Modern C++, heterogeneous computing & OpenCL SYCL

Ronan Keryell

Khronos OpenCL SYCL committee

05/12/2015

IWOCL 2015 SYCL Tutorial
Outline

1. C++14
2. C++ dialects for OpenCL (and heterogeneous computing)
3. OpenCL SYCL 1.2
   - C++... putting everything altogether
4. OpenCL SYCL 2.1...
5. Conclusion
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 parallel task forces
  - Parallelism TS (Technical Specification) with Parallel STL
  - Concurrency TS (threads, mutex...)
  - Array TS (multidimensional arrays à la Fortran)
  - Transactional Memory TS...

Race to parallelism! Definitely matters for HPC and heterogeneous computing!

C++ is a complete new language

- Forget about C++98, C++03...
- Send your proposals and get involved in C++ committee (pushing heterogeneous computing)!
Modern C++ & HPC

- Huge library improvements
  - `<thread>` library and multithread memory model `<atomic>` ➔ 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; });
  ```
Lot of meta-programming improvements to make meta-programming easy:

- variadic templates, type traits `<type_traits>`...

- Make simple things simpler to be able to write generic numerical libraries, etc.

- Automatic type inference for terse programming

  ► Python 3.x (interpreted):

  ```python
def add(x, y):
    return x + y
print(add(2, 3))  # 5
print(add("2", "3"))  # 23
```

  ► 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
```

Without using templated code! 

```cpp
//template <typename >
```
Modern C++ & HPC

- R-value references & std::move semantics
  - matrix_A = matrix_B + matrix_C
    - Avoid copying (TB, PB, EB... 😊) when assigning or function return
- Avoid raw pointers, malloc()/free()/delete[]: use references and smart pointers instead

// Allocate a double with new() and wrap it in a smart pointer
auto gen() { return std::make_shared<double> { 3.14 }; }

[...]
{
  auto p = gen(), q = p;
  *q = 2.718;
  // Out of scope, no longer use of the memory: deallocation happens here
}

- Lot of other amazing stuff...
- Allow both low-level & high-level programming... Useful for heterogeneous computing
Outline

1. C++14

2. C++ dialects for OpenCL (and heterogeneous computing)

3. OpenCL SYCL 1.2
   - C++... putting everything altogether

4. OpenCL SYCL 2.1...

5. Conclusion
OpenCL 2.1 C++ kernel language

- Announced at GDC, March 2015
- Move from C99-based kernel language to C++14-based

```cpp
// Template classes to express OpenCL address spaces
local_array<int, N> array;
local<float> v;
constant_ptr<double> p;
// Use C++11 generalized attributes, to ignore vector dependencies
[[safelen(8), ivdep]]
for (int i = 0; i < N; i++)
    // Can infer that offset >= 8
    array[i+offset] = array[i] + 42;
```
OpenCL 2.1 C++ kernel language

- Kernel side enqueue
  - Replace OpenCL 2 infamous Apple GCD block syntax by C++11 lambda

```cpp
kernel void main_kernel(int N, int *array) {
  // Only work-item 0 will launch a new kernel
  if (get_global_id(0) == 0)
    // Wait for the end of this work-group before starting the new kernel
    get_default_queue().enqueue_kernel(CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP, 
                                          ndrange { N },
                                          [=] kernel {
                                            array[get_global_id(0)] = 7;
                                          });
}
```

- C++14 memory model and atomic operations
- Newer SPIR-V binary IR format
OpenCL 2.1 C++ kernel language

- Amazing progress but no single source solution à la CUDA yet
  - Still need to play with OpenCL host API to deal with buffers, etc.
Bolt C++

- Developed by AMD on top of OpenCL, C++AMP or TBB

```cpp
#include <bolt/cl/sort.h>
#include <vector>
#include <algorithm>

int main() {
    // generate random data (on host)
    std::vector<int> a(8192);
    std::generate(a.begin(), a.end(), rand);
    // sort, run on best device in the platform
    bolt::cl::sort(a.begin(), a.end());
    return 0;
}
```

- Simple!
Bolt C++

- But...
  - No direct interoperability with OpenCL world
  - No specific compiler required with OpenCL — some special syntax to define operation on device
    - OpenCL kernel source strings for complex operations with macros $BOLT\_FUNCTOR()$, $BOLT\_CREATE\_TYPENAME()$, $BOLT\_CREATE\_CLCODE()$...
    - Work better with AMD Static C++ Kernel Language Extension (now in OpenCL 2.1) & best with C++AMP (but no OpenCL interoperability...)
Boost.Compute

