SYCL in the OpenVX ecosystem

Andrew Richards, Codeplay
Embedded Vision Summit, May 2017
Over 100 members worldwide
Any company is welcome to join
Who am I?

- Chair of the SYCL group
- Chair of HSA Software Group
- CEO of Codeplay
  - We build a C/C++ compiler for GPUs in 2002
  - 60 staff in Edinburgh, Scotland
  - We build programming tools for heterogeneous processors
  - OpenCL, SYCL, + others
How does SYCL fit into OpenVX, vision & AI?

1. You need a graph system for AI/vision
   ✓ OpenVX

2. You need hand-coded kernels for common tasks
   ✓ OpenVX

3. You need to be able to write custom operations
   ➢ SYCL
What is SYCL for?

- Modern C++ lets us separate the **what** from the **how**:
  - We want to separate **what** the user wants to do: science, computer vision, AI ...
  - And enable the **how** to be: *run fast on an OpenCL device*

- Modern C++ supports and encourages this separation
What we want to achieve

- We want to enable a C++ ecosystem for OpenCL:
  - Must run on OpenCL devices: GPUs, CPUs, FPGAs, DSPs etc
  - C++ template libraries
  - Tools: compilers, debuggers, IDEs, optimizers
  - Training, example programs
  - Long-term support for current and future OpenCL features
Why a new standard?

- There are already very established ways to map C++ to parallel processors
  - So we follow the established approaches

- There are specifics to do with OpenCL we need to map to C++
  - We have worked hard to be an *enabler* for other C++ parallel standards

- We add no more than we need to
Where does SYCL fit in?
OpenCL / SYCL Stack

- User application code
- C++ template libraries
- SYCL for OpenCL
- SPIR
- OpenCL Devices
- CPU
- GPU
- FPGA
- DSP
Philosophy

- With SYCL, we wanted to align with the direction the C++ standard is going
  - And we also need to future-proof for future OpenCL device capabilities

- Key decisions:
  - We will not add any language extensions to C++
  - We will work with existing C++ compilers
  - We will provide the full OpenCL feature-set in C++
  - Everything must compile and run on the host as well as an OpenCL device
Where does SYCL fit in? - Language style

**C++ Embedded DSLs**
- e.g.: Sh/RapidMind, Halide, Boost.compute
- Pros: Works with existing C++ compilers
- Cons: Compile-time compilation, control-flow, composability

**C++ Kernel languages**
- e.g.: GLSL, OpenCL C and C++ kernel languages
- Pros: Explicit offload, independent host/device code & compilers, run-time adaptation, popular in graphics
- Cons: Hard to compose cross-device

**C++ single-source**
- e.g.: SYCL, CUDA, OpenMP, C++ AMP
- Pros: Composability, easy to use, offline compilation and validation
- Cons: Host/device compiler conflict

---

Vector<float> a, b;
auto expr = a + b;
Vector<float> r = expr.eval();

Kernel myKernel;
myKernel.load ("myKernel");
myKernel.compile ();
myKernel.setArg (0, a);
float r = myKernel.run ();

void myKernel (float *arg) {
    return arg * 456.7f;
}

Vector<float> a, b, r;
parallel_for (a.range (), [&] (int id) {
    r [id] = a [id] + b [id];
});

---

C++ template library uses overloading to build up expression tree to compile at runtime

Host (CPU) code loads and compiles kernel for specific device, sets args and runs

Single source file contains code for host & device
Comparison of SYCL & OpenVX

- SYCL is a general programming model
- OpenVX is a vision graph system
- SYCL makes you write your own graph system
- OpenVX distributes a graph across an entire system
- SYCL makes you write your own nodes
- OpenVX uses built-in nodes

In AI applications, we see:

