From post-modern generic C++ to generic heterogeneous reconfigurable accelerators with the Khronos Group SYCL standard

Ronan Keryell (rkeryell@xilinx.com)
Xilinx Research Labs, San José, California
2018/07/05 @ COMPAS 2018, Toulouse, France

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...

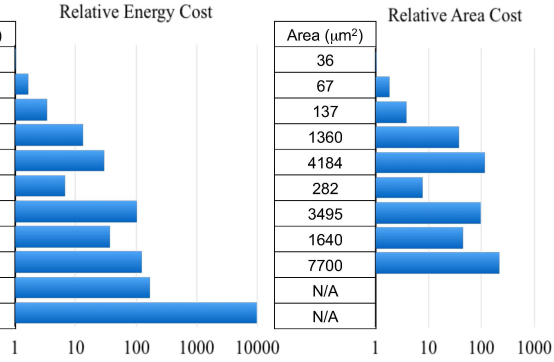
(Very old) 45nm technology characteristics

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

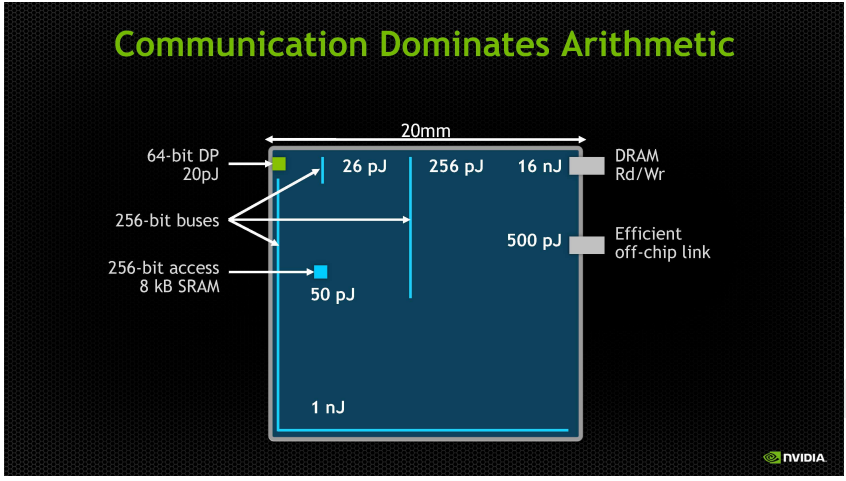


Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

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 ☺

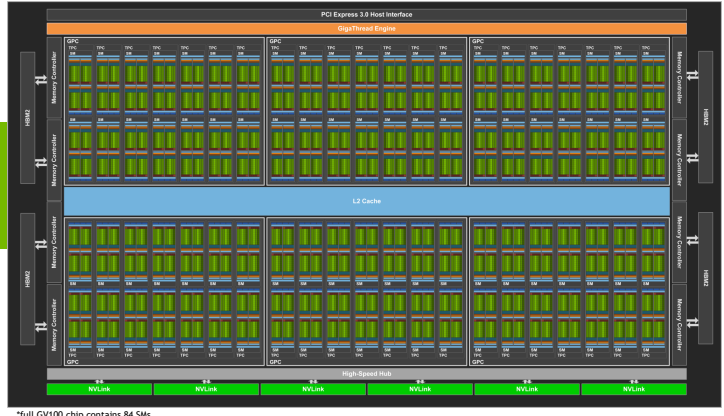
nVidia Tesla V100 @ HotChips 2017

TESLA V100

21B transistors
815 mm^2

80 SM
5120 CUDA Cores
640 Tensor Cores

16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

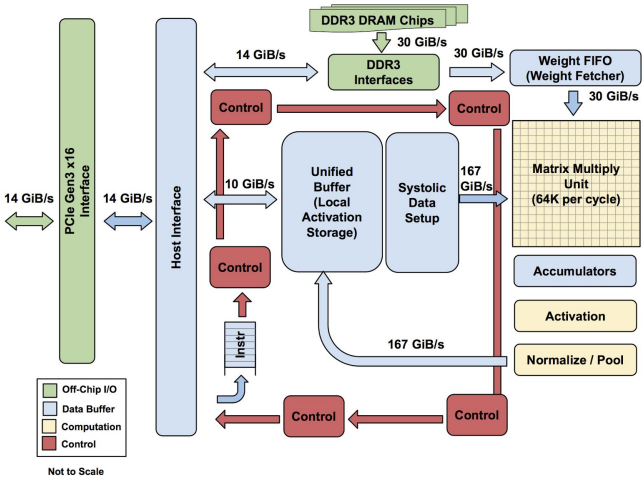


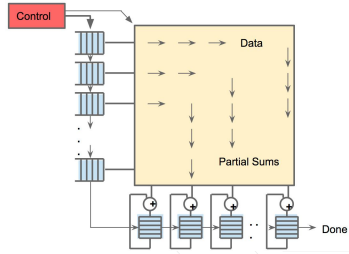
*full GV100 chip contains 84 SMs

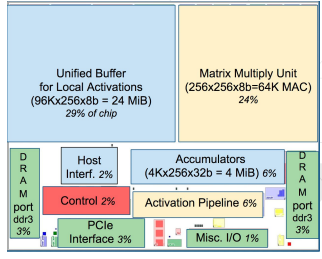
nVidia Tesla V100 SM with Tensor Core @ HotChips 2017

SM CORE

Google Tensor Processing Unit (TPU) 2016

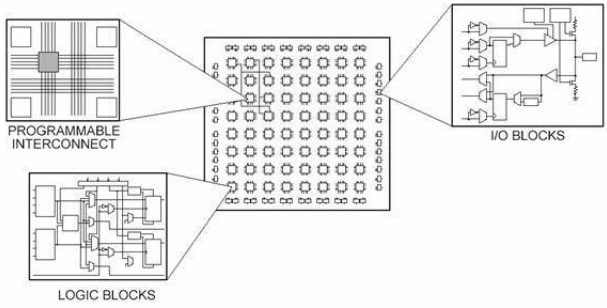






https://en.wikipedia.org/wiki/Tensor_processing_unit

Deconstructivism in (hardware) architecture: FPGA



https://www.quora.com/What-is-FPGA-How-does-that-works

Xilinx VCU1525 PCIe board (Ultrascale+ VU9P)

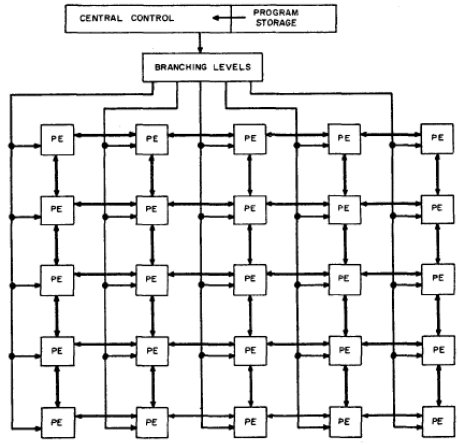
- Specialized reprogrammable hardware for computationally intensive applications
- Dual slot PCIe full-length, full height form-factor compliant
- Delivers 10-100x performance acceleration over server CPUs with a board designed to support up to 225W
- SDAccel platform reference design for custom board support
- Supported with SDAccel Development Environment for OpenCL, C, C++ and RTL
- VU9P Virtex UltraScale+ FPGA
- 21 TOPs (8-bit integer precision)

- 346Mb on chip memory
- 64GB on board DDR4 DIMM memory

https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html

Not a completely 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...

