# Intro to SYCL/DPC++ for GPUs

Jeff Hammond

# DOE Exascale Systems (2021+)

Neither of these systems is has a many-core CPU or an NVIDIA GPU...

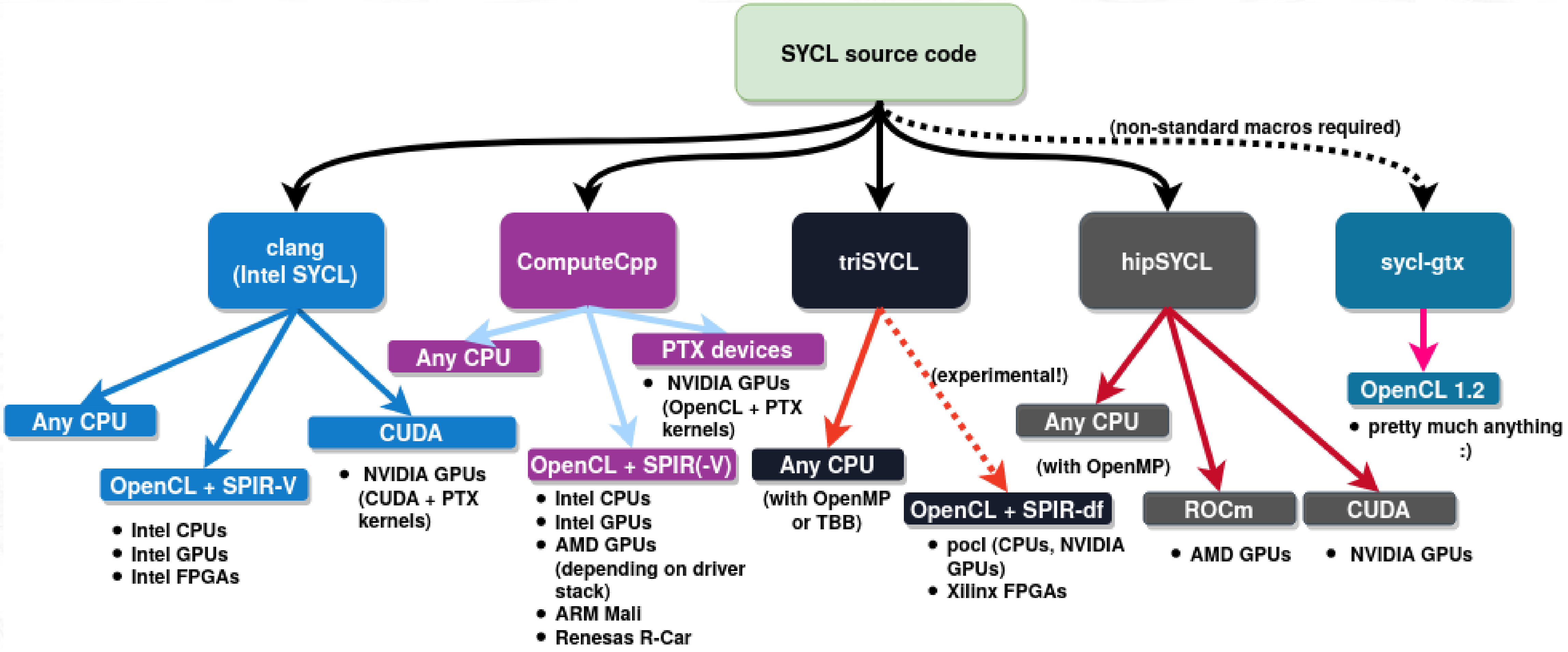


What programming model(s) take a developer from 2012 to 2022?

# Overview of the SYCL Ecosystem for GPUs

- Intel Data Parallel C++ https://software.intel.com/en-us/oneapi/base-kit
  - oneAPI product compiler based on Clang/LLVM (open-source).
  - Supports a number of GPU extensions, including USM (pointers).
  - Supports **Intel GPU**, CPU, FPGA (by Intel) and **NVIDIA** (by CodePlay)
- CodePlay ComputeCpp https://developer.codeplay.com/home/
  - Product compiler (commercial support and free community edition).
  - Supports a number of GPU extensions, including USM (pointers).
  - Supports OpenCL/SPIR-V devices (e.g. **Intel GPU**) and **NVIDIA** (via PTX).
- University of Heidelberg's hipSYCL https://github.com/illuhad/hipSYCL
  - Based on Clang/LLVM, i.e. CUDA Clang (open-source).
  - Supports CPU (OpenMP), **NVIDIA** (CUDA) and **AMD** GPU (HIP/ROCm).

