Modern C++ for accelerators: a SYCL deep dive
May 2018, Embedded Vision Summit

Andrew Richards
C++ for heterogeneous acceleration

• C++ kernel language: add C++ to existing OpenCL C kernels
• SYCL: single-source C++ enables easy porting
  • How to port
  • How to get great performance
  • How to add safety in future
• ISO C++: The main C++ standard with acceleration coming
C++ kernel language

- Separate-source, OpenCL-style, but with C++
- Supports the OpenCL 2.2 feature set
- Kernel language compiler is open-source
  - https://github.com/KhronosGroup/OpenCL-CXX
- Compiles to SPIR or OpenCL SPIR-V
  - Can distribute the SPIR/SPIR-V kernels and run in OpenCL 2.2
**SYCL** Design Philosophy (30000 ft view)

1. Single source, pure modern C++;
3. SYCL Next should aim to maintain backwards compatibility with SYCL 1.2.1 as much as possible. Future SYCL releases should strive for backwards compatibility;
4. The SYCL specification should aim to align as much as possible with the direction of the ISO C++ standard with the aim that the two languages continue to work well together;
5. SYCL will not create C++ language extensions, but instead add features via C++ library, even if the underlying implementation requires compiler support;
6. The SYCL group will look at the development of the ISO C++ standard and work on how to enable these capabilities in SYCL Next, and bring any feature requests to the OpenCL and SPIR groups;
7. The SYCL group will proactively engage with C++ open-source projects, the ISO C++ standards community, and the advisory panel to ensure that SYCL supports the widest possible ecosystem of C++ software;
8. Lead the effort for heterogeneous compute for modern C++ while integrating with ISO C++ on all ranges of device.
Ghost of SYCL Future (May Change)

- OpenCL 1.2
- OpenCL C Kernel Language
- 2011
- OpenCL

- SYCL 1.2
- C++11 Single source programming
- 2015
- OpenCL

- SYCL 1.2.1
- C++11 Single source programming
- 2017
- OpenCL

- SYCL 2019
- C++17 Single source programming
- 2019
- OpenCL

- SYCL 2021
- C++20 Single source programming
- 2021
- OpenCL

- SPIR
- OpenCL 2.1
- SPIR-V in Core
- 2015

- SPIR
- OpenCL 2.2
- C++ Kernel Language
- 2017
How to port existing C++ to SYCL
CUDA to SYCL

SYCL version

```cpp
#include <CL/sycl.hpp>
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(cl::sycl::nd_item<1> item, int n, float *x, float *y)
{
    int index = item.get_local(0);
    int stride = item.get_local_range(0);
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

c1::sycl::buffer<float> X(N);
c1::sycl::buffer<float> Y(N);

gpu_queue.submit([&](cl::sycl::handler &cgh) {
    auto aX = x.get_access<cl::sycl::access::mode::read>(cgh);
    auto aY = y.get_access<cl::sycl::access::mode::read_write>(cgh);
    cgh.parallel_for<class add_functor>(
        cl::sycl::nd_range<1>(cl::sycl::range<1>(256),
            cl::sycl::range<1>(256)),
        [=](cl::sycl::nd_item<1> it) {
            add(it, N, aX.get_pointer(), aY.get_pointer());
        });
});
```

CUDA version

```cpp
#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__ void add(int n, float *x, float *y)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

add <<<1, 256 >>>(N, x, y);
cudaDeviceSynchronize();
```
CUDA to SYCL: Combined Source File

```cpp
#include <CL/sycl.hpp>
#include <iostream>
#include <math.h>
#include <parallelize.h>

parallelize([&] () {
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
});
```

```c
// function to add the elements of two arrays
#ifdef __SYCL__
void add(cl::sycl::nd_item<1> item, int n, float *x, float *y)
{
    int index = item.get_local(0);
    int stride = item.get_local_range(0);
}
#else
__global__ void add(int n, float *x, float *y)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
}
#endif

// CUDA to SYCL: Combined Source File
#ifdef __SYCL__
    cl::sycl::buffer<float> x(N);
    cl::sycl::buffer<float> y(N);
#else
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));
#endif

#ifdef __SYCL__
    gpu_queue.submit([&](cl::sycl::handler &cgh) {
        auto aX = x.get_access<cl::sycl::access::mode::read>(cgh);
        auto aY = y.get_access<cl::sycl::access::mode::read_write>(cgh);
        cgh.parallel_for<class add_functor>(
            cl::sycl::nd_range<1>(cl::sycl::range<1>(256),
                                cl::sycl::range<1>(256)),
            [=](cl::sycl::nd_item<1> it) {
                add(it, N, aX.get_pointer(), aY.get_pointer());
            });
    });
#else
    add <<<1, 256 >>>(N, x, y);
#endif
```
CUDA to SYCL: Modular Source Files