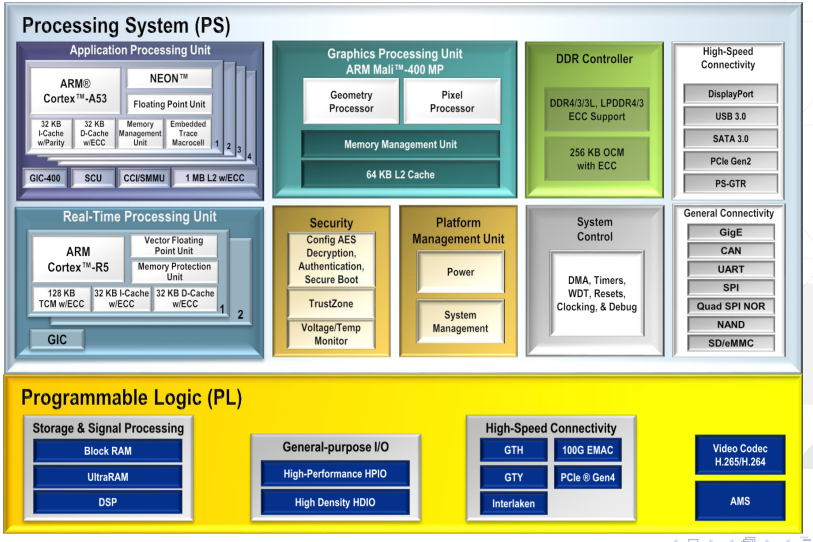
Stream computing: IBM 7950 Harvest 1962

- Coprocessor of the IBM Stretch (IBM 7030 Data Processing System)
  - IBM 7951 Stream coprocessor (3.10^6 characters/s)
  - IBM 7952 High performance core storage
  - IBM 7955 Magnetic tape system, also known as TRACTOR
  - IBM 7959 High speed I/O exchange
- Installed NSA for cryptanalysis

https://en.wikipedia.org/wiki/IBM_7950_Harvest

But now deconstructivism on a chip

FPGA in MPSoC: Xilinx Zynq UltraScale+ MPSoC



Typical modern MPSoC (for power consumption reasons)

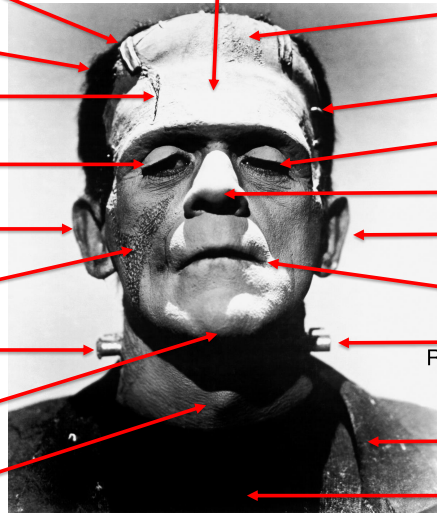
- Several CPU
  - Different kinds of CPU
- Different kinds of accelerators
  - Vision
  - ML
  - ...
- GPU
- DSP
- Some even with FPGA
- Lot of I/O
- Different power domains
- Various on-chip memories
- ...

All Programmable [with hardware mind set]

Programming style

- Anything resolves to writing 0 and 1 in (configuration) memory
  - Great unification achieved !
  - Well done !
  - Problem solved !
- Actually done by Eckert, Mauchly and Von Neumann around 1944-1950... ☺

Heterogeneous computing [software mind set nightmare]



SYCL

XILINX

¿ Can we do better ?

Outline

1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

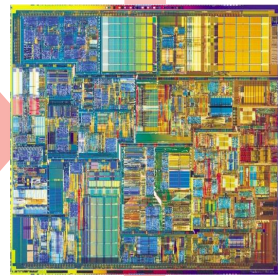
COLLADA

3D for the Web

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

WebGL glTF

NNEF OpenVX



OpenGL|SC OpenVG

Vulkan OpenGL|ES

OpenGL OpenXR

Vision and Neural Networks

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

Real-time 2D/3D

- Virtual and Augmented Reality
- Cross-platform gaming and UI
- CG Visual Effects
- CAD and Product Design
- Safety-critical displays

SPIR SYCL

OpenCL

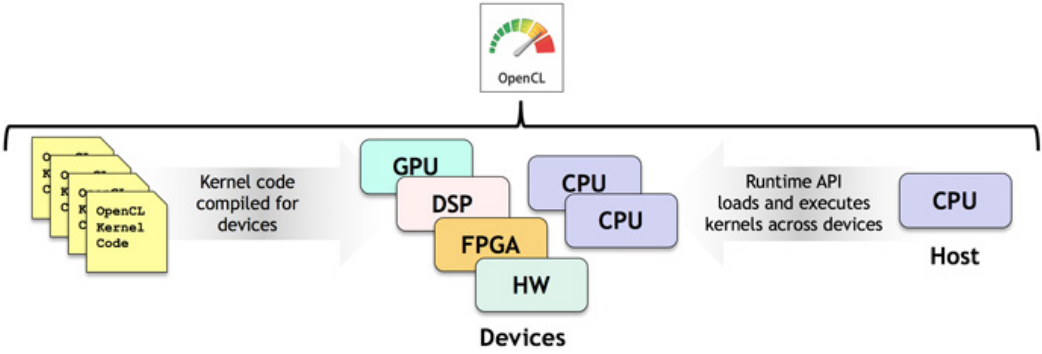
Parallel Computation

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

Outline

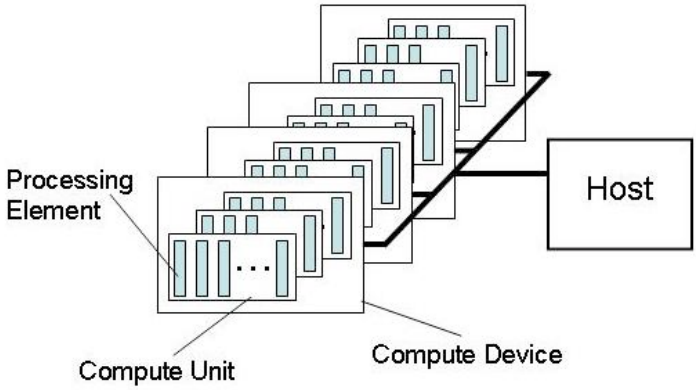
1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

OpenCL



- OpenCL: 2 host 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...