- Boost library accepted in 2015 https://github.com/boostorg/compute
- Provide 2 levels of abstraction
  - High-level parallel STL
  - Low-level C++ wrapping of OpenCL concepts
// Get a default command queue on the default accelerator
auto queue = boost::compute::system::default_queue();
// Allocate a vector in a buffer on the device
boost::compute::vector<float> device_vector { N, queue.get_context() };
boost::compute::iota(device_vector.begin(), device_vector.end(), 0);

// Create an equivalent OpenCL kernel
BOOST.Compute_FUNCTION(float, add_four, (float x), { return x + 4; });
boost::compute::transform(device_vector.begin(), device_vector.end(),
                          device_vector.begin(), add_four, queue);
boost::compute::sort(device_vector.begin(), device_vector.end(), queue);
// Lambda expression equivalent
boost::compute::transform(device_vector.begin(), device_vector.end(),
                          device_vector.begin(),
                          boost::compute::lambda::_1 * 3 - 4, queue);
• Elegant implicit C++ conversions between OpenCL and Boost.Compute types for finer control and optimizations

```cpp
auto command_queue = boost::compute::system::default_queue();
auto context = command_queue.get_context();
auto program =
    boost::compute::program::create_with_source_file(kernel_file_name, context);
program.build();
boost::compute::kernel im2col_kernel { program, "im2col" };

boost::compute::buffer im_buffer { context, image_size*sizeof(float),
    CL_MEM_READ_ONLY };
command_queue.enqueue_write_buffer(im_buffer, 0 /* Offset */,
    im_data.size()*sizeof(decltype(im_data)::value_type),
    im_data.data());
```
C++ dialects for OpenCL (and heterogeneous computing)

Boost.Compute

```cpp
im2col_kernel.set_args(im_buffer,
height, width,
ksz_h, ksize_w,
pad_h, pad_w,
stride_h, stride_w,
height_col, width_col,
data_col);

command_queue.enqueue_nd_range_kernel(kernel,
  boost::compute::extents<1> { 0 } /* global work offset */,
  boost::compute::extents<1> { workitems } /* global work-item */,
  boost::compute::extents<1> { workgroup_size }; /* Work group size */);
```

- Provide program caching
- Direct OpenCL interoperability for extreme performance
- No specific compiler required \(\rightarrow\) some special syntax to define operation on device
- Probably the right tool to use to translate CUDA & Thrust to OpenCL world
C++ dialects for OpenCL (and heterogeneous computing)

**VexCL**

- Parallel STL similar to Boost.Compute + mathematical libraries
  - [https://github.com/ddemidov/vexcl](https://github.com/ddemidov/vexcl)
  - Random generators (Random123)
  - FFT
  - Tensor operations
  - Sparse matrix-vector products
  - Stencil convolutions
  - ...

- OpenCL (CL.hpp or Boost.Compute) & CUDA back-end

- Allow device vectors & operations to span different accelerators from different vendors in a same context

```cpp
vex::Context ctx { vex::Filter::Type { CLDEVICE_TYPE_GPU } && vex::Filter::DoublePrecision }; vex::vector<double> A { ctx, N }, B { ctx, N }, C { ctx, N }; A = 2 * B - sin(C);```

Modern C++, heterogeneous computing & OpenCL SYCL IWOCL 2015
C++ dialects for OpenCL (and heterogeneous computing) ▶ VexCL (II) ▶ Allow easy interoperability with back-end

```cpp
// Get the cl_buffer storing A on the device 2
auto clBuffer = A(2);
```

• Use heroic meta-programming to generate kernels without using specific compiler with deep embedded DSL

▶ Use symbolic types (prototypal arguments) to extract function structure

