C++ SYCL

Learn how to use SYCL to offload computation to accelerators

rakshith.krishnappa@intel.com

intel.

C++ SYCL

- Objective
  - Learn how to use SYCL to offload computation to accelerators

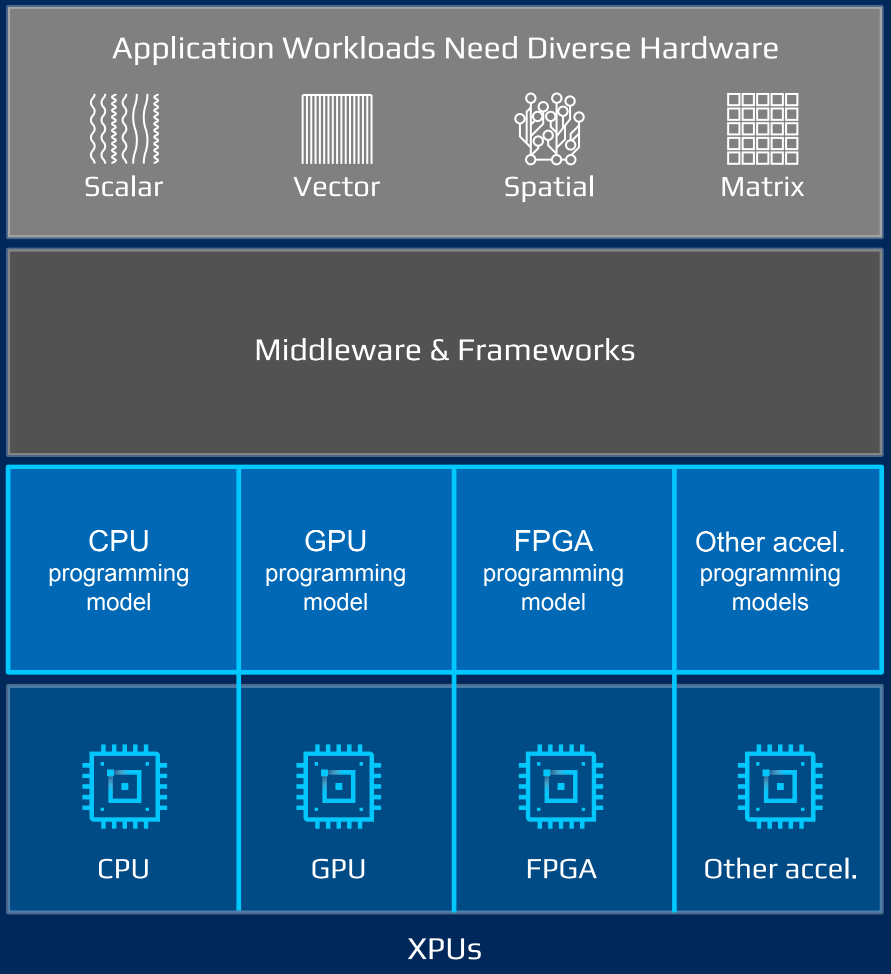
- Agenda
  - Why use SYCL for offloading computation
  - C++ SYCL example
  - Compiling C++ SYCL
  - Hands-on workshop on Intel Developer Cloud

Why Offload Computation?

Why Offload Computation to accelerators?

Large computational problems run substantially faster on specialized hardware accelerators like a GPU than on a CPU.

Accelerators like GPU can run many smaller computations at once by making use of parallelism in the hardware.



Why SYCL?

What is SYCL?

SYCL (pronounced ‘sickle’) is a royalty-free, cross-platform abstraction layer that:

