C++ on Accelerators: Supporting Single-Source SYCL and HSA Programming Models Using Clang

Victor Lomüller, Ralph Potter, Uwe Dolinsky

{victor,ralph,uwe}@codeplay.com

Codeplay Software Ltd.

17th March, 2016

Outline

1. Single Source Programming Models
2. Offloading C++ Code to Accelerators
3. Performance Results
4. Conclusion

SINGLE SOURCE PROGRAMMING MODELS

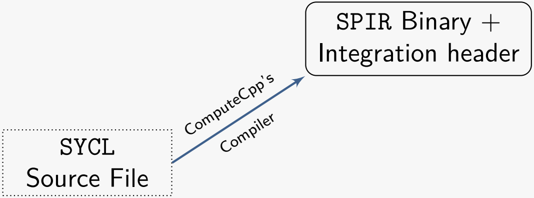
A few definitions

- "Host" is your system CPU
- "Device" is any accelerator: GPU, CPU, DSP...
- "Work-item" is a thread in OpenCL
- "Work-group" is a group of threads that can cooperate in OpenCL

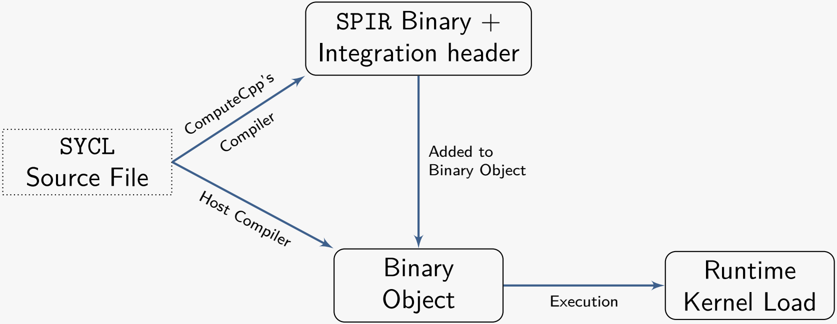
SYCL: Single Source C++ ecosystem for OpenCL

- An open and royalty-free standard from the Khronos Group.
- Cross-platform C++11 ecosystem for OpenCL 1.2.
  - Kernels and invocation code share the same source file
  - Standard C++ layer around OpenCL
  - No language extension
  - Works with any C++11 compiler
- Ease OpenCL application development

SYCL: Compilation



SYCL: Compilation



SYCL Vector Addition

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T> class SimpleVadd;

template<typename T>
void simple_vadd(T *VA, T *VB, T *VC, unsigned ORDER) {
  queue q;
  buffer<T, 1> bA(VA, range<1>(ORDER));
  buffer<T, 1> bB(VB, range<1>(ORDER));
  buffer<T, 1> bC(VC, range<1>(ORDER));

  q.submit([&](handler &cgh) {
    auto pA = bA.template get_access<access::mode::read>(cgh);
    auto pB = bB.template get_access<access::mode::read>(cgh);
    auto pC = bC.template get_access<access::mode::write>(cgh);

   cgh.parallel_for<class SimpleVadd<T> >(
        range<1>(ORDER), [=](id<1> it) {
          pC[it] = pA[it] + pB[it];
    });
  });
}
int main() {
  int A[4] = {1,2,3,4}, B[4] = {1,2,3,4}, C[4];
  simple_vadd<int>(A, B, C, 4);
  return 0;
}
```

SYCL Vector Addition

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T> class SimpleVadd;

template<typename T>
void simple_vadd(T *VA, T *VB, T *VC, unsigned ORDER) {
  queue q;
  buffer<T, 1> bA(VA, range<1>(ORDER));
  buffer<T, 1> bB(VB, range<1>(ORDER));
  buffer<T, 1> bC(VC, range<1>(ORDER));

  q.submit([&](handler &cgh) {
    auto pA = bA.template get_access<access::mode::read>(cgh);
    auto pB = bB.template get_access<access::mode::read>(cgh);
    auto pC = bC.template get_access<access::mode::write>(cgh);

    cgh.parallel_for<class SimpleVadd<T> >(
        range<1>(ORDER), [=](id<1> it) {
          pC[it] = pA[it] + pB[it];
    });
  });
}
int main() {
  int A[4] = {1,2,3,4}, B[4] = {1,2,3,4}, C[4];
  simple_vadd<int>(A, B, C, 4);
  return 0;
}
```

SYCL kernel

SYCL Vector Addition

```
cgh.parallel_for<class SimpleVadd<T> >(
  range<1>(ORDER), [=](id<1> it) {
    pC[it] = pA[it] + pB[it];
});
```

What is needed for running the kernel on the device ?

