XILINX
ALL PROGRAMMABLE™

Open standards from Khronos for heterogeneous computing: Vulkan, SPIR-V & OpenCL SYCL

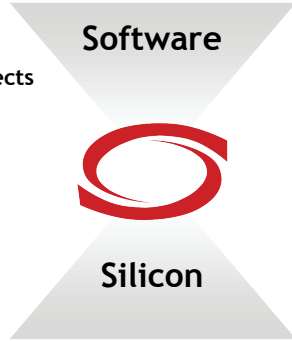
Ronan Keryell
Xilinx Research Labs & Khronos OpenCL SYCL committee member

2015/12/16

Khronos Connects Software to Silicon

Open Consortium creating OPEN STANDARD APIs for hardware acceleration
Any company is welcome – many international members – one company one vote

ROYALTY-FREE specifications
State-of-the art IP framework protects members AND the standards



International, non-profit organization
Membership fees cover operating and engineering expenses

Low-level silicon APIs needed on every platform
Graphics, parallel compute, rich media, vision, sensor and camera processing

API Specifications AND Conformance Tests for cross-vendor portability

Strong industry momentum
100s of man years invested by industry experts

Well over a *BILLION* people use Khronos APIs *Every Day…*

Outline

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 value...
    - 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 !

SPIR-V transforms the language ecosystem

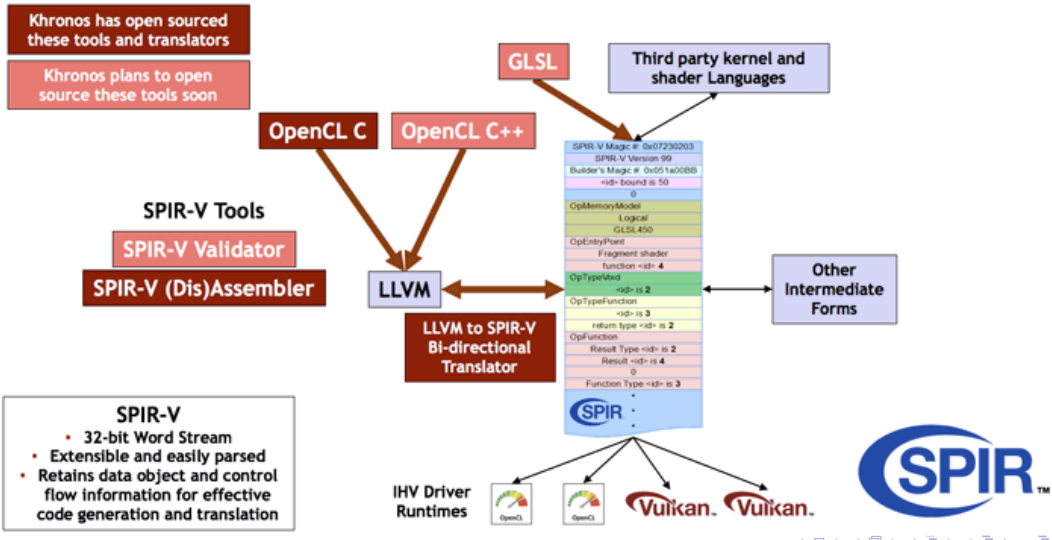
- First multi-API, intermediate language for parallel compute *and* graphics
  - Native representation for Vulkan shader and OpenCL kernel source languages
  - `https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf`
- Cross-vendor intermediate representation
  - Language front-ends can easily 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

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

Not based on LLVM to isolate from LLVM roadmap changes

Driving SPIR-V Open Source ecosystem



SPIR-V 1.0 Resources

- SPIR-V 1.0 specification available in Khronos Registry
  `https://www.khronos.org/registry/spir-v`
- Feedback forum for questions and feedback
  `https://forums.khronos.org/showthread.php/12919-Feedback-SPIR-V`
- Whitepaper “An Introduction to SPIR-V”
  `https://www.khronos.org/registry/spir-v/papers/WhitePaper.html`
- Bug reporting
  `https://www.khronos.org/bugzilla/enter_bug.cgi?product=SPIR-V`
- SPIR-V tools project including an assembler, binary module parser, disassembler & validator for SPIR-V `https://github.com/KhronosGroup/SPIRV-Tools`
- LLVM framework with SPIR-V support including an LLVM↔SPIR-V bi-directional converter `https://github.com/KhronosGroup/SPIRV-LLVM`
- GLSL compiler in development

