SYCL for HiHat

Gordon Brown – Staff Software Engineer, SYCL
Ruyman Reyes - Principal Software Eng., Programming Models
Michael Wong – VP of R&D, SYCL Chair, Chair of ISO C++ TM/Low Latency
Andrew Richards – CEO Codeplay

HiHat 2017
Acknowledgement Disclaimer

Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.

I even lifted this acknowledgement and disclaimer from some of them.

But I claim all credit for errors, and stupid mistakes. These are mine, all mine!
Legal Disclaimer

This work represents the view of the author and does not necessarily represent the view of Codeplay.

Other company, product, and service names may be trademarks or service marks of others.
Codeplay - Connecting AI to Silicon

Products

- **ComputeCpp™**: C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™
- **ComputeAorta™**: The heart of Codeplay's compute technology enabling OpenCL™, SPIR™, HSA™ and Vulkan™

Company

- High-performance software solutions for custom heterogeneous systems
- Enabling the toughest processor systems with tools and middleware based on open standards
- Established 2002 in Scotland
- ~70 employees

Addressable Markets

- Automotive (ISO 26262)
- IoT, Smartphones & Tablets
- High Performance Compute (HPC)
- Medical & Industrial

Technologies: Vision Processing, Machine Learning, Artificial Intelligence, Big Data Compute

Customers

- Broadcom
- Renesas
- Imagination
- Qualcomm
- Movidius
- Intel
- AMD

© 2017 Codeplay Software Ltd.
Agenda

• SYCL
• SYCL Example
• SYCL for HiHat
• Distributed & Heterogeneous Programming in C/C++ (DHPCC++)
• BoF at SC17
Codeplay Goals

- To gauge whether it is worth doing ComputeSuite/SYCL stack HiHat-compatible
- To collaborate with the HiHat community in the overall research direction
- To evaluate if possible to be a “provider” for HiHat community (e.g., custom work on request or deployment of stack for HiHat community)
- To consolidate efforts of HiHat with C++ standardization
- To evaluate HiHat as a suitable safety-critical layer
- To integrate SYCL into ISO C++ along with other Modern C++ Heterogeneous/distributed frameworks
SYCL: A New Approach to Heterogeneous Programming in C++
SYCL for OpenCL

➢ Cross-platform, single-source, high-level, C++ programming layer
➢ Built on top of OpenCL and based on standard C++14
The SYCL Ecosystem

C++ Application

C++ Template Library

C++ Template Library

C++ Template Library

SYCL for OpenCL

OpenCL

CPU

GPU

APU

Accelerator

FPGA

DSP
How does SYCL improve heterogeneous offload and performance portability?

- SYCL is entirely standard C++
- SYCL compiles to SPIR
- SYCL supports a multi compilation single source model
Single Compilation Model

1. **C++ Source File**
   - **Device Source**

2. **Device Compiler**
   - **CPU Compiler**
   - **CPU Object**

3. **Linker**
   - **x86 ISA (Embedded Device Object)**
   - **x86 CPU**

4. **GPU**
Single Compilation Model

- C++ Source File
- Single Source Host & Device Compiler
- x86 ISA (Embedded Device Object)
- Tied to a single compiler chain
- x86 CPU
- GPU
Single Compilation Model

- **C++ Source File**
  - 3 different language extensions
  - 3 different compilers
  - 3 different executables

- **C++ AMP Source**
  - C++ AMP Compiler
  - x86 ISA (Embedded AMD ISA)
  - x86 CPU
  - AMD GPU

- **CUDA Source**
  - CUDA Compiler
  - x86 ISA (Embedded NVidia ISA)
  - X86 CPU
  - NVidia GPU

- **OpenMP Source**
  - OpenMP Compiler
  - x86 ISA (Embedded x86)
  - X86 CPU
  - SIMD CPU
SYCL is Entirely Standard C++

global_ vec_add(float *a, float *b, float *c) {
    return c[i] = a[i] + b[i];
}

vector<float> a, b, c;

#pragma parallel_for
for(int i = 0; i < a.size(); i++) {
    c[i] = a[i] + b[i];
}

cgh.parallel_for<class vec_add>(range, [=](cl::sycl::id<2> idx) {
    c[idx] = a[idx] + c[idx];
});

array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
    c[idx] = a[idx] + b[idx];
});

cgh.parallel_for<class vec_add>(range, [=](cl::sycl::id<2> idx) {
    c[idx] = a[idx] + c[idx];
});
SYCL Targets a Wide Range of Devices when using SPIR or SPIRV