- lambda body
- `accessor::operator[](id)`
- `id` copy constructor
- … and all functions used by `accessor::operator[]` and `id` copy constructor

SYCL Vector Addition

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T>
class SimpleVadd;

template <typename T>
void do_add(T *pC, const T *pA, const T *pB, size_t idx) {
  pC[it] = pA[it] + pB[it];
}

template <typename T>
void simple_vadd(T *VA, T *VB, T *VC, unsigned ORDER) {
  queue q;
  buffer<T, 1> bA(VA, range<1>(ORDER));
  buffer<T, 1> bB(VB, range<1>(ORDER));
  buffer<T, 1> bC(VC, range<1>(ORDER));

  q.submit([&](handler &cgh) {
    auto pA = bA.template get_access<access::mode::read>(cgh);
    auto pB = bB.template get_access<access::mode::read>(cgh);
    auto pC = bC.template get_access<access::mode::write>(cgh);

   cgh.parallel_for<class SimpleVadd<T> >(
        range<1>(ORDER), [=](id<1> it) {
     do_add(pC.get_pointer(), pA.get_pointer(), pB.get_pointer(), it);
    });
  });
}
```

Perform the addition in a separate function

SYCL Vector Addition: Host View

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T>
class SimpleVadd;

template <typename T>
void do_add(T *pC, const T *pA, const T *pB, size_t idx) {
  pC[it] = pA[it] + pB[it];
}

template <typename T>
void simple_vadd(T *VA, T *VB, T *VC, unsigned ORDER) {
  queue q;
  buffer<T, 1> bA(VA, range<1>(ORDER));
  buffer<T, 1> bB(VB, range<1>(ORDER));
  buffer<T, 1> bC(VC, range<1>(ORDER));

  q.submit([&](handler &cgh) {
    auto pA = bA.template get_access<access::mode::read>(cgh);
    auto pB = bB.template get_access<access::mode::read>(cgh);
    auto pC = bC.template get_access<access::mode::write>(cgh);

   cgh.parallel_for<class SimpleVadd<T> >(
        range<1>(ORDER), [=](id<1> it) {
      do_add(pC.get_pointer(), pA.get_pointer(), pB.get_pointer(), it);
    });
  });
}
```

```
template <typename T>
void do_add(T *pC, const T *pA, const T *pB, size_t idx);
```

SYCL Vector Addition: Device View

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
template <typename T>
class SimpleVadd;

template <typename T>
void do_add(T *pC, const T *pA, const T *pB, size_t idx) {
  pC[it] = pA[it] + pB[it];
}

template <typename T>
void simple_vadd(T *VA, T *VB, T *VC, unsigned ORDER) {
  queue q;
  buffer<T, 1> bA(VA, range<1>(ORDER));
  buffer<T, 1> bB(VB, range<1>(ORDER));
  buffer<T, 1> bC(VC, range<1>(ORDER));

  q.submit([&](handler &cgh) {
    auto pA = bA.template get_access<access::mode::read>(cgh);
    auto pB = bB.template get_access<access::mode::read>(cgh);
    auto pC = bC.template get_access<access::mode::write>(cgh);

   cgh.parallel_for<class SimpleVadd<T> >(
       range<1>(ORDER), [=](id<1> it) {
     do_add(pC.get_pointer(), pA.get_pointer(), pB.get_pointer(), it);
    });
  });
}
```

```
template <typename T>
void do_add(__global T *pC, const __global T *pA,
            const __global T *pB, size_t idx);
```

SYCL Hierarchical