- People needing pre-optimized graph nodes
- People need to optimize whole graphs
- Developers/researchers need to write their own nodes
Comparison of SYCL & CUDA

```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,
    cl::sycl::global_ptr<float> x, cl::sycl::global_ptr<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];
}

// encapsulate data in SYCL buffers
cl::sycl::buffer<float> x(N);
cl::sycl::buffer<float> y(N);

// create a scope to define the lifetime of the SYCL objects
// create a SYCL queue for a GPU
cl::sycl::gpu_selector selectgpu;
cl::sycl::device gpu_device(selectgpu);
cl::sycl::queue gpu_queue(gpu_device);

// submit this work to the SYCL queue
gpu_queue.submit([&](cl::sycl::handler &cgh) {
    // request access to the data on the OpenCL GPU
    auto aX = x.get_access<cl::sycl::access::mode::read>(cgh);
    auto aY = y.get_access<cl::sycl::access::mode::read_write>(cgh);
    // Run kernel on 1M elements on the OpenCL GPU
    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, aY);
        });
});
```

```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];
}

// Allocate Unified Memory - accessible from CPU or GPU
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

// Run kernel on 1M elements on the GPU
add <<<1, 256 >>>(N, x, y);

// Wait for GPU to finish before accessing on host
cudaDeviceSynchronize();
```
Why use C++ single-source programming?

- Widely used, especially in AI
- Runs on lots of platforms
- Kernel fusion
- Abstractions to enable performance-portability
- Composability
- Integrates with: OpenCL, OpenVX etc
Kernel fusion

Most parallel processors are bandwidth bound

\[ a = b \times c + d \times f \]

if \(a\), \(b\), \(c\), \(d\) and \(f\) are vectors, and:

if we execute operations separately, bandwidth-bound, but:

if we fuse into just one kernel, perf is much better
Graph programming: some numbers

In this example, we perform 3 image processing operations on an accelerator and compare 3 systems when executing individual nodes, or a whole graph.

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

Halide and SYCL use kernel fusion, whereas OpenCV does not. For all 3 systems, the performance of the whole graph is significantly better than individual nodes executed on their own.
VisionCpp with SYCL (or OpenMP)

This graph is created in C++ at compile time, so can be optimized at compile time. This allows fast start up.

Source on github
Expressing the execution for a device

C++/OpenMP

```
1: template <typename Expr, typename... Acc>
void sycl (handler& cgh, Expr expr, Acc... acc) {

    // sycl accessor for accessing data on device
    auto outPtr = expr.out->template get_accessor<write>(cgh);

    // sycl range representing valid range of accessing data
    auto rng = range<2>(Expr::Rows, Expr::Cols);

    // sycl parallel for for parallelising execution across the range
    cgh.parallel_for<Type>(rng, [=](item<2> itemID) {

        // rebuilding accessor tuple on the device
        auto tuple = make_tuple(acc);

        // calling the eval function for each pixel
        outPtr[itemID] = expr.eval(itemID, tuple);

    });
}
```

```
1: template <typename Expr, typename... Acc>
void cpp (Expr expr, Acc... acc) {

    // output pointer for accessing data on host
    auto outPtr = expr.out->get();

    // valid range for accessing data on host
    auto rng = range (Expr::Rows, Expr::Cols);

    // rebuilding the tuple of input pointer on host
    auto tuple = make_tuple(acc);

    // OpenMP directive for parallelising for loop
    #pragma omp parallel for
    for(size_t i=0; i<rng.rows; i++)
        for(size_t j=0; j<rng.cols; j++)
            outPtr[indx] = expr.eval(index (i, j), tuple);
}
```
TensorFlow for OpenCL and SYCL

- Same source code supports CUDA and SYCL
  - via #ifdefs

- In branches and trunk, being merged into trunk

- Supported, continuously tested
Applying fusion to TensorFlow Eigen

This is how TensorFlow uses Eigen to achieve kernel-fusion.

CUDA does this for NVIDIA GPUs, SYCL is used here for AMD GPUs.

Performance improvement at size 4,000

Unfused performance improvement: AMD GPU vs multi-core Intel CPU

Total performance improvement delivered by SYCL is both of these graphs combined.
Continuous Integration Testing
C++ Expression trees: Eigen Tensors

This creates an Eigen device

This code is the standard Eigen approach to linear algebra, which TensorFlow has adapted to tensors and accelerated with CUDA.

