The C++ for OpenCL Programming Language Documentation (Draft)

Khronos® OpenCL Working Group

Version V2.2-11-20-gd5c9ab4, Wed, 16 Oct 2019 15:56:54 +0000: from git branch: AnastasiaStulova-cxx4opencl_draft commit: d5c9ab49d979c9630d65f0ce167ff8708a4c415f

Table of Contents

Chapter 1. Introduction

This language is built on top of OpenCL C v2.0 and C++17 enabling most of regular C++ features in OpenCL kernel code. Most functionality from OpenCL C and C++ is inherited. Restrictions are documented in *Restrictions* sections.

The main aim of this document is to describe behavior that is not documented in the OpenCL C and C++17 specifications. This is mainly related to interactions between OpenCL and C++ language features.

Chapter 2. The C++ for OpenCL Programming Language

This chapter documents various language features of C++ for OpenCL.

2.1. Restrictions

This section describes various restrictions and limitations for language features from OpenCL v2.0 and C++17.

2.1.1. Restrictions to C++ features

The following C++ language features are not supported:

- Virtual functions
- Exceptions
- `dynamic_cast` operator
- Non-placement `new`/`delete` operators
- Standard C++ libraries

2.2. Address spaces

C++ for OpenCL inherits address space behavior from `OpenCL C v2.0 s6.5`.

This section only documents behavior related to C++ features. For example conversion rules are extended from the qualification conversion but the compatibility is determined using notation of sets and overlapping of address spaces from Embedded C (`ISO/IEC JTC1 SC22 WG14 N1021 s3.1.3`). For OpenCL it means that implicit conversions are allowed from a named address space (except for `constant) to generic (OpenCL C v2.0 6.5.5). The reverse conversion is only allowed explicitly. The constant` address space does not overlap with any other and therefore no valid conversion between `__constant` and any other address space exists. Most of the rules follow this logic.

2.2.1. Casts

C-style casts follow `OpenCL C v2.0 rules (s6.5.5)`. All cast operators permit conversion to generic address space implicitly. However converting from generic to named address spaces can only be done using `addrspace_cast`. Note that conversions between `__constant` and any other other address space are disallowed.

TODO: Example with `addrspace_cast`.

2.2.2. Deduction

Address spaces are not deduced for:

- non-pointer/non-reference template parameters or any dependent types except for template specializations.
- non-pointer/non-reference class members except for static data members that are deduced to `__global` address space.
- non-pointer/non-reference alias declarations.
- decltype expressions.

```
template <typename T>
void foo() {
  T m; // address space of m will be known at template instantiation time.
  T * ptr; // ptr points to generic address space object.
  T & ref = ...; // ref references an object in generic address space.
};

template <int N>
struct S {
  int i; // i has no address space.
  static int ii; // ii is in global address space.
  int * ptr; // ptr points to generic address space int.
  int & ref = ...; // ref references int in generic address space.
};

template <int N>
void bar()
{
  S<N> s; // s is in __private address space.
}
```

TODO: Example for type alias and decltype!

2.2.3. References

Reference types can be qualified with an address space.

```
__private int & ref = ...; // references int in __private address space.
```

By default references will refer to generic address space objects, except for dependent types that are not template specializations (see *Deduction*). Address space compatibility checks are performed when references are bound to values. The logic follows the rules from address space pointer conversion (`OpenCL v2.0 s6.5.5`).

2.2.4. Default address space

All non-static member functions take an implicit object parameter `this` that is a pointer type. By default the `this` pointer parameter is in the generic address space. All concrete objects passed as an argument to the implicit `this` parameter will be converted to the generic address space first if such conversion is valid. Therefore programs using objects in the `constant address space will not be compiled unless the address space is explicitly specified using address space qualifiers on member functions (see Member function qualifier) as the conversion between constant` and generic is disallowed. Member function qualifiers can also be used in case conversion to the generic address space is undesirable (even if it is legal). For example, a method can be implemented to exploit memory access coalescing for segments with memory bank. This not only applies to regular member functions but to constructors and destructors too.

2.2.5. Member function qualifier

C++ for OpenCL allows specifying an address space qualifier on member functions to signal that they are to be used with objects constructed in a specific address space. This works just the same as qualifying member functions with `const` or any other qualifiers. The overloading resolution will select the candidate with the most specific address space if multiple candidates are provided. If there is no conversion to an address space among candidates, compilation will fail with a diagnostic.

```
struct C {
  void foo() __local;
  void foo();
};

__kernel void bar() {
  __local C c1;
  C c2;
  __constant C c3;
  c1.foo(); // will resolve to the first foo.
  c2.foo(); // will resolve to the second foo.
  c3.foo(); // error due to mismatching address spaces - can't convert to
            // __local or generic.
}
```

2.2.6. Implicit special members

All implicit special members (default, copy or move constructor, copy or move assignment, destructor) will be generated with the generic address space.

```
class C {
  // Has the following implicitly defined member functions
  // void C() /*__generic*/;
  // void C(const /*__generic*/ C &) /*__generic*/;
  // void C(/*__generic*/ C &&) /*__generic*/;
  // operator= '/*__generic*/ C &(/*__generic*/ C &&)';
  // operator= '/*__generic*/ C &(const /*__generic*/ C &) /*__generic*/;
}
```

2.2.7. Builtin operators

All builtin operators are available in the specific address spaces, thus no conversion to generic address space is performed.

2.2.8. Templates

There is no deduction of address spaces in non-pointer/non-reference template parameters and dependent types (see *Deduction*). The address space of a template parameter is deduced during type deduction if it is not explicitly provided in the instantiation.

```
 1 template<typename T>
 2 void foo(T* i){
 3   T var;
 4 }
 5
 6 __global int g;
 7 void bar(){
 8   foo(&g); // error: template instantiation failed as function scope variable
 9            // appears to be declared in __global address space (see line 3).
10 }
```

It is not legal to specify multiple different address spaces between template definition and instantiation. If multiple different address spaces are specified in a template definition and instantiation, compilation of such a program will fail with a diagnostic.

```
template <typename T>
void foo() {
  __private T var;
}

void bar() {
  foo<__global int>(); // error: conflicting address space qualifiers are provided
                       // __global and __private.
}
```

Once a template has been instantiated, regular restrictions for address spaces will apply.

```
template<typename T>
void foo(){
  T var;
}

void bar(){
  foo<__global int>(); // error: function scope variable cannot be declared in
                       // __global address space.
}
```

2.2.9. Temporary materialization

All temporaries are materialized in the __private address space. If a reference with another address space is bound to them, a conversion will be generated in case it is valid, otherwise compilation will fail with a diagnostic.

```
int bar(const unsigned int &i);

void foo() {
  bar(1); // temporary is created in __private address space but converted
          // to generic address space of parameter reference.
}

__global const int& f(__global float &ref) {
  return ref; // error: address space mismatch between temporary object
              // created to hold value converted float->int and return
              // value type (can't convert from __private to __global).
}
```

2.2.10. Initialization of local and constant address space objects

TODO

Acknowledgements

The C++ for OpenCL documentation is the result of the contributions of many people. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

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