XILINX
ALL PROGRAMMABLE™

SYCL C++ for heterogeneous computing

From single-source modern C++ down to FPGA

Ronan Keryell @ LTP-LaMHA workshop

rkeryell at xilinx dot com

Xilinx Research Labs

11/10/2017

Power wall & speed of light: the final frontier...

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

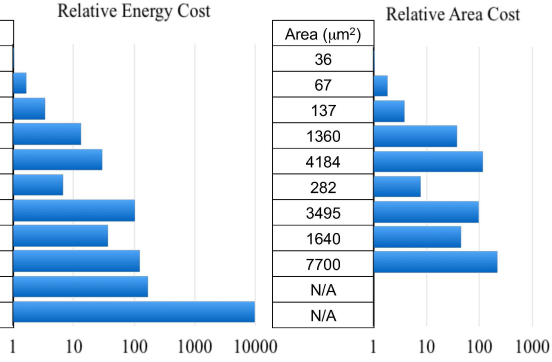
(Rather old) 45nm technology characteristics

Tutorial on "*High-Performance Hardware for Machine Learning*", William Dally at NIPS, December 7th, 2015

`https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf`

Cost of Operations

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

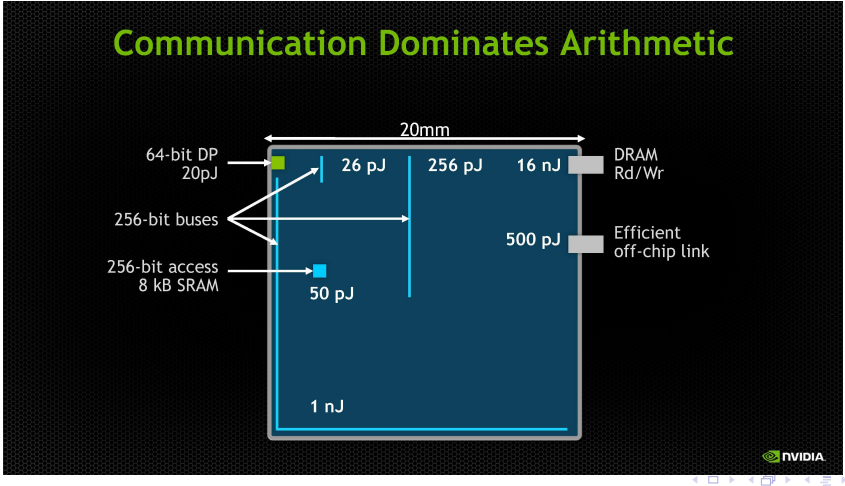


Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Space-time traveling

"*Challenges for Future Computing Systems*", William J. Dally, January 19, 2015, HiPEAC 2015.

`http://www.cs.colostate.edu/~cs575dl/Sp2015/Lectures/Dally2015.pdf`



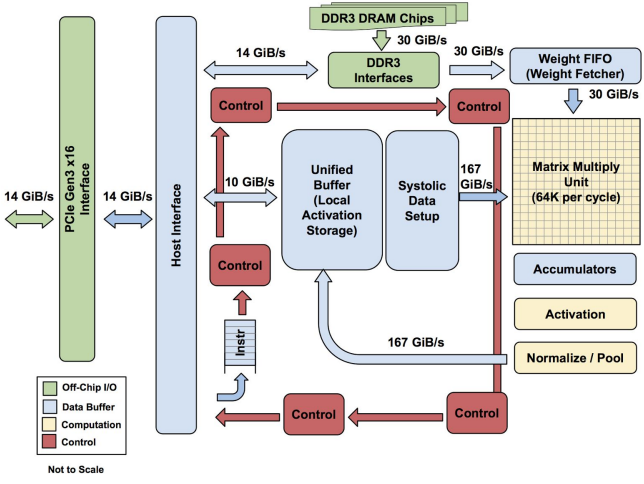
Power wall & speed of light: implications

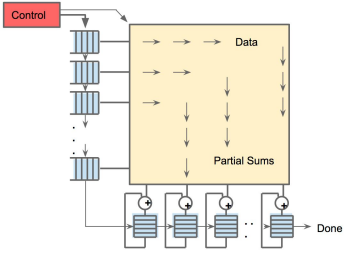
- Change hardware and software
  - Use locality & hierarchy
  - Massive parallelism
- NUMA & distributed memories
  - ⇝ New memory address spaces (local, constant, global, non-coherent...)
  - ⇝ PiM (Processor-in-Memory), Near-Memory Processing (in 3D memory stack controller)...
- Specialize architecture
- Power on-demand only what is required

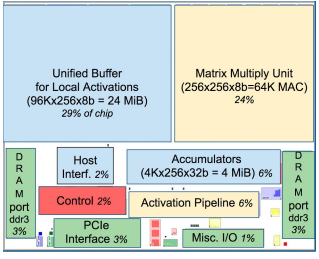
Nice take-away: the battery limitation may produce better programmers in the future ☺

From AMD Fiji XT GPU (2015)...

... Google Tensor Processing Unit (TPU)

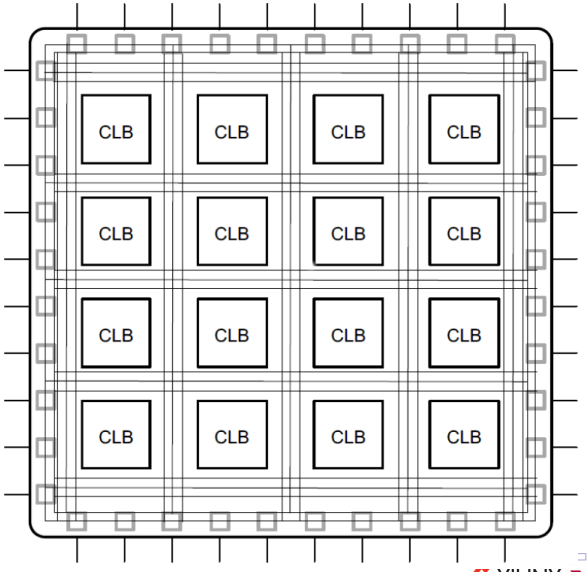




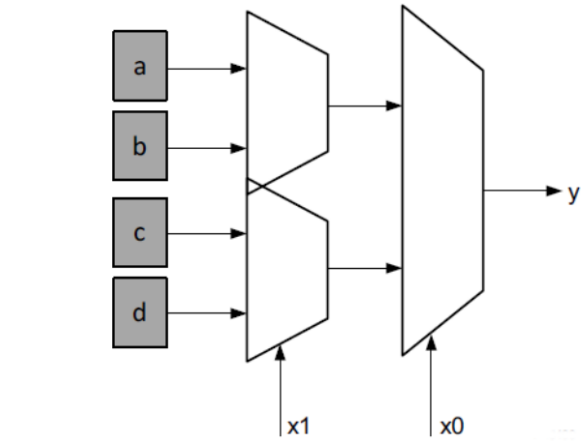


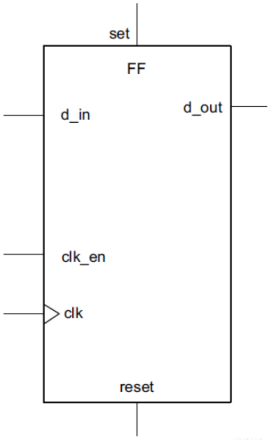
https://en.wikipedia.org/wiki/Tensor_processing_unit

... to Field-Programmable Gate Array (FPGA)



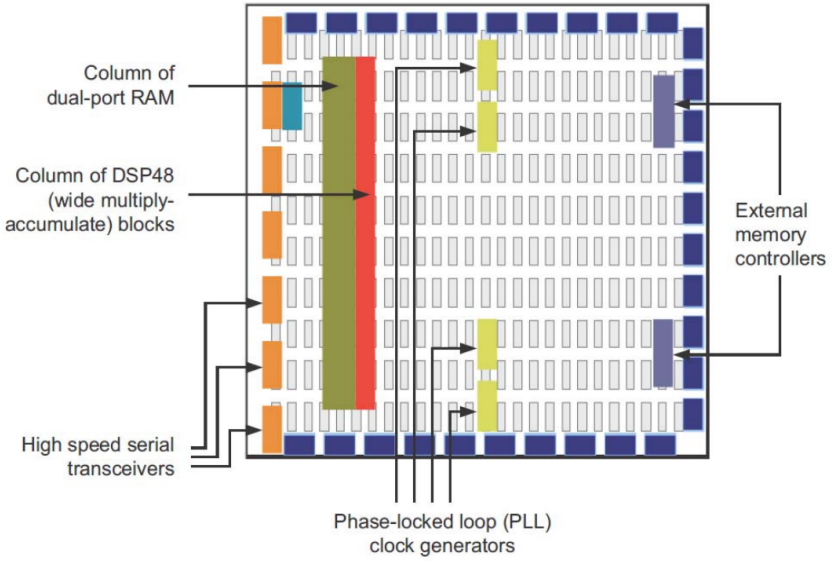
Basic architecture = Lookup Table + Flip-Flop storage + Interconnect



