Neil Trevett
Vice President Mobile Ecosystem at NVIDIA
President of Khronos and Chair of the OpenCL Working Group
GDC, March 2015
Khronos Connects Software to Silicon

Open Consortium creating ROYALTY-FREE, OPEN STANDARD APIs for hardware acceleration

Defining the roadmap for low-level silicon interfaces needed on every platform

Graphics, compute, rich media, vision, sensor and camera processing

Rigorous specifications AND conformance tests for cross-vendor portability

Acceleration APIs BY the Industry FOR the Industry

Well over a BILLION people use Khronos APIs Every Day...
OpenCL Desktop Usage

• Broad commercial uptake of OpenCL
  - Imaging, video, vision, simulation
  - Adobe, Apple, SONY, Corel, ArcSoft
  - Dassault, Houdini, Mathematica, MAYA...

• “OpenCL” on Sourceforge, Github, Google Code, Bitbucket finds over 2,000 projects
  - OpenCL implementations - Beignet, pocl
  - VLC, X264, FFmpeg, Handbrake
  - GIMP, ImageMagick, IrfanView
  - Hadoop, Memcached
  - WinZip, Crypto++ Etc. Etc.

• Desktop benchmarks use OpenCL
  - PCMark 8 - video chat and edit
  - Basemark CL, CompuBench Desktop

https://www.khronos.org/opencl/resources/opencl-applications-using-opencl
OpenCL on Mobile

- Mobile SOCs now beginning to need more than just ‘GPU Compute’
  - Multi-core CPUs, GPUs, DSPs
- OpenCL can provide a single programming framework for all processors on a SOC
  - Even ISPs and specialized hardware blocks with Built-in Kernels for custom HW

OpenCL is shipping today on multiple mobile processors and cores
OpenCL as Parallel Language Backend

- **WebCL**: JavaScript binding for initiation of OpenCL C kernels
- **Halide**: Language for image processing and computational photography
- **C++ AMP**: Accelerated Native Parallelism with Microsoft Visual C++
- **aparapi**: Embedded array language for Haskell
- **intel**: Compiler directives for Fortran, C and C++
- **OpenACC**: Directives for Accelerators
- **River Trail**: Language extensions to JavaScript
- **PyOpenCL**: Python wrapper around OpenCL
- **Harlan**: High level language for GPU programming

Approaching 200 languages, frameworks and projects using OpenCL as a compiler target to access vendor optimized, heterogeneous compute runtimes
OpenCL - Portable Heterogeneous Computing

- Portable Heterogeneous programming of diverse compute resources
  - Targeting supercomputers -> embedded systems -> mobile devices
- One code tree can be executed on CPUs, GPUs, DSPs, FPGA and hardware
  - Dynamically interrogate system load and balance work across available processors
- OpenCL = Two APIs and Kernel language
  - C Platform Layer API to query, select and initialize compute devices
  - C Runtime API to build and execute kernels across multiple devices
OpenCL 2.1 Provisional Released!

- New OpenCL C++ kernel language based on a subset of C++14
  - Significantly enhanced programmer productivity and code performance
  - OpenCL C still supported to preserve kernel code investment
- Support for the new Khronos SPIR-V intermediate language in core
  - SPIR-V now used by both OpenCL 2.1 and the new Vulkan graphics API
- Runs on any OpenCL 2.0-capable hardware
  - Only driver update required
SPIR-V Unleashes Language Innovation

- First cross-API, intermediate language for parallel compute and graphics
  - Can natively represent Vulkan and OpenCL source languages
  - Includes full flow control, graphics and parallel constructs not in LLVM
- Splitting compiler chain across vendors enables software/hardware innovation
  - Front-ends for languages can access multiple hardware run-times
  - Diverse hardware can reuse production quality language frontends
  - Tools for program analysis and optimization in SPIR form

SPIR-V is a significant convergence in the language ecosystem for graphics and parallel computation

SPIR-V is supported in core in both Vulkan and OpenCL 2.1

Diverse Languages and Frameworks

Tools for analysis and optimization

Standard Portable Intermediate Representation

Hardware runtimes on multiple architectures
### Evolution of SPIR

- **SPIR-V** is first fully specified Khronos-defined SPIR standard
  - Isolated from LLVM roadmap changes
  - Khronos will open source SPIR-V <-> LLVM conversion tools

