Arm, Cambridge, UK

C++ for OpenCL Programming Language

Presented by
Anastasia Stulova

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

IWOCL 2020, 27–29 April 2020

Outline

Introduction

Key features

Case study

Testing and evaluation

Development flow

Related work

Resources

Future Work

Outline

Introduction

Key features

Case study

Testing and evaluation

Development flow

Related work

Resources

Future Work

Why C++ for OpenCL?

- OpenCL is well-established technology in many areas.
- Growth in complexity of applications run on accelerators.
- Separate host-device flow allows max customization for any architecture.
  - Developing mature optimizing compilers is time consuming and not always practical.

What is C++ for OpenCL?

<u>OpenCL C v2.0</u>:
- kernels,
- address spaces,
- special types,
...

+

<u>C++17</u>:
- classes,
- inheritance,
- templates,
- type deduction,
...

=

C++ for OpenCL

```
$ cat test.cl
template<class T> T add( T x, T y ) {
  return x + y;
}
__kernel void test(__global float *a, __global float *b) {
  auto index = get_global_id(0);
  a[index] = add(b[index], b[index + 1]);
}

$ clang -std=clc++ test.cl
```

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

Design goals

- Backwards compatibility to OpenCL C (v2.0).
  - Reuse existing code, libraries.
  - Reuse existing tools.
  - Familiar development flow.
- Enable as much of modern C++ as possible.
  - Gradual transition to familiar C++ programming paradigms.

Outline

Introduction

Key features

Case study

Testing and evaluation

Development flow

Related work

Resources

Future Work

Differences with OpenCL C

- Implicit conversions are stricter.

```
const int *ptrconst;
int *ptr = ptrconst; // invalid initialization discards const qualifier
```

- Explicit representation of NULL using `nullptr`.
- `restrict` is not supported.
  - Clang provides experimental support of `__restrict`.
- More restricted usage of `goto`.
- ObjC Blocks are not supported.

C++ feature restrictions

- Virtual functions.
- Exceptions.
- RTTI e.g. `dynamic_cast`, `typeid`.
- Non-placement `new`/`delete` operators.
- C++ std libs.

Improved OpenCL C features

- Variadic macros.
- Atomics.
  - Operators with C11 atomic types.
  - Legacy atomics with generic address space.
- ...

Address spaces 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 the `this` parameter point to?

Address spaces 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 the `this` parameter point to?

- Class declarations are parsed ahead of object instantiations.
- Member function definitions are typically in a separate translation unit.

Address spaces 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 the `this` parameter point to?

- Class declarations are parsed ahead of object instantiations.
- Member function definitions are typically 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 - 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
```

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` cannot be converted to/from `/*__generic*/`.

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                         // ptr to '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
11 c1.dosomething();       // calls method from line 2 casting ptr to 'c1' to __generic
12 c2.dosomething();       // calls method from line 2 casting ptr to 'c2' to __generic
```

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                         // ptr to '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
11 c1.dosomething();       // calls method from line 2 casting ptr to 'c1' to __generic
12 c2.dosomething();       // calls method from line 2 casting ptr to 'c2' to __generic
```

Note: methods used with `__constant` addr space objects have to be overloaded using address space method qualifier explicitly.

Address spaces - other rules

- Default address space follows OpenCL C v2.0 logic.
  - References inherit rules from pointers => `/*__generic*/`.
  - Static data members are in `__global`.
  - No default for non-pointer/reference dependent types (i.e. template params), `decltype` or alias declarations.
- Lambdas can be qualified by an address space like methods.

```
[&] (int i)  __global { ... };
```

- Special `addrspace_cast` operator.

```
/*__generic*/ int *genptr = ...;
__global int *globptr = addrspace_cast<__global int*>(genptr);
```

- More elaborate description in the official documentation.

https://github.com/KhronosGroup/Khronosdotorg/blob/master/api/opencl/assets/CXX_for_OpenCL.pdf

Global constructors/destructors

- Global variables are shared among kernels.
  - Initialization/destruction cannot be done at the boundaries of kernel execution.

Global constructors/destructors

- Global variables are shared among kernels.
  - Initialization/destruction cannot be done at the boundaries of kernel execution.
- Solution.
  - ctors - changed initialization stub to a kernel function.
    - Can be enqueued from host before kernel executions.
    - In OpenCL v2.0 drivers application has to perform this step manually.
    - Clang generates a kernel with initialization code per translation unit that can be queried from the binary (see `https://clang.llvm.org/docs/UsersManual.html#constructing-and-destroying-global-objects`).

Global constructors/destructors

- Global variables are shared among kernels.
  - Initialization/destruction cannot be done at the boundaries of kernel execution.
- Solution.
  - ctors - changed initialization stub to a kernel function.
    - Can be enqueued from host before kernel executions.
    - In OpenCL v2.0 drivers application has to perform this step manually.
    - Clang generates a kernel with initialization code per translation unit that can be queried from the binary (see `https://clang.llvm.org/docs/UsersManual.html#constructing-and-destroying-global-objects`).
  - dtors - WIP, requires large ABI change due to incompatibility with OpenCL execution model.
    - Potentially less critical as program context is destroyed at this point.

Kernel function in C++ mode

OpenCL host API:

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

- Name has to be preserved during the 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.

Kernel function in C++ mode

OpenCL host API:

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

- Name has to be preserved during the 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 `extern C`.

Outline

- Introduction
- Key features
- Case study
- Testing and evaluation
- Development flow
- Related work
- Resources
- Future Work

Convolution from Arm Compute Library - row computation

Statically compute sum of N factors!