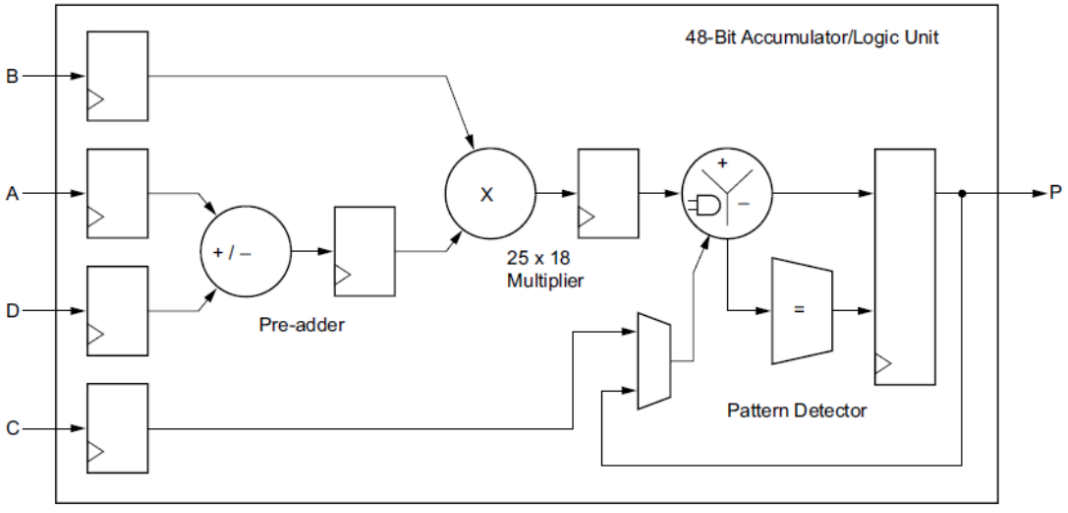


Typical Xilinx LUT have 6 inputs

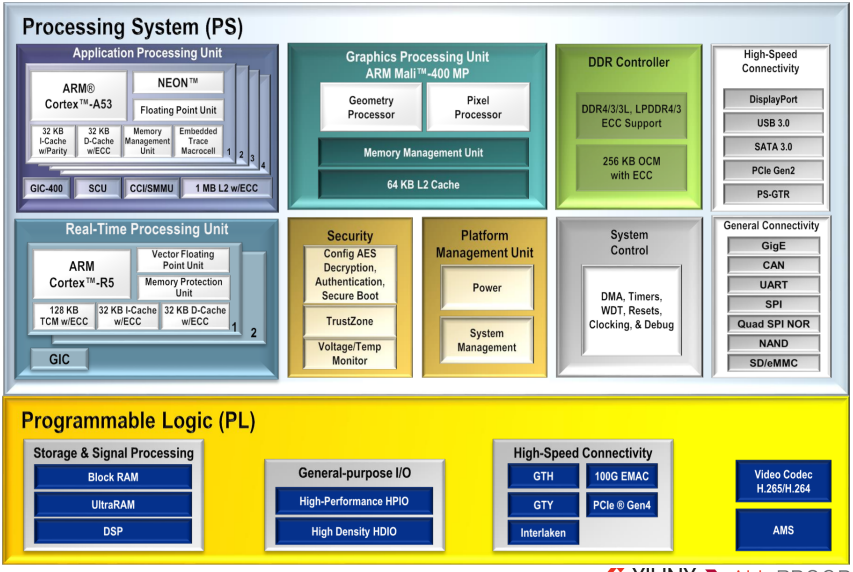
Global view of programmable logic part



DSP48 block overview



...FPGA in MPSoC: Xilinx Zynq UltraScale+ MPSoC

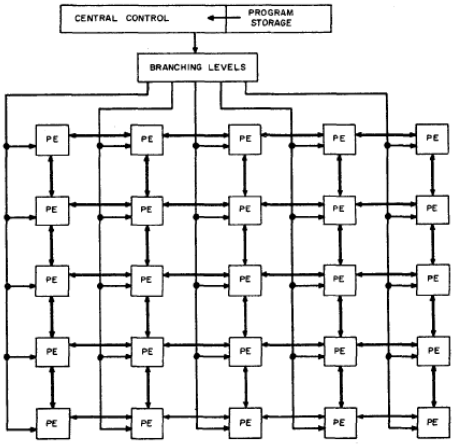


Xilinx Zynq UltraScale+ MPSoC programming

- Vivado
  - Hardware basic block integration
  - RTL (Verilog & VHDL) programming
- Vivado HLS
  - C & C++ high-level synthesis down to LUT, DSP & BRAM...
- SDAccel
  - OpenCL
- SDSoC
  - C & C++ with #pragma
- SDNet
  - Generate routers from network protocol description
- Various libraries
  - OpenCV, DNN...
- Linux
  - Usual CPU multicore programming

Not a new problem...

- SOLOMON
  - Daniel L. Slotnick. « The SOLOMON computer. » *Proceedings of the December 4-6, 1962, fall joint computer conference*. p. 97–107. 1962
  - Target application: *“data reduction, communication, character recognition, optimization, guidance and control, orbit calculations, hydrodynamics, heat flow, diffusion, radar data processing, and numerical weather forecasting”*
  - Diode + transistor logic in 10-pin TO5 package



But now not only for computing performance...

Outline

Connecting Software to Silicon

COLLADA

3D for the Web
- Real-time apps and games in-browser
- Efficiently delivering runtime 3D assets

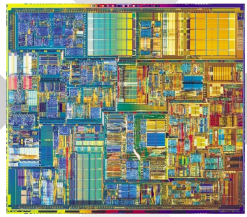
WebGL glTF

NNEF

OpenVX

Vision and Neural Networks
- Tracking and odometry
- Scene analysis/understanding
- Neural Network inferencing

Parallel Computation
- Machine Learning acceleration
- Embedded vision processing
- High Performance Computing (HPC)



OpenCL

SPIR SYCL

OpenGL|SC OpenVG

Vulkan OpenGL|ES

OpenGL

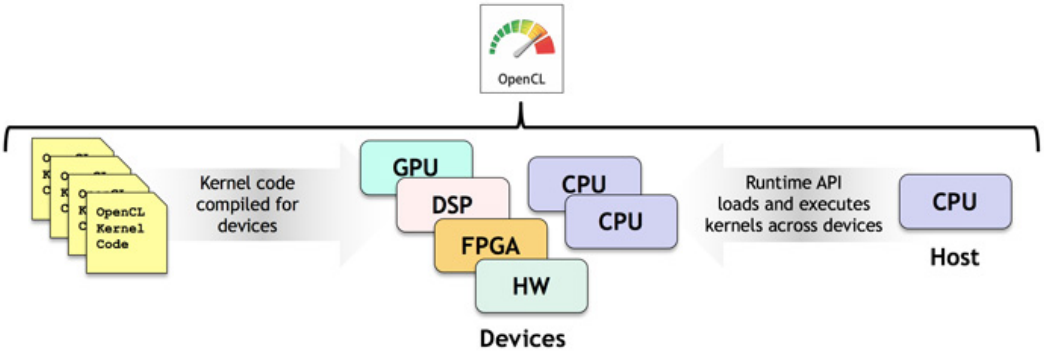
'VR Initiative'

Real-time 2D/3D
- Virtual and Augmented Reality
- Cross-platform gaming and UI
- CG Visual Effects
- CAD and Product Design
- Safety-critical displays

KHRONOS GROUP

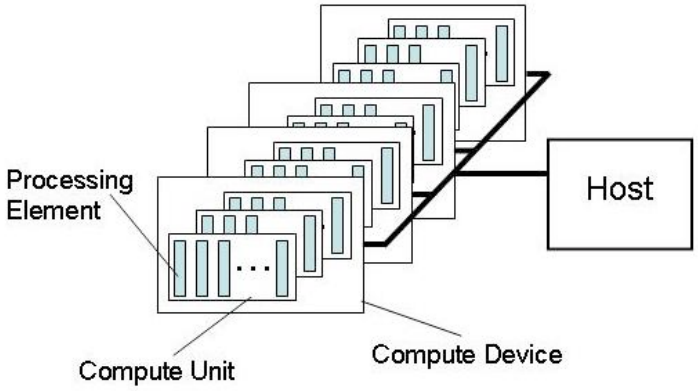
Outline

OpenCL



- OpenCL: 2 APIs and 2 kernel languages
  - C Platform Layer API to query, select and initialize compute devices
  - OpenCL C 2.0 and OpenCL C++ 2.2 kernel languages to write separate parallel code
  - C Runtime API to build and execute kernels across multiple devices
- One code tree can be executed on CPU, GPU, DSP, FPGA...

