SYCL: a programming standard for heterogeneous computing based on modern C++

Ronan Keryell (rkeryell@xilinx.com)

Khronos Group & Xilinx Research Labs, San José, California

2019/10/09 @ Association of C and C++ Users - San Francisco Bay Area

KHRONOS GROUP

Outline

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...

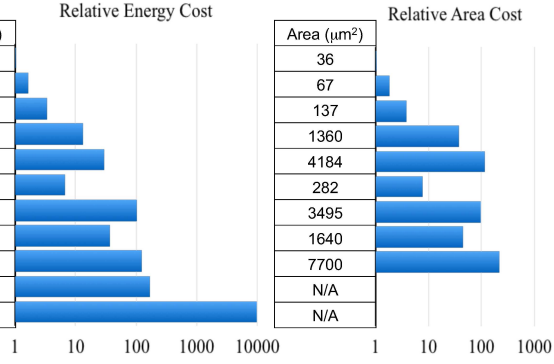
(Very old) 45nm technology characteristics

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

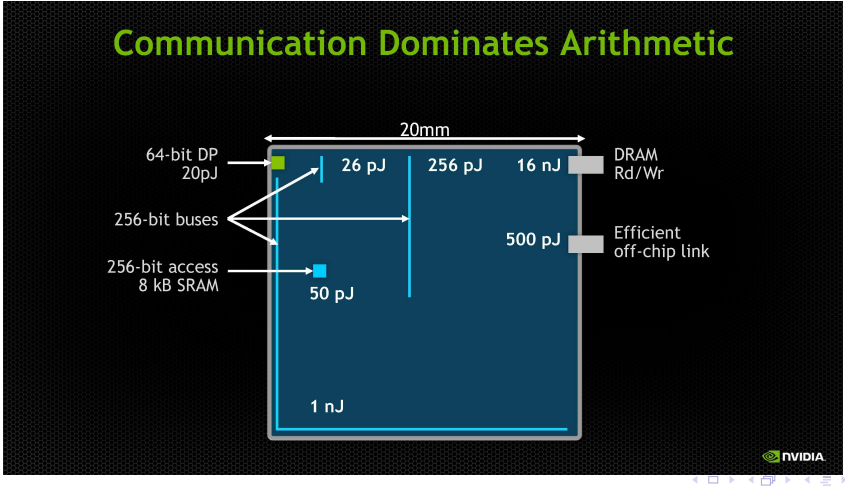


Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

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 ☺

ACM Turing award 2017: John L. Hennessy & David A. Patterson

A New Golden Age for Computer Architecture

John L. Hennessy, David A. Patterson, Communications of the ACM, February 2019, Vol. 62 No. 2, Pages 48-60

`https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext`

Award lecture: `https://www.youtube.com/watch?v=3LVeEjsn8Ts`

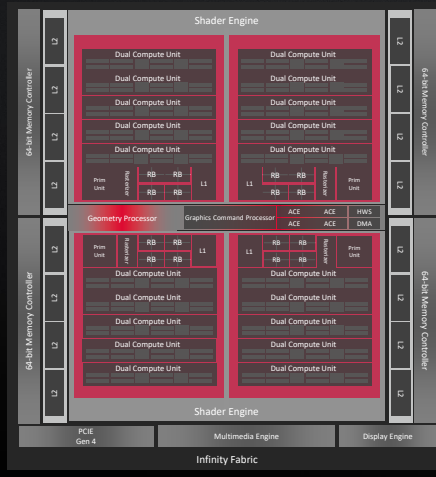
- *"Those who cannot remember the past are condemned to repeat it"*, George Santayana, 1905
- *"What we have before us are some breathtaking opportunities disguised as insoluble problems"*, John Gardner, 1965
- ⇝ Agile Hardware & Software Development
  - Open Architectures, inspired by the success of open source software

"The next decade will see a Cambrian explosion of novel computer architectures, meaning exciting times for computer architects in academia and in industry."

AMD 7nm “Navi” GPU @ HotChips-31 2019

"NAVI"

- Radeon Display Engine
  - New High Resolution HDR Displays
  - New Levels of Compression
- Radeon Multi-Media Engine
  - Seamless Streaming
  - Improved Encoding
- New Graphics RDNA Architecture
  - New Compute Units
  - Multilevel Cache
  - Streamlined Engine



AMD RYZEN RADEON™

SYCL: a programming standard for heterogeneous computing based on modern C++

Nvidia Turing GPU @ HotChips-31 2019

INTRODUCING TURING

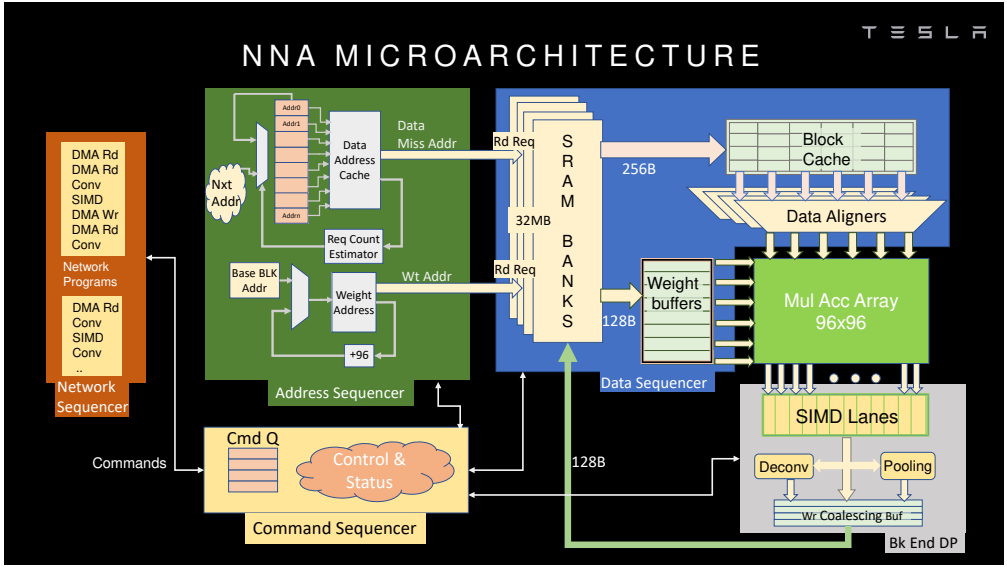
TU102 – TITAN RTX
18.6 BILLION TRANSISTORS

SM	72
CUDA CORES	4608
TENSOR CORES	576
RT CORES	72
GEOMETRY UNITS	36
TEXTURE UNITS	288
ROP UNITS	96
MEMORY	384-bit 7 GHz GDDR6
NVLINK CHANNELS	2

But wait...

92% of HotChips 2019 presentations are not about GPU!

