codeplay

Implementing the SYCL for OpenCL Shared Source C++ Programming Model using Clang/LLVM

Gordon Brown

Runtime Engineer, Codeplay

Visit us at

www.codeplay.com

45 York Place
Edinburgh
EH1 3HP
United Kingdom


codeplay

Agenda

- Overview of SYCL
- SYCL Example: Vector Add
- Shared Source Programming Model
- Implementing SYCL Using Clang/LLVM

Overview of SYCL

Motivation

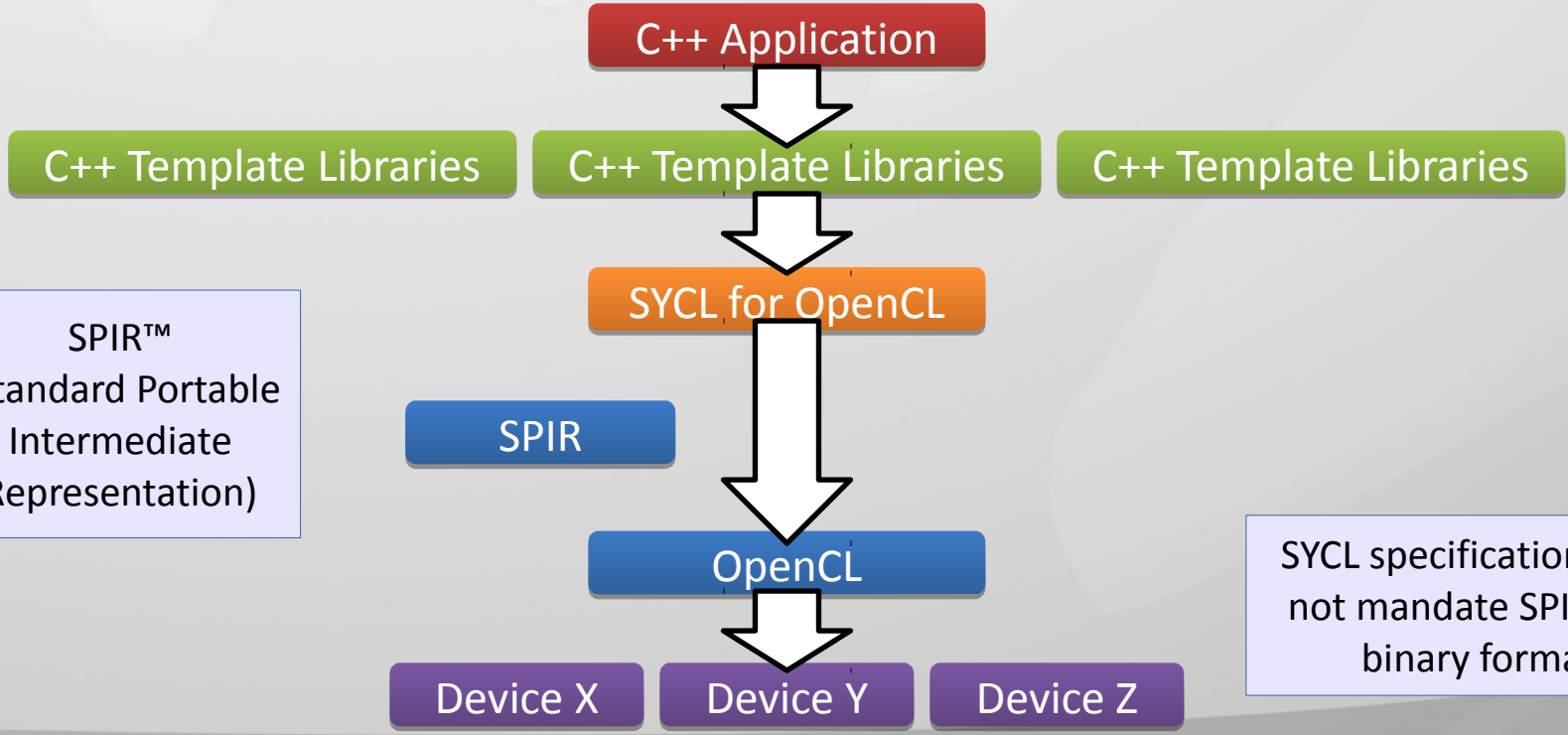
- To make GPGPU simpler and more accessible.
- To create a C++ for OpenCL™ ecosystem.
  - Combine the ease of use and flexibility of C++ and the portability and efficiency of OpenCL.
- To provide a foundation for constructing complex and reusable template algorithms:
  - parallel_reduce(), parallel_map(), parallel_sort()
- To define an open and portable standard.

SYCL for OpenCL

Cross platform, single source, C++ programming layer

Built on top of OpenCL and based on standard C++11.

The SYCL Ecosystem



SPIR™ (Standard Portable Intermediate Representation)

SYCL specification does not mandate SPIR as a binary format

SYCL Standard Roadmap

- Current State:
  - Second provisional specification being announced here at Supercomputing 2014.
- Next Steps:
  - Full specification based on feedback.
  - Conformance test suite to ensure compatibility.
  - Release implementations.