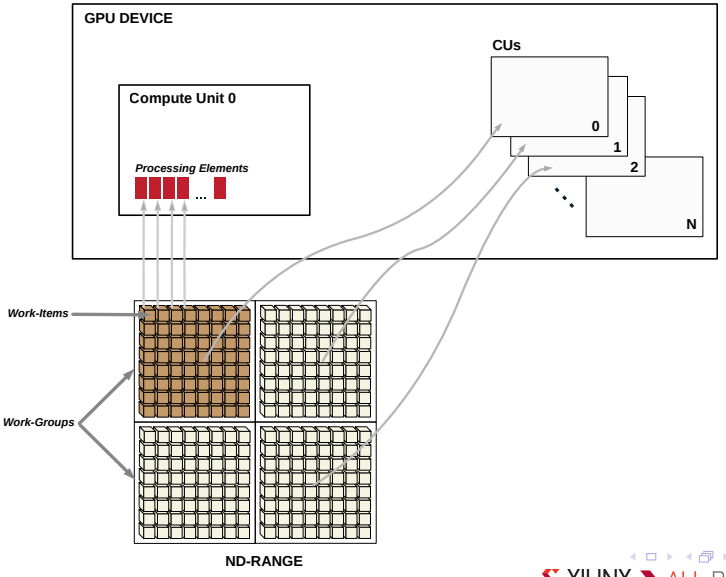
Architecture model



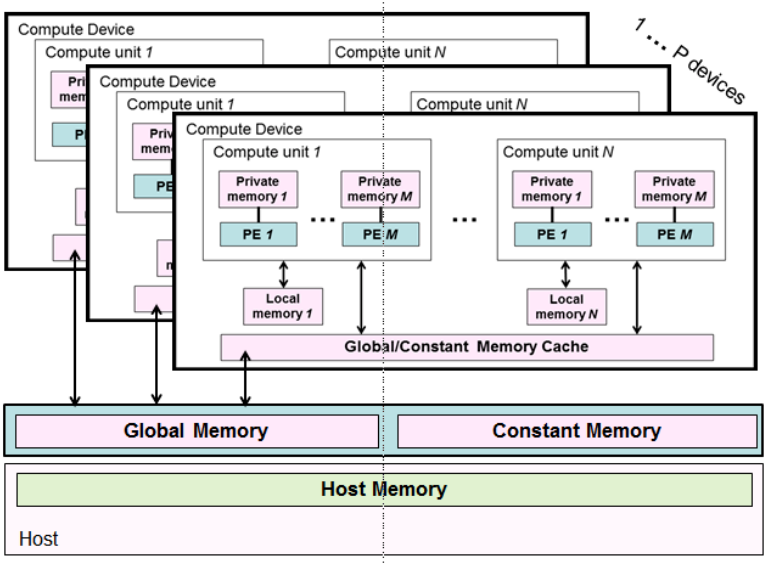
- Host threads launch computational *kernels* on *accelerators*

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

Execution model



Memory model



Vector add (≈ Hello World) in C++ using OpenCL C host API (I)

$$\vec{c} = \vec{a} + \vec{b}$$

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  #if defined(__APPLE__)
6  #include <OpenCL/cl.h>
7  #else
8  #include <CL/cl.h>
9  #endif
10
11 /* Transform the value of a given symbol to a string. Since we expect a
12    macro symbol, use a double evaluation... */
13 #define _stringG(s) #s
14
15 #define _stringifY(s) _stringG(s)
16
17 /** Throw a nicer error message in the code by adding the file name and
18     the position */
19 #define THROW_ERROR(message)                                          \
20   throw std::domain_error(std::string("In file " __FILE__ " at line "   \
21                                       _stringifY(__LINE__) "\n") + message)
```

Vector add (≈ Hello World) in C++ using OpenCL C host API (II)

```
22
23  /** Test for an OpenCL error and display a message */
24  #define OCL_TEST_ERROR_MSG(status, msg) do {                              \
25      if ((status) !=  CL_SUCCESS)                                          \
26        THROW_ERROR(std::string(msg) + std::to_string(status));             \
27    } while(0)
28
29    /** Do an OpenCL function call and test for execution error */
30  #define OCL_ERROR(func) do {                                              \
31      cl_int _st = func;                                                    \
32      if (_st !=  CL_SUCCESS)                                               \
33        THROW_ERROR(_stringifY(func) " returns error " + std::to_string(_st)); \
34    } while(0)
35
36  constexpr size_t N = 3;
37
38  using Vector = float[N];
39
40  int main() {
41    Vector a = { 1, 2, 3 };
42    Vector b = { 5, 6, 8 };
43    Vector c;
44
45    cl_int status;
46
47    // Get the number of OpenCL platforms on the machine
```

Vector add (≈ Hello World) in C++ using OpenCL C host API *(III)*

```
48    cl_uint num_platforms;
49    OCL_ERROR(clGetPlatformIDs(0, NULL, &num_platforms));
50
51    std::vector<cl_platform_id> platforms(num_platforms);
52    OCL_ERROR(clGetPlatformIDs(num_platforms, platforms.data(), NULL));
53
54
55    cl_context context;
56    bool found_context = false;
57    for (auto platform : platforms) {
58      // Describe the context to query
59      cl_context_properties cps[] = {
60        CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
61        0
62      };
63      // Create an OpenCL context from our platform
64      context = clCreateContextFromType(cps,
65                                        CL_DEVICE_TYPE_ALL,
66                                        NULL,
67                                        NULL,
68                                        &status);
69      if (status ==  CL_SUCCESS) {
70        found_context = true;
71        break;
72      }
73    }
```

Vector add (≈ Hello World) in C++ using OpenCL C host API *(IV)*

```
74    if (!found_context)
75      THROW_ERROR("Cannot found a context");
76
77    // Get the first device
78    cl_device_id device;
79    OCL_ERROR(clGetContextInfo(context, CL_CONTEXT_DEVICES,
80                               sizeof(device), &device, NULL));
81
82    // Create an OpenCL command queue
83    cl_command_queue command_queue =
84      clCreateCommandQueueWithProperties(context, device, NULL, &status);
85    OCL_TEST_ERROR_MSG(status, "Cannot create the command queue");
86
87    // The input buffers for OpenCL
88     cl_mem buffer_a =
89       clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(a), NULL, &status);
90     OCL_TEST_ERROR_MSG(status, "Cannot create buffer_a");
91     cl_mem buffer_b =
92       clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(b), NULL, &status);
93     OCL_TEST_ERROR_MSG(status, "Cannot create buffer_b");
94
95    // The output buffer for OpenCL
96     cl_mem buffer_c =
97       clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(c), NULL, &status);
98     OCL_TEST_ERROR_MSG(status, "Cannot create buffer_c");
99
```

Vector add (≈ Hello World) in C++ using OpenCL C host API *(V)*

```
100     // Construct an OpenCL program from the source file
101     const char kernel_source[] = R"(
102   __kernel void vector_add(const __global float *a,
103                            const __global float *b,
104                            __global float *c) {
105     c[get_global_id(0)] = a[get_global_id(0)] + b[get_global_id(0)];
106   }
107   )";
108     const char *kernel_sources = kernel_source;
109     const size_t kernel_size = sizeof(kernel_source);
110     cl_program program = clCreateProgramWithSource(context, 1, &kernel_sources,
111                                                    &kernel_size, &status);
112     OCL_TEST_ERROR_MSG(status, "Cannot create program");
113
114     OCL_ERROR(clBuildProgram(program, 1, &device, "", NULL, NULL));
115
116     cl_kernel kernel = clCreateKernel(program, "vector_add", &status);
117     OCL_TEST_ERROR_MSG(status, "Cannot find the kernel");
118
119     // Send the input data to the accelerator
120     OCL_ERROR(clEnqueueWriteBuffer(command_queue, buffer_a, true, 0 /* Offset */,
121                                    sizeof(a), &a[0], 0, NULL, NULL));
122     OCL_ERROR(clEnqueueWriteBuffer(command_queue, buffer_b, true, 0 /* Offset */,
123                                    sizeof(b), &b[0], 0, NULL, NULL));
124
125     OCL_ERROR(clSetKernelArg(kernel, 0, sizeof(buffer_a), &buffer_a));
```

Vector add (≈ Hello World) in C++ using OpenCL C host API *(VI)*

```
126     OCL_ERROR(clSetKernelArg(kernel, 1, sizeof(buffer_b), &buffer_b));
127     OCL_ERROR(clSetKernelArg(kernel, 2, sizeof(buffer_c), &buffer_c));
128
129     // Launch the kernel
130     const size_t global_work_size { N };
131     OCL_ERROR(clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
132                                      &global_work_size, NULL,
133                                      0, NULL, NULL));
134
135     // Get the output data from the accelerator
136     OCL_ERROR(clEnqueueReadBuffer(command_queue, buffer_c, true, 0 /* Offset */,
137                                   sizeof(c), &c[0], 0, NULL, NULL));
138
139
140     std::cout << std::endl << "Result:" << std::endl;
141     for(auto e : c)
142       std::cout << e << " ";
143     std::cout << std::endl;
144   }
```

- Verbose but full control to allow construction of higher-level frameworks
- Not single-source yet
  - No type-safety between host and device code

C++ wrapper for OpenCL API from Khronos: CL/cl2.hpp *(I)*

https://github.com/KhronosGroup/OpenCL-CLHPP