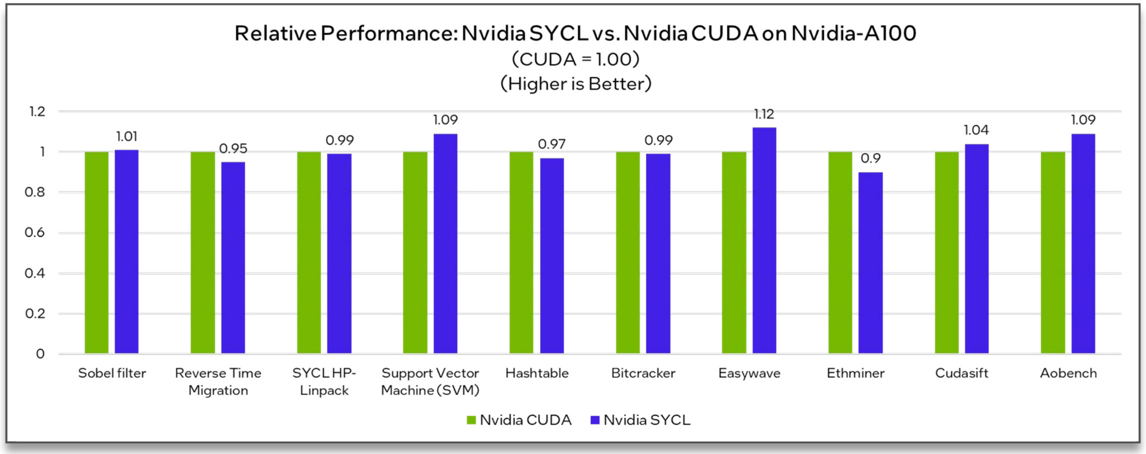
- Enables code for heterogeneous and offload processors to be written using modern ISO C++ (at least C++ 17).
- Provides APIs and abstractions to find devices (e.g. CPUs, GPUs, FPGAs) on which code can be executed, and to manage data resources and code execution on those devices.

https://www.khronos.org/sycl/

Accelerating Choice with SYCL

Khronos Group Standard

- Open, standards-based
- Multiarchitecture performance
- Freedom from vendor lock-in
- Comparable performance to native CUDA on Nvidia GPUs
- Extension of widely used C++ language
- Speed code migration via open source SYCLomatic or Intel® DPC++ Compatibility Tool



Testing Date: Performance results are **based on testing by Intel as of August 15, 2022** and may not reflect all publicly available updates.

Configuration Details and Workload Setup: Intel® Xeon® Platinum 8360Y CPU @ 2.4GHz, 2 socket, Hyper Thread On, Turbo On, 256GB Hynix DDR4-3200, ucode 0x000363. GPU: Nvidia A100 PCIe 80GB GPU memory. Software: SYCL open source/CLANG 15.0.0, CUDA SDK 11.7 with NVIDIA-NVCC 11.7.64, cuMath 11.7, cuDNN 11.7, Ubuntu 22.04.1. SYCL open source/CLANG compiler switches: -fscycl-targets=nvptx64-nvidia-cuda, NVIDIA NVCC compiler switches: -O3 –gencode arch=compute_80, code=sm_80. Represented workloads with Intel optimizations.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary.

Architectures	Intel | Nvidia | AMD CPU/GPU | RISC-V | ARM Mali | PowerVR | Xilinx

Productive and Performant SYCL Compiler

Intel® oneAPI DPC++/C++ Compiler

Uncompromised parallel programming productivity and performance across CPUs and accelerators

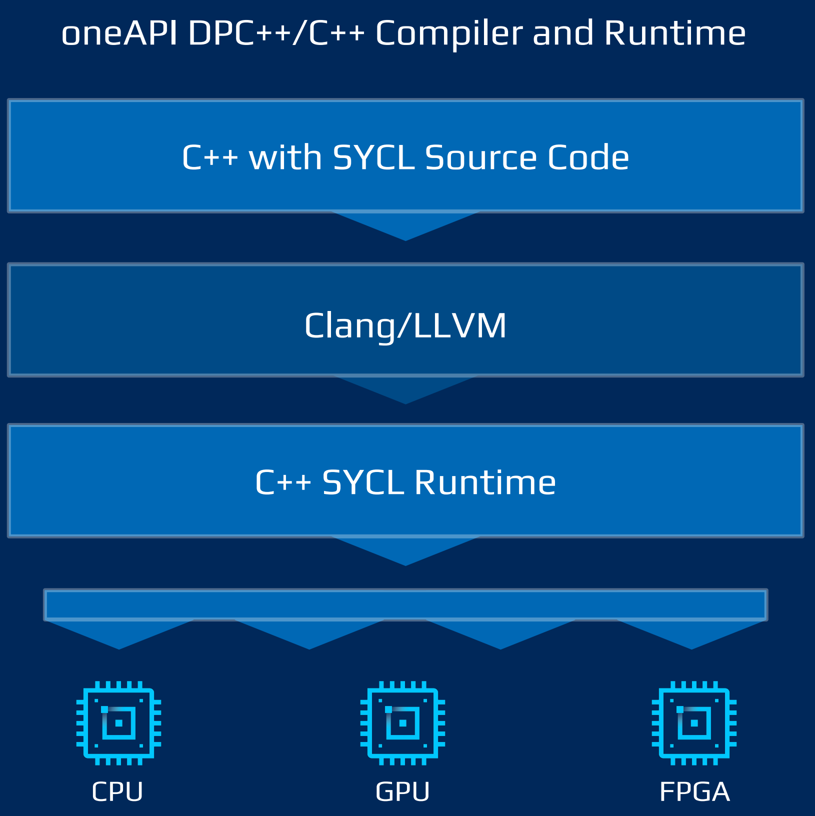
- Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single architecture proprietary language