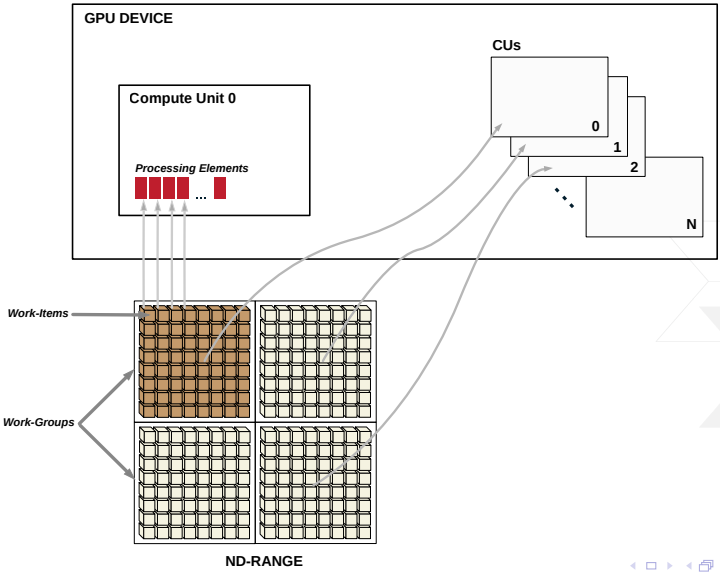
Architecture model



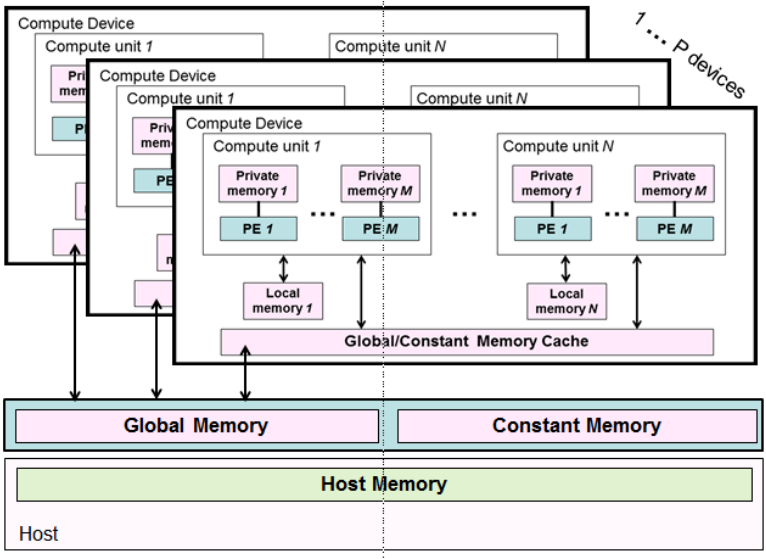
- Host threads launch computational *kernels* on *accelerators*

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

Execution model



Memory model



Outline

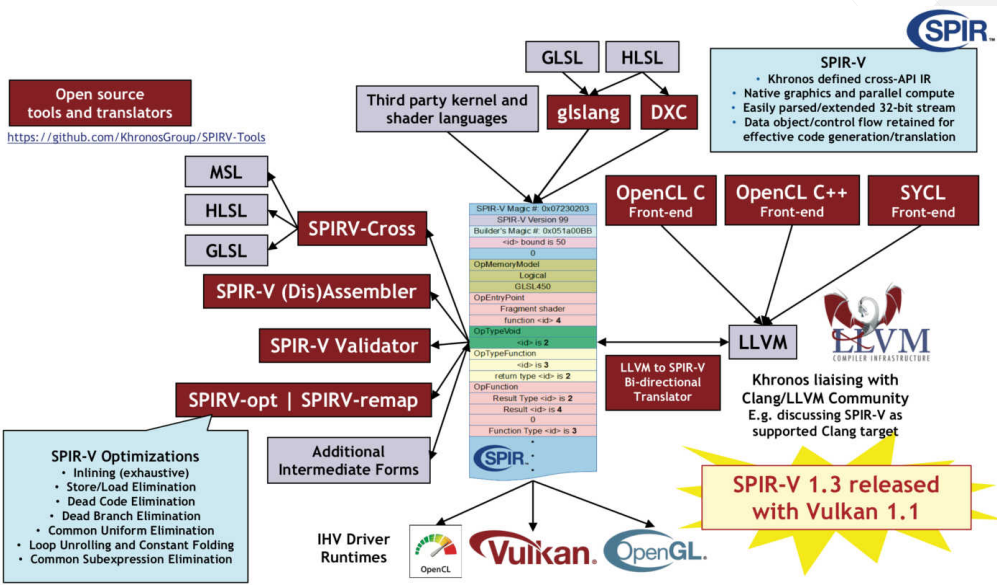
1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

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 !

Driving SPIR-V Open Source ecosystem



OpenCL/OpenGL/Vulkan/SPIR(-V): the language way

- Solve programming of heterogeneous accelerators
- Solve host-device interaction in a standard way
- Modeled according to graphics programming from the 90's & GPU-focused
  - Focused on portability across lot of GPU platforms
  - Fast JIT for platform adaptativity
- Require writing 2 unrelated programs using 2 different languages
  - Host Device
- Do not solve the global application problem
  - No type-safety between host and device for generic code
  - Unrelated memory address-spaces between host & device code
  - No global interprocedural & cross host/device optimizations in compiler
- Not single-source ☹
  - Part of the Frankenstein's programming model... ☹

Outline

1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

Position argument

Which language for unified heterogeneous computing?

-  Entry cost
- ∃ thousands of dead parallel languages...
  -  Exit cost
- Use standard solutions with open source implementations
- Only the full application matter
- So a kernel language does not really matter
- Avoid stitching and Frankenstein programming
- Need an holistic approach

Remember C++ ?

2-line description by Bjarne Stroustrup

- Direct mapping to hardware
- Zero-overhead abstraction

¿ And what about post-modern C++ ?

Modern Python/Modern C++/Old C++

- Python 3.6

```
v = [ 1,2,3,5,7 ]
for e in v:
  print(e)
```

- C++17

```
std::vector v { 1,2,3,5,7 };
for (auto e : v)
  std::cout << e << std::endl;
```

- C++03

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(5);
v.push_back(7);
for (std::vector<int>::iterator i =
       v.begin(); i != v.end(); ++i)
  std::cout << *i << std::endl;
```

Back to Python...

Modern C++ : like Python but with speed and type safety

- 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>` ☺

Back to Python...

⇝ Modern C++ : like Python but with speed and type safety

- 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>`~~ ☺

Generic variadic lambdas & operator interpolation