<table>
<thead>
<tr>
<th></th>
<th>SPIR 1.2</th>
<th>SPIR 2.0</th>
<th>SPIR-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM Interaction</td>
<td>Uses LLVM 3.2</td>
<td>Uses LLVM 3.4</td>
<td>Does not use LLVM - 100% Khronos defined</td>
</tr>
<tr>
<td>Compute Constructs</td>
<td>Metadata/Intrinsics</td>
<td>Metadata/Intrinsics</td>
<td>Native</td>
</tr>
<tr>
<td>Graphics Constructs</td>
<td>No</td>
<td>No</td>
<td>Native</td>
</tr>
<tr>
<td>Supported Language Feature Set</td>
<td>OpenCL C 1.2</td>
<td>OpenCL C 1.2 OpenCL C 2.0</td>
<td>OpenCL C 1.2 OpenCL C 2.0 OpenCL C++ GLSL (under development)</td>
</tr>
<tr>
<td>OpenCL Ingestion</td>
<td>OpenCL 1.2 Extension</td>
<td>OpenCL 2.0 Extension</td>
<td>OpenCL 2.1 CORE</td>
</tr>
<tr>
<td>Vulkan Ingestion</td>
<td>-</td>
<td>-</td>
<td>Vulkan Core</td>
</tr>
</tbody>
</table>
New OpenCL 2.1 Compiler Ecosystem

- OpenCL C Kernel Source
- Diverse, domain-specific Languages, frameworks and tools
- SPIR Generator (e.g. patched Clang)
- OpenCL C++ Kernel Source
- OpenCL C++ to SPIR-V Compiler

SPIR-V is in core OpenCL 2.1
SPIR-V designed as compiler target

OpenCL 2.1 runtime can ingest OpenCL C OR SPIR-V

Device X  Device Y  Device Z

Khronos considering open source project for OpenCL C++ front-end

https://github.com/KhronosGroup/SPIR
Continue the Conversation at …

• Tutorials and Workshops
• Research Papers, Technical Papers and Posters
  - Promoting the evolution and advancement of OpenCL
  - Platforms: CPU, GPU, FPGA and Embedded
  - Applications: All uses of OpenCL, inc. SYCL and SPIR

Featuring OpenCL 2.1 Feedback Session

Sponsored by: KHRONOS GROUP

www.iwocl.org
Call to Action

• Khronos seeking feedback on Vulkan, SPIR and OpenCL 2.1
  - Links provided on Khronos forums
  - https://www.khronos.org/opencl/opencl_feedback_forum
  - https://www.khronos.org/spir_v_feedback_forum
  - https://www.khronos.org/vulkan/vulkan_feedback_forum

• Any company or organization is welcome to join Khronos for a voice and a vote in any of these standards
  - www.khronos.org
# Speakers

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
<th>Company</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neil Trevett</td>
<td>OpenCL Chair</td>
<td>NVIDIA</td>
<td>OpenCL Ecosystem</td>
</tr>
<tr>
<td>Lee Howes</td>
<td>Senior Staff Engineer</td>
<td>Qualcomm</td>
<td>OpenCL 2.1 and OpenCL C++</td>
</tr>
<tr>
<td>Raun Krisch</td>
<td>Staff Engineer</td>
<td>Intel</td>
<td>SPIR-V</td>
</tr>
<tr>
<td>Colin Riley</td>
<td>Games Technology Director</td>
<td>Codeplay</td>
<td>SYCL</td>
</tr>
<tr>
<td>Laszlo Kishonti</td>
<td>CEO</td>
<td>Kishonti</td>
<td>Compute Benchmarking</td>
</tr>
<tr>
<td>All</td>
<td></td>
<td></td>
<td>Wrap-up and Questions</td>
</tr>
</tbody>
</table>
OpenCL 2.1 API
GDC 2015
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
OpenCL 2.1 API Create From IL

- Three defined forms of program creation:
  - From source
  - From binary
  - From IL

- IL is a well-defined input for a platform
  - Primarily SPIR-V

- Fully defined API support for the SPIR-V intermediate language
  - SPIR-V input generated offline, or through a runtime compiler library interface
  - IL is the primary input method for OpenCL kernels
OpenCL 2.1 API Subgroups

