WIKIPEDIA

SYCL

SYCL is a higher-level programming model for OpenCL as a single-source domain specific embedded language (DSEL) based on pure C++11 for SYCL 1.2.1 to improve programming productivity. This is a standard developed by Khronos Group, announced in March 2014.

SYCL

Original author(s)	Khronos Group
Developer(s)	Khronos Group
Initial release	March 2014
Stable release	1.2.1 revision 7 / April 27, 2020
Operating system	Cross-platform
Platform	Cross-platform
Type	High-level programming language
Website	www.khronos.org/sycl/ (https://www.khronos.org/sycl/)

Contents

- **Purpose**
- **Versions**
- **Example**
- **Tutorials**
- **Comparison with other APIs**
- **See also**
- **References**
- **External links**

Purpose

SYCL (pronounced 'sickle') is a royalty-free, cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++. SYCL enables single source development where C++ template functions can contain both host and device code to construct complex algorithms that use OpenCL acceleration, and then re-use them throughout their source code on different types of data.

While originally developed for use with OpenCL and SPIR, it is actually a more general heterogeneous framework able to target other systems. For example, the hipSYCL implementation targets CUDA. While the SYCL standard started as the higher-level programming model sub-group of the OpenCL working group, it is a Khronos Group workgroup independent from the OpenCL working group since September 20, 2019.

Versions

The latest version is SYCL 1.2.1 revision 7 which was published on April 27, 2020 (the first version was published on December 6, 2017[1]).

SYCL was introduced at GDC in March 2014 with provisional version 1.2,[2] then the SYCL 1.2 final version was introduced at IWOCL 2015 in May 2015.[3]

SYCL 2.2 provisional was introduced at IWOCL 2016 in May 2016[4] targeting C++14 and OpenCL 2.2. But the SYCL committee preferred not to finalize this version and is working on a more flexible SYCL specification to address the increasing diversity of current accelerators, including artificial-intelligence engines.

The public version is:

- SYCL 1.2.1 targeting OpenCL 1.2 hardware features with an OpenCL 1.2 interoperability mode.

Example

The following example shows the single-source pure C++ programming model defining an implicit task graph of 3 kernels running on a default accelerator.

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

// Declare some types just to give names to compute kernels
class init_a;
class init_b;
class matrix_add;

using namespace cl::sycl;

// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;

int main() {
  // Create a queue to work on default device
  queue q;
  // Create some 2D buffers with N×M float values for our matrices
  buffer<double, 2> a{{N, M}};
  buffer<double, 2> b{{N, M}};
  buffer<double, 2> c{{N, M}};
  // First launch an asynchronous kernel to initialize buffer "a"
  q.submit([&](handler &cgh) {
    // The kernel writes "a", so get a write accessor to it
    auto A = a.get_access<access::mode::write>(cgh);

    // Enqueue parallel kernel on an N×M 2D iteration space
    cgh.parallel_for<init_a>(range<2>{N, M}, [=](item<1> index) {
      A[index] = index[0] * 2 + index[1];
    });
  });
  // Launch an asynchronous kernel to initialize buffer "b"
  q.submit([&](handler &cgh) {
    // The kernel writes to "b", so get a write accessor on it
    auto B = b.get_access<access::mode::write>(cgh);
    // Enqueue a parallel kernel on an N×M 2D iteration space
    cgh.parallel_for<init_b>(range<2>{N, M}, [=](item<1> index) {
      B[index] = index[0] * 2014 + index[1] * 42;
    });
  });
  // Launch an asynchronous kernel to compute matrix addition c = a + b
  q.submit([&](handler &cgh) {
    // In the kernel "a" and "b" are read, but "c" is written.
    // Since the kernel reads "a" and "b", the runtime will implicitly add
    // a producer-consumer dependency to the previous kernels producing them.
    auto A = a.get_access<access::mode::read>(cgh);
    auto B = b.get_access<access::mode::read>(cgh);
    auto C = c.get_access<access::mode::write>(cgh);

    // Enqueue a parallel kernel on an N×M 2D iteration space
    cgh.parallel_for<matrix_add>(
        range<2>{N, M}, [=](item<1> index) { C[index] = A[index] + B[index]; });
```

