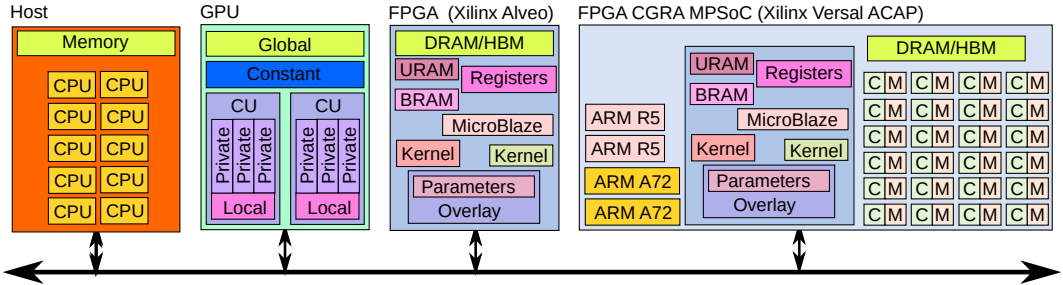
Advanced SYCL tutorial

Introduction and recapitulation of previous episodes

Ronan Keryell (ronan.keryell@amd.com)
Fellow Software Development Engineer @ AMD Research & Advanced Development
Khronos SYCL specification editor & ISO C++ committee member

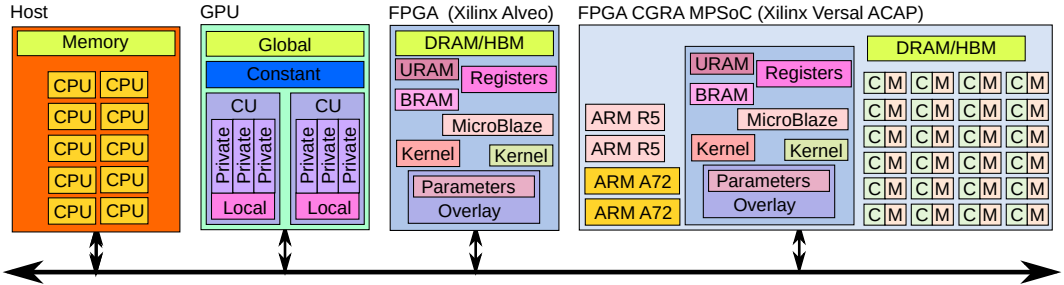
2023/04/18 @ SYCLcon

Typical modern/future system



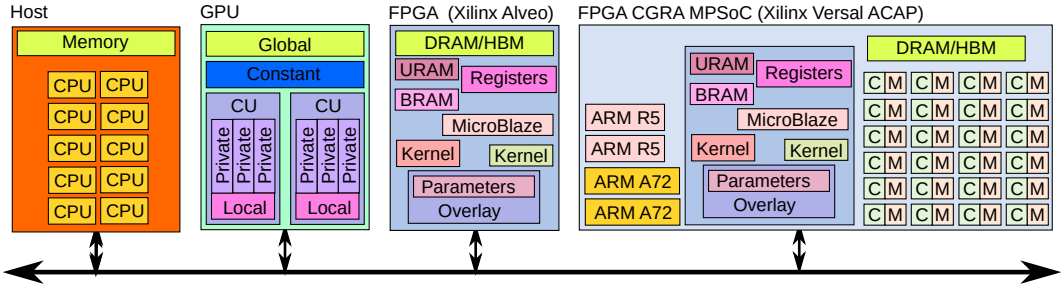
- Add your own accelerator to this picture...
- Scale this from embedded system to data-center/HPC level...
- Need a programming model for the *full* system...
- Tim Mattson's law: no new language! ☺

Typical modern/future system



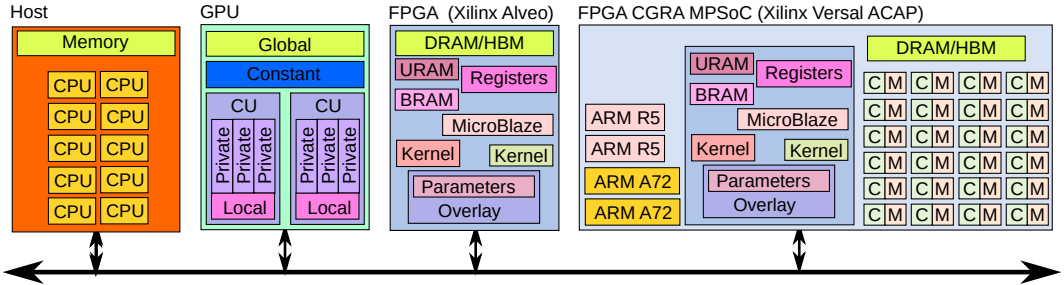
- Add your own accelerator to this picture...
- Scale this from embedded system to data-center/HPC level. . .
- Need a programming model for the *full* system. . .
- Tim Mattson's law: no new language! ☺

Typical modern/future system



- Add your own accelerator to this picture...
- Scale this from embedded system to data-center/HPC level. . .
- Need a programming model for the *full* system. . .
- Tim Mattson's law: no new language! ☺

Typical modern/future system



- Add your own accelerator to this picture...
- Scale this from embedded system to data-center/HPC level. . .
- Need a programming model for the *full* system. . .
- Tim Mattson's law: no new language! ☺

Remember C++ ?