```
#define LOCAL_RANGE ...
[...]
h.parallel_for_work_group<class sycl_reduction>(
  range<1>(std::max(length, LOCAL_RANGE) / LOCAL_RANGE),
  range<1>(LOCAL_RANGE), [=](group<1> grp) {
    int scratch[LOCAL_RANGE];

    parallel_for_work_item(grp, [=](item<1> it) {
      int globalId = grp.get() * it.get_range() + it.get();
      if (globalId < length) scratch[it.get()] = aIn[globalId];
    });

    int min = (length < local) ? length : local;
    for (int offset = min / 2; offset > 1; offset /= 2) {
      parallel_for_work_item(grp, range<1>(offset), [=](item<1> it) {
        scratch[it] += scratch[it + offset];
      });
    }
    aOut[grp.get()] = scratch[0] + scratch[1];
 });
```

SYCL Hierarchical

```
#define LOCAL_RANGE ...
[...]
h.parallel_for_work_group<class sycl_reduction>(
  range<1>(std::max(length, LOCAL_RANGE) / LOCAL_RANGE),
  range<1>(LOCAL_RANGE), [=](group<1> grp) {
    int scratch[LOCAL_RANGE];

    parallel_for_work_item(grp, [=](item<1> it) {
      int globalId = grp.get() * it.get_range() + it.get();
      if (globalId < length) scratch[it.get()] = aIn[globalId];
    });

    int min = (length < local) ? length : local;
    for (int offset = min / 2; offset > 1; offset /= 2) {
      parallel_for_work_item(grp, range<1>(offset), [=](item<1> it) {
        scratch[it] += scratch[it + offset];
      });
    }
    aOut[grp.get()] = scratch[0] + scratch[1];
 });
```

Normal per-work-item execution

SYCL Hierarchical

```
#define LOCAL_RANGE ...
[...]
h.parallel_for_work_group<class sycl_reduction>(
  range<1>(std::max(length, LOCAL_RANGE) / LOCAL_RANGE),
  range<1>(LOCAL_RANGE), [=](group<1> grp) {
    int scratch[LOCAL_RANGE];

    parallel_for_work_item(grp, [=](item<1> it) {
      int globalId = grp.get() * it.get_range() + it.get();
      if (globalId < length) scratch[it.get()] = aIn[globalId];
    });

    int min = (length < local) ? length : local;
    for (int offset = min / 2; offset > 1; offset /= 2) {
      parallel_for_work_item(grp, range<1>(offset), [=](item<1> it) {
        scratch[it] += scratch[it + offset];
      });
    }
    aOut[grp.get()] = scratch[0] + scratch[1];
 });
```

Allocated in the OpenCL local memory

Normal per-work-item execution

Once per-work-group execution

SYCL Hierarchical

```
#define LOCAL_RANGE ...
void do_sum(int& dst, int a, int b) {
  dst = a + b;
}
[...]
h.parallel_for_work_group<class sycl_reduction>(
  range<1>(std::max(length, LOCAL_RANGE) / LOCAL_RANGE),
  range<1>(LOCAL_RANGE), [=](group<1> grp) {
  int scratch[LOCAL_RANGE];

  parallel_for_work_item(grp, [=](item<1> it) {
    int globalId = grp.get() * it.get_range() + it.get();
    if (globalId < length) scratch[it.get()] = aIn[globalId];
  });

  int min = (length < local) ? length : local;
  for (int offset = min / 2; offset > 1; offset /= 2) {
    parallel_for_work_item(grp, range<1>(offset), [=](item<1> it) {
      // scratch[it] += scratch[it + offset];
      do_sum(scratch[it], scratch[it], scratch[it + offset]);
    });
  }
  // aOut[grp.get()] = scratch[0] + scratch[1];
  do_sum(scratch[0], scratch[0], scratch[1]);
  do_sum(aOut[grp.get()], scratch[0], scratch[1]);
});
```

Perform the addition in a separate function

SYCL Hierarchical: Host View

```
#define LOCAL_RANGE ...
void do_sum(int& dst, int a, int b) {
  dst = a + b;
}
[...]
h.parallel_for_work_group<class sycl_reduction>(
  range<1>(std::max(length, LOCAL_RANGE) / LOCAL_RANGE),
  range<1>(LOCAL_RANGE), [=](group<1> grp) {
  int scratch[LOCAL_RANGE];

  parallel_for_work_item(grp, [=](item<1> it) {
    int globalId = grp.get() * it.get_range() + it.get();
    if (globalId < length) scratch[it.get()] = aIn[globalId];
  });

  int min = (length < local) ? length : local;
  for (int offset = min / 2; offset > 1; offset /= 2) {
    parallel_for_work_item(grp, range<1>(offset), [=](item<1> it) {
      // scratch[it] += scratch[it + offset];
      do_sum(scratch[it], scratch[it], scratch[it + offset]);
    });
  }
  // aOut[grp.get()] = scratch[0] + scratch[1];
  do_sum(scratch[0], scratch[0], scratch[1]);
  do_sum(aOut[grp.get()], scratch[0], scratch[1]);
});
```