- Separation of device-specific details into a header file
- Core AI code in source files without device-specific code
- “Separation of Concerns”
- Standard C++ software engineering approach

```cpp
#include <CL/sycl.hpp>
#include <iostream>
#include <math.h>

#include <parallelize.h>

#pragma omp parallel for
for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
```
The SYCL ecosystem

- SYCL-BLAS: git://github.com/codeplaysoftware/sycl-blas.git
- SYCL-ML: git://github.com/codeplaysoftware/SYCL-ML.git
- VisionCpp: git://github.com/codeplaysoftware/visioncpp.git
- Eigen: https://bitbucket.org/benoitsteiner/opencl
- C++ 17 Parallel STL: git://github.com/KhronosGroup/SyclParallelSTL.git
How to get great performance
Performance of AI on GPUs: kernel fusion

Getting good performance on GPUs requires kernel fusion

• Hand-coding can never beat kernel fusion for complex algorithms
• Kernel fusion fuses kernels together, reducing off-chip memory bandwidth
In this example, we perform 3 image processing operations on an accelerator and compare 3 systems when executing individual nodes, or combined nodes.

The system is an AMD APU and the operations are: RGB->HSV, channel masking, HSV->RGB.

Hand-optimized kernels, but no kernel fusion

Runtime-generated kernels, with kernel-fusion

C++ kernels, with kernel-fusion

Effect of combining graph nodes on performance

Time: Lower is faster

Performance with and without kernel fusion
// the node which gets the input data from OpenCV
auto in = visioncpp::terminal<visioncpp::pixel::U8C1, COLS, ROWS,
    visioncpp::memory_type::Buffer2D>(input.data());

// the node which gets the output data
auto out = visioncpp::terminal<visioncpp::pixel::U8C1, COLS, ROWS,
    visioncpp::memory_type::Buffer2D>(output.get());

// convert to Float
auto fin = visioncpp::point_operation<visioncpp::OP_U8C1ToFloat>(in);

// apply mean filter to smooth the image
auto mean_filter = visioncpp::terminal<float, filter_size, filter_size,
    visioncpp::memory_type::Buffer2D,
    visioncpp::scope::Constant>(mean_array);
auto mean = visioncpp::neighbour_operation<visioncpp::OP_Filter2D_One>(fin, mean_filter);

// applying sobel_x filter
auto x_filter = visioncpp::terminal<float, 3, 3, visioncpp::memory_type::Buffer2D,
    visioncpp::scope::Constant>(sobel_x);
auto sobel_x = visioncpp::neighbour_operation<visioncpp::OP_Filter2D_One>(mean, x_filter);

auto y_filter = visioncpp::terminal<float, 3, 3, visioncpp::memory_type::Buffer2D,
    visioncpp::scope::Constant>(sobel_y);
auto sobel_y = visioncpp::neighbour_operation<visioncpp::OP_Filter2D_One>(mean, y_filter);

auto intensity = visioncpp::point_operation<visioncpp::OP_Magnitude>(sobel_x, sobel_y);

// convert to uchar
auto uintensity = visioncpp::point_operation<visioncpp::OP_FloatToU8C1>(intensity);

// assign operation
auto pipe = visioncpp::assign(out, uintensity);

visioncpp::execute<visioncpp::policy::Fuse, 8, 8, 8>(pipe, dev);
// functor that calculates the magnitude of each pixel based on Sobel
struct OP_Magnitude {
    template <typename T1, typename T2>
    float operator()(const T1& t1, const T2& t2) {
        return cl::sycl::clamp(cl::sycl::sqrt(t1 * t1 + t2 * t2), 0.0f, 1.0f);
    }
};

#define rgb2gray(color) (color)(x, y, 0, 0) = image(x, y, 0, 0);
#define grayValue (int)(0.299 * R + 0.587 * G + 0.114 * B);
#define gray(x, y, 0, 0) grayValue;

VisionCpp: Defining custom operations

Running on R-Car V3M on IMP CVengine using OpenCL, SYCL and VisionCpp
Adding acceleration to ISO C++
C++
Directions
Group: P0939

Direction for ISO C++

B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong

Revision History
- This is the initial version.

Main sections
- History
- Long-term Aims (decades)
- Medium-term Aims (3-10 years)
- Priorities for C++20
- Process Issues
- The C++ Programmer’s Bill of Rights
What is C++?

C++ is a language for defining and using lightweight abstractions.

C++ supports building resource constrained applications and software infrastructure.

C++ supports large-scale software development.
How do we want C++ to develop?