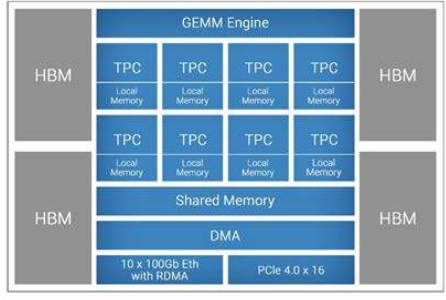
Tesla Neural Network Accelerator @ HotChips-31 2019



Habana Gaudi Processor for training @ HotChips-31 2019

Gaudi Processor Architecture

- Heterogenous compute architecture
  - TPC, GEMM & DMA using a shared SRAM
- VLIW SIMD TPC 2.0 Core (C-programmable)
- GEMM operations engine
- Tensor addressing
- Robust to any address stride
- Latency hiding capabilities
- PCIe Gen4.0 x16
- 4 HBMs: 2GT/s, 32 GB capacity, BW 1 TB/sec
- 10 ports of 100Gb Ethernet, or 20x50 GbE
  - With integrated RDMA over Converged Ethernet (RoCE v2)
- Dedicated HW and TPC ISA for special functions acceleration (e.g. Sigmoid, GeLU, Tanh)
- Mixed-precision data types: FP32, BF16, INT32, INT16, INT8, UINT32, UINT16 and UINT8

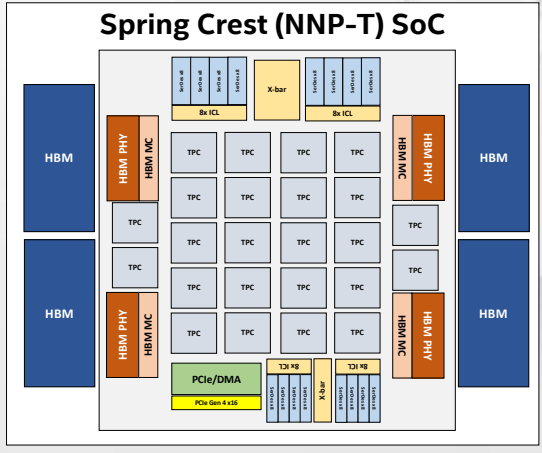


TSMC – 16nm

Intel Nervana NNP-T for training @ HotChips-31 2019

SPRING CREST (NNP-T) SOC

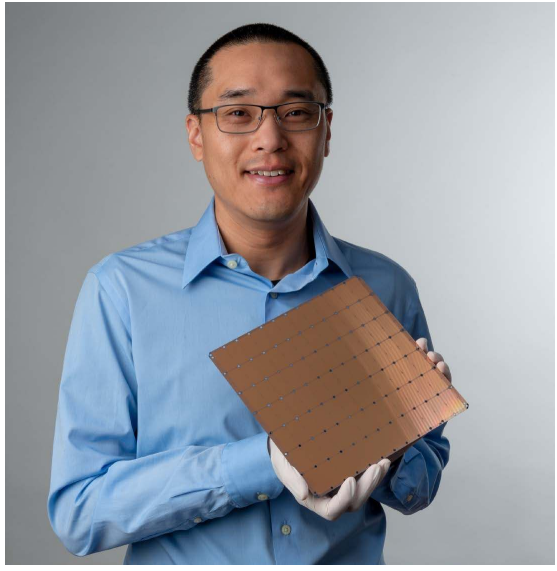
- PCIe Gen 4 x16 EP
- 4x HBM2
- 64 lanes SerDes
- 24 Tensor Processors
- Up to 119 TOPS
- 60 MB on-chip distributed memory
- Management CPU and Interfaces
- 2.5D packaging



Cerebras Wafer-Scale Deep Learning @ HotChips-31 2019

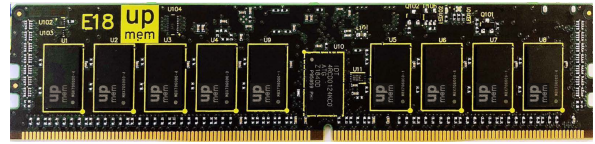
Largest Chip Ever Built

- 46,225 mm^2 silicon
- 1.2 trillion transistors
- 400,000 AI optimized cores
- 18 Gigabytes of On-chip Memory
- 9 PByte/s memory bandwidth
- 100 Pbit/s fabric bandwidth
- TSMC 16nm process

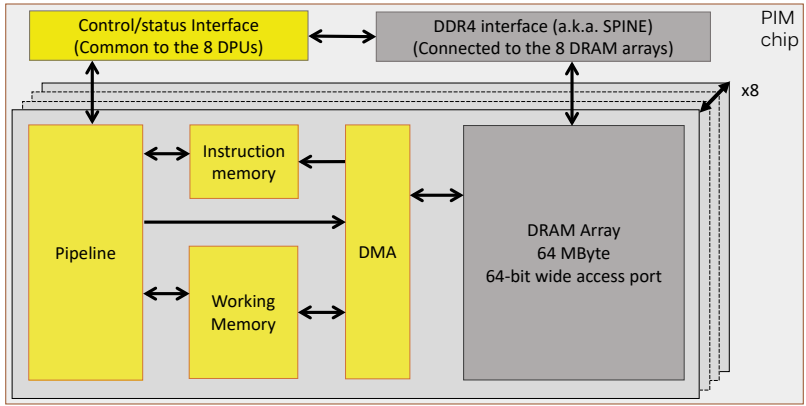


cerebras

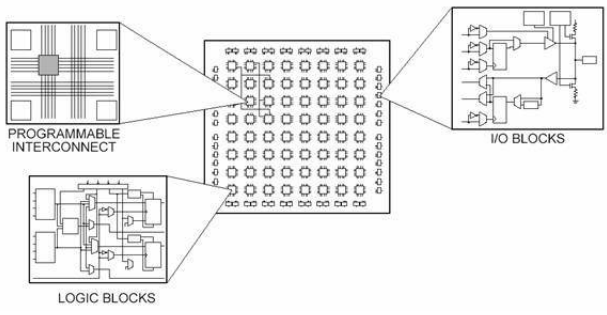
UPMEM: Processing In Memory accelerator @ HotChips-31 2019



PIM chip Block Diagram



Deconstructivism in (hardware) architecture: FPGA



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

Xilinx 7nm Versal AI Core VC1902

Versal Series Overview

Compute Engines

- Scalar Processors in every device
- Enhanced Programmable Logic
- New AI and enhanced DSP Engines

NoC and Memory

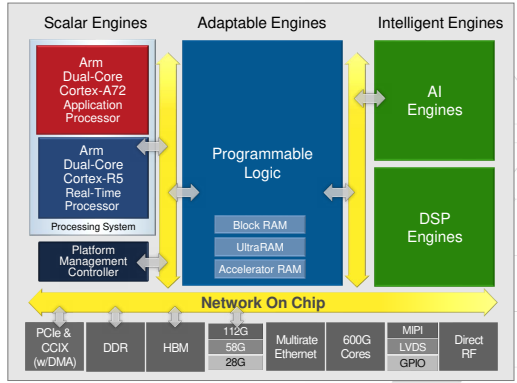
- High BW Network-on-Chip
- Hardened [LP]DDR4/5, and HBM