- CPU
- GPU
- APU
- Accelerator
- FPGA
- DSP
Multi Compilation Model

- C++ Source File
- Device Code
- CPU Compiler
- CPU Object
- Linker
- x86 ISA (Embedded SPIR)
- Online Finalizer
- x86 CPU
- GPU
- SYCL Compiler
- SPIR
- SYCL device compiler
- Generating SPIR
Multi Compilation Model

GCC, Clang, VisualC++, Intel C++
Multi Compilation Model
Multi Compilation Model

C++ Source File → CPU Compiler → CPU Object → Linker → x86 ISA (Embedded SPIR) → x86 CPU

Device Code File → SYCL Compiler → SYCL Compiler → SPIR → Linker → OpenCL Online Finalizer

Device can be selected at runtime

Standard IR allows for better performance portability

SYCL does not mandate SPIR

© 2016 Codeplay Software Ltd.
Multi Compilation Model

C++ Source File

Device Code

CPU Compiler → CPU Object

SYCL Compiler → SPIR

SYCL Compiler → PTX

Linker → x86 ISA (Embedded SPIR)

OpenCL Online Finalizer

x86 CPU

SIMD CPU

GPU

APU

FPGA

DSP

© 2016 Codeplay Software Ltd.
Multi Compilation Model

PTX binary can be selected for NVidia GPUs at runtime
How does SYCL support different ways of representing parallelism?

➢ SYCL is an explicit parallelism model

➢ SYCL is a queue execution model

➢ SYCL supports both task and data parallelism
Representing Parallelism

cgh.single_task([&](){
    /* task parallel task executed once*/
});

cgh.parallel_for(range<2>(64, 64), [&](id<2> idx){
    /* data parallel task executed across a range */
});
How does SYCL make data movement more efficient?

➢ SYCL separates the storage and access of data

➢ SYCL can specify where data should be stored/allocated

➢ SYCL creates automatic data dependency graphs
Separating Storage & Access

Buffers managed data across host CPU and one or more devices

Buffers and accessors type safe access across host and device

Accessors are used to describe access

Buffers and accessors type safe access across host and device
Storing/Allocating Memory in Different Regions

Buffer

- Global Accessor: Memory stored in global memory region
- Constant Accessor: Memory stored in read-only memory region
- Local Accessor: Memory allocated in group memory region

Kernel
Data Dependency Task Graphs

Buffer A
- Read Accessor
- Write Accessor

Buffer B
- Read Accessor
- Write Accessor

Buffer C
- Read Accessor
- Write Accessor

Buffer D
- Read Accessor
- Write Accessor

Kernel A

Kernel B

Kernel C

Kernel A → Kernel C
Kernel B → Kernel C
Benefits of Data Dependency Graphs

• Allows you to describe your problems in terms of relationships
  • Don’t need to en-queue explicit copies

• Removes the need for complex event handling
  • Dependencies between kernels are automatically constructed

• Allows the runtime to make data movement optimizations
  • Pre-emptively copy data to a device before kernels
  • Avoid unnecessarily copying data back to the host after execution on a device
  • Avoid copies of data that you don’t need
Agenda

- SYCL
- SYCL Example
- SYCL for HiHat
- Distributed & Heterogeneous Programming in C/C++ (DHPCC++)
- BoF at SC17
So what does SYCL look like?

➢ Here is a simple example SYCL application; a vector add
Example: Vector Add
Example: Vector Add

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

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out) {
```
Example: Vector Add

```c++
#include <CL/sycl.hpp>

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out) {
    cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
    cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
    cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
}
```

The buffers synchronise upon destruction.
#include <CL/sycl.hpp>

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out) {
    cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
    cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
    cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
    cl::sycl::queue defaultQueue;
}

Example: Vector Add

```c++
#include <CL/sycl.hpp>

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out) {
    cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
    cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
    cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
    cl::sycl::queue defaultQueue;
    defaultQueue.submit([&](cl::sycl::handler &cgh) {
        });
}