This is the code adapted to use SYCL

This expression is fused into a single kernel on the device

```c++
cl::sycl::gpu_selector s; // use OpenCL GPU
cl::sycl::queue q(s);
Eigen::SyclDevice sycl_device(q);

array<int, 3> tensorRange = {{100, 10, 20}};
Tensor<DataType, 3, DataLayout> in1(tensorRange);
Tensor<DataType, 3, DataLayout> in2(tensorRange);
Tensor<DataType, 3, DataLayout> in3(tensorRange);
Tensor<DataType, 3, DataLayout> out(tensorRange);

in2 = in2.random();
in3 = in3.random();

DataType *gpu_in1_data  = static_cast<DataType*>(sycl_device.allocate(in1.size()*sizeof(DataType)));
DataType *gpu_in2_data  = static_cast<DataType*>(sycl_device.allocate(in2.size()*sizeof(DataType)));
DataType *gpu_in3_data  = static_cast<DataType*>(sycl_device.allocate(in3.size()*sizeof(DataType)));
DataType *gpu_out_data = static_cast<DataType*>(sycl_device.allocate(out.size()*sizeof(DataType)));

TensorMap<Tensor<DataType, 3, DataLayout>> gpu_in1(gpu_in1_data, tensorRange);
TensorMap<Tensor<DataType, 3, DataLayout>> gpu_in2(gpu_in2_data, tensorRange);
TensorMap<Tensor<DataType, 3, DataLayout>> gpu_in3(gpu_in3_data, tensorRange);
TensorMap<Tensor<DataType, 3, DataLayout>> gpu_out(gpu_out_data, tensorRange);

//a*3.14f + b*2.7f
gpu_out.device(sycl_device) = gpu_in1 * gpu_in1.constant(3.14f) + gpu_in2 * gpu_in2.constant(2.7f);
sycl_device.memcpyDeviceToHost(out.data(),gpu_out_data,(out.size())*sizeof(DataType));
sycl_device.synchronize();
```
Tensor operation as functor on device

```cpp
template<typename Expr, typename FunctorExpr, typename TupleType >
struct ExecExprFunctorKernel {
  typedef typename internal::createPlaceHolderExpression<Expr>::Type PlaceHolderExpr;
  typedef typename Expr::Index Index;
  FunctorExpr functors;
  TupleType tuple_of_accessors;
  Index range;

  ExecExprFunctorKernel (Index range_, FunctorExpr functors_, TupleType tuple_of_accessors_)
    : functors (functors_), tuple_of_accessors (tuple_of_accessors_), range (range_) {};

  void operator()(cl::sycl::nd_item<1> itemID) {
    typedef typename internal::ConvertToDeviceExpression<Expr>::Type DevExpr;
    auto device_expr = internal::createDeviceExpression<DevExpr, PlaceHolderExpr>(functors,
                                                                               tuple_of_accessors);
    auto device_evaluator = Eigen::TensorEvaluator<decltype(device_expr.expr) Eigen::DefaultDevice>(device_expr.expr,
                                                                                                          Eigen::DefaultDevice());
    typename DevExpr::Index gId = static_cast<typename DevExpr::Index>(itemID.get_global_linear_id ());
    if (gId < range)
      device_evaluator.evalScalar(gId);
  }
};
```
Enqueue tensor operation to device

Tensor expression is in \textit{Expr} type

- Package up the data references and expression to be evaluated for sending to device
- Add work to queue
- Enqueue data parallel work to device

```cpp
template <typename Expr, typename Dev>
void run (Expr &expr, Dev &dev) {
  Eigen::TensorEvaluator<Expr, Dev> evaluator (expr, dev);
  const bool needs_assign = evaluator.evalSubExprsIfNeeded (NULL);
  if (needs_assign) {
    typedef decltype(internal::extractFunctors(evaluator)) FunctorExpr;
    FunctorExpr functors = internal::extractFunctors(evaluator);
    dev.sycl_queue().submit([&](cl::sycl::handler &cgh) {
      // create a tuple of accessors from Evaluator
      typedef decltype(internal::createTupleOfAccessors<decltype(evaluator)>)(cgh, evaluator)) TupleType;
      TupleType tuple_of_accessors = internal::createTupleOfAccessors<decltype(evaluator)> (cgh, evaluator);
      typename Expr::Index range, GRange, tileSize;
      dev.parallel_for_setup(static_cast<typename Expr::Index> (evaluator.dimensions().TotalSize()) tileSize, range, GRange);
      cgh.parallel_for (cl::sycl::nd_range<1> (cl::sycl::range<1>(GRange),
          cl::sycl::range<1>(tileSize)),
        ExecExprFunctorKernel<Expr,FunctorExpr,TupleType> (range, functors,
            tuple_of_accessors
          ));
    });
  dev.asynchronousExec();
}
evaluator.cleanup();
```
SYCL ecosystem

- SYCL.tech - [http://sycl.tech](http://sycl.tech)
- SYCLBLAS - SYCL BLAS library that supports kernel fusion
- Eigen - used in TensorFlow for custom operations
- TensorFlow
- VisionCpp - demonstration of how to build C++ graphs for vision
- C++ 17 Parallel STL for SYCL - supports the new C++ 17 Parallel STL standard
How do I get SYCL?