High-Speed Interfaces

- PCIe & CCIX up to Gen5
- Ethernet MAC up to 600Gbps

SerDes and RF

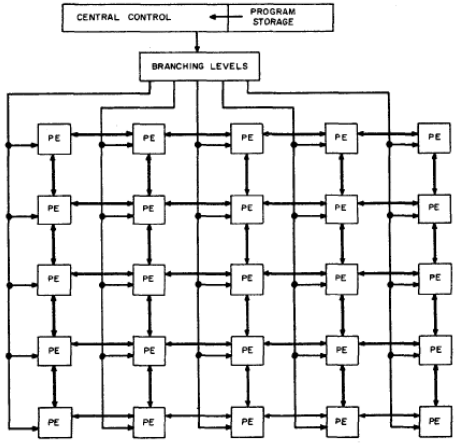
- SerDes Up to 112G PAM4
- Integrated ADC/DAC



SYCL: a programming standard for heterogeneous computing based on modern C++

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

Typical modern MPSoC (for power consumption reasons)

What's new? Everything is on-chip now!!!

- 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
- ...

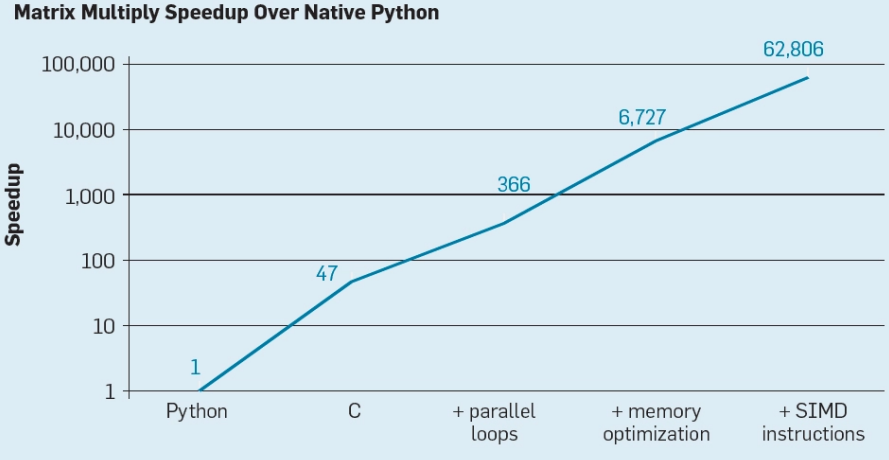
Outline

All Programmable [with hardware mind set]

Programming style

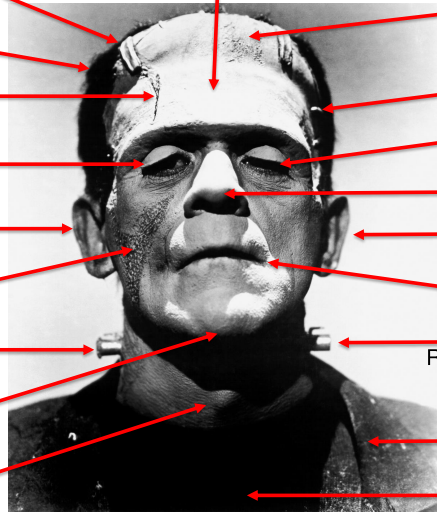
- Anything resolves to writing 0 and 1 in (configuration) memory
  - Great unification achieved !
  - Well done !
  - Problem solved !
- Actually done by Eckert, Mauchly and Von Neumann around 1944-1950... ☺

Huge programming dilemma: there's plenty of room at the top!



Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl, "*There's plenty of room at the top: What will drive growth in computer performance after Moore's Law ends?*" unpublished manuscript submitted for publication, 2019.

Heterogeneous computing [software ~~mind set~~ nightmare]



¿ Can we do better ?

Outline

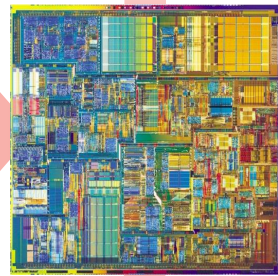
COLLADA

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

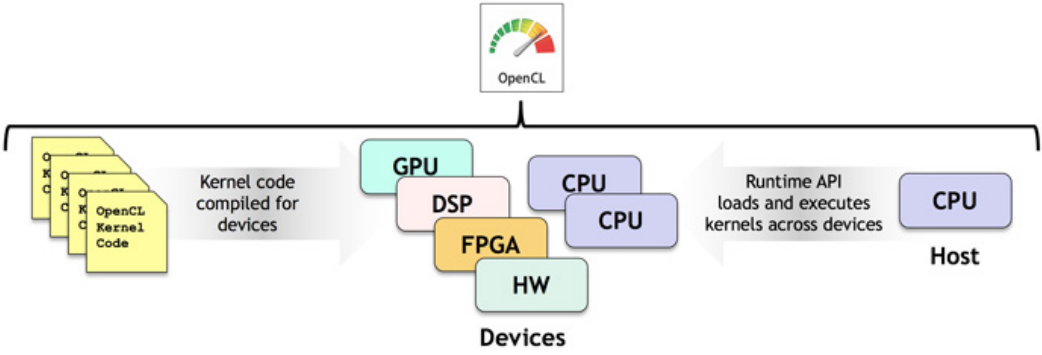
OpenCL

Parallel Computation

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

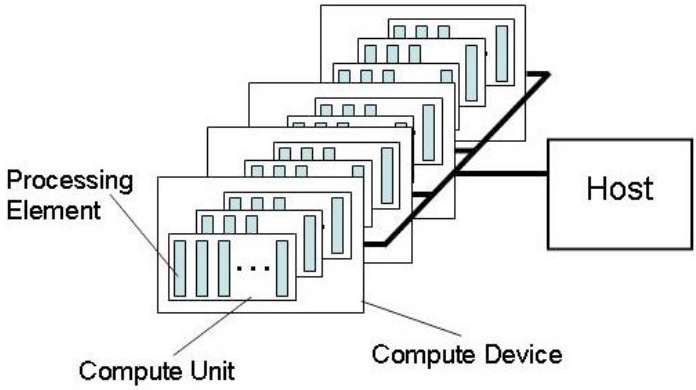
Outline

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...