Create a command group to define an asynchronous task
```
Example: Vector Add

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

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out) {
    cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
    cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
    cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
    cl::sycl::queue defaultQueue;
    defaultQueue.submit([&] (cl::sycl::handler &cgh) {
        auto inputAPtr = inputABuf.get_access<cl::sycl::access::read>(cgh);
        auto inputBPtr = inputBBuf.get_access<cl::sycl::access::read>(cgh);
        auto outputPtr = outputBuf.get_access<cl::sycl::access::write>(cgh);
    });
}
```
Example: Vector Add

```cpp
#include <CL/sycl.hpp>
template <typename T> kernel;

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out) {
  cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
  cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
  cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
  cl::sycl::queue defaultQueue;
  defaultQueue.submit([&](cl::sycl::handler &cgh) {
    auto inputAPtr = inputABuf.get_access<cl::sycl::access::read>(cgh);
    auto inputBPtr = inputBBuf.get_access<cl::sycl::access::read>(cgh);
    auto outputPtr = outputBuf.get_access<cl::sycl::access::write>(cgh);
    cgh.parallel_for<kernel<T>>(cl::sycl::range<1>(out.size()), [=](cl::sycl::id<1> idx) {
      // Your device code goes here
    });
  });
}
```

You must provide a name for the lambda.

Create a parallel_for to define the device code.
Example: Vector Add

```cpp
#include <CL/sycl.hpp>
template <typename T> kernel;

template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out) {
    cl::sycl::buffer<T, 1> inputABuf(inA.data(), out.size());
    cl::sycl::buffer<T, 1> inputBBuf(inB.data(), out.size());
    cl::sycl::buffer<T, 1> outputBuf(out.data(), out.size());
    cl::sycl::queue defaultQueue;
    defaultQueue.submit([&](cl::sycl::handler &cgh) {
        auto inputAPtr = inputABuf.get_access<cl::sycl::access::read>(cgh);
        auto inputBPtr = inputBBuf.get_access<cl::sycl::access::read>(cgh);
        auto outputPtr = outputBuf.get_access<cl::sycl::access::write>(cgh);
        cgh.parallel_for<kernel<T>>(cl::sycl::range<1>(out.size())),
            [=](cl::sycl::id<1> idx) {
                outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
            });
    });
}```
Example: Vector Add

```cpp
template <typename T>
void parallel_add(std::vector<T> inA, std::vector<T> inB, std::vector<T> out);

