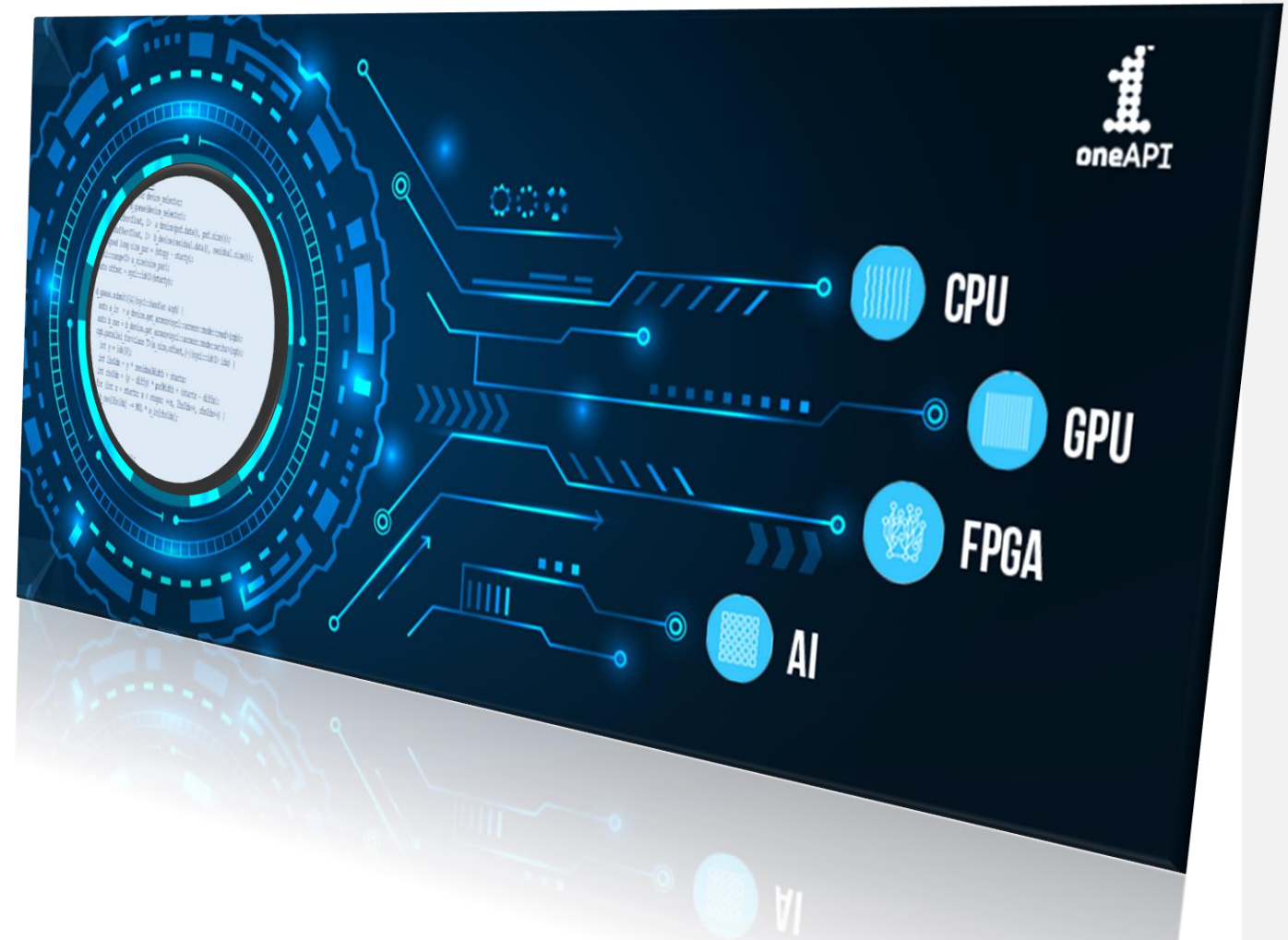
# A Unified Programming Model

## Multiple Architectures

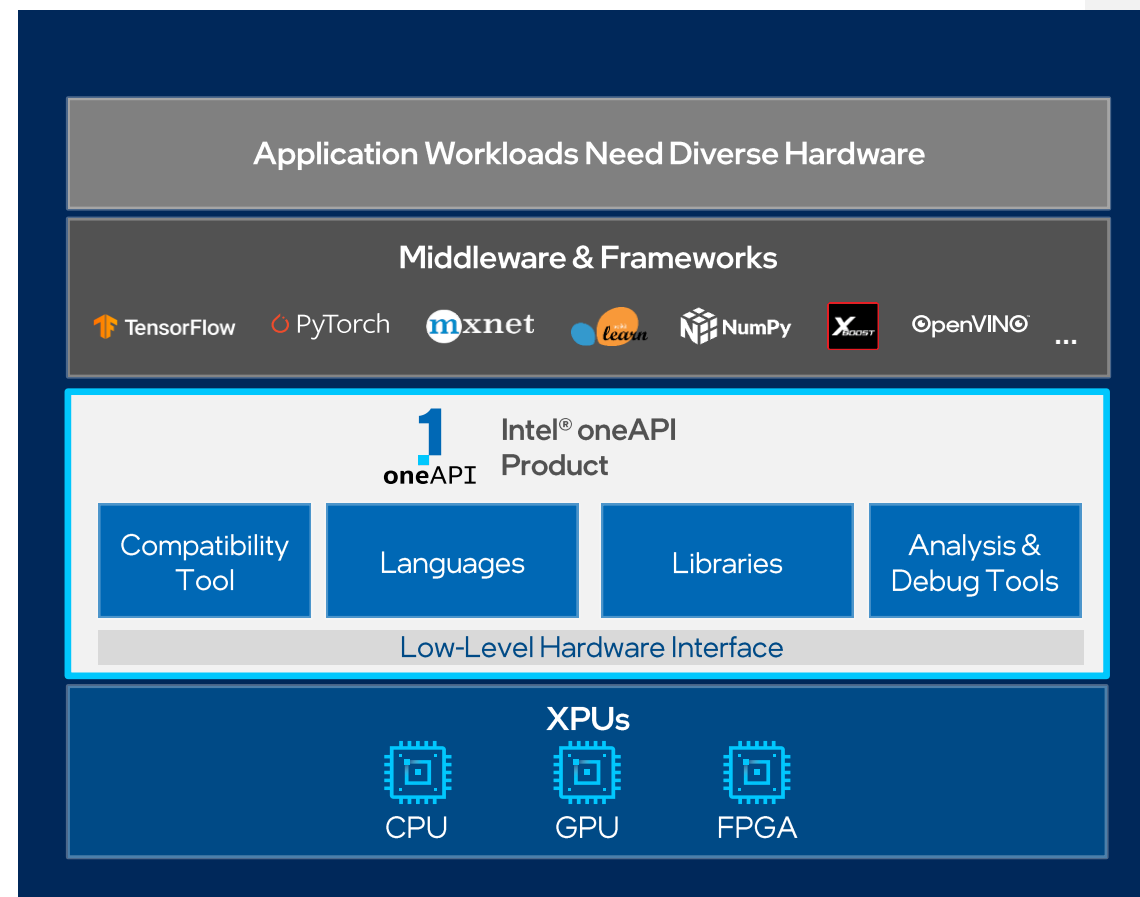
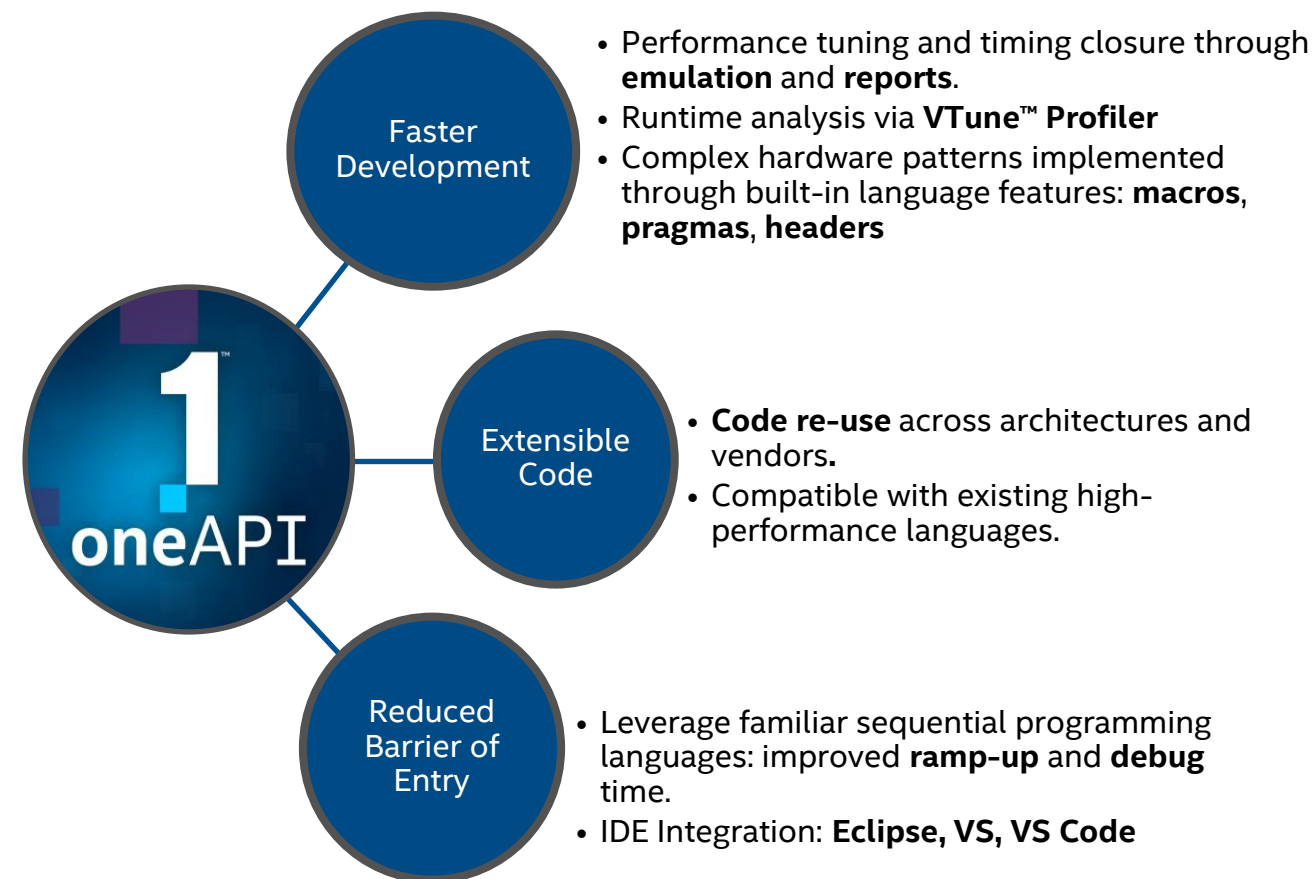
The **oneAPI** product delivers a unified programming model to simplify development across diverse architectures.

It guarantees:

- **Common developer experience** across Scalar, Vector, Matrix and Spatial architectures (CPU, GPU, AI and FPGA)
- Uncompromised native high-level language performance
- Industry standardization and open specifications

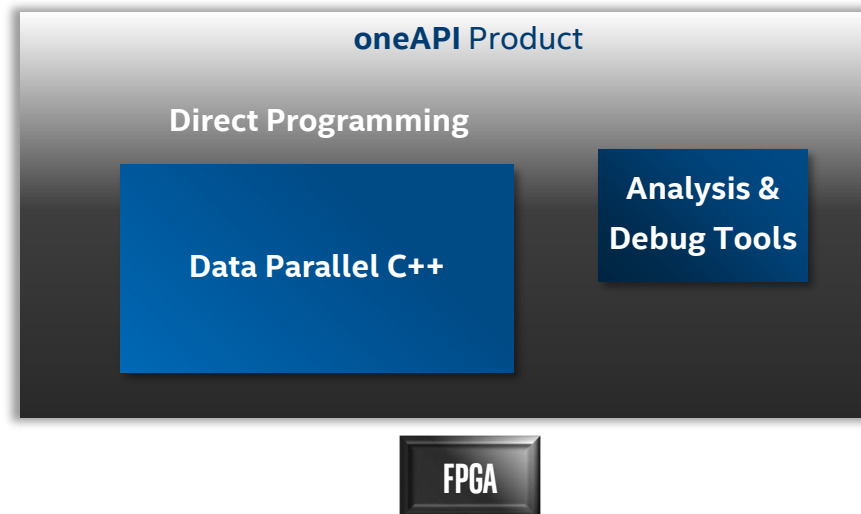
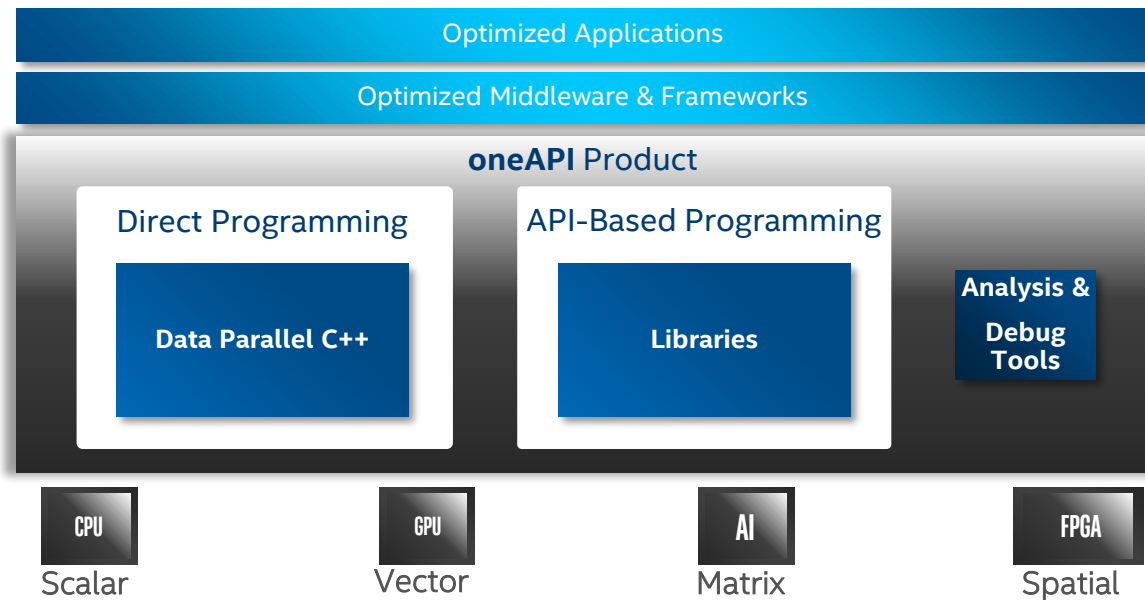


# Intel® oneAPI Product

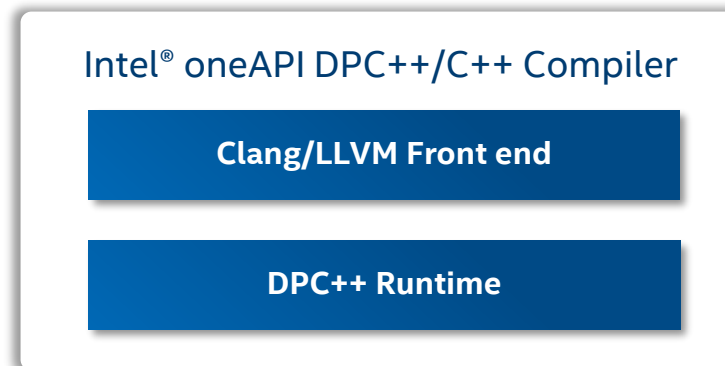
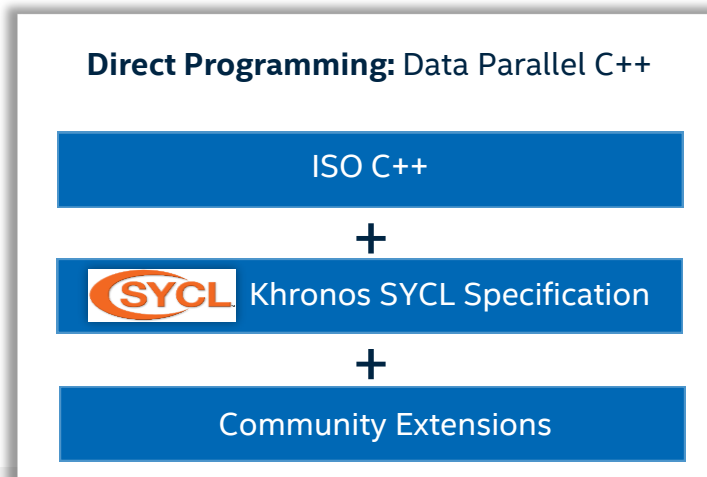


[Available Now](#)

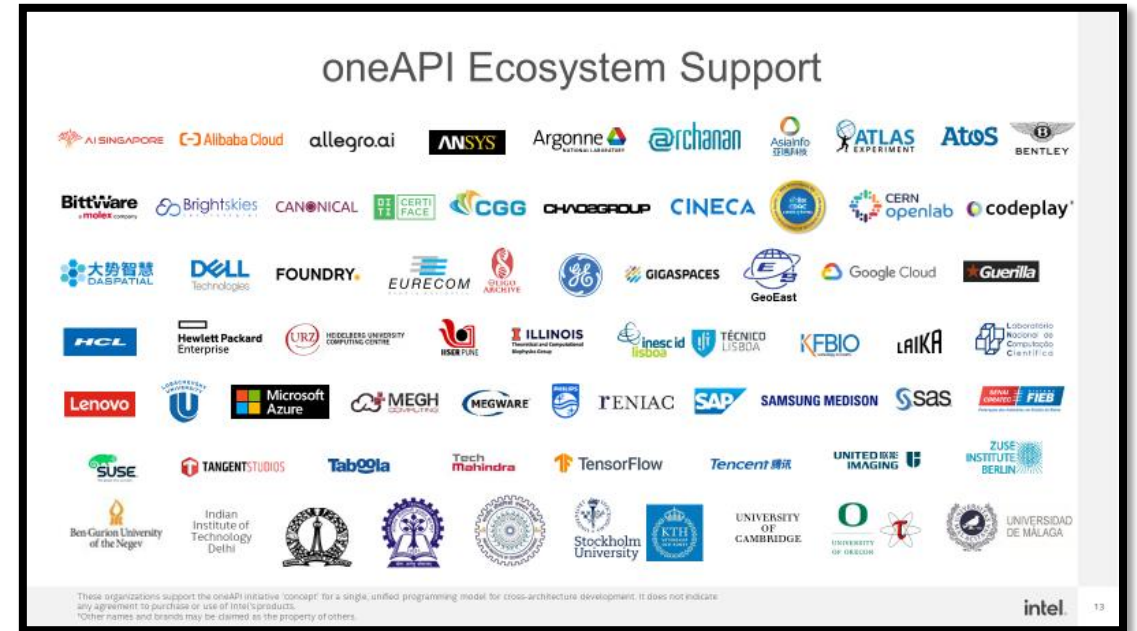
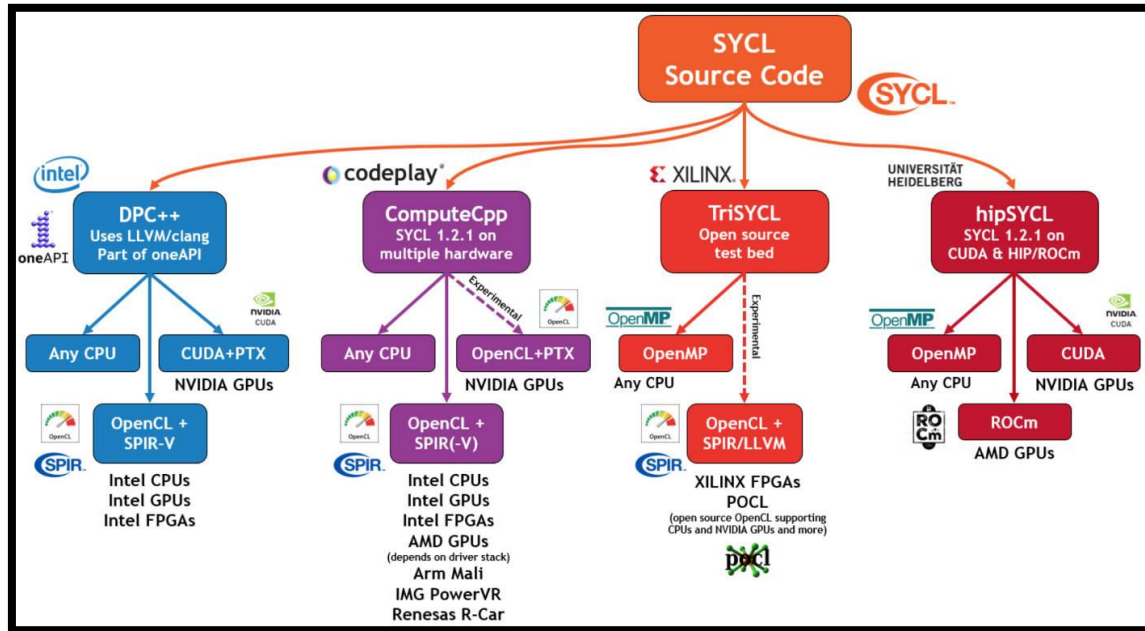
# Intel® oneAPI Base Toolkit for FPGAs