```cpp
// Set recorder for expression sequence
std::ostringstream body;
veex::generator::set_recorder(body);
veex::symbolic<double> sym_x { veex::symbolic<double>::VectorParameter };
sym_x = sin(sym_x) + 3;
sym_x = cos(2*sym_x) + 5;
// Build kernel from the recorded sequence
auto foobar = veex::generator::build_kernel(ctx, "foobar", body.str(), sym_x);
```
VexCL

// Now use the kernel
foobar(A);

- VexCL is probably the most advanced tool to generate OpenCL without requiring a specific compiler...
- Interoperable with OpenCL, Boost.Compute for extreme performance & ViennaCL
- Kernel caching to avoid useless compiling
- Probably the right tool to use to translate CUDA & Thrust to OpenCL world
ViennaCL

https://github.com/viennacl/viennacl-dev

- OpenCL/CUDA/OpenMP back-end
- Similar to VexCL for sharing context between various platforms
- Linear algebra (dense & sparse)
- Iterative solvers
- FFT
- OpenCL kernel generator from high-level expressions
- Some interoperability with Matlab
C++ dialects for OpenCL (and heterogeneous computing)

C++AMP

// Use iota algorithm in C++AMP
#include <amp.h>
#include <iostream>

enum { NWITEMS = 512 };

int data[NWITEMS];

// To avoid writing Concurrency:: everywhere
using namespace Concurrency;

void iota_n(size_t n, int dst[]) {
    // Select the first true accelerator found as the default one
    for (auto const & acc : accelerator::get_all())
        if (!acc.get_is_emulated()) {
            accelerator::set_default(acc.get_device_path());
            break;
        }

    // Define the iteration space
    extent<1> e(n);
    // Create a buffer from the given array memory
    array_view<int, 1> a(e, dst);
    // Is there a better way to express write-only data?
    a.discard_data();
    // Execute a kernel in parallel

parallel_for_each(e,
    // Define the kernel to execute
    [=] (Concurrency::index<1> i) restrict(amp) {
        a[i] = i[0];
    });
    // In the destruction of array_view "a" happening here,
    // the data are copied back before iota_n() returns
}

- Developed by Microsoft, AMD & MultiCoreWare
- Single source: easy to write kernels
- Require specific compiler
- Not pure C++ (restrict, tile_static)
- No OpenCL interoperability
- Difficult to optimize the data transfers
C++ dialects for OpenCL (and heterogeneous computing)

OpenMP 4

```c
#include <stdio.h>

enum { NWITEMS = 512 };
int array[NWITEMS];

void iota_n(size_t n, int dst[n]) {
    #pragma omp target map(from: dst[0:n-1])
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
        dst[i] = i;
}

int main(int argc, const char *argv[]) {
    iota_n(NWITEMS, array);
    // Display results
    for (int i = 0; i < NWITEMS; i++)
        printf("%d_%d\n", i, array[i]);
    return 0;
}
```

- Old HPC standard from the 90's
- Use `#pragma` to express parallelism
- OpenMP 4 extends it to accelerators
  - Work-group parallelism
  - Work-item parallelism
- Deal with CPU & heterogeneous computing parallelism
- No LDS support
- No OpenCL interoperability
- But quite simple! Single source...

Modern C++, heterogeneous computing & OpenCL SYCL
Other (non-)OpenCL C++ framework

- ArrayFire, Aura, CLOGS, hemi, HPL, Kokkos, MTL4, SkelCL, SkePU, EasyCL...
- nVidia CUDA 7 now C++11-based
  - Single source \( \rightarrow \) simpler for the programmer
  - nVidia Thrust \( \approx \) parallel STL+map-reduce on top of CUDA, OpenMP or TBB
    - [https://github.com/thrust/thrust](https://github.com/thrust/thrust)
      - Not very clean because device pointers returned by `cudaMalloc()` do not have a special type
        \( \rightarrow \) use some ugly casts
- OpenACC \( \approx \) OpenMP 4 restricted to accelerators + LDS finer control
Missing link...

- No tool providing
  - OpenCL interoperability
  - Modern C++ environment
  - Single source for programming productivity
Outline

1. C++14
2. C++ dialects for OpenCL (and heterogeneous computing)
3. OpenCL SYCL 1.2
   - C++... putting everything altogether
4. OpenCL SYCL 2.1...
5. Conclusion
Puns and pronunciation explained

OpenCL SYCL

sickle [ 'si-kəl ]

OpenCL SPIR

spear [ 'spir ]
OpenCL SYCL goals

- Ease of use
  - Single source programming model
    - Take advantage of CUDA & C++AMP simplicity and power
    - Compiled for host and device(s)
- Easy 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
  - Exceptions for error handling
- Portability across platforms and compilers
- Providing the full OpenCL feature set and seamless integration with existing OpenCL code
- Task graph programming model with interface à la TBB/Cilk (C++17)
- High performance

http://www.khronos.org/opencl/sycl
Complete example of matrix addition in OpenCL SYCL

```cpp
#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
        myQueue.submit([&](handler& cgh) {
            // Define the data used/produced
            auto ka = A.get_access<access::read>(cgh);
            auto kb = B.get_access<access::read>(cgh);
            auto kc = C.get_access<access::write>(cgh);