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

    parallel_add(inputA, inputB, output, count);
}
```
Complete Ecosystem: Applications on top of SYCL

- ISO C++ ParallelSTL TS and C++17 Parallel STL running on CPU and GPU
- Vision applications for self-driving cars ADAS
- Machine learning with Eigen and Tensorflow
- ISO C++ Parallel STL with Ranges
- SYCL-BLAS library
- Game AI libraries

http://sycl.tech
Comparison with KoKKos, Raja, SYCL, HPX

Similarities
• All exclusively C++
• All use Modern C++11, 14
• All use some form of execution policy to separate concerns
• All have some form of dimension shape for range of data
• All are aiming to be subsumed/integrated into future C++ Standard, but want to continue future exploratory research

Individual features

SYCL: Separate Memory Storage and Data access model using Accessors and Storage Buffers using Dependency graph Multiple Compilation Single Source model

Kokkos: Memory Space tells where user data resides (Host, GPU, HBM) Layout Space tells how user data is laid-out (Row/column major, AOS, SOA)

HPX: Distributed Computing nodes with asynchronous algorithm execution Execution Policy with executors (Par.on executors) and Grainsize

Raja: IndexSet and Segments
Agenda

- SYCL
- SYCL Example
- SYCL for HiHat
- Distributed & Heterogeneous Programming in C/C++ (DHPCC++)
- BoF at SC17
Standards vs Implementations
Current ComputeCpp components

SYCL offers a data-flow programming model for C++ that enables usage of heterogeneous platforms.

ComputeCpp implements the SYCL interface on top of a Runtime Scheduler with Memory dependency mechanism.

The current target of ComputeCpp is OpenCL, since the SYCL interface is based on OpenCL (e.g., has interoperability functions). However, ComputeCpp itself is target agnostic.
ComputeCpp components with Hihat

SYCL offers a data-flow programming model for C++ that enables usage of heterogeneous platforms.

ComputeCpp implements the SYCL interface on top of a Runtime Scheduler with Memory dependency mechanism.

The current target of ComputeCpp is OpenCL, since the SYCL interface is based on OpenCL (e.g., has interoperability functions). However, ComputeCpp itself is target agnostic.

HIHAT could easily act as our lower level, target-specific API. Later we could evaluate user-level layer.
ComputeCpp components with HiHat and C++20

SYCL offers a data-flow programming model for C++ that enables usage of heterogeneous platforms.

ComputeCpp implements the SYCL interface on top of a Runtime Scheduler with Memory dependency mechanism.

The current target of ComputeCpp is OpenCL, since the SYCL interface is based on OpenCL (e.g., has interoperability functions). However, ComputeCpp itself is target agnostic.

HIHAT could easily act as our lower level, target-specific API. Later we could evaluate user-level layer.

Future evolution

SYCL interface

C++2020 Heterogeneous Interface
SYCL on top of HiHAT?

What we provide

- High-level interface
- Retargetable to different backends
- Little overhead, simple programming
- Fully open standard
- Implementations customized to particular hardware stacks
- In future: align more closely with C++ futures/executors/coroutines
- In future: add Safety Critical SYCL

HiHAT wishlist

- Offer a low-level, close-to-metal interface
- Reuse of standard components but also ability to plug-in “binary blobs” for vendor-specific components
- Fully async API
- Device capability levels (i.e, this device can do this but not that)
- (Ideally) Time-constraint operations (for safety critical)
SYCL command group to HiHat example

1: `queue.submit([&](handler &h) {`
2:   `auto accA = bufA.get_access<access::mode::read>(h);`
3:   `auto accB = bufB.get_access<access::mode::write>(h);`
4:   `h.parallel_for<class myName>({bufB.size()}, [=](id<1> i) { accA[i] *= accB[i]; });`
5: });`

6: `hhuAlloc(size, platformTrait, &bufA.get_view(h), ...);
7: hhuCopy(Host To Device);`
8: `hhuAlloc(size, platformTrait, &bufB.get_view(h), ...);
9: hhuCopy(Host To Device);`
10: `void blob[2];
11: hhClosure closure;
12: hhActionHndl invokeHandle;
13: hhnRegFunc(HiHat_myName, Resource, 0, &invokeHandle);
14: blob[0] = accA; blob[2] = accB;
15: hhnMkClosure(invokeHandle, blob, 0, &closure);
16: hhuInvoke(closure, exec_pool, exec_cfg, Resource, NULL, &invokeHandle) ;`
How does Codeplay business work with HiHat?

Codeplay Software is a medium size, (currently) self-funded company.
Our work comes from Customer requests for compiler and runtime implementations or deployment of our ComputeSuite stack (SYCL + OpenCL + Custom hardware support) to customers.

- Questions to HiHat
  - What is the open source license? (GPL vs Apache)
  - What are the protections for IP? (e.g. can we take ideas into customer projects?)
  - Can we add HiHat support to our stack (e.g SYCL + HiHat + custom hardware) and make that a closed source implementation using parts of the open source components?
  - Will it be a certification process for HiHat-compliant devices/implementations?
The strength of Codeplay as a company

• Value proposition: fulltime customer support
• Long lifetime of company: in existence since 2002
• Deep commitment to Open Standards
• Active Research sponsorship (Ph.D) and projects
• Experience on all forms of accelerators
• Chairs key Standard Work groups
Agenda

• SYCL
• SYCL Example
• SYCL for HiHat
• Distributed & Heterogeneous Programming in C/C++ (DHPCC++)
• BoF at SC17
SC 17 Bof Distributed/Heterogeneous C++ in HPC

bof139s1 / Distributed & Heterogeneous Programming in C++ for HPC
which will be lead by Hal Finkel, especially if you will be at SC17 on Wednesday noon. Please let us know. Thanks.

http://sc17.supercomputing.org/conference-overview/sc17-schedule/

1. Kokkos: Carter Edwards
2. HPX: Hartmut Kaiser
3. Raja: David Beckinsale
4. SYCL: Michael Wong/Ronan Keryell/Ralph Potter
6. C++ AMP?: we agree to not add this group
7. C++ Standard: Michael Wong/Carter Edwards
8. AMD HCC: Tony Tye/Ben Sanders
10. Khronos Standards: Ronan Keryell

Other interests:
- Affinity BoF - bof154s1 / Cross-Layer Allocation and Management of Hardware Resources in Shared Memory Nodes

Khronos Booth talks at SC17

1. ParallelSTl from CPU to GPU
2. Machine learning with SYCL
3. Overview of SYCL and ComputeCPp
4. Xilinx SYCL implements TriSYCL on FPGA
Workshop on Distributed & Heterogeneous Programming in C/C++
At IWOCL 2018 Oxford

Topics of interest include, but are not limited to the following: Please consider submitted 10 page full paper (referred), 5 page short paper, or abstract-only talks.

- Future Heterogeneous programming C/C++ proposals (SYCL, Kokkos, Raja, HPX, C++AMP, Boost.Compute, CUDA …)
- ISO C/C++ related proposals and development including current related concurrency, parallelism, coroutines, executors
- C/C++ programming models for OpenCL
- Language Design Topics such as parallelism model, data model, data movement, memory layout, target platforms, static and dynamic compilation
- Applications implemented using these models including Neural Network, machine vision, HPC, CFD as well as exascale applications
- C/C++ Libraries using these models
- New proposals to any of the above specifications
- Integration of these models with other programming models
- Compilation techniques to optimize kernels using any of (clang, gcc, ..) or other compilation systems
- Performance or functional comparisons between any of these programming models
- Implementation of these models on novel architectures (FPGA, DSP, …) such as clusters, NUMA and PGAS
- Using these models in fault-tolerant systems
- Porting applications from one model to the other
- Reports on implementations
- Research on Performance Portability
- Debuggers, profilers and other tools
- Usage in a Safety and/or security context
- Applications implemented using similar models
- Other C++ Frameworks such as Chombo, Charm++ C++ Actor Framework, UPC++ and similar
Codeplay Goals

- To gauge whether it is worth doing ComputeSuite/SYCL stack HiHat-compatible
- To collaborate with the HiHat community in the overall research direction
- To evaluate if possible to be a “provider” for HiHat community (e.g, custom work on request or deployment of stack for HiHat community)
- To consolidate efforts of HiHat with C++ standardization
- To evaluate HiHat as a suitable safety-critical layer
- To integrate SYCL into ISO C++ along with other Modern C++ Heterogeneous/distributed frameworks
Thanks

@codeplaysoft
info@codeplay.com
codeplay.com
Backup
Heterogeneous Offloading
How do we offload code to a heterogeneous device?

➢ This can be answered by looking at the C++ compilation model
C++ Compilation Model

C++ Source File → CPU Compiler → CPU Object → Linker → x86 ISA → x86 CPU
C++ Compilation Model

C++ Source File → CPU Compiler → CPU Object → Linker → x86 ISA → x86 CPU
C++ Compilation Model

C++ Source File → CPU Compiler → CPU Object → Linker → x86 ISA → x86 CPU

?
How can we compile source code for a sub architectures?

➢ Separate source

➢ Single source
Separate Source Compilation Model

float *a, *b, *c;
...
kernel k = clCreateKernel(..., "my_kernel", ...);
clEnqueueWriteBuffer(..., size, a, ...);
clEnqueueWriteBuffer(..., size, a, ...);
clEnqueueNDRange(..., k, 1, {size, 1, 1}, ...);
clEnqueueWriteBuffer(..., size, c, ...);

Here we’re using OpenCL as an example

void my_kernel(__global float *a, __global float *b,
               __global float *c) {
    int id = get_global_id(0);
    c[id] = a[id] + b[id];
}
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
  c[idx] = a[idx] + b[idx];
});

Here we are using C++ AMP as an example
Here we are using C++ AMP as an example