```
void do_sum(int& dst, int a, int b)
```

```
void do_sum(int& dst, int a, int b)
```

```
void do_sum(int& dst, int a, int b)
```

SYCL Hierarchical: Device View

```
#define LOCAL_RANGE ...
void do_sum(int& dst, int a, int b) {
  dst = a + b;
}
[...]
h.parallel_for_work_group<class sycl_reduction>(
  range<1>(std::max(length, LOCAL_RANGE) / LOCAL_RANGE),
  range<1>(LOCAL_RANGE), [=](group<1> grp) {
  int scratch[LOCAL_RANGE];

  parallel_for_work_item(grp, [=](item<1> it) {
    int globalId = grp.get() * it.get_range() + it.get();
    if (globalId < length) scratch[it.get()] = aIn[globalId];
  });

  int min = (length < local) ? length : local;
  for (int offset = min / 2; offset > 1; offset /= 2) {
    parallel_for_work_item(grp, range<1>(offset), [=](item<1> it) {
      // scratch[it] += scratch[it + offset];
      do_sum(scratch[it], scratch[it], scratch[it + offset]);
    });
  }
  // aOut[grp.get()] = scratch[0] + scratch[1];
  do_sum(scratch[0], scratch[0], scratch[1]);
  do_sum(aOut[grp.get()], scratch[0], scratch[1]);
});
```

```
void do_add(__local int &dst, int a, int b);
```

```
void do_add(__local int &dst, int a, int b);
```

```
void do_add(__global int &dst, int a, int b);
```

SYCL Hierarchical

```
#define LOCAL_RANGE ...
void do_sum(int& dst, int a, int b) {
  dst = a + b;
}
[...]
h.parallel_for_work_group<class sycl_reduction>(
  range<1>(std::max(length, LOCAL_RANGE) / LOCAL_RANGE),
  range<1>(LOCAL_RANGE), [=](group<1> grp) {
  int scratch[LOCAL_RANGE];

  parallel_for_work_item(grp, [=](item<1> it) {
    int globalId = grp.get() * it.get_range() + it.get();
    if (globalId < length) scratch[it.get()] = aIn[globalId];
  });

  int min = (length < local) ? length : local;
  for (int offset = min / 2; offset > 1; offset /= 2) {
    parallel_for_work_item(grp, range<1>(offset), [=](item<1> it) {
      // scratch[it] += scratch[it + offset];
      do_sum(scratch[it], scratch[it], scratch[it + offset]);
    });
  }
  // aOut[grp.get()] = scratch[0] + scratch[1];
  do_sum(scratch[0], scratch[0], scratch[1]);
  do_sum(aOut[grp.get()], scratch[0], scratch[1]);
});
```

Not (exactly) the same function!

```
void do_add(__local int &dst, int a, int b);
```

```
void do_add(__local int &dst, int a, int b);
```

```
void do_add(__global int &dst, int a, int b);
```

C++ Front-end for HSAIL

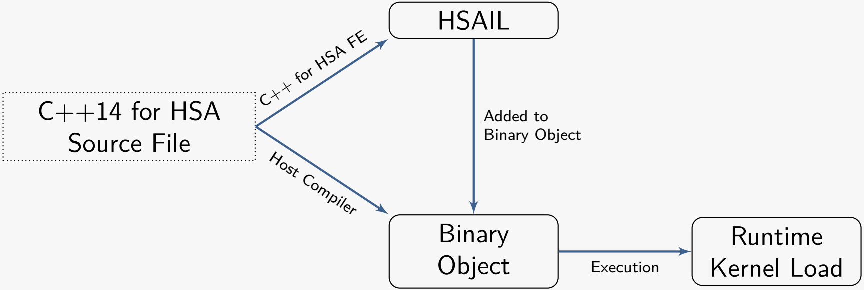
HSA Foundation

- Defines an intermediate language: HSAIL (HSA Intermediate Language)
- Defines a runtime for compute
- Primarily targets heterogeneous SoCs: CPU + Accelerators are on the same chip
- More aimed to support systems with Shared Virtual Memory (but not limited to)
- Provide some features not provided by OpenCL's lowest common denominator model
  - e.g. function pointers, recursion, alloca

Ralph's work

- C++14 based programming model
- Compile to HSAIL (HSA Intermediate Language)

C++ Front-end for HSAIL: Compilation



C++ Front-end for HSAIL: Example