```cpp
#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 (here from left)
  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;
}
```

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

Make C++ more complex to make it. . . simpler!

- Modern C++ quite simpler to use
- But still compatible with old C++ & C
  - Globally more complex for. . . implementers!
- Keep **end-user only focused on simpler modern features**
- ISO C++ committee working on **C++ Core Guidelines**
  - `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`
- Subliminal messages
  - Learn modern C++. . . and teach students or customers! ☺
  - Forget about what you know from old C/C++ ☺

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

- Language rebooted in 2011
  - ⇝ Larger attraction & interest
- Is there any precedent of programming language developed cooperatively by of 200+ people democracy?
- ⇝ Use... parallelism ! ☺
  - Generic working groups:
    - Core Working Group. Library Working Group, Evolution Working Group, Library Evolution Working Group
  - Specialized Study Groups
    - SG1, Concurrency: Olivier Giroux (nVidia). Concurrency, parallelism, SIMD, memory model... SYCL is a candidate here!
    - SG6, Numerics: Lawrence Crowl. Numerics topics, fixed point, decimal floating point, fractions, arbitrary precision...
    - SG14, Game Development & Low Latency: Michael Wong (Codeplay): embedded systems, heterogeneous computing
- Production of Technical Specifications before standardization
  - Implemented in several compilers
  - Allow experiments in the fields
  - Provide feedback to the ISO C++ Committee

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

- Very successful & ubiquitous language
  - Millions of programmers
  - Billions of users: any use of cars, IoT, IT, smartphone, Internet... depend on C/C++
- Very active since C++11 (C++14, C++17, C++20...)
- Interoperability: seamless interaction with embedded world, libraries, OS, C...
- ⇝ Unique existing position in embedded system to control the full stack!!!
- Full-stack: combine both low-level aspects with high-level programming, multi-paradigm
  - Pay only for what you need
- Open-source production-grade compilers (GCC & Clang/LLVM) & tools
  - Implemented before specification
    - ... because it is too complex to do otherwise!!!
- Generic programming + classes can be used to define Domain Specific Embedded Language (DSEL)
- Not directly targeting heterogeneous computing...
  - But extensible through classes (⇝ DSEL)
  - Also extensible with #pragma (already in Xilinx tools Vivado HLS, SDSoC & SDAccel for FPGA) and attributes

Outline

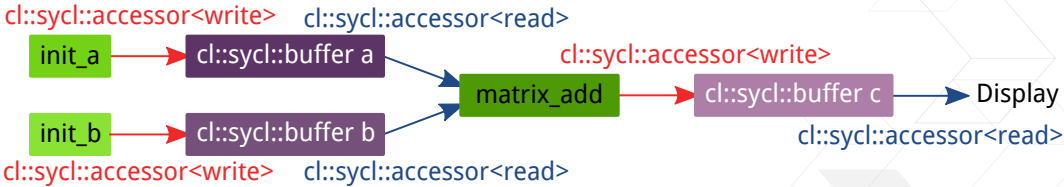
1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

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

Task graph programming — the code

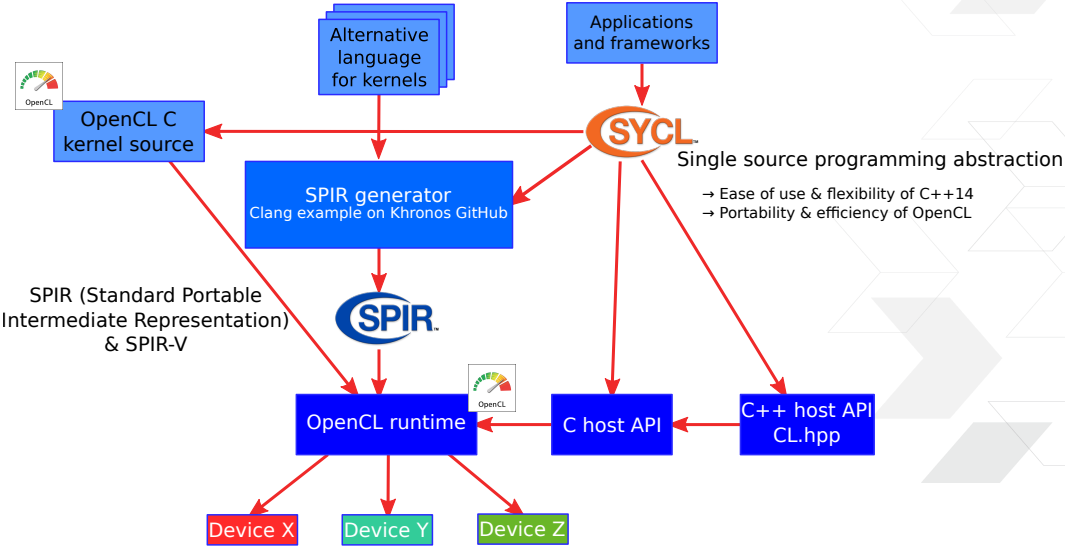
```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;
int main() {
  { // By sticking all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block