SYCL Example: Vector Add

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T>
void parallel_vadd(std::vector<T> inputA, std::vector<T> inputB, std::vector<T> output) {
  buffer<float, 1> inputABuf(inputA, inputA.size());
  buffer<float, 1> inputBBuf(inputB, inputB.size());
  buffer<float, 1> outputBuf(output, output.size());
  queue defaultQueue;
  command_group(defaultQueue, [&] () {
    auto inputAPtr = inputABuf.get_access<access::read>();
    auto inputBPtr = inputBBuf.get_access<access::read>();
    auto outputPtr = outputBuf.get_access<access::write>();
    parallel_for< vadd<T> >(range<1>(output.size()), ([=](id<1> idx)  {
      ouptPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    }));
  });
}
```

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

int main() {
  return 0;
}
```

The SYCL runtime is in sycl.hpp and within the cl::sycl namespace

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

int main() {
  int count = 1024;
  std::vector<float> inputA(count) = { /* input a */ };
  std::vector<float> inputB(count) = { /* input b */ };
  std::vector<float> output(count) = { /* output */ };
  return 0;
}
```

Construct and initialise three std::vector objects of 1024 float elements, two inputs and one output.

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

int main() {
  int count = 1024;
  std::vector<float> inputA(count) = { /* input a */ };
  std::vector<float> inputB(count) = { /* input b */ };
  std::vector<float> output(count) = { /* output */ };
  {
    buffer<float, 1> inputABuf(inputA.data(), inputA.size());
    buffer<float, 1> inputBBuf(inputB.data(), inputB.size());
    buffer<float, 1> outputBuf(output.data(), output.size());
  }
  return 0;
}
```

Construct three SYCL buffers and initialise them with the data from the std::vectors.

Data is synchronised by RAII

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

int main() {
  int count = 1024;
  std::vector<float> inputA(count) = { /* input a */ };
  std::vector<float> inputB(count) = { /* input b */ };
  std::vector<float> output(count) = { /* output */ };
  {
    buffer<float, 1> inputABuf(inputA.data(), inputA.size());
    buffer<float, 1> inputBBuf(inputB.data(), inputB.size());
    buffer<float, 1> outputBuf(output.data(), output.size());
    queue defaultQueue;
  }
  return 0;
}
```

There are many other options for device discovery and configuration.

Construct a SYCLL queue to execute work on a device.

```
...

int main() {
  int count = 1024;
  std::vector<float> inputA(count) = { /* input a */ };
  std::vector<float> inputB(count) = { /* input b*/ };
  std::vector<float> output(count) = { /* output */ };
  {
    buffer<float, 1> inputABuf(inputA.data(), inputA.size());
    buffer<float, 1> inputBBuf(inputB.data(), inputB.size());
    buffer<float, 1> outputBuf(output.data(), output.size());
    queue defaultQueue;
    command_group(defaultQueue, [&] () {
    });
  }
  return 0;
}
```

The command_group is en-queued asynchronously and is thread safe.

Construct a SYCL command_group to define the work to be en-queued on a device.

```
...
std::vector<float> inputA(count) = { /* input a */ };
std::vector<float> inputB(count) = { /* input b*/ };
std::vector<float> output(count) = { /* output */ };
{
  buffer<float, 1> inputABuf(inputA.data(), inputA.size());
  buffer<float, 1> inputBBuf(inputB.data(), inputB.size());
  buffer<float, 1> outputBuf(output.data(), output.size());
  queue defaultQueue;
  command_group(defaultQueue, [&] () {
    auto inputAPtr = inputABuf.get_access<access::read>();
    auto inputBPtr = inputBBuf.get_access<access::read>();
    auto outputPtr = outputBuf.get_access<access::write>();
  });
}
...
```

The SYCL runtime used accessors to track dependencies across command_groups.

Construct three SYCL accessors with the appropriate access modes, to give the device access to the data.

```
...
{
  buffer<float, 1> inputABuf(inputA.data(), inputA.size());
  buffer<float, 1> inputBBuf(inputB.data(), inputB.size());
  buffer<float, 1> outputBuf(output.data(), output.size());
  queue defaultQueue;
  command_group(defaultQueue, [&] () {
    auto inputAPtr = inputABuf.get_access<access::read>();
    auto inputBPtr = inputBBuf.get_access<access::read>();
    auto outputPtr = outputBuf.get_access<access::write>();
    parallel_for<class vadd>(range<1>(count), ([=](id<1> idx) {
    }));
  });
}
...
```

There are additional more complex APIs.

Call parallel_for() to execute a kernel function.

The range provided to the parallel_for() should match the size of the data buffers.

The typename 'vadd' is used to name the lambda.

```
...
{
  buffer<float, 1> inputABuf(inputA.data(), inputA.size());
  buffer<float, 1> inputBBuf(inputB.data(), inputB.size());
  buffer<float, 1> outputBuf(output.data(), output.size());
  queue defaultQueue;
  command_group(defaultQueue, [&] () {
    auto inputAPtr = inputABuf.get_access<access::read>();
    auto inputBPtr = inputBBuf.get_access<access::read>();
    auto outputPtr = outputBuf.get_access<access::write>();
    parallel_for<class vadd>(range<1>(count), ([=](id<1> idx) {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    }));
  });
}
...
```

The body of the lambda expression is what is compiled into an OpenCL kernel by the SYCL device compiler.

Use the subscript operator on the accessors to read and write the data.

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

void parallel_vadd(std::vector<float> &inputA, std::vector<float> &inputB, std::vector<float> &output) {
  buffer<float, 1> inputABuf(inputA.data(), inputA.size());
  buffer<float, 1> inputBBuf(inputB.data(), inputB.size());
  buffer<float, 1> outputBuf(output.data(), output.size());
  queue defaultQueue;
  command_group(defaultQueue, [&] () {
    auto inputAPtr = inputABuf.get_access<access::read>();
    auto inputBPtr = inputBBuf.get_access<access::read>();
    auto outputPtr = outputBuf.get_access<access::write>();
    parallel_for<class vadd>(range<1>(count), ([=](id<1> idx) {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    }));
  });
}
```