2-line description by Bjarne Stroustrup

- Direct mapping to hardware
- Zero-overhead abstraction

Modern Python/C/Modern C++/Old C++

- Python 3.11

```python
v = [ 1, 2, 3, 5, 7 ]
print(v)
```

https://godbolt.org/z/Kq9vc1jhY

- C99 (also usable in C++)

```c
#include <stdio.h>
int a[] = { 1, 2, 3, 5, 7 };
for (int i = 0;
     i < sizeof(a)/sizeof(a[0]);
     ++i)
  printf("%d ", a[i]);
```

- C++23

```cpp
import std;
std::vector v { 1, 2, 3, 5, 7 };
std::println("{}", v);
```

- C++03

```cpp
#include <iostream>
#include <vector>
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(5);
v.push_back(7);
for (std::vector<int>::iterator i =
        v.begin(); i != v.end(); ++i)
  std::cout << *i << std::endl;
```

But...
No heterogeneous computing in C++

☹

KHRONOS GROUP

Close to 200 members worldwide
Any organization is welcome to join

AMD, Apple, arm, Epic Games, Google, Huawei, IKEA, Imagination, intel, NVIDIA, Qualcomm, Samsung, Sony, Tencent 腾讯, Valve, VeriSilicon

3D Incorporated, 3dMD, 7invensun 七鑫易维, acer, adidas, Adobe, AIMOTIVE, AMD, ALL3D, Alibaba.com, Almalence, amazon.com, Ambarella, AGI, ARSKAN, Argonne National Laboratory, ASHLEY, 中天恒星 AT STELLAR, AUTODESK, avataar, BASEMARK, BINOMIAL, blaize, BLIZZARD ENTERTAINMENT, blender, BOLT GRAPHICS, BOSCH Invented for life, botspot, BRENWILL, BROADCOM, ByteDance, CableLabs, CAPPASITY, CESIUM, cgtrader, codeweavers, CO, Collins Aerospace, Continental, CoreAVI, DGG, DASSAULT SYSTEMES, DMP, DIMENCO, DisplayLink, dopple, EA, emersya, ERICSSON, FIXSTARS, THE FORGE, Fraunhofer, GLENFLY, GO2 VIRTUAL, hp, hexa, HITACHI Inspire the Next, HOLOCHIP, HONOR, htc, id, igalia, immersion, Imperial College London, INNOSILICON, MTA SZTAKI, Inseye, JUICE, KDAB, KITTYHAWK, 炘知科技 KIFISENSE, KNU KYUNGPOOK NATIONAL UNIVERSITY, Linaro, LIV, Loft3Di, Los Alamos NATIONAL LABORATORY, logitech, LUNARG, Magic Leap, MATROX GRAPHICS STUDIO, matrix, matrox, MEDIATEK, Mercedes-Benz, Meta, METAVERSE, METAX 沐曦集成电路, Microsoft, moica, 摩尔线程 MOORE THREADS, mozilla, National Institute of Standards and Technology, NaturalPoint, The New York Times, NETFLIX, NIHON UNIVERSITY, Nintendo, NOKIA, nreal, NSI-TEXE, NXP, oppo, OSU Oregon State, otoy, OTSL, OVR, OXIDE, Penumbra, pluto, PERIPHERAL CORTEX, ptc, Red Hat, RENESAS, RIVE, Rokid, 서울대학교 SEOUL NATIONAL UNIVERSITY, shopify, shutterstock, SIRU INNOVATIONS, SIDEQUEST, SILICONARTS, SKYWORTH VR, socionext, STREAM High Performance Computing, SuperDNA 3D Lab, Synaptics, SYNOPSYS, TAKUMI, TAMPERE UNIVERSITY OF TECHNOLOGY, TELEDYNE IMAGING Everywhereyoulook, Target, Texas Instruments, Think Silicon, TRAVERSE RESEARCH, THREEDY instant 3Dhub, ThreeKit, tobii, ultraleap, Unity, University of BRISTOL, UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN, universität innsbruck, UNIVERSITÄT DES SAARLANDES, UNIVERSITY OF TORONTO, TOHOKU UNIVERSITY, University of Windsor, UX3D, VARJO, 融博半导体 CHARGEON SEMICONDUCTOR, vivo, vmware, VNTANA, YVR, wayfair, XE3D, XIANGDIXIAN 翔迪科技, ZUSE INSTITUTE BERLIN

Liaisons: Cooperation with industry associations and organizations

AREA, AUTOSAR, cmva, ETSI, GTI, HAPTICS INDUSTRY FORUM, ISO JTC1 IEC, Open Geospatial Consortium, Smithsonian, TIFCA, VESA, W3C, web3D CONSORTIUM