- Subgroup functionality moved into core

- Expose hardware threads/warps/wavefronts and their cross-lane operations

- Host queries for forward progress extension, and workgroup->subgroup mapping
  - Enables analysis of concurrency guarantees from the host for correct algorithm construction
OpenCL 2.1 API Clone Kernel

• Full copying of kernel objects is not possible in OpenCL 2.0
  - Either copy as a reference
  - Or recreate the kernel from scratch

• Recreation is imperfect
  - A caller has to maintain a lot of separate state and reconstruct the kernel object

• clCloneKernel copies a kernel object to produce an identical clone
  - Useful to pass to a second host thread, for example
  - Or when writing C++ wrappers for kernel objects
OpenCL 2.1 API Low-latency Timer

- Associate clocks between host and OpenCL devices

- Query host and device timer at a given point in time to relate device and host timers

- Profile when commands are run, even in the presence of high latency of dispatch

- Regularly query device timer maintaining relationship to host timer
Priority and throttle hints

- **KHR extension for OpenCL 2.1**

- **Three priorities for each queue**
  - Ensures that commands from a higher priority queue execute before those from a low priority queue where possible

- **Three throttle levels for each queue**
  - Provides a hint to the runtime to judge DVFS settings or other performance/power tradeoffs
OpenCL C++

• A static subset of C++14
  - Frees developers from low-level coding details without sacrificing performance

• Classes, templates, operator overloading etc..
  - Reusable device libraries and containers - fast and elegant sharable code
  - Templates enables meta-programming for highly adaptive software
  - Lambdas used to implement nested/dynamic parallelism

• A standard library optimized for data-parallel programming
  - Leverages C++11 standard library features
    - Atomics, meta-programming & type traits, math functions...
  - New library features
    - Work-item & Work-group functions, Dynamic parallelism, Image & Pipe functions...
Restrictions

- The dynamic_cast operator
- Type identification
- Recursive function calls
- Operators: new, delete, noexcept
- goto
- Storage qualifiers: register, thread_local
- Function pointers
- Virtual functions and abstract classes
- Exception handling
- C++ standard library
A Simple C++ Kernel

#include <opencl_stdbase>
using namespace cl;

template<typename T>
void add_vectors(const T* srcA, const T* srcB, T* dst)
{
    size_t id = get_global_id(0);
    dst[id] = srcA[id] + srcB[id];
}

kernel void
add_vectors_float(const float* srcA, const float* srcB, float* dst)
{
    add_vectors<float>(srcA, srcB, dst);
}

kernel void
add_vectors_float4(const float4* srcA, const float4* srcB, float4* dst)
{
    add_vectors<float4>(srcA, srcB, dst);
}
OpenCL C++ Data Types

- **Scalars**
  - bool, char, short, int, float etc.
  - size_t, ptrdiff_t, intptr_t, uintptr_t, void

- **Vectors**
  - Vectors of 2, 3, 4, 8 or 16 boolean, integer or floating-point data types
  - Examples: int2, short4, float8, half3, bool4

- **Images**
  - Now template
    - `image2d<typename T, image_access::a>`
  - T specifies the color type returned by image reads or input to image writes
  - image_access can be sample, read, write or read_write
  - Example: `image2d<float, image_access::read>`

- **Sampler**
OpenCL C++ Data Types

- **Pipe**
  - Host constructed:
    - `pipe<typename T, pipe_access::a>`
  - New program-scope pipes:
    - `program_pipe<typename T>`

- **Types used for enqueuing kernels**
  - queue, event and ndrange
OpenCL C++ Address Spaces

- OpenCL C specifies *global*, *local*, *constant* and *private* type qualifiers

- OpenCL C++ 2.1 does not use these address space qualifiers
  - Pointers also refer to allocations in the generic address space

- For local memory allocations, the following types must be used:
  - `local_ptr<typename T>`
  - `local_array<typename T, size_t N>`
  - `local<T>`

- For constant memory allocations, the following types must be used:
  - `constant_ptr<typename T>`
  - `constant_array<typename T, size_t N>`
  - `constant<T>`
OpenCL C++ Standard Library

- Work-Item Functions: local id queries etc.