    // 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 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
    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;
}
```

SYCL 1.2.1 = Pure C++ based DSEL

Implement concepts useful for heterogeneous computing

- Modern C++ features available for OpenCL
  - Builds on the features of C++11, with additional support for C++14 and C++17
  - Enables ISO C++17 Parallel STL programs to be accelerated on OpenCL devices
  - Simplifies the porting of existing templated C++ Libraries and frameworks, i.e., Eigen, TensorFlow...
- Generic heterogeneous computing model
  - On CPUs, GPUs, FPGAs...
  - Hierarchical parallelism
- Portability across platforms and compilers
- Single source programming model
  - Better type safety
  - Simpler and cleaner code
  - Compiled host and device code
- Asynchronous task graph
  - Describes implicitly with kernel tasks using buffers through accessors
  - Automatic overlap kernel executions and communications
- Only Queues needed to direct computations on devices
  - Runtime handles multiple platforms, devices, and context
- Provides the full OpenCL feature set
- Interoperability with multiple languages
  - OpenCL, OpenGL, Vulkan, OpenVX, DirectX, and other vendor APIs, i.e., HLS C++ & RTL Xilinx FPGA kernels!
- Host fall-back
  - Easily develops and debugs applications on the host without a device
  - No specific compiler needed for experimenting on host
  - Full system architectural emulation for free!

SYCL in OpenCL ecosystem



C++17 STL has parallel algorithms now

- 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::execution::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`

SYCL-Next is coming...

- SYCL $\equiv$ generic heterogeneous computing model beyond OpenCL
- SYCL DSEL task graph model is pretty generic and not only OpenCL-centric
  - Close to run-times 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? ☺
- Can target various accelerators, other SoC processors (ARM R5, Microblaze, ML...)
- 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

Outline

1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

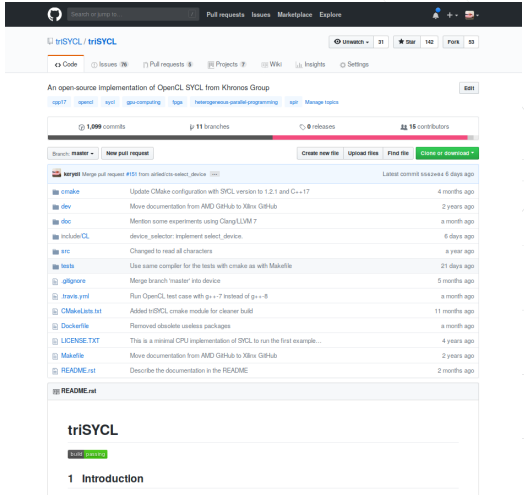
Known implementations of SYCL

- ComputeCPP by Codeplay `https://www.codeplay.com/products/computecpp`
  - Full SYCL 1.2.1 implementation since July 2018
  - Outlining compiler generating SPIR
  - Run on any OpenCL device and CPU
  - Google & CodePlay have SYCL version of Eigen & TensorFlow using ComputeCpp
- 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`

triSYCL

- Open Source SYCL 1.2.1
- 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

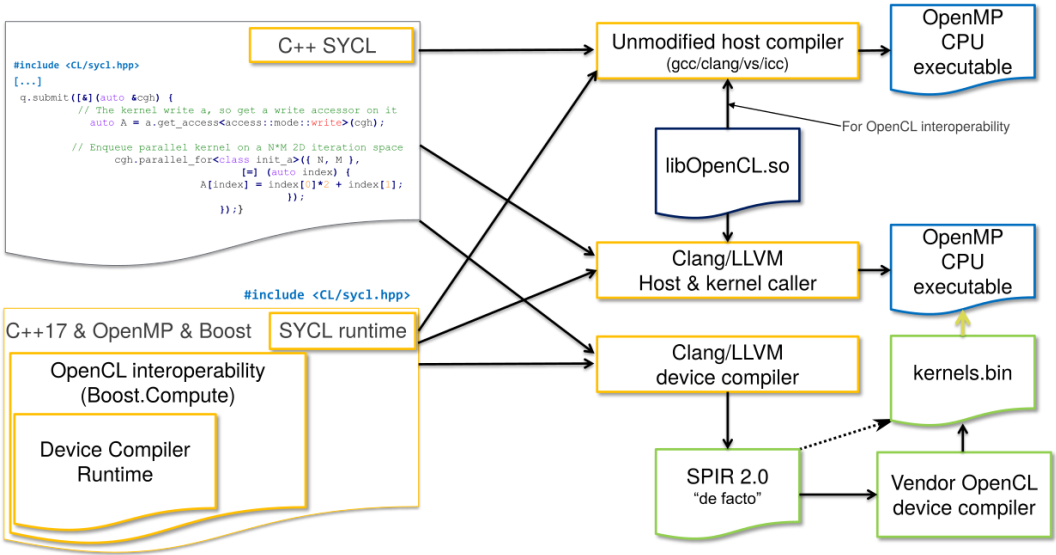
`https://github.com/triSYCL/triSYCL`



Design strategy

- Rely on open-source and post-modernism
  - Do not solve again solved problems
    - Use OpenMP, STL & Boost for zen style
  - 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



Low-level view of the device compiler workflow

- Modified Clang/LLVM 3.9
- Move to Clang/LLVM 7.0
  - Updated PoCL to support 7.0 version
- Makefile to control the compilation for now

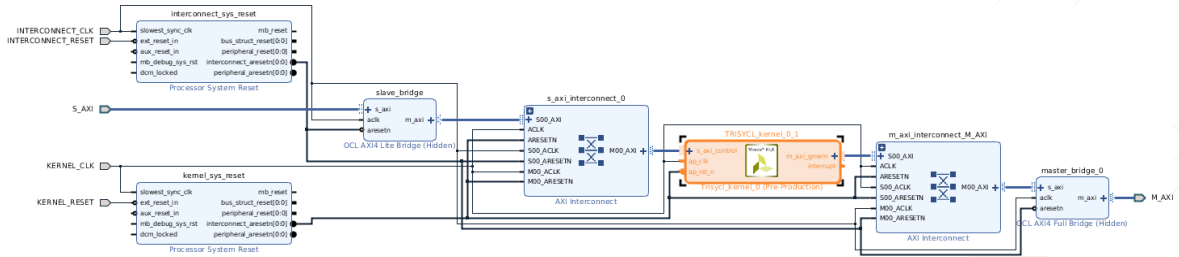
Example of compilation to device (FPGA...)

```
#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;
}
```

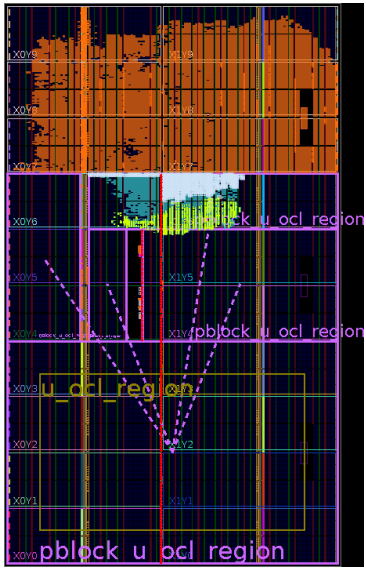
SPIR 2.0 “de facto” output in Clang 3.9.1

```
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 %"_ZZZ9test_mainiPPcENK3$_1clERN2cl4sycl7handlerEENKUlvE_clEv.exit", label %for.body.i

"_ZZZ9test_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 `xocc` ingestion... Diagram in Vivado



After Xilinx SDx `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
```

Outline

1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

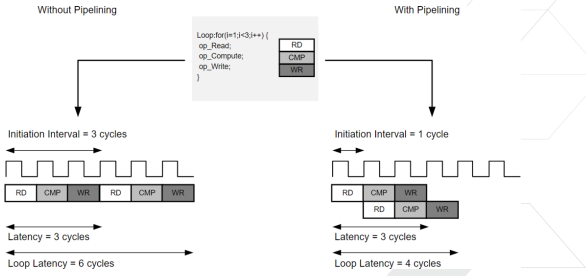
Let's try some SYCL vendor extensions. . .

- SYCL reserves `cl::sycl::vendor` namespace for some *vendor*
- Why not `cl::sycl::vendor::xilinx` to experiment with FPGA extensions?

Pipelining loops on FPGA

- Loop instructions sequentially executed by default
- Loop iteration starts only after last operation from previous iteration
- Elemental operations are synthesized anyway...
- Sequential pessimism ⇝ idle hardware and loss of performance ☹
- ⇝ Use loop pipelining for more parallelism
- Efficiency measure in hardware realm: Initiation Interval (II)

- Clock cycles between the starting times of consecutive loop iterations
- II can be 1 if no dependency and short operations



Decorating code for FPGA pipelining in triSYCL

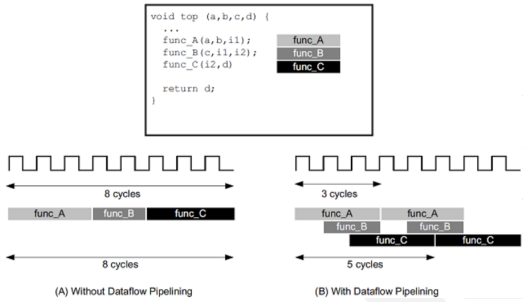
```
template<typename T, typename U>
void compute(T (&buffer_in)[BLOCK_SIZE],
             U (&buffer_out)[BLOCK_SIZE]) {
  for(int i = 0; i < NUM_ROWS; ++i) {
    for (int j = 0; j < WORD_PER_ROW; ++j) {
      vendor::xilinx::pipeline([&] {
        int inTmp = buffer_in[WORD_PER_ROW*i+j];
        int outTmp = inTmp * ALPHA;
        buffer_out[WORD_PER_ROW*i+j] = outTmp;
      });
    }
  }
}
```

- Use native C++ construct instead of alien #pragma or attribute (vendor OpenCL or HLS C++...)
- Compatible with metaprogramming
- Implementation
  - No need for specific parser/tool-chain!
  - Just use lambda + intrinsics! ☺