```
1  #include <iostream>
2  #include <iterator>
3  #define CL_HPP_ENABLE_EXCEPTIONS
4  #include <CL/cl2.hpp>
5
6  constexpr size_t N = 3;
7  using Vector = float[N];
8
9  int main() {
10   Vector a = { 1, 2, 3 };
11   Vector b = { 5, 6, 8 };
12   Vector c;
13
14   // The input read-only buffers for OpenCL on default context
15   cl::Buffer buffer_a { std::begin(a), std::end(a), true};
16   cl::Buffer buffer_b { std::begin(b), std::end(b), true};
17   // The output buffer for OpenCL on default context
18   cl::Buffer buffer_c { CL_MEM_WRITE_ONLY, sizeof(c) };
19
20   // Construct an OpenCL program from the source file
21   const char kernel_source[] = R"(
```

C++ wrapper for OpenCL API from Khronos: CL/cl2.hpp (II)

```
22  __kernel void vector_add(const __global float *a,
23                           const __global float *b,
24                           __global float *c) {
25    c[get_global_id(0)] = a[get_global_id(0)] + b[get_global_id(0)];
26  }
27  )";
28    // Compile and build the program
29    cl::Program p { kernel_source, true };
30    // Create the kernel functor taking 3 buffers as parameter
31    cl::KernelFunctor<cl::Buffer, cl::Buffer, cl::Buffer> k { p, "vector_add" };
32
33    // Call the kernel with N work-items on default command queue
34    k(cl::EnqueueArgs(cl::NDRange(N)), buffer_a, buffer_b, buffer_c);
35
36    // Get the output data from the accelerator
37    cl::copy(buffer_c, std::begin(c), std::end(c));
38
39    std::cout << std::endl << "Result:" << std::endl;
40    for(auto e : c)
41      std::cout << e << " ";
42    std::cout << std::endl;
43  }
```

C++ wrapper for OpenCL API from Khronos: CL/cl2.hpp *(III)*

- Shorter but not single-source yet framework
  - No type-safety between host and device code

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 vendor...
    - Writing a C++17 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

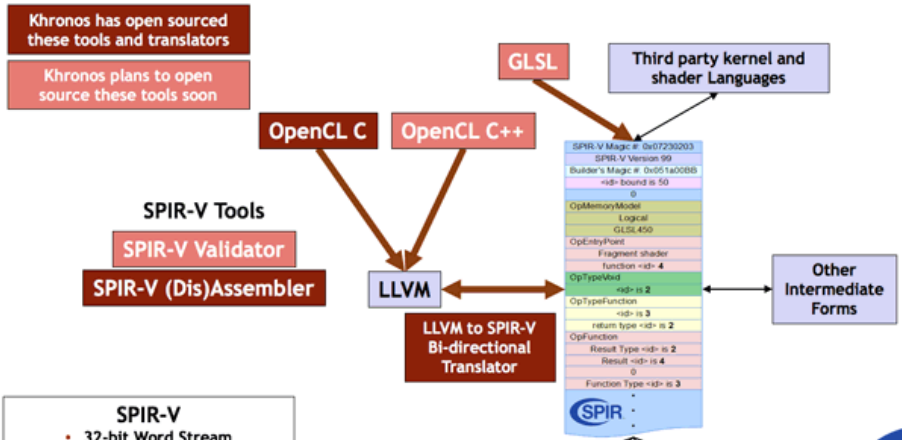
- First multi-API, intermediate language for parallel compute *and* graphics
  - Native representation for Vulkan shader and OpenCL kernel source languages
  - `https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf`
- Cross-vendor intermediate representation
  - Language front-ends can easily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

SPIR-V is no longer based on LLVM to isolate from LLVM roadmap changes

Driving SPIR-V Open Source ecosystem



Outline

Embedded systems: still a mix & match programming style

- Old C
- Old C++
- Language extensions
- Python
- Verilog
- VHDL
- TCL
- XML
- OpenCL
- Applications libraries
- Several assembly languages
- OS kernel code
- Alien firmware
- ...

¿¿¿ How to avoid Frankenstein programming ???

Position argument

Which language for unified heterogeneous computing?

-  Entry cost
- $\exists$ thousands of dead parallel languages...
  - Exit cost
- Use standard solutions with open source implementations

Remember C++ ?

2-line description by Bjarne Stroustrup

- Direct mapping to hardware
- Zero-overhead abstraction

Position argument 2: start with modern C++...

- Very successful & ubiquitous language
- Very active since C++11 (C++14, C++17...)
- Interoperability: seamless interaction with embedded world, libraries, OS...
- Full-stack: combine both low-level aspects with high-level programming
  - Pay only for what you need
- Open-source production-grade compilers (GCC & Clang/LLVM) & tools
- Classes can be used to define Domain Specific Embedded Language (DSEL)
- Not directly targeting FPGA...
  - But extensible through classes (DSEL)
  - Extensible with #pragma (already in Xilinx tools Vivado HLS, SDSoC & SDAccel) and attributes

Even better with modern C++ (C++14, C++17) (I)

- Huge library improvements, parallelism...
- Simpler syntax, type inference in constructors

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

Even better with modern C++ (C++14, C++17) *(II)*

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

```
def add(x, y):
  return x + y
print(add(2, 3))      # 5
print(add("2", "3")) # 23
print(add(2, "Boom")) # Fails at run-time :-(
```

  - Same in C++14 but compiled + static compile-time type-checking:

```
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
std::cout << add(2, "Boom"s) << std::endl; // Does not compile :-)
```

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

Even better with modern C++ (C++14, C++17) *(III)*

- Generic variadic lambdas & operator interpolation

```
#include <iostream>
#include <string>
using namespace std::string_literals;
// Define an adder on anything.
// Use new C++14 generic variadic lambda syntax
auto add = [] (auto... args) {
  // Use new C++17 operator folding syntax
  return (... + args);
};