# SYCL Ecosystem as of March 2020



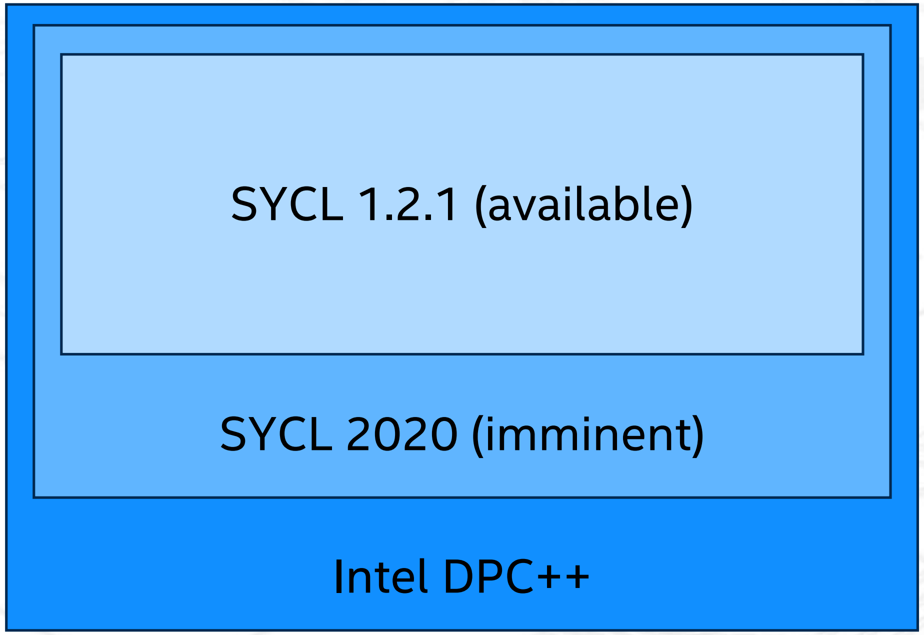

https://github.com/illuhad/hipSYCL/blob/master/doc/img/sycl-targets.png

# What is Data Parallel C++?

Intel DPC++ will be based on SYCL 2020, which is the next version of the standard after SYCL 1.2.1.

SYCL 2020 is expected to include Intel's USM proposal, which is aligned with existing GPU usage models.

We expect standard-compliant SYCL to be sufficient to realize full performance on Intel GPUs, but will add extensions to meet user needs.




Intel's extensions – both the documentation and the implementation source code - are currently available on GitHub.

# Why SYCL?

OpenCL has a well-defined, portable execution model, but is considered too verbose by application programmers and lacks good C++ support.

SYCL is based on purely modern C++, which allows it to support heterogeneous accelerators within a single-source model.

SYCL parallelism is similar to TBB and the C++ STL while giving users explicit control over hardware resources when they want it.

SYCL is the first standard programming model designed for heterogeneous programming with modern C++

# OpenCL Example

**nstream.cl**

```
__kernel void nstream(
    int length,
    double s,
    __global double * A,
    __global double * B,
    __global double * C)
{
    int i = get_global_id(0);
    A[i] += B[i] + s * C[i];
}
```

**nstream.cpp**

```
cl::Context gpu(CL_DEVICE_TYPE_GPU, &err);
cl::CommandQueue queue(gpu);

cl::Program program(gpu,
    prk::opencl::loadProgram("nstream.cl"), true);

auto kernel = cl::make_kernel<int, double, cl::Buffer,
    cl::Buffer, cl::Buffer>(program, "nstream", &err);

auto d_a = cl::Buffer(gpu, begin(h_a), end(h_a));
auto d_b = cl::Buffer(gpu, begin(h_b), end(h_b));
auto d_c = cl::Buffer(gpu, begin(h_c), end(h_c));

kernel(cl::EnqueueArgs(queue, cl::NDRange(length)),
    length, scalar, d_a, d_b, d_c);
queue.finish();
```

# SYCL 1.2.1 Example

```
sycl::gpu_selector device_selector;
sycl::queue q(device_selector);
```

Retains OpenCL's ability to easily target different devices in the same thread.

```
sycl::buffer<double> d_A { h_A.data(), h_A.size() };
sycl::buffer<double> d_B { h_B.data(), h_B.size() };
sycl::buffer<double> d_C { h_C.data(), h_C.size() };

q.submit([&](sycl::handler& h)
{
    auto A = d_A.get_access<sycl::access::mode::read_write>(h);
    auto B = d_B.get_access<sycl::access::mode::read>(h);
    auto C = d_C.get_access<sycl::access::mode::read>(h);

    h.parallel_for<class nstream>(sycl::range<1>{n}, [=] (sycl::id<1> it) {
        int i = it[0];
        A[i] += B[i] + s * C[i];
    });
});
q.wait();
```

Accessors create DAG to trigger data movement and represent execution dependencies.