```
namespace cl::sycl::vendor::xilinx {
auto pipeline = [] (auto functor) noexcept {
  /* SSDM instruction is inserted before the argument
     functor to guide xocc to do pipeline. */
  _ssdm_op_SpecPipeline(1, 1, 0, 0, "");
  functor();
};
}
```

Dataflow optimization on FPGA

- On CPU
  - Functions are executed sequentially
- On FPGA
  - Functions are implemented in hardware. . .
  - They coexist!
- Possible to execute them in parallel! ☺
- Even better when in a loop



- Dataflow execution mode
  - Each function scheduled as soon as data is available
  - Using FIFOs to forward data

Decorating code for dataflow execution in triSYCL

```
cgh.single_task<class add>(
  [=,
   d_a = drt::accessor<decltype(a_a)> { a_a },
   d_b = drt::accessor<decltype(a_b)> { a_b }
   ] {
    int buffer_in[BLOCK_SIZE];
    int buffer_out[BLOCK_SIZE];
    vendor::xilinx::dataflow([&] {
        readInput(buffer_in, d_b);
        compute(buffer_in, buffer_out);
        writeOutput(buffer_out, d_a);
      });
  });
```

- Use native C++ construct instead of alien #pragma or attribute (vendor OpenCL or HLS C++...)
- Compatible with metaprogramming
- Implementation
  - No need for specific parser/tool-chain!
  - Just use lambda + intrinsics! ☺

```
namespace cl::sycl::vendor::xilinx {
auto dataflow = [] (auto functor) noexcept {
  /* SSDM instruction is inserted before the argument
     functor to guide xocc to do dataflow. */
  _ssdm_op_SpecDataflowPipeline(-1, "");
  functor();
};
}
```

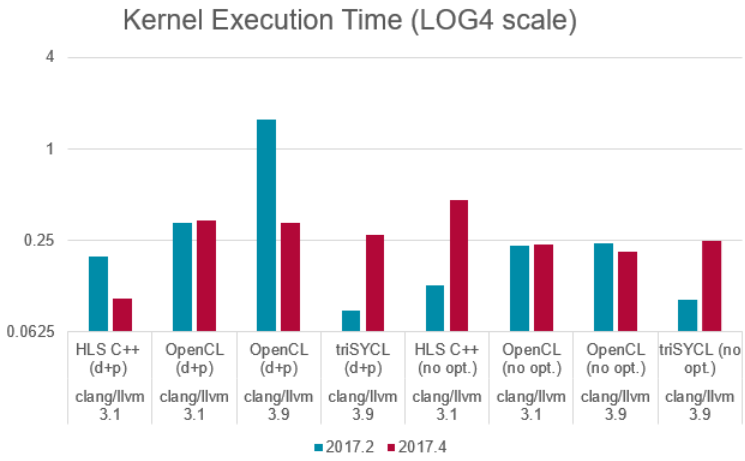
Hardware & Software testing context

- FPGA (ADM-PCIE-7V3)
- CPU (Intel core i7-6700)
- Linux Ubuntu 17.10
- triSYCL device compiler using Clang/LLVM 3.9
- 2017.2 Xilinx xocc compiler using Clang/LLVM 3.1
- 2017.2 Xilinx xocc compiler using Clang/LLVM 3.9
- 2017.4 Xilinx xocc compiler using Clang/LLVM 3.1
- 2017.4 Xilinx xocc compiler using Clang/LLVM 3.9
- Xilinx SDx OpenCL runtime 2017.2

Experiments :

- triSYCL (non optimized)
  - Single-source triSYCL on FPGA
- triSYCL (optimized)
  - Single-source triSYCL on FPGA with Xilinx-specific optimizations
- OpenCL (non optimized)
  - Targeting Xilinx OpenCL on FPGA
- OpenCL (optimized)
  - Targeting Xilinx OpenCL on FPGA with Xilinx-specific optimizations
- HLS C++ (non optimized) as built-in OpenCL kernel
  - Xilinx HLS C++ on FPGA
- HLS C++ (optimized) as built-in OpenCL kernel
  - Xilinx HLS C++ on FPGA with Xilinx-specific optimizations

Performance on FPGA — read/write rows of 2D Array

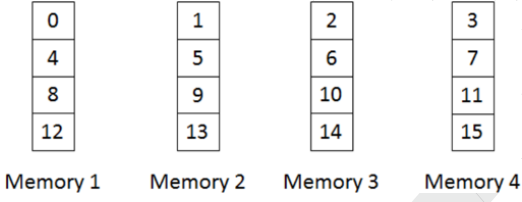


d: dataflow, **p**: pipelining, **no opt**: non optimized (but `-O3` triSYCL)

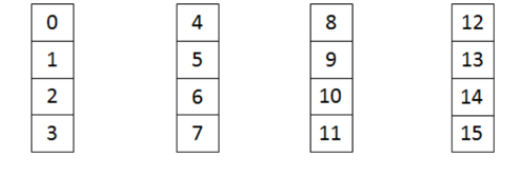
Partitioning memories (I)

- Remember memory bank conflicts on Cray vector computers in the 70's?
- In FPGA world, even memory is configurable!
- Example of array with 16 elements...
- Cyclic Partitioning
  - Each array element distributed to physical memory banks in order and cyclically

- Banks accessed in parallel → improved bandwidth
- Reduce latency for pipelined sequential accesses

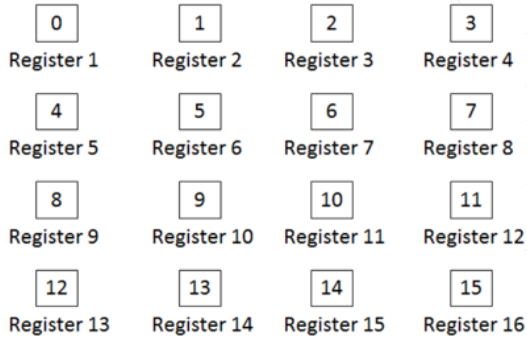


Partitioning memories (II)



- Block Partitioning
  - Each array element distributed to physical memory banks by block and in order
  - Banks accessed in parallel → improved bandwidth
  - Reduce latency for pipelined accesses with some distribution

- Complete Partitioning
  - Extreme distribution
  - Extreme bandwidth
  - Low latency



`partition_array` class in triSYCL use case (I)

Enhanced std::array

```
cgh.single_task<class add>([=] {
  // Cyclic partition for A as matrix
  // multiplication needs row-wise
  // parallel access
  xilinx::partition_array<Type, BLOCK_SIZE,
    xilinx::partition::cyclic<MAX_DIM>> A;
  // Block partition for B as matrix
  // multiplication needs column-wise
  // parallel access
  xilinx::partition_array<Type, BLOCK_SIZE,
    xilinx::partition::block<MAX_DIM>> B;
  xilinx::partition_array<Type, BLOCK_SIZE> C;
  [...]
});
```

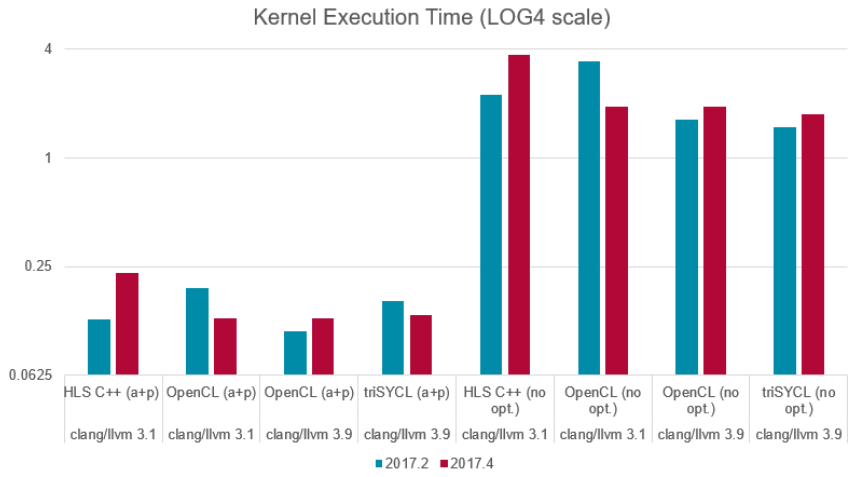
Xilinx Vivado HLS C++