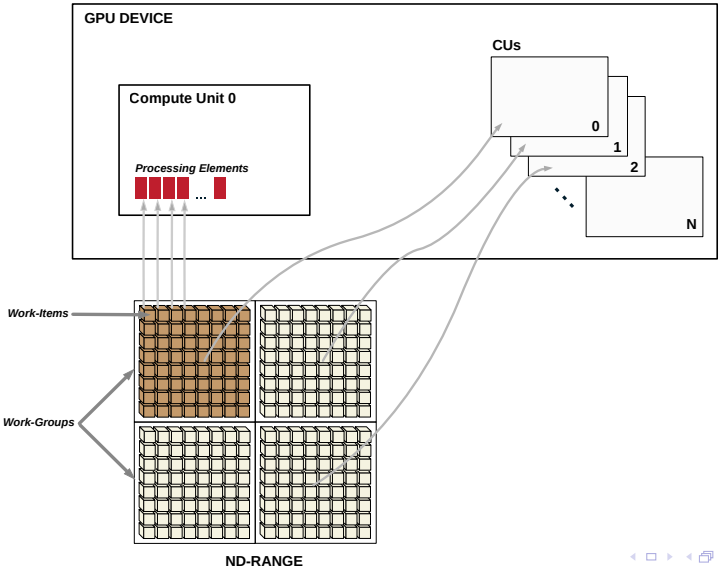
Architecture model: hierarchical organization



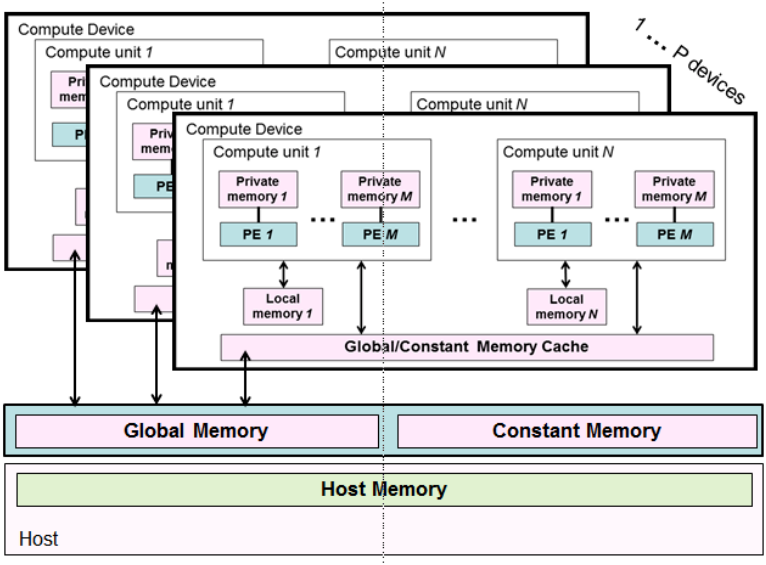
- Host threads launch computational *kernels* on *accelerators*

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

Execution model: embarrassingly parallel



Memory model: express memory locality



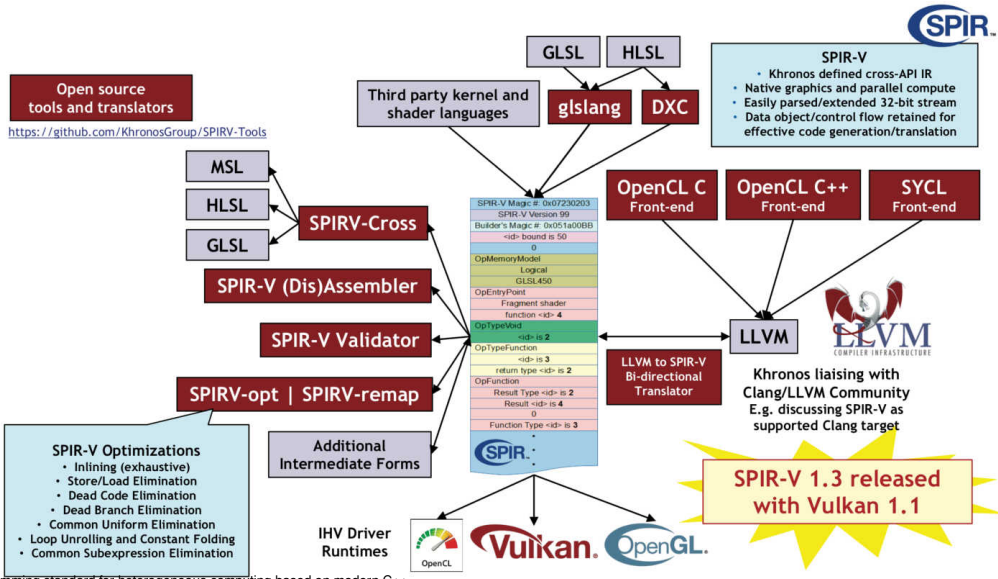
Outline

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++ 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
  - What about the other 92% architectures at HotChips 2019? ☹
- Not single-source & single-language ☹
  - Part of the Frankenstein's programming model... ☹
- 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

Outline

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

Outline

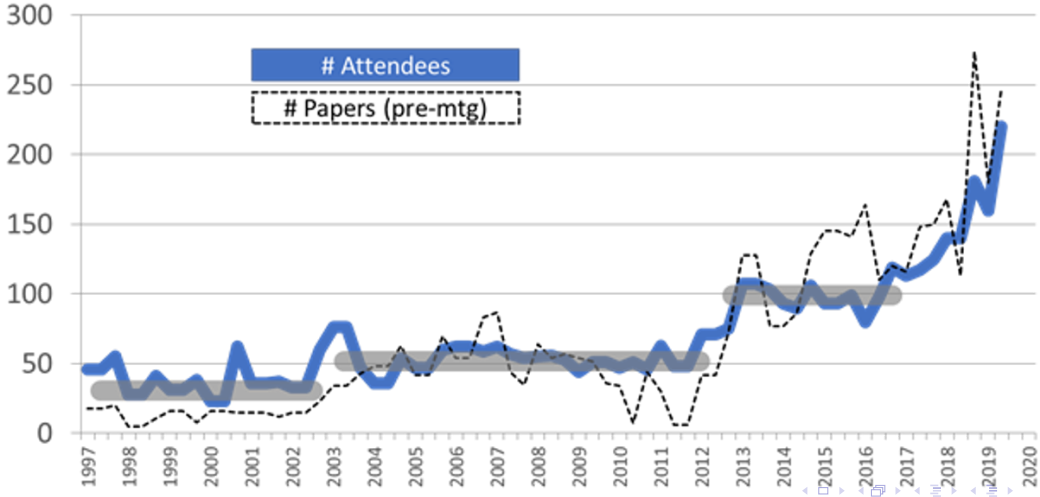
Remember C++ ?

2-line description by Bjarne Stroustrup

- Direct mapping to hardware
- Zero-overhead abstraction

¿ And what about post-modern C++ ?