            // Create & call OpenCL kernel named "mat_add"
            cgh.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.

    return 0;
}
```
Asynchronous task graph model

- Theoretical graph of an application described *implicitly* with kernel tasks using buffers through accessors

```
cl::sycl::accessor<write> init_a
cl::sycl::buffer a
cl::sycl::accessor<read>

init_b
cl::sycl::buffer b
cl::sycl::accessor<write>

matrix_add
cl::sycl::buffer c
cl::sycl::accessor<write>
cl::sycl::accessor<read>
```

- Possible schedule by SYCL runtime:

```
init_b init_a matrix_add
```

〜ardware 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 (AMD synchronous queues, AMD compute rings, AMD DMA rings...)
#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
    myQueue.submit([&](auto &cgh) {
        // The kernel write a, so get a write accessor on it
        auto A = a.get_access<access::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
    myQueue.submit([&](auto &cgh) {
        // The kernel write b, so get a write accessor on it
        auto B = b.get_access<access::write>(cgh);

        // Enqueue 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
    myQueue.submit([&](auto &cgh) {
        // In the kernel a and b are read, but c is written
        auto A = a.get_access<access::read>(cgh);
        auto B = b.get_access<access::read>(cgh);
        auto C = c.get_access<access::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::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

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

my_queue.submit([&](auto &cgh) {
    auto in = my_buffer.get_access<access::read>(cgh);
    auto out = my_buffer.get_access<access::write>(cgh);
    // Iterate on the work-group
    cgh.parallel_for_workgroup<class hierarchical>({ size, gsize

        [=](group<> grp) {
            // Code executed only once per work-group
            std::cerr << "Gid=" << grp[0] << std::endl;
            // Iterate on the work-items of a work-group
            cgh.parallel_for_workitem(grp, [=](item<> tile) {
                std::cerr << "id=" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id" "id"
                << tile.get_local()[0] << std::endl;
                out[tile] = in[tile] * 2;
            });
            // Can have other cgh.parallel_for_workitem() here...
        });
    });
```

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_workitem()
  - Performance-portable between CPU and GPU
  - No need to think about barriers (automatically deduced)
  - Easier to compose components & algorithms
  - Ready for future GPU with non uniform work-group size
C++11 allocators

- ∃ C++11 allocators to control the way objects are allocated in memory
  - For example to allocate some vectors on some storage
  - Concept of `scoped_allocator` to control storage of nested data structures
  - Example: vector of strings, with vector data and string data allocated in different memory areas (speed, power consumption, caching, read-only...)
- SYCL reuses `allocator` to specify how `buffer` and `image` are allocated on the host side
Outline

1. C++14
2. C++ dialects for OpenCL (and heterogeneous computing)
3. OpenCL SYCL 1.2
   - C++... putting everything altogether
4. OpenCL SYCL 2.1...
5. 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 (Partitioned Global Address Space) DSeL (Domain-Specific embedded Language, such as Coarray C++...)
  - Remote accelerators in clusters
  - Use SYCL buffer allocator for
    - RDMA
    - Out-of-core, mapping to a file
    - PiM (Processor in Memory)
    - ...

Modern C++, heterogeneous computing & OpenCL SYCL
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
Poor-man SVM with C++11 + SYCL

• For complex data structures
  ► Objects need to be in buffers to be shipped between CPU and devices
  ► Do not want marshaling/unmarshaling objects...
  ► Use C++11 allocator to allocate some objects in 1 SYCL buffer
    ■ Useful to send efficiently data through MPI and RDMA too!
  ► But since no SVM, not same address on CPU and GPU side...
    ■ How to deal with pointers?
    ■ Override all pointer accessed (for example use std::pointer_trait) to do address translation on kernel side
      Cost: 1 addition per *p

• When no or inefficient SVM...
  ► Also useful optimization when need to work on a copy only on the GPU
    ■ Only allocation on GPU side
    ■ Spare some TLB trashing on the CPU
¿¿¿Fortran???

- Fortran 2003 introduces C-interoperability that can be used for C++ interoperability... 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
- Perhaps the right time to switch your application to modern C++? 😊
Using SYCL-like models in other areas

- SYCL $\equiv$ generic heterogeneous computing model beyond OpenCL
  - queue expresses where computations happen
  - parallel_for launches computations
  - accessor defines the way we access data
  - buffer for storing data
  - allocator for defining how data are allocated/backed

- Example for HSA: almost direct mapping à la OpenCL

- Example in PiM world
  - Use queue to run on some PiM chips
  - Use allocator to distribute data structures or to allocate buffer in special memory (memory page, chip...)
  - Use accessor to use alternative data access (split address from computation, streaming only, PGAS...)
  - Use pointer_trait to use specific way to interact with memory
  - ...