Outline

The Need for Vulkan

Ground-up design of a modern open standard API for driving high-efficiency graphics and compute on GPUs used across diverse devices

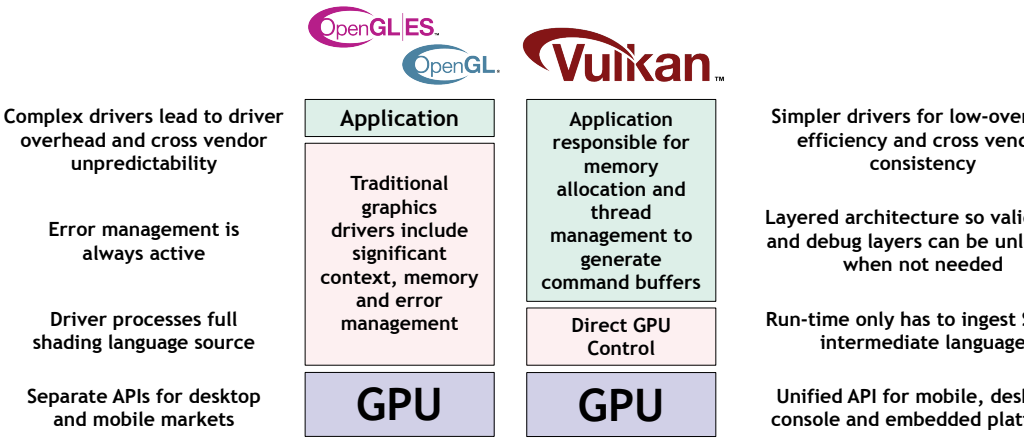
Vulkan™

- Simpler drivers for low-overhead efficiency and cross vendor consistency
- Unified API for mobile, desktop, console and embedded platforms
- Layered architecture so validation and debug layers unloaded when not needed

In the twenty two years since OpenGL was invented – the architecture of GPUs and platforms has changed radically

GPUs being used for graphics, compute and vision processing on a rapidly *increasing* diversity of platforms – *increasing* the need for cross-platform standards

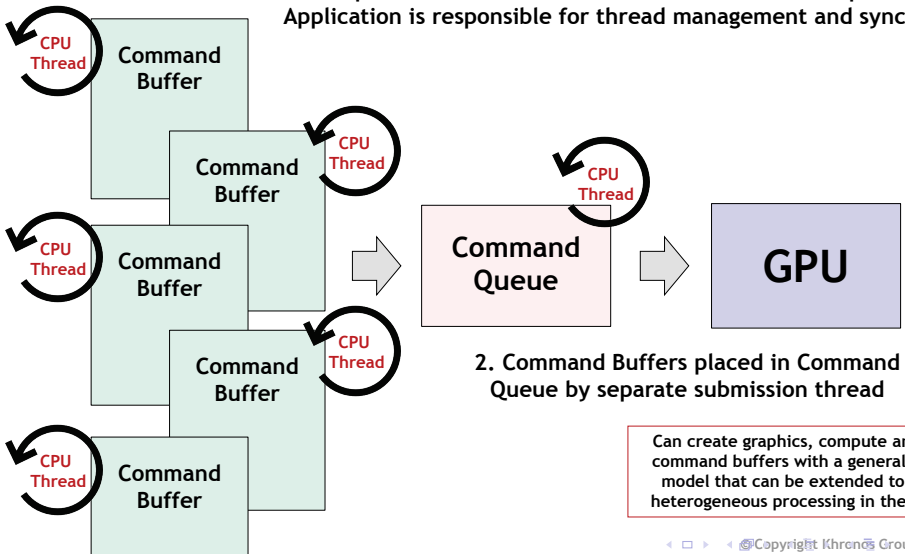
Vulkan Explicit GPU Control



Vulkan delivers the maximized performance and cross platform portability needed by sophisticated engines, middleware and apps