int main() {
 std::cout << "The result is: " << add(1, 2, 3) << std::endl;
 std::cout << "The result is: " << add("begin"s, "end"s) << std::endl;
}
```

Even better with modern C++ (C++14, C++17) *(IV)*

Without using templated code! ~~`template <typename >`~~ ☺
Try this example on `https://wandbox.org/permlink/LambcvLqmyaSV4JL`

More about C++, metaprogramming and functional programming

- Boost.Hana: Your standard library for metaprogramming, Louis Dionne `https://github.com/boostorg/hana`
- Fun with Lambdas: C++14 Style, Sumant Tambe `http://cpptruths.blogspot.fr/2014/03/fun-with-lambdas-c14-style-part-1.html` `http://cpptruths.blogspot.fr/2014/05/fun-with-lambdas-c14-style-part-2.html` `http://cpptruths.blogspot.fr/2014/03/fun-with-lambdas-c14-style-part-3.html`
- Bartosz Milewski's Programming Cafe Concurrency, C++, Haskell, Category Theory `https://bartoszmilewski.com/?s=C++`
- David Sankel: Monoids, Monads, and Applicative Functors: Repeated Software Patterns, CPPCON'2016 `https://www.youtube.com/watch?v=DiisKQAkGM4`
- `https://isocpp.org`
- C++ Core Guidelines `https://github.com/isocpp/CppCoreGuidelines` with some tools to verify them

1 C++ version/3 years ⇝ Parallelizing C++ committee itself

- SG1, Concurrency: Hans Boehm (Google). Concurrency and parallelism
- SG2, Modules. Work on possible refinement or replacement for the header-based build model
- SG3, File System: Beman Dawes (Boost). Boost.Filesystem v3
- SG4, Networking. Networking related libraries, including sockets and HTTP
- SG5, Transactional Memory: Michael Wong (IBM). Exploring transactional memory constructs for potential future addition to the C++ language
- SG6, Numerics: Lawrence Crowl. Numerics topics, including but not limited to fixed point, decimal floating point, and fractions
- SG7, Reflection: Chandler Carruth (Google). Initially focusing on compile-time reflection capabilities
- SG8, Concepts. Near-term focus is on a convergence between the static if proposals and the parameter-type-constraints subset of concepts
- SG9, Ranges: Marshall Clow (Qualcomm). How to update the standard library with a range concept rather than naked iterator pairs, including containers and range-based algorithms
- SG10, Feature Test: Clark Nelson (Intel). Investigation into whether and how to standardize a way for portable code to check whether a particular C++ product implements a feature yet, as we continue to extend the standard
- SG11, Databases. Database-related library interfaces
- SG12, Undefined and Unspecified Behavior: Gabriel Dos Reis (Microsoft). A systematic review to catalog cases of undefined and unspecified behavior in the standard and recommend a coherent set of changes to define and/or specify the behavior
- SG13, HMI (Human/Machine Interface). Selected low-level graphic/pointing I/O primitives
- SG14, Game Development & Low Latency: Michael Wong (IBM). Topics of interest to game developers and (other) low-latency programming requirements

SG14

C++ sub-group working on “Games, Low Latency, Real-time applications, Graphics, Financial Trading” `https://groups.google.com/a/isocpp.org/forum/#!forum/sg14`

- Very intense activity & lot of attraction right now
- Example of proposals
  - SIMD & DSP operations
  - GPU/Accelerator design
  - P0037R0 Fixed point real numbers John McFarlane LEWG SG14/SG6: Lawrence Crowl
  - Low level bit manipulation
  - Lower precision floating point
  - P0059R0 Add rings to the Standard Library Guy Davidson LEWG SG14: Michael
  - Array View and Bounds checking
  - P0038R0 Flat Containers Sean Middleditch LEWG SG14: Patrice Roy
  - P0039R0 Extending `raw_storage_iterator` Brent Friedman LEWG SG14: Billy Baker
  - P0040R0 Extending memory management tools Brent Friedman LEWG SG14: Billy Baker
- Fusion of Embedded C++ working group into SG14 on 2016/02/16

Missing link...

- No tool providing
  - Modern C++ environment
  - Heterogeneous computing (FPGA...) *à la* SG14
  - Single source for programming productivity
    - Templated code across kernels
    - Type safety across heterogeneous execution
  - OpenCL interoperability to recycle other libraries and portability
  - Backed by open standard

SYCL 2.2 ≡ pure C++17 DSEL

Implement concepts useful for heterogeneous computing

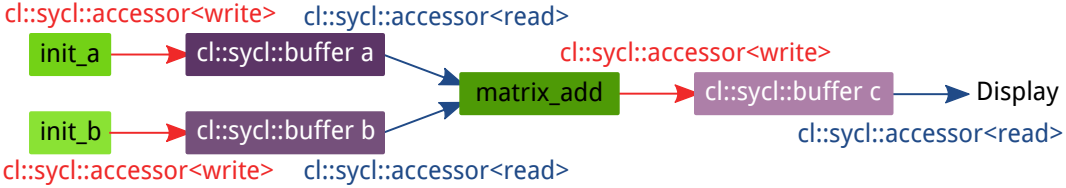
- SYngle-Çource programming modeL
  - Take advantage of CUDA on steroids & OpenMP simplicity and expressiveness
  - Compiled for host *and* device(s)
  - Enabling the creation of C++ higher level programming models & C++ templated libraries
  - System-level programming (SYstemCL)
- Asynchronous task graph
- Queues to direct computations on devices
- Hierarchical parallelism & kernel-side enqueue
- Buffers to define location-independent storage
  - Cross-device & cross-platforms
- Accessors to express usage for buffers and pipes: read/write/...
  - No explicit move
  - Automatic overlapping of communication/computation
- Hierarchical storage
  - Rely on C++ allocator to specify storage (SVM...)
  - Usual OpenCL-style global/local/private
- Most modern C++ features available for OpenCL
  - Programming interface based on abstraction of OpenCL components (data management, error handling...)
  - Provide OpenCL interoperability
- Directly executable DSEL
  - Host fallback & emulation for free
  - *No specific compiler needed for experimenting on host* ⇝ *Prototype with SYCL instead of OpenCL or SPIR-V !*
  - Debug and symmetry for SIMD/multithread on host

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];

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

 Matrix c;

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

Asynchronous task graph model

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



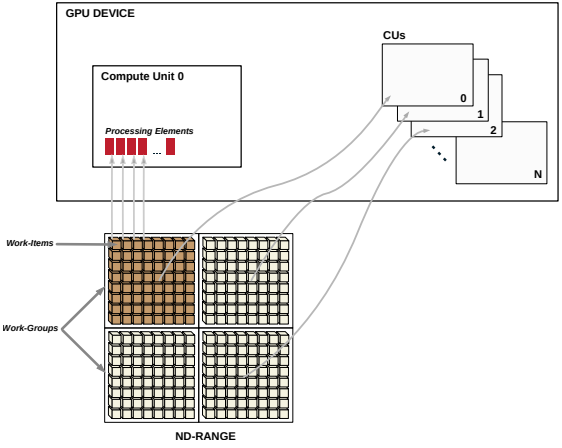
- Possible schedule by SYCL runtime:

⇝ 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 default device
    queue q;
    // Create some 2D buffers of float for our matrices
    buffer<double, 2> a({ N, M });
    buffer<double, 2> b({ N, M });
    buffer<double, 2> c({ N, M });
    // Launch a first asynchronous kernel to initialize a
    q.submit([&](auto &cgh) {
      // The kernel write a, so get a write accessor on it
      auto A = a.get_access<access::mode::write>(cgh);

      // Enqueue parallel kernel on a N*M 2D iteration space
      cgh.parallel_for<class init_a>({ N, M },
                        [=] (auto index) {
                          A[index] = index[0]*2 + index[1];
                        });
    });
    // Launch an asynchronous kernel to initialize b
    q.submit([&](auto &cgh) {
      // The kernel write b, so get a write accessor on it
      auto B = b.get_access<access::mode::write>(cgh);
      /* From the access pattern above, the SYCL runtime detect
         this command_group is independant from the first one
         and can be scheduled independently */

      // Enqueue parallel kernel on a N*M 2D iteration space
      cgh.parallel_for<class init_b>({ N, M },
                        [=] (auto index) {
                          B[index] = index[0]*2014 + index[1]*42;
                        });
    });
    // Launch an asynchronous kernel to compute matrix addition c = a + b
    q.submit([&](auto &cgh) {
      // In the kernel a and b are read, but c is written
      auto A = a.get_access<access::mode::read>(cgh);
      auto B = b.get_access<access::mode::read>(cgh);
      auto C = c.get_access<access::mode::write>(cgh);
      // From these accessors, the SYCL runtime will ensure that when
      // this kernel is run, the kernels computing a and b completed

      // Enqueue a parallel kernel on a N*M 2D iteration space
      cgh.parallel_for<class matrix_add>({ N, M },
                        [=] (auto index) {
                          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::mode::read, access::target::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);
        }
  }
  std::cout << "Good computation!" << std::endl;
  return 0;
}
```

Remember the OpenCL execution model?



How to control this 2-level parallelism in a a simple way?

From work-groups & work-items to hierarchical parallelism (I)

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

        localData[tile] = val;
      });
      // There is an implicit barrier here
      // Iterate on all the work-items
      group.parallel_for_work_item([&](item<1> tile) {
        ↓↓↓↓ ↓↓↓↓ ↓↓↓↓ ↓↓↓↓ ↓↓↓↓
        float sum = 0.f;
        for (unsigned offset = 0; offset < radius; ++offset)
          sum += filterLocal[offset]*localData[tile + offset + radius];
        float result = sum/(2*radius + 1);
        convolutionResult[tile] = result;
      });
      // There is an implicit barrier here
      // Iterate on all the work-items
      group.parallel_for_work_item(group, [&](item<1> tile)
        out_access[group*blocksize + tile] = convolutionResult[tile];
      });
      // There is an implicit barrier here
    });
});
```

From work-groups & work-items to hierarchical parallelism *(II)*

Very close to OpenMP 4 style! ☺

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

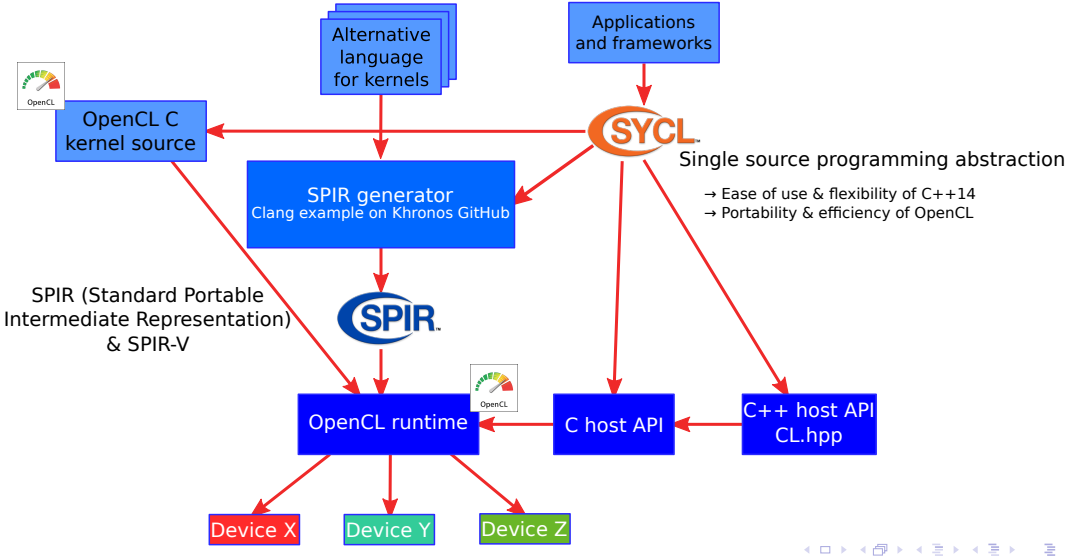
OpenCL interoperability mode

- SYCL $\equiv$ very generic parallel model
- Specific to SYCL: OpenCL interoperability *if needed*
- Can interact with OpenCL/Vulkan/OpenGL/... program or libraries *with no* overhead
- Value for OpenCL programmers: keep high-level features of SYCL
  - Simplify boilerplate and housekeeping
    - No explicit buffer transfer
    - Task and data dependency graphs
    - Templated C++ code
- The user can call any existing OpenCL kernel
  - Even HLS C++ & RTL Xilinx FPGA kernels !
  - Avoid writing painful OpenCL C/C++ host code

Example of SYCL program running explicit OpenCL code

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

#include <CL/sycl.hpp>

using namespace cl::sycl;

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

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

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

  // Create buffers from a & b vectors
  buffer<float> A { std::begin(a), std::end(a) };
  buffer<float> B { std::begin(b), std::end(b) };

  {
    // A buffer of N float using the storage of c
    buffer<float> C { c, N };

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

    // Build a kernel from the OpenCL kernel
    program.build();

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

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

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

  return 0;
}
```

SYCL in OpenCL ecosystem



Parallel STL towards C++17 proposal

- Parallel STL now in C++17 (from 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 implementation ☺ `https://github.com/KhronosGroup/SyclParallelSTL`

Using SYCL-like models in other areas

- SYCL ≡ generic heterogeneous computing model beyond OpenCL
  - `device` abstracts the accelerators
  - `queue` allows to launch tasks with computations overlapping communications and pipelining
  - `parallel_for<>` for // 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 (Processor-in-Memory)/Near-Memory Computing 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 bank/transposition or relocation

Future

- SYCL DSEL task graph model is pretty generic and not only OpenCL-centric
  - Close to run-time such as StarPU, Nanos++, OpenAMP... and can deal with remote nodes, even with lower level API such as MPI, MCAPI...
    - SYCL can target these runtimes!
  - Actually even not restricted to C++ either (SYPyCL, SYJaCL, SYJSCL, SYCaml...). SYFortranCL on top of Fortran 2008?