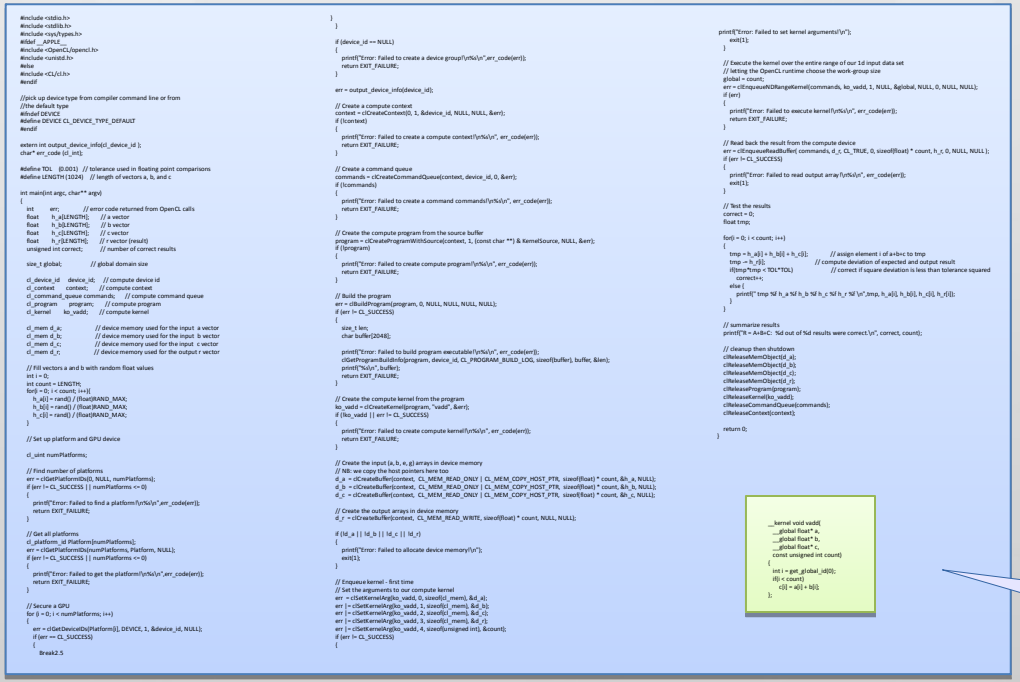
Create a function that takes the input and output vectors

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), inputA.size());
  buffer<T, 1> inputBBuf(inputB.data(), inputB.size());
  buffer<T, 1> outputBuf(output.data(), output.size());
  queue defaultQueue;
  command_group(defaultQueue, [&] () {
    auto inputAPtr = inputABuf.get_access<access::read>();
    auto inputBPtr = inputBBuf.get_access<access::read>();
    auto outputPtr = outputBuf.get_access<access::write>();
    parallel_for< vadd<T> >(range<1>(count), ([=](id<1> idx) {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    }));
  });
}
```

Template the function
By the data type

The typename 'vadd' must also be templated as the lambda expression is template dependant.

Comparison with OpenCL



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#include <unistd.h>
#else
#include <CL/cl.h>
#endif

//pick up device type from compiler command line or from
//the default type
#ifndef DEVICE
#define DEVICE CL_DEVICE_TYPE_DEFAULT
#endif

extern int output_device_info(cl_device_id );
char* err_code (cl_int);

#define TOL    (0.001)   // tolerance used in floating point comparisons
#define LENGTH (1024)    // length of vectors a, b, and c

int main(int argc, char** argv)
{
    int     err;            // error code returned from OpenCL calls
    float   h_a[LENGTH];    // a vector
    float   h_b[LENGTH];    // b vector
    float   h_c[LENGTH];    // c vector
    float   h_r[LENGTH];    // r vector (result)
    unsigned int correct;   // number of correct results

    size_t global;          // global domain size

    cl_device_id    device_id;   // compute device id
    cl_context      context;     // compute context
    cl_command_queue commands;   // compute command queue
    cl_program      program;     // compute program
    cl_kernel       ko_vadd;     // compute kernel

    cl_mem d_a;                  // device memory used for the input  a vector
    cl_mem d_b;                  // device memory used for the input  b vector
    cl_mem d_c;                  // device memory used for the input  c vector
    cl_mem d_r;                  // device memory used for the output r vector

    // Fill vectors a and b with random float values
    int i = 0;
    int count = LENGTH;
    for(i = 0; i < count; i++){
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
        h_c[i] = rand() / (float)RAND_MAX;
    }

    // Set up platform and GPU device

    cl_uint numPlatforms;

    // Find number of platforms
    err = clGetPlatformIDs(0, NULL, &numPlatforms);
    if (err != CL_SUCCESS || numPlatforms <= 0)
    {
        printf("Error: Failed to find a platform!\n%s\n",err_code(err));
        return EXIT_FAILURE;
    }

    // Get all platforms
    cl_platform_id Platform[numPlatforms];
    err = clGetPlatformIDs(numPlatforms, Platform, NULL);
    if (err != CL_SUCCESS || numPlatforms <= 0)
    {
        printf("Error: Failed to get the platform!\n%s\n",err_code(err));
        return EXIT_FAILURE;
    }

    // Secure a GPU
    for (i = 0; i < numPlatforms; i++)
    {
        err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id, NULL);
        if (err == CL_SUCCESS)
        {
            Break2.5
        }
    }

    if (device_id == NULL)
    {
        printf("Error: Failed to create a device group!\n%s\n",err_code(err));
        return EXIT_FAILURE;
    }

    err = output_device_info(device_id);

    // Create a compute context
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    if (!context)
    {