Vulkan Multi-threading Efficiency

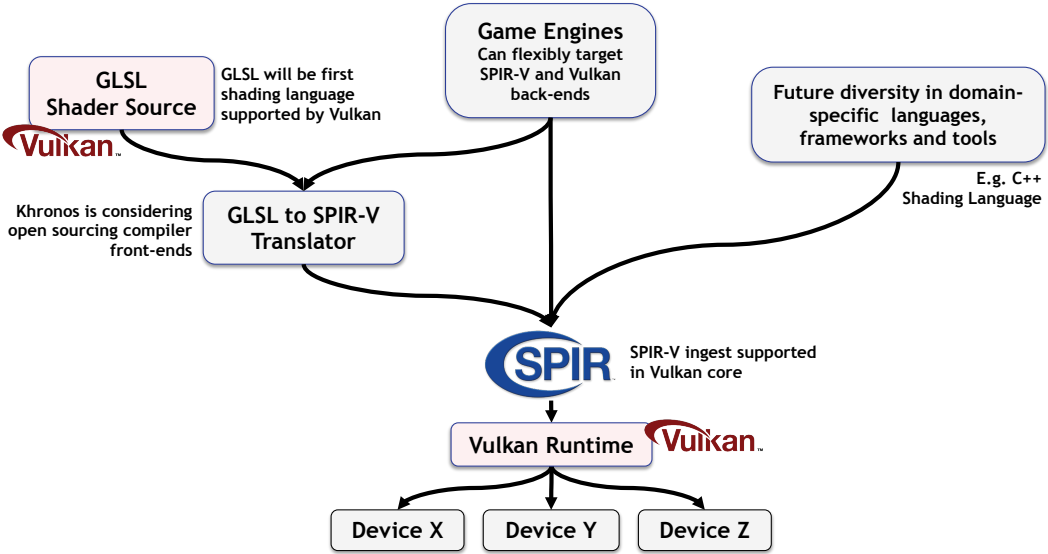
1. Multiple threads can construct Command Buffers in parallel
Application is responsible for thread management and synch



2. Command Buffers placed in Command Queue by separate submission thread

Can create graphics, compute and DMA command buffers with a general queue model that can be extended to more heterogeneous processing in the future