Outline

Known implementations of SYCL

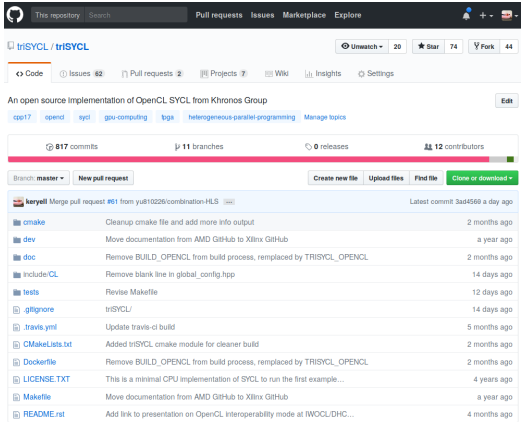
- ComputeCPP by Codeplay `https://www.codeplay.com/products/computecpp`
  - Most advanced SYCL 1.2 implementation
  - Outlining compiler generating SPIR
  - Run on any OpenCL device and CPU
  - Google & CodePlay have SYCL version of Eigen & TensorFlow using ComputeCPP
    `https://bitbucket.org/benoitsteiner/opencl`
    `https://github.com/benoitsteiner/tensorflow-opencl`
    `https://docs.google.com/spreadsheets/d/1YbHn7dAFPPG_PgTtgCJlWhMGorUPYsF681TsZ4Y4LP0`
- sycl-gtx `https://github.com/ProGTX/sycl-gtx`
  - Open source
  - No (outlining) compiler ⇝ use some macros with different syntax
- triSYCL `https://github.com/triSYCL/triSYCL`

Outline

triSYCL

- Open Source SYCL 1.2/2.2
- Uses C++17 templated classes
- Used by Khronos to define the SYCL and OpenCL C++ standard
  - Languages are now too complex to be defined without implementing...
- On-going implementation started at AMD and now led by Xilinx
- OpenMP for host parallelism
- Boost.Compute for OpenCL interaction
- Prototype of device compiler for Xilinx FPGA

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



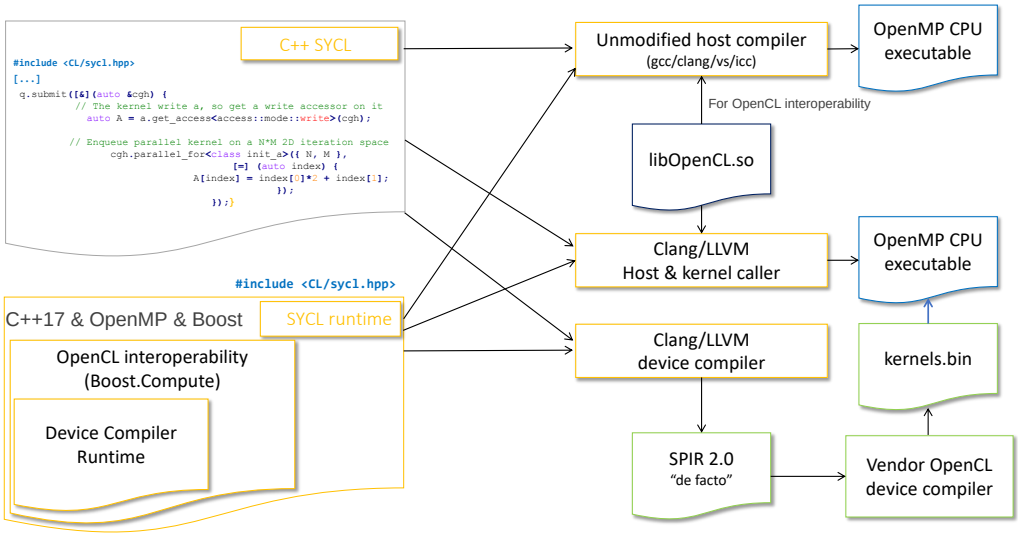
triSYCL (more details...)

- Pure modern C++ implementation
  - Use OpenMP for computation on CPU + std::thread for task graph
  - Rely on STL & Boost for zen style
  - CPU emulation for free
    - Quite useful for debugging
  - More focused on correctness than performance for now (array bound check...)
- Provide OpenCL-interoperability mode: can reuse existing OpenCL code
- Some extensions (Xilinx blocking pipes)
- On-going outlining compiler based on open-source Clang/LLVM compiler

Design strategy

- Rely on open-source and post-modernism
  - Do not solve again solved problems
  - Hardware companies often lag behind... ☹
  - Eagerly procrastinate
    - Wait for the others to do your work ☺
    - The best code we can write is the code we do not write (no bug, at least!)
- Develop as more as possible in library & runtime
  - Runtime in pure C++17 & C++ extensions and metaprogramming
- When not enough, add LLVM IR massaging (harder)
- When not enough, add Clang AST massaging (harder++)

triSYCL device compiler workflow



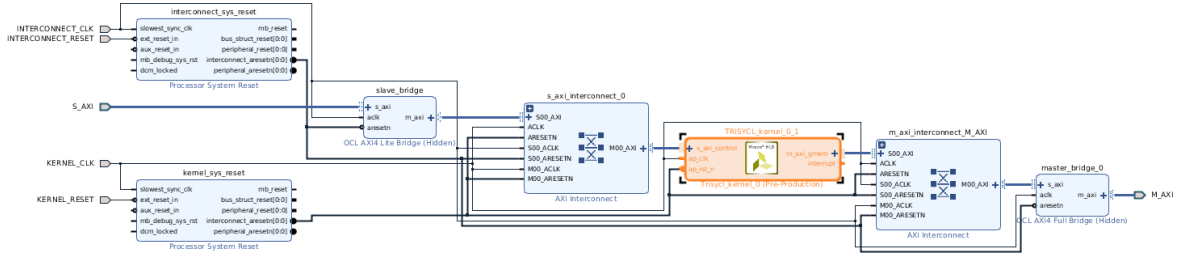
Example of compilation to device (FPGA...)

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

#include <boost/test/minimal.hpp>

using namespace cl::sycl;

constexpr size_t N = 300;
using Type = int;

int test_main(int argc, char *argv[]) {
  buffer<Type> a { N };
  buffer<Type> b { N };
  buffer<Type> c { N };

  {
    auto a_b = b.get_access<access::mode::discard_write>();
    // Initialize buffer with increasing numbers starting at 0
    std::iota(a_b.begin(), a_b.end(), 0);
  }

  {
    auto a_c = c.get_access<access::mode::discard_write>();
    // Initialize buffer with increasing numbers starting at 5
    std::iota(a_c.begin(), a_c.end(), 5);
  }

  queue q { default_selector {} };

  // Launch a kernel to do the summation
  q.submit([&] (handler &cgh) {
      // Get access to the data
      auto a_a = a.get_access<access::mode::discard_write>(cgh);
      auto a_b = b.get_access<access::mode::read>(cgh);
      auto a_c = c.get_access<access::mode::read>(cgh);

      // A typical FPGA-style pipelined kernel
      cgh.single_task<class add>([=,
          // Current limitation of device compiler
          d_a = drt::accessor<decltype(a_a)> { a_a },
          d_b = drt::accessor<decltype(a_b)> { a_b },
          d_c = drt::accessor<decltype(a_c)> { a_c }] {
          for (unsigned int i = 0 ; i < N; ++i)
            d_a[i] = d_b[i] + d_c[i];
        });
    });

  // Verify the result
  auto a_a = a.get_access<access::mode::read>();
  for (unsigned int i = 0 ; i < a.get_count(); ++i)
    BOOST_CHECK(a_a[i] == 5 + 2*i);

  return 0;
}
```

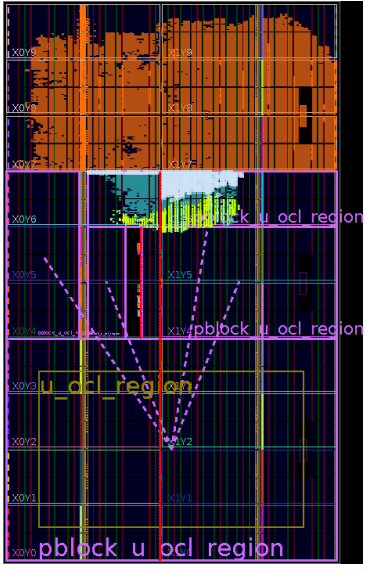
SPIR 2.0 “de facto” output in Clang 3.9.1

