SYCL COMPILER

zero-cost abstraction and type safety for heterogeneous computing

Andrew Savonichev, 2019 European LLVM Developers Meeting

Agenda

- What is SYCL?
- "Hello, world!" in SYCL
- Scheduler
- Single source
- Compilation flow
- SPIR-V
- Integration header
- Upstream plan

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

What is SYCL?

SYCL is a cross platform abstraction layer for heterogeneous compute developed by Khronos Group.

It allows code for heterogeneous processors (CPU, GPU, FPGA, etc.) to be written in a “single-source” style using standard C++11.

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

“Hello, world!”

```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
}
```

“Hello, world!”

```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
  cl::sycl::buffer<T, 1> bufferA(A.data(), A.size());
  cl::sycl::buffer<T, 1> bufferB(B.data(), B.size());
  cl::sycl::buffer<T, 1> bufferC(C.data(), C.size());
}
```

Step 1: create buffers (represent both host and device memory)

“Hello, world!”

```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
  cl::sycl::buffer<T, 1> bufferA(A.data(), A.size());
  cl::sycl::buffer<T, 1> bufferB(B.data(), B.size());
  cl::sycl::buffer<T, 1> bufferC(C.data(), C.size());

  cl::sycl::queue deviceQueue;
}
```

Step 2: create a queue on a default device (you can optionally specify a device type)

“Hello, world!”

```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
  cl::sycl::buffer<T, 1> bufferA(A.data(), A.size());
  cl::sycl::buffer<T, 1> bufferB(B.data(), B.size());
  cl::sycl::buffer<T, 1> bufferC(C.data(), C.size());

  cl::sycl::queue deviceQueue;
  deviceQueue.submit([&](cl::sycl::handler& cgh) {
    ...
  });
}
```

Step 3: submit an operation for (asynchronous) execution

“Hello, world!”

```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
  cl::sycl::buffer<T, 1> bufferA(A.data(), A.size());
  cl::sycl::buffer<T, 1> bufferB(B.data(), B.size());
  cl::sycl::buffer<T, 1> bufferC(C.data(), C.size());

  cl::sycl::queue deviceQueue;
  deviceQueue.submit([&](cl::sycl::handler& cgh) {
    auto accessorA = bufferA.get_access<sycl_read>(cgh);
    auto accessorB = bufferB.get_access<sycl_read>(cgh);
    auto accessorC = bufferC.get_access<sycl_write>(cgh);
  });
}
```

Step 4: create buffer accessors: represent a buffer + an access type

“Hello, world!”

```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
  cl::sycl::buffer<T, 1> bufferA(A.data(), A.size());
  cl::sycl::buffer<T, 1> bufferB(B.data(), B.size());
  cl::sycl::buffer<T, 1> bufferC(C.data(), C.size());

  cl::sycl::queue deviceQueue;
  deviceQueue.submit([&](cl::sycl::handler& cgh) {
    auto accessorA = bufferA.get_access<sycl_read>(cgh);
    auto accessorB = bufferB.get_access<sycl_read>(cgh);
    auto accessorC = bufferC.get_access<sycl_write>(cgh);

    cgh.parallel_for<class vec_add>(cl::sycl::range<1>(A.size()),
        [=](cl::sycl::id<1> wiID) {
          ...
        });
  });
}
```

Step 6: create a kernel with name and NDRange dimensions

Optimization Notice

“Hello, world!”

```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
  cl::sycl::buffer<T, 1> bufferA(A.data(), A.size());
  cl::sycl::buffer<T, 1> bufferB(B.data(), B.size());
  cl::sycl::buffer<T, 1> bufferC(C.data(), C.size());

  cl::sycl::queue deviceQueue;
  deviceQueue.submit([&](cl::sycl::handler& cgh) {
    auto accessorA = bufferA.get_access<sycl_read>(cgh);
    auto accessorB = bufferB.get_access<sycl_read>(cgh);
    auto accessorC = bufferC.get_access<sycl_write>(cgh);

    cgh.parallel_for<class vec_add>(cl::sycl::range<1>(A.size()),
        [=](cl::sycl::id<1> wiID) {
          accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
        });
  });
}
```

Step 7: write a kernel

“Hello, world!”

