Multiarchitecture Programming for Accelerated Compute
Freedom of Choice for Hardware

oneAPI Industry Initiative & Intel® oneAPI Tools

藺杰 (Auber Lin)

SATG/DSE/DTCE/APCAP/PRC Customer Engineering Team

Nov. 2023

Agenda

- oneAPI Goal
- Intel oneAPI product - Toolkits

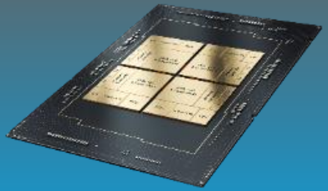
oneAPI Goal

Modern Applications Demand Increased Processing

Diverse accelerators needed to meet today's performance requirements:

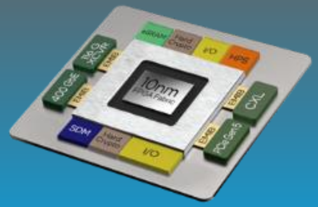
48% of developers target heterogeneous systems
that use more than one kind of processor or core[1]

CPU



GPU

FPGA



Other Accelerators

Developer Challenges: Multiple Architectures, Vendors, and Programming Models

oneAPI

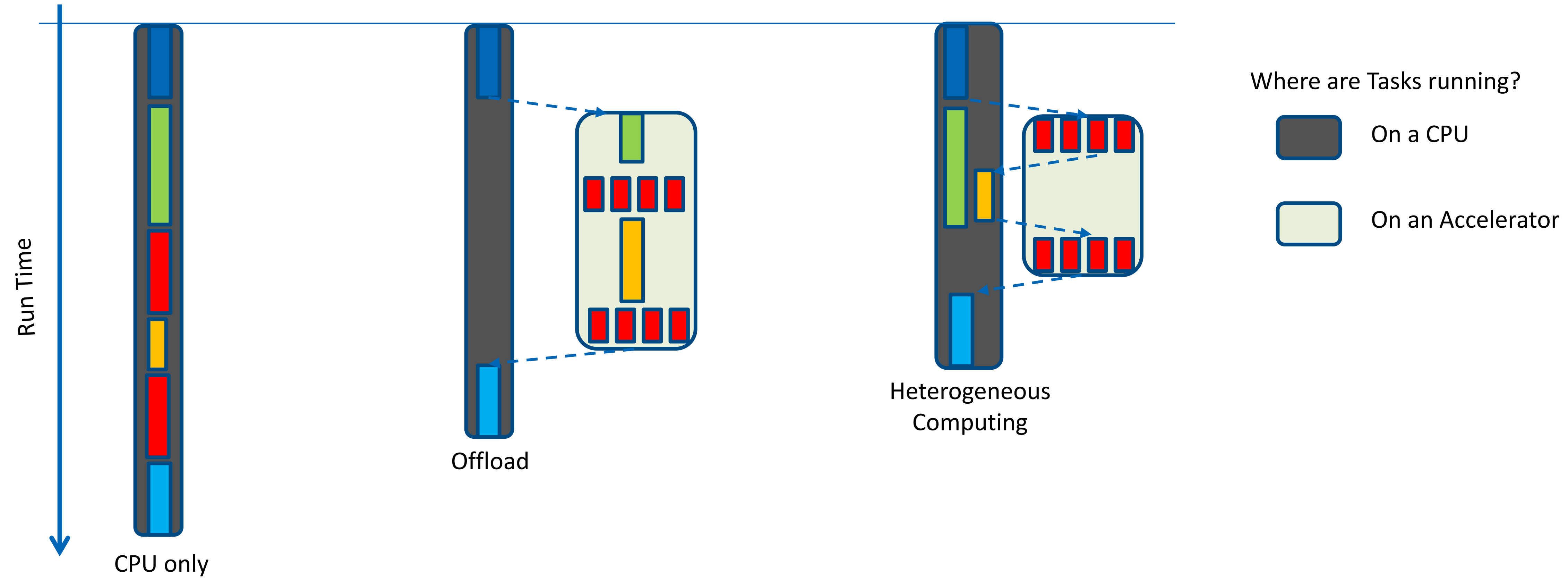
Open, Standards-based, Multiarchitecture Programming

1. Evans Data Global Development Survey Report 22.1, June 2022

Offload v.s. Heterogenous Computing

- Offload

The CPU moves work to an accelerator and waits for the answer



异构计算软件生态现状

Domain		Example Workloads	Software Ecosystem
AI	DL	• Training • Inference	High level frameworks targeting *proprietary interfaces*
	ML	• SVM • k-means	No established standards
	Big Data	• Data mining • Analytics	Distributed computing on Xeon
HPC		• Simulation • Modeling	*CUDA based GPU support*
Visual		• Video • Transcode	Well-established industry standards

- Domain Specific Languages
- Modular, pluggable
- One size does not fit all
- Sparse and unstructured
- Distributed

oneAPI Industry Initiative

Break the Chains of Proprietary Lock-in

Freedom to Make Your Best Choice

- C++ programming model for multiple architectures and vendors
- Cross-architecture code reuse for freedom from vendor lock-in

Realize all the Hardware Value

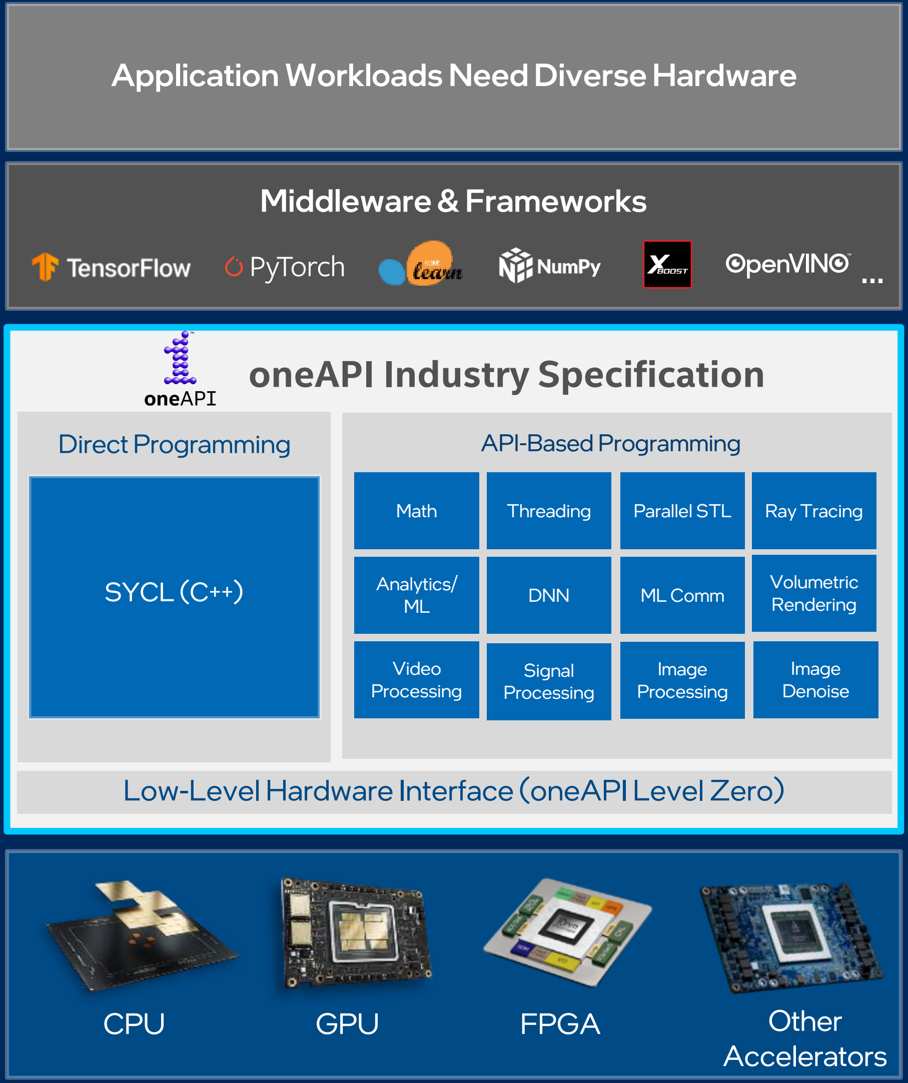
- Performance across CPU, GPUs, FPGAs, and other accelerators
- Expose and exploit cutting-edge features of the latest hardware