```cpp
array_view<float> a, b, c;
extent<2> e(64, 64);
parallel_for_each(e, [=](index<2> idx) restrict(amp) {
    c[idx] = a[idx] + b[idx];
});
```
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
    c[idx] = a[idx] + b[idx];
});

Here we are using C++ AMP as an example
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
    c[idx] = a[idx] + b[idx];
});

Here we are using C++ AMP as an example
Benefits of Single Source

• Device code is written in C++ in the same source file as the host CPU code

• Allows compile-time evaluation of device code

• Supports type safety across host CPU and device

• Supports generic programming

• Removes the need to distribute source code
Describing Parallelism
How do you represent the different forms of parallelism?

➢ Directive vs explicit parallelism

➢ Task vs data parallelism

➢ Queue vs stream execution
Directive vs Explicit Parallelism

Examples:
• OpenMP, OpenACC

Implementation:
• Compiler transforms code to be parallel based on pragmas

Examples:
• SYCL, CUDA, TBB, Fibers, C++11 Threads

Implementation:
• An API is used to explicitly enqueue one or more threads

Here we’re using OpenMP as an example

```cpp
vector<float> a, b, c;
#pragma omp parallel for
for(int i = 0; i < a.size(); i++) {
    c[i] = a[i] + b[i];
}
```

Here we’re using C++ AMP as an example