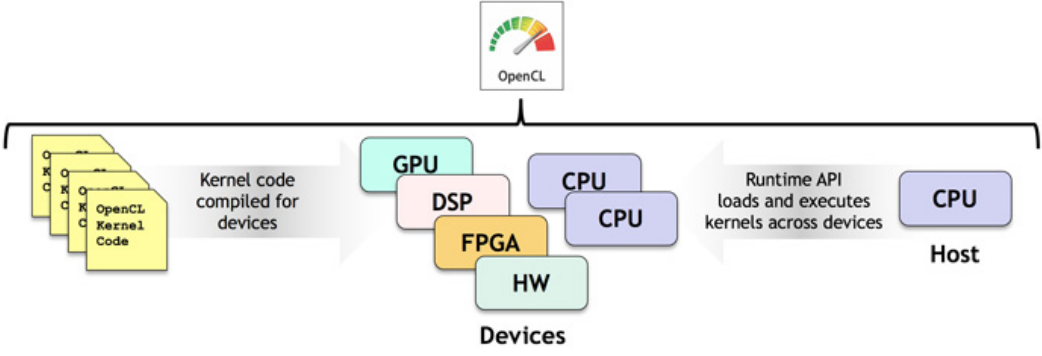
Vulkan Language Ecosystem



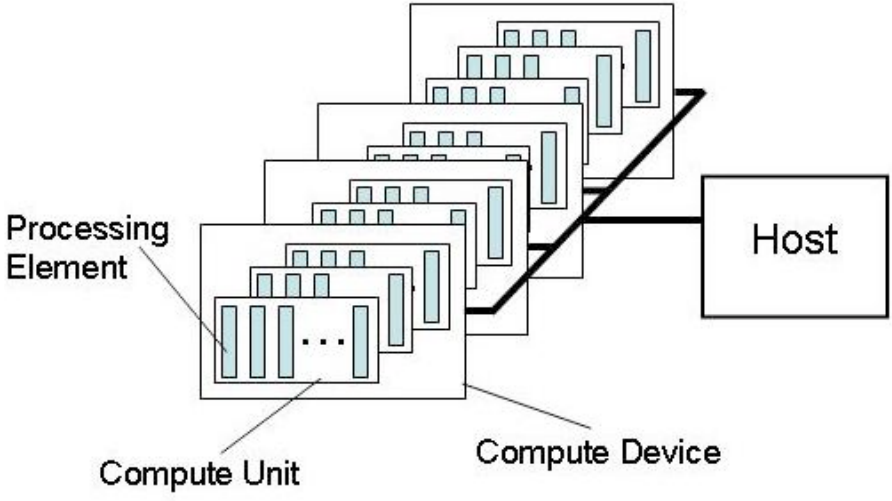
Outline

OpenCL

- OpenCL: 2 APIs and 2 kernel languages
  - C Platform Layer API to query, select and initialize compute devices
  - OpenCL C and (soon) OpenCL C++ kernel languages to write parallel code
  - C Runtime API to build and execute kernels across multiple devices
- One code tree can be executed on CPU, GPU, DSP, FPGA and hardware
  Dynamically balance work across available processors

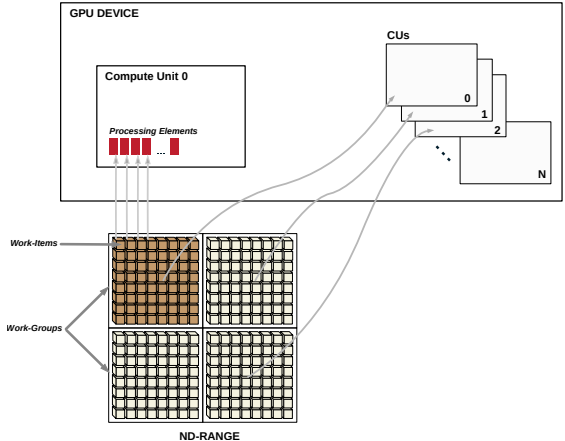


Architecture model

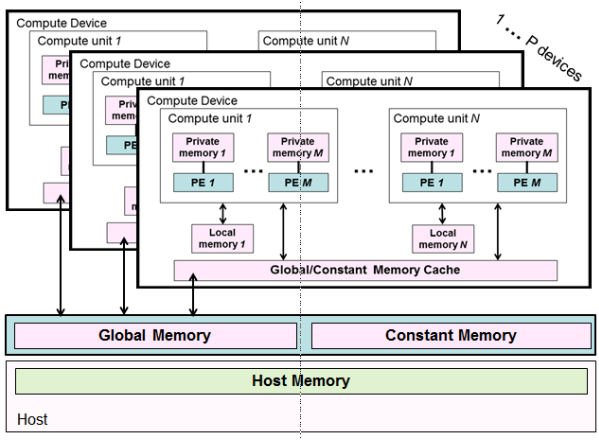


- Host threads launch computational *kernels* on *accelerators*

Execution model



Memory model



Outline

Share Virtual Memory (SVM) *(I)*

3 variations...

Coarse-Grained memory buffer SVM

- Sharing at the granularity of regions of OpenCL buffer objects
  - clSVMAlloc()
  - clSVMFree()
- Consistency is enforced at synchronization points
- Update views with clEnqueueSVMMap(), clEnqueueSVMUnmap(), clEnqueueMapBuffer() and clEnqueueUnmapMemObject() commands
- Similar to non-SVM but allows shared pointer-based data structures

Share Virtual Memory (SVM) *(II)*

Fine-Grained memory buffer SVM

- Sharing at the granularity of individual loads/stores into bytes within OpenCL buffer memory objects
- Consistency guaranteed only at synchronization points
- Optional OpenCL atomics to provide fine-grained consistency
  - No need to use previous ... Map()/ ... Unmap()

Share Virtual Memory (SVM) *(III)*

Fine-Grained **system** SVM *à la* C(++)11

- Sharing occurs at the granularity of individual loads/stores into bytes occurring **anywhere within the host memory**
  - Allow normal memory such as malloc()
- Loads and stores may be cached so consistency is guaranteed only at synchronization points
- Optional OpenCL atomics to provide fine-grained consistency