```
using; ModuleID = 'device_compiler/single_task_vector_add_drt.kernel.bc
source_filename = "device_compiler/single_task_vector_add_drt.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "spir64"
declare i32 @__gxx_personality_v0(...)
; Function Attrs: noinline norecurse nounwind uwtable
define spir_kernel void @TRISYCL_kernel_0(i32 addrspace(1)* %f.0.0.0.val, i32 addrspace(1)* %f.0.1.0.val, i32 addrspace(1)* %f.0.2.0.val) unnar
entry:
  br label %for.body.i
for.body.i:                                       ; preds = %for.body.i, %entry
  %indvars.iv.i = phi i64 [ 0, %entry ], [ %indvars.iv.next.i, %for.body.i ]
  %arrayidx.i.i = getelementptr inbounds i32, i32 addrspace(1)* %f.0.1.0.val, i64 %indvars.iv.i
  %0 = load i32, i32 addrspace(1)* %arrayidx.i.i, align 4, !tbaa !7
  %arrayidx.i15.i = getelementptr inbounds i32, i32 addrspace(1)* %f.0.2.0.val, i64 %indvars.iv.i
  %1 = load i32, i32 addrspace(1)* %arrayidx.i15.i, align 4, !tbaa !7
  %add.i = add nsw i32 %1, %0
  %arrayidx.i13.i = getelementptr inbounds i32, i32 addrspace(1)* %f.0.0.0.val, i64 %indvars.iv.i
  store i32 %add.i, i32 addrspace(1)* %arrayidx.i13.i, align 4, !tbaa !7
  %indvars.iv.next.i = add nuw nsw i64 %indvars.iv.i, 1
  %exitcond.i = icmp eq i64 %indvars.iv.next.i, 300
  br i1 %exitcond.i, label %"_ZZ9test_mainiPPcENK3$_1clERN2cl4sycl7handlerEENKUlvE_clEv.exit", label %for.body.i

"_ZZ9test_mainiPPcENK3$_1clERN2cl4sycl7handlerEENKUlvE_clEv.exit": ; preds = %for.body.i
  ret void
}
attributes #0 = { noinline norecurse nounwind uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="false
!llvm.ident = !{!0}
!opencl.spir.version = !{!1}
!opencl.ocl.version = !{!2}
!0 = !{!"clang version 3.9.1 "}
!1 = !{i32 2, i32 0}
!2 = !{i32 1, i32 2}
!3 = !{i32 1, i32 1, i32 1}
!4 = !{!"int *", !"int *", !"int *"}
!5 = !{!"", !"", !""}
!6 = !{!"read_write", !"read_write", !"read_write"}
!7 = !{!8, !8, i64 0}
!8 = !{!"int", !9, i64 0}
!9 = !{!"omnipotent char", !10, i64 0}
```

After Xilinx SDx 2017.2 xocc ingestion... Diagram in Vivado



After Xilinx SDx 2017.2 xocc ingestion... Layout in Vivado



Code execution on real FPGA

(Unit test in debug mode)

```
rkeryell@xirjoant40:~/Xilinx/Projects/OpenCL/SYCL/triSYCL/branch/device/tests (device)$ device_compiler/single_task
binary_size = 5978794
task::add_prelude
task::add_prelude
task::add_prelude
accessor(Accessor &a) : &a = 0x7ffd39395f40
 &buffer =0x7ffd39395f50
accessor(Accessor &a) : &a = 0x7ffd39395f30
 &buffer =0x7ffd39395f60
accessor(Accessor &a) : &a = 0x7ffd39395f20
 &buffer =0x7ffd39395f70
single_task &f = 0x7ffd39395f50
task::prelude
schedule_kernel &k = 0x1516060
Setting up  _ZN2cl4sycl6detail18instantiate_kernelIZZ9test_mainiPPcENK3$_1clERNS0_7handlerEE3addZZ9test_mainiS4_ENK
 aka TRISYCL_kernel_0
Name device xilinx_adm-pcie-7v3_1ddr_3_0
serialize_accessor_arg index =0, size = 4, arg = 0
serialize_accessor_arg index =1, size = 4, arg = 0x1
serialize_accessor_arg index =2, size = 4, arg = 0x2

**** no errors detected
```

triSYCL device compiler ideas

- Use a lot of C++ tricks in SYCL runtime to mark kernels and so on
- Compile with Clang `-O3 -fno-vectorize -fno-unroll-loops`
  - Generated very optimized kernel code
  - But do not generate futuristic vectorized SPIR (but still work with PoCL...)
- Mark and sweep approach for kernel selection
  - Mark all non kernel code with internal linkage
  - Apply existing dead-code elimination pass
- C++ made it harder
  - Crash SPIR consumers...
  - New `RELGCD` pass to remove variable storing former static constructors/destructors
  - Rename kernels because mangled name (`$...`) crashed some SPIR consumers

Device compiler

- For now implemented in Makefile ☺
  - Need to build a Clang driver at some point

```
# Process bitcode with SYCL passes to generate kernels
%.kernel.bc: %.pre_kernel.ll
  $(LLVM_BUILD_DIR)/bin/opt -load $(LLVM_BUILD_DIR)/lib/SYCL.so \
    -globalopt -deadargelim -argpromotion -deadargelim \
    -SYCL-kernel-filter -globaldce -RELGCD -inSPIRation \
    -o $@ $<
# Process bitcode with SYCL passes to generate kernel callers
%.kernel_caller.bc: %.pre_kernel_caller.ll
  $(LLVM_BUILD_DIR)/bin/opt -load $(LLVM_BUILD_DIR)/lib/SYCL.so \
    -globalopt -deadargelim -SYCL-args-flattening \
    -SYCL-serialize-arguments -deadargelim -o $@ $<
```

Open-source developments

- Runtime for device compiler not merged yet
  - `https://github.com/triSYCL/triSYCL/tree/device`
  - `https://xilinx.github.io/triSYCL/Doxygen/triSYCL/html`
- LLVM `https://github.com/triSYCL/llvm/tree/ronan/sycl`
- Clang `https://github.com/triSYCL/clang/tree/ronan/sycl`
- Please contribute ☺

Outline

TensorFlow SYCL

- TensorFlow: famous open-source framework for machine learning by Google
  - Define data flow graphs in which edges (tensors) and nodes (operations)
  - Initial TensorFlow version from Google supports CPU & nVidia GPU with CUDA
  - Other devices with XLA compiler
  - Comes with TensorBoard, a data visualization toolkit
- SYCL version started in 2015 by Codeplay
  - CUDA is single-source C++, SYCL easier to use than OpenCL C/C++
  - Joint effort by Codeplay, Google, Xilinx, Oracle...
  - Up-streamed directly in `https://github.com/tensorflow/tensorflow`
- Eigen: C++ library with mathematical & tensor operations
  - Use template metaprogramming to do kernel fusion
  - Extended with SYCL devices and SYCL memory management
- Tensorflow
  - Add SYCL devices
- Developed and tested with Codeplay ComputeCpp
  - Interesting to test with another SYCL implementation: triSYCL

Eigen

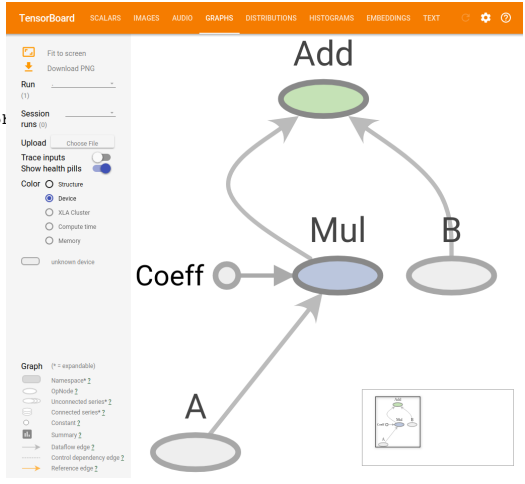
- Puts the “tensor” in TensorFlow
- C++ template library for linear algebra
  - Tensor module developed by Google and the SYCL extension by Codeplay
  - Single-source
  - Multiple devices available : Eigen thread pool, CUDA, SYCL
  - Lazy evaluation and kernel fusion built-in
  - Explicit scheduler
  - Follow CUDA low-level memory management
  - 2 previous points do not fully take advantage of SYCL high-level concepts
- Worked with ComputeCPP and now triSYCL
  - Available upstream `https://bitbucket.org/eigen/eigen`
  - Reuse triSYCL CMake module from the SYCL Parallel STL

SYCL Eigen example: computing (a + b) * b with tensors

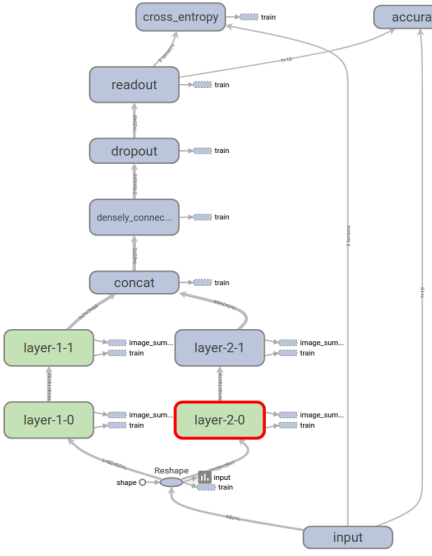
```
std::vector<cl::sycl::device> devices = Eigen::get_sycl_supported_devices();
QueueInterface queueInterface(devices[0]);
auto s_device = Eigen::SyclDevice(&queueInterface);

