

oneAPI SYCL Essentials

Ben Odom

oneAPI's implementation of SYCL



intel[®]

Introduction to oneAPI

- **Agenda**

- Introduction & Overview to oneAPI
- Introduction to the Intel® DevCloud – for Hands On Section
- Introduction to Jupyter notebooks used for training
- Introduction to SYCL
- SYCL Program Structure
- Graphs and Dependences

- **Hands On**

- Introduction to SYCL - Simple
- Complex multiplication
- Vtune & Advisor

Learning Objectives

Explain how oneAPI can solve the **challenges of programming** in a heterogeneous world

Use **oneAPI solutions** to enable your workflows

Experiment with **oneAPI tools and libraries** on the Intel® DevCloud

Understand the SYCL language and programming model

Use **device selection** to **offload kernel workloads**

Build a sample SYCL application through hands-on lab exercises

Cross-Architecture Programming for Accelerated Compute, Freedom of Choice for Hardware

oneAPI: Industry Initiative & Intel Products

One Intel Software & Architecture group
Intel Architecture, Graphics & Software
November 2020



All information provided in this deck is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Programming Challenges

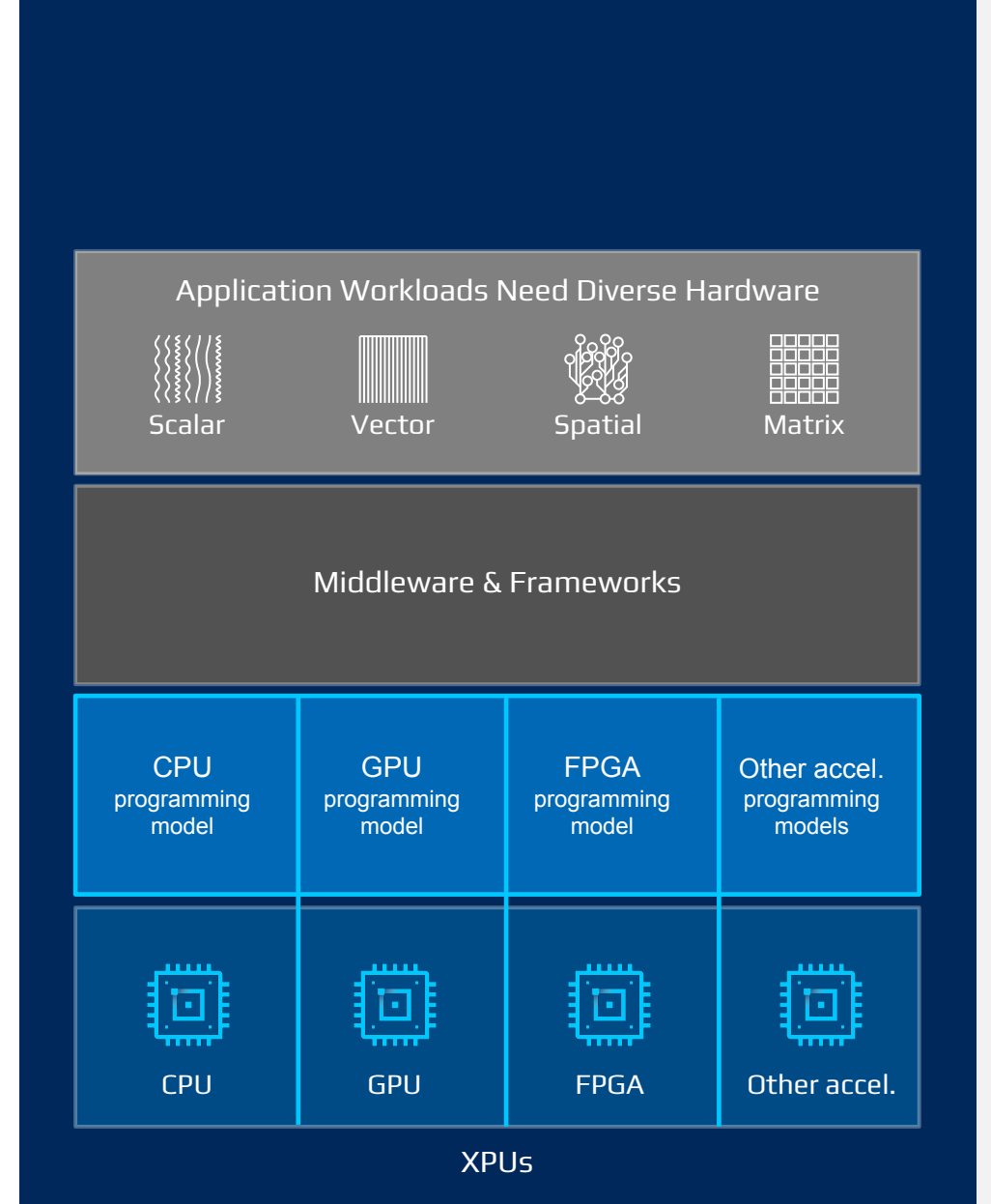
for Multiple Architectures

Growth in specialized workloads

Variety of data-centric hardware required

Separate programming models and toolchains for each architecture are required today

Software development complexity limits freedom of architectural choice



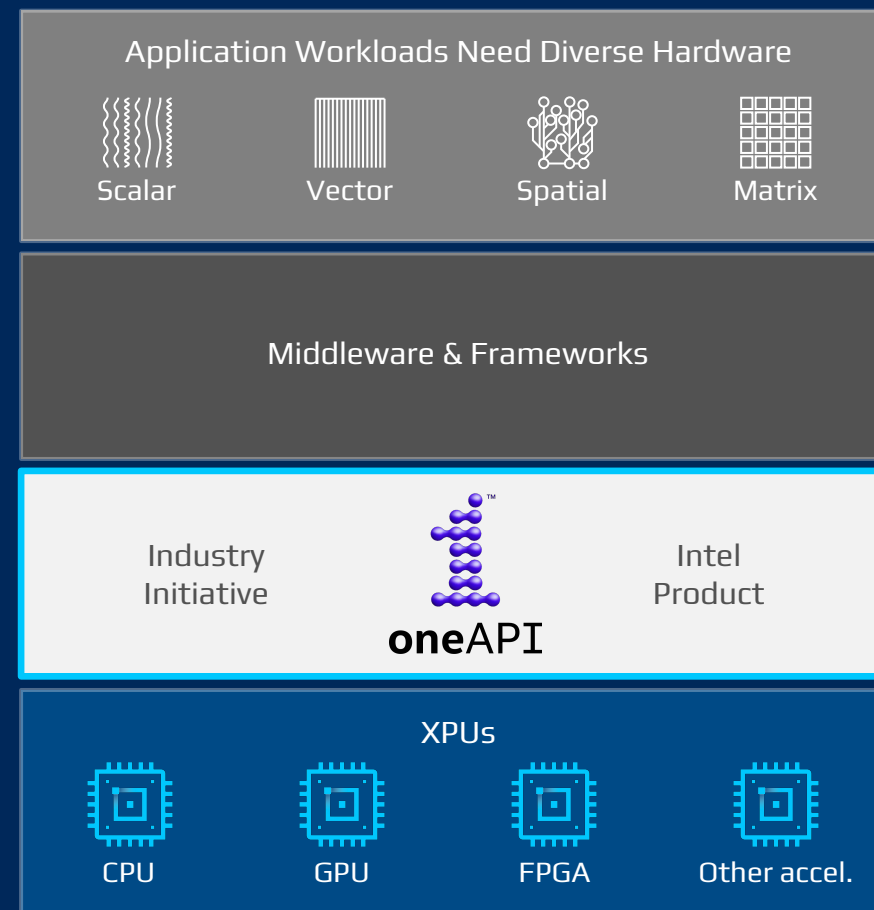
Introducing oneAPI

Cross-architecture programming that delivers freedom to choose the best hardware

Based on industry standards and open specifications

Exposes cutting-edge performance features of latest hardware

Compatible with existing high-performance languages and programming models including C++, OpenMP, Fortran, and MPI