```
int A[MAX_DIM * MAX_DIM];
int B[MAX_DIM * MAX_DIM];
int C[MAX_DIM * MAX_DIM];
#pragma HLS ARRAY_PARTITION variable=A dim=1 \
                            cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 \
                            block factor=64
```

Xilinx SDx OpenCL C

```
int A[MAX_DIM * MAX_DIM]
__attribute__((xcl_array_partition
               (cyclic, MAX_DIM, 1)));
int B[MAX_DIM * MAX_DIM]
__attribute__((xcl_array_partition
               (block, MAX_DIM, 1)));
int C[MAX_DIM * MAX_DIM];
```

Effect of array block and cyclic partitioning on matrix multiplication



a: array partitioning, **p**: pipelining, **no opt**: non optimized (but `-O3` triSYCL)

Outline

1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

Single-source C++ enables generic libraries

¿How to create a generic vector adder taking any number of vectors of any type?
Not possible to do this with OpenCL C or OpenCL C++... Not single-source C++ ! ☹

- Single-source is the strength of CUDA
  - Allow generic programming for heterogeneous computing nVidia devices
  - Type safety across host/device boundary
  - Avoid spaghetti Frankenstein's programming of OpenCL/OpenGL/Vulkan
  - Based on widely used C++ multi-paradigm language
  - But cross-platform, modern design, multi-vendor, DSP, FPGA, other accelerators
- ⇝ SYCL: portable cross-vendor single-source C++ standard from Khronos
- Explains why TensorFlow/Eigen uses CUDA or SYCL...

Generic adder in 25 lines of SYCL & C++17

```
auto generic_adder = [] (auto... inputs) {
  auto a = boost::hana::make_tuple(
    buffer<typename decltype(inputs)::value_type>
      { std::begin(inputs),
        std::end(inputs) }...);
  auto compute = [] (auto args) {
    // f(... f(f(f(x1, x2), x3), x4) ..., xn)
    return boost::hana::fold_left(args, [] (auto x, auto y)
                                        { return x + y; });
  };
  auto size = a[0_c].get_count();
  auto pseudo_result =
    compute(boost::hana::make_tuple(*std::begin(inputs)...));
  using return_value_type = decltype(pseudo_result);
  buffer<return_value_type> output { size };
  queue {}.submit([&] (handler& cgh) {
      auto ka = boost::hana::transform(a, [&] (auto b) {
          return b.template get_access<access::mode::read>(cgh);
        });
      auto ko = output.template
        get_access<access::mode::discard_write>(cgh);

      cgh.parallel_for<class gen_add>(size, [=] (id<1> i) {
          auto operands =
            boost::hana::transform(ka, [&] (auto acc) {
              return acc[i];
            });
          ko[i] = compute(operands);
        });
    });
  return output.template get_access<access::mode::read_write>();
};


int main() {
  std::vector<int> u { 1, 2, 3 };
  std::vector<float> v { 5, 6, 7 };
  for (auto e : generic_adder(u, v))
    std::cout << e << ' ';
  std::cout << std::endl;
  std::vector<double> a { 1, 2.5, 3.25, 10.125 };
  std::set<char> b { 5, 6, 7, 2 };
  std::list<float> c { -55, 6.5, -7.5, 0 };
  for (auto e : generic_adder(a, b, c))
    std::cout << e << ' ';
  std::cout << std::endl;
  return 0;
}
```

```
6 8 10
52 14 1.75 17.125
```

Generic executor in 25 lines of SYCL & C++17

```
auto generic_executor = [] (auto op, auto... inputs) {
  auto a = boost::hana::make_tuple(
    buffer<typename decltype(inputs)::value_type>
      { std::begin(inputs),
        std::end(inputs) }...);
  auto compute = [] (auto args) {
    // f(... f(f(f(x1, x2), x3), x4) ..., xn)
    return boost::hana::fold_left(args, op);
  };
  auto size = a[0_c].get_count();
  auto pseudo_result =
    compute(boost::hana::make_tuple(*std::begin(inputs)...));
  using return_value_type = decltype(pseudo_result);
  buffer<return_value_type> output { size };
  queue {}.submit([&] (handler& cgh) {
      auto ka = boost::hana::transform(a, [&] (auto b) {
          return b.template get_access<access::mode::read>(cgh);
        });
      auto ko = output.template
        get_access<access::mode::discard_write>(cgh);

      cgh.parallel_for<class gen_add>(size, [=] (id<1> i) {
          auto operands =
            boost::hana::transform(ka, [&] (auto acc) {
              return acc[i];
            });
          ko[i] = compute(operands);
        });
    });
  return output.template get_access<access::mode::read_write>();
};


int main() {
  std::vector<int> u { 1, 2, 3 };
  std::vector<float> v { 5, 6, 7 };
  for (auto e : generic_executor([] (auto x, auto y)
                                 { return x + y; }, u, v))

    std::cout << e << ' ';
  std::cout << std::endl;
  std::vector<double> a { 1, 2.5, 3.25, 10.125 };
  std::set<char> b { 5, 6, 7, 2 };
  std::list<float> c { -55, 6.5, -7.5, 0 };
  for (auto e : generic_executor([] (auto x, auto y)
                                 { return 3*x - 7*y; },
    std::cout << e << ' ';
  std::cout << std::endl;
  return 0;
}
```

6 8 10
352 -128 -44.25 -55.875

Modern metaprogramming as... hardware design tool

- Alternative implementation of