// Define the shape of the rank 3 tensors
IndexType sizeDim1 = 100, sizeDim2 = 20, sizeDim3 = 20;
Eigen::array<IndexType, 3> tensorRange = { { sizeDim1, sizeDim2, sizeDim3 } };
Eigen::Tensor<DataType, 3, DataLayout, IndexType> in1 { tensorRange };
Eigen::Tensor<DataType, 3, DataLayout, IndexType> in2 { tensorRange };
Eigen::Tensor<DataType, 3, DataLayout, IndexType> out { tensorRange };
Eigen::Tensor<DataType, 3, DataLayout, IndexType> out_host { tensorRange };

// Fill tensors with random values
in1.setRandom();
in2.setRandom();

// Allocate device memory for input and output tensors
auto gpu_in1_data = static_cast<DataType*>(s_device.allocate(in1.dimensions().TotalSize() * sizeof(DataType)));
auto gpu_in2_data = static_cast<DataType*>(s_device.allocate(in2.dimensions().TotalSize() * sizeof(DataType)));
auto gpu_out_data = static_cast<DataType*>(s_device.allocate(out.dimensions().TotalSize() * sizeof(DataType)));

// Create TensorMap from device memory
Eigen::TensorMap<Eigen::Tensor<DataType, 3, DataLayout, IndexType>> gpu_in1 { gpu_in1_data, tensorRange };
Eigen::TensorMap<Eigen::Tensor<DataType, 3, DataLayout, IndexType>> gpu_in2 { gpu_in2_data, tensorRange };
Eigen::TensorMap<Eigen::Tensor<DataType, 3, DataLayout, IndexType>> gpu_out { gpu_out_data, tensorRange };

// Copy the input data to the device
s_device.memcpyHostToDevice(gpu_in1_data, in1.data(), (in1.dimensions().TotalSize()) * sizeof(DataType));
s_device.memcpyHostToDevice(gpu_in2_data, in2.data(), (in2.dimensions().TotalSize()) * sizeof(DataType));
// c = (a + b) * b done on the sycl_device
gpu_out.device(s_device) = (gpu_in1 + gpu_in2) * gpu_in2;
// Copy the data back to the host
s_device.memcpyDeviceToHost(out.data(), gpu_out_data, (out.dimensions().TotalSize()) * sizeof(DataType));
// c = (a + b) * b done on the CPU
out_host = (in1 + in2) * in2;
```

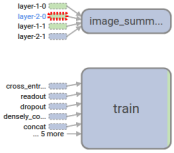
TensorFlow SYCL example

```
import tensorflow as tf
sess = tf.InteractiveSession()
file_writer = tf.summary.FileWriter('logs', sess.graph)
# To output a new version of the graph:
def ug():
  file_writer.add_graph(sess.graph)
  file_writer.flush()
coeff = tf.constant(3.0, tf.float32, name = "Coeff")
a = tf.placeholder(tf.float32, name = "A")
b = tf.placeholder(tf.float32, name = "B")
with tf.device(tf.DeviceSpec(device_type="SYCL")):
  product = tf.multiply(coeff, a, name = "Mul")
with tf.device(tf.DeviceSpec(device_type="CPU")):
  linear_model = tf.add(product, b, name = "Add")
print(sess.run(linear_model, {a : 3, b : 4.5 }))
ug()

13.5
```



TensorFlow SYCL to recognize handwritten digits

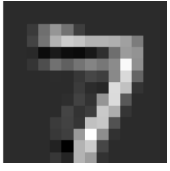




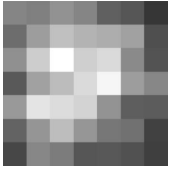
Layer 1-0

Layer 1-1

Input



Layer 2-0



Layer 2-1

Adding OpenCL interoperability to Eigen

- Goal: Introduce the ability to use native OpenCL kernels in Eigen
- Using OpenCL kernels:
  - Use existing optimised kernels (for FPGA)
  - Target specific accelerators (FPGA too ☺)
- SYCL OpenCL interoperability mode allows for that possibility
- Implemented as a new Eigen operation
  - Takes an arbitrary number of inputs
  - User-provided OpenCL file and kernel name
  - Accepts binary or OpenCL source file
  - Can use dynamically SDx xocc on Xilinx platform
    - Beware of the compilation time at the first run ☺

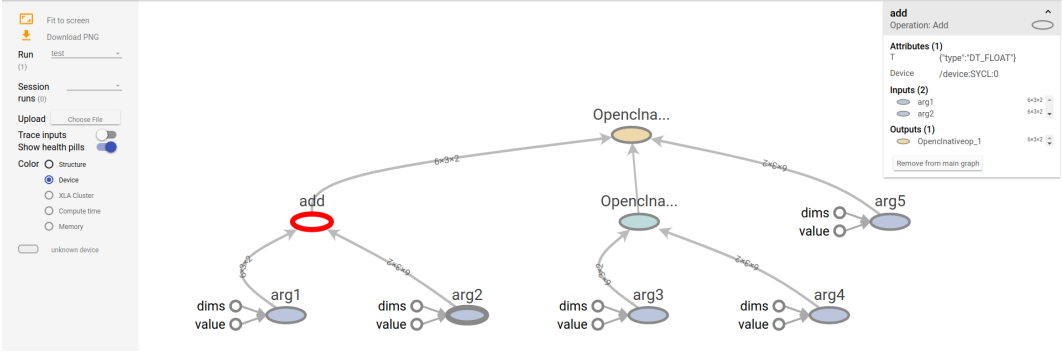
```
nativeOCL(void** arg_list, size_t arg_num, std::string kernel_name,
          std::string file_name, bool is_bin)
```

OpenCL interoperability in TensorFlow *(I)*

SYCL:0 ⇝ Host; SYCL:1 ⇝ FPGA (Xilinx OpenCL); SYCL:2 ⇝ CPU (Intel OpenCL)

```
import tensorflow as tf
def testOclOp(self):
  conf = tf.ConfigProto(allow_soft_placement=False, device_count={'SYCL': 3})
  sess = tf.InteractiveSession(config=conf)

  with tf.device('/cpu:0'):
    arg1 = tf.fill([6,3,2], 11.5, name="arg1")
    arg2 = tf.fill([6,3,2], 10.5, name="arg2")
    arg3 = tf.fill([6,3,2], 5.0, name="arg3")
    arg4 = tf.fill([6,3,2], 2.0, name="arg4")
    arg5 = tf.fill([6,3,2], 2.0, name="arg5")

  with tf.device('/device:SYCL:0'):
    add_node = arg1 + arg2

  with tf.device('/device:SYCL:2'):
    mul_node = tf.user_ops.ocl_native_op(input_list=[arg3, arg4], output_type=tf.float32, shape=[6,3,2],
                                         file_name="/path/to/VecMul.cl", kernel_name="vector_mul", is_binary=False)

  with tf.device('/device:SYCL:1'):
    result = tf.user_ops.ocl_native_op(input_list=[add_node, mul_node, arg5], output_type=tf.float32, shape=[6,3,2],
                                       file_name="/path/to/VecAddMul.xclbin", kernel_name="vector_add_mul", is_binary=True)

res = sess.run([result])
print(res[0])
writer = tf.summary.FileWriter('/tmp/tensorflow/logs/test', sess.graph)
writer.close()
```

OpenCL interoperability in TensorFlow *(II)*



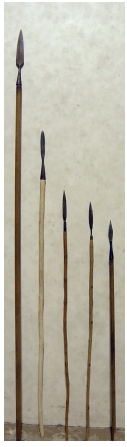
Outline

Puns and pronunciation explained

OpenCL SYCL

sickle ['si-kəl]

OpenCL SPIR



spear ['spir]

Conclusion

- Heterogeneous computing is everywhere and here to stay
  - Modern FPGA are complex MP-SoC
  - Full systems & HPC machines add other complexity levels...
- In post-modern C++ we trust: only way to control the full stack
- SYCL C++ standard from Khronos Group
  - Pure modern C++17 DSEL for heterogeneous computing
  - Candidate for ISO C++ WG21 SG14 standard
  - ISO C++ is an elitist democracy: only what is implemented and working is ratified
- triSYCL
  - Open Source on-going implementation. Join us!
  - On-going implementation of device compiler with Clang/LLVM
    - Single-source C++ to accelerators & FPGA synthesis
- Other implementations and libraries (Eigen, TensorFlow...) on `http://sycl.tech`
- Do it the standard way! Participate to the standards to have a real impact!

The sickle (SYCL) and the spear (SPIR)