Intel® oneAPI Toolkits

A complete set of proven developer tools expanded from CPU to XPU



Intel® oneAPI Base Toolkit

Native Code Developers



A core set of high-performance tools for building C++, SYCL applications & oneAPI library-based applications

Add-on Domain-specific Toolkits

Specialized Workloads



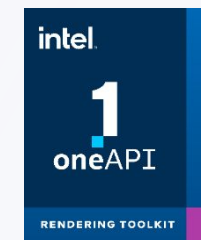
Intel® oneAPI Tools for HPC

Deliver fast Fortran, OpenMP & MPI applications that scale



Intel® oneAPI Tools for IoT

Build efficient, reliable solutions that run at network's edge



Intel® oneAPI Rendering Toolkit

Create performant, high-fidelity visualization applications

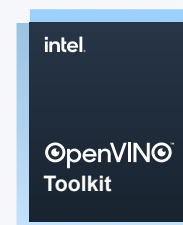
Toolkits powered by oneAPI

Data Scientists & AI Developers



Intel® AI Analytics Toolkit

Accelerate machine learning & data science pipelines with optimized DL frameworks & high-performing Python libraries



Intel® Distribution of OpenVINO™ Toolkit

Deploy high performance inference & applications from edge to cloud

Intel® oneAPI Base Toolkit

Accelerate Data-centric Workloads

A core set of core tools and libraries for developing high-performance applications on Intel® CPUs, GPUs, and FPGAs.

Who Uses It?

- A broad range of developers across industries
- Add-on toolkit users since this is the base for all toolkits

Top Features/Benefits

- Data Parallel C++ compiler, library and analysis tools
- DPC++ Compatibility tool helps migrate existing code written in CUDA
- Python distribution includes accelerated scikit-learn, NumPy, SciPy libraries
- Optimized performance libraries for threading, math, data analytics, deep learning, and video/image/signal processing

Intel® oneAPI Base Toolkit

Direct Programming

Intel® oneAPI DPC++/C++ Compiler

Intel® DPC++ Compatibility Tool

Intel® Distribution for Python

Intel® FPGA Add-on for oneAPI Base Toolkit

API-Based Programming

Intel® oneAPI DPC++ Library oneDPL

Intel® oneAPI Math Kernel Library - oneMKL

Intel® oneAPI Data Analytics Library - oneDAL

Intel® oneAPI Threading Building Blocks - oneTBB

Intel® oneAPI Video Processing Library - oneVPL

Intel® oneAPI Collective Communications Library oneCCL

Intel® oneAPI Deep Neural Network Library - oneDNN

Intel® Integrated Performance Primitives - Intel® IPP

Analysis & debug Tools

Intel® VTune™ Profiler

Intel® Advisor

Intel® Distribution for GDB



Intel® DPC++ Compatibility Tool

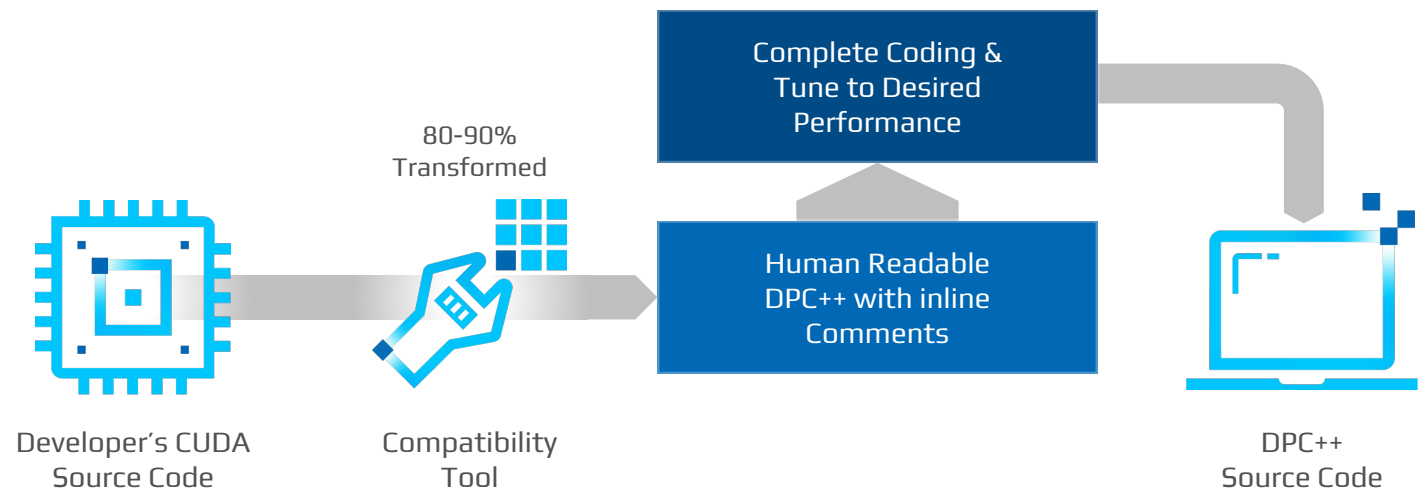
Minimizes Code Migration Time

Assists developers migrating code written in CUDA to DPC++ once, generating **human readable** code wherever possible

~80-90% of code typically migrates automatically

Inline comments are provided to help developers finish porting the application

Intel DPC ++ Compatibility Tool Usage Flow



Intel® VTune™ Profiler

SYCL Profiling-Tune for CPU, GPU & FPGA

Analyze SYCL

See the lines of SYCL that consume the most time

Tune for Intel CPUs, GPUs & FPGAs

Optimize for any supported hardware accelerator

Optimize Offload

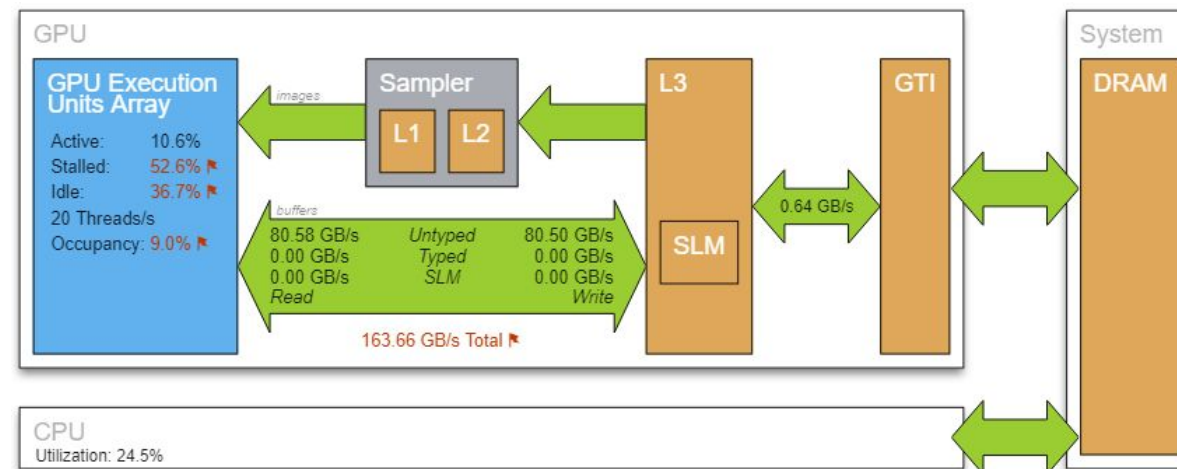
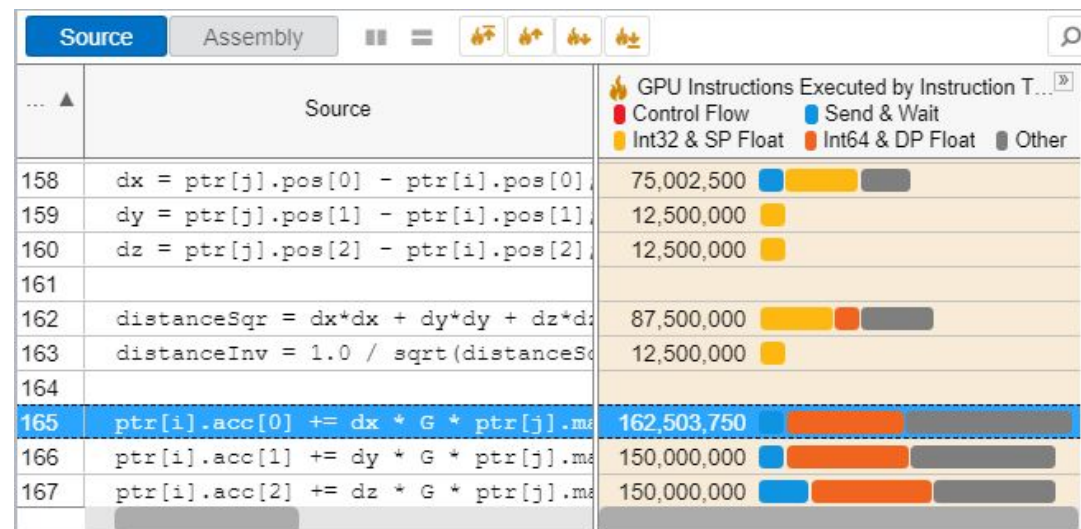
Tune OpenMP offload performance