- Math Functions
  - Single-, double- and half- precision floating-point

- Atomics: a subset of the C++11 atomics library

- Image Functions: image reads with a sampler, sampler-less reads and writes

- Work- & Sub-group Functions: barrier, broadcast, scan, reduction, all and any

- Pipe Functions: read, write, reserve, commit, num_packets

- Dynamic Parallelism: Functions for enqueuing kernels on the device

- C++11 type traits with additions
Nested/Dynamic Parallelism

• Kernels can independently launch work on the device
  - without host interaction

• Kernels can enqueue:
  - a kernel function or
  - code represented as a kernel lambda function

• Control execution order with event dependencies (user events or markers)

• A kernel lambda function is described as:
  - [ capture-list ] ( params ) kernel { body }
Dynamic Parallelism - Data Types

• Queue
  - A device queue that you can use to enqueue kernels

• Event
  - Identifies a specific kernel enqueued on the device, a user-event or a marker
  - Can be used for the following:
    - Control order of execution of enqueued kernels
    - Capture profiling information

• ND-range
  - Specifies the size of data-parallel work-load over which an enqueued kernel executes
  - Global work size, local work size
  - 1-, 2- or 3- dimensional work sizes
# Dynamic Parallelism - A Simple Example

```cpp
#include <opencl_stdlib>
using namespace cl;

void
do_something(int *a, int *b, int *c)
{
    ...
}

kernel void
my_kernel(int *a, int *b, int *c)
{
    ndrange range(get_global_size(0));

    auto x = [=] kernel { do_something(a, b, c); }
    queue q = get_default_queue();

    // enqueue the kernel
    q.enqueue_kernel(CLK_ENQUEUE_FLAGS_WAIT_KERNEL, ndrange, x);
}
```
Dynamic Parallelism – Another Example

#include <opencl_stdlib>
using namespace cl;

extern void barA_kernel(...);
extern void barB_kernel(...);

kernel void
foo(queue q0, queue q1, ...)
{
... 
clk_event_t evt0;
// enqueue kernel to queue q0
q0.enqueue_kernel(CLK_ENQUEUE_FLAGS_NO_WAIT, 
    range_A, 
    0, NULL, &evt0, 
    [=] kernel {barA_kernel(...);});

// enqueue kernel to queue q1
q1.enqueue_kernel(CLK_ENQUEUE_FLAGS_NO_WAIT, 
    range_B, 
    1, &evt0, NULL, 
    [=] kernel {barB_kernel(...);});

// release event evt0. This will get released after barA_kernel enqueued in queue q0 has
// finished execution and barB_kernel enqueued in queue q1 which waits for evt0 is submitted
// for execution i.e. wait for evt0 is satisfied.
evt0.release();
}
Compile-time Dataflow

- **program_pipe**
  - A pipe object declared in program scope

- **Compile-time dataflow**
  - Program_pipe connectivity known at compile time (vs at run-time, host allocated and host connected pipes)

- **Enables exploitation of device specific features**
  - Localized interconnects and memory structures (bounded reach and/or visibility)
  - Higher performance / lower power

- **Declaration**
  - `program_pipe<T>::program_pipe(size_t max_packets, const pipe_properties *properties=null)`
Compile-time Dataflow - An Example

```cpp
program_pipe<float2> myGlobalPipe1(200);

kernel void pipeline()
{
    event evt;

    q.enqueue_kernel(flags, ndrange, 0, null, &evt,
                     [=] {
            // produce
            pipe<int, pipe_access::write> p(myGlobalPipe1);
            p.write_pipe(...);
        });

    q.enqueue_kernel(flags, ndrange, 1, &evt, null,
                     [=] {
            // consume
            pipe<int, pipe_access::read> p(myGlobalPipe1);
            p.read_pipe(...);
        });
}
```
Goal:
1) Portable binary representation of shader programs and compute kernels for GPUs and parallel computers
2) Target for OpenCL C/C++, GLSL, and other compute kernel languages

Enables compiler ecosystem for portable parallel programs
What is SPIR-V?

- Intermediate language for input to Khronos graphics and compute APIs
  - Fully specified Khronos-defined standard
  - Can natively represent Khronos graphics and compute source languages
  - Allows for conversion to/from LLVM IR
    - Khronos is working on creating LLVM <-> SPIR-V conversion tools
      (seeking feedback - is this useful?)