Develop & Deploy Software with Peace of Mind

- Open industry standards provide a safe, clear path to the future
- Interoperable with familiar languages and programming models including Fortran, Python, OpenMP, and MPI
- Powerful libraries for acceleration of domain-specific functions

oneAPI

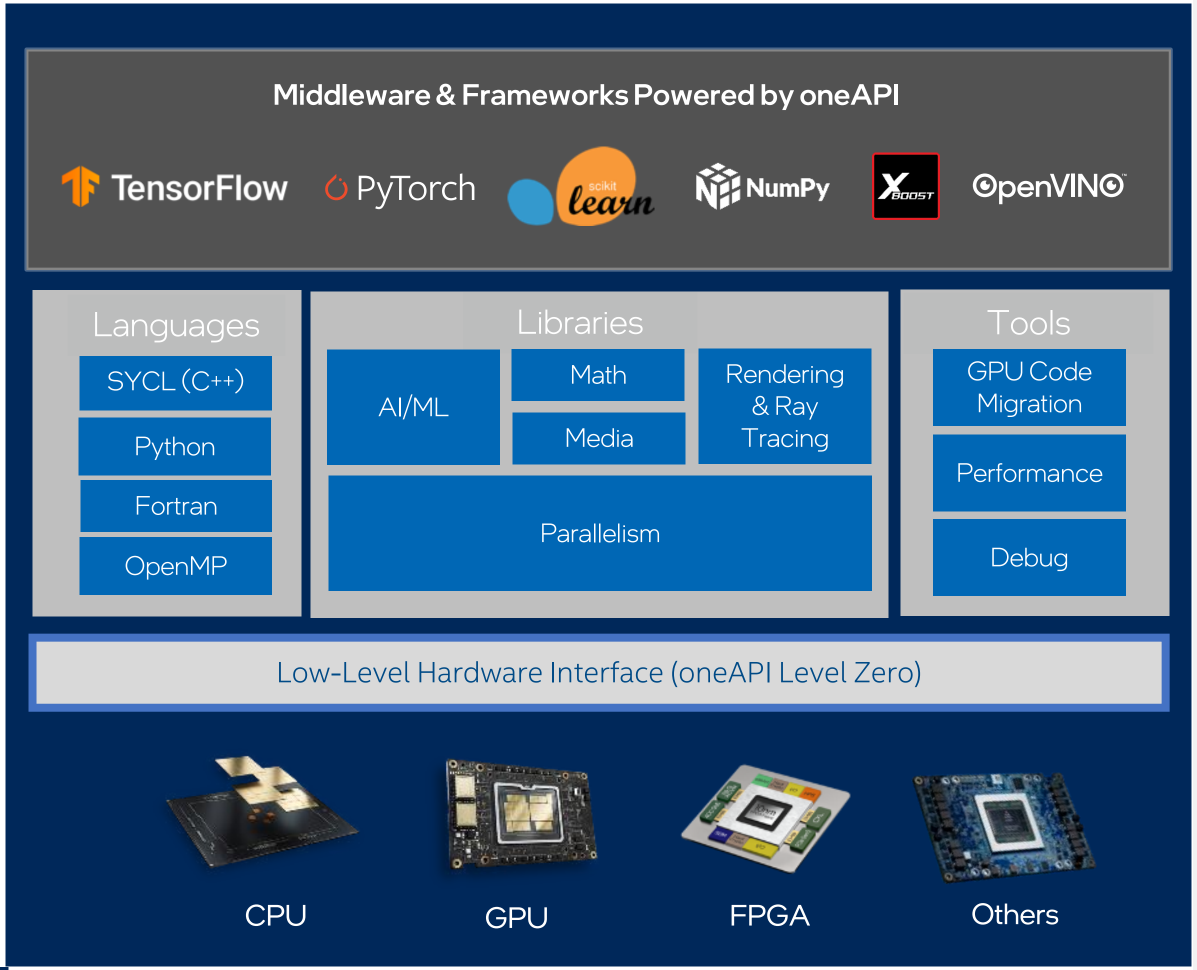
The productive, smart path to freedom for accelerated computing from the economic and technical burdens of proprietary programming models



Intel® Developer Tools Supporting oneAPI

A complete set of proven tools expanded from CPU to accelerators

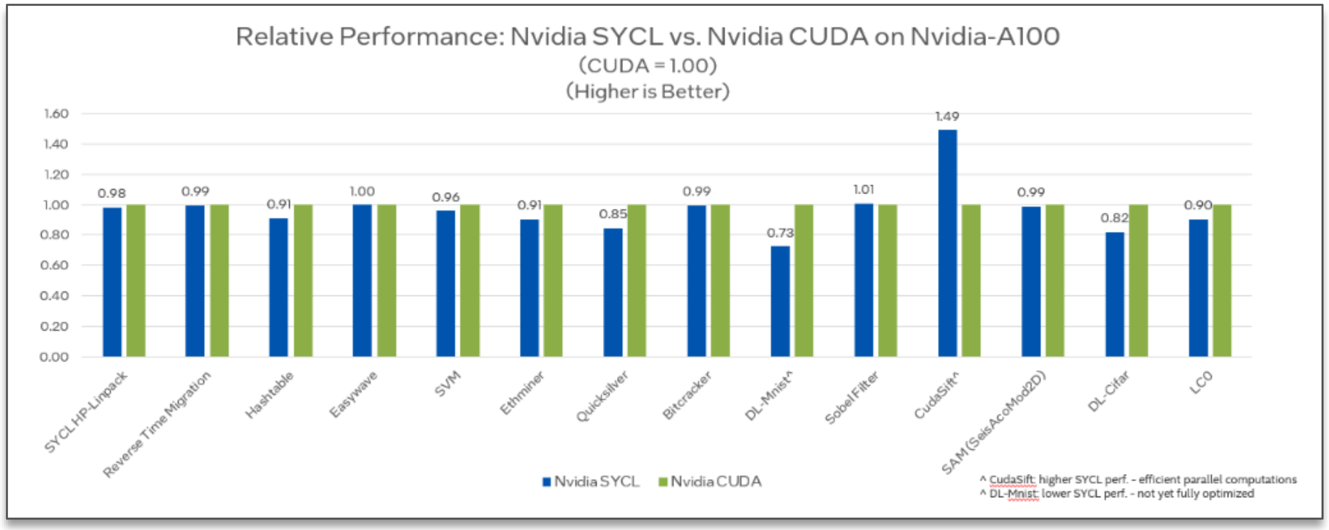
- Advanced compilers, libraries, and analysis, debug, and porting tools
- Full support for C, C++ with SYCL, Python, Fortran, MPI, OpenMP
- Intel® Advisor determines device target mix before you write your code
- Intel's compilers optimize code to take full advantage of multiarchitecture workload distribution.
- Intel® VTune™ Profiler analyzes hotspots to optimize code performance
- Intel AI tools support acceleration of major deep learning and machine learning frameworks



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

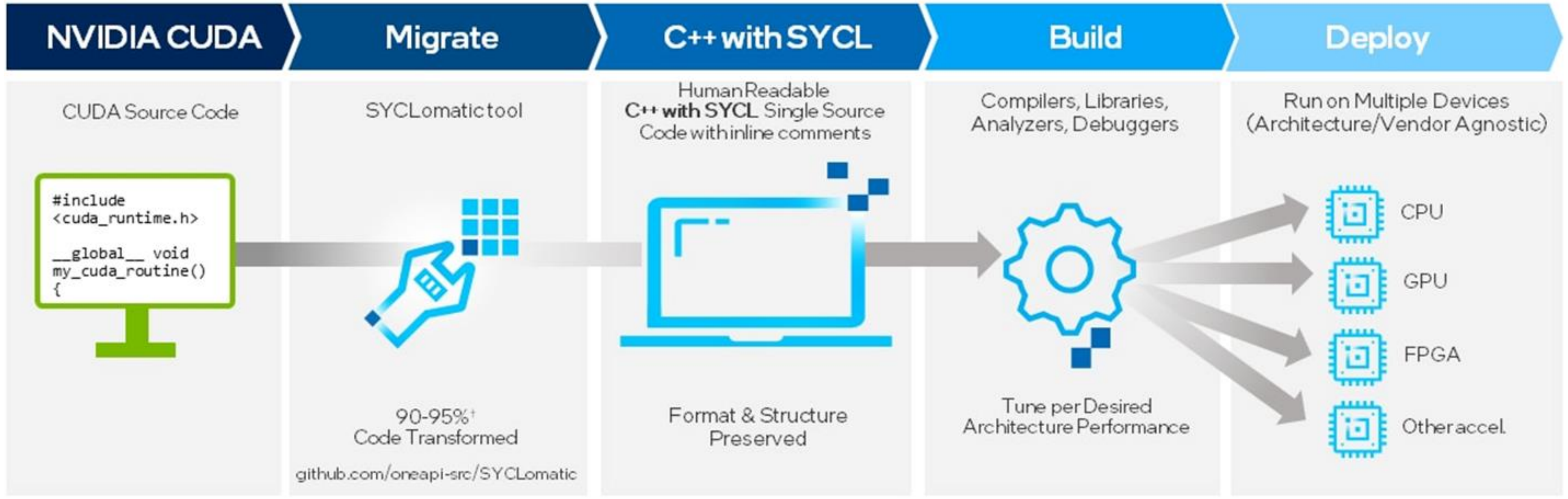


Architectures	Intel | Nvidia | AMD CPU/GPU | RISC-V | ARM Mali | PowerVR | Xilinx