Source Code

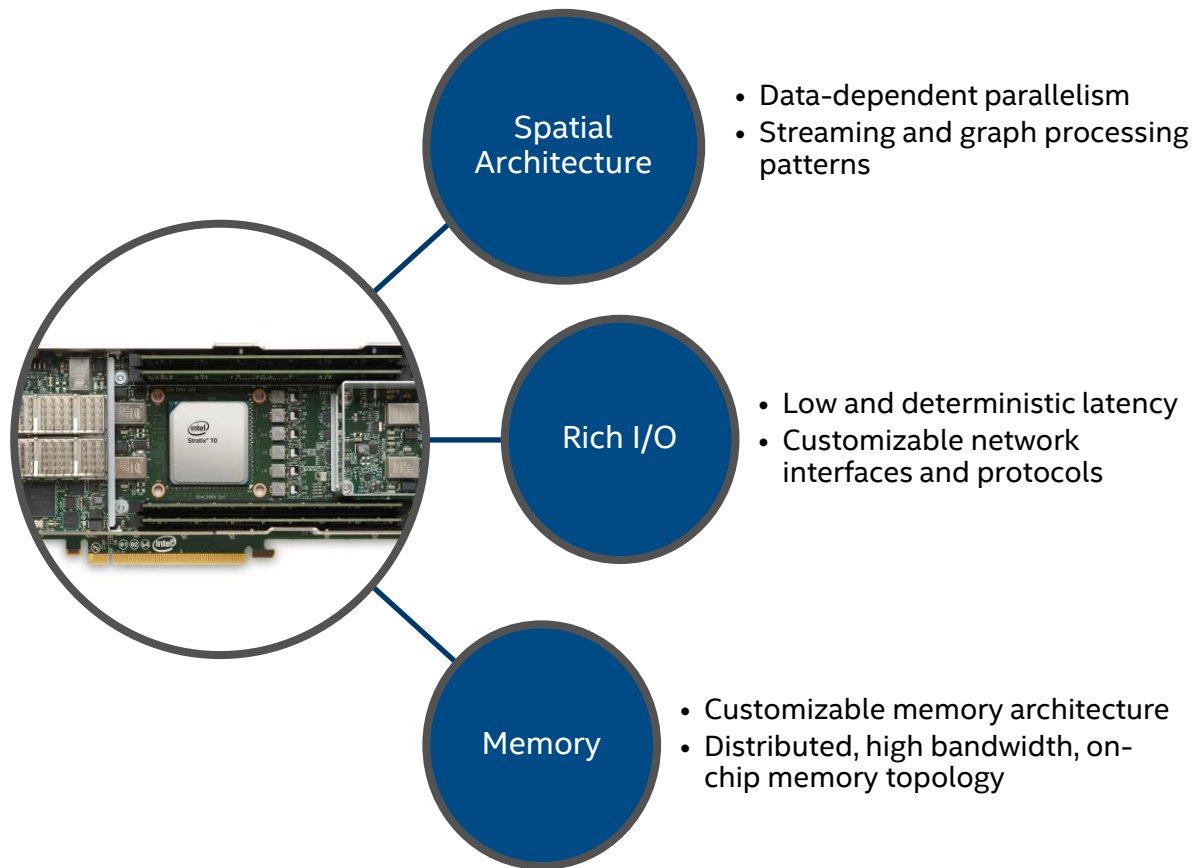


# SYCL and oneAPI Product Ecosystem

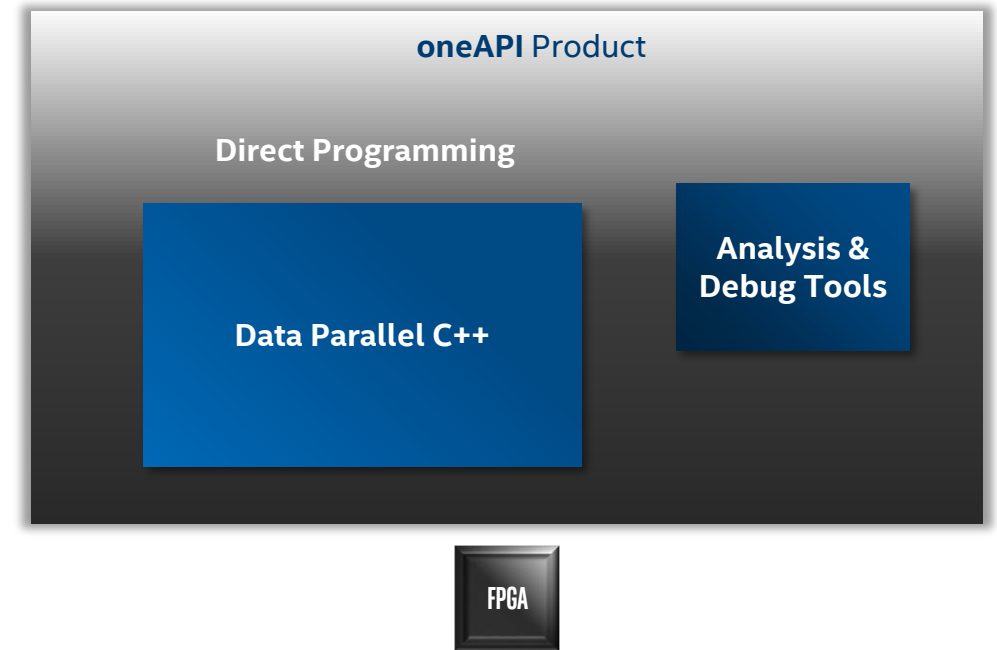


Source: <https://www.khronos.org/sycl/>

# Intel® FPGAs + Intel® oneAPI Toolkits



+



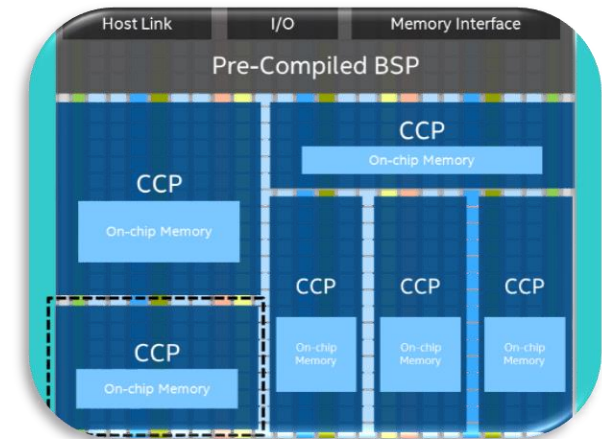
# Getting Started with oneAPI on an FPGA



Intel® oneAPI Base Toolkit



Intel® FPGA Add-on for oneAPI Base Toolkit



Board Support Package (BSP)

**Note:** Developers using custom platforms should [download](#) the Intel® FPGA Add-on for Intel® Custom Platforms with the respective Intel® Quartus® version and obtain a BSP from their 3<sup>rd</sup> part platform vendor.



# Section: Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits

## Sub-Topics:

- Introduction to oneAPI
- **Introduction to DPC++**
- What are FPGAs and Why Should I Care About Programming Them?
- Development Flow for Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits

# Data Parallel C++ (DPC++)

- Based on C++ and SYCL
  - SYCL is based on OpenCL
  - Think of it as SYCL + extensions
- Allows for single-source targeting of accelerators
  - (Doesn't require multiple files)
- Open specification
- Common language meant to target all XPUs
  - You do still need to “tune”
- Goal is for the language to incorporate everything needed to get the best performance out of every architecture



# DPC++: Three Scopes

- DPC++ Programs consist of 3 scopes:
  - **Application scope** - Normal host code
  - **Command group scope** - Submitting data and commands that are for the accelerator
  - **Kernel scope** – Code executed on the accelerator
- The full capabilities of C++ are available at application and command group scope
- At kernel scope there are limitations in accepted C++
  - Most important is no recursive code
  - See SYCL specification for complete list

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
    // Set up a DPC++ device queue  
    queue q(selector);  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

**Application Scope**

**Command Group Scope**

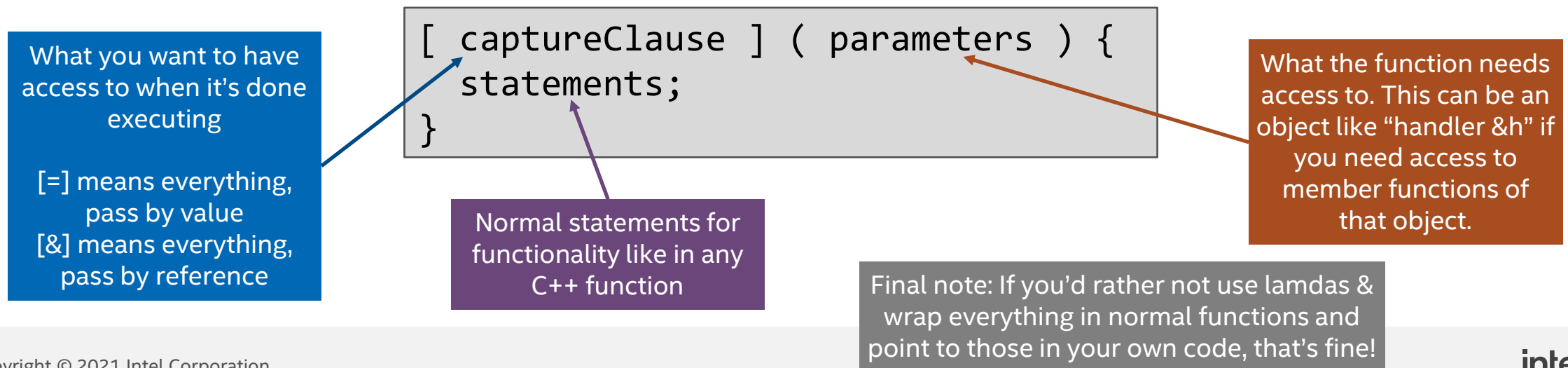
**Kernel Scope**

# The “Runtime”

- The DPC++/SYCL runtime is the program running in the background to control the execution and data passing needs of the heterogeneous compute execution
- It handles:
  - Kernel and host execution in an order imposed by data dependency needs (discussed later)
  - Passing data back and forth between the host and device
  - Querying the device
  - Etc.

# A Note About Lambda Functions

- Two common constructs in DPC++: (1) Queue submissions and (2) kernel dispatch functions take **function pointers** as arguments
- This doesn't lend itself to simple, in-line code
- To write examples that are short and simple, we use **Lambda functions**
- **Lambda functions** are un-named functions used in-line with other code
- If you are not familiar with them, here is a simple guide



# DPC++ Class: `device`

- The `device` class represents the accelerators in a oneAPI system
- The `device` class contains member functions for querying information about the device
- The function `get_info` gives information about the device:
  - Name, vendor, and version of the device
  - Width for built in types, clock frequency, cache width and sizes, online or offline

```
// Get all of the devices a system is capable of operating
std::vector<device> my_devices = device::get_devices();
// Grab the first device to print info out for
device my_device = my_devices[0];
// Print the name of the first device
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```

# DPC++ Class: `device_selector`

- The `device_selector` enables the selection of a device to execute kernels on
- Use the selector when you create a `queue` (covered next)
- The code sample shows use of several example device selectors, including an FPGA

```
// Other example selectors that are not FPGAs
// default_selector selector;
// host_selector selector;
// cpu_selector selector;
// gpu_selector selector;

// Create the selector as an fpga_selector type
INTEL::fpga_selector selector;

// Use the selector when you create a queue
queue q(selector);
```

# DPC++ Class: `queue`

- A `queue` is a mechanism where work is submitted to a `device`
- A queue submits command groups to be executed by the SYCL runtime
- A `queue.submit()` is the beginning of the command scope
  - Groups of work to be executed by the SYCL runtime on an accelerator
- A queue maps to a single device

```
// Declare a queue to a device
queue q(selector);

// Submit things to the queue
q.submit([&](handler& h) {
    // COMMAND GROUP CODE
});
```

The handler is a class that contains all of the command group functions of SYCL

You can think of it as an abstraction of the runtime

This keeps us from having to type `handler::` again and again in the command group scope

# DPC++ Class: `kernel`

- The `kernel` encapsulates code that will be run on the accelerator
- A `kernel` object is not explicitly constructed by the user
- It is constructed when a kernel dispatch function, such as `parallel_for()` or `single_task()` is called

```
q.submit([&](handler& h) {  
  
    // The "kernel" is everything after the kernel dispatch function  
    h.single_task<VectorAdd>([=]() {  
  
        // Everything inside here is "KERNEL SCOPE"  
        for (int i = 0; i < kSize; ++i) {  
            c[i] = a[i] + b[i];  
        }  
    });
```

# 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
- Look like CPU code
  - Contain an outer loop to process all data
- Ideal for & recommended for **FPGAs**

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

**CPU  
Implementation**

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

**FPGA Kernel  
Implementation**



# Parallel Kernels

- Parallel Kernels allow multiple instances of an operation to execute in parallel
- Parallel kernels are expressed using the `parallel_for()` function
- Kernels that are easily expressed in this way do well on **GPUs**
  - Will be functional in an FPGA, but usually result in a less optimal implementation

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

**CPU  
Implementation**

**GPU Kernel  
Implementation**

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

# DPC++ Class: `buffer` and `accessor`

## ■ `buffer`

- Encapsulates data in a SYCL application
- Across both devices and host!

## ■ `accessor`

- Mechanism to access buffer data
- Determines data dependencies in that order kernel executions (covered later)

```
int main() {
    ... // Code to set up standard C++ vectors

    buffer buf_a(vector_a);
    buffer buf_b(vector_b);
    buffer buf_c(vector_c);

    queue q(selector);

    q.submit([&](handler& h) {
        accessor a(buf_a, h, read_only);
        accessor b(buf_b, h, read_only);
        accessor c(buf_c, h, write_only);

        h.single_task<VectorAdd>([=]() {
            for (int i = 0; i < kSize; i++) {
                c[i] = a[i] + b[i];
            }
        });
    });
};
```

# #Include Files

- oneAPI programs require the include of `cl/sycl.hpp`
- Programs targeting FPGAs require the include of `cl/sycl/INTEL/fpga_extensions.hpp`

```
// Always include these at the top of your program
```

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

```
#include <CL/sycl/INTEL/fpga_extensions.hpp>
```

# DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

# DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
    // Set up a DPC++ device queue  
    queue q(selector);  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

# DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

# DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

# DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data



# DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

Step 6: Send a kernel for execution

# DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

**Step 1:** Create a device selector targeting the FPGA

**Step 2:** Create a device queue, using the FPGA device selector

**Step 3:** Create buffers

**Step 4:** Submit a command for execution

**Step 5:** Create buffer accessors so the FPGA can access the data

**Step 6:** Send a kernel for execution

**Done!**

The contents of buf\_c are copied to \*c when the function finishes

(because of the buffer destruction of buf\_c)

```

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

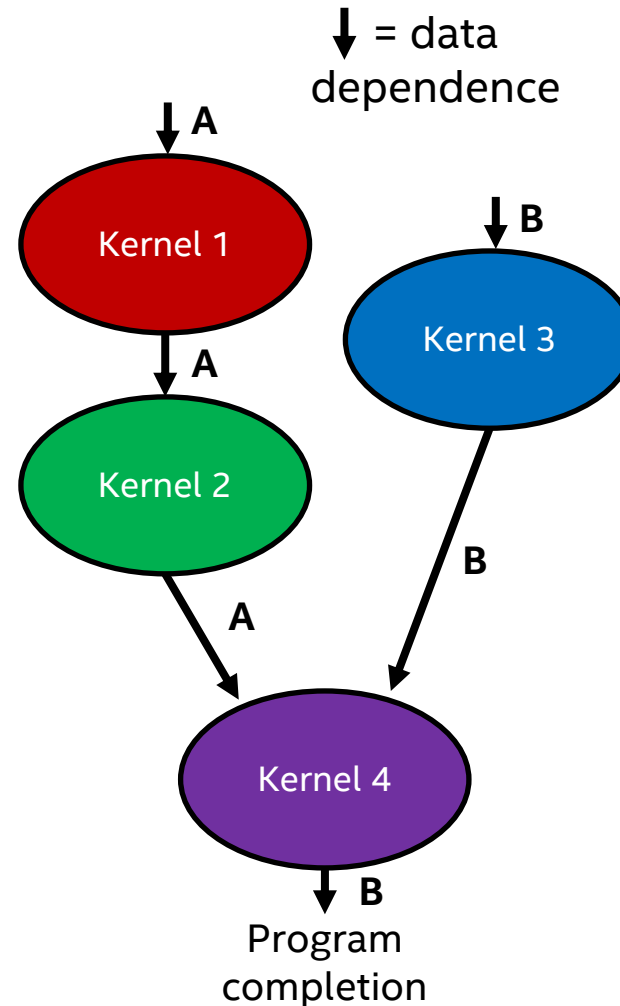
  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); }); } Kernel 2

  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
}

```

# Kernel Execution Order

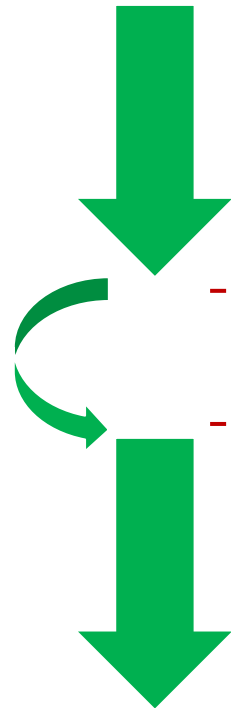


- Kernels can execute at the same time
  - If no data dependencies
- Accessors are used to determine dependencies
- Execution ordering is automatically determined

# Asynchronous Host/Kernel Execution

- The execution of the host code is asynchronous to what is being executed on the accelerator
- If you need synchronization, you must impose that yourself

Host code execution



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

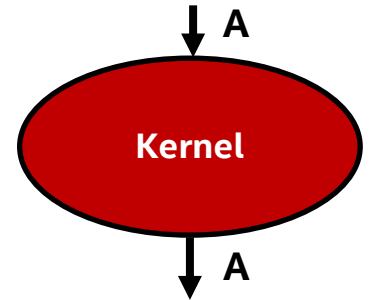
int main() {
    auto R = range<1>{ num };
    buffer<int> A{ R };

    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;
}
```

Kernel Execution



# Synchronization Method 1: Host Accessor

- In the **command scope**, accessors are created for the accelerator

- In the **application scope**, accessors are created for the host

- A host accessor creates a dependency node in the execution graph
  - Execution at the host is blocked until the data is ready

```
int main() {  
    constexpr int N = 100;  
    auto R = range<1>(N);  
    std::vector<double> v(N, 10);  
    queue q;
```

```
    buffer<double, 1> buf(v.data(), R);  
    q.submit([&](handler& h) {  
        auto a = buf.get_access<access::mode::read_write>(h);  
        h.parallel_for(R, [=](id<1> i) {  
            a[i] -= 2;  
        });  
    });
```

**Command Scope**

```
    auto b = buf.get_access<access::mode::read>();  
    for (int i = 0; i < N; i++)  
        std::cout << v[i] << "\n";  
    return 0;
```

**Application Scope**

```
}
```

# Synchronization Method 2: Buffer Destruction

- Buffer creation happens within a separate function scope
- When execution advances beyond this function scope, buffer destructor is invoked
- Relinquishes ownership of data and copies back the data to the host memory
- Scope can also be created with simple use of { }

```
#include <CL/sycl.hpp>
constexpr int N=100;
using namespace cl::sycl;

void dpcpp_code(std::vector<double> &v, queue &q){
    auto R = range<1>(N);
    buffer<double, 1> buf(v.data(), R);
    q.submit([&](handler& h) {
        auto a = buf.get_access<access::mode::read_write>(h);
        h.parallel_for(R, [=](id<1> i) {
            a[i] -= 2;
        });
    });
}

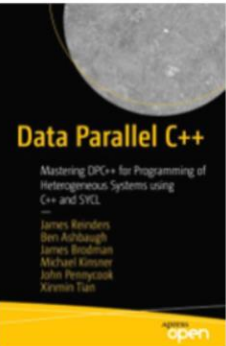
int main() {
    std::vector<double> v(N, 10);
    queue q;
    dpcpp_code(v,q);
    for (int i = 0; i < N; i++)
        std::cout << v[i] << "\n";
    return 0;
}
```

# Learn More About DPC++

- Download DPC++ book for free
  - <https://link.springer.com/book/10.1007%2F978-1-4842-5574-2>

- DPC++ Training Modules

- [https://devcloud.intel.com/oneapi/get\\_started/baseTrainingModules/](https://devcloud.intel.com/oneapi/get_started/baseTrainingModules/)



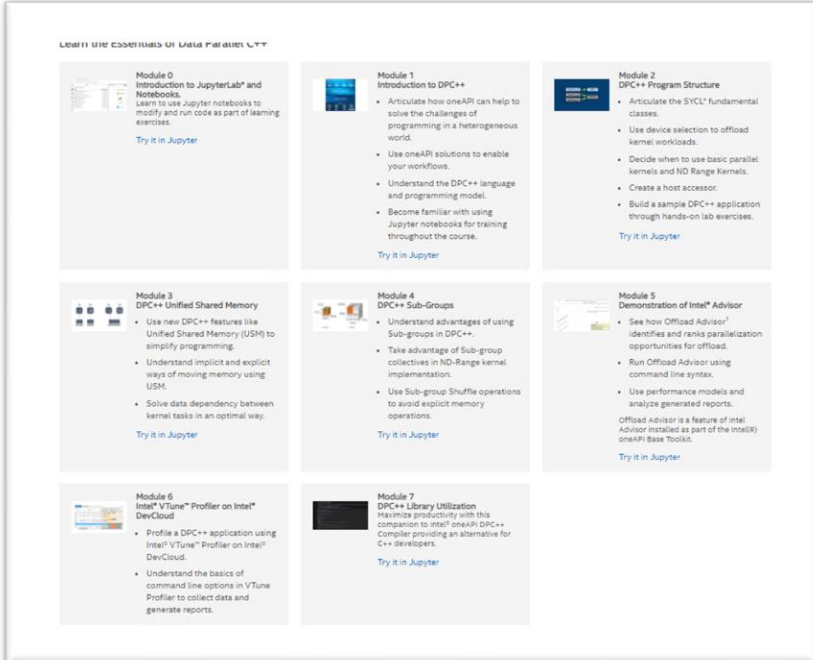
**Data Parallel C++**  
Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Authors [\(view affiliations\)](#)  
James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, Xinmin Tian

Open Access | Book

1 Citations | 21 Mentions | 207k Downloads

[Download book PDF](#) [Download book EPUB](#)