Wide Range of Performance Profiles

CPU, GPU, FPGA, threading, memory, cache, storage...

Supports Popular Languages

SYCL, C, C++, Fortran, Python, Go, Java, or a mix



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

Intel® Advisor

Design Assistant - Design for Modern Hardware

Offload Advisor

Estimate performance of offloading to an accelerator

Roofline Analysis

Optimize CPU/GPU code for memory and compute

Vectorization Advisor

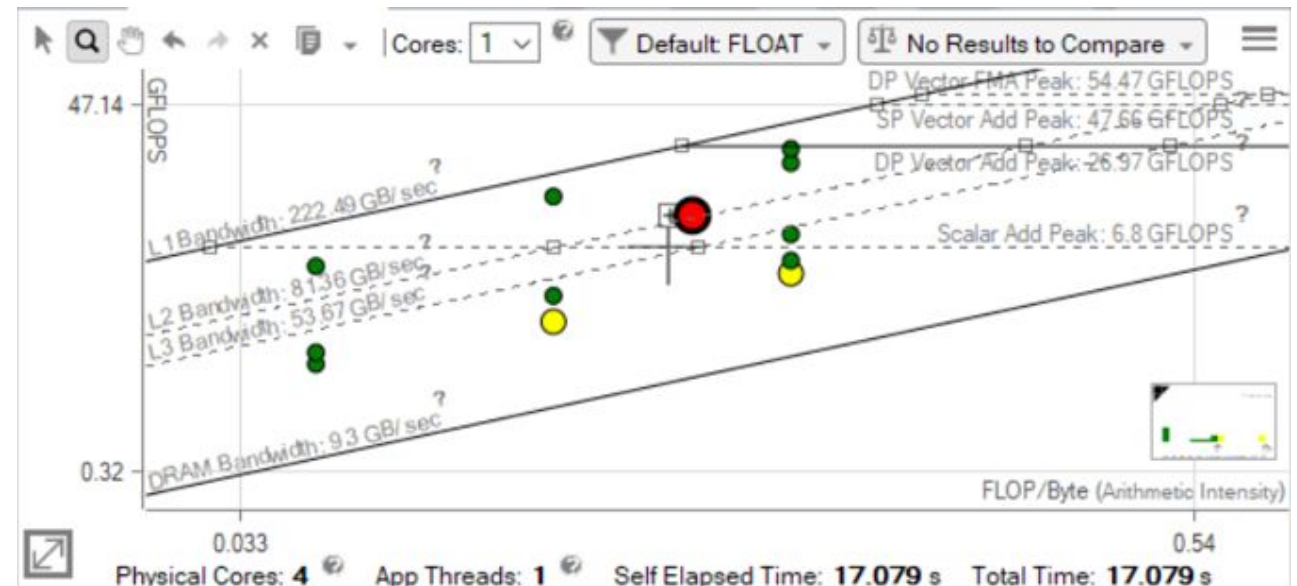
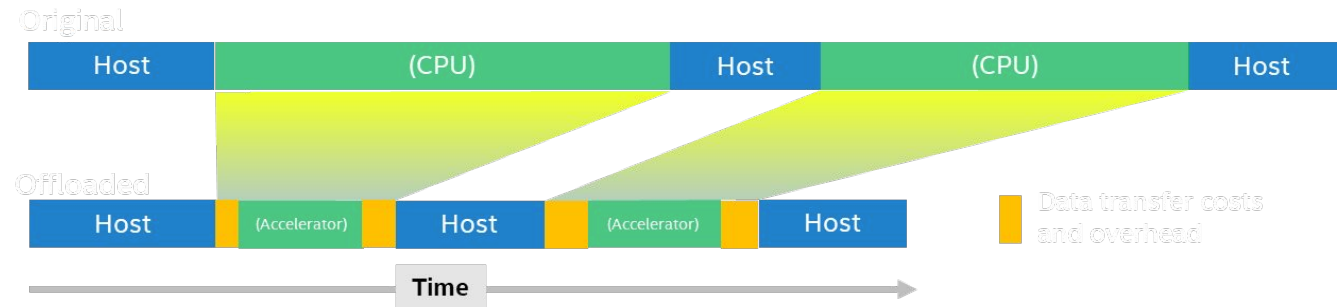
Add and optimize vectorization

Threading Advisor

Add effective threading to unthreaded applications

Flow Graph Analyzer

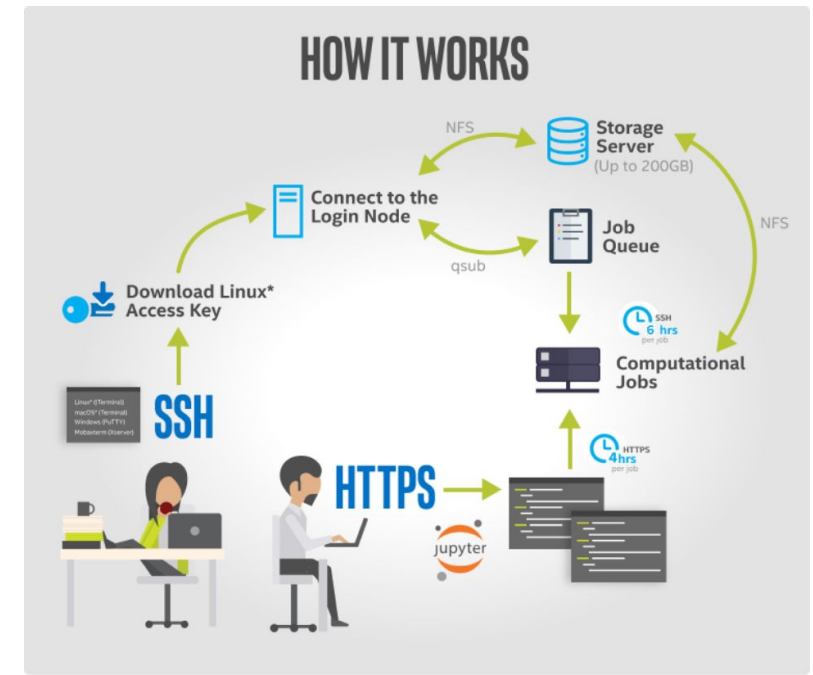
Create and analyze efficient flow graphs



Setup Intel® DevCloud and Jupyter Environment

Intel® Devcloud for oneAPI

- A development sandbox to develop, test and run workloads across a range of Intel CPUs, GPUs, and FPGAs using Intel® oneAPI beta software
- A fast way to start coding
- Try the oneAPI toolkits, compilers, performance libraries, and tools
- Get 120 days of free access to the latest Intel® hardware and oneAPI software
- No downloads; No hardware acquisition; No installation



INTEL® DEVCLOUD
Sign Up

Get to Know Intel oneAPI (Beta) 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, and extensions are totally possible.

