Single-source SYCL C++ on Xilinx FPGA
Xilinx Research Labs
Khronos booth @SC17 2017/11/12—19
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...
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
Zynq UltraScale+ MPSoC overview
Multiple power domains
- Low power domain
- Full power domain
- PL power domain

Power gating
- A53 per core
- L2 and OCM RAM
- GPU, USB
- R5s & TCM
- Video CODEC

Sleep Mode
- 35mW sleep mode
- Suspend to DDR with power off
Programmable Logic (PL): Field-Programmable Gate Array (FPGA)
Basic architecture =
Lookup Table + Flip-Flop storage + Interconnect

Typical Xilinx LUT have 6 inputs
Global view of programmable logic part
DSP48 block overview
Once upon a time the Xilinx XC2064...

1985: the First FPGA
- 64 flip-flops
- 128 3-LUT
- 58 I/O pins
- 18 MHz (toggle)
- 2 µm 2LM
Since then...

- 10,000x more logic...
  - Plus embedded IP
    - Memory
    - Microprocessors
    - DSP
    - Gigabit+ Serial I/O

- 100x faster
- 5,000x lower power/gate
- 10,000x lower cost/gate

Zynq UltraScale+ MPSoC overview: All Programmable...
...Xilinx Zynq UltraScale+ MPSoC programming

- **Vivado**
  - Hardware basic block integration
  - RTL (Verilog & VHDL) programming

- **Vivado HLS**
  - C & C++ high-level synthesis
  - Down to LUT, DSP & BRAM...

- **SDAccel**
  - Khronos Group OpenCL

- **SDSoC**
  - C & C++ with #pragma

- **SDNet**
  - Generate routers from network protocol description

- **Various libraries**
  - OpenCV, DNN...

- **Linux**
  - Usual CPU multicore programming

- **OpenAMP**
  - Real-time ARM R5
Position argument

Which language for unified heterogeneous computing?

- Entry cost

- ∃ thousands of dead parallel languages...

- Exit cost

- Use standard solutions with open source implementations
Remember C++?

2-line description by Bjarne Stroustrup

- Direct mapping to hardware
- Zero-overhead abstraction

⇒ Unique existing position in embedded system to control the full stack!!

- C++ rebooted in 2011
  - 1 new version every 3 years
  - Shipping what is implemented

- Easier
  - Simpler to do simple things

- More powerful
  - Possible to do more complex things
Position argument 2: start with modern C++...

- Very successful & ubiquitous language
- Interoperability: seamless interaction with embedded world, libraries, OS...
- Full-stack: combine both low-level aspects with high-level programming
  - Pay only for what you need
- Open-source production-grade compilers (GCC & Clang/LLVM) & tools
- Classes can be used to define Domain Specific Embedded Language (DSEL)
- Not directly targeting FPGA, GPU, DSP...
  - But extensible through classes (→ DSEL)
  - Extensible with #pragma (already in Xilinx tools Vivado HLS, SDSoC & SDAccel) and attributes
Even better with modern C++ (C++14, C++17, C++2a)

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

```cpp
std::vector my_vector { 1, 2, 3, 4, 5 };

// Display each element
for (auto e : my_vector)
    std::cout << e;

// Increment each element
for (auto &e : my_vector)
    e += 1;
```
Modern C++: like Python but with speed and type safety

➤ Python 3.x (interpreted):

```python
def add(x, y):
    return x + y

print(add(2, 3))  # Output: 5
print(add("2", "3"))  # Output: 23
print(add(2, "Boom"))  # Fails at run-time :-(
```

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

```cpp
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 :-(
```

➤ Automatic type inference for terse generic programming and type safety

– Without `template` keyword!
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
  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;
}

Terse generic programming and type safety
  – Without template keyword!
```
No tool providing...

- Modern C++ environment
- Heterogeneous computing (FPGA...) à la ISO C++ SG14 (low latency, HPC, embedded...)

- **Single source** for programming productivity
  - Templated code across kernels
  - Type safety across heterogeneous execution

- Backed by **open standard**

- OpenCL interoperability to recycle other libraries and portability
Complete example of matrix addition in OpenCL SYCL