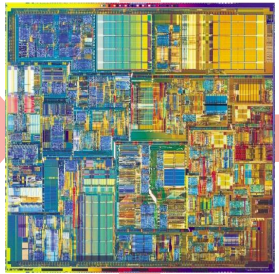
COLLADA

3D for the Web

- Real-time apps and games in-browser
- Efficiently delivering runtime 3D assets

WebGL glTF

NNEF OpenVX



OpenGL|SC OpenVG Vulkan OpenGL|ES OpenGL OpenXR

Vision and Neural Networks

- Tracking and odometry
- Scene analysis/understanding
- Neural Network inferencing

Real-time 2D/3D

- Virtual and Augmented Reality
- Cross-platform gaming and UI
- CG Visual Effects
- CAD and Product Design
- Safety-critical displays

Parallel Computation

- Machine Learning acceleration
- Embedded vision processing
- High Performance Computing (HPC)

OpenCL SPIR SYCL

SYCL 2020 from Khronos Group, published on 2021-02-09

KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON

Developers Conformance Membership News & Events About

SYCL™

SYCL (pronounced 'sickle') is a royalty-free, cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file.

SYCL 2020 is Here!

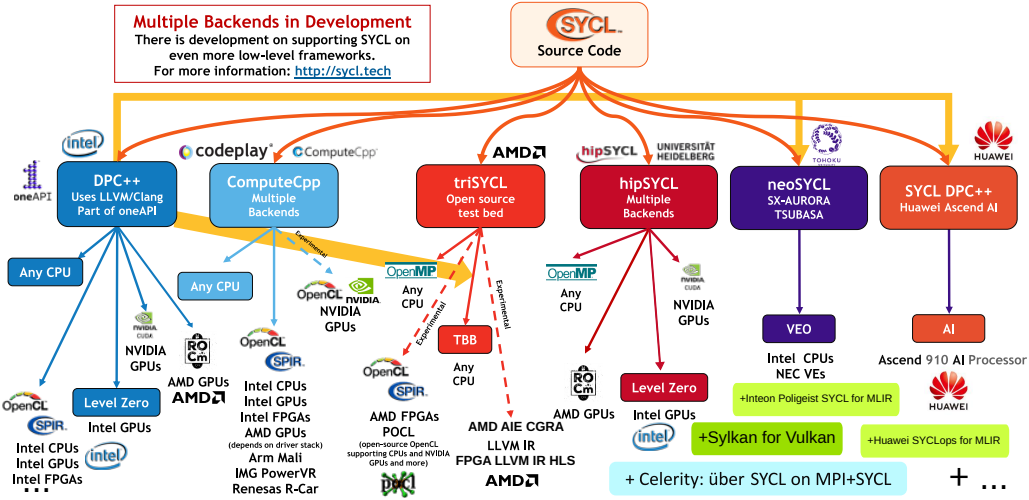
The SYCL 2020 Specification was launched on Feb 9th, 2021. The specification is now publicly available to enable feedback from developers and implementers before release of the SYCL 2020 Adopters Program to enable implementers to be officially conformant.

Press Release | Specification | Resources | Feedback | Blog | Slide Deck | Reference Guide

"SYCL 2020's primary goal is to achieve closer convergence with ISO C++, furthering our work to bring parallel heterogeneous programming to modern C++ through open standards. SYCL can leverage diverse processors to accelerate problems in many application domains including HPC, automotive, and machine learning," said Michael Wong, Codeplay distinguished engineer, ISO C++ Directions

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

SYCL ecosystem is growing



https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know

SYCL 2020 ≡ heterogeneous simplicity with modern C++

```
#include <iostream>
#include <sycl/sycl.hpp>
constexpr int n = 32;

int main () {
  sycl::buffer<int> buf { n };
  sycl::queue {}.submit([&](auto &h) {
      sycl::accessor a { buf, h, sycl::write_only, sycl::no_init };
      h.parallel_for(n, [=](auto i) { a[i] = i; });
  });
  for (sycl::host_accessor a { buf }; auto e : a)
      std::cout << e << std::end;
}
```

SYCL 2020 ≡ heterogeneous simplicity with modern C++

Abstract storage

- Host or device (remote) memory

```
#include <iostream>
#include <sycl/sycl.hpp>
constexpr int n = 32;

int main () {
  sycl::buffer<int> buf{ n };
  sycl::queue {}.submit([&](auto &h) {
      sycl::accessor a { buf, h, sycl::write_only, sycl::no_init };
      h.parallel_for(n, [=](auto i) { a[i] = i; });
  });
  for (sycl::host_accessor a { buf }; auto e : a)
      std::cout << e << std::end;
}
```

SYCL 2020 ≡ heterogeneous simplicity with modern C++

Abstract storage

- Host or device (remote) memory

Code executed on device ("kernel")