Khronos SYCL Standard

- Delivers C++ productivity benefits, using common and familiar C and C++ constructs
- Created by Khronos Group to support data parallelism and heterogeneous programming

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

Learn More & Download



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

C++ Example

C++

```cpp
#include <iostream>

int main(){


    // initialize some data array
    const int N = 16;
    float data[N];
    for(int i=0;i<N;i++) data[i] = i;

    // computation on CPU

    for(int i=0;i<N;i++) data[i] = data[i] * 5;


    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

- Let's look at this simple C++ Code
  - We initialize a data array
  - We do some computation on each element of the array
  - Print the output
- Out next goal is to do the computation on GPU
  - **How do we do this?**

C++ SYCL for Offloading

C++

```cpp
#include <iostream>

int main(){


    // initialize some data array
    const int N = 16;
    float data[N];
    for(int i=0;i<N;i++) data[i] = i;

    // computation on CPU

    for(int i=0;i<N;i++) data[i] = data[i] * 5;


    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ SYCL

```cpp
#include <sycl/sycl.hpp>
#include <iostream>

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.single_task([=](){
      for(int i=0;i<N;i++) data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ SYCL for Offloading

What is SYCL doing here:

1. Select GPU device for offloading
2. Allocate memory so that both host and device can access
3. Submit a kernel task to device for computation and wait for completion

C++ SYCL

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

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.single_task([=](){
      for(int i=0;i<N;i++) data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ SYCL – Device Selection

sycl::queue is used schedule a task to execute on a device

The device can be specified when sycl::queue is constructed

- `sycl::queue q(sycl::gpu_selector_v);`
- `sycl::queue q(sycl::cpu_selector_v);`
- `sycl::queue q(sycl::accelerator_selector_v);`
- `sycl::queue q(sycl::default_selector_v);`
- `sycl::queue q;`

We will learn more about this in SYCL Essentials – Program Structure Module

C++ SYCL

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

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.single_task([=](){
      for(int i=0;i<N;i++) data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ SYCL – Memory Allocation

sycl::malloc_shared is used here to allocate memory that can be accessed by both host and device and data movement happens implicitly

There is also sycl::malloc_device, which allocates memory on device, which allows more controlled explicit data movement, which is recommended for performance.

We will learn more about this in SYCL Essentials – Unified Shared Memory Module

C++ SYCL

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

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.single_task([=](){
      for(int i=0;i<N;i++) data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ SYCL – Submitting Task to Device

q.single_task is the most basic method to submit a task to execute on device.

The kernel execution happens asynchronously, so we have to synchronize with host.

.wait() method is used to synchronize with host by waiting for task completion.

There is also q.parallel_for, which allows submitting a task and enables parallel execution on device.

We will learn more about this in SYCL Essentials – Program Structure Module

C++ SYCL

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

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.single_task([=](){
      for(int i=0;i<N;i++) data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ SYCL – Parallel Execution

C++ SYCL – single_task

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

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.single_task([=](){
      for(int i=0;i<N;i++) data[i] = data[i] * 5;   // "for(int i=0;i<N;i++)" struck through on slide
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ SYCL – parallel_for

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

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.parallel_for(N,[=](auto i){
      data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

This is basics of C++ SYCL

- Device Selection for offloading computation
- Memory Allocation that can be accessed by host and device
- Submit computation task for parallel execution on device.

C++ SYCL

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

int main(){
    // select device for offload
    sycl::queue q(sycl::gpu_selector_v);

    // initialize some data array
    const int N = 16;
    auto data = sycl::malloc_shared<float>(N, q);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on GPU
    q.parallel_for(N,[=](auto i){
      data[i] = data[i] * 5;
    }).wait();

    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

SYCL Library for Offloading

There are many SYCL libraries available that will simplify offloading certain types of computations to devices.

SYCL Library for Offloading

C++