Get Access Now
Required Fields(*)

First Name *

Last Name *

Email Address *

Country / Region *
- Select -

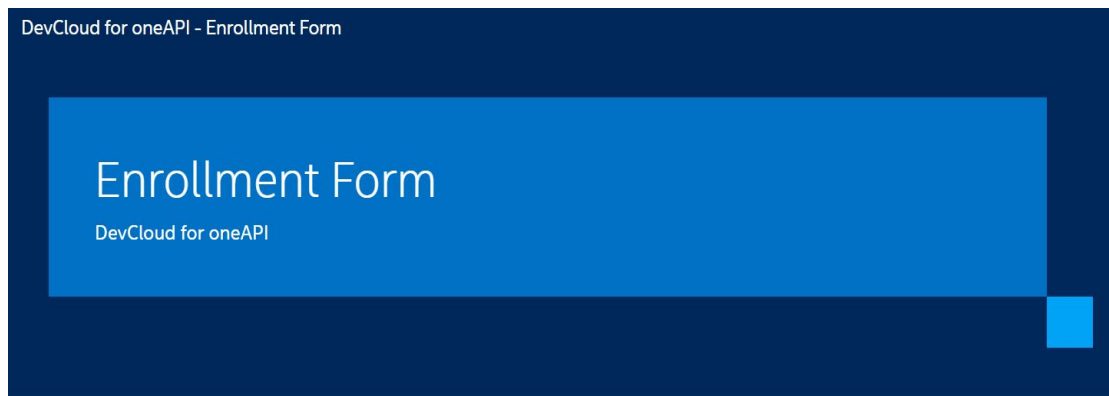
Company or University *

Which hardware and accelerator architectures are you developing for? (Select all that apply.)

- ☐ ASICs (application-specific integrated circuits)
- ☐ CPU
- ☐ FPGA (field-programmable gate array)
- ☐ GPGPU (general-purpose GPU)

Register to Intel® Devcloud for oneAPI

- Step 1: Register or Sign into Intel Developer Zone



Step 1: Sign in or Register

To get an Intel® DevCloud account, you must first create a Basic Intel® Account

Sign in

Register

- Step 2: Activate Intel Devcloud Account

Step 2: Activate Intel® DevCloud for oneAPI

To get free access, tell us a bit more about yourself and how you would like to use the Intel DevCloud.

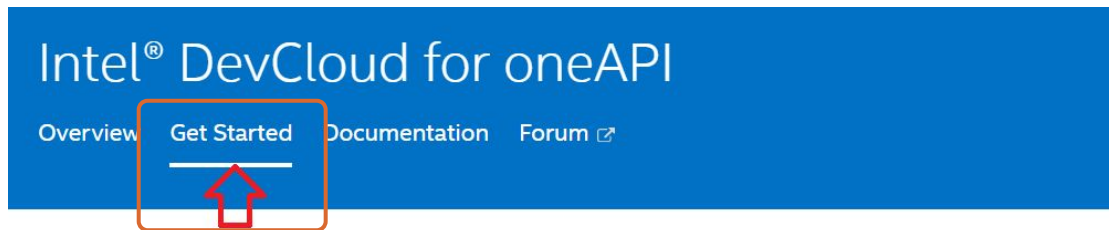
Required Fields(*)

* First Name First Name	* Country/region Please select a country/region
* Last Name Last Name	* Company or University Company or Academic Institution
* Email Address Business Email	* What type of developer are you? -Select-
* Which hardware and accelerator architecture are you developing for?(Select all that apply) <input type="checkbox"/> ASICs (application-specific integrated circuits) <input type="checkbox"/> CPU <input type="checkbox"/> FPGA (field-programmable gate array) <input type="checkbox"/> GPGPU (general-purpose GPU) <input type="checkbox"/> GPU <input type="checkbox"/> Integrated Graphics	
Do you have an event code provided by Intel? (Optional) 	

<https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html>


Get Started with Intel® Devcloud for oneAPI

- Step 3: Click on Get Started button



Explore Intel oneAPI Toolkits in the DevCloud

These toolkits are for performance-driven applications—HPC, IoT, advanced rendering, deep learning—to see what it includes, explore training modules, and go deeper with developer guides.



Intel® oneAPI Base Toolkit

Build and deploy high-performance, data-centric applications across diverse hardware

[Get Started with your first Sample](#) [View Training Modules](#)

The toolkit includes:

- Step 4: Scroll Down to the bottom of the page and click on Launch JupyterLab

Connect with Jupyter* Lab



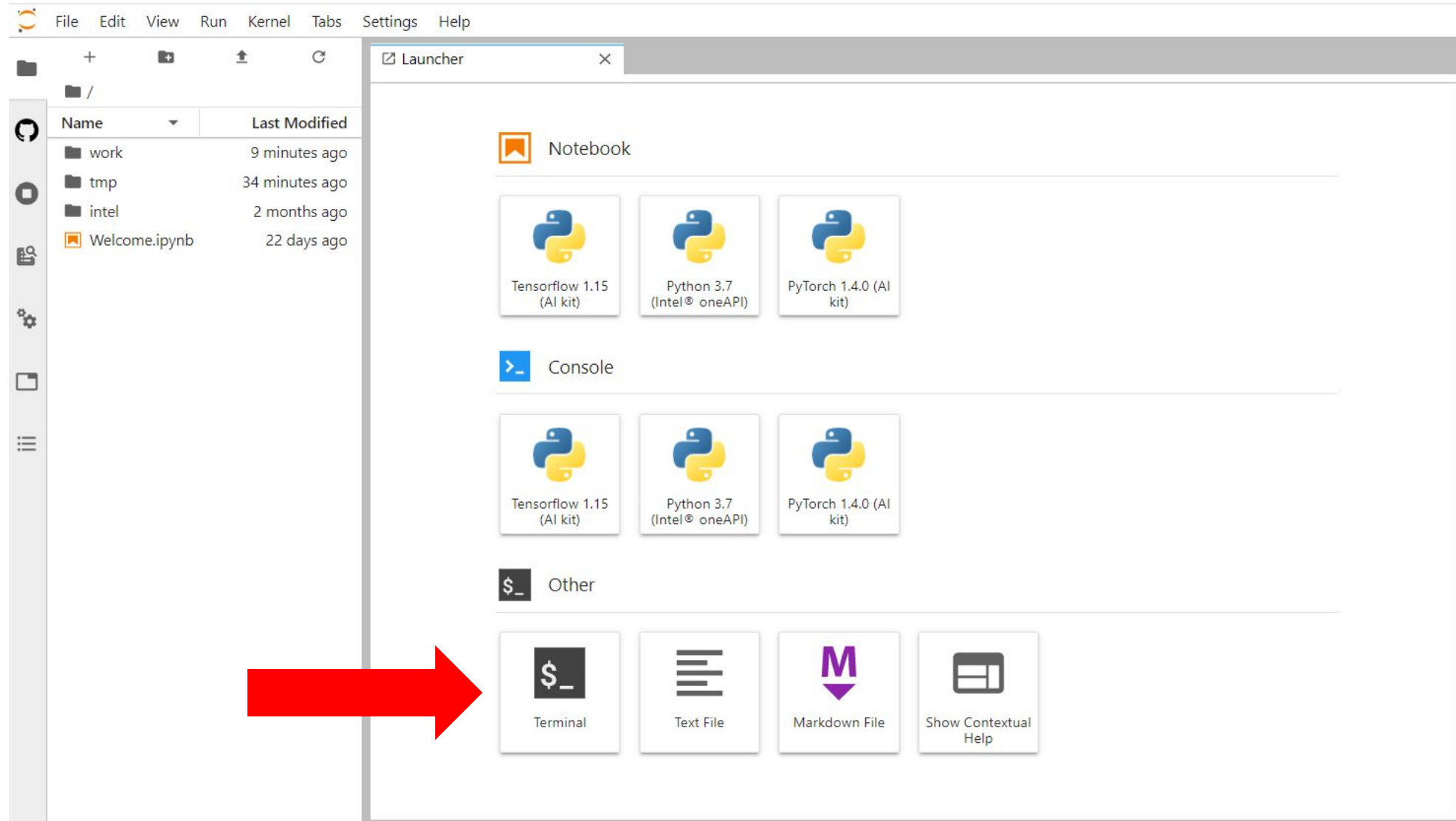
Connect with Jupyter* Notebook

Use Jupyter Notebook to learn about how oneAPI can solve the challenges of programming in a heterogeneous world and understand the Data Parallel C++ (DPC++) language and programming model.