```
template<typename T /*conv data type*/, size_t N /*conv dim*/>
class onerow
{
  uchar16 data;
  const short (&mat)[N]; // matrix of coefficients
  ...
  template<size_t S /*step number*/> T mulacc() = delete;
  template<> T mulacc<0>() { return vec_cast<T>(data.s01234567) * mat[0];}
  template<> T mulacc<1>() { return vec_cast<T>(data.s12345678) * mat[1] + mulacc<0>();}
  ...
  // up to (conv dim - 1)
  template<> T mulacc<8>() { return vec_cast<T>(data.s89abcdef) * mat[8] + mulacc<7>();}
};
```

Convolution continued - NxN

Compute full NxN convolution using `onerow` helper.

```
template<typename T, size_t N>
inline T convolution(Image &src, const short (&mat)[N][N], uint scale)
{
    T pixels = 0;

    for (size_t i = 0; i < N; ++i)
    {
      uchar16 temp = vload16(0, src.offset(-((int)N) / 2, i - N / 2));
      onerow<T, N> rowi(temp, mat[i]);
      pixels += rowi.template mulacc<N - 1>();
    }

    return pixels / static_cast<T>(scale);
}
```

Convolution continued - kernel with 3x3 convolution

```
class Image {
   ...
   __global uchar *offset(int x, int y);
};

// using vector convert functions from OpenCL C
template<typename To, typename From> inline To vec_cast(From ty);

__kernel void convolution3x3_static(...) {
  ...
  short8 pixels = convolution<short8, 3>
      (src, {{MAT0, MAT1, MAT2}, {MAT3, MAT4, MAT5}, {MAT6, MAT7, MAT8}}, SCALE);
```

Convolution continued - kernel with 3x3 convolution

```
class Image {
   ...
   __global uchar *offset(int x, int y);
};

// using vector convert functions from OpenCL C
template<typename To, typename From> inline To vec_cast(From ty);