```cpp
#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 released
    // Copy back the buffer data with RAII behaviour.
    std::cout << "c[0][2] = " << c[0][2] << std::endl;
    return 0;
}
```
SYCL 2.2 = pure C++17 DSEL

- Implement concepts useful for heterogeneous computing
- Asynchronous task graph
- Hierarchical parallelism & kernel-side enqueue
- Queues to direct computations on devices
- Single-source programming model
  - Take advantage of CUDA on steroids & OpenMP simplicity and power
  - Compiled for host and device(s)
  - Enabling the creation of C++ higher level programming models & C++ templated libraries
  - System-level programming (SYstemCL)
- Buffers to define location-independent storage

- Accessors to express usage for buffers and pipes: read/write/...
  - No explicit data motion
  - Automatic overlapping of communication/computation

- Hierarchical storage
  - Rely on C++ allocator to specify storage (SVM...)
  - Usual OpenCL-style global/local/private

- Most modern C++ features available for OpenCL
  - Programming interface based on abstraction of OpenCL components (data management, error handling...)
  - Provide OpenCL interoperability

- Directly executable DSEL
  - Host fall-back & emulation for free
  - No specific compiler needed for experimenting on host
  - Debug and symmetry for SIMD/multithread on host
Asynchronous task graph model in SYCL

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

```
cl::sycl::accessor<write> cl::sycl::accessor<read>
init_a  cl::sycl::buffer  cl::sycl::accessor<write>
          matrix_add  cl::sycl::buffer  Display
init_b  cl::sycl::buffer

cl::sycl::accessor<write> cl::sycl::accessor<read>
```

- Possible schedule by SYCL runtime:
  - init_b init_a matrix_add Display

- 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
#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);

        // 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 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);

        // Enqueue a parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class matrix_add>({N, M},
        [=] (auto 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>();

    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;
}
OpenCL interoperability mode

→ SYCL → Very generic parallel model

→ Allows interaction with OpenCL/Vulkan/OpenGL without overhead

→ Keeps the high-level features of SYCL
  – No explicit buffer transfer
  – Task and data dependency graphs
  – Templated C++ code
  ...

→ The user can call any existing OpenCL kernel
  – Even HLS C++ & RTL Xilinx FPGA kernels!
  – Avoid writing painful OpenCL C/C++ host code
OpenCL built-in kernels

- OpenCL built-in kernels are very common in FPGA world
- Written in Verilog/VHDL or Vivado HLS C++
  - But with SDAccel OpenCL kernel interface

Typical use cases
- Kernel libraries
- Linear algebra
- Machine learning
- Computer vision
- Direct access to hardware: wire-speed Ethernet…

SYCL OpenCL interoperability mode can be used to simplify usage of these kernels
Using OpenCL interoperability mode in SYCL

#include <CL/sycl.hpp>
using namespace cl::sycl;

cconstexpr 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 default 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, int offset) { 
            c[get_global_id(0)] = a[get_global_id(0)] + b[get_global_id(0)] 
                + offset; 
        }")", 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), 
            cl::sycl::cl_int { 42 } );
        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;
}
Implementations
Known implementations of SYCL

- **ComputeCpp by Codeplay** [https://www.codeplay.com/products/computecpp](https://www.codeplay.com/products/computecpp)
  - Most advanced SYCL 1.2 implementation
  - Outlining compiler generating SPIR
  - Run on any OpenCL device and CPU
  - Can run TensorFlow SYCL

- **sycl-gtx** [https://github.com/ProGTX/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](https://github.com/triSYCL/triSYCL)
triSYCL

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

» Pure C++ implementation & CPU-only implementation for now
  - Use OpenMP for computation on CPU + \texttt{std::thread} for task graph
  - Rely on STL & Boost for zen style
  - CPU emulation for free
    • Quite useful for debugging
  - More focused on correctness than performance for now (array bound check...)

» Provide OpenCL-interoperability mode: can reuse existing OpenCL code

» Some extensions (Xilinx blocking pipes)

» On-going outlining compiler based on open-source Clang/LLVM compiler
#include <CL/sycl.hpp>