LEARN THE ESSENTIALS OF DATA PARALLEL C++

- Module 0** Introduction to JupyterLab® and Notebooks.  
Learn to use Jupyter notebooks to modify and run code as part of learning exercises.  
Try it in Jupyter
- Module 1** Introduction to DPC++  
• Articulate how oneAPI can help to solve the challenges of programming in a heterogeneous world.  
• Use oneAPI solutions to enable your workflows.  
• Understand the DPC++ language and programming model.  
• Become familiar with using Jupyter notebooks for training throughout the course.  
Try it in Jupyter
- Module 2** DPC++ Program Structure  
• Articulate the SYCL® fundamental classes.  
• Use device selection to offload kernel workloads.  
• Decide when to use basic parallel kernels and ND Range Kernels.  
• Create a host accessor.  
• Build a sample DPC++ application through hands-on lab exercises.  
Try it in Jupyter
- Module 3** DPC++ Unified Shared Memory  
• Use new DPC++ features like Unified Shared Memory (USM) to simplify programming.  
• Understand implicit and explicit ways of moving memory using USM.  
• Solve data dependency between kernel tasks in an optimal way.  
Try it in Jupyter
- Module 4** DPC++ Sub-Groups  
• Understand advantages of using Sub-groups in DPC++.  
• Take advantage of Sub-group collectives in ND-Range kernel implementation.  
• Use Sub-group Shuffle operations to avoid explicit memory operations.  
Try it in Jupyter
- Module 5** Demonstration of Intel® Advisor  
• See how Offload Advisor<sup>1</sup> identifies and ranks parallelization opportunities for offload.  
• Run Offload Advisor using command line syntax.  
• Use performance models and analyze generated reports.  
Offload Advisor is a feature of Intel Advisor installed as part of the Intel® oneAPI Base Toolkit.  
Try it in Jupyter
- Module 6** Intel® VTune™ Profiler on Intel® DevCloud  
• Profile a DPC++ application using Intel® VTune™ Profiler on Intel® DevCloud.  
• Understand the basics of command line options in VTune Profiler to collect data and generate reports.
- Module 7** DPC++ Library Utilization  
Maximize productivity with this companion to Intel® oneAPI DPC++ Compiler providing an alternative for C++ developers.  
Try it in Jupyter



# Section: Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits

## Sub-Topics:

- Introduction to oneAPI
- Introduction to DPC++
- **What are FPGAs and Why Should I Care About Programming Them?**
- Development Flow for Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits



# “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

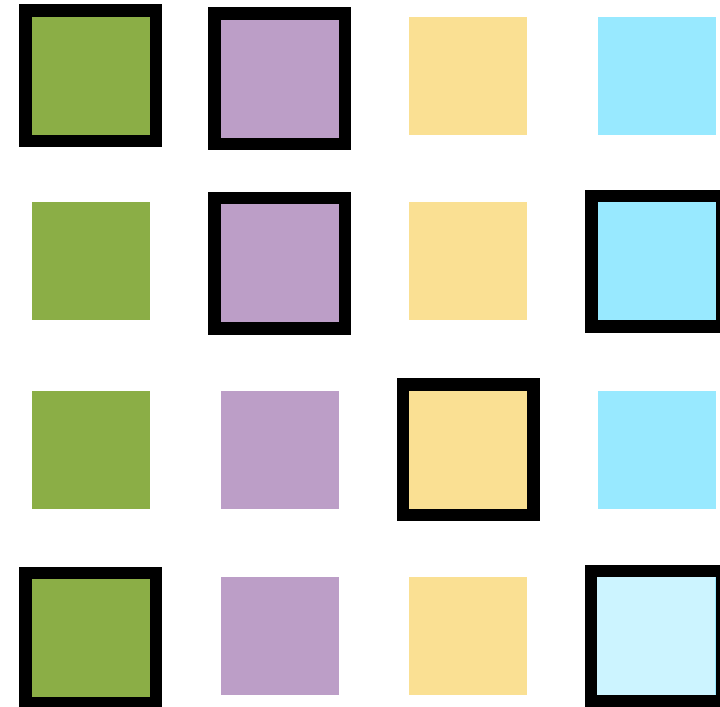
The FPGA is made up of small building blocks of logic and other functions



# How an FPGA Becomes What You Want It To Be

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

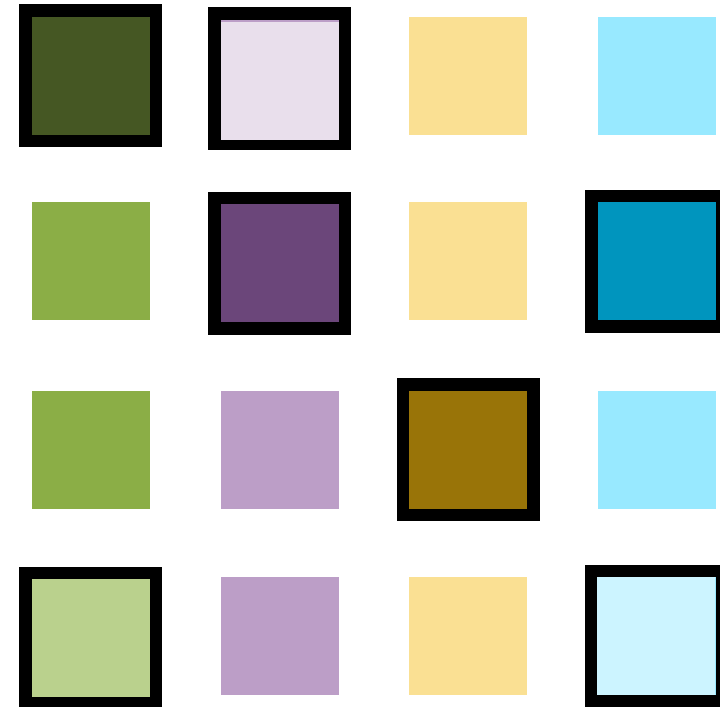
- You choose building blocks



# How an FPGA Becomes What You Want It To Be

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

- You choose building blocks
- You configure those building blocks

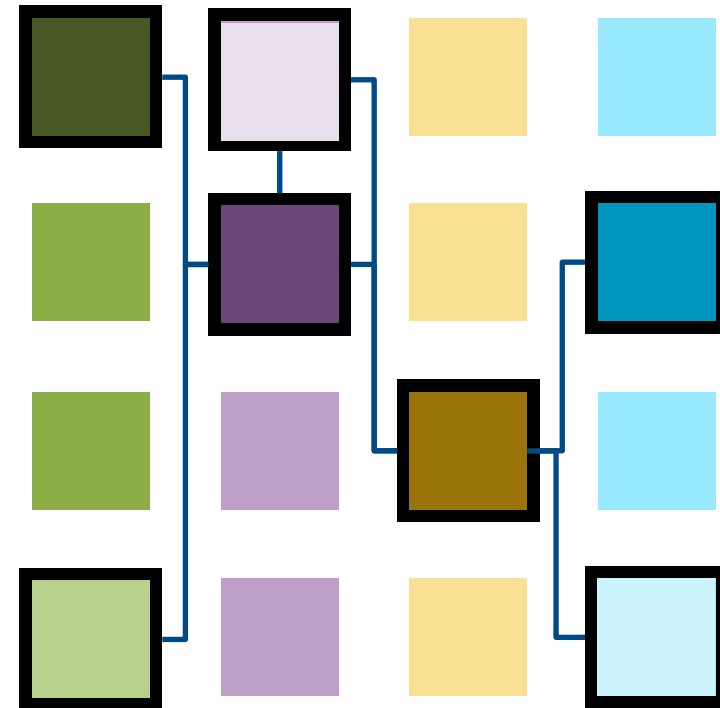


# How an FPGA Becomes What You Want It To Be

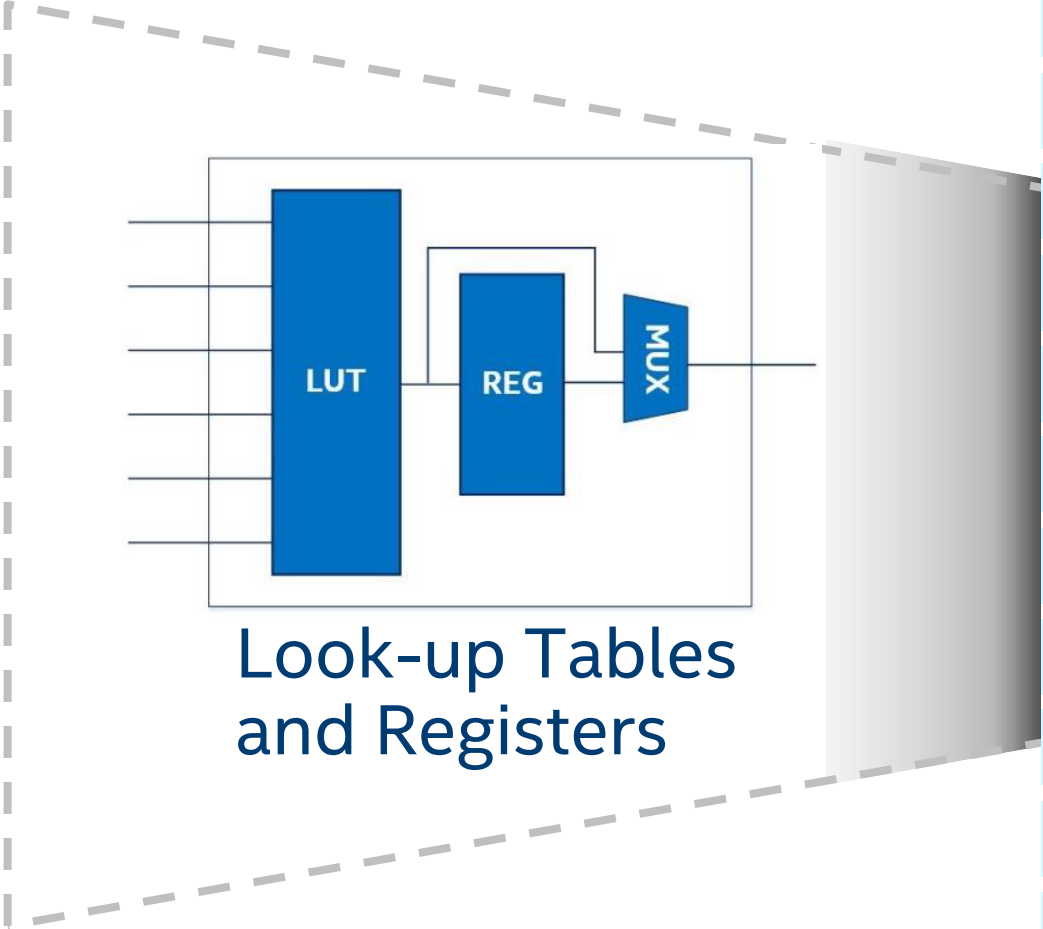
The FPGA is made up of small building blocks of logic and other functions

- You choose building blocks
- You configure the building blocks
- You connect the building blocks

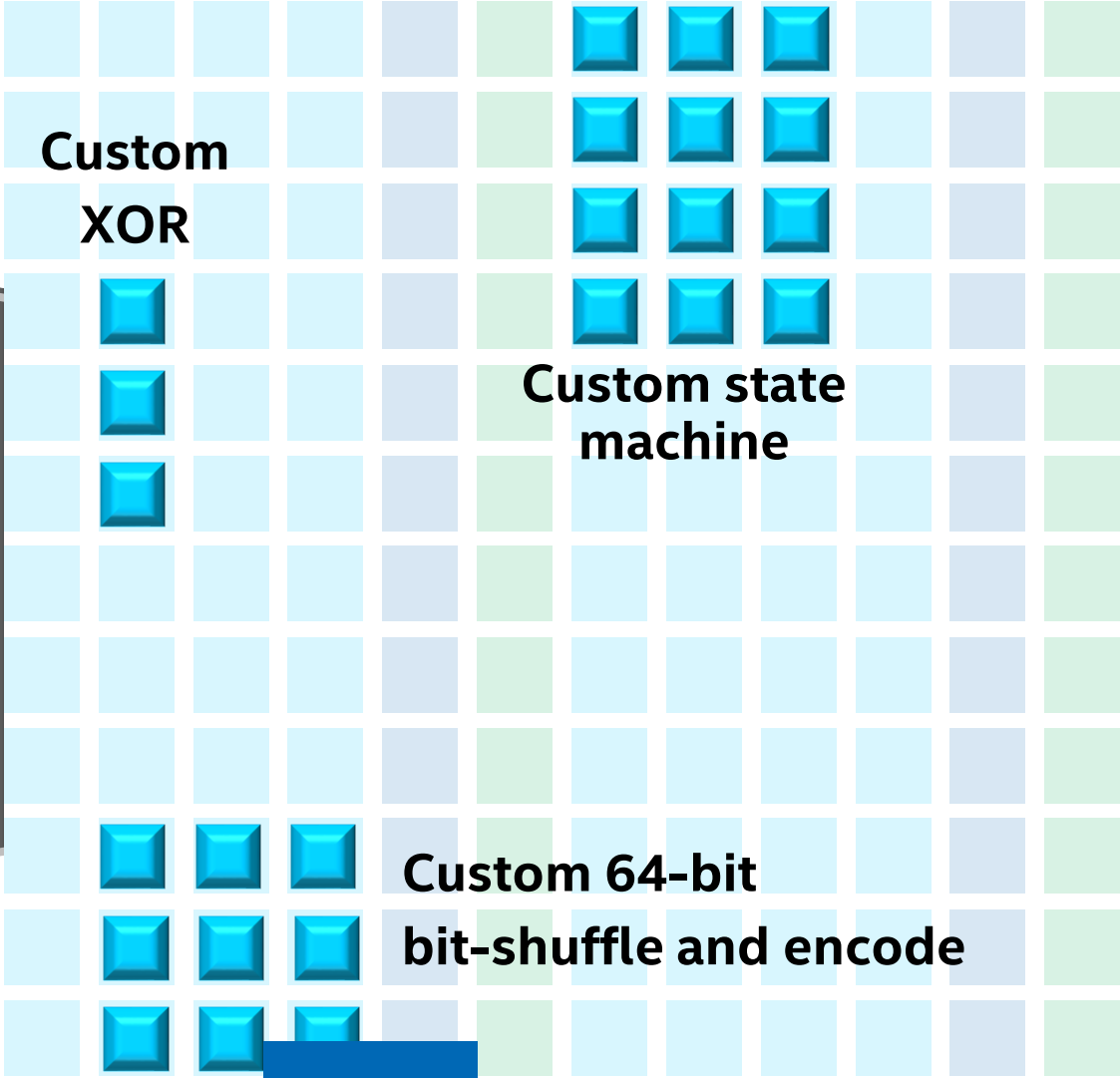
These custom choices determine the FPGA's functionality



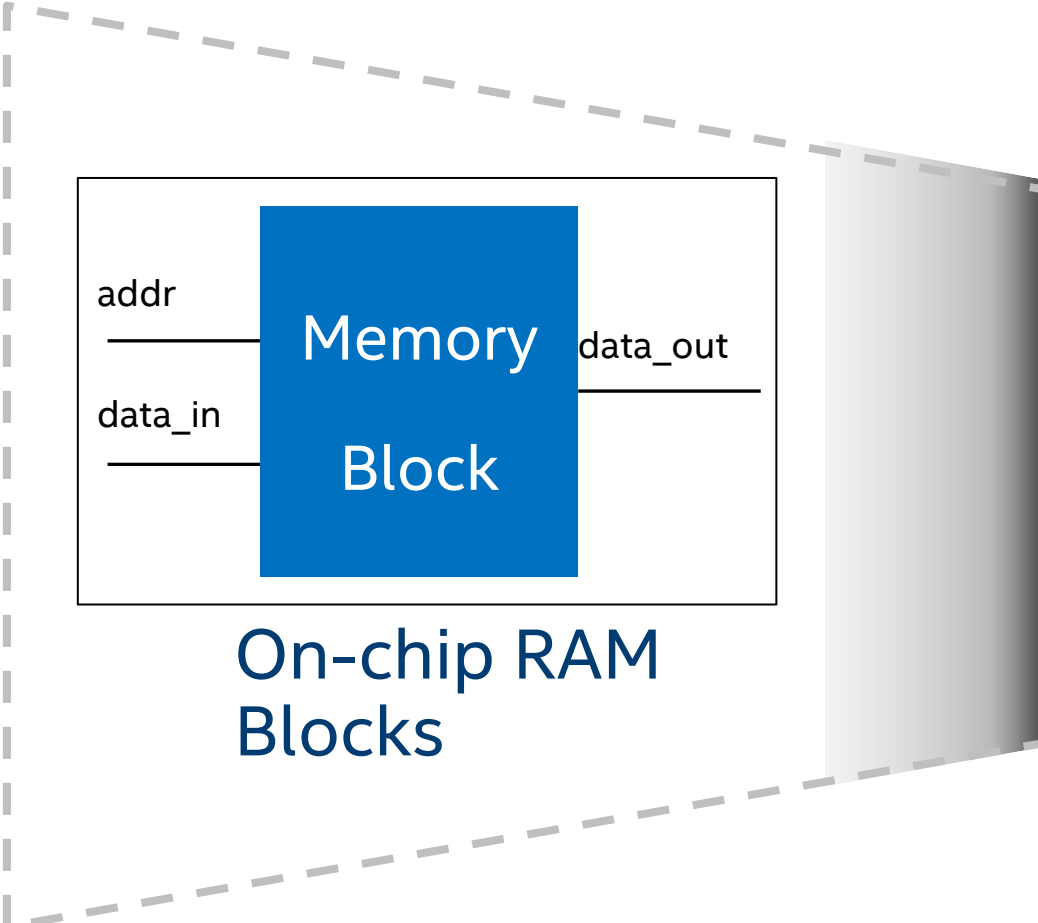
# Blocks Used to Build What You've Coded



