Modern C++, OpenCL SYCL & OpenCL CL2.hpp

Ronan Keryell

AMD & Khronos OpenCL SYCL committee
Ronan.Keryell at amd dot com

11/18/2014

—

SuperComputing 2014

—

OpenCL BoF

Outline

1. 2 great things in 2014
   - 2014 take 1: C++14
   - 2014 take 2: OpenCL SYCL
2. OpenCL SYCL
   - C++... putting everything altogether
   - Possible future extensions & applications
3. C++ `CL.hpp` OpenCL wrapper
4. Conclusion

Outline

C++14

- 2 Open Source compilers available *before* ratification (GCC & Clang/LLVM)
- Confirm new momentum & pace: 1 major (C++11) and 1 minor (C++14) version on a 6-year cycle
- Next big version expected in 2017 (C++1z)
  - Already being implemented! ☺
- Monolithic committee replaced by many smaller task forces
  - Parallelism TS (Technical Specification)
  - Concurrency TS
  - Definitely matters for HPC and heterogeneous computing!

C++ is a complete new language

- Forget about C++98, C++03...
- Now open and public
- Send your proposals and get involved!

Modern C++ & HPC (I)

- Huge library improvements
  - <thread> library and multithread memory model <atomic> $\leadsto$ HPC
  - Hash-map
  - Algorithms
  - Random numbers
  - ...
- Uniform initialization and range-based for loop

```
std::vector<int> my_vector { 1, 2, 3, 4, 5 };
for (int &e : my_vector)
  e += 1;
```

- Easy functional programming style with *lambda* (anonymous) functions

```
std::transform(std::begin(v), std::end(v), [] (int v) { return 2*v; });
```

Modern C++ & HPC (II)

- R-value references & std :: move semantics
  - matrix_A = matrix_B + matrix_C
    - Avoid copying (TB, PB... ☹) when assigning or function return
  - Without messing up with references or worse... pointers!
  - (Remember Chandler's talk on abstractions & C++ yesterday at the LLVM-HPC workshop?)
- Lot of meta-programming improvements to make meta-programming ~~easy~~ easier: variadic templates, type traits <type_traits>...
- Make simple things simpler to be able to write generic numerical libraries, etc.

Modern C++ & HPC (III)

- Automatic type inference for terse programming
  - Python 3.x:

```python
def add(x, y):
  return x + y

print(add(2, 3))       # 5
print(add("2", "3"))  # 23
```

  - Same in C++14 but also with 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"s, "3"s) << std::endl; // 23
```

Without using templated code! ~~`template <typename >`~~ ☺

- Lot of other amazing stuff...

AMD

Outline

1. 2 great things in 2014
   - 2014 take 1: C++14
   - 2014 take 2: OpenCL SYCL
2. OpenCL SYCL
   - C++... putting everything altogether
   - Possible future extensions & applications
3. C++ `CL.hpp` OpenCL wrapper
4. Conclusion

Announcing SYCL C++ for OpenCL

- Need C++-post-modernism to help OpenCL & accelerator worlds...
- Provisional version 1 at GDC 2014, March 2014
- Provisional version 2 at SC 2014, November 2014 (you are the lucky ones! ☺)
  - Khronos internal discussion are not public...
  - ⇝ Make intermediate versions to help Open Source implementations! ☺

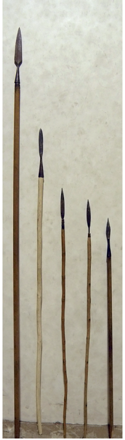
AMD

Puns and pronunciation explained

OpenCL SYCL

sickle ['si-kəl]

OpenCL SPIR



spear ['spir]

Outline

1. 2 great things in 2014
   - 2014 take 1: C++14
   - 2014 take 2: OpenCL SYCL
2. OpenCL SYCL
   - C++... putting everything altogether
   - Possible future extensions & applications
3. C++ `CL.hpp` OpenCL wrapper
4. Conclusion

OpenCL SYCL goals

- Ease of use
  - Single source programming model
    - SYCL source compiled for host *and* device(s)
- Development/debugging on host: *host* fall-back target
- Programming interface based on abstraction of OpenCL components (data management, error handling...)
- Most modern C++ features available for OpenCL
  - Enabling the creation of higher level programming models
  - C++ templated libraries based on OpenCL
- Portability across platforms and compilers
- Providing the full OpenCL feature set and seamless integration with existing OpenCL code
- Task graph programming model
- High performance

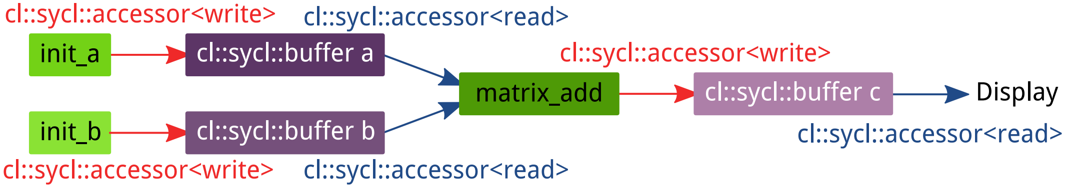
`http://www.khronos.org/opencl/sycl`

Complete example of matrix addition in OpenCL SYCL