- "Single-source"
- Seamless integration in host code
- Type-safety
- Asynchronous execution

```
#include <iostream>
#include <sycl/sycl.hpp>
constexpr int n = 32;

int main () {
  sycl::buffer<int> buf{ n };
  sycl::queue {}.submit([&](auto &h) {
      sycl::accessor a { buf, h, sycl::write_only, sycl::no_init };
      h.parallel_for(n, [=](auto i) { a[i] = i; });
  });
  for (sycl::host_accessor a { buf }; auto e : a)
      std::cout << e << std::end;
}
```

SYCL 2020 ≡ heterogeneous simplicity with modern C++

Abstract storage

- Host or device (remote) memory

```
#include <iostream>
#include <sycl/sycl.hpp>
constexpr int n = 32;

int main () {
  sycl::buffer<int> buf{ n };
  sycl::queue {}.submit([&](auto &h) {
      sycl::accessor a { buf, h, sycl::write_only, sycl::no_init };
      h.parallel_for(n, [=](auto i) { a[i] = i; });
  });
  for (sycl::host_accessor a { buf }; auto e : a)
      std::cout << e << std::end;
}
```

Code executed on device ("kernel")

- "Single-source"
- Seamless integration in host code
- Type-safety
- Asynchronous execution

Accessor

- Express access intention
- Implicit data flow graph
- Automatic data transfers across devices
- Overlap computation & communication

SYCL 2020 ≡ heterogeneous simplicity with modern C++

Queue

- Direct work to specific accelerator
- Submission of a command group

Abstract storage

- Host or device (remote) memory

```
#include <iostream>
#include <sycl/sycl.hpp>
constexpr int n = 32;

int main () {
  sycl::buffer<int> buf{ n };
  sycl::queue {}.submit([&](auto &h) {
      sycl::accessor a { buf, h, sycl::write_only, sycl::no_init };
      h.parallel_for(n, [=](auto i) { a[i] = i; });
  });
  for (sycl::host_accessor a { buf }; auto e : a)
      std::cout << e << std::end;
}
```

Code executed on device ("kernel")

- "Single-source"
- Seamless integration in host code
- Type-safety
- Asynchronous execution

Accessor

- Express access intention
- Implicit data flow graph
- Automatic data transfers across devices
- Overlap computation & communication

SYCL 2020 with unified shared memory (USM)

```cpp
// Using buffers and accessors

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

constexpr int n = 32;

int main () {
 sycl::buffer<int> buf { n };

 sycl::queue {}.submit([&](auto &h) {
  sycl::accessor a { buf, h, sycl::write_only,
                     sycl::no_init };
  h.parallel_for(n, [=](auto i) { a[i] = i; });
 });

 for (sycl::host_accessor a { buf }; auto e : a)
  std::cout << e << std::endl;
}
```

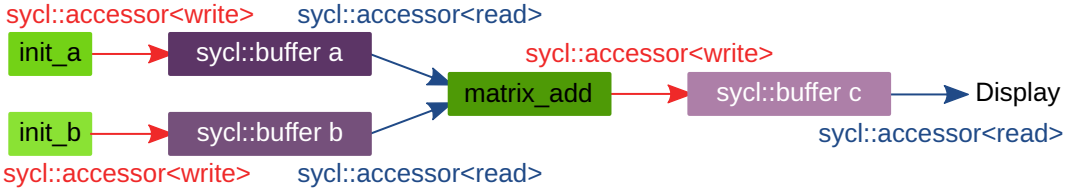
```cpp
// Using USM only

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

constexpr int n = 32;

int main () {
 sycl::queue q;
 int* a = sycl::malloc_shared<int>(n, q);

 q.parallel_for(n, [=](auto i) { a[i] = i; });
 q.wait();

 for (int i = 0; i < n; i++)
  std::cout << a[i] << std::endl;

 sycl::free(a, q);
}
```