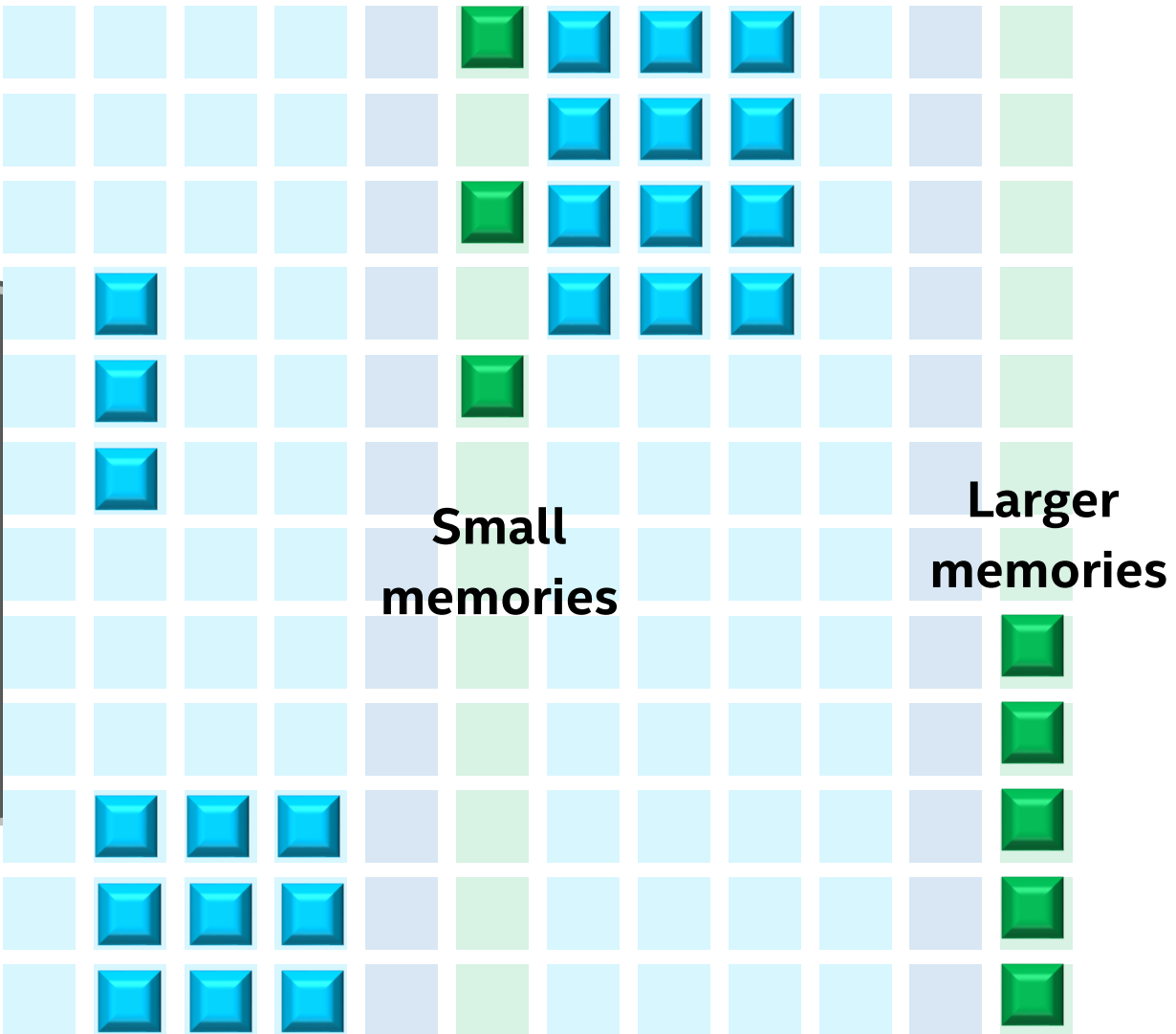
Look-up Tables and Registers



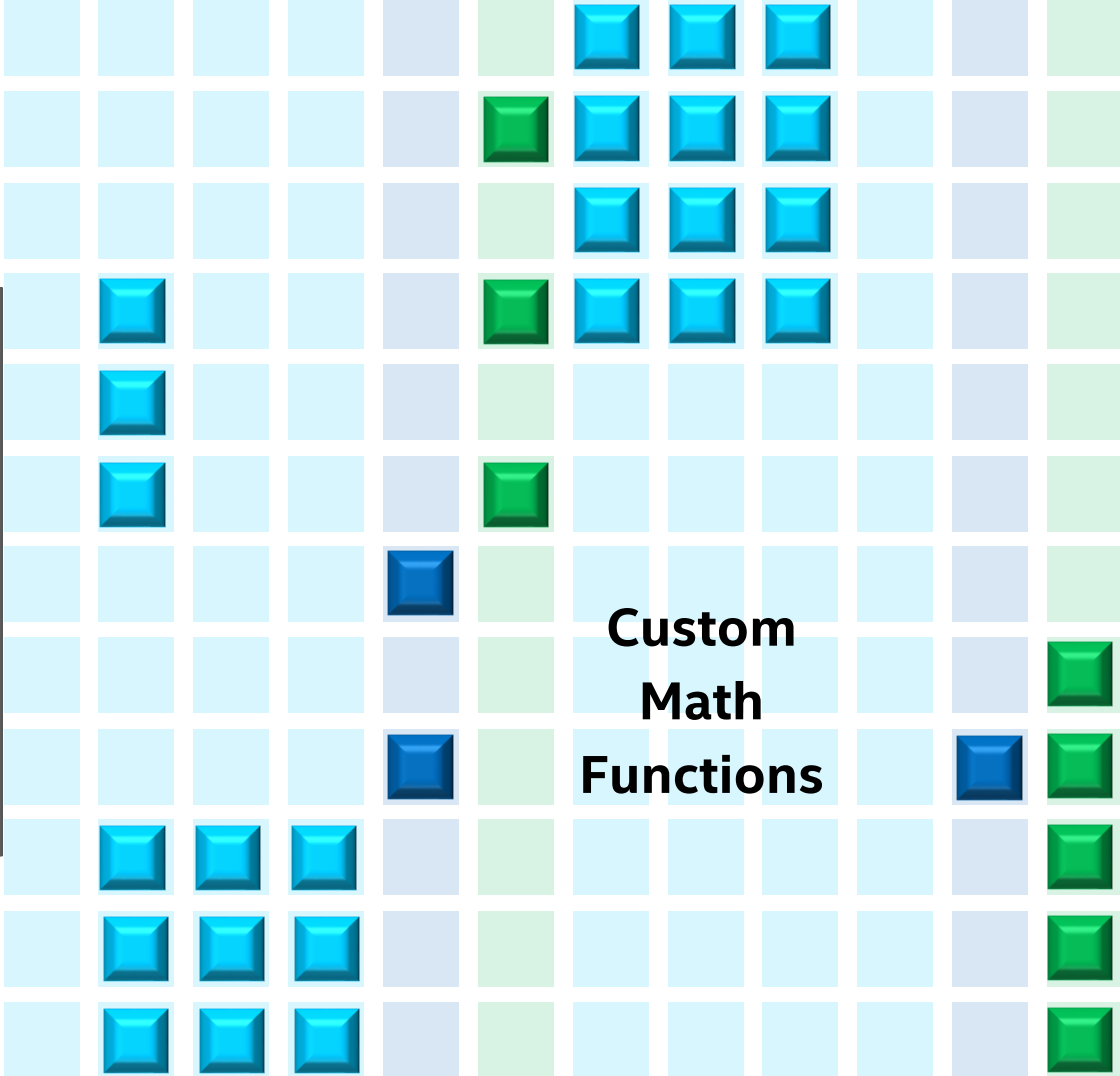
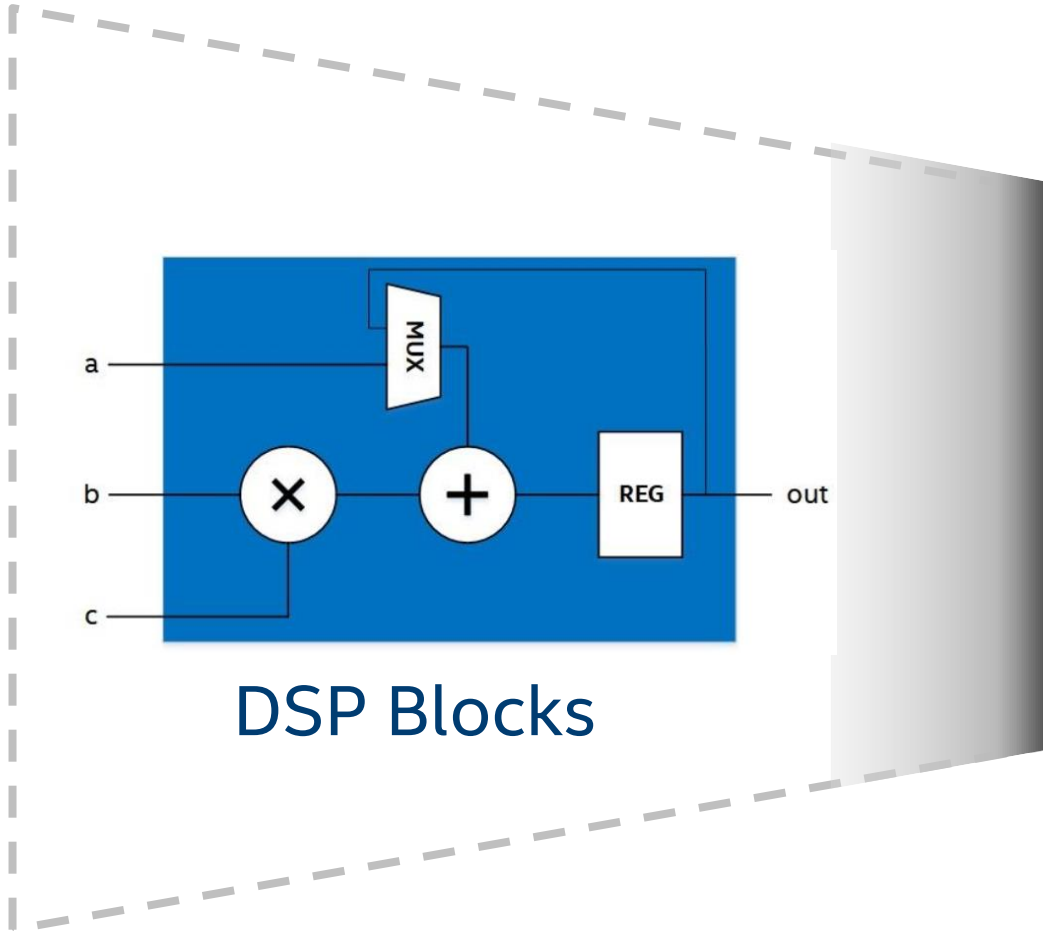
# Blocks Used to Build What You've Coded



On-chip RAM Blocks



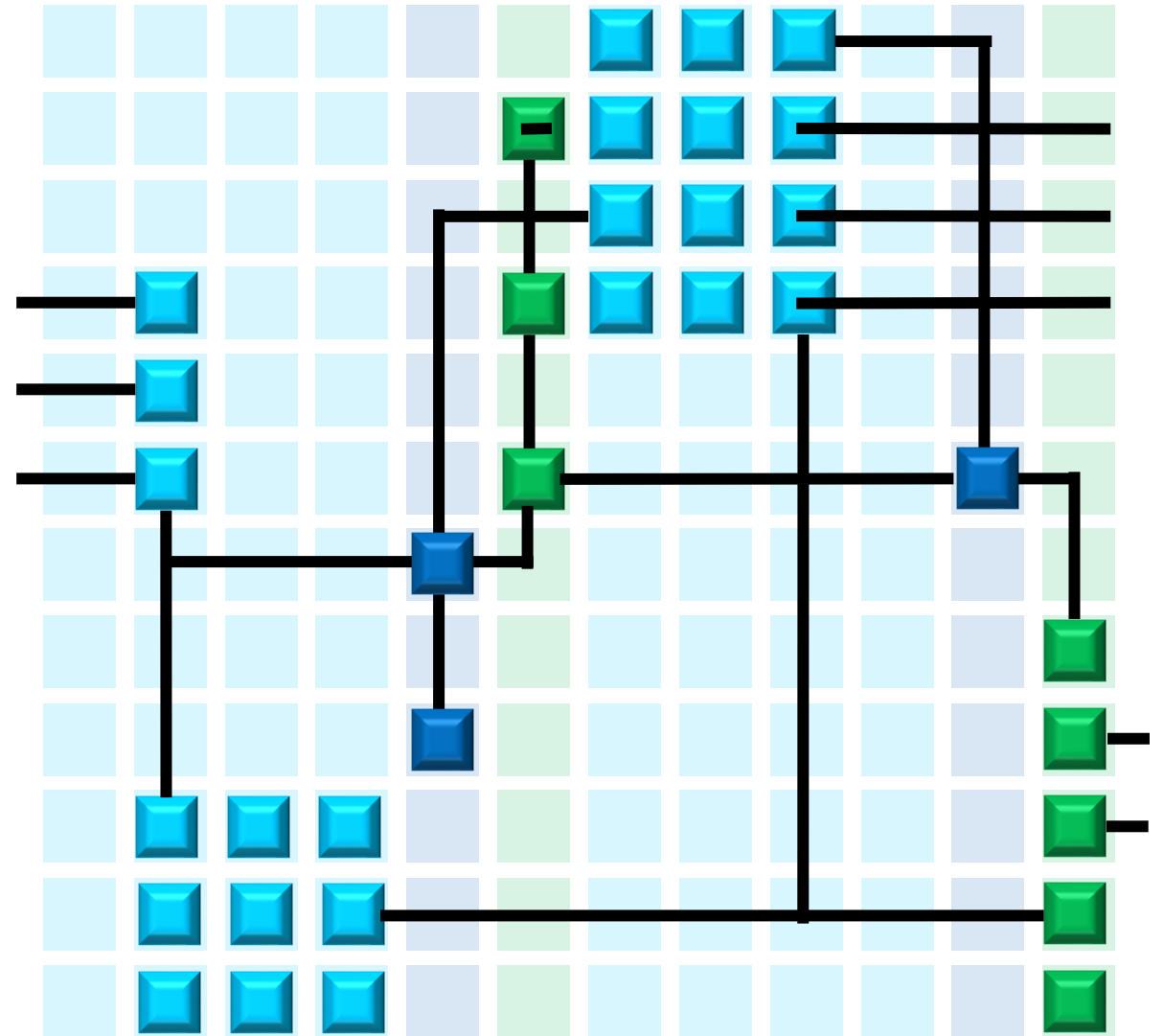
# Blocks Used to Build What You've Coded





# Then, It's All Connected Together

Blocks are connected with [custom routing](#) determined by your code

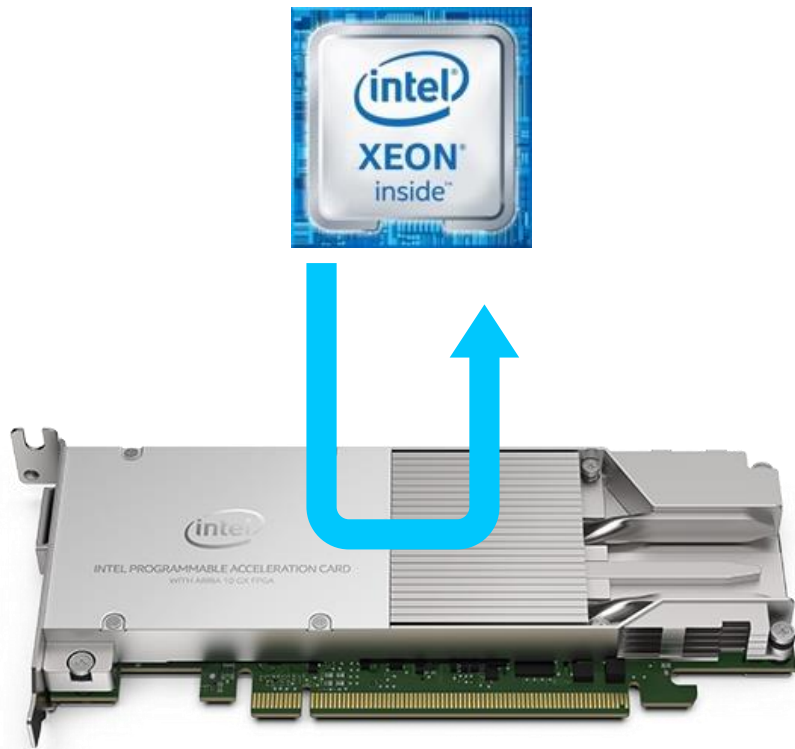


# What About Connecting to the Host?

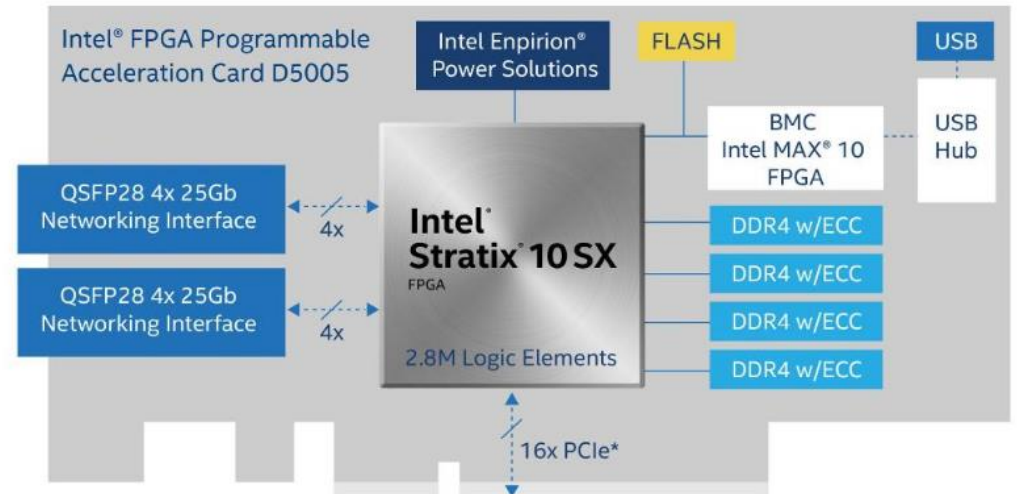
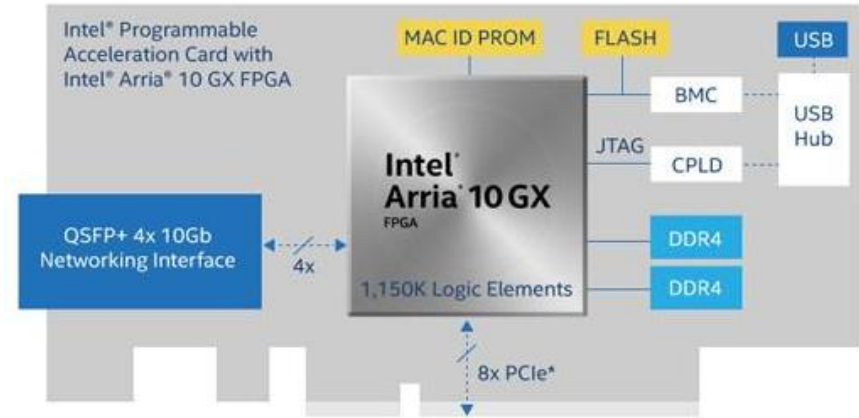
Accelerated functions run on a PCIe attached FPGA card

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 FPGA Cards Available to use with oneAPI



Why should I care about programming for an FPGA?

It all comes down to the advantage of custom hardware.

**IMAGE LOSSLESS COMPRESSION:  
ACCELERATING PERFORMANCE**

**GENOMICS SEQUENCING**



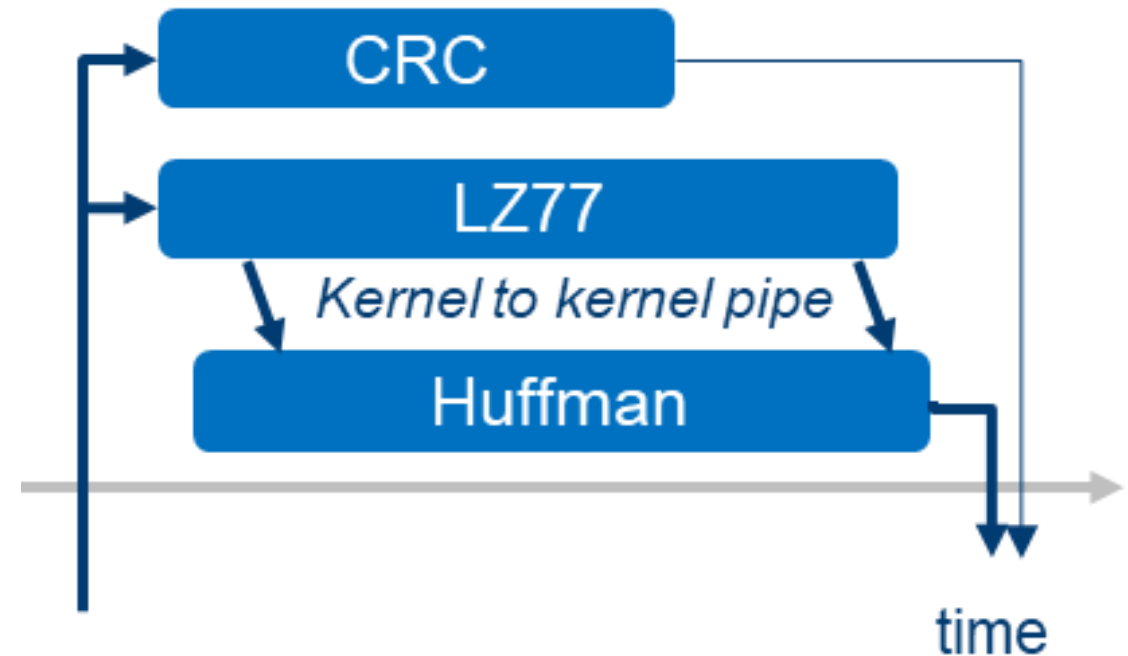
Sample FPGA Workloads

**KEY VALUE STORE  
ACCELERATING THROUGHPUT**

**DATABASE ANALYTICS  
ACCELERATION**

# Gzip Compression

- FPGA Example included with the Intel<sup>®</sup> oneAPI Base Toolkit
- Concurrent kernels for LZ77, Huffman encoding and CRC
- You are encouraged to try it for yourself!





# Section: Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits

## Sub-Topics:

- Introduction to oneAPI
- Introduction to DPC++
- What are FPGAs and Why Should I Care About Programming Them?
- **Development Flow for Using FPGAs with the Intel<sup>®</sup> oneAPI Toolkits**

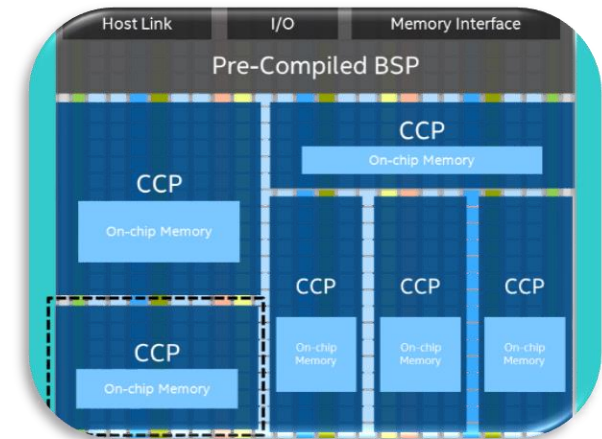
# Getting Started with oneAPI on an FPGA



Intel® oneAPI Base Toolkit



Intel® FPGA Add-on for oneAPI Base Toolkit



Board Support Package (BSP)

**Note:** Developers using custom platforms should [download](#) the Intel® FPGA Add-on for Intel® Custom Platforms with the respective Intel® Quartus® version and obtain a BSP from their 3<sup>rd</sup> part platform vendor.



# Installing oneAPI

- Get started by visiting the Intel<sup>®</sup> Software Developer Zone page for the Intel<sup>®</sup> oneAPI Toolkits
  - <https://software.intel.com/en-us/oneapi>
- Get the Intel<sup>®</sup> 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<sup>®</sup> Quartus<sup>®</sup> 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`

## Get to Know oneAPI Now

No hardware acquisitions, system configurations, or software installations.

### A Fast Way to Start Coding

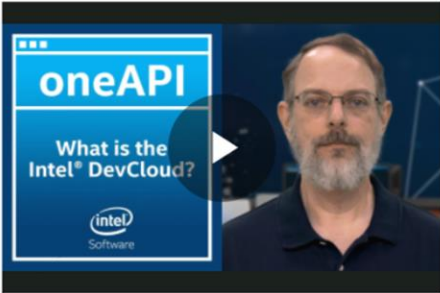
Are you a forward-thinking developer interested in the next generation of data-centric computing innovation?

You've come to the right place.

The Intel DevCloud is a development sandbox to learn about and program oneAPI cross-architecture applications.

Sign up now for full access to the latest Intel CPUs, GPUs, and FPGAs, Intel oneAPI Toolkits, and the new programming language, Data Parallel C++ (DPC++).

Access is free for 120 days with the possibility of an extension.