*Improve support for large-scale dependable software*

*Improve support for high-level concurrency models*

*Simplify language use*

*Address major sources of dissatisfaction*

*Address major sources of error*
C++ rests on two pillars

A direct map to hardware (initially from C)

Zero-overhead abstraction in production code (initially from Simula, where it wasn’t zero-overhead)
4.3 Concrete Suggestions

- Pattern matching
- Exception and error returns
- Static reflection
- Modern networking

**Modern hardware:**

We need better support for modern hardware, such as executors/execution context, affinity support in C++ leading to heterogeneous/distributed computing support, SIMD/task blocks, more concurrency data structures, improved atomics/memory model/lock-free data structures support. The challenge is to turn this (incomplete) laundry list into a coherent set of facilities and to introduce them in a manner that leaves each new standard with a coherent subset of our ideal.

- Simple graphics and interaction
- Anything from the Priorities for C++20 that didn’t make C++20
**Use the Proper Abstraction today with C++17**

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>How is it supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>C++11/14/17 threads, async</td>
</tr>
<tr>
<td>HW threads</td>
<td>C++11/14/17 threads, async</td>
</tr>
<tr>
<td>Vectors</td>
<td>Parallelism TS2</td>
</tr>
<tr>
<td>Atomic, Fences, lockfree, futures, counters, transactions</td>
<td>C++11/14/17 atomics, Concurrent TS1, Transactional Memory TS1</td>
</tr>
<tr>
<td>Parallel Loops</td>
<td>Async, TBB:parallel_invoke, Parallelism TS1, C++17 parallel algorithms</td>
</tr>
<tr>
<td>Heterogeneous offload, fpga</td>
<td>OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja</td>
</tr>
<tr>
<td>Distributed</td>
<td>HPX, MPI, UPC++</td>
</tr>
<tr>
<td>Caches</td>
<td>Not supported</td>
</tr>
<tr>
<td>Numa</td>
<td>Not supported</td>
</tr>
<tr>
<td>TLS</td>
<td>C++11 TLS (thread_local)</td>
</tr>
</tbody>
</table>
C++ Std Timeline/status
https://isocpp.org/std/status


2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | IS: trunk

Library TR1
C++0x -> 11
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS

TSees: feature branches for beta release & then merge


File System
Lib Fundamentals 1
Parallelism 1
Concepts
Ranges
Concurrency 1

Networking
Lib Fundamentals 2
Parallelism 2
Modules
Coroutines

Reflection
Lib Fundamentals 3
Executors
Concurrency 2
2D Graphics

TS bars start and end where work on detailed specification wording starts (“adopt initial working draft”) and ends (“send to publication”). Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change.

Arrays (abandoned)
Roadmap to safety
Codeplay: Starting the Safety Critical Journey

2016
- Exploration and research

2017
2017 September
- Partnership with Renesas
- Codeplay joins MISRA C++

2018 Q2
- Start ISO26262 certification
- Development of process and Work Plan

2019-2020
- Development of C++ SC product/stack

2017 Q1
- Requirements Analysis
- Investment opportunities

2019-2020
- Development of C++ SC product/stack
Why SYCL?

• Survey on our automotive customers reveals C++ is the preferred development language

• Semiconductor and hardware companies typically offer C++ low-level libraries, lots of vendor-specific extensions

• However, application developers write increasingly complex applications, e.g. Vision and Machine Learning

• Only exception: NVIDIA, offering complete proprietary solutions

• SYCL and its ecosystem is the only open-standard C++ programming model alternative
Why Safe SYCL

• Automotive customers want to use same programming interface across all their platforms
• Current SYCL can be used for prototyping and non-safe development
• Deployment on cars requires a safe product
• But cannot make safe product without a safe language!
SYCL vs Safety

• Currently, MISRA 2008 specification and AUTOSAR rules are required for any safety critical development
• Both are based on ancient C++ rules and use cases
• AUTOSAR updated guidelines will cover C++11 and C++14
  • But heavily based on MISRA 2008
• Codeplay taking two simultaneous paths:
  • Will work on current SYCL vs MISRA 2008 guidelines
  • Joined MISRA committee to push forward new C++ guidelines for automotive that enable modern C++ and SYCL programming on safe systems
What do we need from the SYCL group

MISRA validation effort
- Codeplay will work on sub-setting/validation of SYCL over the next couple of months
- Anyone interested in participating?
- Is the group interested in periodic reports of progress?

SYCL specification "purification"
- We need a clear SYCL feature list to define later a safe-subset of features
- We need to clarify all the undefined behavior that involves the safe-subset
- This is good work for the specification anyway!

MISRA/C++ standardization effort
- Codeplay is working with MISRA to bring modern C++ into safety standards
- Codeplay is also following and contributing to C++ and AUTOSAR efforts
- Any collaboration from the group is appreciated
How to get SYCL

- ComputeCpp: community edition free in Codeplay website
- triSYCL: open-source SYCL
- Ecosystem: sycl.tech
Q&A