ISO C++ committee proposal papers & attendance



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

```
#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++ ☺

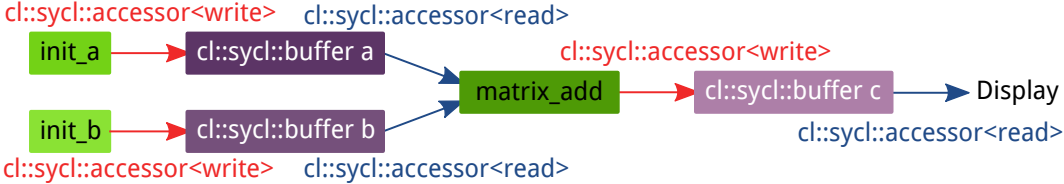
Outline

Complete example of matrix addition in 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 back-end event graph
  - Runtime uses as many threads & back-end queues as necessary

Task graph programming — the full code

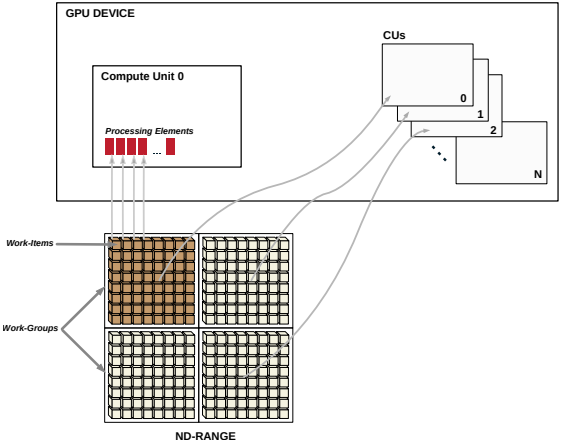
```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;
int main() {
  { // By sticking all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block.
    // Create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

Implement concepts useful for heterogeneous computing

- Modern C++ features available for accelerators
  - Builds on the features of C++11, with additional support for more modern C++
  - Enables ISO C++17 Parallel STL programs to be accelerated
  - 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
- Interoperability with multiple languages according to back-end
  - OpenCL, OpenGL, Vulkan, OpenVX, DirectX, CUDA...
- 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!

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

```
// 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) {
      // 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, [&](h_item<1> tile) {
        filterLocal[tile] = filter_access[tile];
      });
      // There is an implicit work-group barrier here

      range<1> inputRange{ blocksize + 2*radius };

      // Iterate on inputRange work-items
      group.parallel_for_work_item(inputRange, [&](h_item<1> tile) {
        float val = 0.f;
        int readAddress = tile.get_global_id() - radius;
        if (readAddress >= 0 && readAddress < size)
          val = in_access[readAddress];

        localData[tile] = val;
      });
      // There is an implicit work-group barrier here

      // Iterate on all the work-items
      group.parallel_for_work_item([&](h_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 work-group barrier here
      // Iterate on all the work-items
      group.parallel_for_work_item(group, [&](h_item<1> tile)
        out_access[tile.get_global_id(0)] = convolutionResult[tile];
      });
      // There is an implicit work-group barrier here
    });
});
```

Very close to OpenMP 5 style! ☺

- Easy to understand the concept of work-groups
- Easy to write work-group only code
  - Local variables inside parallel_for_work_group are mapped to OpenCL local or CUDA shared address space
- 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 implementation based on OpenCL
- 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 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;
}
```

C++17 STL has parallel algorithms now

- Parallel STL now in C++17 https://en.cppreference.com/w/cpp/algorithm/sort

```
// 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

Outline

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 on... Nvidia GPU
  - Type safety across host/device boundary
  - Avoid spaghetti Frankenstein's programming of OpenCL/OpenGL/Vulkan
  - Based on widely used multi-paradigm C++ language
  - But not cross-platform, not multi-vendor, no DSP, no FPGA or other accelerators... ☹

⇝ SYCL: portable cross-vendor single-source C++ standard from Khronos

- This explains why TensorFlow/Eigen uses CUDA or SYCL with C++...

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

```
auto generic_adder = [] (auto... inputs) {
  // Use a tupple of heterogeneous buffers to wrap the inputs
  auto a = boost::hana::make_tuple(
    buffer<typename decltype(inputs)::value_type>
      { std::begin(inputs), std::end(inputs) }...);
  // The basic computation
  auto compute = [] (auto args) {
    // f(... f(f(f(x1, x2), x3), x4) ..., xn)
    return boost::hana::fold_left(args, [] (auto a, auto b)
                                          { return a + b; });
  };
  // Use the range of the first argument as the range
  // of the result and computation
  auto size = a[0_c].get_count();
  // Infer the type of the output from 1 computation on inputs
  using return_value_type = decltype
    (compute(boost::hana::make_tuple(*std::begin(inputs)...)));
  // Allocate the buffer for the result
  buffer<return_value_type> output { size };
  // Submit a command-group to the device
  queue {}.submit([&] (handler& cgh) {
      // Define the data used as a tuple of read accessors
      auto ka = boost::hana::transform(a, [&] (auto b) {
          return b.template get_access<access::mode::read>(cgh);
        });
      // Data are produced to a write accessor to the output buffer
      auto ko = output.template
        get_access<access::mode::discard_write>(cgh);

      // Define the data-parallel kernel
      cgh.parallel_for<class gen_add>(size, [=] (id<1> i) {
          // Pack operands an elemental computation in a tuple
          auto operands =
            boost::hana::transform(ka, [&] (auto acc) {
              return acc[i];
            });
          // Assign computation on the operands to the elemental result
          ko[i] = compute(operands);
        });
    });
    // Return a host accessor on the output buffer
    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) {
  // Use a tupple of heterogeneous buffers to wrap the inputs
  auto a = boost::hana::make_tuple(
    buffer<typename decltype(inputs)::value_type>
      { std::begin(inputs), std::end(inputs) }...);
  // The element-wise computation
  auto compute = [] (auto args) {
    // f(... f(f(f(x1, x2), x3), x4) ..., xn)
    return boost::hana::fold_left(args, op);
  };
  // Use the range of the first argument as the range
  // of the result and computation */
  auto size = a[0_c].get_count();
  // Infer the type of the output from 1 computation on inputs
  using return_value_type = decltype
    (compute(boost::hana::make_tuple(*std::begin(inputs)...)));
  // Allocate the buffer for the result
  buffer<return_value_type> output { size };
  // Submit a command-group to the device
  queue {}.submit([&] (handler& cgh) {
      // Define the data used as a tuple of read accessors
      auto ka = boost::hana::transform(a, [&] (auto b) {
        return b.template get_access<access::mode::read>(cgh);
      });
    // Data are produced to a write accessor to the output buffer
      auto ko = output.template
        get_access<access::mode::discard_write>(cgh);
      // Define the data-parallel kernel
      cgh.parallel_for<class gen_add>(size, [=] (id<1> i) {
          // Pack operands an elemental computation in a tuple
          auto operands =
            boost::hana::transform(ka, [&] (auto acc) {
              return acc[i];
            });
          // Assign computation on the operands to the elemental result
          ko[i] = compute(operands);
        });
  });
  // Return a host accessor on the output buffer
  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

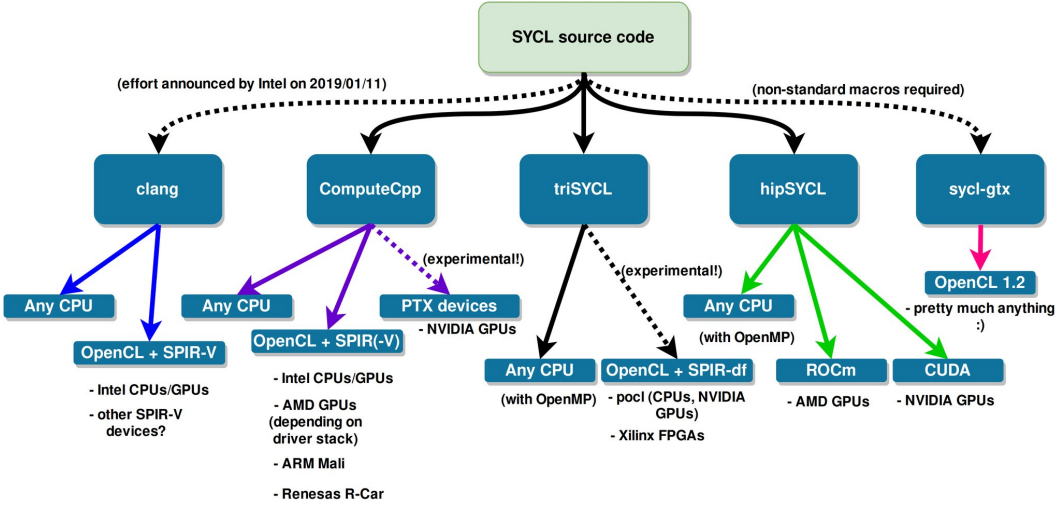
- Alternative implementation of