Testing Date: Performance results are **based on testing by Intel as of April 15, 2023** 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 0xd000363. GPU: Nvidia A100 PCIe 80GB GPU memory. Software: SYCL open source/CLANG 17.0.0, CUDA SDK 12.0 with NVIDIA-NVCC 12.0.76, cuMath 12.0, cuDNN 12.0, 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.
SYCL is a trademark of the Khronos Group Inc.

SYCLomatic: CUDA* to SYCL* Migration Made Easy

Choose where to run your software, don't let the software choose for you.


NVIDIA CUDA
Migrate
C++ with SYCL
Build
Deploy
CUDA Source Code
SYCLomatic tool
Human Readable C++ with SYCL Single Source Code with inline comments
Compilers, Libraries, Analyzers, Debuggers
Run on Multiple Devices (Architecture/Vendor Agnostic)
#include <cuda_runtime.h>
__global__ void my_cuda_routine() {
CPU
GPU
FPGA
Other accel.
90-95%† Code Transformed
github.com/oneapi-src/SYCLomatic
Format & Structure Preserved
Tune per Desired Architecture Performance

Open source SYCLomatic tool assists developers migrating code written in CUDA to C++ with SYCL, generating **human readable** code wherever possible

~90-95% of code typically migrates automatically[1]

Inline comments are provided to help developers finish porting the application

Intel® DPC++ Compatibility Tool is Intel's implementation, available in the Intel® oneAPI Base Toolkit

github.com/oneapi-src/SYCLomatic

[1]Intel estimates as of March 2023. Based on measurements on a set of 85 HPC benchmarks and samples, with examples like Rodinia, SHOC, PENNANT. Results may vary.
*Other names and brands may be claimed as the property of others. SYCL is a trademark of the Khronos Group Inc.

Codeplay oneAPI Plug-ins for Nvidia* & AMD*

Support for Nvidia & AMD GPUs to Intel® oneAPI Base Toolkit

oneAPI for NVIDIA & AMD GPUs

- Free download of binary plugins to Intel® oneAPI DPC++/C++ Compiler:
- Nvidia GPU
- AMD beta GPU
- No need to build from source!
- Plug-ins updated quarterly in-sync with SYCL 2020 conformance & performance

Priority Support

- Available through Intel, Codeplay & our channel
- Requires Intel Priority Support for Intel® oneAPI DPC++/C++ Compiler
- Intel takes first call, Codeplay delivers backend support
- Codeplay provides access to older plug-in versions

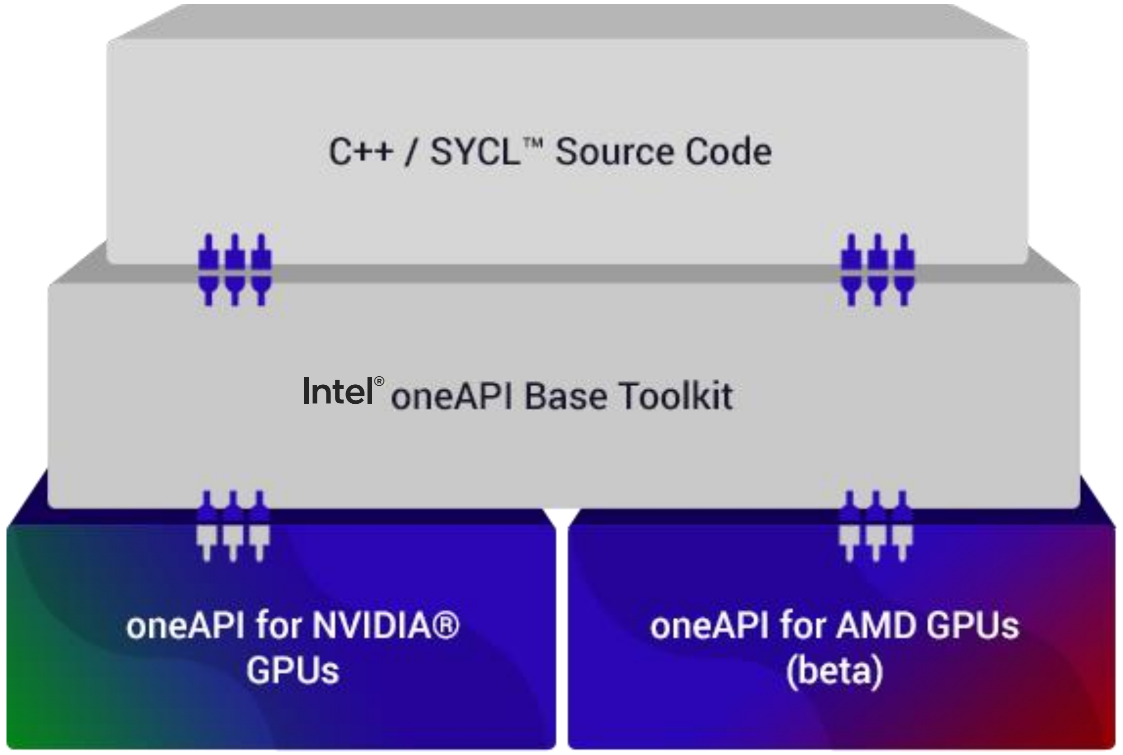


Image courtesy of Codeplay Software Ltd.

Nvidia GPU plug-in

AMD GPU plug-in

Codeplay blog

Codeplay press release

*Other names and brands may be claimed as the property of others.

oneAPI Libraries

Domain	Name	Description	Open Spec	Open Source
Parallel Programming	oneDPL	Data Parallel C++ Library including Parallel STL	Yes	Yes
	oneTBB	Threading Building Blocks	Yes	Yes
AI & ML	oneDNN	Deep Neural Networks	Yes	Yes
	oneCCL	Collective Communications	Yes	Yes
	oneDAL	Data Analytics and Machine learning	Yes	Yes
Math	oneMKL	Math Kernels: linear algebra, FFT, random numger generation	Yes	Partial
Video	oneVPL	Video Processing: encode, decode, transcode	Yes	Yes
Ray Tracing	Embree, VKL, OID, OSPRay	Geometric & Volumetric Ray Tracing, Image Denoise, Scalable Rendering	Yes	Yes

oneAPI Industry Momentum

End Users

Accrad Accelerated Radiology
@rchanan
FOUNDRY. MODO
BioDataAnalysis GmbH Next Generation Bio-Medical Image Analysis
GeoEast
BENTLEY
CINESITE
LAIKA
GE
kt
PHILIPS
AUTODESK® ARNOLD
中国石油集团东方地球物理勘探有限责任公司 BGP INC.,CHINA NATIONAL PETROLEUM CORPORATION
iOmniscient Autonomous AI
EURECOM Sophia Antipolis
Brightskies
大势智慧 DASPATIAL
WeBank
SAMSUNG MEDISON
TANGENTANIMATION
allegro.ai
ILLUMINATION MACGUFF
Verizon
MediaKind

National Labs

Argonne NATIONAL LABORATORY
UT-BATTELLE Oak Ridge National Laboratory
CERN openlab
CINECA
Peraton Labs
Laboratório Nacional de Computação Científica
lrz Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities
SANKHYA SUTRA LABS