New pointer __attribute__((nosvm))

Lambda expressions with Block syntax

- From Mac OS X's Grand Central Dispatch, implemented in Clang

```
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
  return num*multiplier;
};
printf("%d\n", myBlock(3)); // prints 21
```

- By-reference closure but const copy for automatic variable
- Equivalent in C++11

```
auto myBlock = [=] (int num) {
  return num*multiplier;
};
```

Device-side enqueue *(I)*

- OpenCL 2 allows nested parallelism
- Child kernel enqueued by kernel on a device-side command queue
- Out-of-order execution
- Use events for synchronization
-  No kernel preemption ⇝ Continuation-passing style! ☺
  `en.wikipedia.org/wiki/Continuation-passing_style`

Device-side enqueue *(II)*

```
// Find and start new jobs
kernel void
evaluate_work(...) {
  /* Check if more work needs to be performed,
     for example a tree or list traversal */
  if (check_new_work(...)) {
    /* Start workers with the right //ism on default
       queue only after the end of this launcher */
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange_1D(compute_size(...)),
                   ^{ real_work(...); });
  }
}
```

Device-side enqueue (III)

```
// Cross-recursion example for dynamic //ism
kernel void
real_work(...) {
  // The real work should be here
  [...]
  /* Queue a single instance of evaluate_work()
     to device default queue to go on recursion */
  if (get_global_id(0) == 0) {
    /* Wait for the *current* kernel execution
       before starting the *new one* */
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange_1D(1),
                   ^{ evaluate_work(...); });
  }
}
```

Collective work-group operators

- Operation involving all work-items inside a work-group
  - int work_group_all(int predicate)
  - int work_group_any(int predicate)
  - gentype work_group_broadcast(gentype a, size_t id_x...)
  - gentype work_group_reduce_*op* (gentype x)
  - gentype work_group_scan_exclusive_*op* (gentype x)
  - gentype work_group_scan_inclusive_*op* (gentype x)
- `http://en.wikipedia.org/wiki/Prefix_sum`

Subgroups

- Represent real execution of work-items inside work-groups
  - 1-dimensional
  - *wavefront* on AMD GPU, *warp* on nVidia GPU
  -  There may be more than 1 subgroup/work-group...
- Coordinate `uint get_sub_group_id()`, `uint get_sub_group_local_id()`, `uint get_sub_group_size()`, `uint get_num_sub_groups()`...
- `void sub_group_barrier(...)`
- Collective operators `sub_group_reduce_op()`, `sub_group_scan_exclusive_op()`, `sub_group_scan_inclusive_op()`, `sub_group_broadcast()`...

Pipe

- Efficient connection between kernels for stream computing
- Ordered sequence of data items
- One kernel can write data into a pipe
- One kernel can read data from a pipe
- *Very* useful on FPGA to implement streaming applications

```
cl_mem clCreatePipe(cl_context context,
                    cl_mem_flags flags,
                    cl_uint pipe_packet_size,
                    cl_uint pipe_max_packets,
                    const cl_pipe_properties *props,
                    cl_int *errcode_ret)
```

- Kernel functions to read/write/test packets from/to pipes
  - read_pipe(), write_pipe()
  - reserve_read_pipe(), reserve_write_pipe(), commit_read_pipe(), commit_write_pipe()

ND range not multiple of work-group size

- Global iteration space may not be multiple of work-group size
- OpenCL 2 introduces concept of non-uniform work-group size
- Last work-group in each dimension may have less work-items
- Up to 8 cases in 3D
  - size_t get_local_size (uint dimindx)
  - size_t get_enqueued_local_size(uint dimindx)

Other improvements in OpenCL 2

- MIPmaps (*multum in parvo* map): textures at different LOD (level of details)
- Local and private memory initialization (*à la* calloc())
- Read/write images __read_write
- Interoperability with OpenGL, Direct3D...
- Images (support for 2D image from buffer, depth images and standard IEC 61996-2-1 sRGB image)
- Linker to use libraries with clLinkProgram()
- Generic address space __generic (for __global, __local or __private)
- Program scope variables in global address space
- C11 atomics
- Clang blocks ($\approx$ C++11 lambda in C)
- int printf (constant char * restrict format, ...) with vector extensions
- Kernel-side events & profiling