Setup Intel® DevCloud and Jupyter Environment

Launch Jupyter and select Terminal



SYCL essentials Course



A COMPLETE DPC++ PROGRAM

Single source

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

Familiar C++

- Library constructs add functionality, such as:

Construct	Purpose
queue	Work targeting
buffer	Data management
parallel_for	Parallelism

```

#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 );

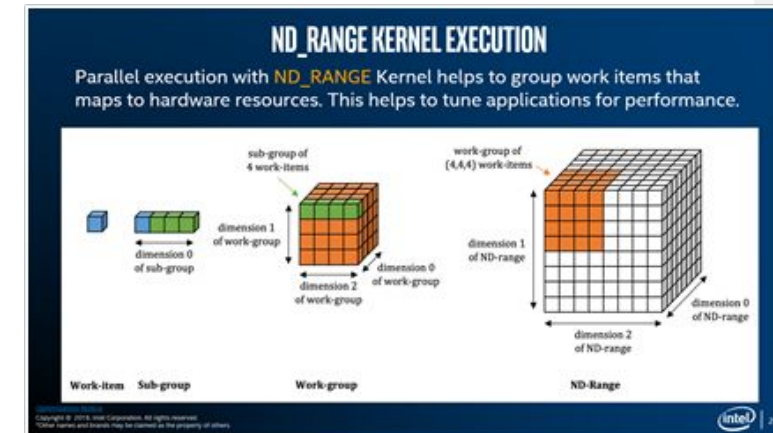
    queue().submit([&](handler& h) {
        auto out =
        A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

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

    return 0;
}
    
```

Host code
Accelerator device code
Host code

Copyright © 2019 Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.



INTEL OFFLOAD ADVISOR (BETA)

- Starting from a baseline binary (running on CPU):
- Helps defining which sections of the code should run on a given accelerator
- Provides performance projection on accelerators (currently gen9 and gen11)

Copyright © 2019 Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.

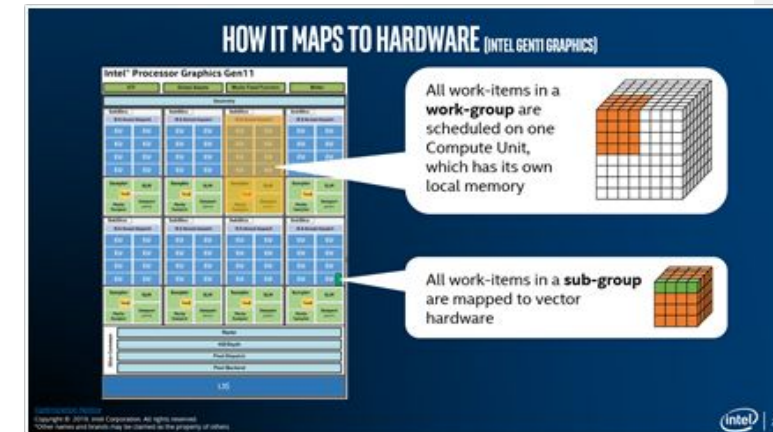
INTEL® VTUNE™ PROFILER: HARDWARE ANALYSIS EXTENDED TO SVMs ARCHITECTURE

DPC++ kernels and their hardware metrics

GPU hardware metrics
GPU compute Shader
GPU L3 Cache misses
GPU Texture Memory

GPU queue and utilization

Copyright © 2019 Intel Corporation. All rights reserved. Other names and brands may be claimed as the property of others.



SYCL Essentials Course Curriculum provides 20 hours of training and exercises using Jupyter Notebooks integrated with Intel® DevCloud

oneAPI's implementation of SYCL

Standards-based, Cross-architecture Language
ISO C++ and Khronos SYCL

Parallelism, productivity and performance for CPUs and Accelerators

- Delivers accelerated computing by exposing hardware features
- Allows code reuse across hardware targets, while permitting custom tuning for specific accelerators
- Provides an open, cross-industry solution to single architecture proprietary lock-in

Based on C++ and SYCL

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

Community Project to drive language enhancements

- Provides extensions to simplify data parallel programming
- Continues evolution through open and cooperative development

Apply your skills to the next innovation, not rewriting software for the next hardware platform

Direct Programming: oneAPI's implementation of SYCL

Community Extensions

Khronos SYCL

ISO C++

What is oneAPI's implementation of SYCL

oneAPI's implementation of SYCL

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

Extends SYCL* standard

Enhance Productivity

- Simple things should be simple to express
- Reduce verbosity and programmer burden

Enhance Performance

- Give programmers control over program execution
- Enable hardware-specific features

Fast-moving open collaboration feeding into the SYCL* standard

- Open source implementation with goal of upstream LLVM
- DPC++ extensions aim to become core SYCL*, or Khronos* extensions

A Complete SYCL Program

Single source

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

Familiar C++

- Library constructs add functionality, such as:

Construct	Purpose
queue	Work targeting
malloc_shared	Data management
parallel_for	Parallelism

Host code
Accelerator device code

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
    queue q;
    int *data = malloc_shared<int>(N, q);
    q.parallel_for(N, [=](auto i) {
        data[i] = i;
    }).wait();
    for (int i=0; i<N; i++) std::cout << data[i] << "\n";
    free(data, q);
    return 0;
}
```

Host code

SYCL Program Structure

- Agenda

- Deciding where code is run
- Data transfers and synchronization
- SYCL execution model and memory model

- Hands On

- Complex Multiplication

Buffer Memory Model

Buffers encapsulate data shared between host and device.

Accessors provide access to data stored in buffers and create data dependences in the graph.

Unified Shared Memory (USM) provides an alternative pointer-based mechanism for managing memory;

```
queue q;  
std::vector<int> v(N, 10);  
{  
    buffer buf(v);  
    q.submit([&](handler& h) {  
        accessor a(buf, h, write_only);  
        h.parallel_for(N, [=](auto i) { a[i] = i; });  
    });  
}  
for (int i = 0; i < N; i++) std::cout << v[i] << " ";
```


Submitting to a Device

- A **device** represents a specific accelerator in the system.
- Work is not submitted to devices directly, but to a **queue** associated with the device.
- Creating a queue for a specific device requires a **device_selector**.

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

Important Classes in SYCL

Class	Functionality
<code>sycl::device</code>	Represents a specific CPU, GPU, FPGA or other device that can execute SYCL kernels.
<code>sycl::queue</code>	Represents a queue to which kernels can be submitted (enqueued). Multiple queues may map to the same <code>sycl::device</code> .
<code>sycl::buffer</code>	Encapsulates an allocation that the runtime can transfer between host and device.
<code>sycl::handler</code>	Used to define a command-group scope that connects buffers to kernels.
<code>sycl::accessor</code>	Used to define the access requirements of specific kernels (e.g. read, write, read-write).
<code>sycl::range</code> , <code>sycl::nd_range</code> <code>sycl::id</code> , <code>sycl::item</code> , <code>sycl::nd_item</code>	Representations of execution ranges and individual execution agents in the range.

Accessor Modes

Access Mode	Description
read_only	Read only Access
write_only	Write-only access. Previous contents not discarded
read_write	Read and Write access

Parallel Kernels

- Parallel Kernel allows 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  
});
```