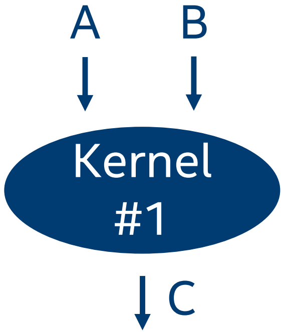
```
template <typename T>
void vector_add(const std::vector<T>& A, const std::vector<T>& B, std::vector<T>& C) {
  cl::sycl::buffer<T, 1> bufferA(A.data(), A.size());
  cl::sycl::buffer<T, 1> bufferB(B.data(), B.size());
  cl::sycl::buffer<T, 1> bufferC(C.data(), C.size());

  cl::sycl::queue deviceQueue;
  deviceQueue.submit([&](cl::sycl::handler& cgh) {
    auto accessorA = bufferA.get_access<sycl_read>(cgh);
    auto accessorB = bufferB.get_access<sycl_read>(cgh);
    auto accessorC = bufferC.get_access<sycl_write>(cgh);

    cgh.parallel_for<class vec_add>(cl::sycl::range<1>(A.size()),
        [=](cl::sycl::id<1> wiID) {
          accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
        });
  });
}
```

That’s it! Buffer C will be memcpy’ed back to a host when bufferC goes out of scope.

SCHEDULER

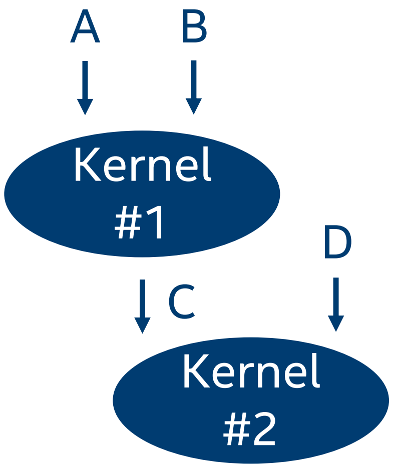
Scheduler

```
deviceQueue.submit([&](cl::sycl::handler& cgh) {
   auto accessorA = bufferA.get_access<sycl_read>(cgh);
   auto accessorB = bufferB.get_access<sycl_read>(cgh);
   auto accessorC = bufferC.get_access<sycl_write>(cgh);

   cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(A.size()),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
 });
```



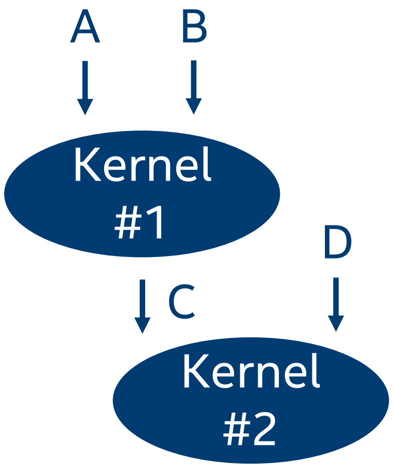
Scheduler

```
deviceQueue.submit([&](cl::sycl::handler& cgh) {
   auto accessorA = bufferA.get_access<sycl_read>(cgh);
   auto accessorB = bufferB.get_access<sycl_read>(cgh);
   auto accessorC = bufferC.get_access<sycl_write>(cgh);

   cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(A.size()),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
 });

deviceQueue.submit([&](cl::sycl::handler& cgh) {
   auto accessorC = bufferC.get_access<sycl_read>(cgh);
   auto accessorD = bufferD.get_access<sycl_read>(cgh);
   auto accessorE = bufferE.get_access<sycl_write>(cgh);

   cgh.parallel_for<class vec_add>(cl::sycl::range<1>(C.size()),
       [=](cl::sycl::id<1> wiID) {
         accessorE[wiID] = accessorC[wiID] + accessorD[wiID];
       });
 });
```



Scheduler

```
deviceQueue.submit([&](cl::sycl::handler& cgh) {
   auto accessorA = bufferA.get_access<sycl_read>(cgh);
   auto accessorB = bufferB.get_access<sycl_read>(cgh);
   auto accessorC = bufferC.get_access<sycl_write>(cgh);

   cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(A.size()),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
 });

deviceQueue.submit([&](cl::sycl::handler& cgh) {
   auto accessorC = bufferC.get_access<sycl_read>(cgh);
   auto accessorD = bufferD.get_access<sycl_read>(cgh);
   auto accessorE = bufferE.get_access<sycl_write>(cgh);

   cgh.parallel_for<class vec_add>(cl::sycl::range<1>(C.size()),
       [=](cl::sycl::id<1> wiID) {
         accessorE[wiID] = accessorC[wiID] + accessorD[wiID];
       });
 });
```



No explicit “wait” operation! SYCL runtime is responsible for synchronization.