- Core for OpenCL 2.1
  - Exposes machine model for OpenCL 1.2, 2.0, 2.1
  - Supports OpenCL 1.2, 2.0, 2.1 kernel languages

- Core for Vulkan
  - Exposes machine model for Vulkan
  - Supports GLSL shader language
  - Note: graphics portion of the spec under development
Why use SPIR?

Without SPIR:

- Vendors shipping source
  - Risk IP leakage
- Limited Portability
  - No ISV control over Front end
  - Different Front end implementation per vendor
- Higher runtime compilation time
- Vendors shipping multiple binaries
  - Complexity
  - Miss optimizations in new compilers
  - Forward compatibility issues

With SPIR:

- Ship a single binary
- Improved Portability
  - ISV can create their own Front end tool chain
  - Can use common Front end across vendors
- Reduced runtime compilation time
- Shipped application can retarget new devices and new vendors

Opportunity to unleash innovation:

Domain Specific Languages, C++ Compilers, Halide, ....
SPIR-V kernel language support

• Full support for OpenCL “C” and “C++” kernel languages
  - Memory and execution models support OpenCL up to v2.1
  - Generic address space
  - Device side kernel enqueue
  - C++11 atomics
  - Pipes
  - More...

• Shading language support in development for Vulkan
  - Adds graphics-specific instructions and resources
  - Work in progress - watch this space!

• Compiler chain split in two
  - Front end compiler emits SPIR-V portable binary IL
  - SPIR-V IL is compiled to machine-specific binary by driver

• OpenCL C++ front end NOT required in driver
  - Khronos working on a C++ kernel language off-line front end
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)
SPIR-V Introduction

• A Binary Intermediate Language
  - A linear stream of words (32-bits)

• Functions inside a module contain a CFG (control-flow graph) of basic blocks
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)
- Functions inside a module contain a CFG (control-flow graph) of basic blocks
- Load/Store instructions are used to access declared variables
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)
- Functions inside a module contain a CFG (control-flow graph) of basic blocks
- Load/Store instructions are used to access declared variables
- Intermediate results are represented using single static-assignment (SSA)
SPIR-V Introduction

• A Binary Intermediate Language
  - A linear stream of words (32-bits)

• Functions inside a module contain a CFG (control-flow graph) of basic blocks

• Load/Store instructions are used to access declared variables

• Intermediate results are represented using single static-assignment (SSA)

• Data objects are represented logically, with hierarchical type information
  - e.g. No flattening of aggregates or assignment to physical registers

```c
struct {
    mat3x4;
    vec4[6];
    int;
};
```
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)

- Functions inside a module contain a CFG (control-flow graph) of basic blocks

- Load/Store instructions are used to access declared variables

- Intermediate results are represented using single static-assignment (SSA)

- Data objects are represented logically, with hierarchical type information
  - e.g. No flattening of aggregates or assignment to physical registers

```c
struct {
    mat3x4;
    vec4[6];
    int;
};
```
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)
- Functions inside a module contain a CFG (control-flow graph) of basic blocks
- Load/Store instructions are used to access declared variables
- Intermediate results are represented using single static-assignment (SSA)
- Data objects are represented logically, with hierarchical type information
  - e.g. No flattening of aggregates or assignment to physical registers
- Selectable addressing model
  - Allow usage of pointers, or dictate a memory which is purely logical.
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)
- Functions inside a module contain a CFG (control-flow graph) of basic blocks
- Load/Store instructions are used to access declared variables
- Intermediate results are represented using single static-assignment (SSA)
- Data objects are represented logically, with hierarchical type information
  - e.g. No flattening of aggregates or assignment to physical registers
- Selectable addressing model
  - Allow usage of pointers, or dictate a memory which is purely logical.
- Can be easily extended

```
1: ExtInstImport "GLSL.std.450" // declare use
...
40: 8 Load 39                   // load a vec4
41: 8 ExtInst 1(GLSL.std.450) 28(sqrt) 40 // take its sqrt
```
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)
- Functions inside a module contain a CFG (control-flow graph) of basic blocks
- Load/Store instructions are used to access declared variables
- Intermediate results are represented using single static-assignment (SSA)
- Data objects are represented logically, with hierarchical type information
  - e.g. No flattening of aggregates or assignment to physical registers
- Selectable addressing model
  - Allow usage of pointers, or dictate a memory which is purely logical.
- Can be easily extended
- Support debug information that can be safely stripped without changing the semantics of SPIR-V modules.
SPIR-V Introduction