```
[[hsa::kernel]]
void vector_add(float* a, float* b,
                float* c) {
  uint32_t i =
    rt::builtin::workitemabsid(0);
  a[i] = b[i] + c[i];
}


float* a, b, c;
// Asynchronous dispatch
auto future =
  rt::parallel_for(rt::throughput,
                   grid_size,
                   vector_add, a, b, c);
future.wait();


// Synchronous dispatch
rt::parallel_for(rt::throughput,
                 grid_size,
                 vector_add, a, b, c);
```

- SYCL and the C++ HSA Front End creates similar challenges for the compiler

C++ Front-end for HSAIL: Example

Address space 0 (clang default)

```
[[hsa::kernel]]
void vector_add(float* a, float* b,
                float* c) {
  uint32_t i =
    rt::builtin::workitemabsid(0);
  a[i] = b[i] + c[i];
}
```

Address space 1

```
float* a, b, c;
// Asynchronous dispatch
auto future =
  rt::parallel_for(rt::throughput,
                   grid_size,
                   vector_add, a, b, c);
future.wait();

// Synchronous dispatch
rt::parallel_for(rt::throughput,
                 grid_size,
                 vector_add, a, b, c);
```

- SYCL and the C++ HSA Front End creates similar challenges for the compiler
- Default address spaces are different:
  - Global address space is mapped to 0 (clang default)
  - Private address space is mapped to 1
- These defaults need to be propagated through all called functions

SYCL vs C++ Front-end for HSAIL

What is common

- The compiler must be able to change variable/argument types
  - A single function in the source file can be derived in many forms
  - Default address space must be configurable
- It needs to maintain several device function versions
  - Required by SYCL's hierarchical API
- Context information is used to determine where a function will be executed

OFFLOADING C++ CODE TO ACCELERATORS

Codeplay's Offload Engine

- Call graph duplication algorithm
  - *Cooper, Pete, et al.* "Offload–automating code migration to heterogeneous multicore systems." High Performance Embedded Architectures and Compilers 2010.
- Successfully used in "Offload for PS3"
  - Integrated in some PS3 games such as NASCAR The Game 2011
  - Work partially funded by the *PEPPHER project*
  - `www.peppher.eu`
- Integrated into clang for heterogeneous system: "OffloadCL for clang"
- Similar works done at the LLVM IR
  - Work partially funded by the *CARP project*
  - `http://carp.doc.ic.ac.uk`

Codeplay's Offload Engine in a nutshell

- Some attributes:
  - `__offload__`: explicitly identify a device function and its space
  - `address_space_of_locals`: to select address space defaults and propagation options
- Set of hooks in clang to intercept
  - Function calls
  - Variable declarations
- Extended overload resolution
  - A host function is different from a device function
  - Support for "multi-space" device functions
  - Calling context is important
- The Offloading core
  - Clone host functions into device functions
  - Address space inference and promotion

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}

void host_function(int * arg) {
  arg = workload(arg);
}

#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}
```

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}

void host_function(int * arg) {
  arg = workload(arg);
}

#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}
```

```
FunctionDecl 0x66a79d0 used workload 'int *(int *)'
|-ParmVarDecl 0x66a7900 used arg 'int *'
`-CompoundStmt 0x66a7be8
  |-DeclStmt 0x66a7b70
  | `-VarDecl 0x66a7a90 end 'int *' cinit
  |   `-BinaryOperator 0x66a7b48 'int *' '+'
  |     |-ImplicitCastExpr 0x66a7b30 'int *'
  |     | `-DeclRefExpr 0x66a7ae8 'int *' lvalue ParmVar 0x66a7900 'arg' 'int *'
  |     `-IntegerLiteral 0x66a7b10 'int' 4
  `-ReturnStmt 0x66a7bc8
    `-ImplicitCastExpr 0x66a7bb0 'int *'
      `-DeclRefExpr 0x66a7b88 'int *' lvalue ParmVar 0x66a7900 'arg' 'int *'
```