ISVs & OSVs

codeplay®
CANONICAL
CERTI FACE
ANACONDA.
Ansys
MAX PLANCK COMPUTING & DATA FACILITY
MPCDF
AI SINGAPORE
sas
Red Hat OpenShift Data Science
KATANA GRAPH
CGG
AIBLE
CHAOSGROUP
MAXON
Ospirent
FOUNDRY.
mercenaries engineering Guerilla
Hisense Medical 海信医疗
SENAI CIMATEC FIEB
SAP
E4 COMPUTER ENGINEERING
YUAN
GIGASPACES innovate with confidence
MEGH COMPUTING
KFBIO
Tech Mahindra
AsiaInfo 亚信科技
SUSE
vmware
UNITED IMAGING

OEMs & SIs

BittWare a molex company
Hewlett Packard Enterprise
DELL Technologies
Atos
Lenovo
HCL
rENIAC
MEGWARE SUPERCOMPUTING · TECHNOLOGY

Universities & Research Institutes

LOBACHEVSKY UNIVERSITY
TECHNION Israel Institute of Technology
UNIVERSITY OF CAMBRIDGE
ICT 中国科学院计算技术研究所 INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES
Ben-Gurion University of the Negev
Berkeley
UC DAVIS UNIVERSITY OF CALIFORNIA
THE UNIVERSITY OF TENNESSEE KNOXVILLE
UNIVERSIDAD COMPLUTENSE MADRID
OLD DOMINION UNIVERSITY
ctc
北京大学软件与微电子学院 School of Software & Microelectronics
THE UNIVERSITY OF UTAH
ILLINOIS
UNIVERSITY OF OREGON
TÉCNICO LISBOA
inesc id lisboa
ZIB
FACULTY OF MATHEMATICS AND PHYSICS Charles University
PURDUE UNIVERSITY Elmore Family School of Electrical and Computer Engineering
University College London
Durham University
University of Stuttgart Germany
Indian Institutes of Technology Delhi / Kharagpur / Roorkee
SDSC SAN DIEGO SUPERCOMPUTER CENTER
सीडैक CDAC
UK RI Science and Technology Facilities Council Scientific Computing
TACC
Indian Institute of Science Bangalore
IISER PUNE Indian Institute of Science Education & Research Pune
UNIVERSIDAD DE MÁLAGA
URZ HEIDELBERG UNIVERSITY COMPUTING CENTRE
NIU Northern Illinois University
University of BRISTOL
Stockholm University
KTH

CSPs & Frameworks

Microsoft Azure
Alibaba Cloud
Google Cloud
TensorFlow
Taboola
Tencent 腾讯
DataRobot
Bai百度
飞桨 PaddlePaddle
NAVER CLOVA
PyTorch

These organizations support the oneAPI initiative for a single, unified programming model for cross-architecture development.
It does not indicate any agreement to purchase or use of Intel's products. *Other names and brands may be claimed as the property of others.

oneAPI Commercial & Community Support Available

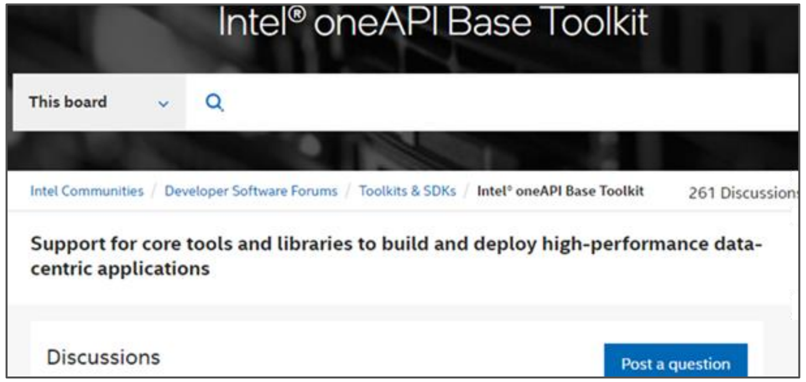
Priority Support for Intel® oneAPI Toolkits

Every paid version of Intel® oneAPI Developer Toolkits includes Priority Support for that toolkit (Intel oneAPI Base, HPC, IOT, & Rendering Toolkits)

- Direct, private interaction with Intel software support engineers
- Accelerated response time
- Access to—and support for—previous Intel products such as Fortran compiler versions, previous toolkit versions, and more
- Intel Technical Consulting Engineers for on-site or online training and consultation at a reduced cost

Free Community Support

- Support via the Intel public Community Forum
- Access to only the latest versions of oneAPI Toolkits
- Access to online tutorials and self-help forums


Intel® oneAPI Base Toolkit
This board
Intel Communities / Developer Software Forums / Toolkits & SDKs / Intel® oneAPI Base Toolkit
261 Discussions
Discussions
Post a question

Support for core tools and libraries to build and deploy high-performance data-centric applications

Maximize Your Performance

With Intel Developer Tools & Hardware Platforms

HPC & Data Center

intel. oneAPI BASE TOOLKIT

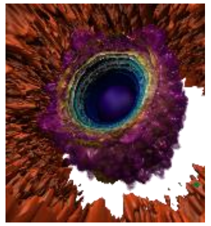
intel. oneAPI HPC TOOLKIT

intel. XEON · intel. DATA CENTER GPU FLEX SERIES · intel. DATA CENTER GPU MAX SERIES · intel. AGILEX

AI & Visualization

intel. AI ANALYTICS TOOLKIT

intel. oneAPI RENDERING TOOLKIT



intel. XEON · intel. DATA CENTER GPU FLEX SERIES · intel. DATA CENTER GPU MAX SERIES · intel. AGILEX

Embedded Systems & IoT

intel. oneAPI IoT TOOLKIT

intel. XEON · intel. ATOM · intel. CORE i3 · intel. CYCLONE

Performance

Optimize compute performance on the latest Intel CPUs, GPUs and FPGAs

Maximize built-in accelerators

Accelerate across AI frameworks

Productivity

Familiar languages and standards

Easily integrate w/ legacy code

Easily migrate CUDA to SYCL

Minimize code re-writes

Freedom

Open alternative to proprietary lock-in

Enables easy architecture retargeting

Code longevity for future hardware

Details about Intel® oneAPI Toolkits

Intel® oneAPI Base Toolkit

Intel® oneAPI HPC Toolkit

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
- SYCLomatic / DPC++ Compatibility tool helps migrate CUDA code to C++ with SYCL
- 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 & Download

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 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. 1 oneAPI BASE TOOLKIT

intel.

Productive and Performant SYCL Compiler

Intel® oneAPI DPC++/C++ Compiler

Uncompromised parallel programming productivity and performance across CPUs and accelerators

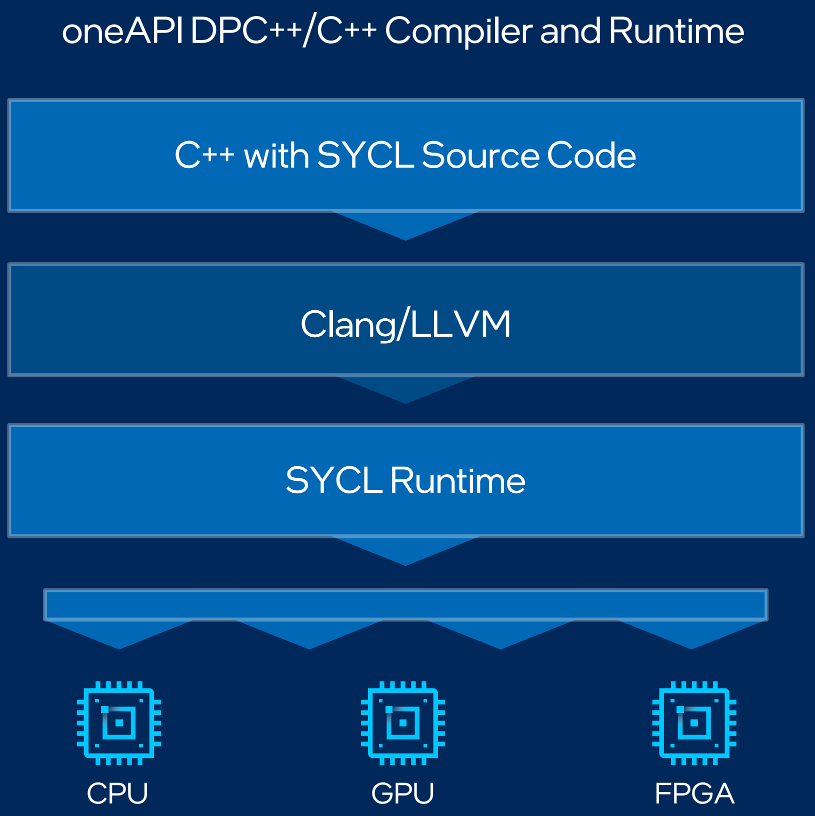
- Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single architecture proprietary language