Parallel loops are explicit like C++ vs. implicit in OpenCL.
Kernel code does not need to live in a separate part of the program.

# Almost SYCL 2020 Example

```
sycl::queue q(gpu_selector{});

auto * A = sycl::malloc_shared<double>(n, q);
auto * B = sycl::malloc_shared<double>(n, q);
auto * C = sycl::malloc_shared<double>(n, q);

q.submit([&](sycl::handler& h)
{
      h.parallel_for(sycl::range<1>{n}, [=] (sycl::id<1> it) {
          int i = it[0];
          A[i] += B[i] + s * C[i];
      });
});
q.wait();

sycl::free(A, q);
sycl::free(B, q);
sycl::free(C, q);
```

Pointers: everyone's favorite footgun!

Lambda names are optional, because Clang knows how it name-mangles.
*-fsycl-unnamed-lambda*

# Data Parallel C++ Example

```
sycl::queue q(gpu_selector{});

auto * A = sycl::malloc_shared<double>(n, q);
auto * B = sycl::malloc_shared<double>(n, q);
auto * C = sycl::malloc_shared<double>(n, q);

q.parallel_for(sycl::range<1>{n}, [=] (sycl::id<1> it) {
     int i = it[0];
     A[i] += B[i] + s * C[i];
});
q.wait();

sycl::free(A, q);
sycl::free(B, q);
sycl::free(C, q);
```

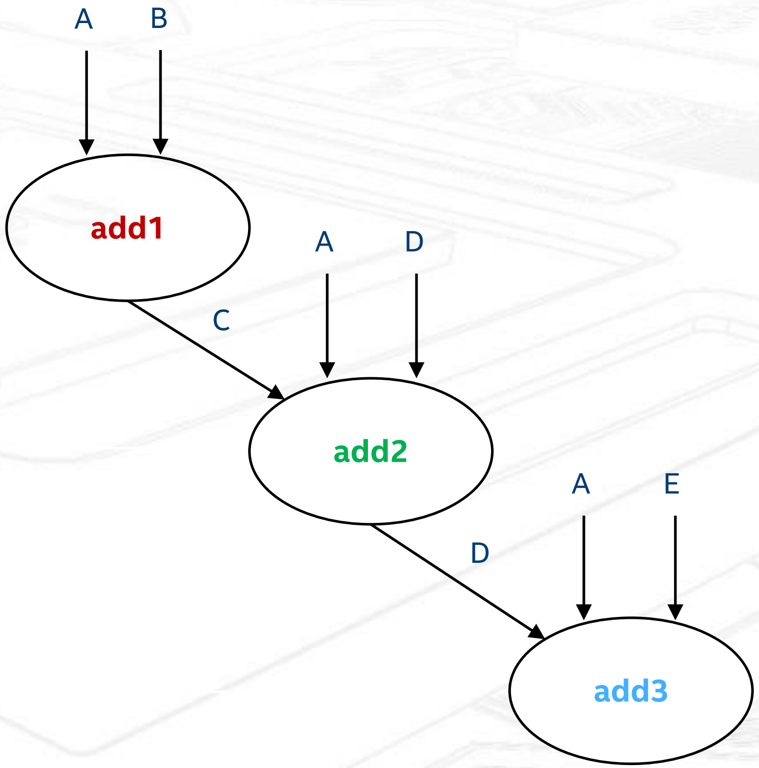
Eliminate unnecessary syntax for expressing kernels.