```
| `-DeclRefExpr 0x66e7e70 'int *(int *)' lvalue Function 0x66a79d0 'workload' 'int *(int *)'
```

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}

void host_function(int * arg) {
  arg = workload(arg);
}

#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}
```

Explicitly offload the function

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}

void host_function(int * arg) {
  arg = workload(arg);
}

#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}

__attribute__((__offload__(2)))
void device_function(__global int * arg) {
  arg = workload(arg);
}
```

Explicitly offload the function in the space 1

Explicitly offload the function in the space 2

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}

void host_function(int * arg) {
  arg = workload(arg);
}

#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}
```

```
workload(__global int * arg);
```

- We need to resolve the call to the "workload" function for an argument `__global int*`
- We make clang recognize the conversion from a non-default address space to the default one as a standard conversion
  - e.g. `int __global **__global*` is a standard conversion of `int***`

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}

void host_function(int * arg) {
  arg = workload(arg);
}

#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}
```

```
__global int * workload(__global int * arg);
```

- We go through the function AST to infer the return type of our new function.
- Conflicting return types raise an error.

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}

void host_function(int * arg) {
  arg = workload(arg);
}

#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}

__global int *  workload(__global int * arg) {
  __global int * end = arg + 4;
  // ...
  return arg;
}
```

- Using the `TreeTransform` class, we rebuild the function body for the duplicated function
- For each `clang::VarDecl`, we infer its type using its initialization
- We reinstantiate templates only if needed

OffloadCL: Simple Example

```
int* workload(int * arg) {
  int * end = arg + 4;
  //...
  return arg;
}
```

```
FunctionDecl 0x66a79d0 used workload 'int *(int *)'
|-ParmVarDecl 0x66a7900 used arg 'int *'
`-CompoundStmt 0x66a7be8
  |-DeclStmt 0x66a7b70
  | `-VarDecl 0x66a7a90 end 'int *' cinit
  |   `-BinaryOperator 0x66a7b48 'int *' '+'
  |     |-ImplicitCastExpr 0x66a7b30 'int *'
  |     | `-DeclRefExpr 0x66a7ae8 'int *' lvalue ParmVar 0x66a7900 'arg' 'int *'
  |     `-IntegerLiteral 0x66a7b10 'int' 4
  `-ReturnStmt 0x66a7bc8
    `-ImplicitCastExpr 0x66a7bb0 'int *'
      `-DeclRefExpr 0x66a7b88 'int *' lvalue ParmVar 0x66a7900 'arg' 'int *'
```

```
void host_function(int * arg) {
  arg = workload(arg);
}
```

```
| `-DeclRefExpr 0x66e7e70 'int *(int *)' lvalue Function 0x66a79d0 'workload' 'int *(int *)'
```

```
#define __global \
__attribute__((address_space(0xFFFF00)))

__attribute__((__offload__))
void device_function(__global int * arg) {
  arg = workload(arg);
}
```

```
| `-DeclRefExpr 0x66e85c0 '__global int *(__global int *)' lvalue
  Function 0x66e8320 'workload' '__global int *(__global int *)'
```

```
__global int *  workload(__global int * arg) {
  __global int * end = arg + 4;
  // ...
  return arg;
}
```

```
FunctionDecl 0x66e8320 used workload '__global int *(__global int *)'
|-ParmVarDecl 0x66e83c0 used arg '__global int *'
|-CompoundStmt 0x66e8598
| |-DeclStmt 0x66e8520
| | `-VarDecl 0x66e8460 end '__global int *' cinit
| |   `-BinaryOperator 0x66e84f8 '__global int *' '+'
| |     |-ImplicitCastExpr 0x66e84e0 '__global int *'
| |     | `-DeclRefExpr 0x66e84b8 '__global int *' lvalue ParmVar 0x66e83c0 'arg' '__glo
| |     `-IntegerLiteral 0x66a7b10 'int' 4
| `-ReturnStmt 0x66e8578
|   `-ImplicitCastExpr 0x66e8560 '__global int *'
|     `-DeclRefExpr 0x66e8538 '__global int *' lvalue ParmVar 0x66e83c0 'arg' '__global
`-OffloadFnAttr 0x66e8420 Implicit 1
```