### Sign Up for Intel® DevCloud for oneAPI

Already have access? [Sign in.](#)

Required Fields(\*)

\* **First Name**

\* **Last Name**

\* **Email Address**

\* **Country / Region**

\* **Company or University**

\* **Which hardware and accelerator architecture are you developing for? (Select all that apply)**

ASICs (application-specific integrated circuits)

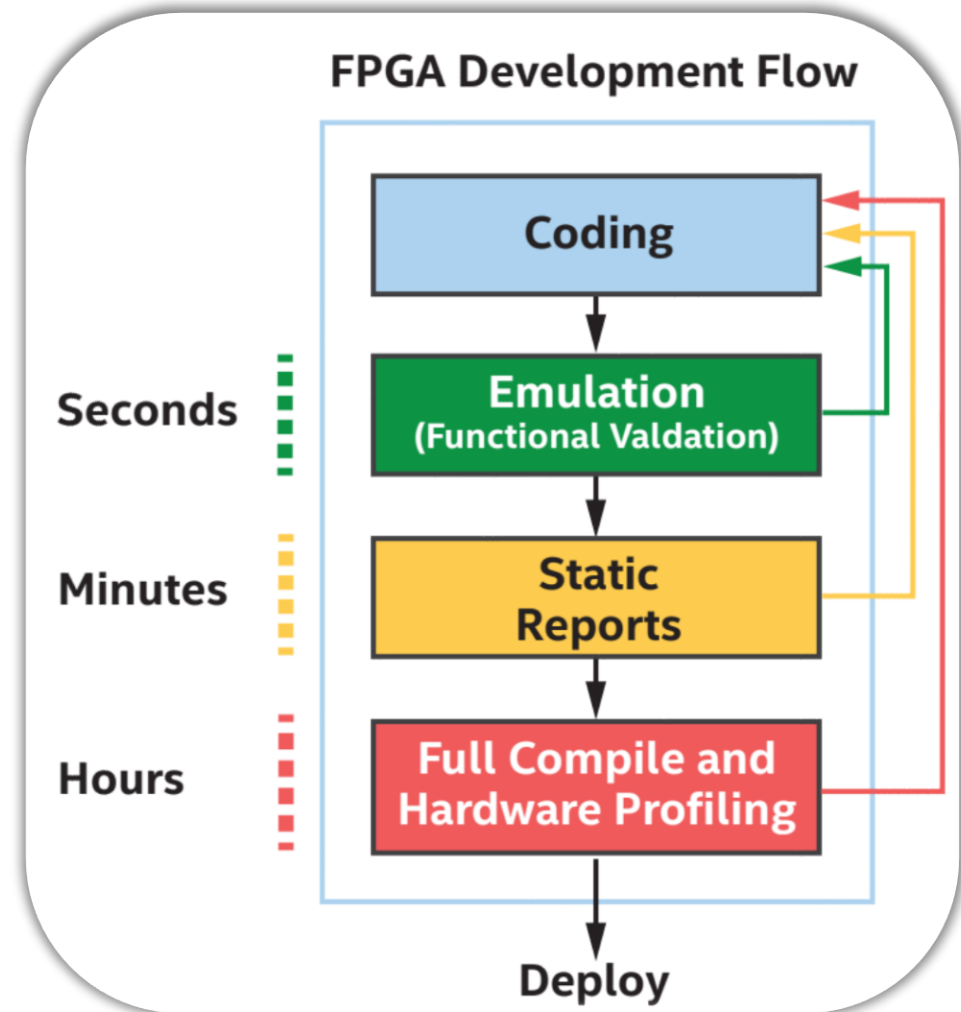
CPU

FPGA (field-programmable gate array)

GPGPU (general-purpose GPU)

# FPGA Development Flow for oneAPI Projects

- FPGA Emulator target (Emulation)
  - Compiles in seconds
  - Runs completely on the host
- Optimization report generation
  - Compiles in seconds to minutes
  - Identify bottlenecks
- FPGA bitstream compilation
  - Compiles in hours
  - Enable profiler to get runtime analysis



# Anatomy of a dpcpp Command Targeting FPGAs

```
dpcpp -fintel fpga *.cpp/*.o [device link options] [-Xs arguments]
```

Target Platform

Link Options

FPGA-Specific Arguments

Language  
DPCPP = Data  
Parallel C++

Input Files  
source or object

# Emulation

## Seconds of Compilation

Does my code give me the correct answers?

Quickly generate code that runs on the x86 host to emulate the FPGA

Developers can:

- Verify functionality of design through CPU compile and emulation.
- Identify quickly syntax and pointer implementation errors for iterative design/algorithm development.
- Enable deep, system-wide debug with Intel® Distribution for GDB.
- Functional debug of SYCL code with FPGA extensions.

# Emulation Command

```
#ifdef FPGA_EMULATOR
    intel::fpga_emulator_selector device_selector;
#else
    intel::fpga_selector device_selector;
#endif
```

Include this construct in  
your code

```
dpcpp -fintel fpga <source_file>.cpp -DFPGA_EMULATOR
```



# Report Generation

## Minutes of Compilation

Where are the bottlenecks?

Quickly generate a report to guide optimization efforts

Developers can:

- Identify any memory, performance, data-flow bottlenecks in their design.
- Receive suggestions for optimization techniques to resolve said bottlenecks.
- Get area and timing estimates of their designs for the desired FPGA.

# Command to Produce an Optimization Report

## Two Step Method:

```
dpcpp -fintel FPGA <source_file>.cpp -c -o <file_name>.o  
dpcpp -fintel FPGA <file_name>.o -fsycl-link -Xshardware
```

## One Step Method:

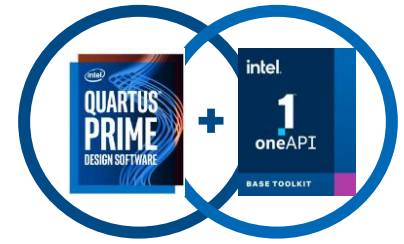
```
dpcpp -fintel FPGA <source_file>.cpp -fsycl-link -Xshardware
```

The default value for `-fsycl-link` is `-fsycl-link=early` which produces an early image object file and report

- A report showing optimization, area, and architectural information will be produced in `<file_name>.prj/reports/`
  - We will discuss more about the report later



# Bitstream Compilation



## Runs Intel Quartus Prime Software “under the hood” (no licensing required)

Developers can:

- Compile FPGA bitstream for their design and run it on an FPGA.
- Attain automated timing closure.
- Obtain In-hardware verification.
- Take advantage of Intel<sup>®</sup> VTune<sup>™</sup> Profiler for real-time analysis of design.

# Compile to FPGA Executable with Profiler

## Two Step Method:

```
dpcpp -fintel fpga <source_file>.cpp -c -o <file_name>.o  
dpcpp -fintel fpga <file_name>.o -Xshardware -Xsprofile
```

## One Step Method:

```
dpcpp -fintel fpga <source_file>.cpp -Xshardware -Xsprofile
```

The profiler will be instrumented within the image and you will be able to run the executable to return information to import into Intel® Vtune Amplifier.

To compile to FPGA executable without profiler, leave off `-Xsprofile`.

# Compiling FPGA Device Separately and Linking

- In the default case, the DPC++ Compiler handles generating the host executable, device image, and final executable
- It is sometimes desirable to compile the host and device separately so changes in the host code do not trigger a long compile

Partition code

has\_kernel.cpp

host\_only.cpp

Then run this command to compile the FPGA image:

```
dpcpp -fintel-fpga has_kernel.cpp -fsycl-link=image -o has_kernel.o -Xhardware
```

This command to produce an object file out of the host only code:


```
dpcpp -fintel-fpga host_only.cpp -c -o host_only.o
```

This command to put the object files together into an executable:

```
dpcpp -fintel-fpga has_kernel.o host_only.o -o a.out -Xhardware
```

This is the long compile

# Lab: Practice the FPGA Development Flow



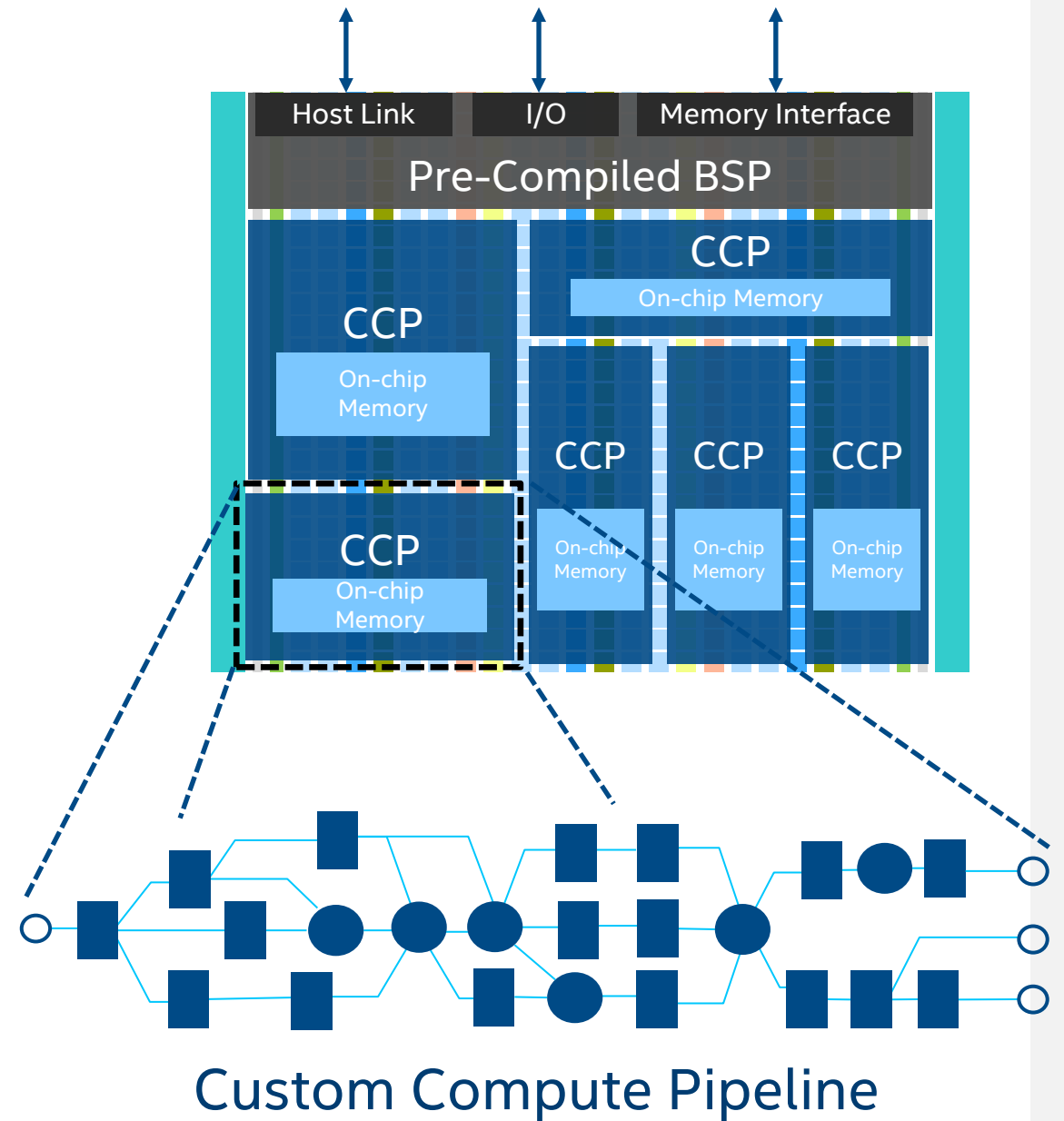
# Section: Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

## Sub-Topics:

- Code to Hardware: An Introduction
- Loop Optimization
- Memory Optimization
- Reports
- Other Optimization Techniques

# Intel® FPGAs

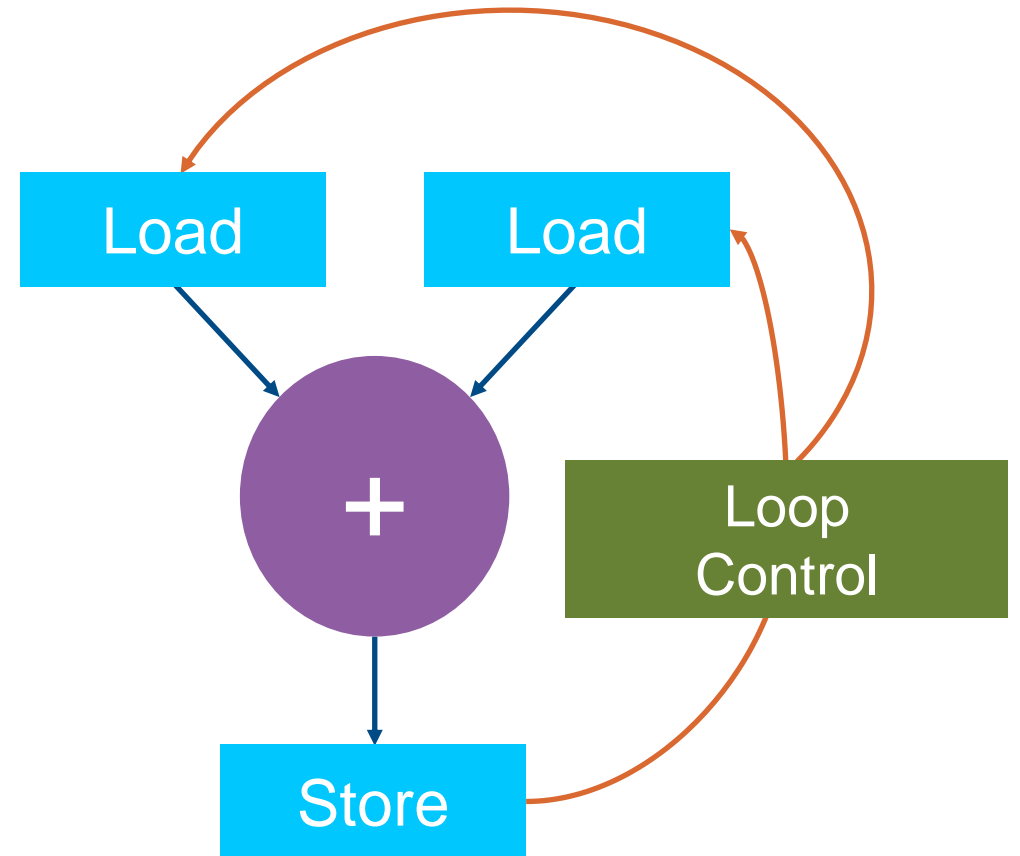
- Implementing Optimized Custom Compute Pipelines (CCPs) synthesized from compiled code



# How Is a Pipeline Built?

- Hardware is added for
  - Computation
  - Memory Loads and Stores
  - Control and scheduling
    - Loops & Conditionals

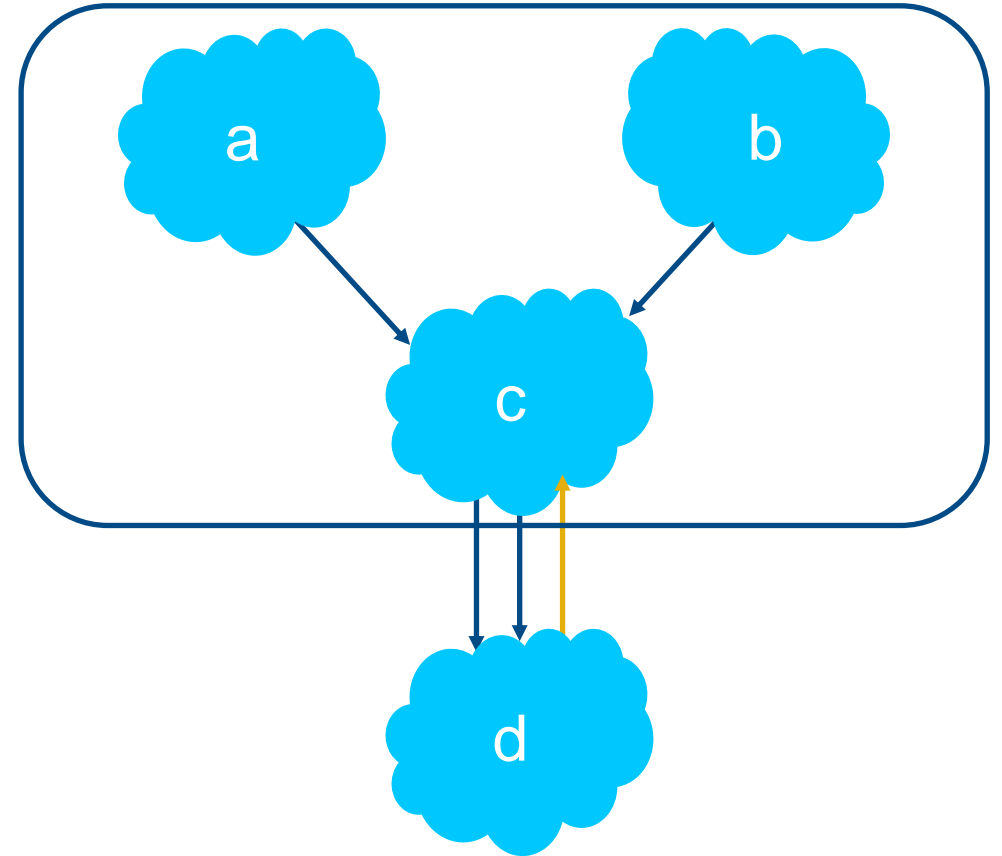
```
for (int i=0; i<LIMIT; i++) {  
    c[i] = a[i] + b[i];  
}
```



— 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

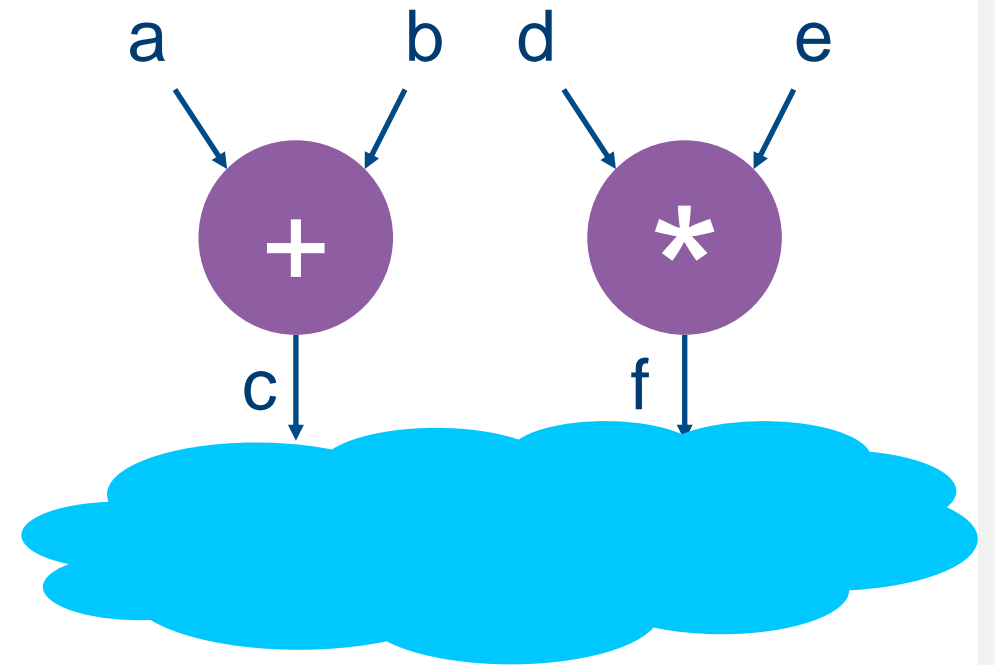




# Simultaneous Independent Operations

- 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

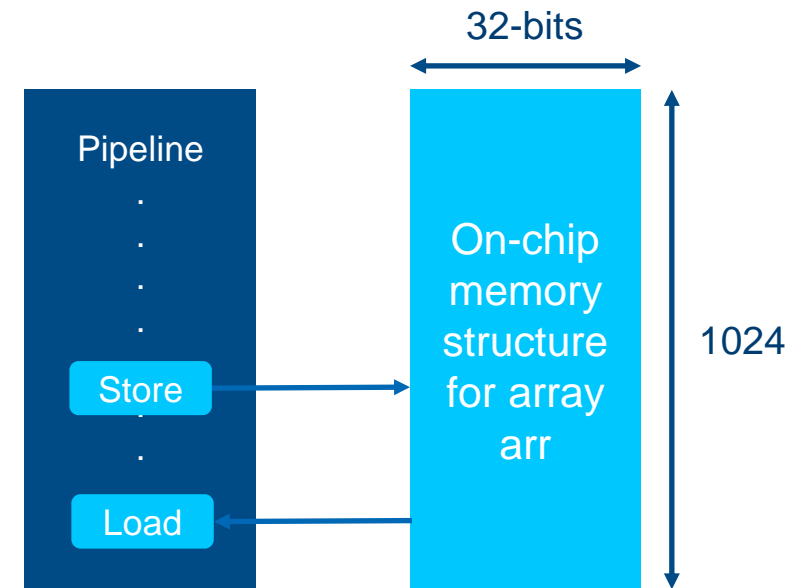
```
c = a + b;  
f = d * e;
```



# On-Chip Memories Built for Kernel 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

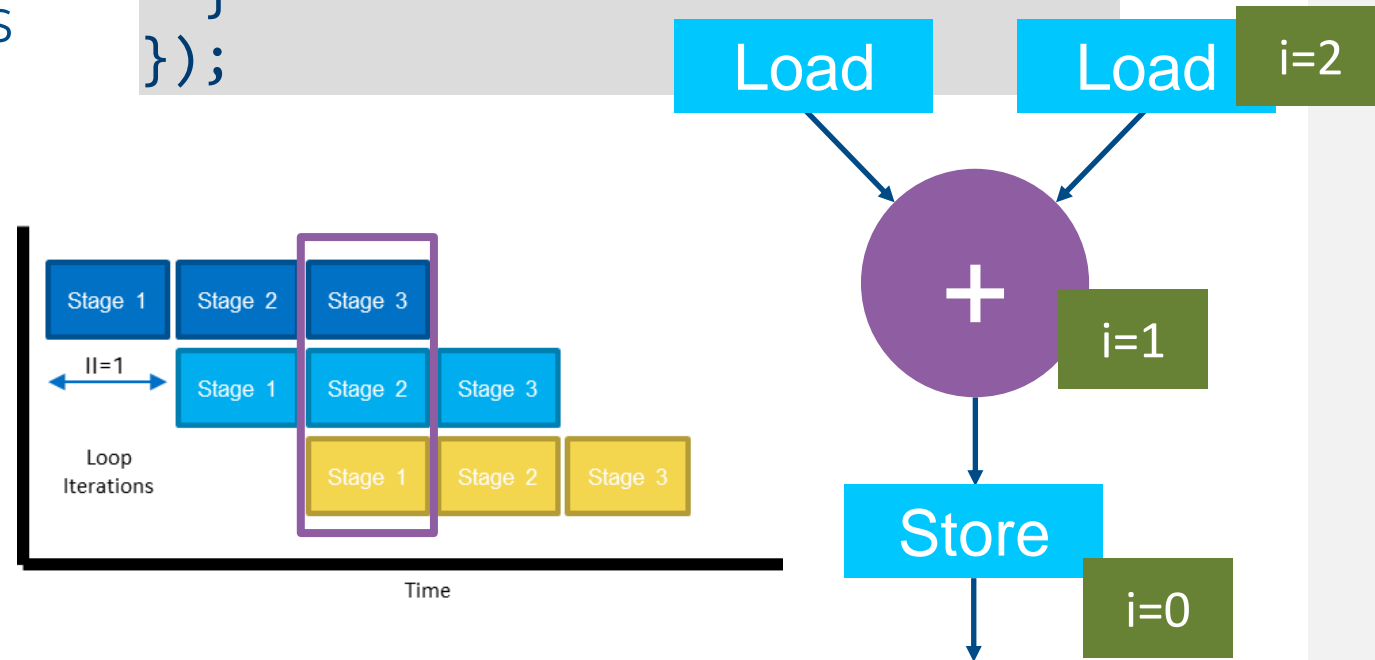
```
//kernel scope
cgh.single_task<>([=]()) {
  int arr[1024];
  ...
  arr[i] = ...; //store to memory
  ...
  ... = arr[j] //load from memory
  ...
} //end kernel scope
```