# oneMKL Example

```
typedef cl::sycl::buffer<double, 1> array;
typedef std::int64_t dim;

void many_dgemm(sycl::queue &q, dim n, dim nb, array &A[], array &B[], array &C[])
{
    const double alpha = 1.0;
    const double beta  = 1.0;
    for (int b=0; b<nb; ++b) {
        array pA = A[b];
        array pB = B[b];
        array pC = C[b];
        mkl::blas::gemm(q, mkl::transpose::nontrans, // opA
                           mkl::transpose::nontrans, // opB
                           n, n, n,                  // m, n, k
                           alpha, pA, n,             // alpha, A, lda
                           pB, n,                    // B, ldb
                           beta, pC, n);             // beta, C, ldc
    }
    q.wait();
}
```

# Graph of Asynchronous Executions

```
 myQueue.submit([&](handler& cgh) {
      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::discardwrite>(cgh);
      cgh.parallel_for<class add1>( range<2>{N, M},
          [=](id<2> index) { c[index] = a[index] + b[index]; });
});

myQueue.submit([&](handler& cgh) {
      auto a = A.get_access<access::mode::read>(cgh);
      auto c = C.get_access<access::mode::read>(cgh);
      auto d = D.get_access<access::mode::write>(cgh);
      cgh.parallel_for<class add2>( range<2>{P, Q},
          [=](id<2> index) { d[index] = a[index] + c[index]; });
});

myQueue.submit([&](handler& cgh) {
      auto a = A.get_access<access::mode::read>(cgh);
      auto d = D.get_access<access::mode::read>(cgh);
      auto e = E.get_access<access::mode::write>(cgh);
      cgh.parallel_for<class add3>( range<2>{S, T},
          [=](id<2> index) { e[index] = a[index] + d[index]; });
});
```

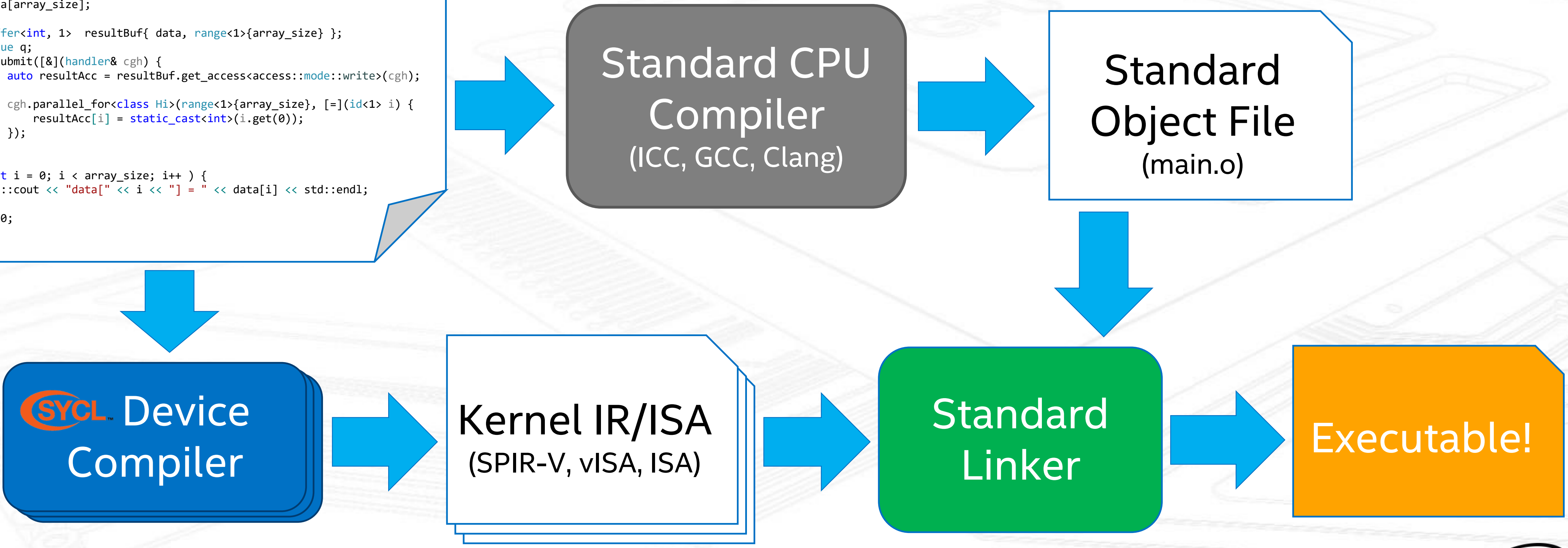



Queues are out-of-order by default – data dependencies order kernel executions.

SYCL Next adds in-order queues for convenience.