- ComputeCpp: From Codeplay (my company)
  - Available for free and works with OpenCL accelerators using SPIR

- triSYCL: open source
  - Doesn’t (yet) work with OpenCL accelerators
What now?

- We are working on supporting OpenCL 2.2 with SYCL 2.2
- We are working on bringing heterogeneous acceleration into a future ISO C++
- We are building out the open standard ecosystem of C++ accelerated software
Questions ?
Where does SYCL fit in? - Parallelism

- **Directive-based parallelism**
  - e.g.: OpenMP, OpenAcc
  - Pros: Original source code is annotated not modified; well-understood
  - Cons: Hard to compose; execution order separate from source code

- **Thread parallelism**
  - e.g.: tbb, C++11 threads, pthreads
  - Pros: Well understood; works with variety of algorithms
  - Cons: Doesn’t map to highly parallel architectures like GPUs & FPGAS

- **Explicit parallelism**
  - e.g.: SYCL, Parallel STL, CUDA, C++AMP
  - Pros: Composable; works with wide variety of processor architectures
  - Cons: Requires user to know the parallelism

Vector<float> a, b, r;
for (int i=0; i< a.size(); i++)
{
    #pragma parallel_for
    r [i] = a [i] + b [i];
}

Vector<float> a, b, r;
Thread t1 = createThread ([&]() {
    sumFirstHalf (r, a, b);
});
Thread t2 = createThread ([&]() {
    sumSecondHalf (r, a, b);
});
t1.wait (); t2.wait ();

Vector<float> a, b;
parallel_for (a.range (), [&](int id)
{
    a [id] = a [id] + b [id];
});
What features of OpenCL do we need?

• We want to make it easy to write **high-performance** OpenCL code in C++
  - SYCL code in C++ must use memory and execute kernels efficiently
  - We must provide developers with all the optimization options they have in OpenCL

• We want to enable all **OpenCL features** in C++ with SYCL
  - Support wide range of OpenCL devices: CPUs, GPUs, FPGAs, DSPs...
  - Data on host: Images and buffers; mapping, DMA and copying
  - Data on device: global/constant/local/private memory; multiple pointer sizes
  - Parallelism: ND ranges, work-groups, work-items, barriers, queues, events
  - Multi-device: Platforms, devices, contexts

• We want to enable OpenCL C code to **interoperate** with C++ SYCL code
  - Sharing of contexts, memory objects etc
Example SYCL Code

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

void func (float *array_a, float *array_b, float *array_c, float *array_r, size_t count)
{
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));
    queue myQueue (gpu_selector);

    myQueue.submit([&](handler& cgh)
    {
        auto a = buf_a.get_access<access::read>(cgh);
        auto b = buf_b.get_access<access::read>(cgh);
        auto c = buf_c.get_access<access::read>(cgh);
        auto r = buf_r.get_access<access::write>(cgh);