Matrix addition as implicit task graph programming

```
#include <sycl/sycl.hpp>
#include <iostream>
// Size of the matrices
constexpr size_t n = 2000;
constexpr size_t m = 3000;
int main() {
 // Create a queue to work on default device
 sycl::queue q;
 // Create some 2D buffers of float for our matrices
 sycl::buffer<double, 2> a({ n, m });
 sycl::buffer<double, 2> b({ n, m });
 sycl::buffer<double, 2> c({ n, m });

 // Launch a first asynchronous kernel to initialize a
 q.submit([&](auto& cgh) {
   // The kernel write a, so get a write accessor on it
   sycl::accessor A { a, cgh, sycl::write_only, sycl::no_init };

   // Enqueue parallel kernel on a n*m 2D iteration space
   cgh.parallel_for<class init_a>({ n, m },
                    [=] (sycl::id<2> index) {
                      A[index] = index[0]*2 + index[1];
                    });
 });

 // Launch an asynchronous kernel to initialize b
 q.submit([&](auto& cgh) {
   // The kernel write b, so get a write accessor on it
   sycl::accessor B { b, cgh, sycl::write_only, sycl::no_init };
   /* From the access pattern above, the SYCL runtime detect
      this command_group is independant from the first one
      and can be scheduled independently */
   // Enqueue a parallel kernel on a n*m 2D iteration space
   cgh.parallel_for<class init_b>({ n, m },
                    [=] (sycl::id<2> index) {
                      B[index] = index[0]*2014 + index[1]*42;
                    });
 });
 // Launch an asynchronous kernel to compute matrix addition c = a + b
 q.submit([&](auto& cgh) {
   // In the kernel a and b are read, but c is written
   sycl::accessor A { a, cgh, sycl::read_only };
   sycl::accessor B { b, cgh, sycl::read_only };
   sycl::accessor C { c, cgh, sycl::write_only, sycl::no_init };
   // From these accessors, the SYCL runtime will ensure that when
   // this kernel is run, the kernels computing a and b completed

   // Enqueue a parallel kernel on a n*m 2D iteration space
   cgh.parallel_for<class matrix_add>({ n, m },
                    [=] (sycl::id<2> index) {
                      C[index] = A[index] + B[index];
                    });
 });
 /* Request an access to read c from the host-side. The SYCL runtime
    ensures that c is ready when the accessor is returned */
 sycl::host_accessor C { c };
 std::cout << std::endl << "Result:" << std::endl;
 for(size_t i = 0; i < n; i++)
   for(size_t j = 0; j < m; j++)
     // Compare the result to the analytic value
     if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
       std::cout << "Wrong value " << C[i][j] << " on element "
                 << i << ' ' << j << std::endl;
       exit(-1);
     }
 std::cout << "Good computation!" << std::endl;
 return 0;
}
```

Asynchronous task graph model

- Theoretical graph of an application described *implicitly* with kernel tasks using buffers through accessors



- Possible schedule by SYCL runtime:

- Automatic overlap of kernels & communications
  - No need for complex events and queue management
  - Even better when looping around in an application
  - Assume it will be translated into pure back-end event graph
  - Runtime uses as many threads & back-end queues as necessary

Other SYCL features

* ≡ In this tutorial

- In-order queue
- Multi-devices
- Atomic references
- Reductions
- Work-group and sub-group algorithms
- Work-group local memory
- Events
- Properties
- Aspects
- Images
- Small vectors (sycl::vec & sycl::marray)
- Multi-pointers
- Kernel bundles
- Specialization constants
- Error handling
- Library functions
- Back-ends
- Interoperability
- ...

21 lines of heterogeneous serendipity on my desktop

https://github.com/triSYCL/sycl/blob/sycl/unified/next/sycl/test/vitis/disabled/inclusive_devices.cpp

```
#include <iostream>
#include <sycl/sycl.hpp>
int main() {
  sycl::buffer<int> v { 10 };
  auto run = [&] (auto device_name, auto work) {
    sycl::queue { [&](sycl::device dev) {
      return (device_name == dev.template get_info<sycl::info::device::name>()) - 1;
    } }.submit([&](auto& h) {
      auto a = sycl::accessor { v, h };
      h.parallel_for(a.size(), [=](int i) { work(i, a); });
    });
  };
  run("Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz", [](auto i, auto a) { a[i] = i; });
  run("Quadro P400", [](auto i, auto a) { a[i] = 2 * a[i]; });
  run("Intel(R) FPGA Emulation Device", [](auto i, auto a) { --a[i]; });
  run("AMD Radeon VII", [](auto i, auto a) { a[i] = a[i] * a[i]; });
  run("xilinx_u200_gen3x16_xdma_base_1", [](auto i, auto a) { a[i] += + 3; });
  for (auto e : sycl::host_accessor { v })
    std::cout << e << ", ";
  std::cout << std::endl;
}
```

```
$DPCPP_HOME/llvm/build/bin/clang++ -std=c++2b -fsycl -fsycl-targets=spir64_x86_64,nvptx64-nvidia-cuda,amdgcn-amd-amdhsa,fpga64_hls_hw,spir64_fpga
-Xsycl-target-backend=amdgcn-amd-amdhsa -offload-arch=gfx906 -Xsycl-target-backend=nvptx64-nvidia-cuda -offload-arch=sm_61 inclusive_devices.cpp -o
inclusive_devices
```

21 lines of heterogeneous serendipity on my desktop

https://github.com/triSYCL/sycl/blob/sycl/unified/next/sycl/test/vitis/disabled/inclusive_devices.cpp

- No `template` or `typename` or `class` or... ☺
- No extension or attribute or... ☺
- Generic & type-safe
- No explicit data motion or boiler-plate code
- Different accelerators/vendors in same program!