        printf("Error: Failed to create a compute context!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Create a command queue
    commands = clCreateCommandQueue(context, device_id, 0, &err);
    if (!commands)
    {
        printf("Error: Failed to create a command commands!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL, &err);
    if (!program)
    {
        printf("Error: Failed to create compute program!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Build the program
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];

        printf("Error: Failed to build program executable!\n%s\n", err_code(err));
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
        return EXIT_FAILURE;
    }

    // Create the compute kernel from the program
    ko_vadd = clCreateKernel(program, "vadd", &err);
    if (!ko_vadd || err != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Create the input (a, b, e, g) arrays in device memory
    // NB: we copy the host pointers here too
    d_a = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  sizeof(float) * count, &h_a, NULL);
    d_b = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  sizeof(float) * count, &h_b, NULL);
    d_c = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  sizeof(float) * count, &h_c, NULL);

    // Create the output arrays in device memory
    d_r = clCreateBuffer(context,  CL_MEM_READ_WRITE, sizeof(float) * count, NULL, NULL);

    if (!d_a || !d_b || !d_c || !d_r)
    {
        printf("Error: Failed to allocate device memory!\n");
        exit(1);
    }

    // Enqueue kernel - first time
    // Set the arguments to our compute kernel
    err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(ko_vadd, 3, sizeof(cl_mem), &d_r);
    err |= clSetKernelArg(ko_vadd, 4, sizeof(unsigned int), &count);
    if (err != CL_SUCCESS)
    {
```

```
    printf("Error: Failed to set kernel arguments! \n");
    exit(1);
  }

  // Execute the kernel over the entire range of our 1d input data set
  // letting the OpenCL runtime choose the work-group size
  global = count;
  err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
  if (err)
  {
    printf("Error: Failed to execute kernel! \n%s\n", err_code(err));
    return EXIT_FAILURE;
  }

  // Read back the result from the compute device
  err = clEnqueueReadBuffer( commands, d_r, CL_TRUE, 0, sizeof(float) * count, h_r, 0, NULL, NULL );
  if (err != CL_SUCCESS)
  {
    printf("Error: Failed to read output array! \n%s\n", err_code(err));
    exit(1);
  }

  // Test the results
  correct = 0;
  float tmp;

  for(i = 0; i < count; i++)
  {
    tmp = h_a[i] + h_b[i] + h_c[i];          // assign element i of a+b+c to tmp
    tmp -= h_r[i];                           // compute deviation of expected and output result
    if(tmp*tmp < TOL*TOL)                    // correct if square deviation is less than tolerance squared
      correct++;
    else {
      printf(" tmp %f h_a %f h_b %f h_c %f h_r %f \n",tmp, h_a[i], h_b[i], h_c[i], h_r[i]);
    }
  }

  // summarize results
  printf("%d + A+B+C:  %d out of %d results were correct.\n", correct, count);

  // cleanup then shutdown
  clReleaseMemObject(d_a);
  clReleaseMemObject(d_b);
  clReleaseMemObject(d_c);
  clReleaseMemObject(d_r);
  clReleaseProgram(program);
  clReleaseKernel(ko_vadd);
  clReleaseCommandQueue(commands);
  clReleaseContext(context);

  return 0;
}
```

```
__kernel void vadd(
    __global float* a,
    __global float* b,
    __global float* c,
    const unsigned int count)
{
    int i = get_global_id(0);
    if(i < count)
      c[i] = a[i] + b[i];
};
```

