Khronos Group SYCL standard
triSYCL Open Source Implementation
Ronan Keryell
Xilinx Research Labs
SC 2016
C++

- Direct mapping to hardware
- Zero-overhead abstraction
Even better with modern C++ (C++14, C++17)

- Huge library improvements
- Simpler syntax
  ```cpp
  std::vector<int> my_vector { 1, 2, 3, 4, 5 }
  for (auto &e : my_vector)
      e += 1;
  ```
- Automatic type inference for terse generic programming
  - Python 3.x (interpreted):
    ```python
    def add(x, y):
        return x + y
    print(add(2, 3))  # 5
    print(add("2", "3"))  # 23
    print(add(2, "Boom"))  # Fails at run-time :-(
    ```
  - Same in C++14 but compiled + static compile-time type-checking:
    ```cpp
    auto add = [] (auto x, auto y) { return x + y; };
    std::cout << add(2, 3) << std::endl;  // 5
    std::cout << add("2", "3") << std::endl;  // 23
    std::cout << add(2, "Boom") << std::endl;  // Does not compile :-(
    ```

Without using templated code!
Power wall: the final frontier...

- **Current physical limits**
  - Power consumption
    - Cannot power-on all the transistors without melting (*dark silicon*)
    - Accessing memory consumes orders of magnitude more energy than a simple computation
    - Moving data inside a chip costs quite more than a computation
  - Speed of light
    - Accessing memory takes the time of $10^4$+ CPU instructions
    - Even moving data across the chip (cache) is slow at 1+ GHz...

- **Specialize architecture**
  - Use locality & hierarchy
  - Massive parallelism
  - NUMA & distributed memories
  - Power-on only what is required
  - Use hardware reconfiguration
Xilinx Zynq UltraScale+ MPSoC Overview

### Processing System (PS)
- **Application Processing Unit**
  - ARM® Cortex™-A53
  - NEON™ Floating Point Unit
  - Memory Management Unit
  - Embedded Trace Macrocell
- **Graphics Processing Unit**
  - ARM Mali™-400 MP
  - Geometry Processor
  - Pixel Processor
  - DDR Controller
  - DDR4/3/3L, LPDDR4/3 ECC Support
  - 256 KB OCM with ECC
- **Real-Time Processing Unit**
  - ARM Cortex™-R5
  - Vector Floating Point Unit
  - Memory Protection Unit
  - TrustZone
  - Voltage/Temp Monitor
- **Security**
  - Config AES, Decryption, Authentication, Secure Boot

### Programmable Logic (PL)
- **Storage & Signal Processing**
  - Block RAM
  - UltraRAM
  - DSP
- **General-purpose I/O**
  - High-Performance HPIO
  - High Density HDIO
- **High-Speed Connectivity**
  - GTH
  - 100G EMAC
  - PCle Gen4
  - AMS

---

Khronos Group SYCL standard
What about C++ for heterogenous computing???

- C++ std::thread is great...
- ...but supposed shared unified memory (SMP)
  - What if accelerator with own separate memory? Not same address space?
  - What if using distributed memory multi-processor system (MPI...)?
- ♻️ Extend the concepts...
  - Replace raw unified-memory with buffer objects
  - Define with accessor objects which/how buffers are used
  - Since accessors are already here to define dependencies, no longer need for std::future/std::promise!
  - Add concept of queue to express where to run the task
  - Also add all goodies for massively parallel accelerators (OpenCL/Vulkan/SPIR-V) in clean C++
Complete example of matrix addition in OpenCL SYCL

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

using namespace cl::sycl;

constexpr size_t N = 2;
constexpr size_t M = 3;
using Matrix = float[N][M];

// Compute sum of matrices a and b into c
int main() {
    Matrix a = { { 1, 2, 3 }, { 4, 5, 6 } };,
    Matrix b = { { 2, 3, 4 }, { 5, 6, 7 } };,
    Matrix c;

    // Create a queue to work on default device
    queue q;
    // Wrap some buffers around our data
    buffer A { &a[0][0], range { N, M } };
    buffer B { &b[0][0], range { N, M } };
    buffer C { &c[0][0], range { N, M } };
    // Enqueue some computation kernel task
    q.submit([&](handler& cgh) {
        // Define the data used/produced
        auto ka = A.get_access<access::mode::read>(cgh);
        auto kb = B.get_access<access::mode::read>(cgh);
        auto kc = C.get_access<access::mode::write>(cgh);
        // Create & call kernel named "mat_add"
        cgh.parallel_for<class mat_add>(range { N, M },
            [=](id<2> i) { kc[i] = ka[i] + kb[i]; });
    }); // End of our commands for this queue