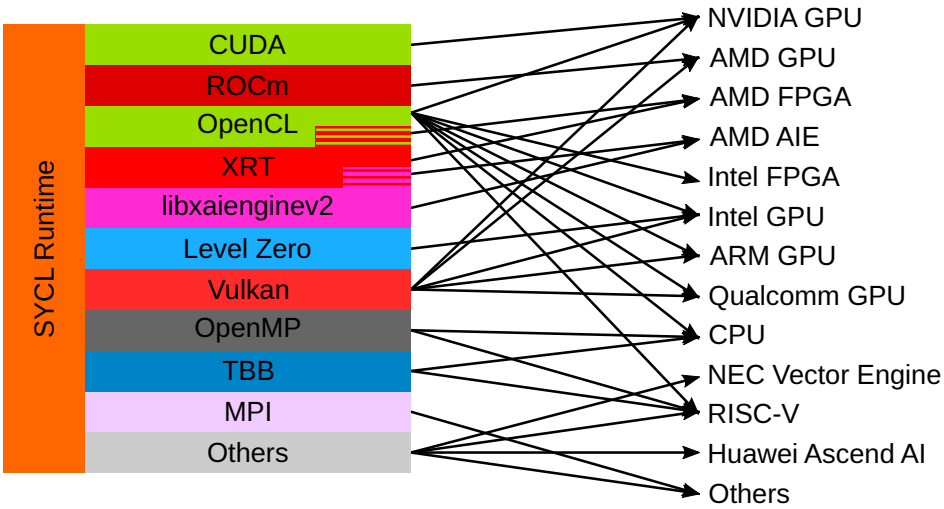
```
#include <iostream>
#include <sycl/sycl.hpp>
int main() {
  sycl::buffer<int> v { 10 };
  auto run = [&] (auto device_name, auto work) {
    sycl::queue { [&](sycl::device dev) {
      return (device_name == dev.template get_info<sycl::info::device::name>()) - 1;
    } }.submit([&](auto& h) {
      auto a = sycl::accessor { v, h };
      h.parallel_for(a.size(), [=](int i) { work(i, a); });
    });
  };
  run("Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz", [](auto i, auto a) { a[i] = i; });
  run("Quadro P400", [](auto i, auto a) { a[i] = 2 * a[i]; });
  run("Intel(R) FPGA Emulation Device", [](auto i, auto a) { --a[i]; });
  run("AMD Radeon VII", [](auto i, auto a) { a[i] = a[i] * a[i]; });
  run("xilinx_u200_gen3x16_xdma_base_1", [](auto i, auto a) { a[i] += + 3; });
  for (auto e : sycl::host_accessor { v })
    std::cout << e << ", ";
  std::cout << std::endl;
}
```

```
$DPCPP_HOME/llvm/build/bin/clang++ -std=c++2b -fsycl -fsycl-targets=spir64_x86_64,nvptx64-nvidia-cuda,amdgcn-amd-amdhsa,fpga64_hls_hw,spir64_fpga
-Xsycl-target-backend=amdgcn-amd-amdhsa -offload-arch=gfx906 -Xsycl-target-backend=nvptx64-nvidia-cuda -offload-arch=sm_61 inclusive_devices.cpp -o
inclusive_devices
```

Inclusive heterogeneous computing with SYCL...

No transistors left behind!

Some of the existing SYCL back-ends



Use cases for SYCL interoperability with backend

- Porting existing code
  - Code already based on OpenCL/CUDA/OpenMP/HIP/... (or whatever backend)
  - Want to change just part of application to use SYCL
- Incorporating a backend module into a SYCL application
  - Application based on SYCL
  - Want to call some OpenCL/CUDA/OpenMP/HIP/... library (or whatever backend)
- Take advantage of backend-specific features
- Disadvantage: reduces portability!
  - Not all implementations may support your backend

⇝ Unique feature of SYCL!

Aksel ALPAY, Thomas APPLENCOURT, Gordon BROWN, Ronan KERYELL and Gregory LUECK. “Using interoperability mode in SYCL 2020.” In SYCLcon 2022: International Workshop on SYCL. Association for Computing Machinery, May 2022. doi:10.1145/3529538.3529997.

`https://www.iwocl.org/wp-content/uploads/39-presentation-iwocl-syclcon-2022-aksel.pdf`

`https://www.youtube.com/watch?v=XIPhuesdqYE`

Type 1: SYCL object from backend object

“Higher-level XRT” for AMD FPGA in 43 lines with SYCL

```
#include <cassert>
#include <sycl/sycl.hpp>
#include <sycl/ext/xilinx/xrt.hpp>
#include <xrt.h>
#include <xrt/xrt_kernel.h>
constexpr int size = 4;
int main() {
  sycl::buffer<int> a { size };
  sycl::buffer<int> b { size };
  sycl::buffer<int> c { size };
  {
    sycl::host_accessor a_a { a };
    sycl::host_accessor a_b { b };
    for (int i = 0; i < size; ++i) {
      a_a[i] = i;
      a_b[i] = i + 42;
    }
  }
  sycl::queue q;
  xrt::device xdev =
    sycl::get_native<sycl::backend::xrt>(q.get_device());
  xrt::kernel xk { xdev, xdev.load_xclbin("vadd.hw_emu.xclbin"),
                   "vadd" };
```

```
  sycl::kernel k
    { sycl::make_kernel<sycl::backend::xrt>(xk, q.get_context()) };

  q.submit([&](sycl::handler& cgh) {
    cgh.set_args(sycl::accessor { a, cgh, sycl::read_only },
                 sycl::accessor { b, cgh, sycl::read_only },
                 sycl::accessor { c, cgh, sycl::write_only,
                                  sycl::no_init },
                 size);
    cgh.single_task(k);
  });
  {
    sycl::host_accessor a_a { a };
    sycl::host_accessor a_b { b };
    sycl::host_accessor a_c { c };
    for (int i = 0; i < size; ++i) {
      int res = a_a[i] + a_b[i];
      int val = a_c[i];
      assert(val == res);
    }
  }
}
```

https://github.com/keryell/heterogeneous_examples/blob/main/vector_add/SYCL/vector_add_XRT_interoperability.cpp