```
  });
  /* Request an access to read "c" from the host-side. The SYCL runtime
     will wait for "c" to be ready available on the host side before
     returning the accessor.
     This means that there is no communication happening in the nested loop below.
   */
  auto C = c.get_access<access::mode::read>();
  std::cout << "\nResult:\n";
  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 << '\n';
        exit(EXIT_FAILURE);
      }

  std::cout << "Good computation!\n";
}
```

Tutorials

There are a few tutorials in the ComputeCpp SYCL guides.[5]

Comparison with other APIs

The open standards SYCL and OpenCL are similar to vendor-specific CUDA from Nvidia.

In the Khronos Group realm, OpenCL is the low-level *non-single source* API and SYCL is the high-level *single-source* C++ domain-specific embedded language.

By comparison, the *single-source* C++ domain-specific embedded language version of CUDA, which is actually named "CUDA *Runtime* API", is somehow similar to SYCL. But there is actually a less known *non single-source* version of CUDA which is called "CUDA *Driver* API", similar to OpenCL, and used for example by the CUDA *Runtime* API implementation itself.

SYCL extends the C++ AMP features relieving the programmer from explicitly transferring the data between the host and devices, by opposition to CUDA (before the introduction of Unified Memory in CUDA 6).

SYCL is higher-level than C++ AMP and CUDA since you do not need building an explicit dependency graph between all the kernels, and provides you automatic asynchronous scheduling of the kernels with communication and computation overlap. This is all done by using the concept of accessors, without requiring any compiler support.

By opposition to C++ AMP and CUDA, SYCL is a pure C++ DSEL without any C++ extension, allowing some basic CPU implementation relying on pure runtime without any specific compiler. This is very useful for debugging application or to prototype for a new architecture without having the architecture and compiler available yet.

The hipSYCL implementation adds SYCL higher-level programming to CUDA.

See also

- C++
- C++ AMP
- CUDA

- Metal
- OpenACC
- OpenCL
- OpenMP
- SPIR
- Vulkan

References

1. Khronos Group (6 December 2017). "The Khronos Group Releases Finalized SYCL 1.2.1" (https://www.khronos.org/news/press/the-khronos-group-releases-finalized-sycl-1.2.1). *Khronos*. Retrieved 12 December 2017.
2. Khronos Group (19 March 2014). "Khronos Releases SYCL 1.2 Provisional Specification" (https://www.khronos.org/news/press/khronos-releases-sycl-1.2-provisional-specification). *Khronos*. Retrieved 20 August 2017.
3. Khronos Group (11 May 2015). "Khronos Releases SYCL 1.2 Final Specification" (https://www.khronos.org/news/press/khronos-releases-sycl-1.2-final-specification-c-single-source-heterogeneous). *Khronos*. Retrieved 20 August 2017.
4. Khronos Group (18 April 2016). "Khronos Releases OpenCL 2.2 Provisional Specification with OpenCL C++ Kernel Language" (https://www.khronos.org/news/press/khronos-releases-opencl-2.2-provisional-spec-opencl-c-kernel-language). *Khronos*. Retrieved 18 September 2017.
5. "Introduction to GPGPU programming with SYCL" (https://developer.codeplay.com/computecppce/latest/sycl-guide-introduction). *Codeplay*. Retrieved 3 October 2017.

External links

- Khronos SYCL webpage (https://www.khronos.org/sycl/)
- The SYCL specifications in Khronos registry (https://www.khronos.org/registry/SYCL/)
- C++17 ParallelSTL in SYCL (https://github.com/KhronosGroup/SyclParallelSTL)
- SYCL tech resources (http://sycl.tech/)
- Codeplay ComputeCpp SYCL implementation (https://www.codeplay.com/products/computesuite/computecpp)
- Implementation of SYCL started by Intel with the goal of Clang/LLVM up-streaming (https://github.com/intel/llvm/tree/sycl)
- hipSYCL: implementation of SYCL 1.2.1 over AMD HIP/NVIDIA CUDA (https://github.com/illuhad/hipSYCL)
- triSYCL open-source SYCL implementation (https://github.com/triSYCL/triSYCL)
- SYCL Conference @ IWOCL (https://www.iwocl.org/sycl-conference/)