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


oneAPI DPC++/C++ Compiler and Runtime
C++ with SYCL Source Code
Clang/LLVM
SYCL Runtime
CPU
GPU
FPGA

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

Analysis & Debug Tools

Get More from Diverse Hardware

Design

Intel® Advisor

- Efficiently offload code to GPUs
- Optimize your CPU/GPU code for memory and compute
- Enable more vector parallelism and improve efficiency
- Add effective threading to unthreaded applications

Debug

Intel® Distribution for GDB

- Multiple accelerator support with CPU, GPU and FPGA
- Enables deep, system-wide debug of SYCL, C, C++, OpenMP and Fortran cross-architecture applications
- IDE Integration into Microsoft Visual Studio, VS Code and Eclipse

Tune

Intel® VTune™ Profiler

- Tune for GPU, CPU, and FPGA
- Optimize offload performance
- Supports SYCL, C, C++, Fortran, Python, Go, Java or a mix of languages

Intel® oneAPI Tools for HPC

Intel® oneAPI HPC Toolkit

Deliver Fast Applications that Scale

What is it?

A toolkit that adds to the Intel® oneAPI Base Toolkit for building high-performance, scalable parallel code on C++, Fortran, SYCL, OpenMP & MPI from enterprise to cloud, and HPC to AI applications.

Who needs this product?

- OEMs/ISVs
- C++, Fortran, OpenMP, MPI Developers

Why is this important?

- Accelerate performance on Intel® Xeon® & Core™ processors & Intel accelerators
- Deliver fast, scalable, reliable parallel code with less effort built on industry standards

[Learn More & Download]

Intel® oneAPI Base & HPC Toolkits

Direct Programming	API-Based Programming	Analysis & debug Tools
Intel® C++ Compiler Classic	Intel® MPI Library	Intel® Inspector
Intel® Fortran Compiler Classic	Intel® oneAPI DPC++ Library oneDPL	Intel® Trace Analyzer & Collector
Intel® Fortran Compiler	Intel® oneAPI Math Kernel Library - oneMKL	Intel® VTune™ Profiler
Intel® oneAPI DPC++/C++ Compiler?	Intel® oneAPI Data Analytics Library - oneDAL	Intel® Advisor
Intel® DPC++ Compatibility Tool	Intel® oneAPI Threading Building Blocks - oneTBB	Intel® Distribution for GDB
Intel® Distribution for Python	Intel® oneAPI Collective Communications Library oneCCL	
Intel® FPGA Add-on for oneAPI Base Toolkit	Intel® oneAPI Deep Neural Network Library - oneDNN	
	Intel® Integrated Performance Primitives – Intel® IPP	

■ Intel® oneAPI **HPC** Toolkit +

■ Intel® oneAPI **Base** Toolkit

oneAPI Resources

software.intel.com/oneapi

Get Started

- software.intel.com/oneapi
- Documentation + dev guides
- Code Samples
- Intel® Developer Cloud

Industry Initiative

- oneAPI.io
- oneAPI open Industry Specification
- Open-source Implementations

Learn

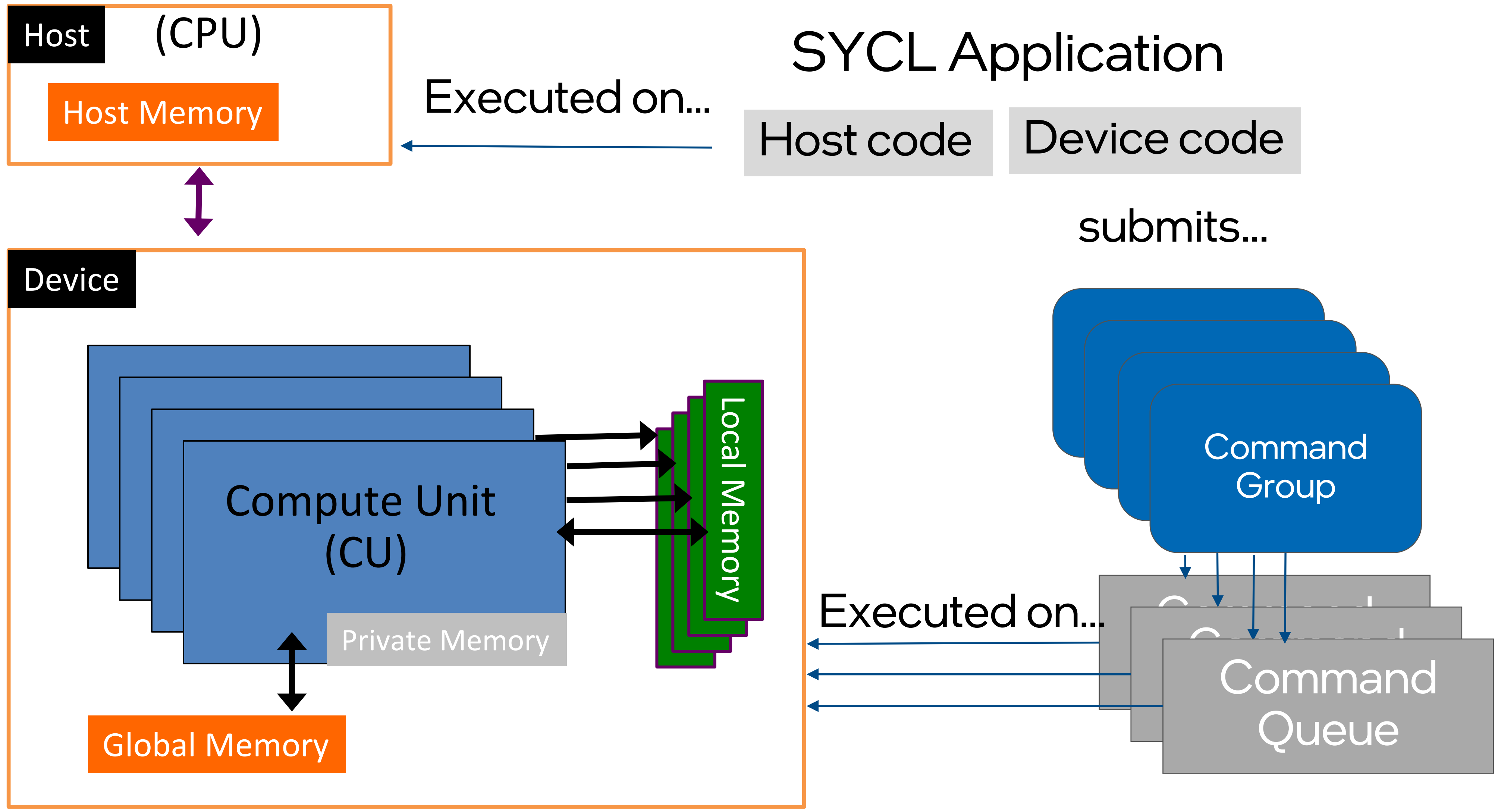
- Training: Webinars & courses
- Intel® DevMesh Innovator Projects
- Summits & Workshops: Live & on-demand virtual workshops, community-led sessions
- Training by certified oneAPI experts worldwide for HPC & AI

Ecosystem

- Community Forums
- Academic Programs: oneAPI Centers of Excellence: research, enabling code, curriculum, teaching

SYCL Basics

SYCL View of Heterogenous Computing Platform



Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024), B(1024), C(1024);
// some data initialization
    {
      buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h) {
          auto A = bufA.get_access(h, read_only);
          auto B = bufB.get_access(h, read_only);
          auto C = bufC.get_access(h, write_only);
          h.parallel_for(1024, [=](auto i){
              C[i] = A[i] + B[i];
          });
      });
    }
for (int i = 0; i < 1024; i++)
       std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

Host code

Accelerator device code

Host code

Anatomy of a SYCL Application

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

int main() {
std::vector<float> A(1024), B(1024), C(1024);
// some data initialization
    {
      buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h) {
          auto A = bufA.get_access(h, read_only);
          auto B = bufB.get_access(h, read_only);
          auto C = bufC.get_access(h, write_only);
          h.parallel_for(1024, [=](auto i){
              C[i] = A[i] + B[i];
          });
      });
    }
for (int i = 0; i < 1024; i++)
       std::cout << "C[" << i << "] = " << C[i] << std::endl;

}
```

Application scope

Command group scope

Device scope

Application scope

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
     {
       buffer bufA {A}, bufB {B}, bufC {C};
       queue q;
       q.submit([&](handler &h) {
           auto A = bufA.get_access(h, read_only);
           auto B = bufB.get_access(h, read_only);
           auto C = bufC.get_access(h, write_only);
           h.parallel_for(1024, [=](auto i){
               C[i] = A[i] + B[i];
           });
       });
     }
 for (int i = 0; i < 1024; i++)

         std::cout << "C[" << i << "] = " << C[i] << std::endl;

 }
```

Buffers creation via host vectors/pointers

Buffers encapsulate data in a SYCL application

- Across both devices and host!

intel.