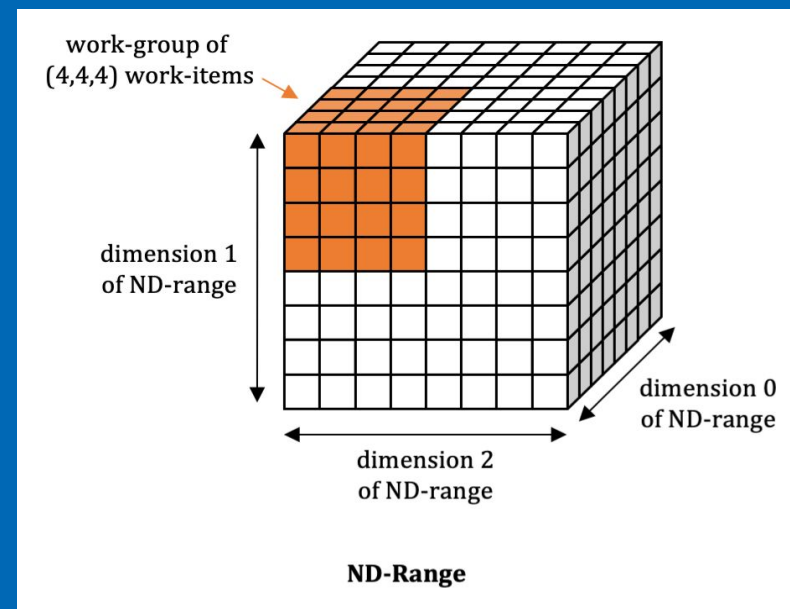
```
h.parallel_for(range<1>(1024), [=](item<1> item){  
    auto idx = item.get_id();  
    auto R = item.get_range();  
    // CODE THAT RUNS ON DEVICE  
});
```

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 express 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.

SYCL Code Anatomy

```
void dpcpp_code(int* a, int* b, int* c) {  
    // Setting up a device queue  
    queue q;  
    // 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 allocated in global memory  
        accessor A(buf_a, h, read_only);  
        accessor B(buf_b, h, read_only);  
        accessor C(buf_c, h, write_only);  
        //Specify the device kernel body as a lambda function  
        h.parallel_for(range<1>(N), [=](auto 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 group for (asynchronous) execution

Step 4: create accessors describing how buffer is used on the device

Step 5: specify kernel function and launch parameters (e.g. group size)

Step 6: specify code to run on the device

Kernel invocations
are executed in
parallel

Kernel is invoked
for each element of
the range

Kernel invocation
has access to the
invocation id

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

Buffer: sub_buffers

A sub-buffer requires three things, a reference to a parent buffer, a base index, and the range of the sub-buffer.

The main advantage of using the sub-buffers is different kernels can operate on different sub buffers concurrently.

Sub Buffer for one dimensional buffer

Sub buffer for a 2-dimensional buffer

```
buffer B(data, range(N));
```

```
buffer<int> B1(B, 0, range{ N / 2 });
```

```
buffer<int> B2(B, 32, range{ N / 2 });
```

```
buffer<int, 2> b10{range{2, 5}};
```

```
buffer b11{b10, id{0, 0}, range{1, 5}};
```

```
buffer b12{b10, id{1, 0}, range{1, 5}};
```

Sub Buffers

Buffer for Vectors

Create sub buffers B1
and B2

Submit q1 using B1

Submit q2 using B2

Create Host accessors

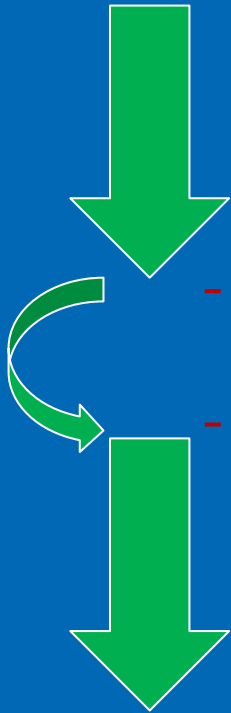
```
int main() {  
    const int N = 64;    const int num1 = 2;    const int num2 = 3;  
    int data[N];  
    for (int i = 0; i < N; i++) data[i] = i;    for (int i = 0; i < N; i++) std::cout << data[i] << " ";  
    buffer B(data, range(N));  
    buffer<int> B1(B, 0, range{ N / 2 });  
    buffer<int> B2(B, 32, range{ N / 2 });  
    queue q1;  
    q1.submit([&](handler& h) {  
        accessor a1(B1, h);  
        h.parallel_for(N/2, [=](auto i) { a1[i] *= num1; });  
    });  
    queue q2;  
    q2.submit([&](handler& h) {  
        accessor a2(B2, h);  
        h.parallel_for(N/2, [=](auto i) { a2[i] *= num2; });  
    });  
    host_accessor b1(B1, read_only);  
    host_accessor b2(B2, read_only);  
    return 0;  
}
```

Asynchronous Execution

Host

Host code execution

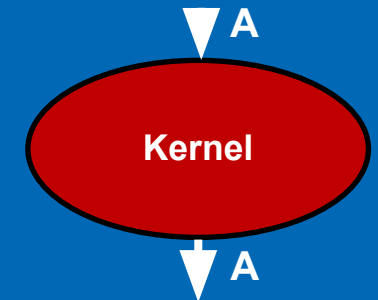
Enqueues kernel to graph, and keeps going



```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
    std::vector<int> data(N);
    {
        buffer A(data);
        queue q;
        q.submit([&](handler& h) {
            accessor out(A, h, write_only);
            ----- h.parallel_for(N, [=](auto i) {
                out[i] = i;
            });
        });
    }
    for (int i=0; i<N; ++i) std::cout << data[i];
}
```

Graph

Graph executes asynchronously to host program



Asynchronous Execution

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

```
    q.submit([&](handler& h) {  
        accessor out(A, h, write_only);  
        h.parallel_for(R, [=](id<1> i) {  
            out[i] = i; }); });
```

} Kernel 1

```
    q.submit([&](handler& h) {  
        accessor out(A, h, write_only);  
        h.parallel_for(R, [=](id<1> i) {  
            out[i] = i; }); });
```

} Kernel 2

```
    q.submit([&](handler& h) {  
        accessor out(B, h, write_only);  
        h.parallel_for(R, [=](id<1> i) {  
            out[i] = i; }); });
```

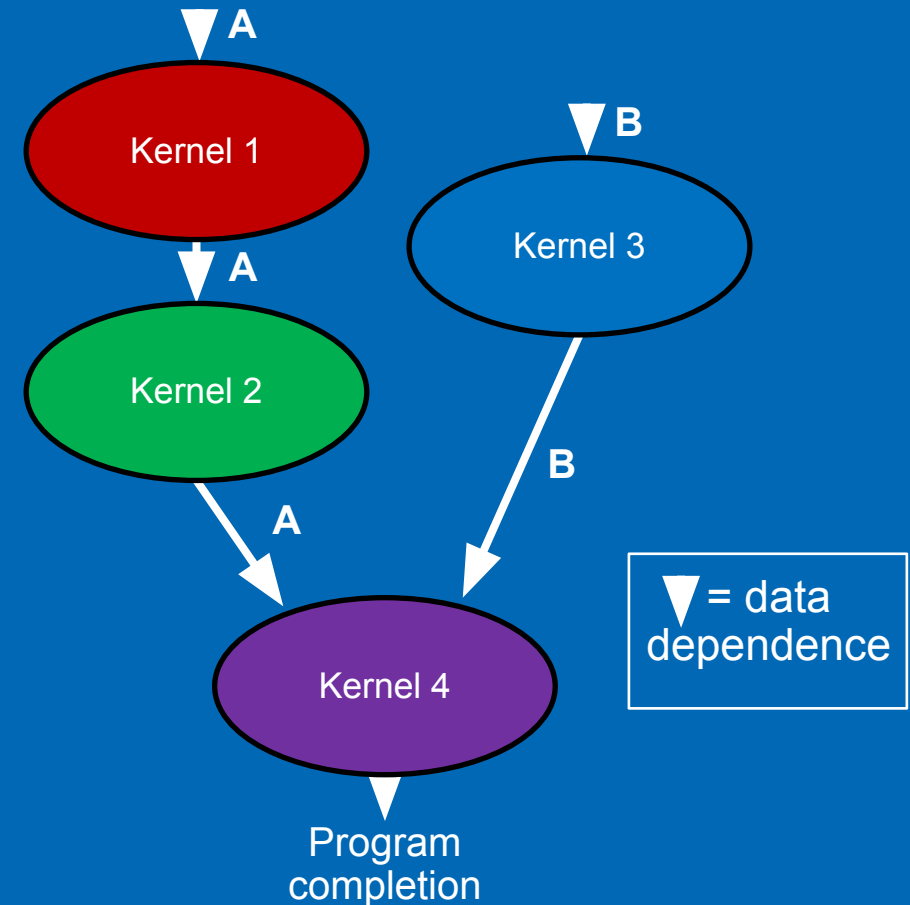
} Kernel 3

