Arm, Cambridge, UK

From C++ for OpenCL to C++ for accelerator devices

Anastasia Stulova, Neil Hickey,
Sven van Haastregt, Marco Antognini, Kevin Petit

LLVM Developers' Meeting, 22–23 October 2019

Outline

About OpenCL

C++ for OpenCL

Implementation in Clang

Generalization to C++

Summary and future work

Outline

About OpenCL

C++ for OpenCL

Implementation in Clang

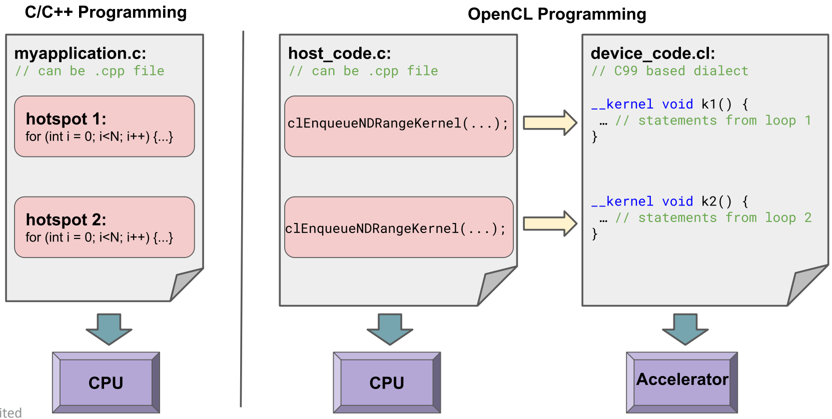
Generalization to C++

Summary and future work

What is OpenCL?

Programming model for offloading computation to accelerators standardized by Khronos Group

- Originally intended for GPGPU
- Evolving towards heterogeneous HW i.e, GPUs, CPUs, DSPs, FPGAs, custom accelerators



Outline

About OpenCL

C++ for OpenCL

Implementation in Clang

Generalization to C++

Summary and future work

What is C++ for OpenCL?

OpenCL C v2.0:
- kernels,
- address spaces,
- special types,
...

\+

C++17:
- classes,
- inheritance,
- templates,
- type deduction,
...

=

C++ for OpenCL

```
> clang -std=clc++ test.cl
template<class T> T add( T x, T y ) {
  return x + y;
}
__kernel void k_float(__global float *a, __global float *b) {
  auto index = get_global_id(0);
  a[index] = add(b[index], b[index+1]);
}
__kernel void k_int(__global int *a, __global int *b) {
  auto index = get_global_id(0);
  a[index] = add(b[index], b[index+1]);
}
```

It is not OpenCL C++ from the Khronos Registry!

Outline

About OpenCL

C++ for OpenCL

Implementation in Clang

Generalization to C++

Summary and future work

Work breakdown

- Driver support
  - Extended build flags `-cl-std`/`-std` to accept `clc++`/`CLC++`
  - Added new language mode
- Enabled OpenCL special types - opaque-like builtin types
  - Unified between C and C++ dialects
- Enabled all OpenCL C functions from builtin header `opencl-c.h`
- Enabled all Khronos standard extensions
- Added various restrictions to C++ features, disallowing:
  - Virtual functions
  - Exceptions
  - `dynamic_cast` operator
  - std libs
- New behavior for interplay between OpenCL and C++ functionality...

Interplay between OpenCL and C++

- Kernel function
- Address spaces
- Global objects construction/destruction
- ...