Typical usage

Adding SYCL functionality to an existing backend-specific application

Type 2: backend object from SYCL object

```
void MyFunc(sycl::device dev) {
#ifdef SYCL_BACKEND_OPENCL
  cl_device_id clDev = sycl::get_native<sycl::backend::opencl>(dev);

  char builtins[SIZE];
  size_t sz;
  clGetDeviceInfo(clDev, CL_DEVICE_BUILT_IN_KERNELS, SIZE, builtins, &sz);
  /* Use OpenCL builtin kernel...*/
#else
  /* fallback if no OpenCL backend */
#endif
}
```

Typical usage

Incorporate a backend-specific library into a SYCL application or take advantage of a backend-specific feature

Type 3: schedule a backend-specific command

```
void MyFunc(sycl::queue q, sycl::buffer<int> buf) {
#ifdef SYCL_BACKEND_OPENCL
  q.submit([&](sycl::handler &cgh) {
    sycl::accessor acc{buf, cgh};
    cgh.host_task([=](sycl::interop_handle &ih) {
      cl_mem clMem = ih.get_native_mem<sycl::backend::opencl>(acc)[0];
      /* use OpenCL APIs with clMem */
    });
  });
#endif
}
```

Typical usage

Incorporate a backend-specific library or feature into a SYCL task graph

3 main implementations

- ComputeCpp by Codeplay
- hipSYCL
- Clang/LLVM SYCL Intel oneAPI DPC++

ComputeCpp by Codeplay

https://www.codeplay.com/products/computesuite/computecpp

- Codeplay is initiator & chair of SYCL Khronos working-group
  - First SYCL demo ever on AMD booth at SC 2014 on AMD GPU!
  - First SYCL 1.2.1 full-compliant implementation in July 2018
  - Started as a developer environment for gaming (Sony PS2 & PS3) in 2000's ☺
  - Clang/LLVM outlining compiler generating SPIR(-V) and other back-ends
  - Runtime for OpenCL device, CPU and other back-ends (CUDA, Vulkan...)
- Free community edition + non-free for customer support
  - Provide several libraries & frameworks such as SYCL versions of Eigen & TensorFlow
- Acquired by Intel in 2022 but still works as independent company
- Implement compute & graphics stacks for customers (Renesas, Imagination...)
- Highly engaged in ISO C++, ADAS (MISRA & AUTOSAR C++), ML, safety critical standards...

hipSYCL

`https://github.com/illuhad/hipSYCL`

- Started by PhD student Aksel Alpay @ Heidelberg University Computing Centre (URZ), Germany ⇝ now full-time tech lead
  - Few full-time employees + around 10 developers part time
- First to demonstrate that SYCL is more general than OpenCL
- Different implementation modes on top of HIP and CUDA
  - Interoperable with CPU, AMD ROCm & Nvidia CUDA environments *at the same time*
  - Interoperable with AMD & Nvidia libraries ☺
  - Can use directly AMD & Nvidia C++ intrinsics ☺
    - Nvidia TensorCore, Nvidia ray-tracing, graphics interoperability...
- Can use Clang/LLVM CUDA/HIP or `nvc++`
- Can also target oneAPI Level Zero
- New SSCP (single-source single-compiler pass) flow
  - Parse code once to CPU and AMD+Intel+Nvidia GPU code for lower compilation time
  - Use LLVM IR as portable IR to JIT towards portable (SPIR-V, PTX) or native (AMD GPU)
    - Supplement lacking of portable IR on AMD GPU
- Good support for CPU too, as pure library or LLVM back-end

Clang/LLVM SYCL oneAPI DPC++ by Intel

`https://github.com/intel/llvm/tree/sycl`

- Part of Intel oneAPI strategy 2018/12/12 `https://www.phoronix.com/scan.php?page=news_item&px=Intel-oneAPI-Announcement`
- Open-sourced in January 2019 with unifying goal: Clang/LLVM up-streaming! ☺
- 2019/06/19 *Direct programming: oneAPI contains a new direct programming language, Data Parallel C++ (DPC++), an open, cross-industry alternative to single architecture proprietary languages. DPC++ delivers parallel programming productivity and performance using a programming model familiar to developers. DPC++ is based on C++, incorporates SYCL [1.2.1] from The Khronos Group and includes language extensions developed in an open community process.* `https://newsroom.intel.com/news/intels-one-api-project-delivers-unified-programming-model-across-diverse-arch`
- Started as different language on top of SYCL 1.2.1 but since SYCL 2020 DPC++ is just 1 SYCL 2020 implementation + set of SYCL extensions
- Based on open-source SPIR-V LLVM translator
- Target CPU, GPU & Intel FPGA
- Runtime for OpenCL, Level Zero, CUDA, HIP