```
#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];

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
   queue myQueue;
   // Wrap some buffers around our data
   buffer<float, 2> A { a, range<2> { N, M } };
   buffer<float, 2> B { b, range<2> { N, M } };
   buffer<float, 2> C { c, range<2> { N, M } };
   // Enqueue some computation kernel task
   command_group (myQueue, [&] () {
     // Define the data used/produced
     auto ka = A.get_access<access::read>();
     auto kb = B.get_access<access::read>();
     auto kc = C.get_access<access::write>();
     // Create & call OpenCL kernel named "mat_add"
     parallel_for<class mat_add>(range<2> { N, M },
        [=](id<2> i) { kc[i] = ka[i] + kb[i]; }
     );
   }); // End of our commands for this queue
  } // End scope, so wait for the queue to complete.
   // Copy back the buffer data with RAII behaviour.
  return 0;
}
```

Asynchronous task graph programming

- Theoretical graph of an application described with tasks kernels using buffers through accessors



- Possible schedule by SYCL runtime:

⤳ Automatic overlap of kernels & communications
  - Even better when looping around in an application

Task graph programming — the code

```
#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

    // Create a queue to work on
    queue myQueue;
    // 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
    command_group (myQueue, [&] () {
        // The kernel write a, so get a write accessor on it
        auto A = a.get_access<access::write>();

        // Enqueue parallel kernel on a N*M 2D iteration space
        parallel_for<class init_a>({ N, M },
                              [=] (id<2> index) {
                                A[index] = index[0]*2 + index[1];
                              });
      });
    // Launch an asynchronous kernel to initialize b
    command_group (myQueue, [&] () {
        // The kernel write b, so get a write accessor on it
        auto B = b.get_access<access::write>();
        /* 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
        parallel_for<class init_b>({ N, M },
                              [=] (id<2> index) {
                                B[index] = index[0]*2014 + index[1]*42;
                              });
      });
    // Launch an asynchronous kernel to compute matrix addition c = a + b
    command_group (myQueue, [&] () {
        // In the kernel a and b are read, but c is written
        auto A = a.get_access<access::read>();
        auto B = b.get_access<access::read>();
        auto C = c.get_access<access::write>();
        // 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
        parallel_for<class matrix_add>({ N, M },
                                   [=] (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 */
    auto C = c.get_access<access::read, access::host_buffer>();
    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);
        }
  } /* End scope of myQueue, this wait for any remaining operations on the
       queue to complete */
  std::cout << "Good computation!" << std::endl;
  return 0;
}
```

From work-groups & work-items to hierarchical parallelism

```
const int size = 10;
int data[size];
const int gsize = 2;
buffer<int> my_buffer { data, size };

command_group(my_queue, [&]() {
 auto in = my_buffer.get_access<access::read>();
 auto out = my_buffer.get_access<access::write>();
 // Iterate on the work-group
 parallel_for_workgroup<class hierarchical>({ size,
                                              gsize },
   [=](group<> grp) {
     std::cerr << "Gid=" << grp[0] << std::endl;
     // Iterate on the work-items of a work-group
     parallel_for_workitem(grp, [=](item<1> tile) {
       std::cerr << "id␣=" << tile.get_local()[0]
                 << "␣" << tile.get_global()[0]
                 << std::endl;
       out[tile] = in[tile] * 2;
     });
   });
});
```

- Easy to understand the concept of work-groups
- Easy to write work-group only code
- Replace code + barriers with several parallel_for_workitem()
  - Performance-portable between CPU and GPU
  - No need to think about barriers (automatically deduced)
  - Easier to compose components & algorithms
- Very close to OpenMP 4 style! ☺

OpenCL interoperability: the SYCL superpower

- By default do not expose OpenCL: make simple things simpler
- For kernel optimization in C++
  - Provide OpenCL C intrinsics in cl :: sycl namespace
  - Provide arithmetic/swizzling with cl :: sycl :: int4, cl :: sycl :: double16...
- Can also call *easily* external OpenCL C kernels
- Access underlying OpenCL objects from SYCL objects