```
#include <iostream>

int main(){
    // initialize some data array
    const int N = 16;
    std::vector<int> data(N);
    for(int i=0;i<N;i++) data[i] = i;

    // computation on CPU

    for(int i=0;i<N;i++) data[i] = data[i] * 5;


    // print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

C++ oneDPL

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <iostream>

int main(){
    //# initialize some data array
    const int N = 16;
    std::vector<int> data(N);
    for(int i=0;i<N;i++) data[i] = i;

    //# computation on GPU using SYCL library (oneDPL)

oneapi::dpl::for_each(oneapi::dpl::execution::dpcpp_de
fault, data.begin(), data.end(), [](int &tmp){ tmp *=
5; });

    //# print output
    for(int i=0;i<N;i++) std::cout << data[i] << "\n";
}
```

SYCL Library for Offloading

- There are many SYCL libraries available that will simplify offloading certain types of computations to devices.
- These libraries can be used to quickly get device offloading to work when starting from C/C++ code or if the computation is simple.
- But using actual SYCL API calls to offload computation allows you better optimize the code for performance by programming to device hardware features.

Compiling C++ code vs C++ SYCL code

- Install oneAPI C++/DPC++ Compiler or Intel oneAPI Base Toolkit
  - Link to Installation Instructions
- Set environment variable for using the Compiler
  - `source /opt/intel/oneapi/setvars.sh`
- Compile C++ for Intel CPUs
  - `icpx test.cpp`
- Compile SYCL for Intel CPUs/GPUs
  - `icpx -fsycl test.cpp`

Compiling SYCL code for Intel, NVIDIA, AMD

- Install oneAPI C++/DPC++ Compiler or Intel oneAPI Base Toolkit
- Install CodePlay Plugin for oneAPI Compiler
- Link to Installation Instructions

- Set environment variable for using the Compiler
  - `source /opt/intel/oneapi/setvars.sh`
- Compile SYCL for Intel CPUs/GPUs
  - `icpx –fsycl test.cpp`
- Compile SYCL for NVIDIA GPUs
  - `icpx -fsycl -fsycl-targets=nvptx64-nvidia-cuda test.cpp`
- Compile SYCL for AMD GPUs
  - `icpx -fsycl -fsycl-targets=amdgcn-amd-amdhsa test.cpp`

C++ SYCL – What's Next?

Memory Management Methods

- Unified Shared Memory
  - We learnt basics with sycl::malloc_shared, which is pointer-based memory.
  - There is better approach with sycl::malloc_device to allocate memory on device and perform more controlled explicit data movement.
- Buffer Memory Model
  - There is an alternative to Unified Shared Memory Model, which provides a memory abstraction for data, allows data representation in 1,2 or 3-dimensions and handles data dependency implicitly between kernel tasks.
  - This memory model can be adopted if it works better for your application development.

C++ SYCL – What's Next?

Advanced Features of SYCL

- ND-Range Kernels which allow grouping executions and map execution to hardware.
- Optimizing Kernel Code:
  - Shared Local Memory usage
  - Atomics to prevent race conditions
  - Mapping to SIMD hardware with Sub-Groups
  - Group Algorithms and libraries to use in kernel code

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.
Your costs and results may vary.

Reference Material

Learn SYCL Programming

- SYCL 2020 Specification
- SYCL Data Parallel C++ Book
- SYCL Academy from CodePlay
- Guided learning path with code samples (Jupyter Notebooks):
  - SYCL Essentials
  - SYCL Performance Portability
  - CUDA to SYCL Migration
  - Intel GPU Optimization with SYCL
- C++ SYCL code samples

Optimizing SYCL code for Intel GPUs

- Refer to [Intel GPU Optimization Guide]
  - Detailed guide explaining how to optimize SYCL code:
    - Thread Mapping and GPU Occupancy Calculation.
    - Memory allocation and transfer optimization when using Buffers or Unified Shared memory.
    - Kernel code optimization – Local memory, Sub-Groups, Atomics, Reduction and more.
    - Using libraries for offload
    - Debugging and Profiling

CUDA to SYCL Migration Portal

- One Stop Portal for CUDA to SYCL Migration
  - Industry Examples of CUDA Migration to SYCL
  - Learn How to Migrate Your Code
  - Guided flow of migrating from CUDA to SYCL
  - Get all the tools and resources necessary for migrating from CUDA to SYCL