SINGLE SOURCE MODEL

Single source - single type system

OpenCL code:

```
host.cpp:

struct point {
    char x;
    char y;
};

point p = {0, 1};
clSetKernelArg(kern, p);
clEnqueueKernel(kern);
```

```
kernel.cl:

struct point {
    char x;
    char y;
};

kernel foo(point *p) {
    assert(p->x == 0 &&
                p->y == 1);
}
```

Single source - single type system

OpenCL code:

```
host.cpp:

struct point {
    char x;
    char y;
};

point p = {0, 1};
clSetKernelArg(kern, p);
clEnqueueKernel(kern);
```

Structs may have different layout!

```
kernel.cl:

struct point {
    char x;
    char y;
};

kernel foo(point *p) {
    assert(p->x == 0 &&
              p->y == 1);
}
```

Single source - single type system

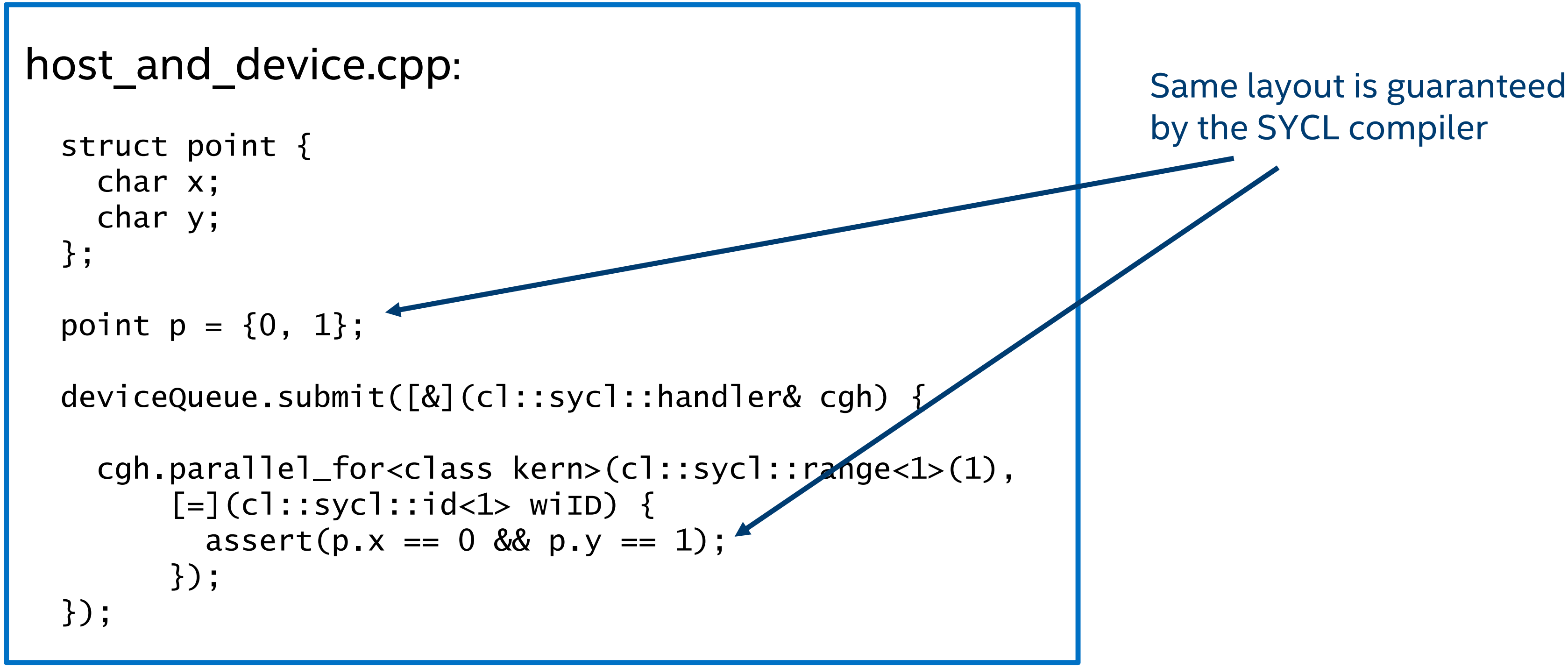
SYCL code:

host_and_device.cpp:

```
struct point {
  char x;
  char y;
};

point p = {0, 1};

deviceQueue.submit([&](cl::sycl::handler& cgh) {

  cgh.parallel_for<class kern>(cl::sycl::range<1>(1),
      [=](cl::sycl::id<1> wiID) {
        assert(p.x == 0 && p.y == 1);
      });
});
```

Single source - single type system

SYCL code:



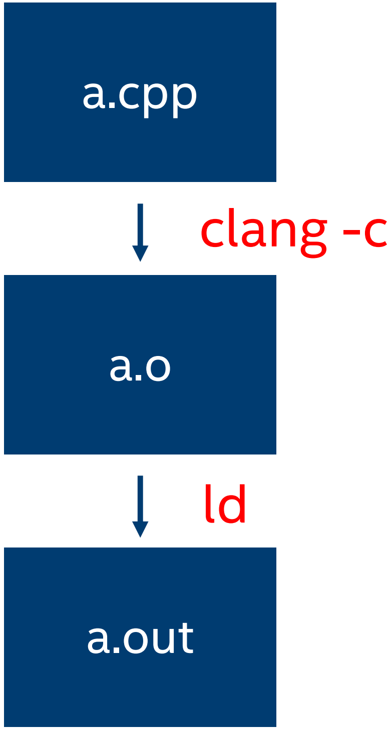
host_and_device.cpp:

```
struct point {
  char x;
  char y;
};

point p = {0, 1};

deviceQueue.submit([&](cl::sycl::handler& cgh) {

  cgh.parallel_for<class kern>(cl::sycl::range<1>(1),
      [=](cl::sycl::id<1> wiID) {
        assert(p.x == 0 && p.y == 1);
      });
});
```

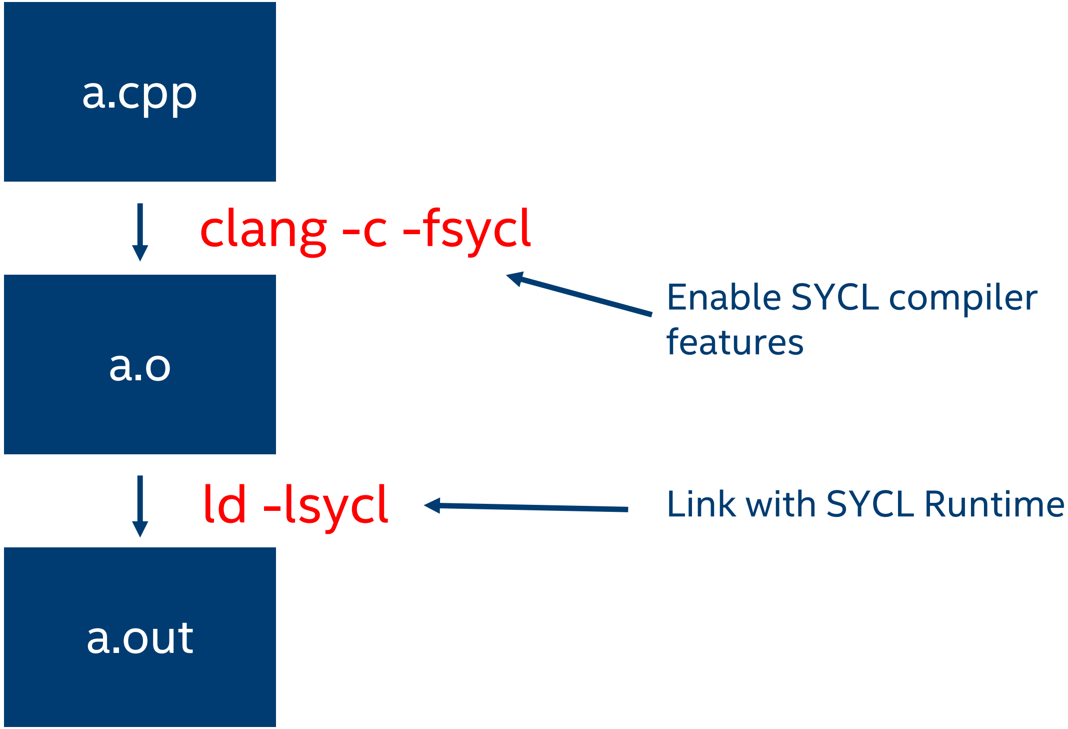
Same layout is guaranteed by the SYCL compiler