    // Copy back the buffer data with RAII behaviour.
    std::cout << "c[0][2] = " << c[0][2] << std::endl;
    return 0;
}
```

Khronos Group SYCL standard
Asynchronous task graph model

- Change example with initialization kernels instead of host?
- Theoretical graph of an application described *implicitly* with kernel tasks using buffers through accessors

```
cl::sycl::buffer a
cl::sycl::buffer b
matrix_add cl::sycl::buffer c
```

- Possible schedule by SYCL runtime:

```
init_b init_a matrix_add
```

Automatic overlap of kernels & communications

- Even better when looping around in an application
- Assume it will be translated into pure OpenCL event graph
- Runtime uses as many threads & OpenCL queues as necessary (GPU synchronous queues, AMD compute rings, AMD DMA rings...)

Khronos Group SYCL standard
Task graph programming — the code

```cpp
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;

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

int main() {
    // By sticking all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block
    queue q;
    // Create some 2D buffers of float for our matrices
    buffer<double, 2> a({N, M});
    buffer<double, 2> b({N, M});
    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
        auto A = a.get_access<access::mode::write>(cgh);
        // Enqueue parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class init_a>({N, M}, [=](auto 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
        auto B = b.get_access<access::mode::write>(cgh);
        // 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},
                                            [=](auto 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
        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);
        // 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},
                                            [=](auto index) {
                                            });
    });
    // Request an access to read c from the host-side. The SYCL runtime
    // ensures that c is ready when the accessor is returned */
    auto C = c.get_access<access::mode::read, access::target::host_buffer>
    for (size_t i = 0; i < N; i++)
        for (size_t j = 0; j < M; j++)
            if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
                std::cout << "Wrong value" << C[i][j] << "on_element" << std::endl;
            }
    std::cout << "Good computation!" << std::endl;
    return 0;
}
```

---

*Khronos Group SYCL standard*
Remember the OpenCL execution model?
// Launch a 1D convolution filter
my_queue.submit([&](handler &cgh) {
    auto in_access = inputB.get_access<access::mode::read>(cgh);
    auto filter_access = filterB.get_access<access::mode::read>(cgh);
    auto out_access = outputB.get_access<access::mode::write>(cgh);
    // Iterate on all the work-group
    cgh.parallel_for_work_group<class convolution>({ size, groupsize },
    [=](group<> group) {
        std::cout << "Group id = " << group.get(0) << std::endl;
        // These are OpenCL local variables used as a cache
        float filterLocal[2*radius + 1];
        float localData[blocksize + 2*radius];
        float convolutionResult[blocksize];
        range<1> filterRange { 2*radius + 1 };
        // Iterate on filterRange work-items
        group.parallel_for_work_item(filterRange, [&] (item<1> tile) {
            filterLocal[tile] = filter_access[tile];
        });
        // There is an implicit barrier here
        range<1> inputRange { blocksize + 2*radius };
        // Iterate on inputRange work-items
        group.parallel_for_work_item(inputRange, [&] (item<1> tile) {
            float val = 0.0f;
            int readAddress = group*blocksize + tile - radius;
            if (readAddress >= 0 && readAddress < size)
                val = in_access[readAddress];
            localData[tile] = val;
        });
        // There is an implicit barrier here
        // Iterate on all the work-items
        group.parallel_for_work_item({size, groupsize},
        {
            float sum = 0.0f;
            for (unsigned offset = 0; offset < radius; ++offset)
                sum += filterLocal[offset]*localData[tile + offset + radius];
            float result = sum/(2*radius + 1);
            convolutionResult[tile] = result;
        });
        // There is an implicit barrier here
        // Iterate on all the work-items
        group.parallel_for_work_item({size, groupsize},
        {
            out_access[group*blocksize + tile] = convolutionResult[tile];
        });
        // There is an implicit barrier here
    });});
From work-groups & work-items to hierarchical parallelism (II)

Very close to OpenMP 4 style! 😊

- Easy to understand the concept of work-groups
- Easy to write work-group only code
- Replace code + barriers with several `parallel_for_work_item()`
  - Performance-portable between CPU and device
  - No need to think about barriers (automatically deduced)
  - Easier to compose components & algorithms
  - Ready for future device with non uniform work-group size
Pipes in OpenCL 2.x

- Simple FIFO objects
- Useful to create dataflow architectures between kernels without host
- Created on the host with some message size + object number
- `read()`/`write()` functions
- Useful on FPGA: avoid global memory transfers!
# include <CL/sycl.hpp>
# include <iostream>
# include <iterator>

using namespace cl::sycl;
constexpr size_t N = 3;
using Vector = float[N];

int main() {
    Vector va = { 1, 2, 3 };  
    Vector vb = { 5, 6, 8 };  
    Vector vc;

    {
        // Create buffers from a & b vectors
        buffer<float> ba { std::begin(va), std::end(va) };  
        buffer<float> bb { std::begin(vb), std::end(vb) };  

        // A buffer of N float using the storage of vc
        buffer<float> bc { vc, N };  

        // A pipe of 2 float elements
        pipe<float> p { 2 };  

        // Create a queue to launch the kernels
        queue q;

        // Launch the producer to stream A to the pipe
        q.submit([&](handler &cgh) {
            // Get write access to the pipe
            auto kp = p.get_access<access::mode::write, access::target::blocking_pipe>(cgh);
            
            // Get read access to the data
            auto ka = ba.get_access<access::mode::read>(cgh);
            
            cgh.single_task<class producer>[=] ()
                {
                    for (int i = 0; i != N; i++)
                        kp << ka[i];
                };
        });

        // Launch the consumer that adds the pipe stream with B to C
        q.submit([&](handler &cgh) {
            // Get read access to the pipe
            auto kp = p.get_access<access::mode::read, access::target::blocking_pipe>(cgh);

            // Get access to the input/output buffers
            auto kb = bb.get_access<access::mode::read>(cgh);
            auto kc = bc.get_access<access::mode::write>(cgh);

            cgh.single_task<class consumer>[=] ()
                {
                    for (int i = 0; i != N; i++)
                        kc[i] = kp.read() + kb[i];
                };
        });

    } // < End scope for the buffers: wait for bc copied back to v

    std::cout << std::endl << " Result: " << std::endl;
    for(auto e : vc)
        std::cout << e << " ";
    std::cout << std::endl;
}

Khronos Group SYCL standard
Motion detection on video in SYCL with blocking static pipes