OffloadCL: Overloading

1. Standard overload resolution
2. Try to select a function in the same space
3. If not possible, try to select host and then duplicate
4. If not possible, select an offloaded function in another space

```
#define __global \
__attribute__((address_space(0xFFFF00)))

void call(int*);
__attribute__((__offload__(2)))
void call(__global int*);

__attribute__((__offload__(2)))
void caller(__global int* ptr) {
  call(ptr);
}

__attribute__((__offload__))
void caller(__global int* ptr) {
  // match against the host function
  // and duplicate
  call(ptr);
}
```

OffloadCL: Overloading

1. Standard overload resolution
2. Try to select a function in the same space
3. If not possible, try to select host and then duplicate
4. If not possible, select an offloaded function in another space
   - There is no hierarchy between spaces

```
__attribute__((__offload__(2)))
void call(int*);
__attribute__((__offload__(3)))
void call(int*);

__attribute__((__offload__(1)))
void caller(int* i) {
  call(i); // the call is ambiguous
}
```

OffloadCL: Overloading

The offloaded state is part of the signature, so:

- Function prototypes can only differ by their return types if in different spaces
- The distinction is made using the calling context

```
#define __global \
__attribute__((address_space(0xFFFF00)))
#define __local \
__attribute__((address_space(0xFFFF01)))

int* get();

__attribute__((__offload__))
__global float* get();

__attribute__((__offload__(2)))
__local int* get();
```

What is missing to have a full SYCL/HSA support ?

OffloadCL creates the infrastructure to support offloading

Does not handle language specifics

- What a kernel entry point is:

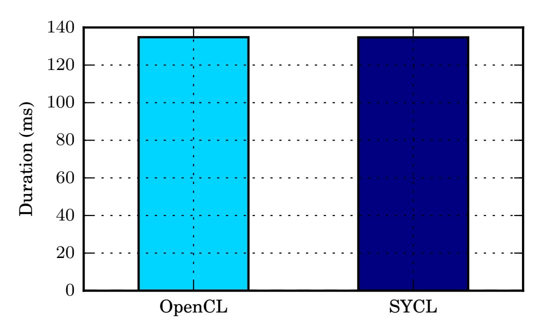
SYCL: it is identified by a `sycl_kernel` attribute (hidden behind `parallel_for`)

HSA FE: it is identified by a `hsa::kernel`

- Languages restrictions, e.g. no function pointers in SYCL
- Compilation product

PERFORMANCE RESULTS

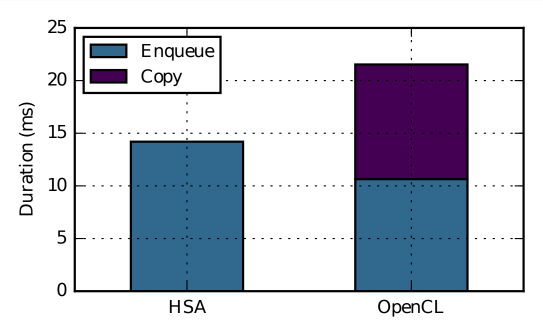
SYCL Performance



8x8 Discrete Cosine Transform

- Measurement made on an AMD Radeon HD 5400 (Cedar)
- DCT is ALU bound

C++ Front End for HSA Performance



8x8 Discrete Cosine Transform

- Measurement made on an AMD A10-7850K APU
- DCT is ALU bound

CONCLUSION

Conclusion

- OffloadCL is our single source enabler technology
- Offers flexibility via
  - Call graph duplication and function space management
  - Extended overloading resolution logic
  - Automatic address space inference and promotion
- Keeps a clear separation between host and device code
- No overhead on the generated code

ComputeCpp

ComputeCpp™

- ComputeCpp is our SYCL implementation
- Offload is the core technology behind ComputeCpp's compiler
- An evaluation program is available
  - Register your interest on our website!

ComputeCPP is trademark of Codeplay Software Ltd.

We're Hiring!
codeplay.com/careers/

codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

@codeplaysoft

info@codeplay.com

codeplay.com