- A Binary Intermediate Language
  - A linear stream of words (32-bits)
- Functions inside a module contain a CFG (control-flow graph) of basic blocks
- Load/Store instructions are used to access declared variables
- Intermediate results are represented using single static-assignment (SSA)
- Data objects are represented logically, with hierarchical type information
  - e.g. No flattening of aggregates or assignment to physical registers
- Selectable addressing model
  - Allow usage of pointers, or dictate a memory which is purely logical.
- Can be easily extended
- Support debug information that can be safely stripped without changing the semantics of SPIR-V modules.
- Khronos is working on creating LLVM <-> SPIR-V conversion tool
SPIR Reference Flow

Generation

Device program source \(\rightarrow\) SPIR Generator \(\rightarrow\) Standard Portable Intermediate

Consumption

Platform specific container

Standard Portable Intermediate \(\rightarrow\) OpenCL Runtime \(\rightarrow\) cl_spir

OpenCL
SPIR Today

**Generation**

- **Device program source** ➔ **OpenCL C/C++** ➔ **Patched Clang** ➔ **SPIR Generator** ➔ **Standard Portable Intermediate**

**Consumption**

- **Platform specific container** ➔ **Standard Portable Intermediate** ➔ **OpenCL Runtime** ➔ **OpenCL** ➔ **cl_spir**

*OpenCL C/C++* and *Patched Clang* are tools used in the generation process, translating device program source code into a standard portable intermediate format. The intermediate format is then handled by the *SPIR Generator*, which produces a standard portable intermediate code. Finally, the *OpenCL Runtime* compiles and runs the code on the target platform.
Call to Action

• Seeking feedback now on SPIR-V provisional
  - A Provisional specification, subject to change based on your feedback
  - Spec available at www.khronos.org/spir
  - Provide feedback at https://www.khronos.org/spir_feedback_forum

• Innovate on the Front end
  - New languages, abstractions
  - Target production quality Back ends

• Innovate on the Back end
  - New target platforms: Multi core, Vector, VLIW...
  - Reuse production quality frontends

• Innovate on Tooling
  - Program analysis, optimization
What is SYCL?

- SYCL
  - Pronounced SICKLE

- Royalty-free, cross platform C++ programming layer
  - Builds on concepts of portability and efficiency

- A Higher Level Model on top of OpenCL
What is SYCL?

• Single source C++
  - Without language extensions

• Single Source Development Cycle
  - C++ source contains functions which can be on both host and device
  - Allows to construct reusable templates for algorithms which use OpenCL for acceleration
The state of play

- 1.2 Provisional Specification was released last year
- Aim is for ratification this year

Provisional specification available at:

https://www.khronos.org/opencl/sycl

Khronos forums have a SYCL thread for feedback
What does SYCL achieve?

- An OpenCL ecosystem based on C++

- C++ developers can easily utilize OpenCL
  - Ability to ship C++ libraries, tools with OpenCL support
  - Implementations from multiple sources

- Enabling future innovation
Overview

User application code

C++ template libraries

SYCL for OpenCL

OpenCL Devices

CPU

GPU

DSP

FPGA

Other technologies

CPU

Custom Processor
Example SYCL code

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

int main ()
{
    ....
    // Device buffers
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));
    queue myQueue;
    command_group(myQueue, [&]()
    {
        // Data accessors
        auto a = buf_a.get_access<access::read>();
        auto b = buf_b.get_access<access::read>();
        auto c = buf_c.get_access<access::read>();
        auto r = buf_r.get_access<access::write>();
        // Kernel
        parallel_for<class three_way_add>(count, [=](id<> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
    ....
}
```
Example SYCL code

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

int main ()
{
    ...
    // Device buffers
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));
    queue myQueue;
    command_group(myQueue, [&]()
    {
        // Data accessors
        auto a = buf_a.get_access<access::read>();
        auto b = buf_b.get_access<access::read>();
        auto c = buf_c.get_access<access::read>();
        auto r = buf_r.get_access<access::write>();
        // Kernel
        parallel_for<class three_way_add>(count, [=](id<> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
    ...
}
```

Some language restrictions within kernels
Build Process Overview

#include <CL/sycl.hpp>

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

    command_group(myQueue, [&]() {
        // Data accessors
        auto a = buf_a.get_access<access::read>();
        auto b = buf_b.get_access<access::read>();
        auto c = buf_c.get_access<access::read>();
        auto r = buf_r.get_access<access::write>();