```
auto compute = [] (auto args) {
  return boost::hana::fold_left(args, [] (auto a, auto b) { return a + b; }
}; // 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
- 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 CGRA for example
- Imagine if SystemC was invented with this future C++ instead of C++98...

Outline

Known implementations of SYCL



Aksel Alpay 2019/01/11 https://twitter.com/illuhad/status/1083863225479892993

ComputeCpp by Codeplay

`https://www.codeplay.com/products/computesuite/computecpp`

- Codeplay is initiator & chair of SYCL Khronos working-group
  - Highly engaged in ISO C++, ADAS (MISRA & AUTOSAR C++), ML, safety critical standards...
  - 2-day tutorial at CppCon 2019 `https://www.codeplay.com/portal/08-05-19-codeplay-at-cppcon-2019-6-talks-2-workshops-and-more`
- First SYCL 1.2.1 full-compliant implementation in July 2018
- Best implementation in class!
  - Started as a developer environment for gaming (Sony PS2 & PS3) in 2000's ☺
  - Clang/LLVM outlining compiler generating SPIR(-V) and other back-ends
  - Runtime for OpenCL device, CPU and other back-ends (CUDA, Vulkan...)
- Free community edition + non-free for customer support
  - Provide several libraries & frameworks such as SYCL versions of Eigen & TensorFlow
- Implement compute & graphics stacks for customers (Renesas, Imagination...)

Clang/LLVM SYCL by Intel

`https://github.com/intel/llvm/tree/sycl`

- Open-sourced in January 2019 with unifying goal: Clang/LLVM up-streaming! ☺
- Target CPU, GPU & Intel FPGA
- Based on open-source SPIR-V LLVM translator
- Runtime for OpenCL
- Part of Intel OneAPI strategy 2018/12/12 `https://www.phoronix.com/scan.php?page=news_item&px=Intel-oneAPI-Announcement`
- 2019/06/19 *Direct programming: One API contains a new direct programming language, Data Parallel C++ (DPC++), an open, cross-industry alternative to single architecture proprietary languages. DPC++ delivers parallel programming productivity and performance using a programming model familiar to developers. DPC++ is based on C++, incorporates SYCL from The Khronos Group and includes language extensions developed in an open community process.*
  `https://newsroom.intel.com/news/intels-one-api-project-delivers-unified-programming-model-across-diverse-arch`

hipSYCL

`https://github.com/illuhad/hipSYCL`

- Started by PhD student Aksel Alpay @ Heidelberg University Computing Centre (URZ), Germany
  - Around 10 developers
- hipSYCL is pure single-source C++ on top of HIP and CUDA
  - Interoperable with AMD ROCm & Nvidia CUDA environments
  - Interoperable with AMD & Nvidia libraries ☺
  - Can directly use AMD & Nvidia C++ intrinsics ☺
    - Nvidia TensorCore, Nvidia ray-tracing, graphics interoperability...
- Can use Clang/LLVM Cuda/HIP or `nvcc`
- Demonstrate that SYCL is more general than OpenCL

triSYCL

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

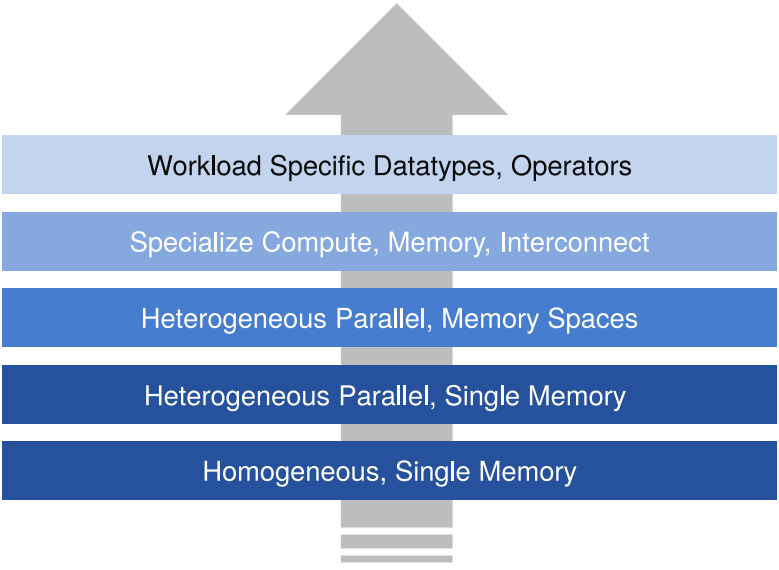
- Open Source SYCL started in 2014 at AMD and now led by Xilinx
  - Used inside Khronos to define the SYCL and OpenCL C++ standard
    - Languages are now too complex to be defined without implementing...
- Use C++20 templated classes
- OpenMP or Intel TBB for host parallelism
- Boost.Compute for OpenCL interaction
- Clang/LLVM device compiler prototype
- Research project targeting Xilinx FPGA and ACAP AIE CGRA
- Experimental fusion with Intel open-source implementation `https://github.com/triSYCL/sycl`

Outline

SYCL as an extensible framework

- 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, SYCoboL...). SYFortranCL on top of Fortran 2018? ☺
- 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

Refinement levels



Domain Lib

Architecture Lib

C++SYCL

C++Parallel Lib

C++20

Open-source extensions as a way to speed-up standardization

- Standardization is a slow process...
- Follow ISO C++ process to speed-up operations ☺
- Publish and implement open-source extensions
  - `https://github.com/codeplaysoftware/standards-proposals`
  - `https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions`
  - `https://github.com/illuhad/hipSYCL/blob/master/doc/extensions.md`
  - `https://github.com/triSYCL/triSYCL/issues?q=is%3Aissue+label%3Aextension+`
- Get feedback from implementers and users ☺
- Can become part of the standard if SYCL members agree
- Some features can be optional to allow maximum performance per platform
  - C++ allows feature testing & adaptability through metaprogramming ☺