```
    q.submit([&](handler& h) {  
        accessor in(A, h, read_only);  
        accessor inout(B, h);  
        h.parallel_for(R, [=](id<1> i) {  
            inout[i] *= in[i]; }); });
```

} Kernel 4

}

Data and control dependences
are resolved by the runtime



Synchronization – Host Accessors

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

int main() {
    std::vector<double> v(N, 10);
    queue q;

    buffer buf(v);
    q.submit([&](handler& h) {
        accessor a(buf, h)
        h.parallel_for(N, [=](auto i) {
            a[i] -= 2;
        });
    });

    host_accessor b(buf, read_only);
    for (int i = 0; i < N; i++)
        std::cout << b[i] << "\n";
    return 0;
}
```

Buffer takes **ownership** of the **data** stored in vector.

Creating host accessor is a **blocking call** and will only return after all enqueued kernels that modify the same buffer in any queue completes execution and the **data is available to the host** via this host accessor.

Synchronization – Buffer Destruction

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

void dpcpp_code(std::vector<double> &v, queue &q){
    buffer buf(v);
    q.submit([&](handler& h) {
        accessor a(buf, h);
        h.parallel_for(N, [=](auto 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;
}
```

Buffer creation happens within a **separate function scope**.

When execution advances beyond this function scope, **buffer destructor is invoked** which relinquishes the ownership of data and **copies back the data to the host** memory.

Custom Device Selector

The following code shows derived **device_selector** that employs a device selector heuristic. The selected device prioritizes a GPU device because the integer rating returned is higher than for CPU or other accelerator.

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class my_device_selector : public device_selector {
public:
    int operator()(const device& dev) const override {
        int rating = 0;
        if (dev.is_gpu() & (dev.get_info<info::device::name>().find("Intel") != std::string::npos))
            rating = 3;
        else if (dev.is_gpu()) rating = 2;
        else if (dev.is_cpu()) rating = 1;
        return rating;
    };
};
int main() {
    my_device_selector selector;
    queue q(selector);
    std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
    return 0;
}
```

Execution Graph Scheduling

Mechanism to achieve proper sequencing of kernels, and data movement in a SYCL application.

- Read-after-Write (RAW) : Occurs when one task needs to read data produced by a different task.
- Write-after-Read (WAR) : Occurs when one task needs to update data after another task has read it.
- Write-after-Write (WAW) : Occurs when two tasks try to write the same data.


```

int main() {
    queue Q;
    //Create Buffers
    buffer A{a};
    buffer B{b};
    buffer C{c};

    Q.submit([&](handler &h) {
        accessor accA(A, h, read_only);
        accessor accB(B, h, write_only);
        h.parallel_for( // computeB
            N, [=](id<1> i) { accB[i] = accA[i] + 1; });
    });

    Q.submit([&](handler &h) {
        accessor accA(A, h, read_only);
        h.parallel_for( // readA
            N, [=](id<1> i) {
                // Useful only as an example
                int data = accA[i];
            });
    });

    Q.submit([&](handler &h) {
        // RAW of buffer B
        accessor accB(B, h, read_only);
        accessor accC(C, h, write_only);
        h.parallel_for( // computeC
            N, [=](id<1> i) { accC[i] = accB[i] + 2; });
    });

    // read C on host
    host_accessor host_accC(C, read_only);
    std::cout << "\n";
    return 0;
}

```

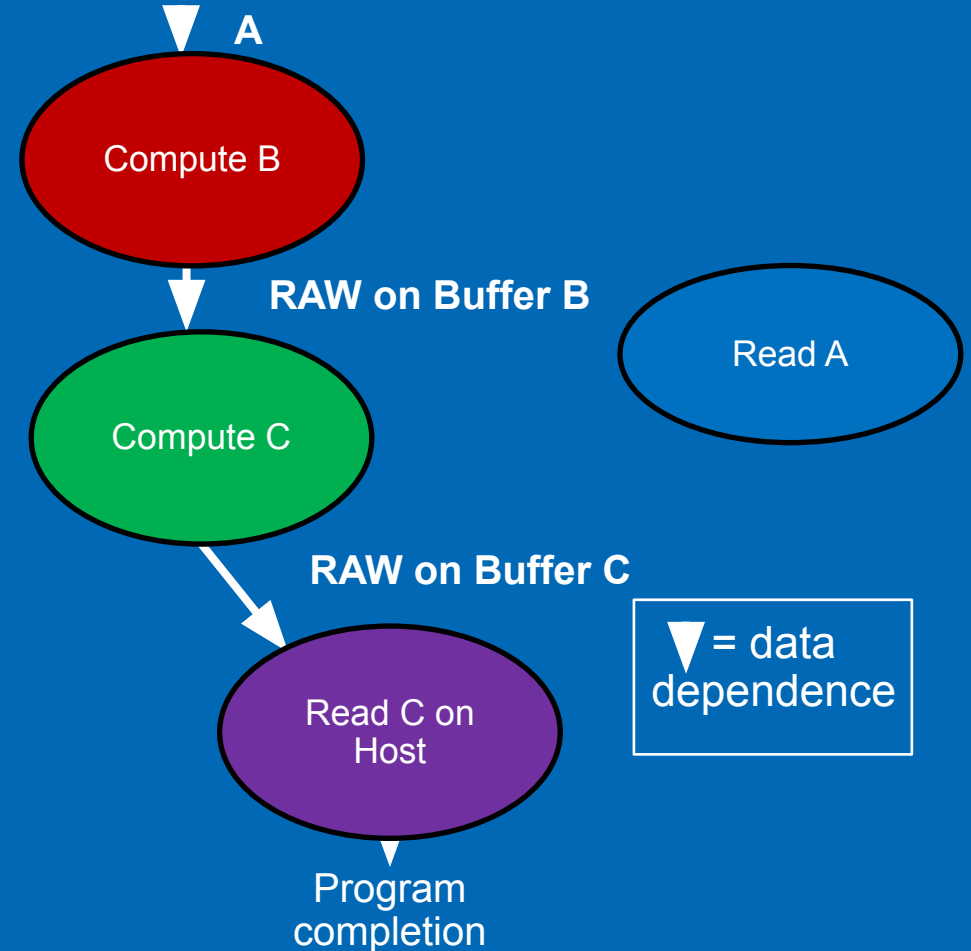
Kernel 1

Kernel 2

Kernel 3

Read after Write (RAW)

Automatic data and control dependence resolution!