Outline

OpenCL 2.1

- Released in November 2015 @ SC15
- Support for the SPIR-V 1.0 intermediate language in core
  - E.g. SPIR-V used to ingest from diverse language front-ends
  - OpenCL C ingestion still supported to preserve kernel code investment
- Subgroups for finer grain control of hardware threading now in core
- Runs on any OpenCL 2.0-capable hardware $\rightsquigarrow$ Only driver update required
- Provisional OpenCL C++ kernel language to write parallel code

OpenCL 2.1 C++ kernel language *(I)*

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

```
// 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 *(II)*

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

```
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 *(III)*

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

[ Spoiler alert: see SYCL later]

OpenCL 2.1 API Enhancements

- `clCreateProgramWithIL`: SPIR-V support built-in to the runtime
- Subgroup query operations
  - Subgroups expose hardware threading in the core feature set
- `clCloneKernel`: enables copying of kernel objects and state
  - Safe implementation of copy constructors in wrapper classes
- Low-latency device timer queries
  - Support alignment of profiling between device and host code
- Priority and throttle hint extensions for queues
  - Specify execution priority on a per-queue basis
- Zero-size enqueue: Zero-sized dispatches are valid from the host

Outline

1. SPIR-V
2. Vulkan
3. OpenCL
   - OpenCL 2.0
   - OpenCL 2.1
4. SYCL
   - C++14
   - C++ dialects for OpenCL (and heterogeneous computing)
   - OpenCL SYCL 1.2
   - OpenCL SYCL 2.2...
   - triSYCL

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 *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 *(I)*

- 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 λ expressions (anonymous functions)

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

Modern C++ & HPC *(II)*

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

```
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:

```
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 >`~~ ☺

Modern C++ & HPC *(III)*

- 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
}
```

Modern C++ & HPC *(IV)*

- C++14 generalizes constexpr to statements

```
constexpr auto fibonacci(int v) {
  long long int u_n_minus_1 = 0;
  auto u_n = u_n_minus_1 + 1;
  for (int i = 1; i < v; ++i) {
    auto tmp = u_n;
    u_n += u_n_minus_1;
    u_n_minus_1 = tmp;
  }
  return u_n;
}
int main() {
  constexpr auto result = fibonacci(80);
  std::cout << result << std::endl;
  return 0;
}
```

Modern C++ & HPC (V)

Compiled to

```
movabsq $23416728348467685, %rsi # imm = 0x533163EF0321E5
movl    $_ZSt4cout, %edi
callq   _ZNSo9_M_insertIxEERSoT_
```

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

Outline

Boost.Compute *(I)*

- 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

Boost.Compute *(II)*

```
// 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);
```

Boost.Compute *(III)*

- Elegant implicit C++ conversions between OpenCL and Boost.Compute types for finer control and optimizations

```
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());
```

Boost.Compute *(IV)*

```
im2col_kernel.set_args(im_buffer,
                       height, width,
                       ksize_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 ⇝ some special syntax to define operation on device
- Probably the right tool to use to move CUDA & Thrust applications to OpenCL world

VexCL (I)

- Parallel STL similar to Boost.Compute + mathematical libraries
  `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

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

VexCL *(II)*

- Allow easy interoperability with back-end

```
// 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

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

VexCL *(III)*

```
// 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

OpenMP 4

```
#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...

Other (non-)OpenCL C++ framework

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

Missing link...

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

Outline

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

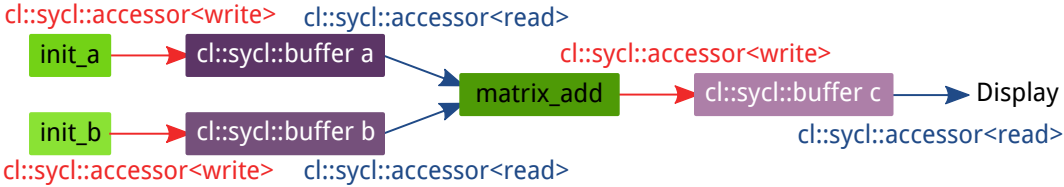
`http://www.khronos.org/opencl/sycl`

Complete example of matrix addition in OpenCL SYCL