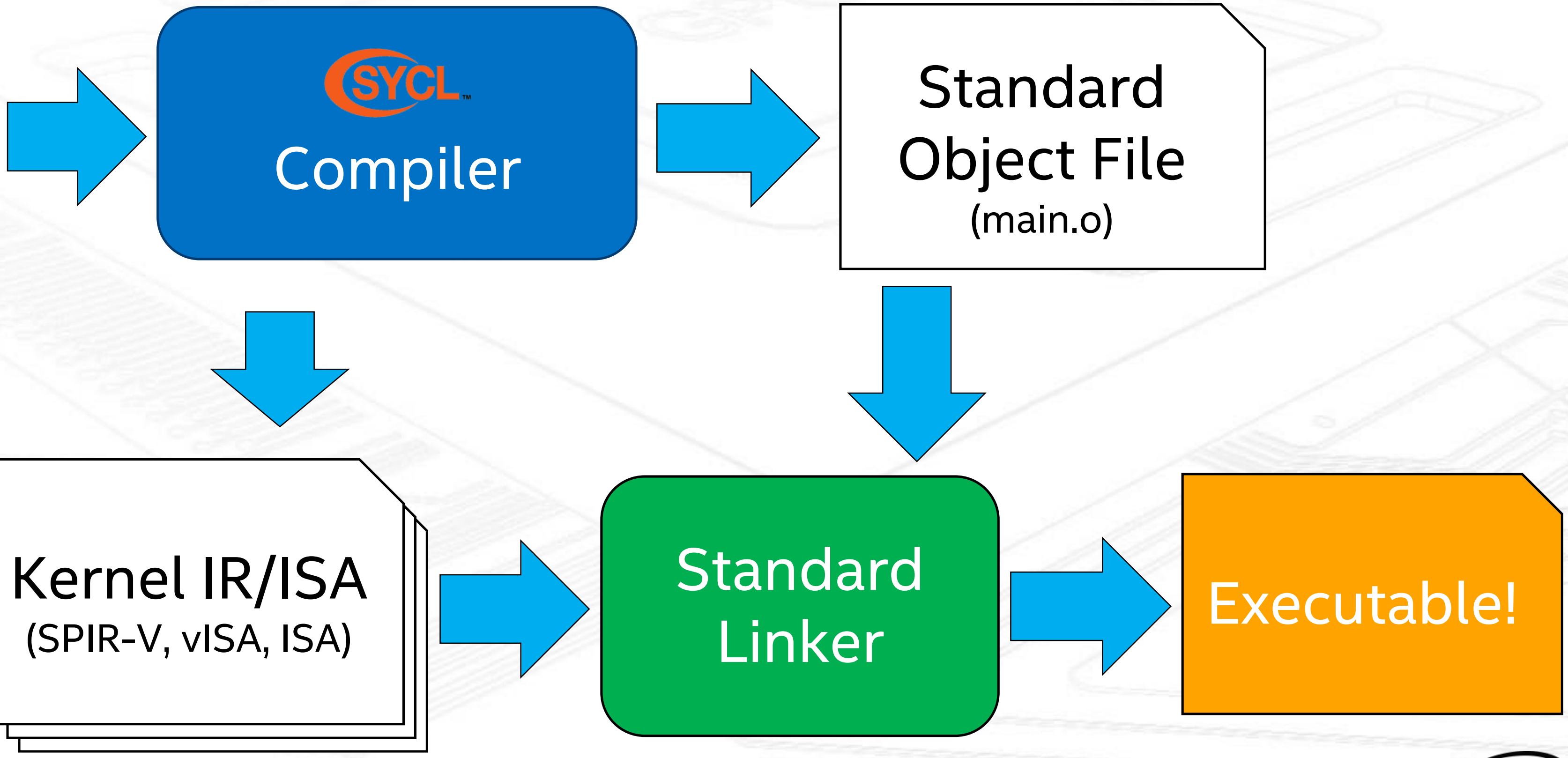
# SYCL Compilation Flow

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;
class Hi;

int main() {
    const size_t array_size = 16;
    int data[array_size];
    {
        buffer<int, 1>  resultBuf{ data, range<1>{array_size} };
        queue q;
        q.submit([&](handler& cgh) {
            auto resultAcc = resultBuf.get_access<access::mode::write>(cgh);