```cpp
auto window_name = "opencv_test";
cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);

cv::Mat rgb_data_in { NUMROWS, NUMCOLS, CV_8UC4 };
cv::Mat rgb_data_prev { NUMROWS, NUMCOLS, CV_8UC4 };
cv::Mat rgb_data_out { NUMROWS, NUMCOLS, CV_8UC4 };

cv::VideoCapture capture;
capture.open("./optical_flow_input.avi");

cv::Mat frame;
capture.read(frame);

cv::Mat small_frame;
cv::resize(frame, small_frame, rgb_data_in.size());

const int from_to[] = { 0, 0, 1, 1, 2, 2 };
cv::mixChannels(&small_frame, 1, &rgb_data_in, 1, from_to, 3);

cv::imshow(window_name, rgb_data_in);
cv::waitKey(30);

// Create a queue to launch the kernels
cl::sycl::queue q;

int framecnt = 0;
// Processing loop
while (capture.read(frame)) {
    cv::swap(rgb_data_in, rgb_data_prev);
    cv::resize(frame, small_frame, rgb_data_in.size());
    cv::mixChannels(&small_frame, 1, &rgb_data_in, 1, from_to, 3);

    cl::sycl::buffer<int> buf_in { (int *) rgb_data_in.data, NUMROWS*NUMCOLS },
    buf_prev { (int *) rgb_data_prev.data, NUMROWS*NUMCOLS },
    buf_out { (int *) rgb_data_out.data, NUMROWS*NUMCOLS };

    // Send the images to the pipes
    read_data(q, buf_in, buf_prev);

    // Color conversion and sobel on the current image
    rgb_pad2ycbcr_in(q);
sobel_filter_pass(q);

    // Color conversion and sobel on the previous image
    // \todo Unify rgb_pad2ycbcr_in and rgb_pad2ycbcr_prev
    rgb_pad2ycbcr_prev(q);
    // \todo Unify sobel_filter and sobel_filter_pass
    sobel_filter(q);

    // Compare 2 sobel outputs
    diff_image(q);
    combo_image(q, 0);