```
cl_mem a_cl_buf = my_buffer.get_accessor<access::read_write,
                                         access::cl_buffer>();
```

  - Interface with external OpenCL framework
  - Interoperability with all OpenCL world: OpenGL, DirectX...

Outline

1. 2 great things in 2014
   - 2014 take 1: C++14
   - 2014 take 2: OpenCL SYCL
2. OpenCL SYCL
   - C++... putting everything altogether
   - Possible future extensions & applications
3. C++ `CL.hpp` OpenCL wrapper
4. Conclusion

Exascale-ready

- Use your own C++ compiler
  - Only kernel outlining needs SYCL compiler
- SYCL with C++ can address most of the hierarchy levels
  - MPI
  - OpenMP
  - C++-based PGAS DSeL (Coarray C++...)
  - Use storage abstraction of SYCL buffer for RDMA, out-of-core, PiM (Processor in Memory)...

Debugging

- Difficult to debug code or detect precondition violation on GPU and at large
- Rely on C++ to help debugging
  - Overload some operations and functions to verify preconditions
  - Hide tracing/verification code in constructors/destructors
  - Can use pure-C++ host implementation for bug-tracking with favorite debugger

C++11 allocators

- SYCL is not a magic wand when no OpenCL 2 system fine-grain shared memory available
- For complex data structures
  - Objects need to be in buffers to be shared between CPU and devices
  - Do not want marshaling/unmarshaling objects...
- $\exists$ C++11 allocators to control the way objects are allocated in memory
  - Use allocators to allocate some objects in OpenCL buffers!
  - Useful to send data through MPI and RDMA too!
  - Use `std::pointer_trait` for address translation on kernel side ☺

¿¿¿Fortran???

- Fortran 2003 introduces C-interop that can be used for C++ interop... SYCL
- C++ boost::multi_array & others provides à la Fortran arrays
  - Allows triplet notation
  - Can be used from inside SYCL to deal with Fortran-like arrays

Full disclosure

- Your favorite Fortran compiler may be written in C++...
- ...so C++ may be already good for you ☺
- Perhaps the right time to switch your application to a higher gear? ☺

Outline

SYCL and fine-grain system shared memory (HSA...)

```
#include <CL/sycl.hpp>
#include <iostream>
#include <vector>
using namespace cl::sycl;
int main() {
  std::vector a { 1, 2, 3 };
  std::vector b { 5, 6, 8 };
  std::vector c(a.size());

  // Enqueue a parallel kernel which is named "vector_add"
  parallel_for<class vector_add>(a.size(), [&] (int index) {
    c[index] = a[index] + b[index];
  });
  // Since there is no queue or no accessor, we assume parallel_for are blocking kernels

  std::cout << std::endl << "Result:" << std::endl;
  for(auto e : c)
    std::cout << e << " ";
  std::cout << std::endl;
  return 0;
}
```

- Very close to OpenMP simplicity
- You can keep the buffers & accessors for compatibility and let SYCL remove the copy for you if possible

Parallel STL towards C++17 proposal

- Current Parallel STL from C++17 proposal N4105 (2014/07/07)
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4105.pdf

```
// Current C++11: standard sequential sort
std::sort(vec.begin(), vec.end());
// C++17: permitting parallel execution and vectorization as well
sort(std::experimental::parallel::par_vec, vec.begin(), vec.end());
```

- Easy to implement in SYCL
- Could also be extended to give a kernel name (profile, debug...):