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
     {
       buffer bufA {A}, bufB {B}, bufC {C};
       queue q;
       q.submit([&](handler &h) {
           auto A = bufA.get_access(h, read_only);
           auto B = bufB.get_access(h, read_only);
           auto C = bufC.get_access(h, write_only);
           h.parallel_for(1024, [=](auto i){
               C[i] = A[i] + B[i];
           });
       });
     }
 for (int i = 0; i < 1024; i++)

          std::cout << "C[" << i << "] = " << C[i] << std::endl;

 }
```

- A queue submits command groups to be executed by the SYCL runtime
- Queue is a mechanism where work is submitted to a device.

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
     {
       buffer bufA {A}, bufB {B}, bufC {C};
       queue q;
       q.submit([&](handler &h) {
           auto A = bufA.get_access(h, read_only);
           auto B = bufB.get_access(h, read_only);
           auto C = bufC.get_access(h, write_only);
           h.parallel_for(1024, [=](auto i){
               C[i] = A[i] + B[i];
           });
       });
     }
 for (int i = 0; i < 1024; i++)

      std::cout << "C[" << i << "] = " << C[i] << std::endl;

 }
```

- Accessors creation
- Mechanism to access buffer data
- Create data dependencies in the SYCL graph that order kernel executions

SYCL Basics

```
std::vector<float> A(1024), B(1024), C(1024);
    {
      buffer bufA {A}, bufB {B}, bufC {C};
      queue q;
      q.submit([&](handler &h) {
          auto A = bufA.get_access(h, read_only);
          auto B = bufB.get_access(h, read_only);
          auto C = bufC.get_access(h, write_only);
          h.parallel_for(1024, [=](auto i){
              C[i] = A[i] + B[i]; }
          );
      });
    }
 for (int i = 0; i < 1024; i++)

       std::cout << "C[" << i << "] = " << C[i] << std::endl;

 }
```

range<1>{1024}

id<1>

- Vector addition kernel enqueues a parallel_for task.
- Pass a function object/lambda to be executed by each work-item

Host-side Memory Model – Buffer & Accessor

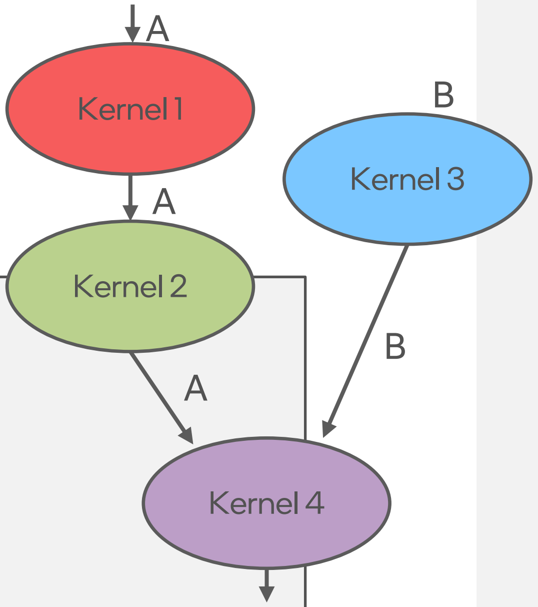
Buffers: Encapsulate data in a SYCL application

- Across both devices and host!

Accessors: Mechanism to access buffer data

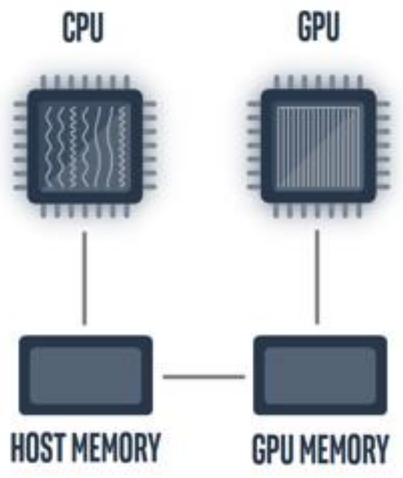
- Create data dependencies in the SYCL graph that order kernel executions

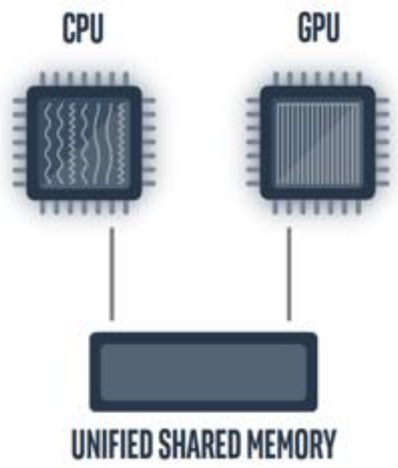
```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    accessor out(A, h, write_only);

    h.parallel_for(R, [=](auto idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) {
        accessor out(A, h, write_only);
    h.parallel_for(R, [=](auto idx) {
      out[idx] = idx[0]; }); });
…
```



intel.

Host-side Memory Model – Unified Shared Memory





Host code

Accelerator device code

Host code

```
using namespace sycl;
int main() {
  queue q;
  float *a = malloc_shared<float>(N, q);
  float *b = malloc_shared<float>(N, q);
  float *c = malloc_shared<float>(N, q);
  q.parallel_for(1024, [=](auto i) {
      c[i] = a[i] + b[i];
  }).wait();
  for (int i=0; i<1024; i++) std::cout << c[i] << "\n";
  free(a, q); free(b, q); free(c, q);
  return 0;
}
```

Notices & Disclaimers

Texas Advanced Computing Center (TACC) Frontera references
Article: *HPCWire: Visualization & Filesystem Use Cases Show Value of Large Memory Fat Notes on Frontera*.
www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf
software.intel.com/content/www/us/en/develop/articles/introduction-to-programming-with-persistent-memory-from-intel.html
wreda.github.io/papers/assise-osdi20.pdf

KFBIO
KFBIO m. tuberculosis screening detectron2 model throughput performance on 2nd Intel® Xeon® Gold 6252 processor: NEW: Test 1 (single instance with PyTorch 1.6: Tested by Intel as of 5/22/2020. 2-socket 2nd Gen Intel® Xeon® Gold 6252 Processor, 24 cores, HT On, Turbo ON, Total Memory 192 GB (12 slots/16 GB/2666 MHz), BIOS: SSE5C620.86B.02.01.0008.031920191559 (ucode: 0x500002c), Ubuntu 18.04.4 LTS, kernel 5.3.0-51-generic, mitigated Test 2 (24 instances with PyTorch 1.6: Tested by Intel as of 5/22/2020. 2-socket 2nd Gen Intel Xeon Gold 6252 Processor, 24 cores, HT On, Turbo ON, Total Memory 192 GB (12 slots/16 GB/2666 MHz), BIOS: SSE5C620.86B.02.01.0008.031920191559 (ucode: 0x500002c), Ubuntu 18.04.4 LTS, kernel 5.3.0-51-generic, mitigated BASELINE: (single instance with PyTorch 1.4): Tested by Intel as of 5/22/2020. 2-socket 2nd Gen Intel Xeon Gold 6252 Processor, 24 cores, HT On, Turbo ON, Total Memory 192 GB (12 slots/16 GB/2666 MHz), BIOS: SSE5C620.86B.02.01.0008.031920191559 (ucode: 0x500002c), Ubuntu 18.04.4 LTS, kernel 5.3.0-51-generic, mitigated.