Developing a SYCL ecosystem

- A programming language is nothing without an ecosystem ☹
- Codeplay started open-source libraries around 2015
  - SYCL-BLAS
  - SYCL-DNN (base of SYCL TensorFlow)
  - SYCL ParallelSTL (C++17 STL with SYCL parallel policies)
- Intel started oneAPI in 2018, similar to Nvidia CUDA ecosystem
- In 2022 oneAPI is independent from Intel with its own board committee (currently chaired by Rod Burns, Codeplay)
  - `https://github.com/oneapi-src/oneAPI-tab`

oneAPI ecosystem

`https://www.oneapi.io/spec`

- DPC++: SYCL is oneAPI's core language for programming accelerators and multiprocessors with some SYCL extensions. Allows developers to reuse code across hardware targets (CPUs and accelerators such as GPUs and FPGAs) and to tune for a specific architecture
- oneDPL: companion to the DPC++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions
- oneDNN: high-performance implementations of primitives for deep learning frameworks
- oneCCL: communication primitives for scaling deep learning frameworks across multiple devices
- Level Zero: system interface for oneAPI languages and libraries
- oneDAL: algorithms for accelerated data science
- oneTBB: library for adding thread-based parallelism to complex applications on multiprocessors
- oneVPL: algorithms for accelerated video processing
- oneMKL: high-performance math routines for science, engineering, and financial applications

Most of these libraries can redirect to native back-end libraries and work with different SYCL implementations

Simplify porting code from CUDA

- SYCL (2020) is higher-level than CUDA (2007) to ease programmer life ☺
  - Buffers for abstract storage, accessors to express dependencies, exceptions instead of error codes...
- But when porting old CUDA code: dealing with raw device pointers, explicit kernel launches without any dependencies... ☹
- ⇝ SYCL 2020 also added lower level API
  - USM to allocate memory and use pointers, ordered queue to fit CUDA default stream
- ReSYCLator Eclipse plugin from Cedevelop 2018
- SYCLomatic from oneAPI
  - `https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html`
  - Open-source Clang-based source-to-source migration tool (similar to HIPify from CUDA to HIP)
  - YAML syntax to express easily new transformations

SYCL extensions

- SYCL ≡ standard for generic heterogeneous programming
- Extensions to use specific hardware features
- Extensions allows also to experiment new features for future SYCL version
  - Similar to ISO C++ TS (Technical Specifications) `https://en.cppreference.com/w/cpp/experimental`
  - Get feedback from implementers and users ☺
  - Can become part of the standard if SYCL members agree
- Some of the existing extensions
  - Codeplay ComputeCpp `https://developer.codeplay.com/products/computecpp/ce/2.11.0/guides/computecpp-extensions` `https://github.com/codeplaysoftware/standards-proposals`
  - oneAPI DPC++ (FPGA, AMX...) `https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions`
  - hipSYCL `https://github.com/illuhad/hipSYCL/blob/develop/doc/extensions.md`
  - Samsung PiM extensions
  - AMD extensions (FPGA, AIE, C++23)
  - Celerity `https://celerity.github.io`
  - ...

SYCL SC for Safety Critical Systems is coming

- Huge demand for embedded acceleration @ low power: automotive, avionics...
- Car: 100+ CPU and accelerators from various vendors
  - World-wide politics can prevent exporting some semiconductors to some countries
  - Need software agility and portability
  - Liaison group between Khronos & AUTOSAR
- March 22, 2022: Khronos launched SYCL Safety-Critical Exploratory Forum
  `https://www.khronos.org/syclsc`
- Targeting possibly RTCA DO-178C Level A / EASA ED-12C (avionics), ISO 26262/21448 (automotive), IEC 61508 (industrial), and IEC 62304 (Medical)
- Actively participating members: AirBus, AMD, ARM, BSC, Codeplay [Intel], CoreAVI, Intellias, Mercedes-Benz, Mobileye [Intel], Nvidia, Qualcomm, VolksWagen
- First envisioned back-end: Vulkan SC, to ease certification of lower runtime
- SYCL SC started as a working group on 2023/03/27

Conclusion

- SYCL is *the* inclusive standard for accelerated computing
  - A dozen of implementations with back-ends & interoperability with other ecosystems (Vulkan, OpenCL, OpenMP, TBB, proprietary: CUDA, HIP, Level 0...)
- Open-source + open standards
  - No user locked-in!
  - Benefit from collaboration for better code quality
  - Still not happy? Do it the standard way! Participate to the standards and to open-source implementations to have a real impact! ☺
- Pure single-source C++ domain-specific language to complement ISO C++
  - Run on CPU with normal compiler for *emulation* and debug
- SYCL for Safety Critical Systems: think beyond usual HPC context for bigger and more heterogeneous markets

Now, let's dive into the tutorial matter...