        cgh.parallel_for<class three_way_add>(count, [=](id<1> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
}
```

#include the SYCL header file

Encapsulate data in SYCL **buffers**

which be mapped or copied to or from OpenCL devices

Create a **queue**, preferably on a GPU, which can execute **kernels**

Submit to the queue all the work described in the handler lambda that follows

Create **accessors** which encapsulate the type of access to data in the buffers

Execute in parallel the work over an **ND range** (in this case ‘count’)

This code is executed in parallel on the device
const int n_items = 32;
range<1> r(n_items);

int array_a[n_items] = { 0 };
int array_b[n_items] = { 0 };

buffer<int, 1> buf_a(array_a, range<1>(r));
buffer<int, 1> buf_b(array_b, range<1>(r));

queue q;
q.submit([&](handler& cgh)
{
    auto acc_a = buf_a.get_access<read_write>(cgh);
    algorithm_a s(acc_a);
    cgh.parallel_for(r, s);
});

q.submit([&](handler& cgh)
{
    auto acc_b = buf_b.get_access<read_write>(cgh);
    algorithm_b s(acc_b);
    cgh.parallel_for(r, s);
});

q.submit([&](handler& cgh)
{
    auto acc_a = buf_a.get_access<read_write>(cgh);
    algorithm_c s(acc_a);
    cgh.parallel_for(r, s);
});
Data access with **accessors**

- Encapsulates the difference between data storage and data access
- Allows creation of a parallel task graph with schedule, synchronization and data movement
- Enables devices to use optimal access to data
  - Including having different pointer sizes on the device to those on the host
  - Allows usage of different address spaces for different data
- Enhanced with call-graph-duplication (for C++ pointers) and explicit pointer classes
  - To enable direct pointer-like access to data on the device
- Portable, because accessors can be implemented as raw pointers
‘Shared source’ approach to single-source

- This is not *required* for SYCL, but is designed as a possible implementation
- Have a different compiler for host and each device
  - Don’t really need to implement different front-end for each device, but can
- **Benefits**
  - Many developers are required to use a specific host compiler
  - Allows front-ends to optimize for specific devices: e.g. CPU, FPGA, GPU, DSP
  - Allows the pre-processor to be used by developers for portability and performance portability
# Example SYCL Code: Building the program

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

int main ()
{
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));

    queue myQueue (gpu_selector);

    myQueue.submit([&](handler& cgh)
    {
        auto a = buf_a.get_access<access::read>(cgh);
        auto b = buf_b.get_access<access::read>(cgh);
        auto c = buf_c.get_access<access::read>(cgh);
        auto r = buf_r.get_access<access::write>(cgh);

        cgh.parallel_for<class three_way_add>(count, [=](id<> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
}
```

On host, the accessors can represent the dependencies in the program. On device, they can be implemented as OpenCL pointers (whether global, local or constant).

This code is extracted by a device compiler and compiled for a device, including any functions or methods called from here. All the code must conform to the OpenCL kernel restrictions (e.g. no recursion). This code can be compiled for different devices from the same source code.

This is the name of the lambda, which is used to enable the host to load the correct compiled device kernel into OpenCL. C++ reflection may remove this requirement.
## Where does SYCL fit in? - Memory model

<table>
<thead>
<tr>
<th><strong>Cache coherent single-address space</strong></th>
<th><strong>Non-coherent single-address space</strong></th>
<th><strong>Multi-address space</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>e.g.</strong>: Multi-core CPUs, HSA, OpenCL 2 System Sharing</td>
<td><strong>e.g.</strong>: HSA Coarse grained, OpenCL 2.x</td>
<td><strong>e.g.:</strong> SYCL 1.2, C++ AMP, OpenCL 1.x</td>
</tr>
<tr>
<td><strong>Pros</strong>: Very easy to program – just pass around pointers (leave ownership issues to user); low-latency offload; very little impact on programming model</td>
<td><strong>Pros</strong>: Doesn’t require (much) OS support or (much) hardware support</td>
<td><strong>Pros</strong>: High performance and efficiency of memory accesses; wide device support</td>
</tr>
<tr>
<td><strong>Cons</strong>: Bandwidth limited; costs power; needs special operating system support</td>
<td><strong>Cons</strong>: Not supported on all processor cores; User must manage ownership</td>
<td><strong>Cons</strong>: Impact on programming model (pointers)</td>
</tr>
</tbody>
</table>

### Cache coherent single-address space

float *a = new float [size];  
processCodeOnDevice (a, size);  

When parallelizing on a system with a cache-coherent single address space, only need to pass around pointers. This makes communication and offloading very low-cost and easy. Requires all memory accesses to go through virtual memory system and caches communicate ownership across all cores.

### Non-coherent single-address space

float *a = NewShared<float> (size);  
a.passOwnershipToDevice (size);  
processCodeOnDevice (a, size);  

All data is still referred to via shared pointers, but the user must manage the memory ownership between different cores.

### Multi-address space

Shared<float> a (size);  
processCodeOnDevice (a);  

Data needs to be encapsulated in new datatypes that are able to manage ownership between host CPU and different devices.