Directions in Parallel Programming: A Perspective from Codeplay

Alastair Murray
Compiler Research Engineer, Codeplay
Overview

• About Codeplay
• How we got to heterogeneous systems
• Parallel programming models
• SYCL for OpenCL
• SYCL examples
• Future C++ and the parallel STL
• Mapping high-level models to hardware
• Conclude
About Codeplay
Codeplay: Heterogeneous Systems Experts

Producing heterogeneous compilers since 2002.

- Developing whole compilers.
- Working on customer’s compilers (LLVM and proprietary tech).
- Optimization, bug-fixing, new languages, testing.
- Infrastructure for testing and tracking performance.

- Debuggers & profilers (e.g. LLDB).
- Collaborative R&D.
- Academic collaborations.
- Creating test-suites.
- Helping define new standards
- GPGPU compute.
- Graphics and games s/w.

32 expert development staff, based in Edinburgh.
Delivering Heterogeneous Standards

Which of these do you need to implement?

- C++
- SYCL
- OpenCL 2.0
- RenderScript
- HSA Foundation
- Windows 8
- Vulkan
- AMP
## What makes Heterogeneous Compiler Projects Unique?

<table>
<thead>
<tr>
<th>CPUs</th>
<th>GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widely understood and standardized.</td>
<td>New technologies and standards every year.</td>
</tr>
<tr>
<td>Can test by running existing software.</td>
<td>Need to write new test software for new features.</td>
</tr>
<tr>
<td>Instruction sets only add new instructions.</td>
<td>New GPUs completely change ISAs.</td>
</tr>
<tr>
<td>Compiler separated from hardware by OS.</td>
<td>Compilers, drivers and OS tightly integrated.</td>
</tr>
<tr>
<td><strong>Common CPU compilers can be 10-20 years old.</strong></td>
<td><strong>Need to implement much faster than CPU compilers!</strong></td>
</tr>
</tbody>
</table>

**CPUs**
- Widely understood and standardized.
- Can test by running existing software.
- Instruction sets only add new instructions.
- Compiler separated from hardware by OS.
- Common CPU compilers can be 10-20 years old.

**GPUs**
- New technologies and standards every year.
- Need to write new test software for new features.
- New GPUs completely change ISAs.
- Compilers, drivers and OS tightly integrated.
- Need to implement much faster than CPU compilers!
We're Hiring!

Open Positions

Full Time/Research Positions
- Compiler Engineer
- Tools Engineer
- Debugger Engineer
- R&D Compiler Engineer
- Runtime Engineer
- Software Engineer (UI Development For IDEs)
- GPGPU Video Codec Research Engineer

Graduate Positions
- Graduate Software Developer
- Intern Software Engineer

Search for: Codeplay jobs
How we got to Heterogeneous Systems
End of Dennard Scaling

- Moore's Law: Double transistor density every 2 years.
- Dennard's Law: Transistors will be faster and lower energy.
End of Dennard Scaling

“Moore's Law put lots more transistors on a chip ... but it's Dennard's Law that made them useful.”

Bob Colwell (Chief Architect at Intel for Pentium Pro through Pentium 4), 2013.
End of Dennard Scaling

Integer application performance (SPECint2000) over time

Microprocessor power dissipation (watts) over time.

Multi-core “Crisis”

- Single-thread performance stalled.
- Parallelism the only way forward.
  - But could regress single-thread performance?
  - How to program it?
- Yet …
Mobile Devices

iPhone, 2007.

http://commons.wikimedia.org/wiki/File:IPhone_2G_PSD_Mock.png
GPUs and GPGPU

• Also in 2007: CUDA 1.0
• Massively parallel.
  – But single compute units simpler than a CPU.
• No need to support “old” programming models.
FPGAs

• Also massively parallel.
  - Many DSP slices and memories.
  - Heterogeneous, customisable

• Can map GPGPU programs to FPGA.