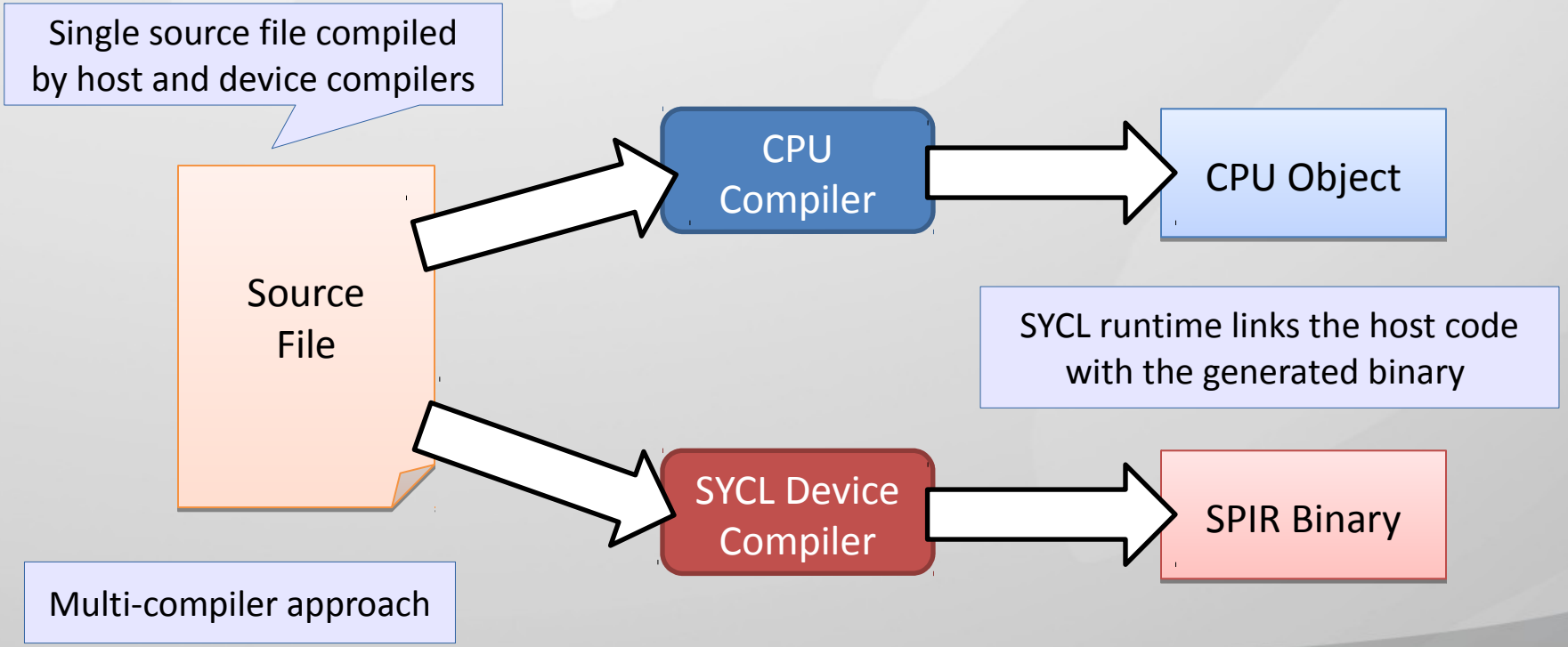
```
#include <CL/sycl.hpp>
using namespace cl::sycl;

int main()
{
  int count = 1024;

  std::vector<float> vecA(count) = { /* input */ };
  std::vector<float> vecB(count) = { /* input */ };
  std::vector<float> vecC(count) = { /* output */ };

  {
    buffer<float, 1> bufA(vecA.data(), vecA.size());
    buffer<float, 1> bufB(vecB.data(), vecB.size());
    buffer<float, 1> bufC(vecC.data(), vecC.size());

    queue myQueue;

    command_group(myQueue, [&] ()
    {
      auto ptrA = bufA.get_access<access::read>();
      auto ptrB = bufB.get_access<access::read>();
      auto ptrC = bufC.get_access<acces::write>();

      parallel_for(range<1>(count), kernel_functor<class vector_add>([=](id<1> idx)
      {
        ptrC[idx] = ptrA[idx] + ptrB[idx];
      }));
    });
  }

  return 0;
}
```