__kernel void convolution3x3_static(...) {
  ...
  short8 pixels = convolution<short8, 3>
      (src, {{MAT0, MAT1, MAT2}, {MAT3, MAT4, MAT5}, {MAT6, MAT7, MAT8}}, SCALE);
```

OpenCL C sources available in

`https://github.com/ARM-software/ComputeLibrary/tree/master/src/core/CL/cl_kernels`.

- ~200 lines of convolution (3x3, 5x5, 7x7, 9x9) can be replaced by ~30 lines in C++ for OpenCL.
- Without observable performance loss!

Outline

Introduction

Key features

Case study

Testing and evaluation

Development flow

Related work

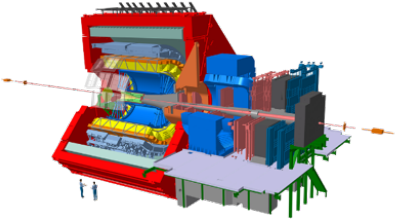
Resources

Future Work

Evaluation

- OpenCL C content is nearly fully supported.
  - Most of conformance v2.0 tests pass (11 fail out of 1384).
  - Validation on benchmarks is in progress.
- Experimental testing for Vulkan using **clspv** and **clvk**.
- Porting applications from other languages.
  - ALICE experiment at CERN.

Porting ALICE event reconstruction to C++ for OpenCL



- ALICE: A Large Ion Collider Experiment.
  - From 2021 it will record collisions of lead nuclei at the LHC at a rate of 50 kHz.
  - Several thousand particles in each collision, whose trajectories must be found (using measured 3d space points).
  - All data will be processed in real time using GPUs.

- Written in generic C++ with preprocessor macros substituted into language keywords (`https://github.com/AliceO2Group/AliceO2`).
  - CUDA (since 2010) and OpenCL 1.2 with AMD C++ extensions (since 2015).
  - Ongoing research to support HIP and C++ for OpenCL.
    - Fully compiled (**~12K lines**) from C++ for OpenCL down to SPIR-V using **clang-10** and **llvm-spirv**.
    - CERN to test SPIR-V injection on Mali Driver.

Image courtesy of CERN

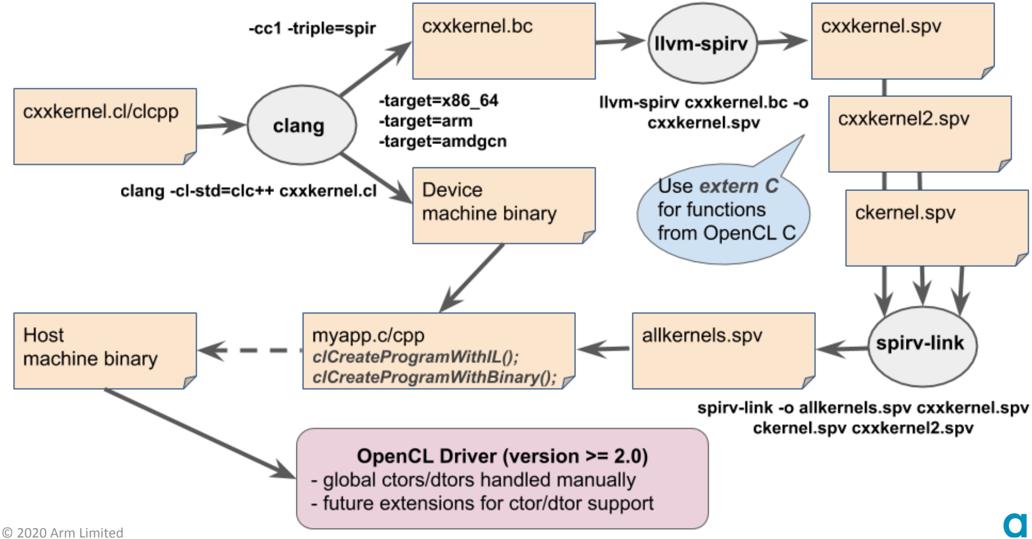
Overheads - compile and runtime, binary size

- OpenCL features are handled in the same way as for OpenCL C.
- C vs C++ is an old debate.
- Most of C style features have the same overhead in C++.
- C++ often hides overheads.
  - E.g. implicit object pointer parameter.
- C++ language facilitates more optimizations.
- Modern compilers are very good at optimizing C++ code.
  - E.g. devirtualization, ctor/dtor inlining.
- A lot of material about writing low overhead C++ code.
  - ISO/IEC TR 18015:2006 - Technical Report on C++ Performance.

Outline

- Introduction
- Key features
- Case study
- Testing and evaluation
- Development flow
- Related work
- Resources
- Future Work

How can applications use C++ for OpenCL?



cxxkernel.cl/clcpp

clang

-cc1 -triple=spir

-target=x86_64
-target=arm
-target=amdgcn

clang -cl-std=clc++ cxxkernel.cl

cxxkernel.bc

llvm-spirv

llvm-spirv cxxkernel.bc -o cxxkernel.spv

cxxkernel.spv

cxxkernel2.spv

ckernel.spv

Use *extern C* for functions from OpenCL C

Device machine binary

spirv-link

spirv-link -o allkernels.spv cxxkernel.spv ckernel.spv cxxkernel2.spv

allkernels.spv

myapp.c/cpp
clCreateProgramWithIL();
clCreateProgramWithBinary();

Host machine binary

OpenCL Driver (version >= 2.0)
- global ctors/dtors handled manually
- future extensions for ctor/dtor support

Outline

Introduction

Key features

Case study

Testing and evaluation

Development flow

Related work

Resources

Future Work

Comparison to other languages

Language	Vendors	Host/dev perf tuning	Host/dev compilation	Single source	Dev flow
C++ for OpenCL	Multiple	Fully manual for any arch	Separate phases	No	OpenCL style
SYCL	Multiple	Compiler / limited manual	Likely separate phases	Yes	C++ library / metaprogramming
CUDA	Nvidia	Compiler	Likely mixed separate + combined phases	Yes	C++ dialect
HIP	Multiple	Compiler	Currently separate	Yes	C++ dialect
Metal SL	Apple	Fully manual	Separate phases	No	OpenCL style

Outline

Introduction

Key features

Case study

Testing and evaluation

Development flow

Related work

Resources

Future Work

Resources

- Detailed documentation can be found in
  `https://github.com/KhronosGroup/Khronosdotorg/blob/master/api/opencl/assets/CXX_for_OpenCL.pdf`
- Any feedback to documentation can be submitted in `https://github.com/KhronosGroup/OpenCL-Docs`
- Information about support in Clang `https://clang.llvm.org/docs/UsersManual.html#cxx-for-opencl`
- Implementation status can be tracked through `https://clang.llvm.org/docs/OpenCLSupport.html`
- Report bugs and any missing features on `https://bugs.llvm.org/`

Outline

Introduction

Key features

Case study

Testing and evaluation

Development flow

Related work

Resources

Future Work

Future work

The plan from **the community**:

- Complete implementation in Clang i.e. missing features or bugs.
- Finalize documentation.
- Add support for C++ libraries.
- Perform full functionality testing.
- Provide more support to/maintenance for the application developers.

Special thanks to the community!!! <3

- To John McCall from Apple for invaluable feedback and reviews!
- To David Rohr from CERN 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!

Presented by
Anastasia Stulova