# 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];  
    }  
});
```

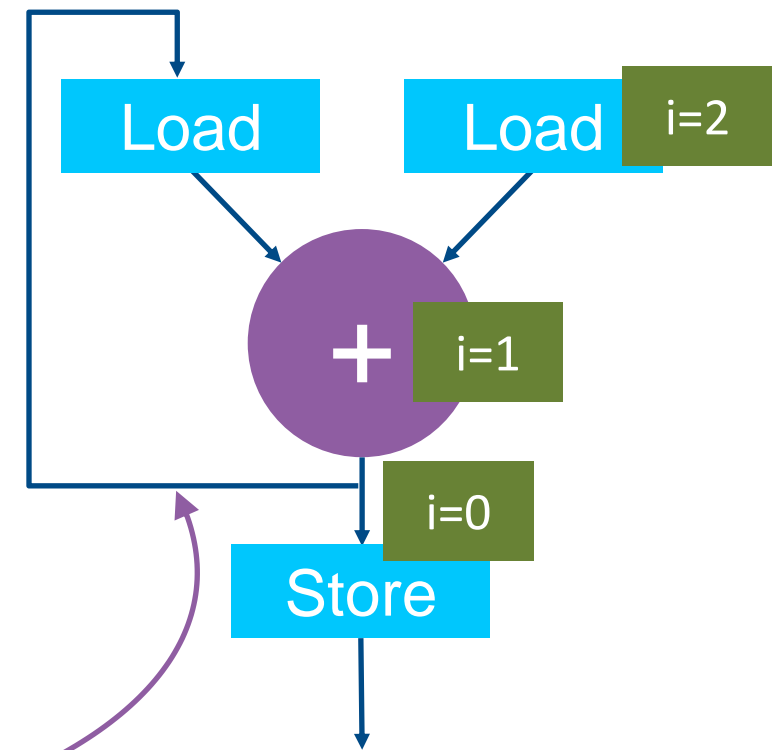


# Dependencies Within the Single Work-Item Kernel

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];  
  }  
});
```

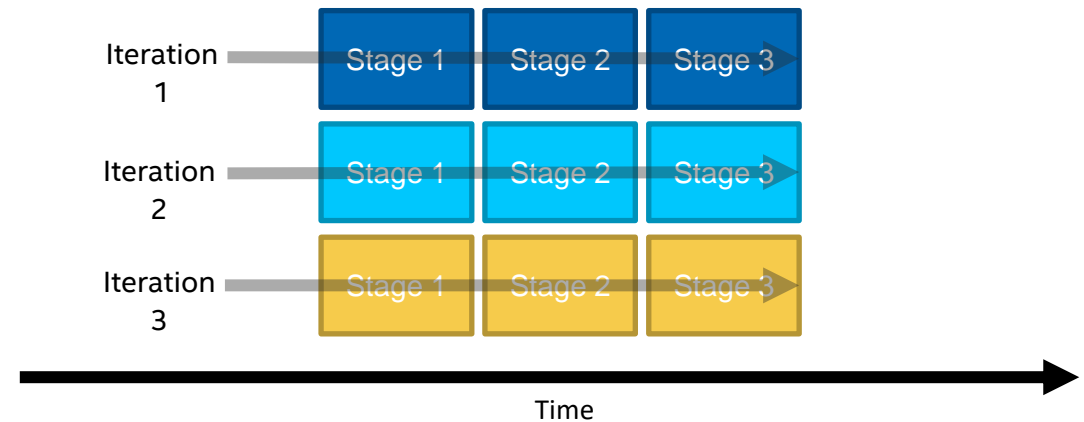
**Key Concept**  
Custom built-in dependencies make FPGAs powerful for many algorithms



# 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];  
    }  
});
```

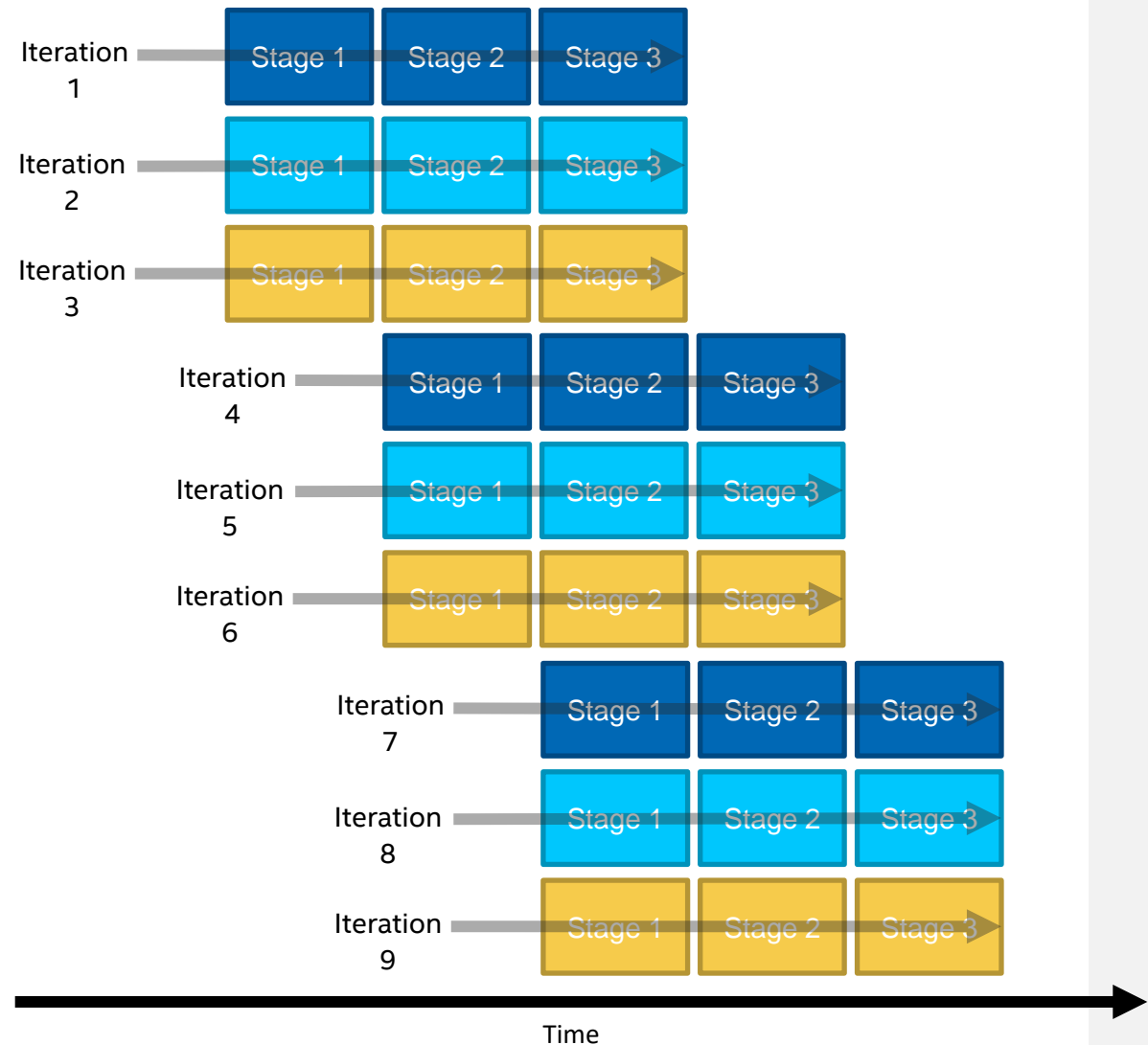


# Unrolled Loops Still Get Pipelined

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

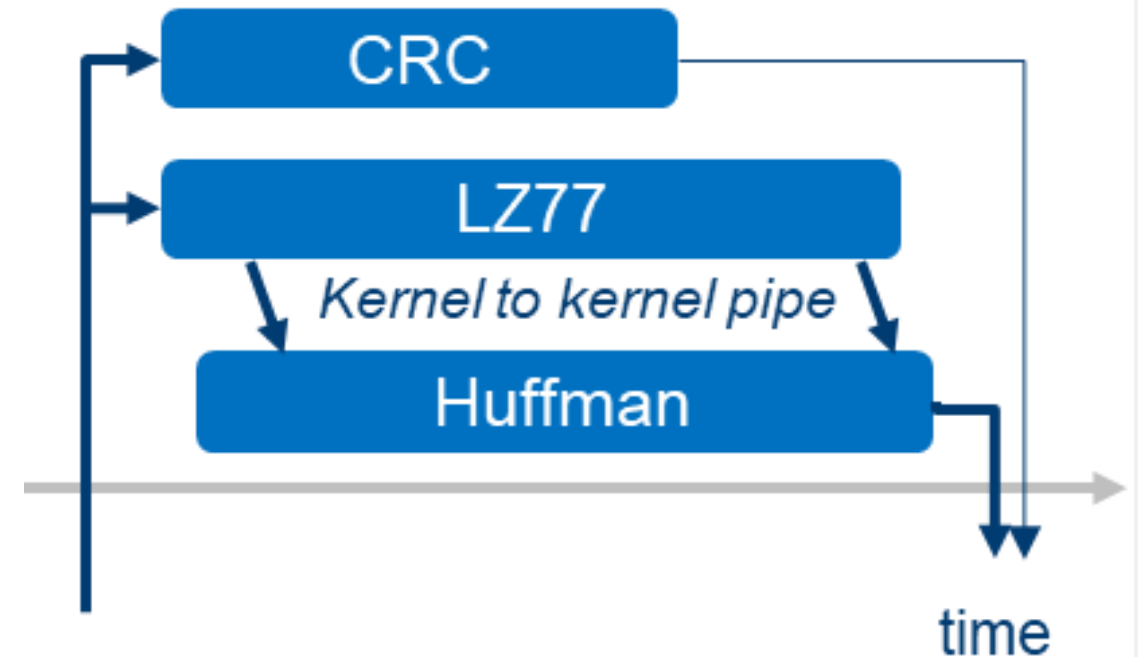
```
handle.single_task<>([=]()) {  
    ... //accessor setup  
    #pragma unroll 3  
    for (int i=1; i<=9; i++) {  
        c[i] += a[i] + b[i];  
    }  
};
```



# 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,  
but, **single\_tasks**  
are recommended  
for FPGAs.

Also, warning: the  
loop optimizations  
we talk about do  
not all apply for  
parallel kernels





# Section: Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

## Sub-Topics:

- Code to Hardware: An Introduction
- **Loop Optimization**
- Memory Optimization
- Reports
- Other Optimization Techniques

# Single Work-Item Kernels

- Single work items kernels are kernels that contain no reference to the work item ID
- Usually launched with the group handler member function `single_task()`
  - Or, launched with other functions without a reference to the work item ID (implying a work group size of 1)
- Contain an outer loop

```
...//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.single_task<class swi_add>([=]() {
        for (unsigned i = 0; i < 128; i++) {
            c[i] = a[i] + b[i];
        }
    });

});

...//application scope
```

# As a Reminder – Parallel Kernels

- Kernels launched with the command group handler member function `parallel_for()` or `parallel_for_work_group()`
- We can build these functionally, but not recommended for best performance
- Much of this section will not apply to parallel kernels

```
...//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);

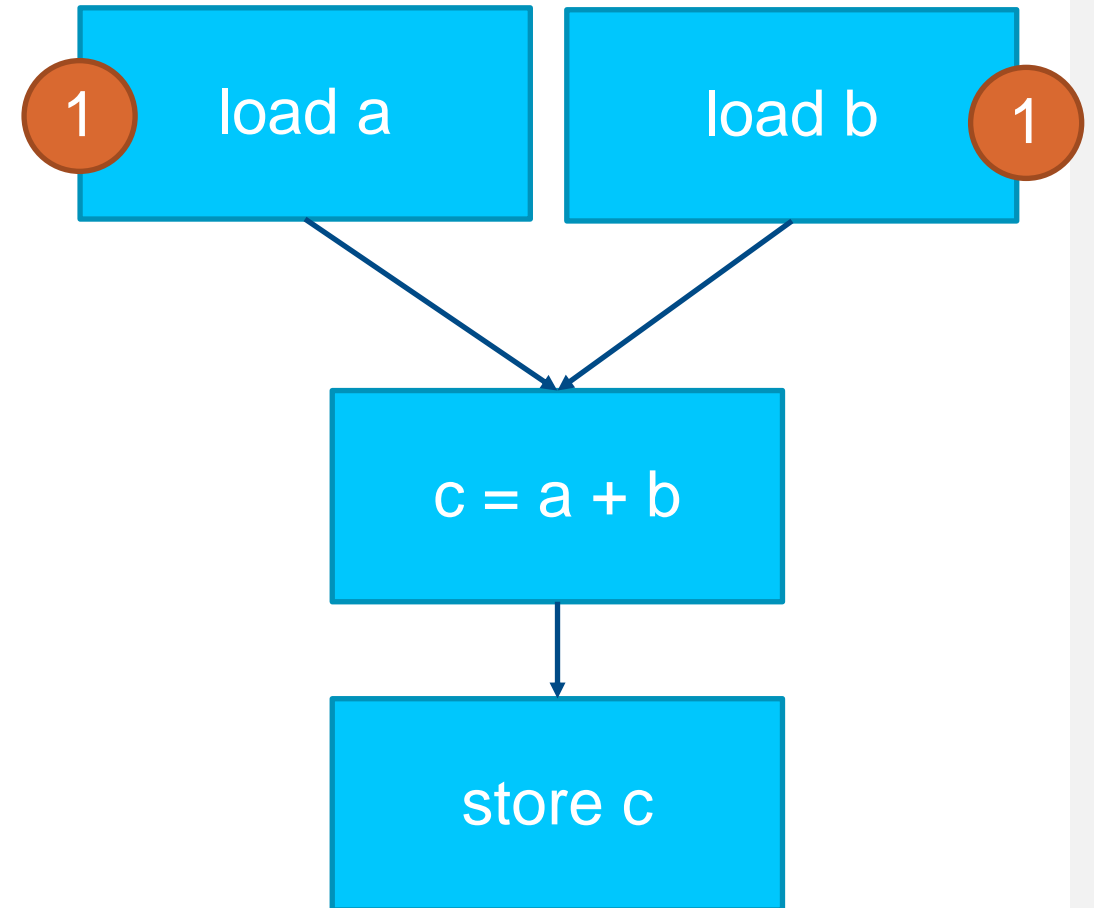
    cgh.parallel_for<class VectorAdd>(num_items, [=](id<1> wiID) {
        c[wiID] = a[wiID] + b[wiID];
    });
});

...//application scope
```

# Understanding Initiation Interval

- dpcpp will infer **pipelined parallel** execution across loop iterations
  - Different stages of pipeline will ideally contain different loop iterations
- Best case is that a new piece of data enters the pipeline each clock cycle

```
...  
cgh.single_task<class swi_add>([=]()) {  
    for (unsigned i = 0; i < 128; i++) {  
        c[i] = a[i] + b[i];  
    }  
};  
...
```

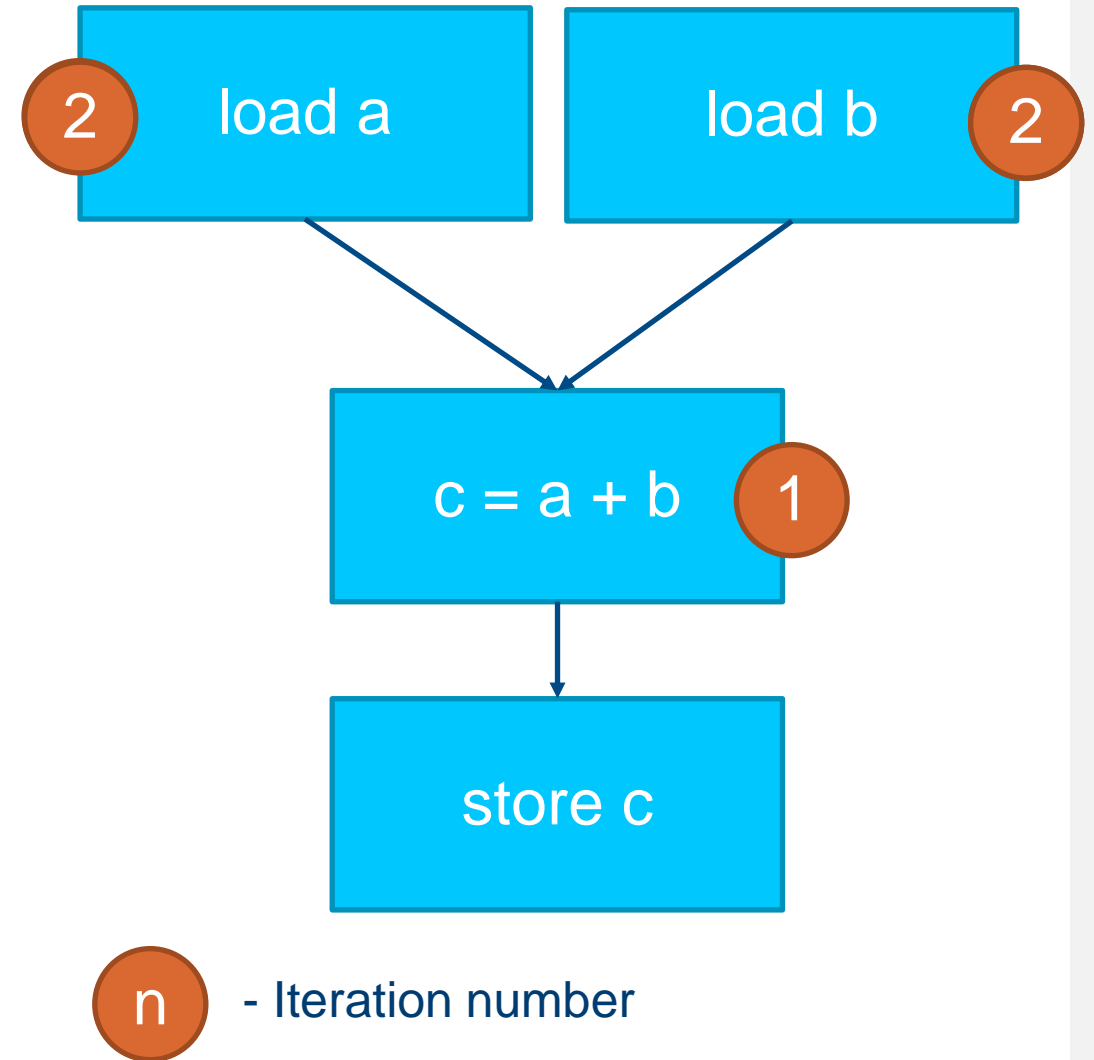


**n** - Iteration number

# Understanding Initiation Interval

- dpcpp will infer **pipelined parallel** execution across loop iterations
  - Different stages of pipeline will ideally contain different loop iterations
- Best case is that a new piece of data enters the pipeline each clock cycle

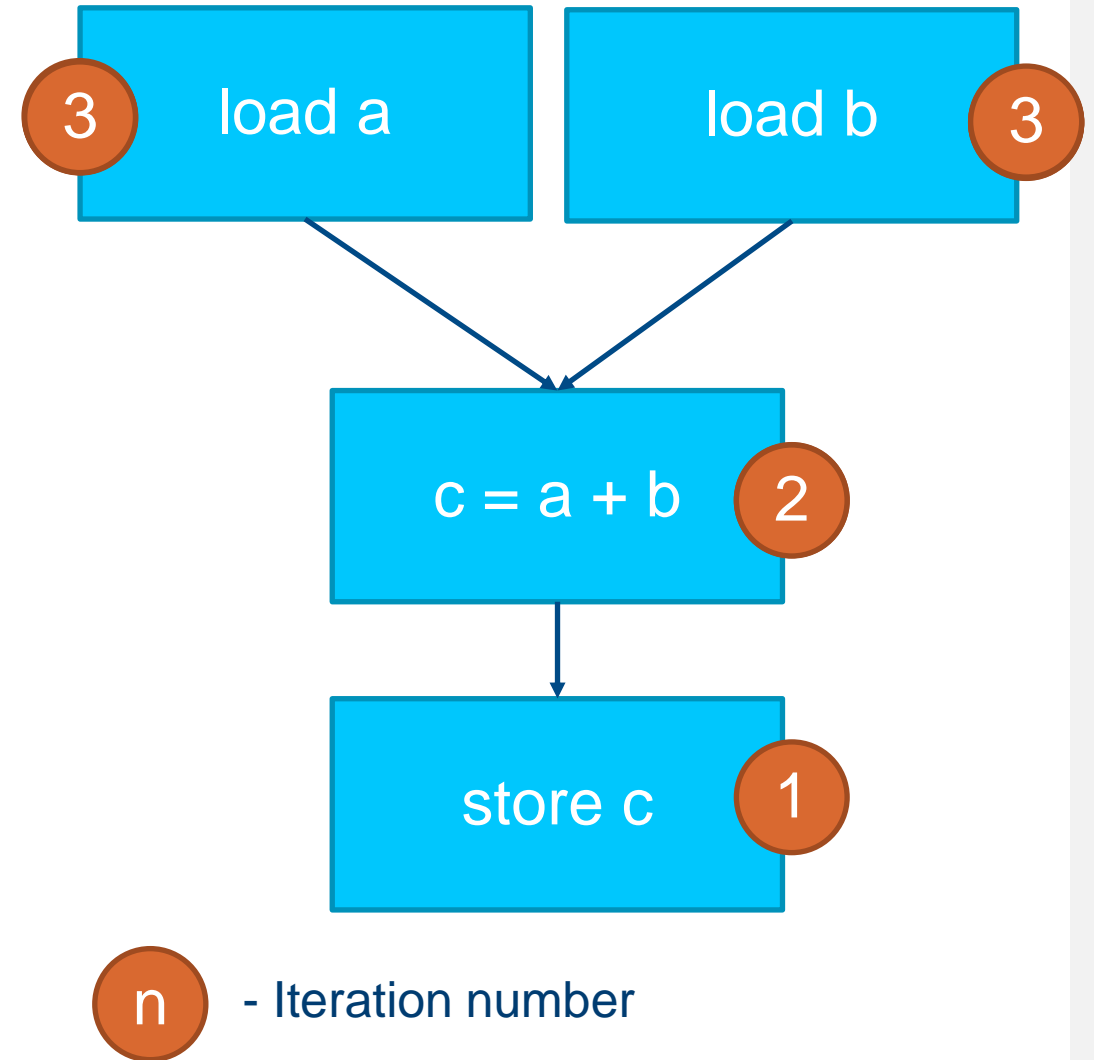
```
...  
cgh.single_task<class swi_add>([=]()) {  
    for (unsigned i = 0; i < 128; i++) {  
        c[i] = a[i] + b[i];  
    }  
};  
...
```



# Understanding Initiation Interval

- dpcpp will infer **pipelined parallel** execution across loop iterations
  - Different stages of pipeline will ideally contain different loop iterations
- Best case is that a new piece of data enters the pipeline each clock cycle