```
auto compute = [] (auto args) {
  return boost::hana::fold_left(args, [] (auto x, auto y) { return x + y; }
}; // f(... f(f(f(x1, x2), x3), x4) ..., xn)
```

  - Possible to use other Boost.Hana algorithms to add some hierarchy in the computation (Wallace's tree...)
  - Or to sort by type to minimize the hardware usage starting with "smallest" types
- Metaprogramming allows various implementations according to the types, sizes...
  - Kernel fusion, pipelined execution...
  - Codeplay VisionCpp, Eigen kernel fusion, Halide DSL...
  - In sync with C++ proposal on executors & execution contexts
- C++2a introspection & metaclasses will allow quite more!
  - Generative programming... `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0707r2.pdf`
    - Mind-blowing... `https://www.youtube.com/watch?v=4AfRAVcThyA`
  - Express directly and specialize code for each PE of a TPU for example
- Imagine if SystemC was invented with C++2a instead of C++98...

Single-source brings more optimization

- Kernel code optimized according to host parameter value or data type
  - A host constant scalar can be inlined into the kernel
    - Save one API call to send the parameter to the kernel
  - A host constant array/tensor can be inlined into the kernel
    - Save one API call to send the parameter to the kernel
    - Replace memory access by direct constant computation
  - Dead-code elimination
  - ...
- Single-source allows kernel fusion with (manual) metaprogramming
  - Kernel fusion heavily used in TensorFlow for example
  - Kernel fusion leads to better performance by reducing the launch & memory overhead
- ⇝ Can lead to better performance compared to split-source (OpenCL, HLS C/C++...)

Outline

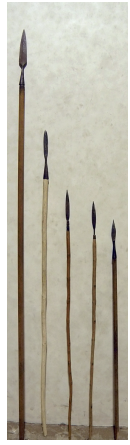
1. Khronos Group: open standards for heterogeneous systems
   - OpenCL
   - SPIR-V
2. Post-modern C++
3. Khronos SYCL C++
   - Implementations
   - FPGA-specific features and optimizations
4. The power of single source
5. Conclusion

Puns and pronunciation explained

OpenCL SYCL

sickle [ˈsi-kəl]

OpenCL SPIR



spear [ˈspir]

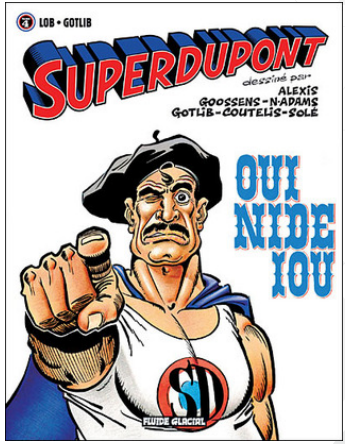
Conclusion

- Heterogeneous computing is everywhere and here to stay
  - Modern FPGA are complex MPSoC
  - Full systems & HPC machines add other complexity levels...
- In post-modern C++ we trust: only way to control the full stack of real hardware
- SYCL C++ standard from Khronos Group
  - Pure modern C++17 DSEL for heterogeneous computing
  - ISO C++ is an elitist democracy: only what is implemented and working is ratified
  - SYCL implementation ⇝ candidate for ISO C++ WG21 SG14 standard
- triSYCL
  - Open Source on-going implementation. Join us!
  - Xilinx is looking for interns, engineers, sabbaticals... ☺
  - On-going implementation: C++ runtime + 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!

Conclusion française pour une audience d'enseignants-chercheurs

- C++ post-moderne
  - Plus simple
  - Plus efficace (*move semantics*, *copy ellision*...)
  - Contrôle fin du matériel toujours possible
- Soyez des enseignants RESPONSABLES ! ☺
  - Déraisonnable de commencer par enseigner le C ($\approx$ 1970)
  - Criminel d'enseigner le vieux C++ ($\leq$ 2003) ☹
  - ⇝ Enseigner C++20 avec approche top-down
    - Algorithmes & structures de données standard, concurrence, SIMD, multi-paradigmes impératif/générique/objet/fonctionnel/contrats, bas niveau en C possible
- Utilisez vos élèves pour participer à des projets libres en... C++ post-moderne ☺
- Les réunions ISO C++ sont gratuites et les français ont plein de temps et de vacances ☺
- Arrêtez de râler et faites des propositions pour améliorer le C++
  - ISO C++ est une démocratie participative !
  - Si proposition acceptée: $\approx$ publication avec impact mondial!
  - Faites évoluer les évaluations CNU/CNRS/INRIA/... pour en tenir compte

The sickle (SYCL) and the spear (SPIR)



Adaptable.
Intelligent.

XILINX.

Power wall & speed of light: the final frontier... 2
(Very old) 45nm technology characteristics 3
Space-time traveling 4
Power wall & speed of light: implications 5
nVidia Tesla V100 @ HotChips 2017 6
nVidia Tesla V100 SM with Tensor Core @ HotChips 2017 7
Google Tensor Processing Unit (TPU) 2016 8
Deconstructivism in (hardware) architecture: FPGA 9
Xilinx VCU1525 PCIe board (Ultrascale+ VU9P) 10
Not a completely new problem... 11
Stream computing: IBM 7950 Harvest 1962 12
But now deconstructivism on a chip 13
Typical modern MPSoC (for power consumption reasons) 14
All Programmable [with hardware mind set] 15
Heterogeneous computing [software ~~mind set~~ nightmare] 16

1 Khronos Group: open standards for heterogeneous systems
Outline 18
OpenCL
Outline 21
OpenCL 22
Architecture model 23
Execution model 24
Memory model 25
SPIR-V
Outline 26
Interoperability nightmare in heterogeneous computing & graphics 27
Driving SPIR-V Open Source ecosystem 28
OpenCL/OpenGL/Vulkan/SPIR(-V): the language way 29

2 Post-modern C++
Outline 30
Position argument 31
Remember C++ ? 32
Modern Python/Modern C++/Old C++ 34
Back to Python...
Modern C++ : like Python but with speed and type safety 35
Back to Python...
Modern C++ : like Python but with speed and type safety 36
Generic variadic lambdas & operator interpolation 37
Make C++ more complex to make it. . . simpler! 38
1 C++ version/3 years → Parallelizing C++ committee itself! 39
Position argument 2: start with modern C++... 40

3 Khronos SYCL C++
Outline 41
Complete example of matrix addition in OpenCL SYCL 42
Asynchronous task graph model 43
Task graph programming — the code 44
SYCL 1.2.1 = Pure C++ based DSEL 45
SYCL in OpenCL ecosystem 46
C++17 STL has parallel algorithms now 47
SYCL-Next is coming... 48
Implementations
Outline 49
Known implementations of SYCL 50
triSYCL 51
Design strategy 52
triSYCL device compiler workflow 53
Low-level view of the device compiler workflow 54
Example of compilation to device (FPGA...) 55
SPIR 2.0 “de facto” output in Clang 3.9.1 56
After Xilinx SDx `xocc` ingestion... Diagram in Vivado 57
After Xilinx SDx `xocc` ingestion... Layout in Vivado 58
Code execution on real FPGA 59
FPGA-specific features and optimizations
Outline 60
Let’s try some SYCL vendor extensions. . . 61
Pipelining loops on FPGA 62
Decorating code for FPGA pipelining in triSYCL 63
Dataflow optimization on FPGA 64
Decorating code for dataflow execution in triSYCL 65
Hardware & Software testing context 66
Performance on FPGA — read/write rows of 2D Array 67

Partitioning memories 68
`partition_array` class in triSYCL use case 70
Effect of array block and cyclic partitioning on matrix multiplication 71

4 The power of single source
Outline 72
Single-source C++ enables generic libraries 73
Generic adder in 25 lines of SYCL & C++17 74
Generic executor in 25 lines of SYCL & C++17 75

Modern metaprogramming as. . . hardware design tool 76
Single-source brings more optimization 77

5 Conclusion
Outline 78
Puns and pronunciation explained 79
Conclusion 80
Conclusion française pour une audience d'enseignants-chercheurs 81
You are here ! 85