```c++
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];
    });
});
```

C++ SYCL

Unmodified host compiler (gcc/clang/vs/icc)

For OpenCL interoperability

libOpenCL.so

Clang/LLVM

Host & kernel caller

OpenMP CPU executable

OpenMP CPU executable

Clang/LLVM
device compiler

kernels.bin

SPIR 2.0

“de facto”

Vendor OpenCL
device compiler

C++17 & OpenMP & Boost

SYCL runtime

OpenCL interoperability (Boost.Compute)

Device Compiler

Runtime

© Copyright 2017 Xilinx
```cpp
#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>([=]
            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;
    });
}
```

Example of compilation to device (FPGA...)

Current limitation:
need to write this

```
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>([=,]
        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 with Clang 3.9.1

using: ModuleID = 'device_compiler/single_task_vector_add_drt bcm'
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 __x86_personality_v0(...) 

; Function Attrs: noinline norecurse nounwind uwtable

define spir_kernel void @TRISYCL_kernel_0(i32 addrspace(i) %f.0.0.0.val, i32 addrspace(i) %f.0.2.0.val) unnamed_addr #0 !kernel_arg_addr_space 14 !kernel_arg_base_type 14 !kernel_arg_type 15 !kernel_arg_access_qual 16 (!llvm.ident = "(!)")

eentry:
  br label %for.body.i

for.body.i:
  %indvars.iv.i = phi i64 [ 0, %entry ], [ %indvars.iv.next.i, %for.body.i ]
  %arrayidx.i.i = getelementptr inbounds i32, i32 addrspace(i) %f.0.1.0.val, i64 %indvars.iv.i
  %0 = load i32, i32 addrspace(i) %arrayidx.i.i, align 4, !tbaa !
  %arrayidx.i.i5 = getelementptr inbounds i32, i32 addrspace(i) %f.0.2.0.val, i64 %indvars.iv.i
  %1 = load i32, i32 addrspace(i) %arrayidx.i.i5, align 4, !tbaa !
  %add.i = add nsw i32 %1, %0
  %arrayidx.i.i3 = getelementptr inbounds i32, i32 addrspace(i) %f.0.0.0.val, i64 %indvars.iv.i
  store i32 %add.i, i32 addrspace(i) %arrayidx.i.i3, align 4, !tbaa !
  %indvars.iv.next.i = add nsw i64 %indvars.iv.i, 1
  %exitcond.i = icmp eq i64 %indvars.iv.next.i, 300
  br i1 %exitcond.i, label %"_Z12test_mainiPPcENK3S_1clERN2cl14sycl7handlerEENKULvE_clEv.exit": %preds = %for.body.i
  ret void

attributes !0 = !{ nolinear norecurse nounwind uwtable "disable-tail-calls"="false" "less-precise-fp-math"="false" "no-frame-pointer-elim="false" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "stack-protector-buffer-size="8" "target-cpu="x86-64" "target-features="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

"_Z12test_mainiPPcENK3S_1clERN2cl14sycl7handlerEENKULvE_clEv.exit": ; preds = %for.body.i

After Xilinx SDx 2017.2 xocc ingestion...
After Xilinx SDx 2017.2 xocc ingestion... FPGA layout!
Code execution on real FPGA

rkeryell@xirjoant40:~/Xilinx/Projects/OpenCL/SYCL/triSYCL/branch/device/tests (device) $
device_compiler/single_task_vector_add_drt.kernel_caller
binary_size = 5978794

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

Setting up

**no errors detected**
Behind the curtain...
Design strategy

➤ Rely on open-source and post-modernism
  – Do not solve again solved problems
  – Hardware companies often lag behind… 😞
  – Eagerly procrastinate
    • Wait for the others to do your work 😊
    • The best code we can write is the code we do not write (no bug, at least!)