```
...
cgh.single_task<class swi_add>([=]()) {
    for (unsigned i = 0; i < 128; i++) {
        c[i] = a[i] + b[i];
    }
});
...
```



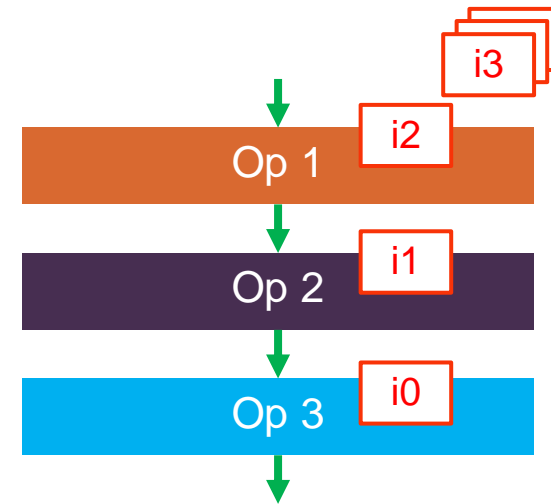
# Loop Pipelining vs Serial Execution

Serial execution is the worst case. One iteration needs to complete fully before a new piece of data enters the pipeline.

Worst Case



Best Case

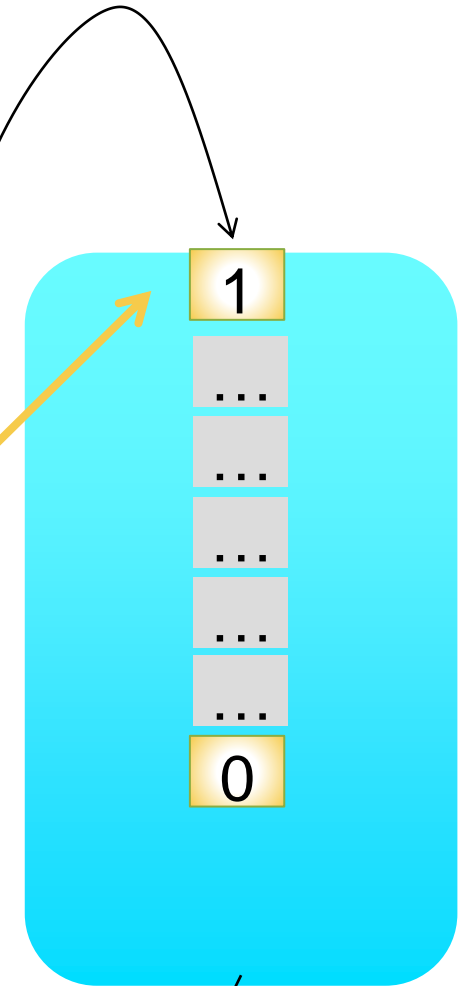


# In-Between Scenario

- Sometimes you must wait more than one clock cycle to input more data
- Because dependencies can't resolve fast enough
- How long you have to wait is called **Initiation Interval** or **II**
- Total number of cycles to run kernel is about (loop iterations)\*II
  - (neglects initial latency)
- Minimizing II is **key** to performance

$$II = 6$$

6 cycles later,  
next iteration  
enter the loop  
body





# Why Could This Happen?

- Memory Dependency

- Kernel cannot retrieve data enough from memory

Report: fpga\_970fa3 - Mozilla Firefox

file:///home/student/DevConFPGALab/original/fpga.prj/reports/report.html#view2?

Reports Summary Throughput Analysis Area Analysis System Viewers

Loops Analysis  Show fully unrolled loops

	Pipelined	II	Speculated iterations	Details
Kernel: const:Hough_transform_kernel (hough_transfo...				Single work-it...
const:Hough_transform_kernelB1 (hough_transfor...	Yes	>=1	0	Serial exe: Me...
const:Hough_transform_kernelB3 (hough_tran...	Yes	>=1	0	Serial exe: Me...
const:Hough_transform_kernelB5 (hough_...	Yes	~339	1	Memory dep...

```
91 auto_sin_table = sin_table_buf.get_access<sycl::access::mode
92 auto_cos_table = cos_table_buf.get_access<sycl::access::mode
93 auto_accumulators = accumulators_buf.get_access<sycl::access
94 //Call the kernel
95 cgh.single_task<class Hough_transform_kernel>{[=]() {
96     for (uint y=0; y<HEIGHT; y++){
97         for (uint x=0; x<WIDTH; x++){
98             unsigned short int increment = 0;
99             if (_pixels[(WIDTH*y)+x] != 0) {
100                 increment = 1;
101             } else {
102                 increment = 0;
103             }
104             for (int theta=0; theta<THETAS; theta++){
105                 int rho = x*cos_table[theta] + y*sin_table[theta];
106                 _accumulators[(THETAS*(rho+RHOS))+theta] += increment
107             }
108         }
109     }
110 }
111 }
112 );
```

\_accumulators[(THETAS\*(rho+RHOS))+theta] += increment;

Value must be retrieved from global memory and incremented

Details

**const::Hough\_transform\_kernel.B5:**

- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Load Operation ([hough\\_transform.cpp: 107](#))
  - To: Store Operation ([hough\\_transform.cpp: 107](#))
- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Load Operation ([hough\\_transform.cpp: 106](#) > [accessor.hpp: 928](#))

# What Can You Do? Use Local Memory

- Transfer global memory contents to local memory before operating on the data

```
...
constexpr int N = 128;
queue.submit([&](handler &cgh) {
    auto A =
        A_buf.get_access<access::mode::read_write>(cgh);

    cgh.single_task<class unoptimized>([=]() {
        for (unsigned i = 0; i < N; i++)
            A[N-i] = A[i];
    });
});
...
```

Non-optimized

```
...
constexpr int N = 128;
queue.submit([&](handler &cgh) {
    auto A =
        A_buf.get_access<access::mode::read_write>(cgh);

    cgh.single_task<class optimized>([=]() {
        int B[N];

        for (unsigned i = 0; i < N; i++)
            B[i] = A[i];

        for (unsigned i = 0; i < N; i++)
            B[N-i] = B[i];

        for (unsigned i = 0; i < N; i++)
            A[i] = B[i];
    });
});
...
```

Optimized

# What Can You Do? Tell the Compiler About Independence

- `[[intel FPGA::ivdep]]`

- Dependencies ignored for all accesses to memory arrays

```
[[intel FPGA::ivdep]]  
for (unsigned i = 1; i < N; i++) {  
    A[i] = A[i - X[i]];  
    B[i] = B[i - Y[i]];  
}
```

Dependency ignored for A and B array

- `[[intel FPGA::ivdep(array_name)]]`

- Dependency ignored for only `array_name` accesses

```
[[intel FPGA::ivdep(A)]]  
for (unsigned i = 1; i < N; i++) {  
    A[i] = A[i - X[i]];  
    B[i] = B[i - Y[i]];  
}
```

Dependency ignored for A array

Dependency for B still enforced

# Why Else Could This Happen?

- Data Dependency
  - Kernel cannot complete a calculation fast enough

```
r_int[k] = ((a_int[k] / b_int[k]) / a_int[1]) / r_int[k-1];
```

Difficult double precision floating point operation must be completed

Report: fpga\_0cbd30 - Mozilla Firefox

file:///home/student/sandbox\_oneAPI/fpga\_compile/bad\_multiply/fpga.prj/reports/rep

Reports Summary Throughput Analysis Area Analysis System Viewers

Loops Analysis  Show fully unrolled loops

	Pipelined	II	Speculated iterations	Details
Kernel: SimpleAdd (memory_dep.cpp:66)				Single work-item...
SimpleAdd.B2 (memory_dep.cpp:71)	Yes	~1	3	
SimpleAdd.B3 (memory_dep.cpp:76)	Yes	38	3	Data dependency
SimpleAdd.B4 (memory_dep.cpp:80)	Yes	~1	3	

memory\_dep.cpp

```
63
64
65 // Kernel
66 cgh.single_task<class SimpleAdd>([=]() {
67     double a_int[ARRAY_SIZE];
68     double b_int[ARRAY_SIZE];
69     double r_int[ARRAY_SIZE];
70
71     for (int i=0; i<ARRAY_SIZE; i++) {
72         a_int[i] = a[i];
73         b_int[i] = b[i];
74     }
75
76     for (int k = 1; k < ARRAY_SIZE; ++k) {
77         r_int[k] = ((a_int[k] / b_int[k]) / a_int[1]) / r_int[k-1];
78     }
79
80     for (int i=0; i<ARRAY_SIZE; i++) {
81         r[i] = r_int[i];
82     }
83 });
84
85 deviceQueue->throw_asynchronous();
86
87
88 } catch (cl::sycl::exception const& e) {
```

Details

**SimpleAdd.B3:**

- Most critical loop feedback path during scheduling:
  - 36.00 clock cycles 64-bit Double-precision Floating-point Divide Operation (memory\_dep.cpp: 77)
- Hyper-Optimized loop structure: n/a
- Stallable instruction: None
- Maximum concurrent iterations: Capacity of loop

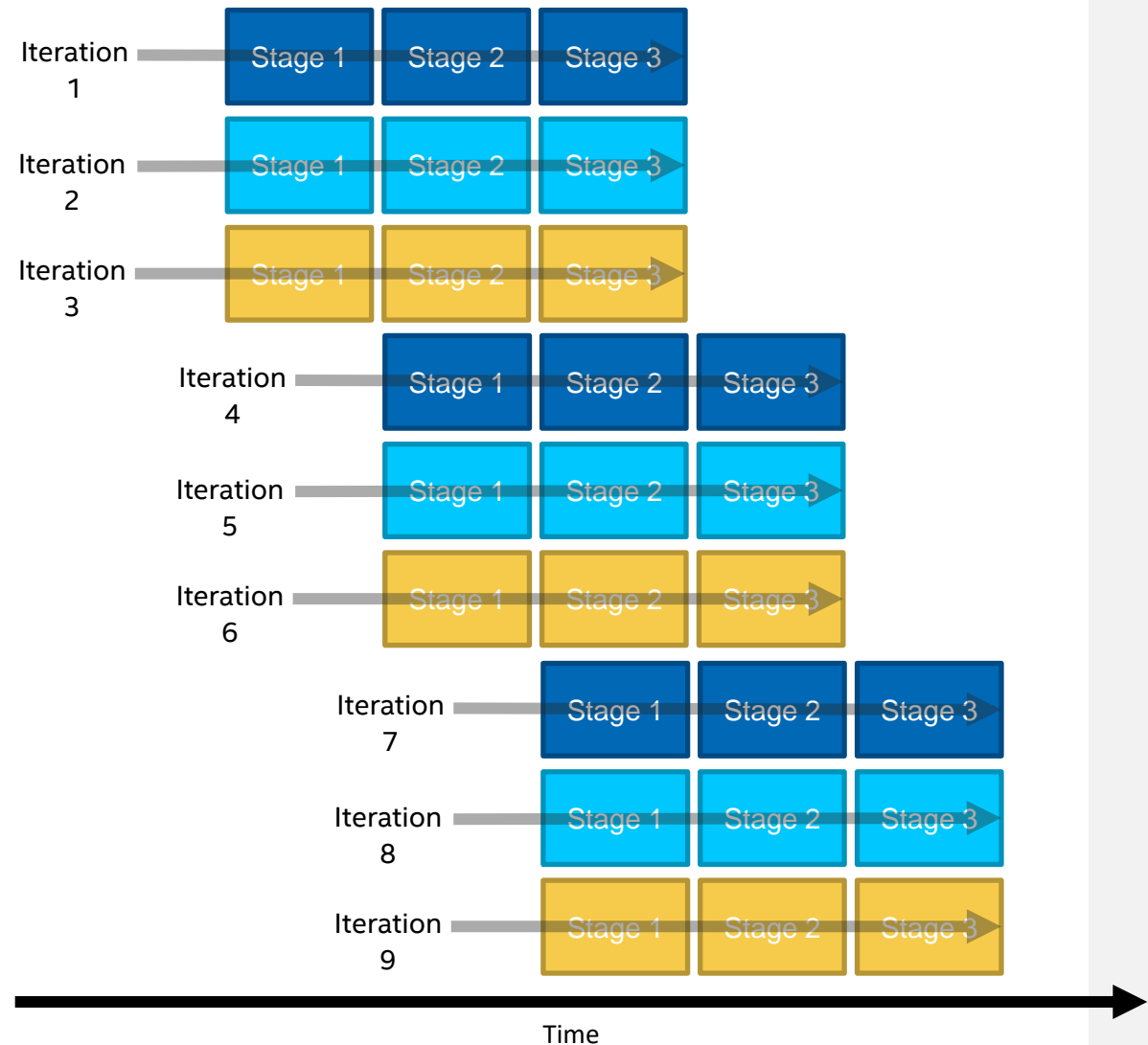
# What Can You Do?

- Do a simpler calculation
- Pre-calculate some of the operations on the host
- Use a simpler type
- Use floating point optimizations (discussed later)
- Advanced technique: Increase time (pipeline stages) between start of calculation and when you use answer
  - See the “Relax Loop-Carried Dependency” in the Optimization Guide for more information

# How Else to Optimize a Loop? Loop Unrolling

- 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];  
    }  
});
```



# Maximum Clock Frequency (Fmax)

- The clock frequency the FPGA will be clocked at depends on what hardware your kernel compiles into
- More complicated hardware cannot run as fast
- The whole kernel will have one clock
- The compiler's heuristic is to get a lower II, sacrificing a higher Fmax

A slow operation can slow down your entire kernel by lowering the clock frequency

# How Can You Tell This Is a Problem?

- Optimization report tells you the target frequency for each loop in your code

```
cgh.single_task<example>([=]() {  
    int res = N;  
    #pragma unroll 8  
    for (int i = 0; i < N; i++) {  
        res += 1;  
        res ^= i;  
    }  
    acc_data[0] = res;  
});
```

	Target II	Scheduled fMAX	Block II	Latency	Max Interleaving Iterations
<b>Kernel: example ( Target Fmax : Not specified MHz ) ( fmaxii.cpp:23 )</b>					
Block: example.B0	Not specified	240.0	1	2	1
Block: example.B2	Not specified	240.0	1	6	1
<b>Loop: example.B1 (fmaxii.cpp:26)</b>					
Block: example.B1	Not specified	106.5	2	7	1



# What Can You Do?

- Make the calculation simpler
- Tell the compiler you'd like to change the trade off between II and Fmax
  - Attribute placed on the line before the loop
  - Set to a higher II than what the loop currently has

```
[[intel fpga::ii(n)]]
```

# Area


- The compiler sacrifices area in order to improve loop performance. What if you would like to save on the area in some parts of your design?

- Give up II for less area
  - Set the II higher than what compiler result is

```
[[intel FPGA::ii(n)]]
```

- Give up loop throughput for area
  - Compiler increases loop concurrency to achieve greater throughput
  - Set the max\_concurrency value lower than what the compiler result is

```
[[intel FPGA::max_concurrency(n)]]
```



# Section: Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

## Sub-Topics:

- Code to Hardware: An Introduction
- Loop Optimization
- **Memory Optimization**
- Reports
- Other Optimization Techniques

# Understanding Board Memory Resources

Memory Type	Physical Implementation	Latency for random access (clock cycles)	Throughput (GB/s)	Capacity (MB)
Global	DDR	240	34.133	8000
Local	On-chip RAM	2	~8000	66
	Registers	2/1	~240	0.2

Key takeaway: many times, the solution for a bottleneck caused by slow memory access will be to use local memory instead of global

# Global Memory Access is Slow – What to Do?

- We've seen this before... This will appear as a memory dependency problem

```
...
constexpr int N = 128;
queue.submit([&](handler &cgh) {
    auto A =
        A_buf.get_access<access::mode::read_write>(cgh);

    cgh.single_task<class unoptimized>([=]() {
        for (unsigned i = 0; i < N; i++)
            A[N-i] = A[i];
    });
});
...
```

**Non-optimized**

```
...
constexpr int N = 128;
queue.submit([&](handler &cgh) {
    auto A =
        A_buf.get_access<access::mode::read_write>(cgh);

    cgh.single_task<class optimized>([=]() {
        int B[N];

        for (unsigned i = 0; i < N; i++)
            B[i] = A[i];

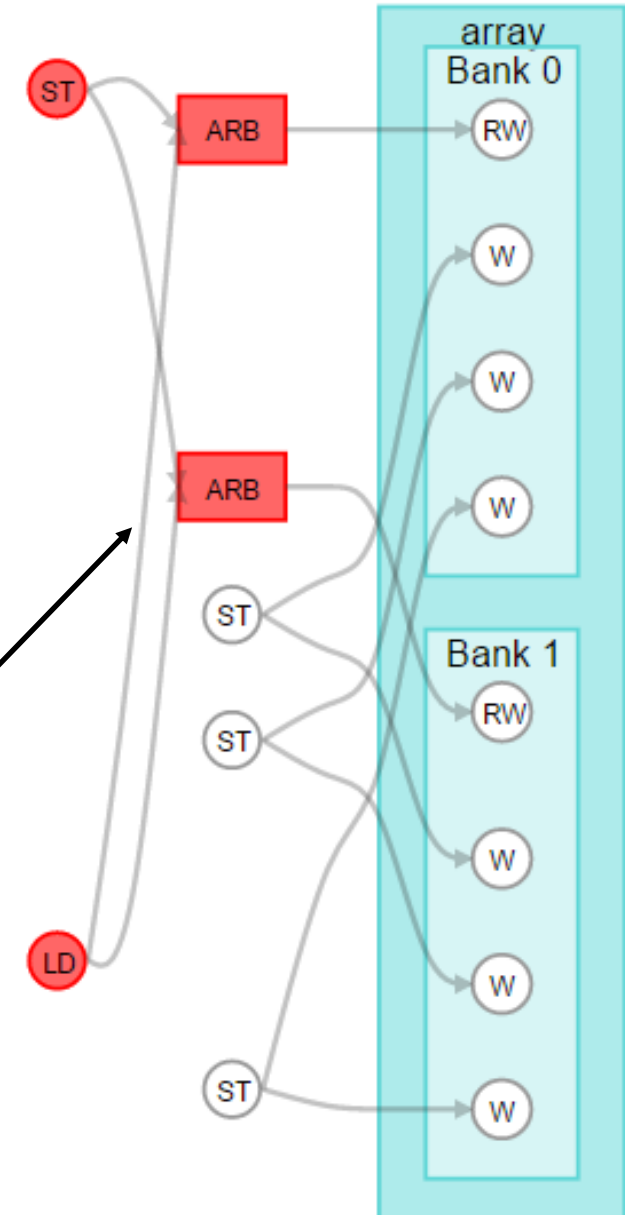
        for (unsigned i = 0; i < N; i++)
            B[N-i] = B[i];

        for (unsigned i = 0; i < N; i++)
            A[i] = B[i];
    });
});
...
```

**Optimized**

# Local Memory Bottlenecks

- If more load and store points want to access the local memory than there are ports available, arbiters will be added
- These can stall, so are a potential bottleneck
- Show up in red in the Memory Viewer section of the optimization report



# Local Memory Bottlenecks



Natively, the memory architecture has 2 ports  
The compiler uses optimizations to minimize arbitration  
Your job is to write code the compiler can optimize

# Double-Pumped Memory Example

- Increase the clock rate to 2x
- Compiler can automatically implement double-pumped memory

```
//kernel scope
```

```
...
```

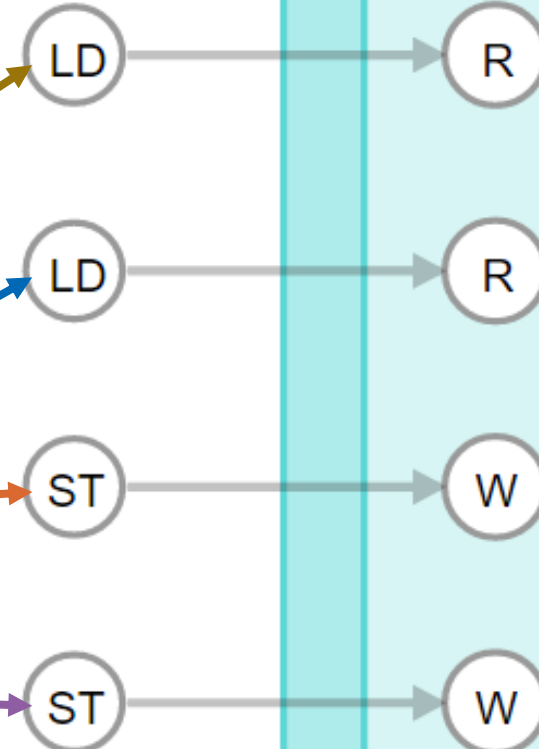
```
int array[1024];
```

```
array[ind1] = val;
```

```
array[ind1+1] = val;
```

```
calc = array[ind2] + array[ind2+1];
```

```
...
```