```cpp
array_view<float> a, b, c;
extent<2> e(64, 64);
parallel_for_each(e, [=](index<2> idx)
    restrict(amp) {
        c[idx] = a[idx] + b[idx];
    });
```
Task vs Data Parallelism

Examples:

• OpenMP, C++11 Threads, TBB

Implementation:

• Multiple (potentially different) tasks are performed in parallel

Examples:

• C++ AMP, CUDA, SYCL, C++17 ParallelSTL

Implementation:

• The same task is performed across a large data set

Here we’re using TBB as an example

```cpp
vector<task> tasks = { ... };
tbb::parallel_for_each(tasks.begin(), tasks.end(), [=](task &v) {
    task();
});
```

Here we’re using CUDA as an example

```cpp
float *a, *b, *c;
cudaMalloc((void **)&a, size);
cudaMalloc((void **)&b, size);
cudaMalloc((void **)&c, size);
vec_add<<<64, 64>>>(a, b, c);
```
Queue vs Stream Execution

Examples:
- C++ AMP, CUDA, SYCL, C++17
- ParallelSTL

Implementation:
- Functions are placed in a queue and executed once per enqueuer

---

Here we’re using CUDA as an example

```c
float *a, *b, *c;
cudaMalloc((void **)a, size);
cudaMalloc((void **)b, size);
cudaMalloc((void **)c, size);
vec_add<<<64, 64>>>(a, b, c);
```

---

Here we’re using BrookGPU as an example

```c
reduce void sum (float a<>, reduce float r<>) {
  r += a;
}
float a<100>;
float r;
sum(a, r);
```
Data Locality & Movement
One of the biggest limiting factor in heterogeneous computing

➢ Cost of data movement in time and power consumption
Cost of Data Movement

• It can take considerable time to move data to a device
  • This varies greatly depending on the architecture

• The bandwidth of a device can impose bottlenecks
  • This reduces the amount of throughput you have on the device

• Performance gain from computation > cost of moving data
  • If the gain is less than the cost of moving the data it’s not worth doing

• Many devices have a hierarchy of memory regions
  • Global, read-only, group, private
  • Each region has different size, affinity and access latency
  • Having the data as close to the computation as possible reduces the cost
Cost of Data Movement

- 64-bit DP Op:
  - 20pJ
- 4x64-bit register read:
  - 50pJ
- 4x64-bit move 1mm:
  - 26pJ
- 4x64-bit move 40mm:
  - 1nJ
- 4x64-bit move DRAM:
  - 16nJ

Credit: Bill Dally, Nvidia, 2010
How do you move data from the host CPU to a device?

➢ Implicit vs explicit data movement
Implicit vs Explicit Data Movement

Examples:
- SYCL, C++ AMP

Implementation:
- Data is moved to the device implicitly via cross host CPU / device data structures

Examples:
- OpenCL, CUDA, OpenMP

Implementation:
- Data is moved to the device via explicit copy APIs

Here we’re using C++ AMP as an example

```cpp
array_view<float> ptr;
extent<2> e(64, 64);
parallel_for_each(e, [=](index<2> idx)
restrict(amp) {
    ptr[idx] *= 2.0f;
});
```

Here we’re using CUDA as an example

```cpp
float *h_a = { ... }, d_a;
cudaMalloc((void **)&d_a, size);
cudaMemcpy(d_a, h_a, size,
cudaMemcpyHostToDevice);
vec_add<<<64, 64>>>(a, b, c);
cudaMemcpy(d_a, h_a, size,
cudaMemcpyDeviceToHost);
```
How do you address memory between host CPU and device?

➢ Multiple address space

➢ Non-coherent single address space

➢ Cache coherent single address space
Comparison of Memory Models

• Multiple address space
  • SYCL 1.2, C++AMP, OpenCL 1.x, CUDA
  • Pointers have keywords or structures for representing different address spaces
  • Allows finer control over where data is stored, but needs to be defined explicitly

• Non-coherent single address space
  • SYCL 2.2, HSA, OpenCL 2.x, CUDA 4
  • Pointers address a shared address space that is mapped between devices
  • Allows the host CPU and device to access the same address, but requires mapping

• Cache coherent single address space
  • SYCL 2.2, HSA, OpenCL 2.x, CUDA 6
  • Pointers address a shared address space (hardware or cache coherent runtime)
  • Allows concurrent access on host CPU and device, but can be inefficient for large data