SYCL for OpenCL

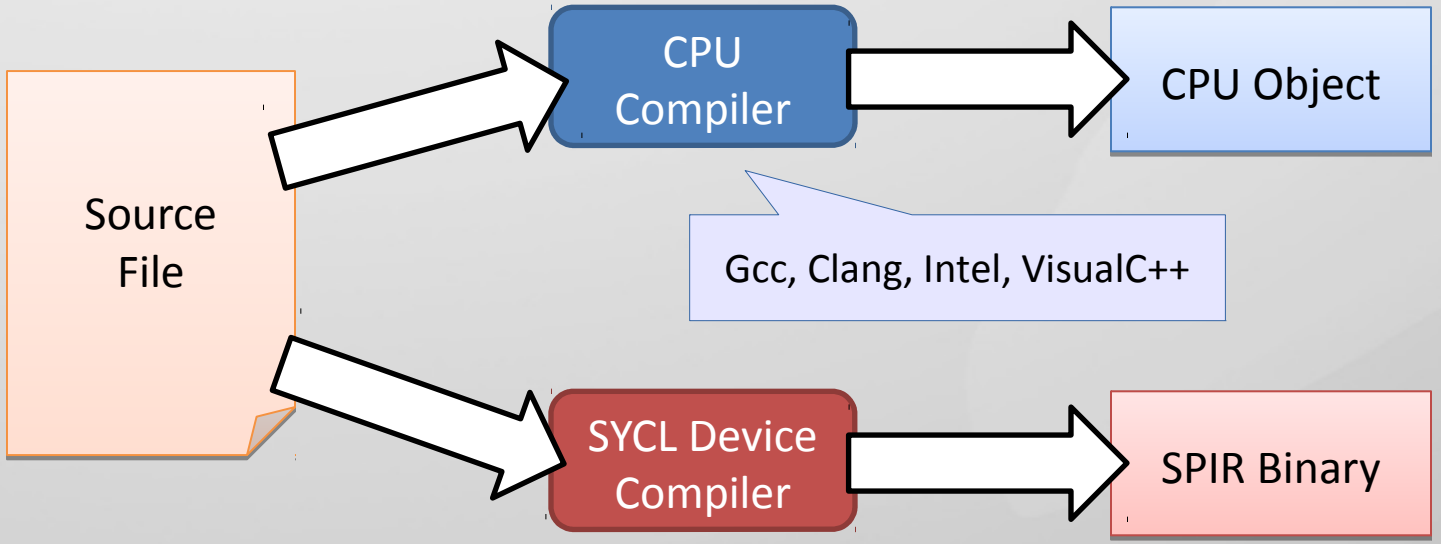
Traditional OpenCL

Shared Source Programming Model

Shared Source

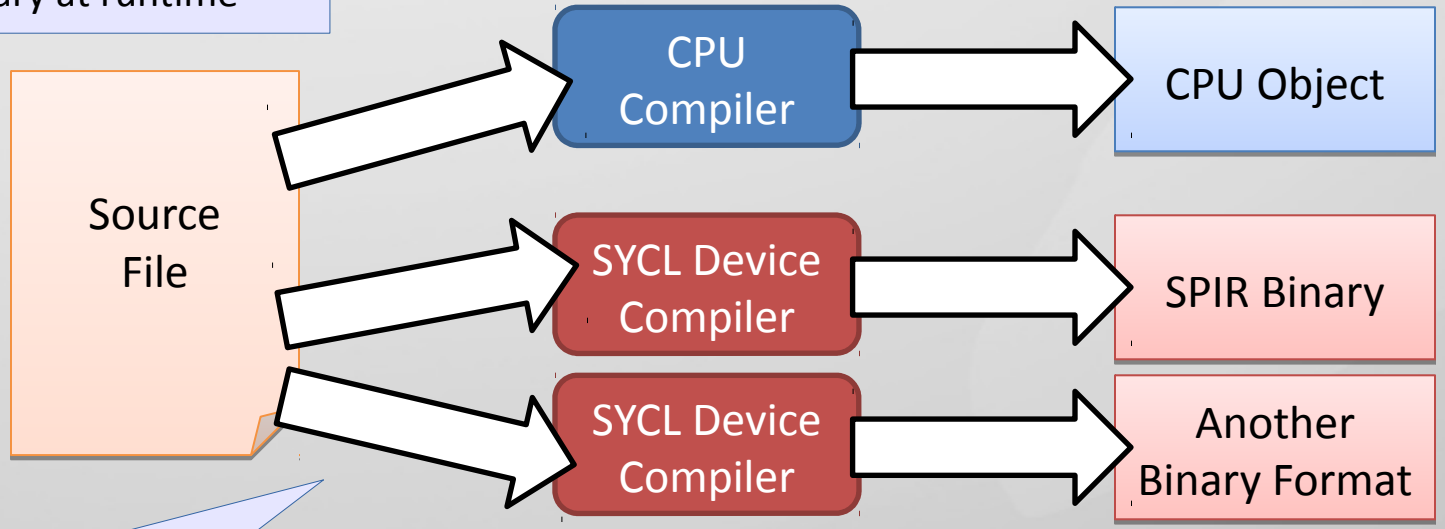


Choose Your Own Host Compiler



Target Multiple Device Compilers

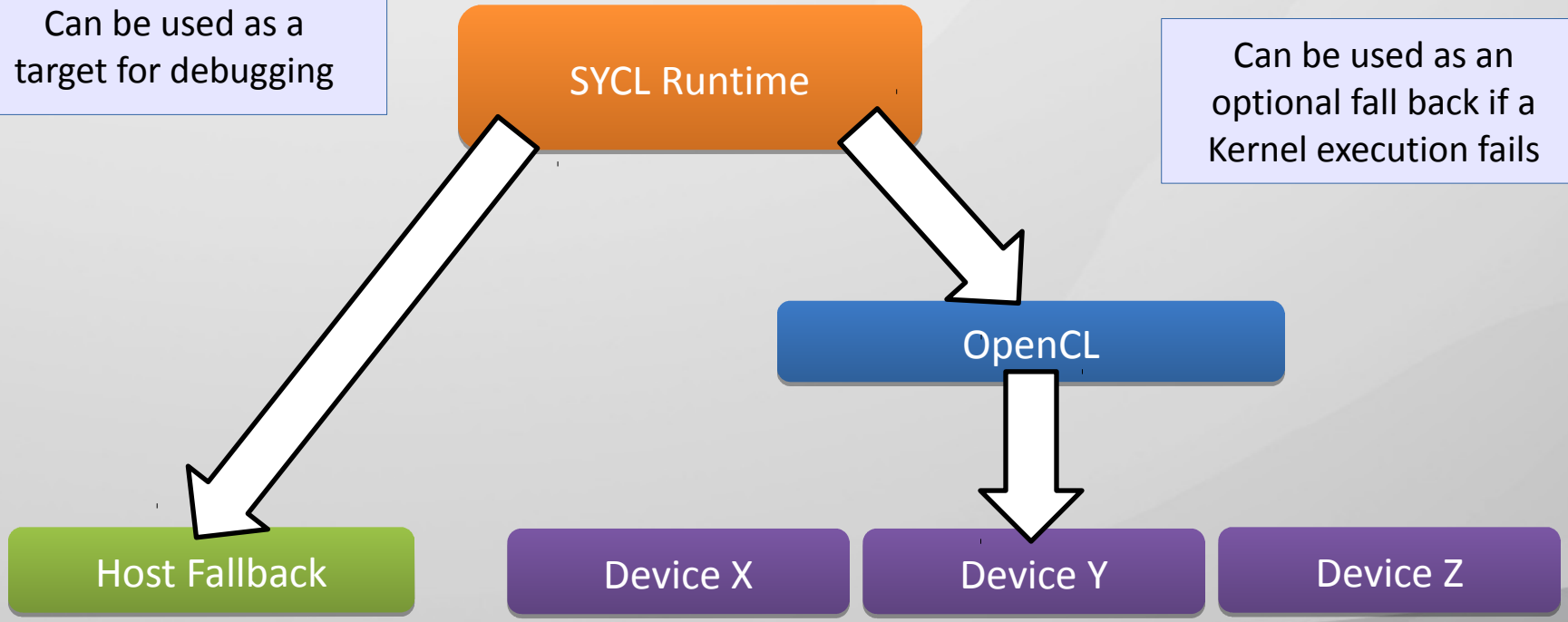
SYCL runtime will choose a binary at runtime