Kernel function in C++ mode

OpenCL host API:

```
clCreateKernel(... "foo" ...); // create kernel with the name 'foo'
```

- Name has to be preserved during device compilation to be referred to/from the host
- Prevent mangling i.e. disallow C++-like function features:
  - Overloading
  - Use as templates
  - Use as member functions

arm

Kernel function in C++ mode

OpenCL host API:

```
clCreateKernel(... "foo" ...); // create kernel with the name 'foo'
```

- Name has to be preserved during device compilation to be referred to/from the host
- Prevent mangling i.e. disallow C++-like function features:
  - Overloading
  - Use as templates
  - Use as member functions
- => Implicitly add C linkage early during parsing

Address spaces

- Language feature to bind objects to memory segments

```
__attribute__((address_space(1))) int i;    // i is located in memory segment ID 1
__attribute__((address_space(1))) int *ptr; // ptr points to int in memory segment ID 1
```

Address spaces

- Language feature to bind objects to memory segments

```
__attribute__((address_space(1))) int i;    // i is located in memory segment ID 1
__attribute__((address_space(1))) int *ptr; // ptr points to int in memory segment ID 1
```

- Part of qualified type
  - Originally defined in Embedded C ISO/IEC JTC1 SC22 WG14 N1169 s5.1
  - Extended to OpenCL kernel language
- Unlike regular qualifier it’s implicitly present
  - Can’t be cast away

Address spaces

- Language feature to bind objects to memory segments

```
__attribute__((address_space(1))) int i;    // i is located in memory segment ID 1
__attribute__((address_space(1))) int *ptr; // ptr points to int in memory segment ID 1
```

- Part of qualified type
  - Originally defined in Embedded C ISO/IEC JTC1 SC22 WG14 N1169 s5.1
  - Extended to OpenCL kernel language
- Unlike regular qualifier it's implicitly present
  - Can't be cast away
- In OpenCL: `__private`, `__global`, `__local`, `__constant`, `/*__generic*/`

Address spaces in C vs C++

- 90 occurrences of ”qualifier” in C99 spec

Address spaces in C vs C++

- 90 occurrences of ”qualifier” in C99 spec
- Language features affected by addr spaces:
  - Type qualifier inference
  - Conversions/Casts
  - Some overloading
  - Generation of IR

Address spaces in C vs C++

- 90 occurrences of ”qualifier” in C99 spec
- Language features affected by addr spaces:
  - Type qualifier inference
  - Conversions/Casts
  - Some overloading
  - Generation of IR

- 207 occurrences in C++17 spec
- Additional features:
  - Cast operators
  - References
  - Templates
  - Type deduction
  - Overloading
  - Implicit object parameter
  - Builtin operators

arm

Address spaces - General issue in C++

In C++ there are abstractions that are specialized e.g. classes and objects

```
__global MyClass c1; // MyClass allocated in global memory
c1.dosomething();    // implicitly dosomething(MyClass *this)
__local MyClass c2;  // MyClass allocated in local memory
c2.dosomething();    // implicitly dosomething(MyClass *this)
```

What address space should `this` param point to?

Address spaces - General issue in C++

In C++ there are abstractions that are specialized e.g. classes and objects

```
__global MyClass c1; // MyClass allocated in global memory
c1.dosomething();    // implicitly dosomething(MyClass *this)
__local MyClass c2;  // MyClass allocated in local memory
c2.dosomething();    // implicitly dosomething(MyClass *this)
```

What address space should `this` param point to?

- Class definition is parsed ahead of object instantiations
- Definition of member functions are commonly in a separate translation unit

Address spaces - General issue in C++

In C++ there are abstractions that are specialized e.g. classes and objects

```
__global MyClass c1; // MyClass allocated in global memory
c1.dosomething();    // implicitly dosomething(MyClass *this)
__local MyClass c2;  // MyClass allocated in local memory
c2.dosomething();    // implicitly dosomething(MyClass *this)
```

What address space should `this` param point to?