➤ Develop as much 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++)
Open-source developments

» Runtime for device compiler not merged yet
  – https://github.com/triSYCL/triSYCL/tree/device

» LLVM https://github.com/triSYCL/llvm/tree/ronan/sycl

» Clang https://github.com/triSYCL/clang/tree/ronan/sycl

» Please contribute 😊
Device compiler

▶ For now implemented in Makefile 😊

  – Need to build a Clang driver at some point

# Process bitcode with SYCL passes to generate kernels
%.kernel.bc: %.pre_kernel.ll

$(LLVM_BUILD_DIR)/bin/opt -load $(LLVM_BUILD_DIR)/lib/SYCL.so \ 
  -globalopt -deadargelim -argpromotion -deadargelim \ 
  -SYCL-kernel-filter -globaldce -RELGCD -inSPIRation \ 
  -o $@ $<

# Process bitcode with SYCL passes to generate kernel callers
%.kernel_caller.bc: %.pre_kernel_caller.ll

$(LLVM_BUILD_DIR)/bin/opt -load $(LLVM_BUILD_DIR)/lib/SYCL.so \ 
  -globalopt -deadargelim -SYCL-args-flattening \ 
  -SYCL-argserialize-arguments -deadargelim -o $@ $<
triSYCL device compiler ideas

➤ Use a lot of C++ tricks in SYCL runtime to mark kernels and so on

➤ Compile with Clang -O3 -fno-vectorize -fno-unroll-loops
  – Generated very optimized kernel code
  – But do not generate futuristic vectorized SPIR (but still work with PoCL…)

➤ Mark and sweep approach for kernel selection
  – Mark all non kernel code with internal linkage
  – Apply existing dead-code elimination pass

➤ C++ made it harder
  – Crash SPIR consumers…
  – New RELGCD pass to remove variable storing former static constructors/destructors
  – Rename kernels because mangled name ($...) crashed some SPIR consumers
SYCL for generic C++ libraries
Single-source C++ enables generic libraries

For example:

¿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++, SDSoC, HLS… 😞

But possible in SYCL, CUDA (nVidia), C++AMP (Microsoft), HCC (AMD)

Explains why TensorFlow/Eigen uses CUDA, SYCL or HCC…
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) {
    return boost::hana::fold_left(args, [] (auto x, auto y) { return x + y; });
};  
auto pseudo_result = compute(boost::hana::make_tuple(*std::begin(inputs)...));
using return_value_type = decltype(pseudo_result);
auto size = a[0_c].get_count();  
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<i> 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 }, v { 5, 6, 7 };
    for (auto e : generic_adder(u, v))
        std::cout << e << ' ';
    std::cout << std::endl;
    std::vector<float> a { 1, 2.5, 3.25, 10.125 }, b { 5, 6, 7, 2 }, 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
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) {
        return boost::hana::fold_left(args, op);
    };
    auto pseudo_result = compute(boost::hana::make_tuple(*std::begin(inputs) ...));
    using return_value_type = decltype(pseudo_result);
    auto size = a[0_c].get_count();
    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<> 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; },
                                      a, b, c))
        std::cout << e << ' ';
    std::cout << std::endl;
    return 0;
}
Alternative implementation of

```cpp
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 Boost.HANA 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...
– Imagine if SystemC was invented with C++2a instead of C++98...
TensorFlow SYCL

- Development started in 2015 with Google & Codeplay
- TensorFlow based on C++ Eigen with metaprogramming for kernel fusion
  → Single source: CUDA for GPU, OpenCL SYCL
- Weekly meeting with Google, Codeplay, Oracle, Xilinx
- SYCL version natively available in https://github.com/tensorflow/tensorflow
- 2 use cases for Xilinx
  - Native SYCL with device compiler: transform C++ kernels into FPGA kernels (HLS style)
  - OpenCL interoperability mode: easy way to hook existing DNN accelerators
import tensorflow as tf

sess = tf.InteractiveSession()
file_writer = tf.summary.FileWriter('logs', sess.graph)

# To output a new version of the graph:
def ug():
    file_writer.add_graph(sess.graph)
    file_writer.flush()

coeff = tf.constant(3.0, tf.float32, name = "Coeff")
a = tf.placeholder(tf.float32, name = "A")
b = tf.placeholder(tf.float32, name = "B")

with tf.device(tf.DeviceSpec(device_type="SYCL")):
    product = tf.multiply(coeff, a, name = "Mul")

with tf.device(tf.DeviceSpec(device_type="CPU")):
    linear_model = tf.add(product, b, name = "Add")

print(sess.run(linear_model, {a : 3, b : 4.5 })))

ug()

13.5
OpenCL interoperability with Tensorflow

A Tensorflow operation was also added

– Uses the Eigen operation in the back-end
– We get a Python interface for free!