```
#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.
   // Copy back the buffer data with RAII behaviour.
 return 0;
}
```

Asynchronous task graph model

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



- Possible schedule by SYCL runtime:

↝ 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...)

Task graph programming — the code

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
// Size of the matrices
constexpr size_t N = 2000;
constexpr 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);
        /* 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
    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) {
                                             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

```
constexpr int size = 10;
int data[size];
constexpr 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<1> tile) {
       std::cerr << "id␣=" << tile.get_local()[0]
                 << "␣" << tile.get_global()[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 `allocator`s 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

Exascale-ready

- Use your own C++ compiler
  - Only kernel outlining needs SYCL compiler
- SYCL as plain 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)
    - ...

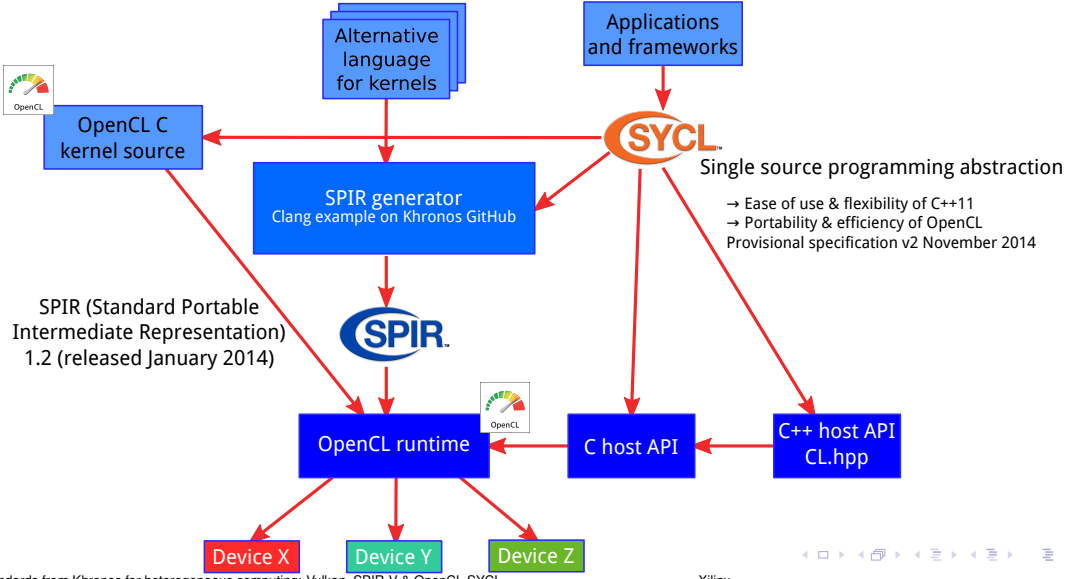
Debugging

- 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

Using SYCL-like models in other areas

- SYCL $\equiv$ generic heterogeneous computing model beyond OpenCL
  - `device` abstracts the accelerators
  - `queue` allows to launch tasks with computations overlapping communications and pipelining
  - `parallel_for` launches computations
  - `accessor` defines the way we access data
  - `buffer` to chose where to store data
  - `allocator` for defining how data are allocated/backed and how pointers work
- 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 such as transposition or relocation
  - ...

SYCL in OpenCL ecosystem



Known implementations

- ComputeCpp by Codeplay `https://www.codeplay.com/products/computecpp`
- SYCL-GTX open source `https://github.com/ProGTX/sycl-gtx`
- triSYCL open source `https://github.com/amd/triSYCL`

Parallel STL towards C++17 proposal

- Current Parallel STL from C++17 proposal N4507 (2015/05/05)
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.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 even be extended to give a kernel name (profile, debug...):
  - Load balancing between CPU and accelerator

```
sycl_policy<class kernelName1> pol;
sort(pol, begin(vec), end(vec));

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); });
```

Open Source https://github.com/KhronosGroup/SyclParallelSTL

Outline

SYCL 2.2 is coming!

