The C++ for OpenCL 1.0 Programming Language Documentation

Khronos® OpenCL Working Group

Wed, 17 Nov 2021 18:07:41 +0000: from git branch: commit: f4f55be2c3ccb35944e5d3d706fde15f31ed45c7

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 C++ and OpenCL C is inherited. Since both OpenCL C and C++ are derived from C and moreover C++ is almost fully backward compatible with C, the main design principle of C++ for OpenCL is to reapply existing OpenCL concepts to C++. Therefore, it is important to refer to The OpenCL Specification, Version 2.0 `section 3.2` and `section 3.3` detailing fundamental differences of OpenCL execution and memory models from the conventional C and C++ view.

This document describes the programming language in details. It is not structured as a standalone document, but rather as an addition to OpenCL C v2.0 defined in The OpenCL C Specification, Version 2.0 and C++17 defined in The C++17 Specification. Where necessary this document refers to the specifications of those languages accordingly. A full understanding of C++ for OpenCL requires familiarity with the specifications or other documentation of both languages that C++ for OpenCL is built upon.

The description of C++ for OpenCL starts from highlighting *the differences to OpenCL C* and *the differences to C++*.

The majority of content covers the behavior that is not documented in the OpenCL C v2.0 `section 6` 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 programming language inherits features from `OpenCL C v2.0, s6` as well as C++17. Detailed aspects of OpenCL and C++ are not described in this document as they can be found in their official specifications.

This section documents various language features of C++ for OpenCL that are not covered in either OpenCL or C++ specifications, in particular:

- any behavior that deviates from C++17;
- any behavior that deviates from OpenCL C v2.0;
- any behavior that is not governed by OpenCL C and C++.

2.1. Difference to C++

C++ for OpenCL supports the majority of standard C++17 features, however, there are some differences that are documented in this section.

2.1.1. Restrictions to C++ features

The following C++ language features are not supported:

- Virtual functions (C++17 `[class.virtual]`);
- References to functions (C++17 `[class.mfct]`);
- Pointers to class member functions (in addition to the regular non-member functions that are already restricted in OpenCL C);
- Exceptions (C++17 `[except]`);
- `dynamic_cast` operator (C++17 `[expr.dynamic.cast]`);
- Non-placement `new`/`delete` operators (C++17 `[expr.new]`/`[expr.delete]`);
- `thread_local` storage class specifier (C++17 `[basic.stc.thread]`);
- Standard C++ libraries (C++17 `[library]`).

Simultaneous initialization of static local objects performed by different work-items is not guaranteed to be free from race-conditions. Whether an implementation provides such a guarantee is indicated by the presence of the `__cpp_threadsafe_static_init` feature test macro[1].

The list above only contains extra restrictions that are not detailed in OpenCL C. As OpenCL restricts a number of C features, the same restrictions are inherited by C++ for OpenCL. The detailed list of C feature restrictions is provided in `OpenCL C v2.0 section 6.9`.

2.2. Difference to OpenCL C

C++ for OpenCL provides backwards compatibility with OpenCL C for the majority of features. However, there are a number of exceptions that are described in this section. Some of them come from the nature of C++ but others are due to improvements in OpenCL features. Most of such improvements do not invalidate old code, but simply provide extra functionality.

2.2.1. C++ related differences

C++ for OpenCL is a different language to OpenCL C and it is derived from C++ inheriting C++'s fundamental design principles. Hence C++ for OpenCL deviates from OpenCL C in the same areas where C++ deviates from C. This results in a more helpful language for developers and facilitates improvements in compilation tools without substantially increasing their complexity.

2.2.1.1. Implicit conversions

C++ is much stricter about conversions between types, especially those that are performed implicitly by the compiler. For example it is not possible to convert a `const` object to non-`const` implicitly. For details please refer to C++17 `[conv]`.

```
void foo(){
  const int *ptrconst;
  int *ptr = ptrconst; // invalid initialization discards const qualifier
}
```

The same applies to narrowing conversions in initialization lists (C++17 `[dcl.init.list]`).

```
struct mytype {
 int i;
};
void foo(uint par){
  mytype var = {
    .i = par // narrowing from uint to int is disallowed
  };
}
```

Some compilers allow silencing this error using a flag (e.g. in Clang `-Wno-error=c++11-narrowing` can be used).

Among other common conversions that will not be compiled in C++ mode there are pointer to integer or integer to pointer type conversions.

```
void foo(){
  int *ptr;
  int i = ptr; // incompatible pointer to integer conversion
}
```

2.2.1.2. Null pointer constant

In C and OpenCL C the null pointer constant is defined using other language features as it is not represented explicitly i.e. commonly it is defined as

```
#define NULL ((void*)0)
```

In C++17 there is an explicit builtin pointer literal `nullptr` that should be used instead (C++17 `[lex.nullptr]`).

`NULL` macro definition in C++ for OpenCL follows C++17 `[support.types.nullptr]` where it is an implementation defined macro and it is not guaranteed to be the same as in OpenCL C. Reusing the definition of `NULL` from OpenCL C does not guarantee that any code with NULL is legal in C++ for OpenCL even if it is legal in OpenCL C.

```
#define NULL ((void*)0)
void foo(){
 int *ptr = NULL; // invalid initialization of int* with void*
}
```

To improve code portability and compatibility, implementations are encouraged to define `NULL` as an alias to pointer literal `nullptr`.