```python
conf = tf.ConfigProto(allow_soft_placement=False)
sess = tf.InteractiveSession(config=conf)

with tf.device('/cpu:0'):
in1 = tf.fill([6,3,2], 18.0, name="in1")
in2 = tf.fill([6,3,2], 12.0, name="in2")
in3 = tf.fill([6,3,2], 2.0, name="in3")

with tf.device('/device:SYCL:0'):
    result = tf.user_ops.ocl_native_op(input_list=[in1, in2, in3], output_type=tf.float32, shape=[6,3,2], file_name="/path/to/kernel.cl", kernel_name="vector_add", is_binary=False)

print(result.eval())
```

TensorBoard graph:
OpenCL interoperability in TensorFlow: enable new features

- Can use smaller data types than the ones available in TensorFlow

- DoReFa-Net (Pruned) – AlexNet like
  - 3915 Images/sec inference
  - 1b Weights, 2b Activations
  - 8.54 TOPS @ 109Mhz
  - 0.432msec latency

  ![Diagram of DoReFa-Net](image)

  - Amazon AWS F1 instance - 1x Xilinx VU9P FPGA
  - Host source: C++, with Khronos OpenCL C++ bindings
  - Kernel source: Xilinx Vivado HLS C++ with OpenCL-compatible kernel API
Conclusion
SYCL: generic heterogeneous programming model

- SYCL could target MCA API, OpenAMP, Vulkan, OpenGL, OpenSHMEM, MPI, StarPU, Nanos++, proprietary DSP API, OpenMP, HSA, CUDA...
- Possible to retarget triSYCL?
- triSYCL architecture is rather generic
- Use generic programming, C++17, STL & Boost for terse programming
  - Do not use directly OpenCL but relies on higher-level C++ Boost.Compute
- Use dynamic polymorphism to target different realms
  - For now deal with host and OpenCL


- Generic model to target other parts of Zynq UltraScale+ MPSoC
Conclusion

- SYCL bring heterogeneous computing to modern C++
  - Candidate for ISO C++ SG1 & SG14

- Provide a continuous path from old C spaghetti code to high-level modern C++17

- Open standards need open-source implementations for acceptance
  - triSYCL: pure C++17, OpenMP and Boost for CPU and OpenCL-compatible devices
  - On-going implementation of device compiler with Clang/LLVM to SPIR-df
    - Full single-source experience
    - Leverage Xilinx SDx & HLS tools to target Xilinx FPGA & Zynq UltraScale+ MPSoC

- Interoperability mode quite useful to simplify plain OpenCL programming (FPGA…)

- SYCL Parallel STL updated for triSYCL

- On-going work to have Google TensorFlow working with triSYCL
  - Improve TensorFlow, triSYCL and SYCL specification

- Join us and improve your C++17/C++20 & Boost skills!