• E.g.: Bing's ranking algorithm runs on FPGAs.

A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. 
Putnam et al., ISCA, 2014.
Parallel Programming Models
Parallel Programming Models

- Only considering heterogeneous programming.
- Better models are essential for writing complex software.
- Classifications:
  - **GPGPU languages** (OpenCL, CUDA).
    - Recently, **low-level hybrid GPU languages** (Metal, Vulkan).
  - **Library based** (ArrayFire, Thrust).
  - **Directive based** (OpenACC, OpenMP 4.0).
  - **High-level GPGPU languages** (C++AMP, SYCL).
OpenCL

Sequential loop

```
for (int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

Data parallel OpenCL

```
int i = get_global_id(0);
if (i < N)
    C[i] = A[i] + B[i];
```
OpenCL

http://rtcmagazine.com/articles/view/103610
SYCL for OpenCL
SYCL for OpenCL

- Pronounced 'sickle'.
- Cross-platform heterogeneous, parallel C++ programming model.
  - Developed within the Khronos Group.
  - Builds on concepts, portability and efficiency of OpenCL.
  - Ease of use and flexibility of C++.
- Single-source C++ development.
- Anything possible in core OpenCL 1.2 should be possible in SYCL.
Motivation Behind SYCL

- 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.
The SYCL Ecosystem

C++ Application

C++ Template Libraries  C++ Template Libraries  C++ Template Libraries

SYCL for OpenCL

OpenCL

Device X  Device Y  Device Z
Host Device

- Host Fallback
- Device X
- Device Y
- Device Z

SYCL Runtime
- Can be used as a target for debugging
- Can be used as an optional fallback if a kernel execution fails

OpenCL
Use Common Libraries

- **Single unified interface**
- **parallel_reduce()**
- **Host CPU**
  - Call standard library functions
- **OpenCL Device**
  - Call OpenCL built-in functions
SYCL Example #1: Vector Add
```cpp
#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>(output.size()), [=](id<1> idx) {
            outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
        });
    });
}
```

This is the bit that runs in parallel.
Comparison with OpenCL

SYCL for OpenCL

Traditional OpenCL
template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
}
#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) {}
```cpp
#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());
}
```

Construct three SYCL buffers and initialise them with the data from the std::vectors. Data is synchronised by RAII.
```cpp
#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;
}

Construct a SYCL queue to execute work on a device.
```
```cpp
#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, [&] () {
        // The command_group is enqueued asynchronously and is thread safe.
    });
}
```

Construct a SYCL command_group to define the work to be enqueued on a device.
```cpp
#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>();
  });
}
```

The SYCL runtime uses accessors to track dependencies across command_groups.

Construct three SYCL accessors with access modes, to give the device access to the data.
```cpp
#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>(output.size()), (id<1>(idx) {
            
        }));
    });
}
```

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

Call parallel_for() to execute a kernel function.

The range provided to the parallel_for() should match the size of the data buffers.
```cpp
#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>(output.size()), ([=](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.
```cpp
#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>(output.size()), ([=](id<1> idx) {
            outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
        }));
    });
}
```

Template the function
SYCL Example #2: Image Processing DSL
Image Processing DSL

- SYCL is an embedded DSL within C++.
- Use same principles to embed an application-orientated DSL within SYCL.
- Embed our domain knowledge to provide compile-time optimisations.
Image Processing DSL

\[ \text{out} = x + y \times 99 \]
Image Processing DSL

\[ \text{out} = x + y \times 99 \]
Image Processing DSL

\[ \text{out} = x + y \times 99 \]

diagram

expr<\text{tag::plus},
expr<\text{tag::terminal, term<int>>,
expr<\text{tag::multiplies},
expr<\text{tag::terminal, term<int>>>,
expr<\text{tag::terminal, term<const int>>>>>
Image Processing DSL

\[ I_{out} = I_x + I_y \times 99 \]
Image Processing DSL

\[ I_{out} = I_x + I_y \times 99 \]