```
sycl_policy<class kernelName1> pol;
for_each(pol, begin(vec), end(vec), [](auto &ans) { ans = 42; });

sycl_policy<class kernelName2> pol2;
// But SYCL allows OpenCL intrinsics in the operation too! :-)
for_each(pol2, vec.begin(), vec.end(),
         [](float &ans) { ans += cl::sycl::sin(ans); });
```

Outline

C++ CL2.hpp in the OpenCL landscape?

- C++ wrapper atop C host only OpenCL API
- Newer CL2.hpp targeting OpenCL 2
  - 1-1 mapping of host C OpenCL API
  - Add new OpenCL2 objects (shared memory, pipes...)
- Move towards modern C++
  - Simplify interaction with C++ and STL
  - RAII (Resource/Responsability Acquisition Is Initialization)
  - Exception safety
  - Variadic templates to call kernels

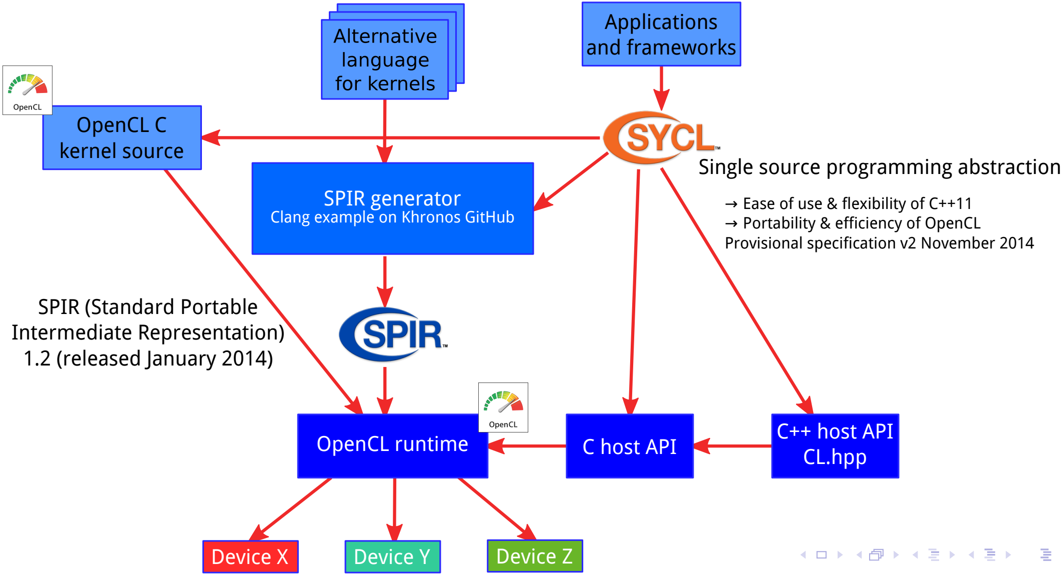
FAQ

- Are CL2.hp and SYCL the same?
  - No: CL2.hpp is just a C++ wrapper atop OpenCL C host API with 1-1 mapping
  - SYCL is quite more high-level and abstract
  - SYCL is single-source

AMD

Outline

SYCL in OpenCL ecosystem



Implementation status

- SYCLONE by Codeplay `https://www.codeplay.com/portal/blogs`
  - Presentation at SC14 LLVM-HPC workshop `http://llvm-hpc-workshop.github.io/talks.html#brown`
  - Come by AMD booth #839 to have demos
  - Talk on Wednesday, Nov 19th, 2:15 PM "*Using the SYCL for OpenCL open standard to accelerate C++ code*" (Andrew Richards, CEO Codeplay), AMD booth #839
- triSYCL Open Source project `https://github.com/amd/triSYCL`
  - Started to help SYCL committee for concept testing and slideware debugging
  - Pure C++14 & OpenMP CPU implementation (no OpenCL yet ☹)
  - Could evolve to a full-fledged implementation...
  - ...Need contributors!
- Next steps
  - Full specification, based on feedback
  - Khronos test suite for implementations
  - Release of implementations

Join us at the SYCL workshop during CGO 2015 in San Francisco, February

Conclusion

- Heterogeneous computing $\rightsquigarrow$ Rewriting applications?
  - Applications are to be refactored regularly anyway...
- Entry cost...
  - SYCL can mix several approaches such as OpenMP + OpenCL + MPI
- Exit cost! ☹
  - Use Open Standards backed by Open Source implementations
  - Be locked or be free!
- SYCL $\equiv$ best of pure modern C++ + OpenCL interoperability + task graph model
- Seamless integration with other C/C++ HPC frameworks: OpenMP, libraries (MPI, numerical), C++ DSeL (PGAS...)
- Like modern C++, SYCL make *simple* things *simpler*...
- ...but still make previously *impossible* things *possible*
- SYCL 1.2: still a provisional specification
  - It will be even better! ☺
  - ¡¡¡Get involved!!!
  - Already working on SYCL 2 targeting OpenCL 2