Cope with OpenCL 2.2 architecture features

- Skip directly to OpenCL 2.2 and C++14
- Kernel side enqueue
- Shared memory between host and accelerator
- Pipes
- Parallel STL C++17
- Array TS

SYCL and fine-grain system shared memory (OpenCL 2)

```
#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 << "␣";
  std::cout << std::endl;
  return 0;
}
```

- This possible syntax could be 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

OpenCL SYCL committee work...

- Weekly telephone meeting
- Define new ways for modern heterogeneous computing with C++
  - Single source host + kernel
  - Replace specific syntax by pure C++ abstractions
- Write SYCL specifications
- Write SYCL conformance test
- Communication & evangelism

SYCL relies on advanced C++

- Latest C++11, C++14, C++1z...
- Metaprogramming
- Implicit type conversion
- ...

↝ Difficult to know what is feasible or even correct...

- Need a prototype to experiment with the concepts
- Double-check the specification
- Test the examples

Same issue with C++ standard and GCC or Clang/LLVM

triSYCL

`https://github.com/amd/triSYCL`

- Started in April 2014 as a side project
- Open Source for community purpose & dissemination
- Pure C++ implementation
  - DSEL (Domain Specific Embedded Language)
  - Rely on STL & Boost for zen style
  - Use OpenMP 3.1 to leverage CPU parallelism
  - No compiler ⇝ cannot generate kernels on GPU yet
- Use Doxygen to generate
  - Documentation of triSYCL implementation itself with implementation details
- Contributions from AMD, Xilinx, LaBRI, Qualcomm, University of West Scotland

What is (not) implemented

- All the small vectors range<>, id<>, nd_range<>, item<>, nd_item<>, group<>
- Parts of buffer<> and accessor<>
- Concepts of address spaces and vec<>
- Most of parallel constructs are implemented (parallel_for <>...)
  - Use OpenMP 3.1 for multicore CPU execution
  - Graph-based asynchronous execution based on <thread>, <condition_variable>, <atomic>, <mutex>...
- Most of command group handler is implemented
- No OpenCL feature is implemented yet
- No host implementation of OpenCL is implemented
  - No image<>
  - No OpenCL-like kernel types & functions

Future developments

- Develop the OpenCL layer
  - Rely on other high-level OpenCL frameworks (Boost.Compute...) ~~NIH~~
  - Should be able to have OpenCL kernels through the kernel interface with OpenCL kernels as a string (non single source kernel)
- Add outliner to Clang/LLVM OpenCL C++ compiler
  - Use open source OpenCL C++ compiler from Khronos
  - Use open source LLVM SPIR-V back-end from Khronos
  - Develop OpenCL SYCL runtime

Conclusion

- Khronos provide a comprehensive framework for heterogeneous computing
- Issues to solve are hard
  - Well accepted standard $\rightsquigarrow$ different implementations
  - Open source projects make dissemination and experiment easier
  - Open source implementations decrease entry cost...
    - $\exists$ Free tool to try
    - Can be used by vendors to develop their own tools
  - ... and decrease exit cost too
- SYCL $\equiv$ post-modern single source C++ for heterogeneous computing
  - Best of pure modern C++ + OpenCL interoperability + task graph model
  - Can be used to improve other higher-level frameworks
  - Pure C++ $\rightsquigarrow$ integration with other C/C++ HPC frameworks: OpenCL, OpenMP, libraries (MPI, numerical), C++ DSeL (PGAS...)...

Reaching out

- Send your good ideas for next standards
- Get involved into Khronos (cheaper for academics ☺)
- IWOCL 2016/04/19–21 `http://www.iwocl.org` Vienna, Austria (before next Khronos F2F)
- SYCL Workshop 2016/03 during PPoPP in Barcelona, Spain
  `http://conf.researchr.org/track/PPoPP-2016/SYCL-2016-papers`
- `http://www.meetup.com/Khronos-Paris-Chapter`
- Join for an internship at Xilinx or sabbatical on these subjects!
- It is time to cook again European project proposals!