Write After Read and Write after Write

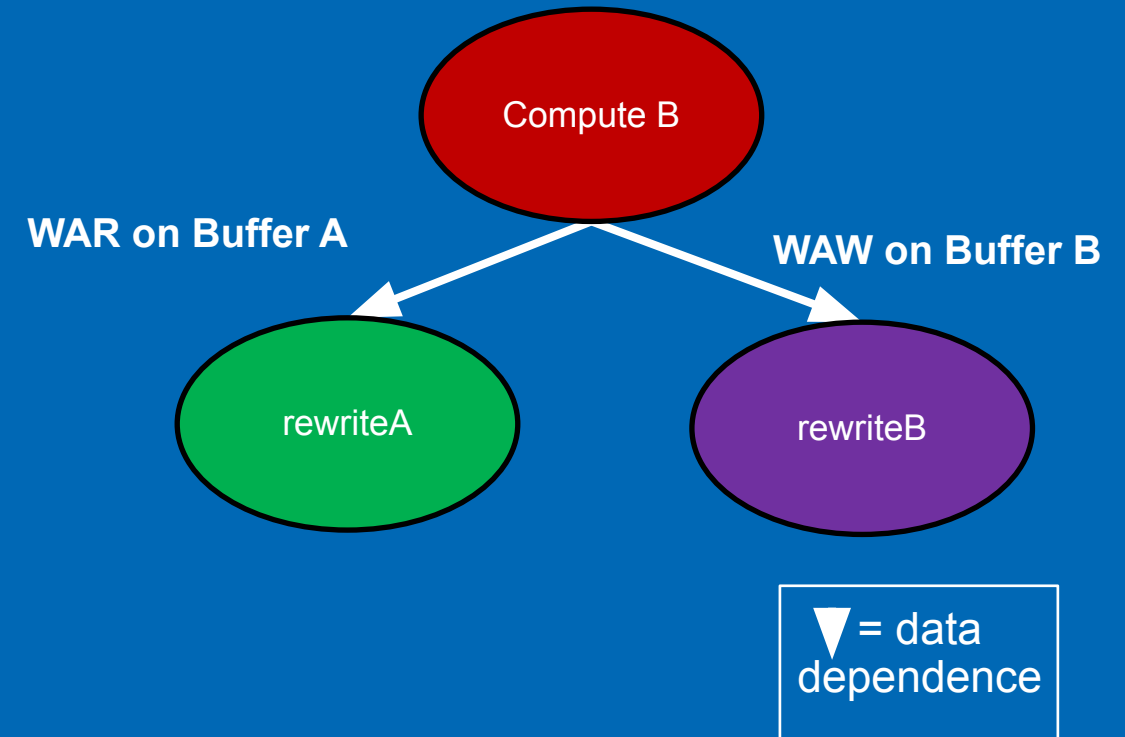
```
queue Q;  
buffer A{a};  
buffer B{b};  
Q.submit([&](handler &h) {  
    accessor accA(A, h, read_only);  
    accessor accB(B, h, write_only);  
    h.parallel_for( // computeB  
        N, [=](id<1> i) {  
            accB[i] = accA[i] + 1;  
        });  
});  
Q.submit([&](handler &h) {  
    // WAR of buffer A  
    accessor accA(A, h, write_only);  
    h.parallel_for( // rewriteA  
        N, [=](id<1> i) {  
            accA[i] = 21 + 21;  
        });  
});  
Q.submit([&](handler &h) {  
    // WAW of buffer B  
    accessor accB(B, h, write_only);  
    h.parallel_for( // rewriteB  
        N, [=](id<1> i) {  
            accB[i] = 30 + 12;  
        });  
});  
host_accessor host_accA(A, read_only);  
host_accessor host_accB(B, read_only);
```

} Kernel 1

} Kernel 2

} Kernel 3

Automatic data and control
dependence resolution!



Linear dependency chain graphs and Y pattern Graphs

- Linear dependence chains where one task executes after another
 - First node represents the initialization of data.
 - Second node presents the reduction operation that will accumulate the data.
- “Y” pattern we independently initialize two different pieces of data.
 - An addition kernel will sum the two vectors together.
 - Finally, the last node in the graph accumulates the result into a single value.

Linear Dependence Using In-order queue

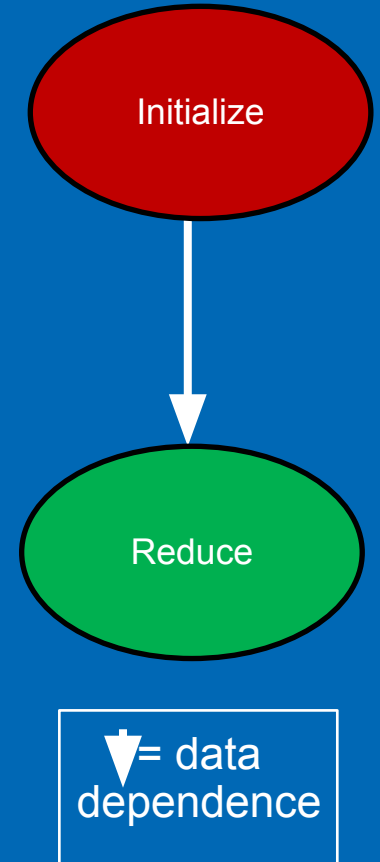
Create In-order
queue

Initialize the data
in Kernel 1

Kernel 2 sums up
the elements

```
constexpr int N = 42;

int main() {
    queue Q{property::queue::in_order()};
    int *data = malloc_shared<int>(N, Q);
    Q.parallel_for(N, [=](id<1> i) { data[i] = 1; });
    Q.single_task([=]() {
        for (int i = 1; i < N; i++)
            data[0] += data[i];
    });
    Q.wait();
    assert(data[0] == N);
    return 0;
}
```



Y Pattern using in-order queues

We can see a “Y” pattern using in-order queues in the below example

In-Order Queue

```
constexpr int N = 42;

int main() {
    queue Q{property::queue::in_order()};

    int *data1 = malloc_shared<int>(N, Q);
    int *data2 = malloc_shared<int>(N, Q);

    Q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });
    Q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });
    Q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

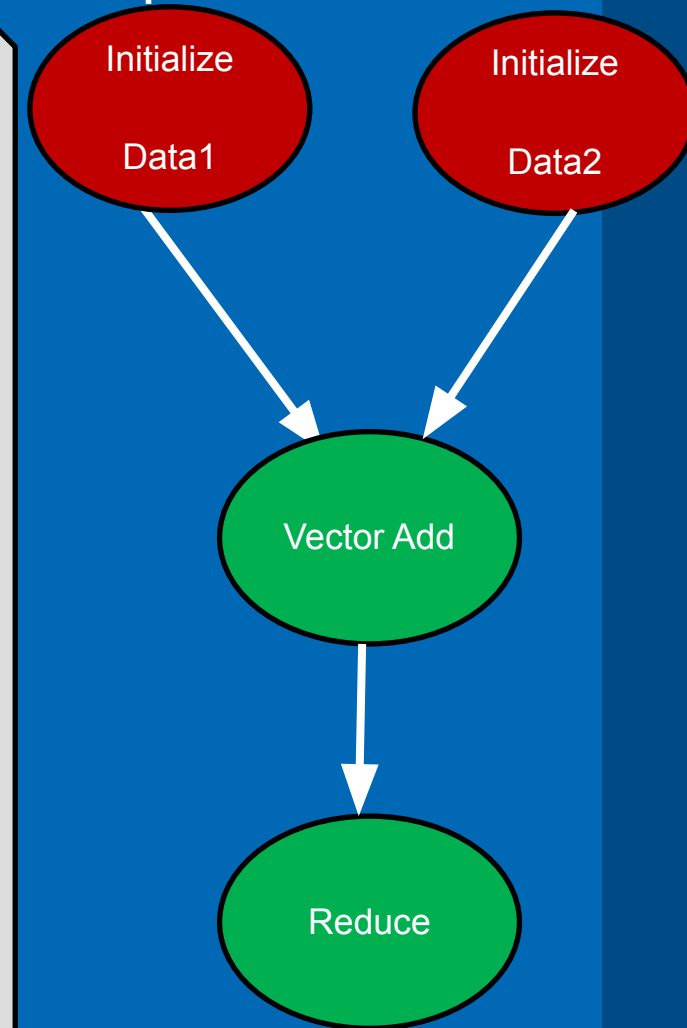
    Q.single_task([=]() {
        for (int i = 1; i < N; i++)
            data1[0] += data1[i];

        data1[0] /= 3;
    });
    Q.wait();

    assert(data1[0] == N);
    return 0;
}
```

Kernel 3 is dependant
on Kernel1 and Kernel2

The final kernel sums
up the elements of the
first array



Recap

- oneAPI solves the challenges of programming in a heterogeneous world
- Take advantage of oneAPI solutions to enable your workflows
- Use the Intel® DevCloud to test-drive oneAPI tools and libraries
- Introduced to SYCL language and programming model
- Important Classes for SYCL application
- Device selection and offloading kernel workloads
- SYCL Buffers, Accessors, Command Group handler, lambda code as kernel
- Utilize different types of data dependences that are important for ensuring execution of graph scheduling



Notices &

- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.
- The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.
- 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.
- INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Copyright ©, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and OpenVINO are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804