Multiple device compilers

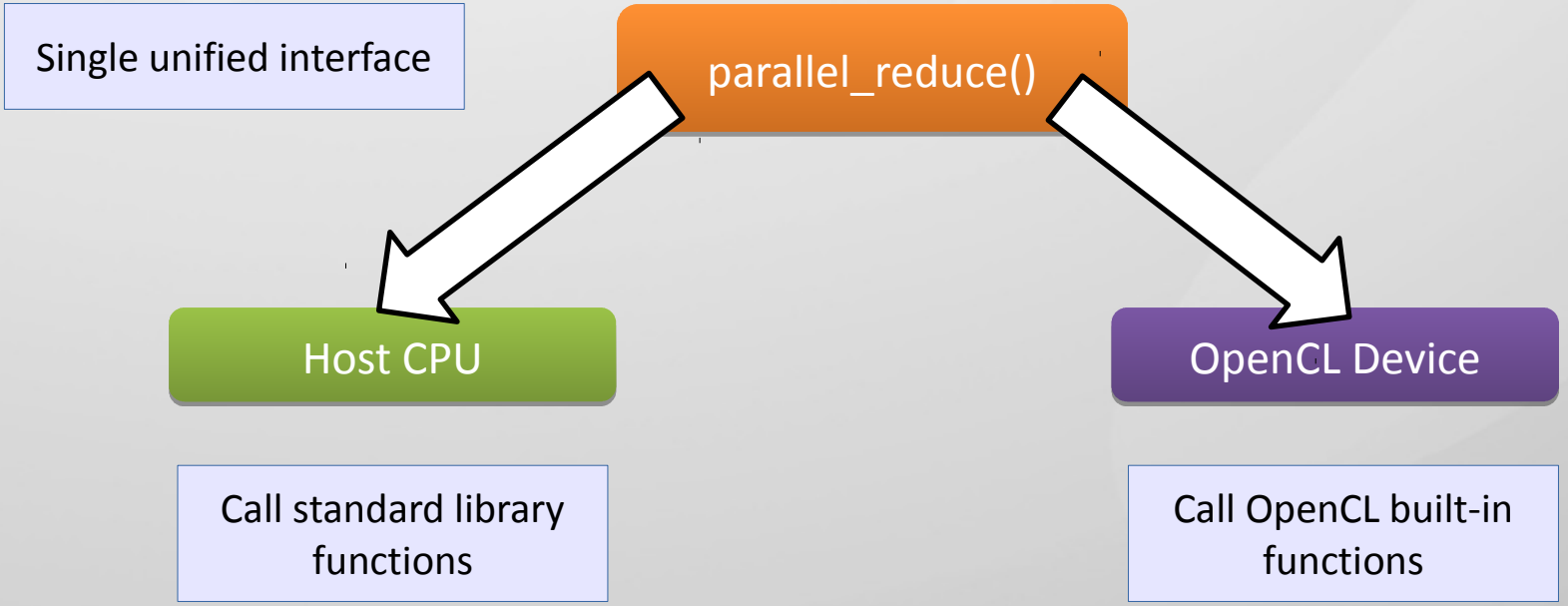
Host Device

Can be used as a target for debugging



Can be used as an optional fall back if a Kernel execution fails

Use Common Libraries



codeplay

Implementing SYCL Using Clang/LLVM

Why Clang/LLVM?

- Clang & LLVM is perfect for implementing innovative compiler technologies:
  - Large amount of contributors and a great developer community.
  - Very feature rich (standard and non-standard).
  - Full of re-usable modules and components.

Topics

- Separating Host & Device Code
- Constructing an OpenCL Kernel Function
- Duplicating Device Functions
- Diagnosing Invalid Device Code
- Supporting OpenCL Types
- Generating a SPIR Binary
- Integrating with SYCL Runtime

Separating Host & Device Code

SYCL Requirement

- SYCL has a unified interface that users develop in.

- Programming model needs to allow the host compiler and SYCL device compiler to extract different code:
  - Types that can be passed between host and device.
  - APIs that have different behavior between host and device.

Implicit Pre-process Macro

- Our SYCL device compiler adds the pre-process macro.
- This is used to separate parts of the SYCL runtime into host and device code.

```
__SYCL_DEVICE_ONLY__
```

Example: accessor

```
accessor<float, 1, access::read> inputPtrA(inputBufferA);
```

Host Compiler

Device Compiler

Construct OpenCL buffer and enqueue to device

Represent OpenCL kernel argument on device

Example: parallel_for()

```
parallel_for<class vadd>(range<1>(count), ([=](id<1> itemID) {
  outputPtr[itemID] = inputPtrA[itemID] + inputPtrB[itemID];
}));
```

Host Compiler

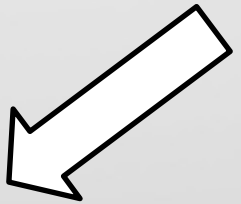
Device Compiler

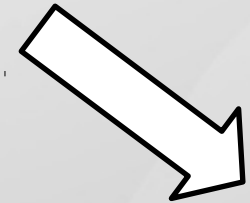
Set arguments and enqueue kernel function

Generate the kernel function

Example: fmin()

```
float4 result = fmin(valA, valB);
```





Host Compiler

Device Compiler

Host-side fmin() function

OpenCL built-in fmin() function

Constructing an OpenCL Kernel Function

SYCL Requirement

- SYCL represents kernel functions using lambda expressions.

- Programming model needs to identify and construct an OpenCL kernel:
  - Name the lambda expression.
  - Convert the body to an OpenCL kernel function.
  - Convert captured variables to OpenCL kernel arguments.

Identify the SYCL Kernel Function

- Our SYCL device compiler uses a Clang attribute to identify the SYCL kernel function.
- This is also used to assign a name to the lambda expression.

```
__attribute__((sycl_kernel(name)))
```

Convert the Lambda Expression

- Our SYCL device compiler manipulates the Clang AST to create the OpenCL kernel function.
  - The lambda expression is converted to an OpenCL compatible function.
  - The captured variables are converted to OpenCL kernel arguments.

Example: parallel_for()

```
parallel_for<class vadd>(range<1>(count), ([=](id<1> itemID) {
  outputPtr[itemID] = inputPtrA[itemID] + inputPtrB[itemID];
}));
```

```
__kernel void sycl_vadd (__global float *inputPtr A,
                         __global float *inputPtrB,
                         __global float *outputPtr,
                         id itemID) {
  outputPtr[itemID] = inputPtrA[itemID] + inputPtrB[itemID];
}
```

Duplicating Device Functions

SYCL Requirement

- SYCL requires all functions that are called by the SYCL kernel function to also be available.

- Programming model needs to transform all called functions into device compatible functions:
  - Traverse call graph.
  - Clone function definitions.
  - Deduce OpenCL address spaces for parameters.

Transforming Device Functions

- Out SYCL device compiler uses the Clang TreeTransform framework:
  - Traverses the Clang AST.
  - Identifies the OpenCL kernel function's call graph.
  - Transforms called functions into device functions.
  - Deduces address spaces for device function based on parameters.

Example: reduction

```
template <typename N, typename L>
void parallel_for(range<1>, L);
```

```
template <typename T>
T reduce_array(T *data);
```

```
template <typename T>
T reduce(T a, T b);
```

```
__kernel void sycl_kernel0(__global float *ptr);
```

```
float reduce_array(__global float *data);
```

```
float reduce(float a, float b);
```

Diagnosing Invalid Device Code

SYCL Requirement

- There are restrictions on the C++ features that SYCL can support based on OpenCL 1.x hardware support.

- Programming model needs to diagnose device code when invalid C++ features are being used:
  - Catch device code that contains invalid features.
  - Generate useful error messages.

Unsupported C++ Features

- Recursion.
- Exception Handling.
- RTTI.
- Dynamic Allocation.
- Dynamic Polymorphism.
- Static Variables.
- Function Pointers.
- Virtual Functions.