    // Color conversion and receive image from pipe
    ycbcr2rgb_pad(q);
    write_data(q, buf_out);
}
```

Khronos Group SYCL standard
Interoperability with OpenCL world

```cpp
#include <iostream>
#include <iterator>
#include <boost/compute.hpp>
#include <boost/test/minimal.hpp>
#include <CL/sycl.hpp>

using namespace cl::sycl;

constexpr size_t N = 3;
using Vector = float[N];

int test_main(int argc, char *argv[]) {
    Vector a = {1, 2, 3};
    Vector b = {5, 6, 8};
    Vector c;

    // Construct the queue from the default OpenCL one
    queue q { boost::compute::system::default_queue() };

    // Create buffers from a & b vectors
    buffer<float> A { std::begin(a), std::end(a) };
    buffer<float> B { std::begin(b), std::end(b) };
    
    // A buffer of N float using the storage of c
    buffer<float> C { c, N };

    // Construct an OpenCL program from the source string
    auto program = boost::compute::program::create_with_source(R"(
    __kernel void vector_add(const __global float *a,
    const __global float *b,
    __global float *c) {
        c[get_global_id(0)] = a[get_global_id(0)] + b[get_global_id(0)];
    }
    ")".
    
    // Build a kernel from the OpenCL kernel
    program.build();

    // Get the OpenCL kernel
    kernel k { boost::compute::kernel { program, "vector_add" } };

    // Launch the vector parallel addition
    q.submit([&](handler &cgh) {
        // The host–device copies are managed transparently by these
        // accessors:
        cgh.set_args(A.get_access<access::mode::read>(cgh),
                     B.get_access<access::mode::read>(cgh),
                     C.get_access<access::mode::write>(cgh));
        cgh.parallel_for(N, k);
    }); // End of our commands for this queue

    // Buffer C goes out of scope and copies back values to c
    std::cout << std::endl << "Result: " << std::endl;
    for (auto e : c)
        std::cout << e << " ";
    std::cout << std::endl;

