What is oneAPI and Data Parallel C++?
Introduction to oneAPI

• Agenda
  a) Introduction & Overview to oneAPI
  b) Introduction to the Intel® DevCloud
  c) Introduction to Jupyter notebooks used for training
  d) Introduction to Data Parallel C++
  e) DPC++ Program Structure

• Hands On
  • Introduction to DPC++ - Simple
  • Complex multiplication
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 Data Parallel C++ (DPC++) language and programming model

Use device selection to offload kernel workloads

Build a sample DPC++ application through hands-on lab exercises
oneAPI: Industry Initiative & Intel Products

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

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

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
oneAPI Industry Initiative

Break the Chains of Proprietary Lock-in

A cross-architecture language based on C++ and SYCL standards

Powerful libraries designed for acceleration of domain-specific functions

Low-level hardware abstraction layer

Open to promote community and industry collaboration

Enables code reuse across architectures and vendors

The productive, smart path to freedom for accelerated computing from the economic and technical burdens of proprietary programming models
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++, Data Parallel C++ 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

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

Learn More: intel.com/oneAPI-BaseKit
Three components:

1. **Standard C++ APIs**: Tested and supported within DPC++ kernels
2. **Parallel STL**: C++17 algorithms extended with DPC++ execution policies
3. **STL Extensions**: Additional algorithms, classes and iterators

```cpp
sycl::queue q;
std::vector<int> v(N);
std::sort(oneapi::dpl::execution::make_device_policy(q), v.begin(), v.end());
```

Recommended for codes using C++17 algorithms, or libraries like Thrust

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® VTune™ Profiler
DPC++ Profiling—Tune for CPU, GPU & FPGA

Analyze Data Parallel C++ (DPC++)
See the lines of DPC++ 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
DPC++, C, C++, Fortran, Python, Go, Java, or a mix

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

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

There will still be a need to tune for each architecture.
SETUP INTEL® DEV CLOUD 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
Register to Devcloud

- **Step 1: Register or Sign into Intel Developer Zone**

- **Step 2: Activate Intel Devcloud Account**
Get Started with Devcloud

- **Step 3:** Click on Get Started button

- **Step 4:** Scroll Down to the bottom of the page and click on Launch JupyterLab
Setup Intel® DevCloud and Jupyter Environment
Launch Jupyter and select Terminal
Commands to input in terminal

Please execute the following commands in the Jupyter Terminal window

`/data/oneapi_workshop/get_jupyter_notebooks.sh`

This command copies workshop into the user directory
Select **Welcome.ipynb**

---

**oneAPI Essentials Modules**

The concepts build on top of each other introducing and reinforcing the concepts of Data Parallel C++.

**Module 0 - Introduction to Jupyter Notebook (Optional)**

*Optional* This module explains how to use Jupyter Notebook which is used in all of the modules to edit and run coding exercises, this can be skipped if you are already familiar with using Jupyter Notebooks.

**Module 1 - Introduction to oneAPI and DPC++**

These initial hands-on exercises introduce you to DPC++ and the goal of oneAPI. In addition, it familiarizes you with the use of Jupyter notebooks as a front-end for all training exercises. This workshop is designed to be used on the DevCloud and includes details on how to submit batch jobs on DevCloud environment.

**Module 2 - DPC++ Program Structure**

These hands-on exercises present six basic DPC++ programs that illustrate the elements of a DPC++ application. You can modify the source code in some of the exercises to become more familiar with DPC++ programming concepts.
DPC++ Essentials Course Curriculum provides 20 hours of training and exercises using Jupyter Notebooks integrated with Intel® DevCloud.
Qsub

• qsub can be used to submit jobs to the DevCloud job queue
• Jobs run asynchronously and report status upon completion
• The traditional way to execute qsub is to pass it a script:
  “qsub <script.sh>”
• qsub requires absolute paths, e.g. /bin/ls
• qsub –w $PWD – Runs in current folder
• Output file is <scriptname>.o<jobid>
QSTAT/QDEL

- qstat displays running jobs
- qdel <jobid> deletes pending jobs
Interactive shells

• Getting an interactive shell
  • qsub –l

• Requesting an iGPU/FPGA node
  • qsub -I -l nodes=1:gpu:ppn=2
  • clinfo – lists iGPU info
Hands-on Coding on Intel DevCloud

Run Simple DPC++ Program
Data Parallel C++

Standards-based, Cross-architecture Language
DPC++ = 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

Direct Programming: Data Parallel C++

Community Extensions
Khronos SYCL
ISO C++
What is Data Parallel C++?

Data Parallel C++

= C++ and SYCL* standard and extensions

Based on modern C++

- C++ productivity benefits and familiar constructs

Standards-based, cross-architecture

- Incorporates the SYCL standard for data parallelism and heterogeneous programming
DPC++ 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

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

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

```
#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;
}
```
DPC++ Program Structure

• Agenda
  • Deciding where code is run
  • Data transfers and synchronization
  • DPC++ 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:

```cpp
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] << " ";
```
## Important Classes in DPC++

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

<table>
<thead>
<tr>
<th>Access Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_only</td>
<td>Read only Access</td>
</tr>
<tr>
<td>write_only</td>
<td>Write-only access. Previous contents not discarded</td>
</tr>
<tr>
<td>read_write</td>
<td>Read and Write access</td>
</tr>
</tbody>
</table>
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];
    });
  });
}

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

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
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**.

```cpp
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;
```
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.

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

```cpp
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

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

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

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

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

- The entire iteration space is divided into smaller groups called **work-groups**, work-items within a work-group are scheduled on a single compute unit on hardware.

- The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution.
The functionality of nd\_range kernels is exposed via \texttt{nd\_range} and \texttt{nd\_item} classes

- \texttt{nd\_range} class represents a grouped execution range using global execution range and the local execution range of each work-group.
- \texttt{nd\_item} class represents an individual instance of a kernel function and allows to query for work-group range and index.

```cpp
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
});
```

**global size**

**work-group size**
Asynchronous Execution

```cpp
#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];
}
```
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; }); });

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

    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 1
Kernel 2
Kernel 3
Kernel 4

Data and control dependences are resolved by the runtime
Synchronization – Host Accessors

```cpp
#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

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.

```cpp
#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;
}
```
Hands-On: Complex Number Multiplication

• In this lab we provide with the source code that computes multiplication of two complex numbers where Complex class is the definition of a custom type that represents complex numbers.

• In this example the student will learn how to create a custom device selector and to target GPU or CPU of a specific vendor. The student will also learn how to pass in a vector of custom Complex class objects in parallel and needs to modify the source code to setup a write accessor and call the Complex class member function as kernel to compute the multiplication.
Hands-on Coding on Intel DevCloud

Complex Multiplication with DPC++
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 DPC++ language and programming model
• Important Classes for DPC++ application
• Device selection and offloading kernel workloads
• DPC++ Buffers, Accessors, Command Group handler, lambda code as kernel
NOTICES & DISCLAIMERS

- 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