Celerity: High-level C++ for Accelerator Clusters

`https://celerity.github.io`

- Scale-out SYCL concepts at the data-center level
  - Based on C++ + SYCL + MPI

```
queue.submit([=](celerity::handler& cgh) {
    auto r_input = input_buf.get_access<cl::sycl::access::mode::read>(cgh, celerity::access::neighborhood<2>(1, 1));
    auto dw_edge = edge_buf.get_access<cl::sycl::access::mode::discard_write>(cgh, celerity::access::one_to_one<2>());
    cgh.parallel_for<class MyEdgeDetectionKernel>(
        cl::sycl::range<2>(img_height - 1, img_width - 1),
        cl::sycl::id<2>(1, 1),
        [=](cl::sycl::item<2> item) {
          int sum = r_input[{item[0] + 1, item[1]}] + r_input[{item[0] - 1, item[1]}]
                  + r_input[{item[0], item[1] + 1}] + r_input[{item[0], item[1] - 1}];
          dw_edge[item] = 255 - std::max(0, sum - (4 * r_input[item]));
        }
    );
});
queue.with_master_access([=](celerity::handler& cgh) {
    auto out = edge_buf.get_access<cl::sycl::access::mode::read>(cgh, edge_buf.get_range());
    cgh.run([=]() {
        stbi_write_png("result.png", img_width, img_height, 1, out.get_pointer(), 0);
    });
});
```

- Other C++ HPC frameworks like HPX or Kokkos provide similar concepts on top of SYCL but with a non-SYCL syntax

Simplify porting code from CUDA

- SYCL is higher-level than CUDA to ease programmer life ☺
  - Buffers for abstract storage, accessors to express dependencies...
- But when porting CUDA code: dealing with raw device pointers, explicit kernel launches without any dependencies... ☹
  - Feedback from porting Eigen & TensorFlow...
  - ⇝ Paradox: cumbersome to fit in standard SYCL model... ☹
- `https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions/OrderedQueue` Ordered queue, in-order execution, like CUDA default stream
- `https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/USM` Unified Shared Memory like in OpenMP 5, various level of sharing according to hardware

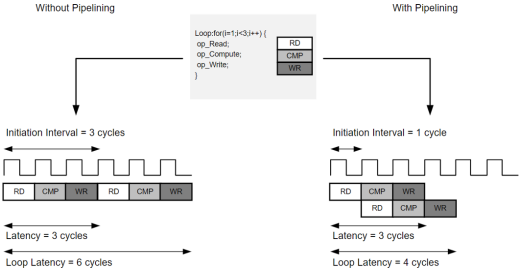
```
sycl::ordered_queue q;
auto dev = q.get_device();
auto ctxt = q.get_context();
float* a = static_cast<float*>(sycl::malloc_shared(10*sizeof(float), dev, ctxt));
float* b = static_cast<float*>(sycl::malloc_shared(10*sizeof(float), dev, ctxt));
float* c = static_cast<float*>(sycl::malloc_shared(10*sizeof(float), dev, ctxt));
q.submit([&](handler& cgh) {
  cgh.parallel_for<class vec_add>(range<1> {10}, [=](id<1> i) {
    c[i] = a[i] + b[i];
  });
});
```

Outline

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



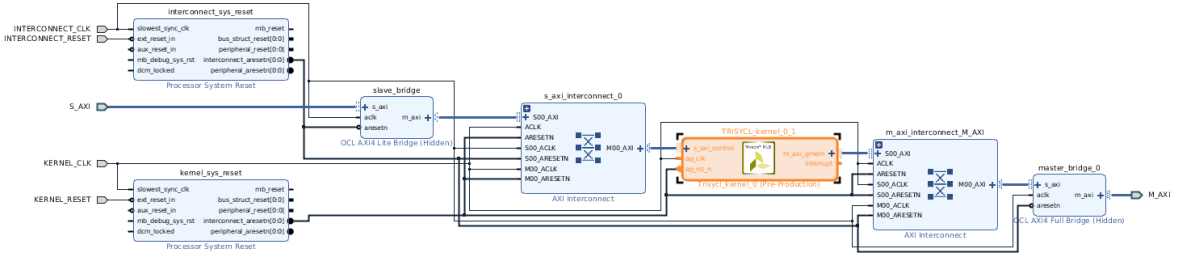
Decorating code for FPGA pipelining in triSYCL

```
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;
      });
    }
  }
}
```

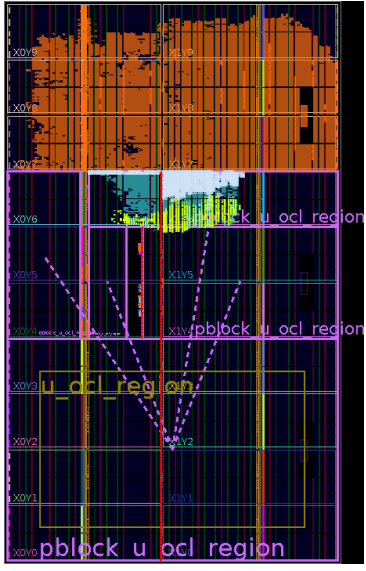
- Use native C++ construct instead of alien #pragma or attribute (vendor OpenCL or HLS C++...)
- Compatible with metaprogramming
- Implementation
  - No need for specific parser/tool-chain!
  - Just use lambda + intrinsics! ☺

```
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();
};
}
```

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



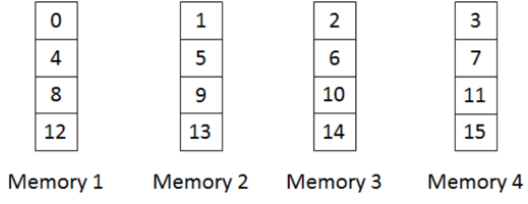
After Xilinx SDx `xocc` ingestion... Layout in Vivado



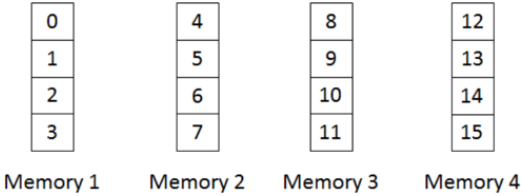
Partitioning memories *(I)*

- 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

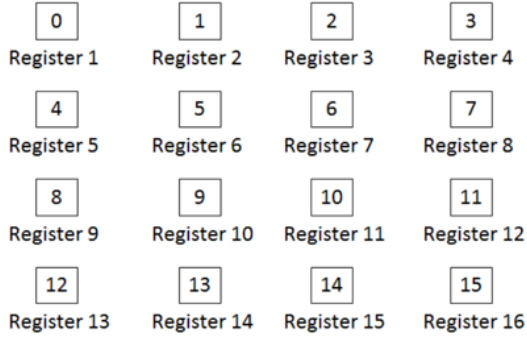


Partitioning memories (II)