Tangent Studios
Configurations for Render Times with Intel® Embree, testing conducted by Tangent Animation Labs. Render farm: 8x Intel® Core™ processors +hyperthread*2 + 128gig. In-office workstations: Intel® Xeon® processors HP blade c7000 chassis, with HP460 gen8 blades - 2x Intel Xeon E5-2650 V2, Eight Core 2.6GHz-128GB. Software: Blender 2.78 with custom build using Intel® Embree. For more information on Tangent's work with Embree, watch this video:
www.youtube.com/watch?time_continue=251&v=_2Ia4h8q3xs&feature=emb_logo
Recreation of the performance numbers can be recreated using Agent327, Blender and Embree.

Chaos Group - Up to 90% Memory Reduction for Displacement
Testing conducted by Chaos Group with Intel® Embree 2020. Software Corona Renderer 5 with Intel Embree. Up to 90% memory reduction calculated using Corona Renderer 5 with regular displacement grids per triangle of 154 bytes versus Corona Renderer 5 with Intel Embree, which has a displacement capability grid of 12 bytes per grid triangle. (12/154 = 7.8% usage or >90% memory reduction.) Recreation of the performance numbers can be accomplished using Corona Renderer 5 and Embree. For more information, visit the Corona Renderer Blog: blog.corona-renderer.com/corona-renderer-5-for-3ds-max-released/

The Addams Family 2 - Gained a 10% to 20%—and sometimes 25%—efficiency in rendering, saving thousands of hours in rendering production time.
Testing Date: Results are based on data conducted by Cinesite 2020-21. 10% to up to 25% rendering efficiency/thousands of hours saved in rendering production time/15 hrs per frame per shot to 12-13 hrs.
Cinesite Configuration: 18-core Intel® Xeon® Scalable processors (W-2295) used in render farm, 2nd gen Intel Xeon processor-based workstations (W-2135 and -2195) used. Rendering tools: Gaffer, Arnold, along with optimizations by Intel® Open Image Denoise.

intel.

oneAPI Essentials

SYCL Program Structure

Learn about SYCL Program Structure, important SYCL Classes and Buffer Memory Model in SYCL

intel®

Learning Objectives

Explain the SYCL fundamental classes

Use device selection to offload kernel workloads

Decide when to use basic parallel kernels and ND-Range kernels

Understand various ways to synchronize data between host and device with using buffer memory model

Write a complete SYCL program that offload computation to accelerator device

oneAPIs implementation of SYCL

oneAPIs implementation of SYCL (DPC++)

= 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
- extensions aim to become core SYCL*, or Khronos* extensions

A Complete SYCL Program

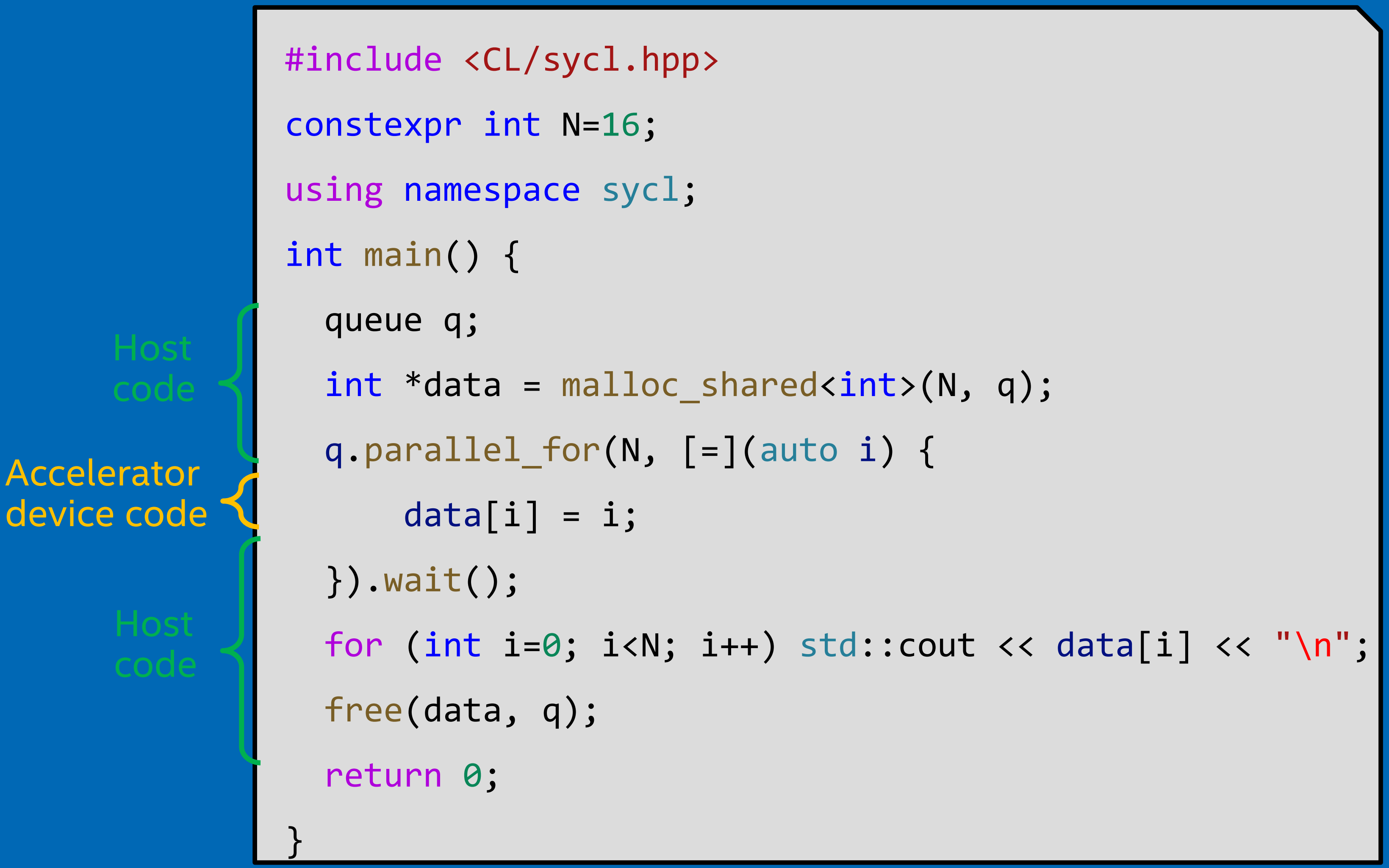
Single source

- 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
Host 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;
}
```

SYCL Classes

Device

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

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

Device Selector

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

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

Queue

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

```
queue q;


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

Choosing Where Device Kernels Run

Work is submitted to queues

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

Create queue targeting any device:	queue();