- Class definition is parsed ahead of object instantiations
- Definition of member functions are commonly in a separate translation unit
- Undesirable to duplicate member functions (at source or binary) for each address space
  - Negatively impacts compilation speed and binary size
- Address spaces are not known ahead of compilation in C++
  - Arbitrarily specified in source using `__attribute__((address_space(N)))`

arm

Address spaces - OpenCL approach

- OpenCL v2.0 defines the *generic* address space

```
__global int a;
__local int b;
/*__generic*/ int *ptr;
if (c)
  ptr = &a;
else
  ptr = &b;
// ptr can point into a segment in either local or global memory
```


arm

Address spaces - OpenCL approach

- OpenCL v2.0 defines the *generic* address space

```
__global int a;
__local int b;
/*__generic*/ int *ptr;
if (c)
  ptr = &a;
else
  ptr = &b;
// ptr can point into a segment in either local or global memory
```

- We use generic address space for abstract behavior in C++
  - Note: `__constant` can't be converted to/from `/*__generic*/`

arm

Address spaces - OpenCL approach example

```
1  class MyClass {
2   void dosomething();            // void dosomething(__generic MyClass *this)
3                                  // MyClass(__generic MyClass *this)
4   MyClass(MyClass &c);           // MyClass(__generic MyClass *this, __generic MyClass &c)
5   MyClass(MyClass &c) __local;   // MyClass(__local MyClass *this, __generic MyClass &c)
6  }
7  __global MyClass c1;      // calls ctor line 3 where arg 'this' is an addr space cast of
8                            // 'c1' from '__global MyClass*' to '__generic MyClass*'
9  __local MyClass c2(c1);   // calls ctor line 5 where arg 'this' is an allocation 'c2' of
10                           // 'MyClass' in __local address space, 2nd arg is as on line 7
```


arm

Address spaces - OpenCL approach example

```
1  class MyClass {
2   void dosomething();           // void dosomething(__generic MyClass *this)
3                                 // MyClass(__generic MyClass *this)
4   MyClass(MyClass &c);          // MyClass(__generic MyClass *this, __generic MyClass &c)
5   MyClass(MyClass &c) __local; // MyClass(__local MyClass *this, __generic MyClass &c)
6  }
7  __global MyClass c1;     // calls ctor line 3 where arg 'this' is an addr space cast of
8                           // 'c1' from '__global MyClass*' to '__generic MyClass*'
9  __local MyClass c2(c1); // calls ctor line 5 where arg 'this' is an allocation 'c2' of
10                          // 'MyClass' in __local address space, 2nd arg is as on line 7
```

Note: methods used with `__constant` addr space objects have to be overloaded explicitly

Global ctors/dtors

- Global variables are shared among kernels
  - Initialization/destruction can't be done at the boundaries of kernel execution

Global ctors/dtors

- Global variables are shared among kernels
  - Initialization/destruction can't be done at the boundaries of kernel execution
- Solution
  - ctors - changed initialization stub to a kernel function
    - Can be invoked from host before kernel executions

Global ctors/dtors

- Global variables are shared among kernels
  - Initialization/destruction can’t be done at the boundaries of kernel execution
- Solution
  - ctors - changed initialization stub to a kernel function
    - Can be invoked from host before kernel executions
  - dtors - standard C++ ABI registers dtor callbacks with global objects as parameters
    - Callbacks aren’t trivial to support between host and device
    - Need to add ability to pass objects in arbitrarily addr space
    - `/*__generic*/` addr space approach won’t work for `__constant`
    - TODO: ABI change

Outline

About OpenCL

C++ for OpenCL

Implementation in Clang

Generalization to C++

Summary and future work

What about ”pure” C++ on accelerators?

Many commonalities in accelerators

- Memory segments (i.e address spaces)
- Vectors/SIMD
- Parallel execution
- ...

arm

Towards generic C++ support for accelerators

- **(WIP)** Only OpenCL logic is conditioned on a language mode

  Generic functionality example (`lib/Sema/SemaOverload.cpp`)

```
void Sema::AddOverloadCandidate(...
  // Check that the constructor is capable of constructing an object in the destination address space.
  if (!Qualifiers::isAddressSpaceSupersetOf(Constructor->getMethodQualifiers().getAddressSpace(),
                                            CandidateSet.getDestAS())) {
    Candidate.Viable = false;
    Candidate.FailureKind = ovl_fail_object_addrspace_mismatch;
  }
```