# Local Memory Replication Example

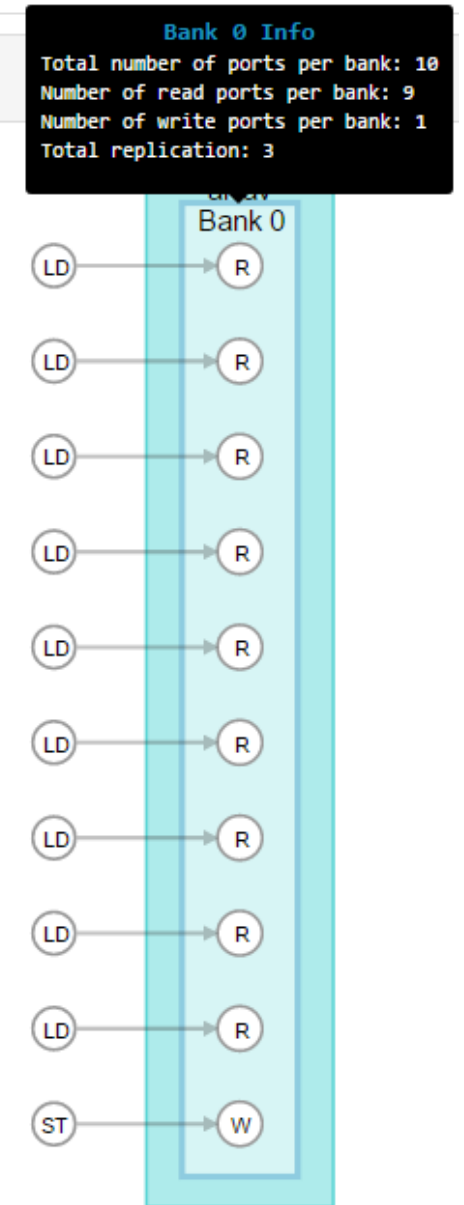
```
//kernel scope
...
  int array[1024];
  int res = 0;

  (ST) array[ind1] = val;
  #pragma unroll
  for (int i = 0; i < 9; i++)
  (LD)  res += array[ind2+i];

  calc = res;
...
```

Turn 4 ports of double-pumped memory to unlimited ports

Drawbacks: logic resources, stores must go to each replication



# Coalescing

```
//kernel scope
...
local int array[1024];
int res = 0;

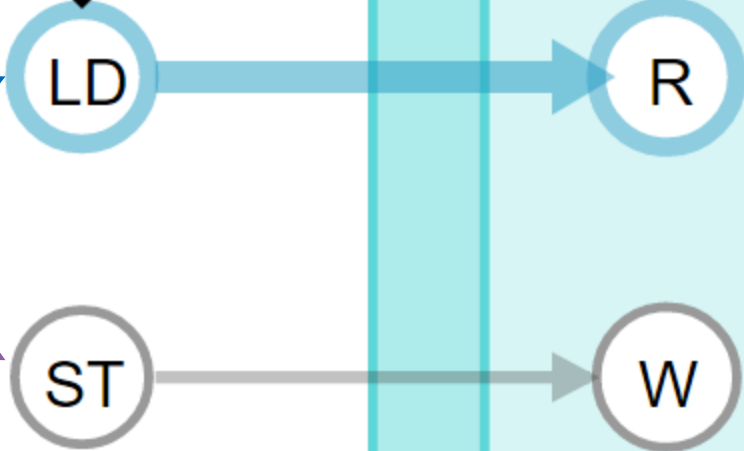
#pragma unroll
for (int i = 0; i < 4; i++)
    array[ind1*4 + i] = val;

#pragma unroll
for (int i = 0; i < 4; i++)
    res += array[ind2*4 + i];

calc = res;
...
```

Load Info  
Width: 128 bits  
Type: Pipelined  
Stall-free: Yes  
Loads from: array  
Start-Cycle: 2  
Latency: 3

Width: 128 bits  
Type: Pipelined  
Stall-free: Yes



Continuous addresses can be coalesced into wider accesses

# Banking

- Divide the memory into independent fractional pieces (banks)

```
//kernel scope
```

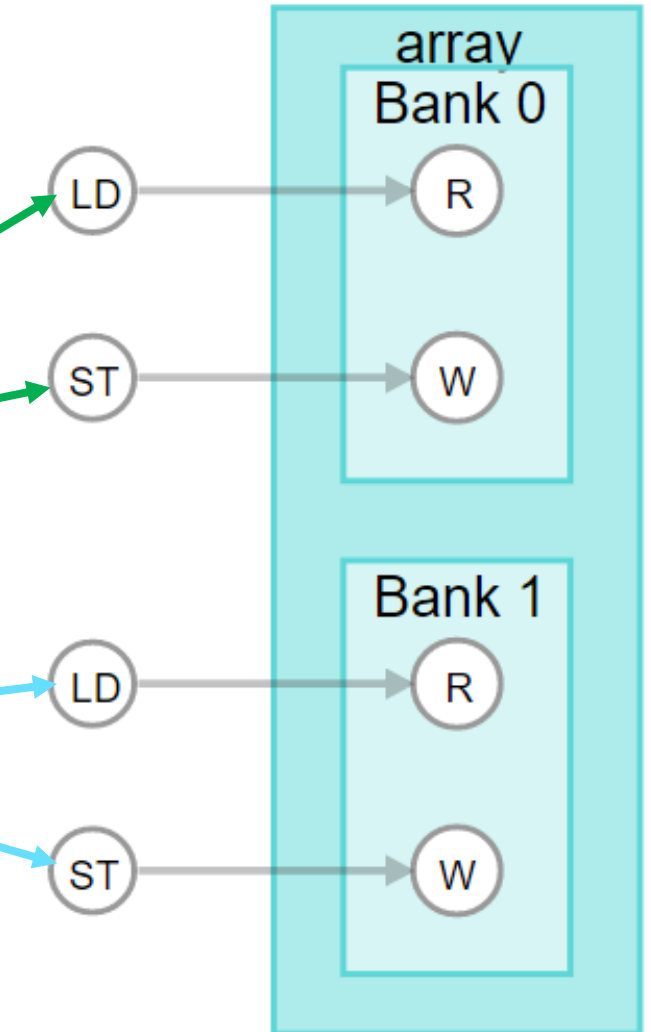
```
...  
int array[1024][2];
```

```
array[ind1][0] = val1;
```

```
array[ind2][1] = val2;
```

```
calc = (array[ind2][0] +  
        array[ind1][1]);
```

```
...
```



# Attributes for Local Memory Optimization

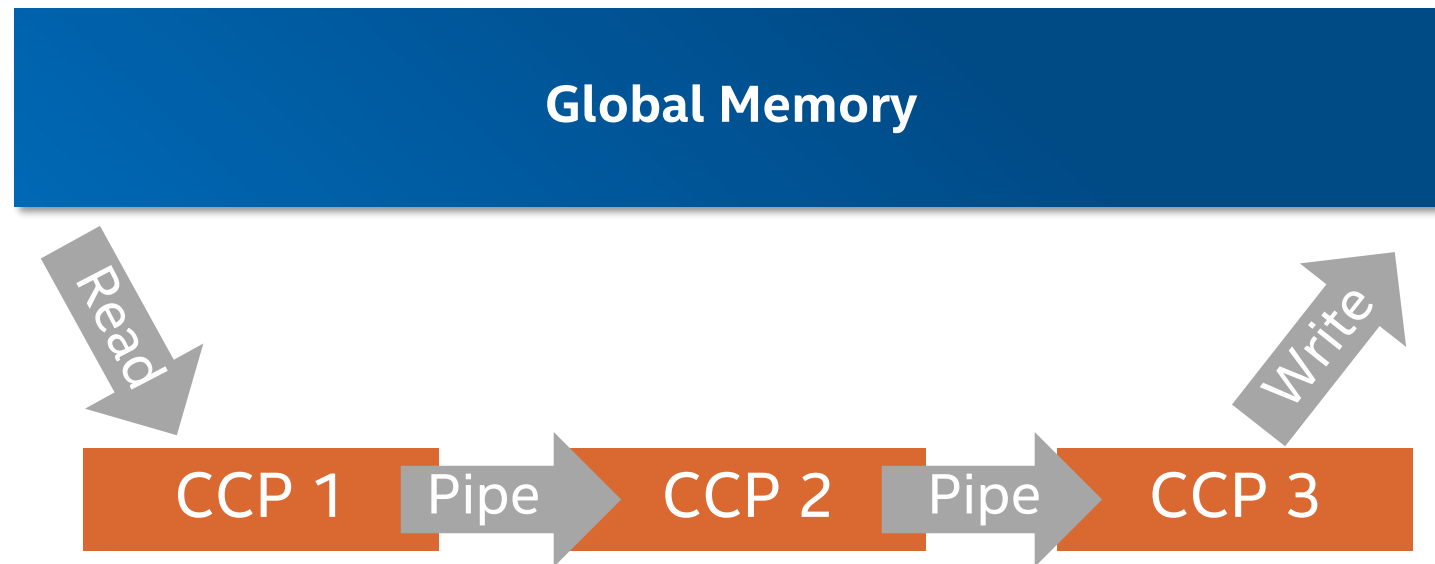
Note: Let the compiler try on it's own first.  
It's very good at inferring an optimal structure!

Attribute	Usage
numbanks	[[intelfpga::numbanks(N)]]
bankwidth	[[intelfpga::bankwidth(N)]]
singlepump	[[intelfpga::singlepump]]
doublepump	[[intelfpga::doublepump]]
max_replicates	[[intelfpga::max_replicates(N)]]
simple_dual_port	[[intelfpga::simple_dual_port]]

Note: This is not a comprehensive list. Consult the Optimization Guide for more.

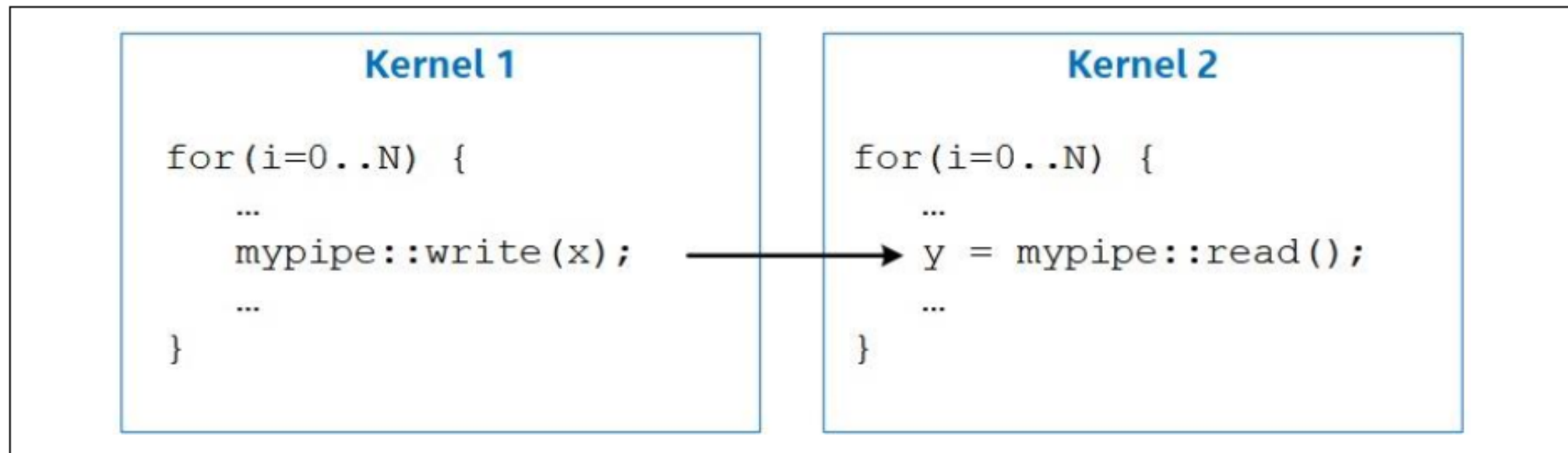
# Pipes – Element the Need for Some Memory


Create custom direct point-to-point communication between CCPs with Pipes



# Task Parallelism By Using Pipes

- Launch separate kernels simultaneously
- Achieve synchronization and data sharing using pipes
  - Make better use of your hardware





# Section: Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

## Sub-Topics:

- Code to Hardware: An Introduction
- Loop Optimization
- Memory Optimization
- **Reports**
- Other Optimization Techniques

# HTML Optimization Report

- Static report showing optimization, area, and architectural information
  - Automatically generated with the object file
    - Located in **<file\_name>.prj\reports\report.html**
  - Dynamic reference information to original source code



# Optimization Report – Throughput Analysis

- Loops Analysis and Fmax II sections
- Actionable feedback on pipeline status of loops
- Show estimated Fmax of each loop

The screenshot shows a web-based optimization report for a file named `hough_transform.cpp`. The browser address bar shows the file path: `file:///home/student/DevConFPGALab/original/fps`. The report is titled "Reports" and includes a "Loops Analysis" section with a checkbox for "Show fully unrolled loops" which is checked.

	Pipelined	II	Specu
Kernel: const:Hough_transform_kernel (hough_transf...			
const:Hough_transform_kernel.B1 (hough_transfor...	Yes	>=1	0
const:Hough_transform_kernel.B3 (hough_tran...	Yes	>=1	0
const:Hough_transform_kernel.B5 (hough...	Yes	~339	1

Below the table is a "Details" section for `const::Hough_transform_kernel.B3:`

- Iteration executed serially across `const::Hough_transform_kernel.B5`. Only a single loop iteration will execute inside this region due to memory dependency:
  - From: Load Operation ([hough\\_transform.cpp: 107](#))
  - To: Store Operation ([hough\\_transform.cpp: 107](#))
- Iteration executed serially across `const::Hough_transform_kernel.B5`. Only a single loop iteration will execute

The code snippet on the right shows the implementation of the Hough transform kernel, with lines 97-110 visible. Line 107 is highlighted, corresponding to the memory dependency mentioned in the details.

# Optimization Report – Area Analysis

- Generate detailed estimated area utilization report of kernel scope code
  - Detailed breakdown of resources by system blocks
  - Provides architectural details of HW
    - Suggestions to resolve inefficiencies

Report: fpga\_970fa3 - Mozilla Firefox

Report: fpga\_970fa3

file:///home/student/DevConFPGALab/original/fpga.f

### Reports

#### Area Analysis of System

(area utilization values are estimated)  
Notation file:X > file:Y indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs
Function overhead	1338	2411
Private Variable: - 'theta' (hough_transform.cpp:105)	27	43
Private Variable:	..	..

#### hough\_transform.cpp

```
98 for (uint x=0; x<WIDTH; x++){
99     unsigned short int increment = 0;
100     if (_pixels[(WIDTH*y)+x] != 0) {
101         increment = 1;
102     } else {
103         increment = 0;
104     }
105     for (int theta=0; theta<THETAS;
106         theta++){
107         int rho = x*_cos_table[theta] + y
108             *_sin_table[theta];
109         _accumulators[(THETAS*(rho+RHOS
110             ))+theta] += increment;
111     }
112 }
```

#### Details

**Private Variable: - 'theta' (hough\_transform.cpp:105):**

- Type: Register
- 1 register of width 9 and depth 342 (depth was increased by a factor of 339 due to a loop initiation interval of 339.)
- 1 register of width 32 and depth 342 (depth was increased by a factor of 339 due to a loop initiation interval of 339.)

# Optimization Report – Graph Viewer

- The system view of the Graph Viewer shows following types of connections
  - Control
  - Memory, if your design has global or local memory
  - Pipes, if your design uses pipes

The screenshot displays the Intel System Viewer Graph Viewer interface. The browser window shows the report URL: `file:///home/student/fpga_trn/OCL_19_1/SimpleKernel/reports/report.html#view4`. The navigation tabs include Reports, Summary, Throughput Analysis, Area Analysis, and System Viewers. The System Viewer section is active, showing a graph with a tooltip for a 'Store' component. The tooltip details are:

- Width: 32 bits
- Type: Burst-coalesced
- Stall-free: No
- Start Cycle: 14
- Latency: 2

The code editor on the right shows the source code for `SimpleKernel.cl`:

```
1 //ACL Kernel
2 _kernel
3 void SimpleKernel(__global const float * restrict in,
4                  __global const float * restrict in2, __global
5                  float * restrict out, uint N)
6 {
7     //Perform the Math Operation
8     for (uint index = 0; index < N; index++)
9         out[index] = in[index] * in2[index];
10 }
```

The details table at the bottom provides a structured view of the store's properties:

Store:	
Width	32 bits
Type	Burst-coalesced
Stall-free	No
Start Cycle	14
Latency	2

# Optimization Report – Schedule Viewer

Schedule in clock cycles for different blocks in your code

The screenshot displays the Intel Schedule Viewer interface. At the top, there are tabs for 'Reports', 'Summary', 'Throughput Analysis', 'Area Analysis', and 'System Viewers'. The 'Schedule List (alpha)' on the left shows a tree view of the code blocks, including 'System', '\_ZTSZZ4mai', and several 'const::Hot' blocks. The main 'Schedule Viewer (alpha)' window shows a Gantt chart titled 'Cluster instruction schedule cycle'. The x-axis represents 'Absolute clock cycle' from 0 to 8. The y-axis lists instruction blocks: '...ter 0', 'i', '+', '...Comp.', and 'Xor'. The chart shows that '...ter 0' starts at cycle 0 and runs for 6 cycles. 'i' starts at cycle 3 and runs for 1 cycle. '+' starts at cycle 4 and runs for 1 cycle. '...Comp.' starts at cycle 4 and runs for 1 cycle. 'Xor' starts at cycle 4 and runs for 1 cycle. A 'Details' window at the bottom shows the source code for 'hough\_transform.cpp'.

```
1 #include <vector>
2 #include <CL/sycl.hpp>
3 #include <CL/sycl/intel/fpga_extensions.hpp>
4 #include <chrono>
5
6 // This file defines the sin and cos values for each degree up to 180
7 #include "sin_cos_values.h"
8
9 #define WIDTH 180
10 #define HEIGHT 120
11 #define IMAGE_SIZE WIDTH*HEIGHT
12 #define THETAS 180
13 #define RHOS 217 //Size of the image diagonally: (sqrt(180^2+120^2))
14 #define NS (1000000000.0) // number of nanoseconds in a second
15
16 using namespace std;
17 using namespace cl;
18
19 // This function reads in a bitmap and outputs an array of pixels
20 void read_image(char *image_array);
21
22 class Hough_Transform_kernel;
23
24 int main() {
25     //Declare arrays
26     ...
```

# HTML Kernel Memory Viewer


- Helps you identify data movement bottlenecks in your kernel design. Illustrates:
  - Memory replication
  - Banking
  - Implemented arbitration
  - Read/write capabilities of each memory port

The screenshot displays the HTML Kernel Memory Viewer interface. At the top, there are navigation tabs: Reports, Summary, Throughput Analysis, Area Analysis, and System Viewers. The main area is divided into three sections:

- Memory List:** A tree view showing the system hierarchy. The selected item is 'accum' under the path 'System > \_ZTSZZ4mai > \_arg\_ > AccessRa > AccessRa > accum'.
- Memory Viewer:** A diagram showing the memory architecture. It features two 'LD' (Load) and 'ST' (Store) ports on the left, each connected to a 'SHARE' block. These 'SHARE' blocks are connected to a central 'accum\_local Bank 0' block, which contains 'R' (Read) and 'W' (Write) ports.
- Code Snippet:** A vertical list of assembly instructions on the right, including '#ir', '#de', 'usi', 'voi', 'cli', and 'int'.

Below the Memory Viewer, there is a 'Details' section for the selected 'accum\_local' memory:

accum_local:	
Requested size	156240 bytes
Implemented size	256 kilobytes = $2^{\text{ceil}(\log_2(\text{Req}))}$
Number of banks	1
Bank width (word size)	16 bits
Bank depth	131072 words



# Section: Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

## Sub-Topics:

- Code to Hardware: An Introduction
- Loop Optimization
- Memory Optimization
- Reports
- **Other Optimization Techniques**

# Avoid Expensive Functions

- Expensive functions take a lot of hardware and run slow
- Examples
  - Integer division and modulo (remainder) operators
  - Most floating-point operations except addition, multiplication, absolute, and comparison
  - Atomic functions

# Inexpensive Functions

- Use instead of expensive functions whenever possible
  - Minimal effects on kernel performance
  - Consumes minimal hardware
- Examples
  - Binary logic operations such as AND, NAND, OR, NOR, XOR, and XNOR
  - Logical operations with one constant argument
  - Shift by constant
  - Integer multiplication and division by a constant that is to the power of 2
  - Bit swapping (Endian adjustment)



# Use Least-“Expensive” Data Type

- Understand cost of each data type in latency and logic usage
  - Logic usage may be  $> 4x$  for double vs. float operations
  - Latency may be much larger for float and double operations compared to fixed point types
- Measure or restrict the range and precision (if possible)
  - Be familiar with the width, range and precision of data types
  - Use half or single precision instead of double (default)
  - Use fixed point instead of floating point
  - Don't use float if short is sufficient

# Floating-Point Optimizations

- Applies to `half`, `float` and `double` data types
- Optimizations will cause small differences in floating-point results
  - **Not** IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) compliant
- Floating-point optimizations:
  - Tree Balancing
  - Reducing Rounding Operations

# Tree-Balancing

- Floating-point operations are not associative
  - Rounding after each operation affects the outcome
  - i.e.  $((a+b) + c) \neq (a+(b+c))$
- By default the compiler doesn't reorder floating-point operations
  - May create an imbalance in a pipeline, costs latency and possibly area
- Manually enable compiler to balance operations
  - For example, create a tree of floating-point additions in SGEMM, rather than a chain
  - Use `-Xsfp-relaxed=true` flag when calling `dpcpp`

# Rounding Operations

- For a series of floating-point operations, IEEE 754 require multiple rounding operation
- Rounding can require significant amount of hardware resources
- Fused floating-point operation
  - Perform only one round at the end of the tree of the floating-point operations
  - Other processor architectures support certain fused instructions such as fused multiply and accumulate (FMAC)
  - Any combination of floating-point operators can be fused
- Use dpcpp compiler switch **-Xsfpc**

# References and Resources

# 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>

# Lab: Optimizing the Hough Transform Kernel

# Legal Disclaimers/Acknowledgements

- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.
- Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit [www.intel.com/benchmarks](http://www.intel.com/benchmarks).
- Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.
- Your costs and results may vary.
- Intel technologies may require enabled hardware, software or service activation
- No product or component can be absolutely secure
- Your costs and results may vary
- Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others
- OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos
- \*Other names and brands may be claimed as the property of others



Copyright © 2021 Intel Corporation.

This document is intended for personal use only.

Unauthorized distribution, modification, public performance, public display, or copying of this material via any medium is strictly prohibited.

intel®