COMPILATION FLOW

Standard C++ flow



Standard C++ flow



a.cpp

clang -c -fsycl

Enable SYCL compiler features

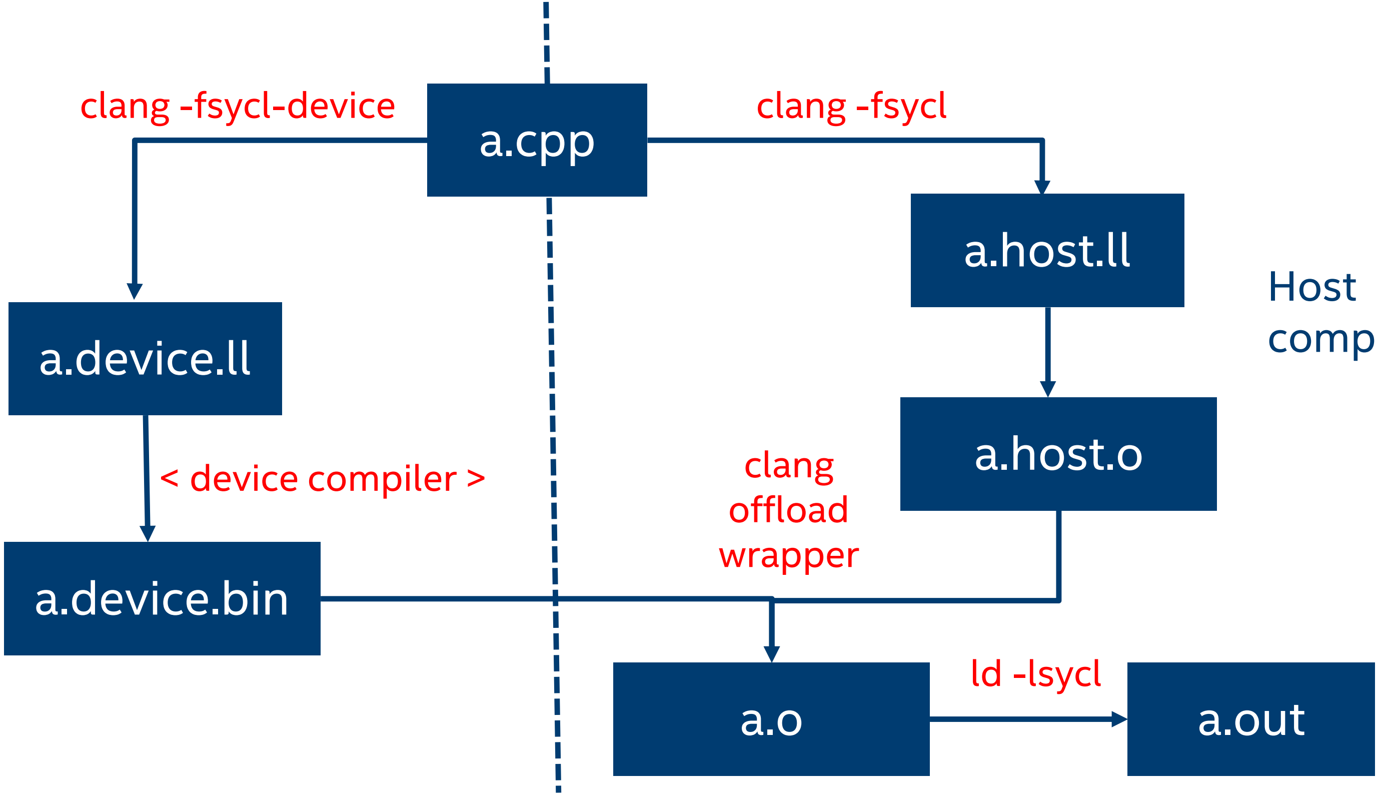
a.o

ld -lsycl

Link with SYCL Runtime

a.out

SYCL flow (under the hood) *



Device compilation

Host compilation

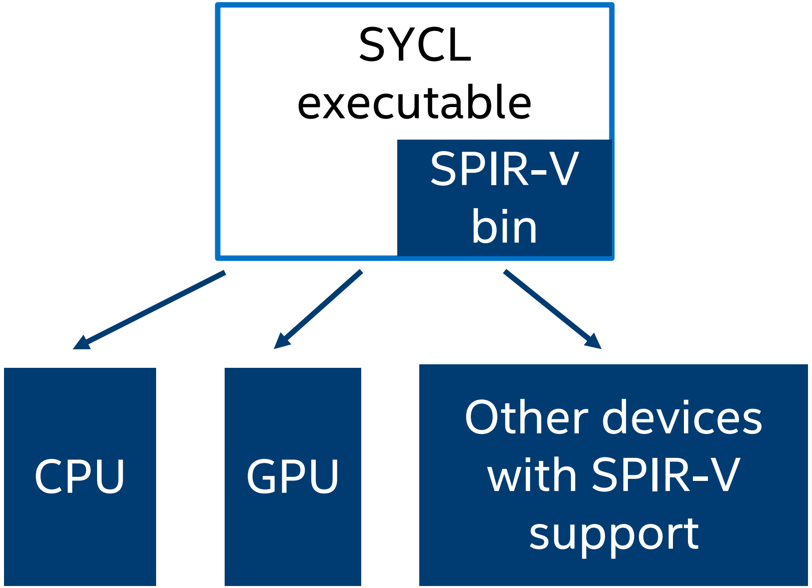
* simplified

SPIR-V support

SPIR-V is a device-agnostic IR originally created for OpenCL and Vulkan.

It allows to run a single SYCL executable On any device that supports SPIR-V.

SPIR-V Translator from LLVM IR to SPIR-V is developed on Github: https://github.com/KhronosGroup/SPIRV-LLVM-Translator



SYCL flow: integration header

```
cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(8),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
```

Step 1:
device compiler extracts the lambda function.

Class name **kernel_1** is a device kernel name.

SYCL flow: integration header

```
cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(8),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
```

Step 1:
device compiler
extracts the
lambda function.

Class name **kernel_1**
is a device kernel
name.

a.device.bin:

```
__kernel kernel_1(T* buf) {
   ...
}
```

SYCL flow: integration header

```
cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(8),
        [=](cl::sycl::id<1> wiID) {
          accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
        });
```

Step 2:
Host code must call the device function by name, and provide the required parameters.