            cgh.parallel_for<class Hi>(range<1>{array_size}, [=](id<1> i) {
                resultAcc[i] = static_cast<int>(i.get(0));
            });
        });
    }
    for( int i = 0; i < array_size; i++ ) {
        std::cout << "data[" << i << "] = " << data[i] << std::endl;
    }
    return 0;
}
```

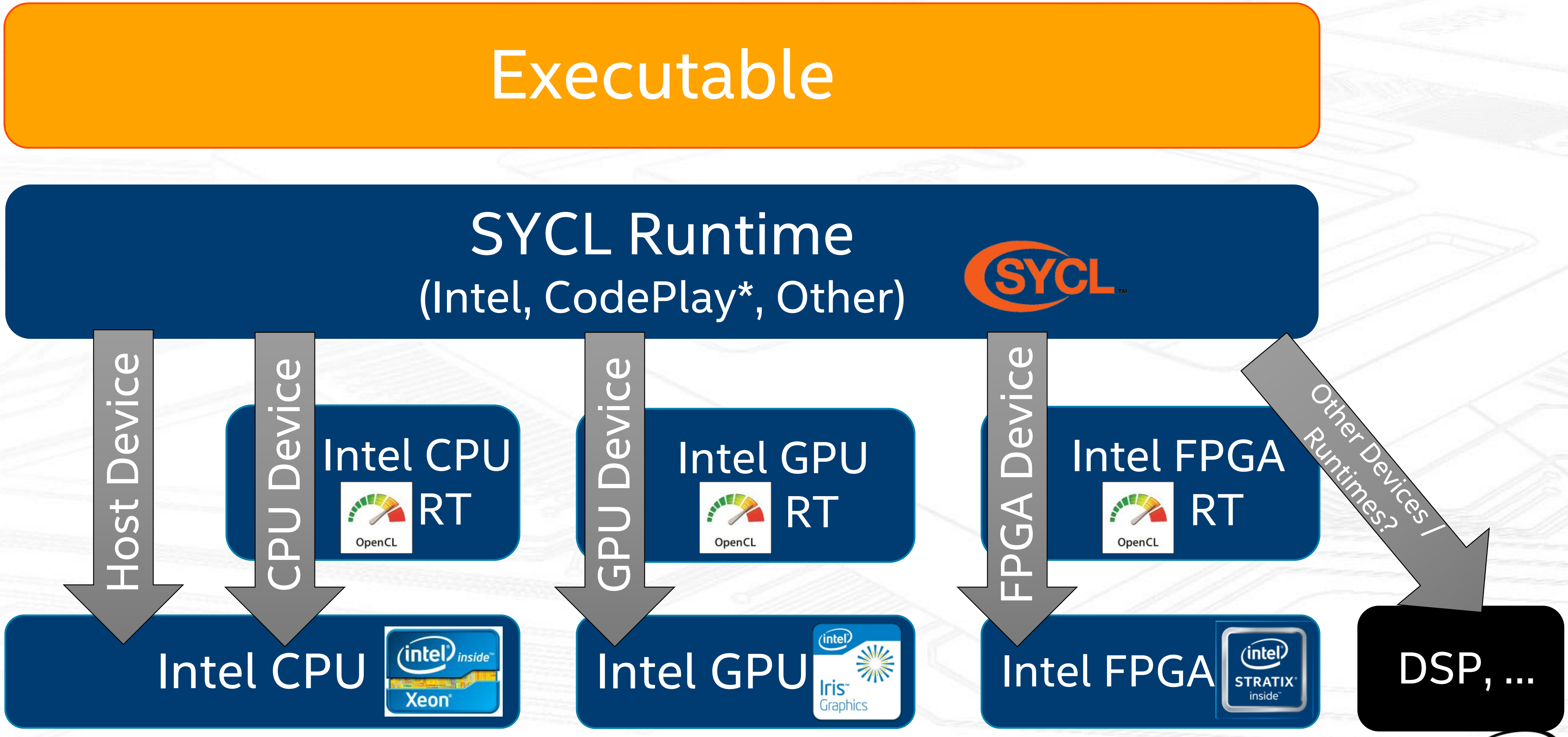


# SYCL Compilation Flow (Unified)

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;
class Hi;

int main() {
    const size_t array_size = 16;
    int data[array_size];
    {
        buffer<int, 1>  resultBuf{ data, range<1>{array_size} };
        queue q;
        q.submit([&](handler& cgh) {
            auto resultAcc = resultBuf.get_access<access::mode::write>(cgh);

            cgh.parallel_for<class Hi>(range<1>{array_size}, [=](id<1> i) {
                resultAcc[i] = static_cast<int>(i.get(0));
            });
        });
    }
    for( int i = 0; i < array_size; i++ ) {
        std::cout << "data[" << i << "] = " << data[i] << std::endl;
    }
    return 0;
}
```



# SYCL Execution Flow



# EXECUTION MODEL

## Data Parallelism is expressed using ND-Ranges

Total Work = # Work-groups x # Work-items per Work-group