        // Kernel
        parallel_for<class three_way_add>(count, [=](id<> i) {
            r[i] = a[i] + b[i] + c[i];
        });
    });
}
Major talking points

- SYCL uses standard C++ with no language extensions

- Tasks, such as memory object creations, mapping, copies and kernel execution, is scheduled automatically
  - Using SYCL allows for dependencies to be tracked automatically
  - Specifying data access rules reduces overheads, allows for efficient scheduling

- Hierarchical Parallelism
const int n_items = 32;
range<1> r(n_items);
int array_a[n_items] = { 0 };  
int array_b[n_items] = { 0 };  
buffer<int, 1> buf_a(array_a, range<1>(r));
buffer<int, 1> buf_b(array_b, range<1>(r));

queue q;
command_group(q, [&]()
{
    auto acc_a = buf_a.get_access<read_write>();
    algorithm_a s(acc_a);
    parallel_for(n_items, s);
});
command_group(q, [&]()
{
    auto acc_b = buf_b.get_access<read_write>();
    algorithm_b s(acc_b);
    parallel_for(n_items, s);
});
command_group(q, [&]()
{
    auto acc_a = buf_a.get_access<read_write>();
    algorithm_c s(acc_a);
    parallel_for(n_items, s);
});
Hierarchical Data Parallelism

buffer<int> my_buffer(data, 10);
auto in_access = my_buffer.access<cl::sycl::access:read_only>();
auto out_access = my_buffer.access<cl::sycl::access:write_only>();

command_group(my_queue, [&]()
{
  parallel_for_workgroup(nd_range(range(size), range(groupsize)),
    lambda<class hierarchical>(=[group_id group])
    {
      parallel_for_workitem(group, [=](tile_id tile)
      {
        out_access[tile] = in_access[tile] * 2;
      });
    });
});

Advantages:
1. Easy to understand the concept of work-groups
2. Performance-portable between CPU and GPU
3. No need to think about barriers (automatically deduced)
4. Easier to compose components & algorithms
What does this mean for developers?

• Can have a standard C++11 codebase targeting multiple OpenCL devices

• SYCL is cross-toolchain as well as cross-platform
  - No language extensions, standard C++ compilers can build SYCL source code

• Device compilers then enable SYCL running on OpenCL devices
  - Can have multiple device compilers linking into final executable
  - Implementation defined
  - Doesn’t affect original source build
What does this mean for developers?

• As it’s C++, a host CPU device implementation can be provided in headers.

• You could implement one simply using C++ threads
  - No external dependencies

• All the synchronization, parallelism wins remain
  - But running on CPU

• Allows for developers to move quickly into writing SYCL code
  - Provides methods for dealing with targets that do not have OpenCL(yet!)
  - Has other development benefits...
What does this mean for developers?

- A fallback CPU implementation is debuggable!
- Using normal C++ debuggers
- Profiling tools also work on CPU device
- Huge bonus for productivity and adoption
  - Cost of entry to use SYCL very low
OpenCL features within SYCL

- Can access OpenCL objects from SYCL objects
- Can construct SYCL objects from OpenCL object

- Interop with OpenGL remains in SYCL
  - Uses the same structures/types

- Developers still have the ability to optimize at a low level should they need to
In Summary

- SYCL: a royalty-free, cross platform C++ programming layer

- An OpenCL ecosystem based on standard C++
  - Single source development without language extensions
  - C++ developers can easily utilize OpenCL in code

- Very low cost to entry

- Features described today
  - Task Graphs, expressive parallelism in code
  - Host fallback enables tools such as debuggers/profilers to work on CPUs
  - Ability to fine tune code where necessary and use OpenCL interop
SYCL for OpenCL

• Provisional available online
  - https://www.khronos.org/opencl/sycl

• Please use the SYCL forum thread for feedback!
  - http://www.khronos.org/opencl/sycl

• Implementations
  - AMD - TriSYCL - https://github.com/amd/triSYCL CPU based test using OpenMP
  - Codeplay have an Implementation in development

• Next steps
  - Full specification, based on feedback
  - Conformance testsuite to ensure compatibility between implementations
  - Release of implementations