- 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++

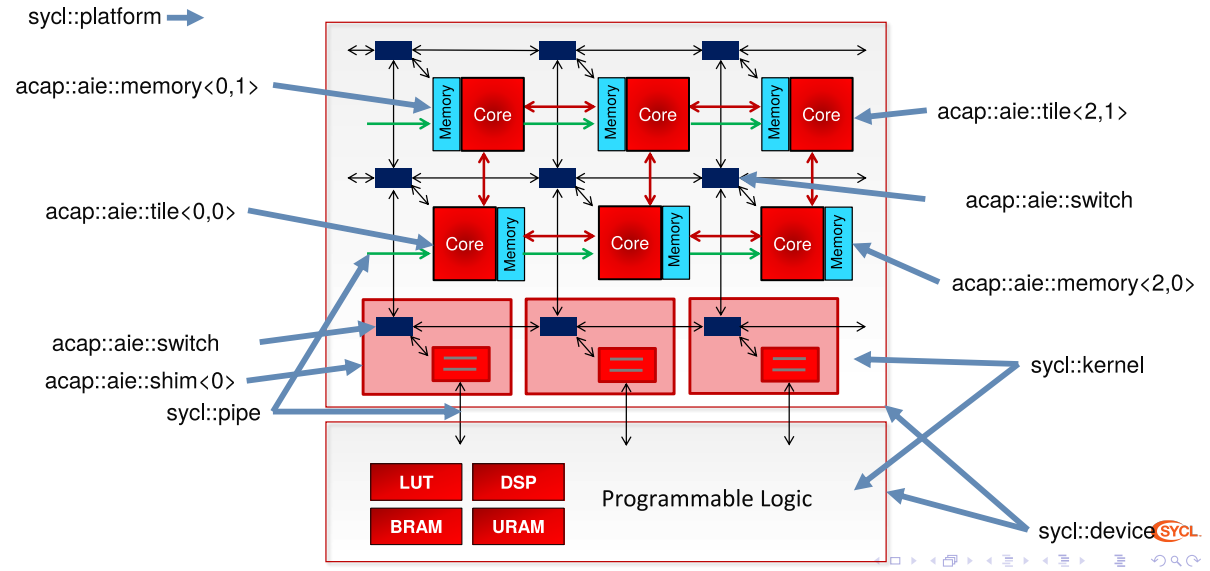
```
int A[MAX_DIM * MAX_DIM];
int B[MAX_DIM * MAX_DIM];
int C[MAX_DIM * MAX_DIM];
#pragma HLS ARRAY_PARTITION variable=A dim=1 \
                            cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 \
                            block factor=64
```

- 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];
```

Outline

Xilinx C++ ACAP: templated 2D SYCL abstractions



C++ ACAP abstractions for the ultra-performance programmer

- Executable DSEL in modern C++
  - Allows debug with CPU tools on laptop (ThreadSanitizer, Valgrind, GDB....)
  - Allow architecture co-design before silicon. No need for expensive RTL emulator/simulator to start with ☺
- 2D layout and connectivity
- AIE cores
- Memory modules
  - Banks
  - Locking units
- DMA
- Cascade stream
- Routing resource
  - Virtual circuit & packet switching
  - Routing tables & broadcast mode

  Interconnection with FPGA programmable logic (PL), GPU, host...
- Events I/O & configuration/modes

Outline

Resources

- https://www.khronos.org/sycl
  - https://www.khronos.org/files/sycl/sycl-121-reference-card.pdf 8 pages of SYCL &
- https://sycl.tech community content
- https://tech.io/playgrounds/48226/introduction-to-sycl/introduction-to-sycl-2 playground on “Introduction to SYCL” with on-line code execution
- Code samples from Codeplay
  https://github.com/codeplaysoftware/computecpp-sdk/tree/master/samples
- Parallel Research Kernels implemented with different frameworks, including SYCL
  - Useful to compare or translate between different programming models
  - https://github.com/ParRes/Kernels

Puns and pronunciation explained

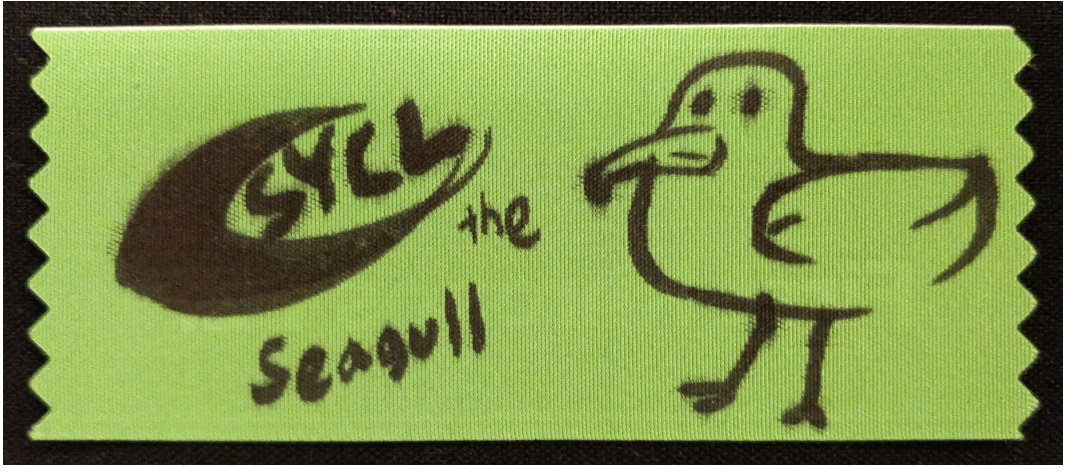
SYCL

sickle ['si-kəl]

SPIR

spear ['spir]

SYCL mascot: SYCL the seagull!



Thanks to Dominic Agoro-Ombaka during Khronos face-to-face meeting in Montréal !

Conclusion

- Heterogeneous computing is everywhere and here to stay
  - *"The next decade will see a Cambrian explosion of novel computer architectures"*
  - Full systems & HPC machines add other complexity levels...
- SYCL C++ standard from Khronos Group
  - In post-modern C++ we trust: only way to control the full stack of real hardware. It's not magic, it's C++! ☺
  - Pure modern C++ DSEL for heterogeneous computing
  - Single-source & single-language to metaprogram the full architecture
  - ISO C++ is an elitist democracy: only what is implemented and working is ratified
    - SYCL implementations ⇝ feedback for ISO C++ WG21 SG14 standard...
- Open-source + open standards
  - No user locked-in!
  - Various implementation with various back-ends
  - Interoperability with other ecosystems (OpenCL, CUDA, Vulkan, proprietary...)
- You are not happy? Do it the standard way! Participate to the standards to have a real impact! ☺