Modern C++, heterogeneous computing & OpenCL SYCL

IWOCL 2015
Outline

1. C++14
2. C++ dialects for OpenCL (and heterogeneous computing)
3. OpenCL SYCL 1.2
   - C++... putting everything altogether
4. OpenCL SYCL 2.1...
5. Conclusion
SYCL 2.1 is coming!

- Skip directly to OpenCL 2.1 and C++14
- Kernel side enqueue
- Shared memory between host and accelerator
- Parallel STL C++17
- Array TS
SYCL and fine-grain system shared memory (OpenCL 2)

```cpp
#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
    parallel_for(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 << "\n";
    std::cout << std::endl;
    return 0;
}
```

- Very close to OpenMP simplicity
- Can still use of buffers & accessors for compatibility & finer control (task graph, optimizations...)
  - SYCL can remove the copy when possible
Outline

1. C++14
2. C++ dialects for OpenCL (and heterogeneous computing)
3. OpenCL SYCL 1.2
   - C++... putting everything altogether
4. OpenCL SYCL 2.1...
5. Conclusion
Many C++ frameworks to leverage OpenCL
  ▶ None of them provides seamless single source
    ■ Require some kind of macros & weird syntax
  ▶ But they should be preferred to plain OpenCL C for productivity
SYCL provides seamless single source with OpenCL interoperability
  ▶ Can be used to improve other higher-level frameworks
SYCL ≡ pure C++ \(\mapsto\) integration with other C/C++ HPC frameworks: OpenCL, OpenMP, libraries (MPI, numerical), C++ DSeL (PGAS...)...
SYCL also interesting as co-design tool for architectural & programming model research (PiM, Near-Memory Computing, various computing models...)
Modern C++ is not just C program in *.cpp file \(\mapsto\) Invest in learning modern C++
## Table of content

### 1. C++14
- Outline
- C++14
  - Modern C++ & HPC

### 2. C++ dialects for OpenCL (and heterogeneous computing)
- Outline
- OpenCL 2.1 C++ kernel language
- Bolt C++
- Boost.Compute
- VexCL
- ViennaCL
- C++AMP
- OpenMP 4
- Other (non-)OpenCL C++ framework
- Missing link...

### 3. OpenCL SYCL 1.2
- Outline
- Puns and pronunciation explained
- OpenCL SYCL goals

<table>
<thead>
<tr>
<th>Outline</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete example of matrix addition in OpenCL SYCL</td>
<td>28</td>
</tr>
<tr>
<td>Asynchronous task graph model</td>
<td>29</td>
</tr>
<tr>
<td>Task graph programming — the code</td>
<td>30</td>
</tr>
<tr>
<td>From work-groups &amp; work-items to hierarchical parallelism</td>
<td>31</td>
</tr>
<tr>
<td>C++11 allocators</td>
<td>32</td>
</tr>
<tr>
<td>C++... putting everything altogether</td>
<td>33</td>
</tr>
<tr>
<td>Outline</td>
<td>33</td>
</tr>
<tr>
<td>Exascale-ready</td>
<td>34</td>
</tr>
<tr>
<td>Debugging</td>
<td>35</td>
</tr>
<tr>
<td>Poor-man SVM with C++11 + SYCL</td>
<td>36</td>
</tr>
<tr>
<td>¿¿¿Fortran???</td>
<td>37</td>
</tr>
<tr>
<td>Using SYCL-like models in other areas</td>
<td>38</td>
</tr>
<tr>
<td>OpenCL SYCL 2.1...</td>
<td>39</td>
</tr>
<tr>
<td>Outline</td>
<td>39</td>
</tr>
<tr>
<td>SYCL 2.1 is coming!</td>
<td>40</td>
</tr>
<tr>
<td>SYCL and fine-grain system shared memory (OpenCL 2)</td>
<td>41</td>
</tr>
<tr>
<td>Conclusion</td>
<td>42</td>
</tr>
<tr>
<td>Outline</td>
<td>43</td>
</tr>
<tr>
<td>Conclusion</td>
<td>44</td>
</tr>
</tbody>
</table>

### 4. Conclusion
- You are here!