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

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

Kernel

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

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

DPC++ language and runtime

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

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
});
```

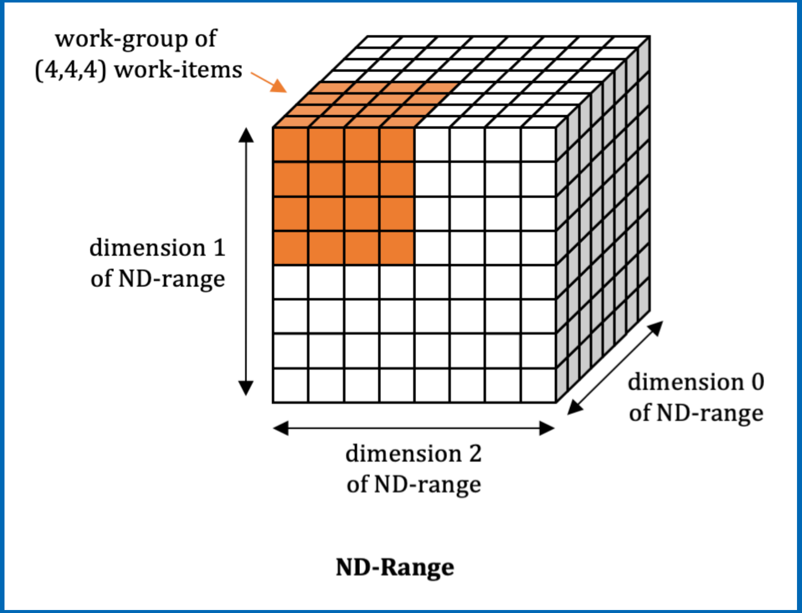
```
h.parallel_for(range<1>(1024), [=](item<1> item){
           auto idx = item.get_id();
           auto R = item.get_range();
           // CODE THAT RUNS ON DEVICE
});
```

ND-Range Kernels

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

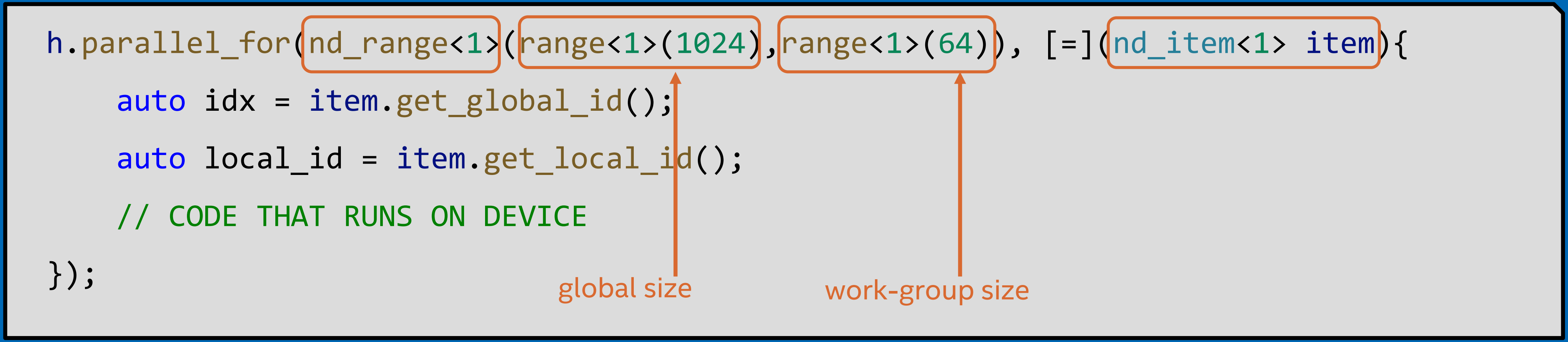
ND-Range kernel is another way to expresses parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units on hardware.

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



ND-Range Kernels

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



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

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

Memory Models

SYCL programs can either use a pointer-based memory model called Unified Shared Memory or can use Buffer-Accessor memory model

- Unified Shared Memory – pointer-based memory model to access data on host and device
- Buffer Memory Model – defines shared array of one, two or three dimensions that can be used by the SYCL kernel and has to be accessed using accessor classes

Unified Shared Memory

Unified Shared Memory enables accessing memory on the host and device with same pointer reference

Setup Unified Shared Memory

Host can initialize

Device can modify

Host has output

```
queue q;
auto data =  malloc_shared<int>(N, q);
for(int i=0;i<N;i++) data[i] = 10;
q.parallel_for(N, [=](auto i){
        data[i] += 1;
}).wait();
for(int i=0;i<N;i++) std::cout << data[i] << " ";
free(data, q);
```

Buffer Memory Model

Buffers: Encapsulate data in a SYCL application

- Across both devices and host!

Accessors: Mechanism to access buffer data

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

```
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] << " ";
```

SYCL Code Anatomy

- SYCL programs require the include of CL/sycl.hpp
- It is recommended to employ the namespace statement to save typing repeated references into the sycl namespace

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

SYCL Code Anatomy

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer 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

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

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

Step 3: submit a command for (asynchronous) execution

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

Step 5: send a kernel (lambda) for execution

Step 6: write a kernel

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

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;
}
```

Asynchronous Execution

Think of a SYCL application as two parts:

1. Host code
2. The graph of kernel executions

These execute independently, except at synchronizing operations

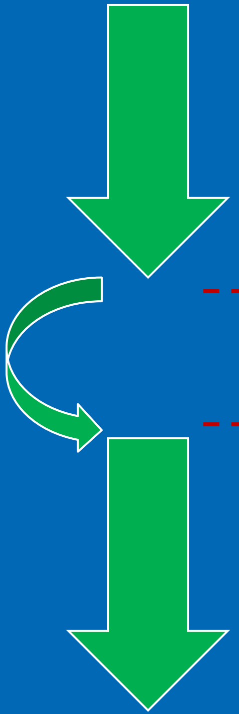
- The host code submits work to build the graph (and can do compute work itself)
- The graph of kernel executions and data movements executes asynchronously from host code, managed by the SYCL runtime

Asynchronous Execution

Host

Host code execution

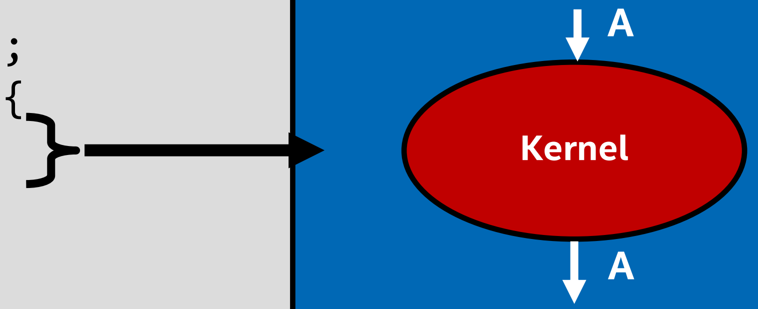
Enqueues kernel to graph, and keeps going



```
#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

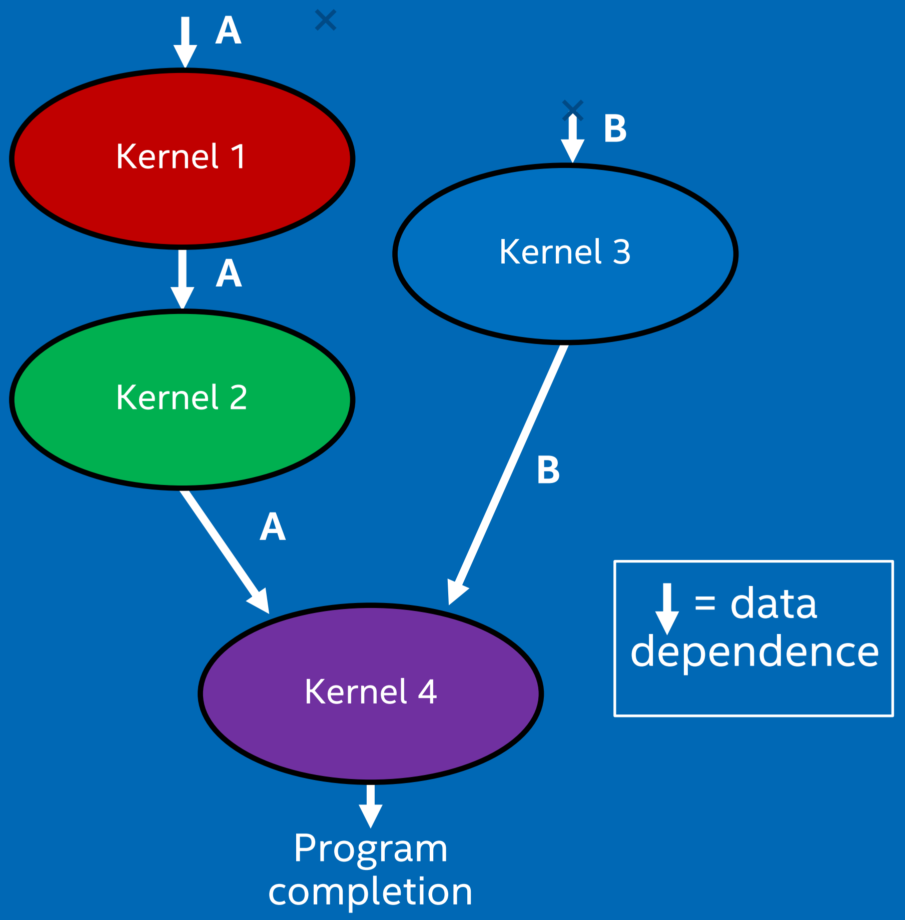
Graph executes asynchronously to host program



Implicit dependency between kernels

```
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
}
```

Automatic data and control dependence resolution!



Host Accessors

- An accessor which uses host buffer access target
- Created outside of command group scope
- The data that this gives access to will be available on the host
- Used to synchronize the data back to the host by constructing the host accessor objects

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.

intel®