Towards generic C++ support for accelerators

- **(WIP)** Only OpenCL logic is conditioned on a language mode

Generic functionality example (`lib/Sema/SemaOverload.cpp`)

```
void Sema::AddOverloadCandidate(...
  // Check that the constructor is capable of constructing an object in the destination address space.
  if (!Qualifiers::isAddressSpaceSupersetOf(Constructor->getMethodQualifiers().getAddressSpace(),
                                            CandidateSet.getDestAS())) {
    Candidate.Viable = false;
    Candidate.FailureKind = ovl_fail_object_addrspace_mismatch;
  }
```

OpenCL specific example (`lib/Sema/SemaDeclCXX.cpp`)

```
CXXConstructorDecl *Sema::DeclareImplicitCopyConstructor(...
  QualType ArgType = Context.getTypeDeclType(ClassDecl);
  if (Context.getLangOpts().OpenCLPlusPlus)
    ArgType = Context.getAddrSpaceQualType(ArgType, LangAS::opencl_generic);
```


arm

Towards generic C++ support for accelerators

- **(WIP)** Only OpenCL logic is conditioned on a language mode

  Generic functionality example (`lib/Sema/SemaOverload.cpp`)

```
void Sema::AddOverloadCandidate(...
  // Check that the constructor is capable of constructing an object in the destination address space.
  if (!Qualifiers::isAddressSpaceSupersetOf(Constructor->getMethodQualifiers().getAddressSpace(),
                                            CandidateSet.getDestAS())) {
    Candidate.Viable = false;
    Candidate.FailureKind = ovl_fail_object_addrspace_mismatch;
  }
```

  OpenCL specific example (`lib/Sema/SemaDeclCXX.cpp`)

```
CXXConstructorDecl *Sema::DeclareImplicitCopyConstructor(...
  QualType ArgType = Context.getTypeDeclType(ClassDecl);
  if (Context.getLangOpts().OpenCLPlusPlus)
    ArgType = Context.getAddrSpaceQualType(ArgType, LangAS::opencl_generic);
```

- **(Future)** Still need to generalize some concepts from OpenCL
  - Will likely require docs/spec work before completion of implementation

arm

Outline

About OpenCL

C++ for OpenCL

Implementation in Clang

Generalization to C++

Summary and future work

Summary

- C++ for OpenCL enabled in Clang 9
  - Mainly backwards compatible with OpenCL C
  - Most of C++ logic is enabled
  - Implemented without big infrastructural and architectural change
  - Experimental phase - many bugs are discovered and missing features

    See `https://clang.llvm.org/docs/OpenCLSupport.html`
- Implementation generalizes address space logic for C++ where applicable
- Documentation can be found in `https://://github.com/KhronosGroup/Khronosdotorg/blob/master/api/opencl/assets/CXX_for_OpenCL.pdf`

Future work

- Complete documentation and implementation for OpenCL
  - To be compiled offline for OpenCL v2.0 compatible drivers into SPIR-V
  - Future extensions for drivers
    - e.g. to avoid manual steps for global ctors/dtors
- Finalize generalization to C++ - concept, documentation, implementation
- Perform full functionality testing

Special thanks to the community!!! <3

- To John McCall for invaluable feedback and reviews!
- To David Rohr for testing, submitting bugs, providing suggestions and being so patient while waiting for bugs to be fixed!
  - Very motivating use of the new language for experiments at CERN!
- To OpenCL WG at Khronos Group for supporting the idea and hosting the documentation!

Thanks! Questions?

Appendix - Key Phabricator reviews

- Enabling OpenCL types and extensions: D57824, D58179, D62208, D62588, D65286
- Address spaces related
  - Reference types: D53764, D58634
  - Implicit object parameter: D54862, D59988
  - Method qualifiers: D55850, D62156, D64569
  - Generalization of method overloading to C++: D57464
  - Inference: D62584, D62591, D65744, D66137
  - Conversions/Casts: D52598, D58346, D60193
- Kernel function mangling: D60454
- Global ctor/dtor: D61488, D62413