These apply only to SYCL kernel functions

AST Checkers

- Our SYCL device compiler performs device code diagnostics using AST checkers.
- These are constructed from Clang's AST matcher framework.

Example: exception handling checker

```
struct ExceptionChecker : internal::SYCLFunctionBodyChecker<Stmt> {
  MatcherType<Stmt> getMatcher() const override {
    return stmt(anyOf(tryStmt(), throwExpr()));
  }
  void checkMatch(const Stmt *S, const BoundNodes &) override {
    diagInvalidConstruct(S, "Using exception handling");
  }
};
```

Supporting OpenCL Types

SYCL Requirement

- SYCL requires its types to map to underlying OpenCL types transparently and with little or no overhead.

- Programming model needs support all OpenCL types underneath SYCL types:
  - Address spaces.
  - Vectors & swizzles.
  - OpenCL built-in types and qualifiers.

OpenCL Support

- Our SYCL device compiler supports OpenCL types by enabling Clangs built-in support:
  - __attribute__((address_space(n)))
  - __attribute__((ext_vector_type(n)))
  - OCLImage2d, OCLSampler, …
  - OpenCLImageAccess

Example: vectors

```
float4 ( a.k.a. vec<float, 4> )
```

```
float __attribute__((ext_vector_type(4)))
```

Example: swizzles

```
float4 lhs, rhs;
lhs.wzyx() = rhs.xzyw();
```

```
float __attribute__((ext_vector_type(4))) lhs, rhs;
lhs.wzyx = rhs.xywx;
```

Example: images

```
accessor<float4, 2, access::read, access::image> ptr(img);
```

```
__read_only image2d_t ptr;
```

Example: address spaces

```
accessor<float, 1, access::read, access::global_buffer> ptr(buf);
```

```
__global float *ptr;
```

Generating a SPIR Binary

SYCL Requirement

- SYCL requires the OpenCL kernel function and other functions to be compiled into a binary that can be built from the OpenCL runtime.

- Programming model needs to generate a SPIR output:
  - SPIR binary.
  - SPIR assembly

Output SPIR Bitcode

- Our SYCL device compiler outputs a SPIR binary using Clang's support for dumping out SPIR IR.
- It also makes use of Clang's support for generating SPIR meta data.

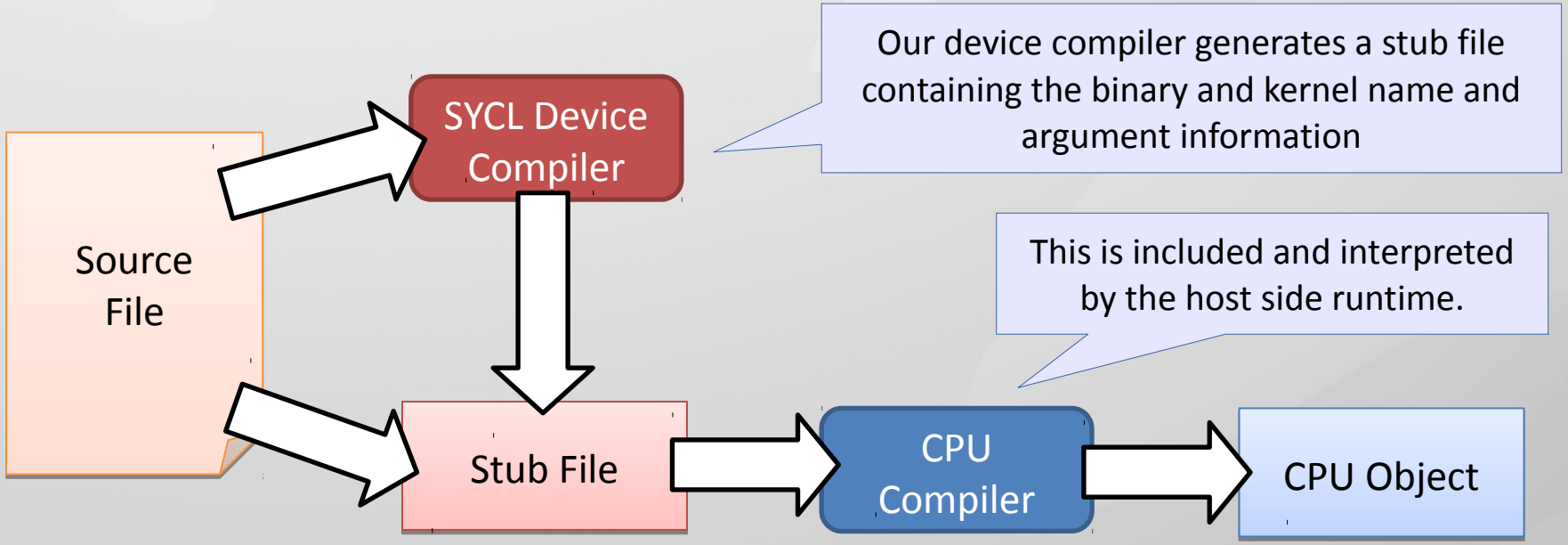
Integrating with SYCL Runtime

SYCL Requirement

- SYCL requires the runtime to build and execute the SPIR binary.

- Programming model needs to incorporate a way for linking device code into the runtime:
  - OpenCL kernel function binary.
  - OpenCL kernel function name.
  - OpenCL kernel argument information.

Shared Source Integration



How to Get Involved

- OpenCL: Version 2.0 and Beyond
  - Tuesday 5:30PM – 7:00PM (Rooms 275 – 277)
- SYCL demos:
  - AMD booth in exhibition
- Come and speak to us here:
  - Myself and Codeplay CEO Andrew Richards
- Read the specification:
  - https://www.khronos.org/opencl/sycl
- Leave feedback on the Khronos forum:
  - https://www.khronos.org/opencl/sycl
- Try out the open source implementation:
  - https://www.github.com/amd/triSYCL