    return 0;
}
```

Khronos Group SYCL standard
Known implementations of SYCL

- **ComputeCPP by Codeplay** [https://www.codeplay.com/products/computecpp](https://www.codeplay.com/products/computecpp)
  - Most advanced SYCL 1.2 implementation
  - Outlining compiler generating SPIR
  - Run on any OpenCL device and CPU
  - Google & CodePlay have SYCL version of Eigen & TensorFlow using ComputeCPP

- **sycl-gtx** [https://github.com/ProGTX/sycl-gtx](https://github.com/ProGTX/sycl-gtx)
  - Open source
  - No (outlining) compiler; use some macros with different syntax

- **triSYCL** [https://github.com/Xilinx/triSYCL](https://github.com/Xilinx/triSYCL)
  - Open Source
  - Some extensions (Xilinx blocking pipes)
  - Outlining compiler still in development; no device support yet
triSYCL

- Open Source implementation using templated C++1z classes
  - On-going implementation started at AMD and now led by Xilinx
  - [https://github.com/Xilinx/triSYCL](https://github.com/Xilinx/triSYCL)
  - ≈ 10 contributors

- Used by Khronos committee to define the SYCL & OpenCL C++ standard
  - Languages are now too complex to be defined without implementing...
  - ∃ private Git repositories for future Khronos & experimental Xilinx versions

- Pure C++ implementation & CPU-only implementation for now
  - Use OpenMP for computation on CPU + std::thread for task graph
  - Rely on STL & Boost for zen style
  - CPU emulation for free
    - Quite useful for debugging
  - More focused on correctness than performance for now (array bound check...)

- Provide OpenCL-interoperability mode: can reuse existing OpenCL code

- On-going OpenCL implementation of outlining compiler based on open source Clang/LLVM compiler

---

Khronos Group SYCL standard
Outline

1 SPIR-V

2 Conclusion
Interoperability nightmare in heterogeneous computing & graphics

- Many programming languages for heterogeneous computing
  - Writing compiler front-end may not be *the* real value for a hardware vendor...
  - Writing a C++1z compiler from scratch is almost impossible...
- Many programming languages for writing shaders
- Convergence in computing (Compute Unit) & graphics (Shader) architectures
  - Same front-end & middle-end compiler optimizations
- Need for some non source-readable portable code for IP protection
  - Defining common low-level representation!

Khronos Group SYCL standard
SPIR-V transforms the language ecosystem

- **First multi-API, intermediate language for parallel compute and graphics**
  - Native representation for Vulkan shader and OpenCL kernel source languages

- **Cross-vendor intermediate representation**
  - Language front-ends can easily access multiple hardware run-times
  - Acceleration hardware can leverage multiple language front-ends
  - Encourages tools for program analysis and optimization in SPIR form
### Evolution of SPIR family

<table>
<thead>
<tr>
<th></th>
<th>SPIR 1.2</th>
<th>SPIR 2.0</th>
<th>SPIR-V 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LLVM Interaction</strong></td>
<td>Uses LLVM 3.2</td>
<td>Uses LLVM 3.4</td>
<td>100% Khronos defined Round-trip lossless conversion</td>
</tr>
<tr>
<td><strong>Compute Constructs</strong></td>
<td>Metadata/Intrinsics</td>
<td>Metadata/Intrinsics</td>
<td>Native</td>
</tr>
<tr>
<td><strong>Graphics Constructs</strong></td>
<td>No</td>
<td>No</td>
<td>Native</td>
</tr>
<tr>
<td><strong>Supported Language Feature Sets</strong></td>
<td>OpenCL C 1.2</td>
<td>OpenCL C 1.2 / 2.0</td>
<td>OpenCL C 1.2 / 2.0 and GLSL</td>
</tr>
<tr>
<td><strong>OpenCL Ingestion</strong></td>
<td>OpenCL 1.2 Extension</td>
<td>OpenCL 2.0 Extension</td>
<td>OpenCL 2.1 Core / 2.0 Extensions</td>
</tr>
<tr>
<td><strong>Vulkan Ingestion</strong></td>
<td>-</td>
<td>-</td>
<td>Vulkan 1.0 Core</td>
</tr>
</tbody>
</table>

Not based on LLVM to isolate from LLVM roadmap changes
Driving SPIR-V Open Source ecosystem

SPIR-V
- 32-bit Word Stream
- Extensible and easily parsed
- Retains data object and control flow information for effective code generation and translation

Khronos has open sourced these tools and translators
Khronos plans to open source these tools soon

SPIR-V Tools
- SPIR-V Validator
- SPIR-V (Dis)Assembler

OpenCL C
OpenCL C++

GLSL
Third party kernel and shader Languages

LLVM
LLVM to SPIR-V Bi-directional Translator

SPIR-V Magic # 0x07230203
SPIR-V Version 99
Builder's Magic # 0x051e000B
$id$ bound is 50

OpMemoryModel
Logical
GLSL 450
OpEntryPoint
Fragment shader
function <id> 4

OpTypeVoid
$id$ is 2

OpTypeFunction
$id$ is 3
return type <id> is 2

OpFunction
Result Type <id> is 2
Result <id> is 4

Function Type <id> is 3

Other Intermediate Forms

IHV Driver Runtimes

Khronos Group SYCL standard

SYCL™
Outline

1. SPIR-V

2. Conclusion
Puns and pronunciation explained

OpenCL SYCL

sickle [ˈsi-kəl]

OpenCL SPIR

spear [ˈspɪr]
Ecosystem: OpenCL, CUDA, SYCL, Vulkan, OpenMP, OpenACC... ?

- **OpenCL 2.2 C++ ≈ NVIDIA CUDA Driver API (non single-source)**
  - Low level for full control
  - Standard platform to build higher framework

- **SYCL 2.2 C++ ≿ NVIDIA CUDA Runtime API, OpenMP, OpenACC (single-source)**
  - Single-source higher-level C++ model for OpenCL programming
  - Domain-specific embedded language (DSEL) based on pure C++14 (1.2)/C++17 (2.2)
  - Do not require specific compiler for host code
  - Provide asynchronous task graph

Khronos Group SYCL standard
Conclusion

- In modern C++17 we trust
- SYCL C++ standard from Khronos Group
  - Pure modern C++ DSEL for heterogeneous computing
  - Candidate for ISO C++ WG21 SG14 standard
  - Provide OpenCL interoperability if needed
- triSYCL
  - Open Source on-going implementation
  - Use only pure C++17, OpenMP and Boost for CPU and OpenCL-compatible mode
  - On-going implementation of device compiler with Clang/LLVM
- Other implementations and libraries (Eigen, TensorFlow...) on http://sycl.tech
- SPIR-V extends OpenCL execution model to any language