// Declare operands.
cl::sycl::image<2> in_x = ...;
cl::sycl::image<2> in_y = ...;
cl::sycl::image<2> out;

// Evaluate x+y*99.
dsl::eval(queue, range,
            _1 = _2 + _3 * 99,
            out, in_x, in_y);
Image Processing DSL

\[ I_{\text{out}} = I_x + I_y \times 99 \]

// Declare operands.
cl::sycl::image<2> in_x = ...;
cl::sycl::image<2> in_y = ...;
cl::sycl::image<2> out;

// Evaluate x+y*99.
dsl::eval(queue, range,
_1 = _2 + _3 * 99,
out, in_x, in_y);

parallel_for < decltype(expr) >(range,
[=](cl::sycl::item<2> item) {
    // Construct a context per thread.
    // Provides implementations of operators.
    dsl::context ctx { item, sampler };

    // Evaluate expression tree using context.
    boost::proto::eval(expr, ctx);
});
Unsharp Mask

http://commons.wikimedia.org/wiki/File:Accutance_example.png
Unsharp Mask

\[ I_{out} = I_{in} + (I_{in} - G(I_{in})) \times w \]

1. Gaussian filter.
2. Subtract images.
3. Multiply by constant
4. Add images.

// Declare operands.
cl::sycl::image<2> in = ...;
cl::sycl::image<2> out;
float w;

// Evaluate unsharp mask.
dsl::eval(queue, range,
    _1 = _2 + (_2 - gaussian(_2)) * w),
    out, in);
Future C++ and the Parallel STL
C++ Roadmap

https://isocpp.org/std/status
Parallel STL

- Proposed parallelised version of the STL.

// Current C++11 sequential sort.
std::sort(data.begin(), data.end());

// C++17 explicitly sequential sort.
using namespace std::experimental::parallel;
std::sort(seq, data.begin(), data.end());

// C++17 parallel sort.
// Still using namespace std::experimental::parallel;
std::sort(par, data.begin(), data.end());
Parallel STL

• Not just built-in algorithms:

  ```cpp
  // C++17 parallel loop.
  using namespace std::experimental::parallel;
  std::for_each(par, data.begin(), data.end(),
    [](float &i) { /* Do something. */ });
  ```

• Is this SYCL example #3?
Mapping High-Level Models to Hardware
The SYCL Ecosystem

C++ Application

C++ Template Libraries

C++ Template Libraries

C++ Template Libraries

SYCL for OpenCL

OpenCL

Device X

Device Y

Device Z
The SYCL Ecosystem

SYCL specification does not mandate SPIR as a binary format.
SPIR

- Standard Portable Intermediate Representation.
- Low-level binary format.
  - Allows other languages to build on OpenCL ecosystem, e.g. SYCL.
- Also: SPIR-V, HSAIL.
Shared Source

Single source file compiled by host and device compilers

Source File

CPU Compiler

CPU Object

SYCL Device Compiler

SYCL runtime links the host code with the generated binary

SPIR Binary
Our device compiler generates a stub file containing the binary and kernel name and argument information. This is included and interpreted by the host side runtime.
Example: accessor

```cpp
accessor<float, 1, access::read> inputAPtr(inputABuf);
```

- **Host Compiler**
  - Construct OpenCL buffer and enqueue to device
- **Device Compiler**
  - Represent OpenCL kernel argument on device
Example: parallel_for()

```c++
parallel_for<class vadd>(range<1>(output.size()), ([=](id<1> idx) {
    outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
}));
```

- **Host Compiler**
  - Set arguments and enqueue kernel function

- **Device Compiler**
  - Generate the kernel function
Example: fmin()
Limitations

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

These apply only to SYCL kernel functions
Conclusion

- Parallel hardware is pervasive ...
- ... but complex software requires better parallel languages.
- Many options providing low-level portability.
Thank you