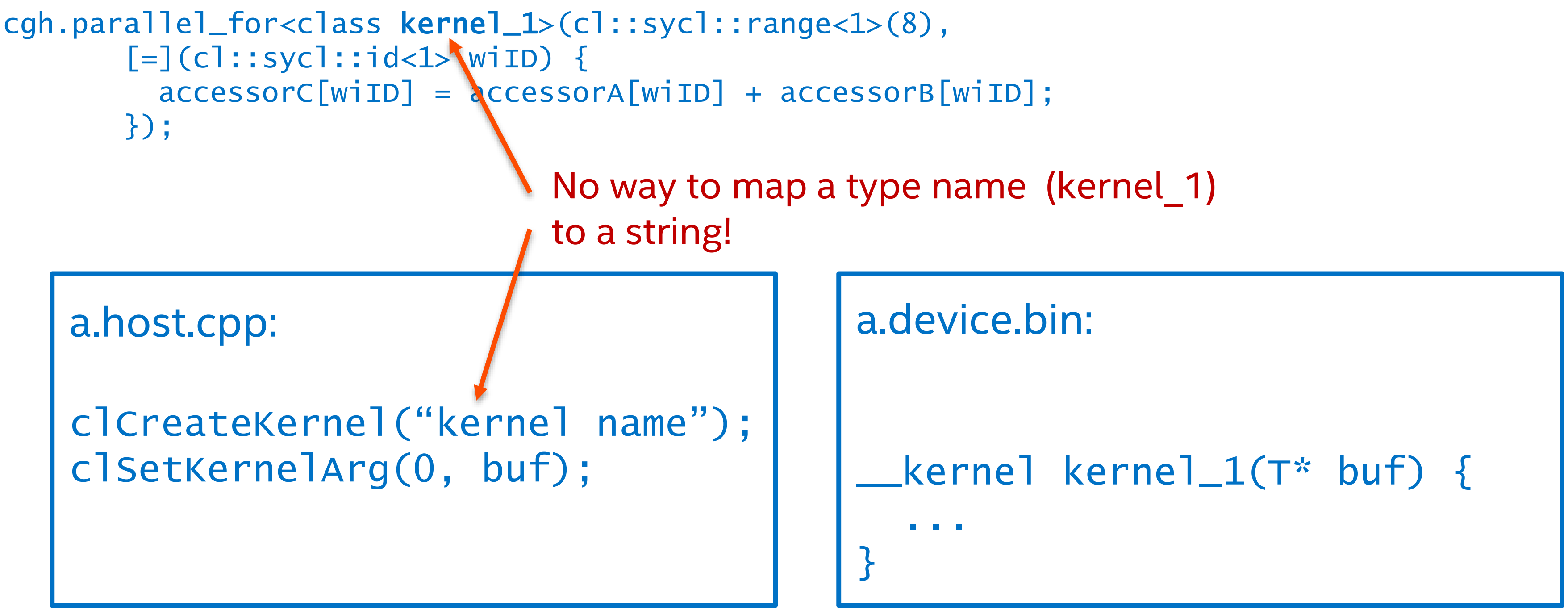
a.host.cpp:

```
clCreateKernel(“kernel name”);
clSetKernelArg(0, buf);
```

a.device.bin:

```
__kernel kernel_1(T* buf) {
   ...
}
```

SYCL flow: integration header



```
cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(8),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
```

No way to map a type name (kernel_1) to a string!

a.host.cpp:

```
clCreateKernel("kernel name");
clSetKernelArg(0, buf);
```

a.device.bin:

```
__kernel kernel_1(T* buf) {
   ...
}
```

SYCL flow: integration header

No way to determine an order of arguments captured by a lambda

```
cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(8),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
```

a.host.cpp:

```
clCreateKernel(“kernel name”);
clSetKernelArg(0, buf);
```

a.device.bin:

```
__kernel kernel_1(T* buf) {
   ...
}
```

SYCL flow: integration header

```
cgh.parallel_for<class kernel_1>(cl::sycl::range<1>(8),
       [=](cl::sycl::id<1> wiID) {
         accessorC[wiID] = accessorA[wiID] + accessorB[wiID];
       });
```

a.int.h:

```
template<>
class KernelDesc<kernel_1> {
  const char* getName();
  unsigned getArgNum();
  ArgDesc getArg(unsigned);
};
```

a.host.cpp:

```
clCreateKernel(
  KernelDesk<T>::getName());
...
```

a.device.bin:

```
__kernel kernel_1(T* buf) {
  ...
}
```

SYCL standard library

SYCL standard library implementation consists of 28 public headers, and ~30 implementation (detail) headers:

- include/CL/sycl/accessor.hpp
- include/CL/sycl/buffer.hpp
- include/CL/sycl/device.hpp
- include/CL/sycl/kernel.hpp
- etc.

SYCL upstream to LLVM.org

- Intel/llvm repository is a staging area to design concepts and prototype solutions
- Contribution to llvm.org is our primary goal
  - RFC: https://lists.llvm.org/pipermail/cfe-dev/2019-January/060811.html
  - First changes to the clang driver are already committed: https://reviews.llvm.org/D57768
  - Detailed plan for upstream: https://github.com/intel/llvm/issues/49
  - SYCL source code: https://github.com/intel/llvm/tree/sycl

Call to action

Please provide inputs to design and implementation

Welcome to contribute ideas/implementation to our sandbox or join us on the path to llvm.org intel®

Software