2.2.1.3. Use of restrict

C++17 does not support `restrict` and therefore C++ for OpenCL can not support it either. Some compilers might provide extensions with some functionality of `restrict` in C++, e.g. `__restrict` in Clang.

This feature only affects optimizations and the source code can be modified by removing it. As a workaround to avoid manual modifications, macro substitutions can be used to either remove the keyword during the preprocessing by defining `restrict` as an empty macro or mapping it to another similar compiler features, e.g. `__restrict` in Clang. This can be done in headers or using `-D` compilation flag.

2.2.1.4. Limitations of goto statements

C++ is more restrictive with respect to entering the scope of variables than C. It is not possible to jump forward over a variable declaration statement apart from some exceptions detailed in C++17 `[stmt.dcl]`.

```
if (cond)
  goto label;
int n = foo();
label:  // invalid: jumping forward over declaration of n
   // ...
```

2.2.1.5. Ternary Selection Operator

The ternary selection operator (`?:`) inherits its behaviour from both C++ and OpenCL C. It operates on three expressions `(exp1 ? exp2 : exp3)`. If all three expressions are scalar values, the C++17 rules for ternary operator are followed. If the result is a vector value, then this is equivalent to calling `select(exp3, exp2, exp1)` as described in `OpenCL C v2.0 s6.13.6`. The rules from OpenCL C impose limitation that `exp1` cannot be a vector of float values. However, `exp1` can be evaluated to a scalar float as it is contextually convertible to bool in C++.

2.2.2. OpenCL specific difference

This section describes where C++ for OpenCL differs from OpenCL C in OpenCL specific behavior.

2.2.2.1. Variadic macros

C++ for OpenCL eliminates the restriction on variadic macros from `OpenCL C v2.0 s6.9.e`. Variadic macros can be used normally as per C++17 `[cpp.replace]`.

2.2.2.2. Predefined macros

The macro `__OPENCL_C_VERSION__` described in `OpenCL C v2.0 s6.10`, is not defined.

The following new predefined macros are added in C++ for OpenCL:

- `__OPENCL_CPP_VERSION__` set to value `100`.
- `__CL_CPP_VERSION_1_0__` also set to `100` and can be used for convenience instead of a literal.

2.2.2.3. Atomic operations

C++ for OpenCL relaxes restriction from `OpenCL C v2.0 s6.13.11` to atomic types allowing them to be used by builtin operators, and not only by builtin functions.

Operators on atomic types behave as described in C++17 sections `[atomics.types.int]` `[atomics.types.pointer]` `[atomics.types.float]`.

```
atomic_int acnt;
acnt++; // equivalent to atomic_fetch_add(&acnt, 1);
```

2.2.2.4. Use of Blocks

Blocks that are defined in `OpenCL C v2.0 s6.12` are not supported and their use can be replaced by lambdas (C++17 `[expr.prim.lambda]`).

The above implies that builtin functions using blocks, such as `enqueue_kernel`, are not supported in C++ for OpenCL.

2.3. 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 C++17 `[conv.qual]` but the compatibility is determined using notation of sets and overlapping of address spaces from `section 5.1.3` of The Embedded C Specification. 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.

2.3.1. Casts

C-style casts follow rules of `OpenCL C v2.0 s6.5.5`. Conversions of pointers and references to the generic address space can be done by any C++ cast operator (as an implicit conversion); converting from generic to named address space can only be done using the dedicated `addrspace_cast` operator. The `addrspace_cast` operator can only convert between address spaces for pointers and references and no other conversions are allowed to occur. Note that conversions between `__constant` and any other other address space are disallowed.

```
int * genptr;
__private float * ptrfloat = reinterpret_cast<__private float*>(genptr); // illegal.
__private float * ptrfloat = addrspace_cast<__private float*>(genptr); // illegal.
__private int * ptr = addrspace_cast<__private int*>(genptr); // legal.
float * genptrfloat = reinterpret_cast<float*>(ptr); // legal.
__constant int * constptr = addrspace_cast<__constant int*>(genptr); // illegal.
```

2.3.2. References

Reference types can be qualified with an address space.

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

By default references 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`).

2.3.3. Deduction & Default address space

This section details what happens if address spaces for types are not provided in the source code explicitly. Most of the logic for address space deduction (i.e. default address space) follows rules from `OpenCL v2.0 s6.5`.

References inherit rules from pointers and therefore refer to generic address space objects by default (see *References*).

Class static data members are deduced to `__global` 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.

Address spaces are not deduced for:

- non-pointer/non-reference template parameters except for template specializations or non-type type based template parameters.
- non-pointer/non-reference class members except for static data members that are deduced to `__global` address space.
- non-pointer/non-reference type 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 int in generic address space.
  int & ref = ...; // 'ref' references int in generic address space.
};

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

```
struct c1 {};
using alias_c1 = c1; // 'alias_c1' is 'c1'.
using alias_c1_ptr = c1 *; // 'alias_c1_ptr' is a generic address space pointer to
'c1'.
```

```
__kernel void foo()
{
  __local int i;
  decltype(i)* ii; // type of 'ii' is '__local int *__private'.
}
```

For the placeholder type specifier `auto` an address space of the outer type is deduced as if it would be any other regular type. However if `auto` is used in a reference or pointer type, the address space of a pointee is taken from the type of the initialization expression. The logic follows rules for `const` and `volatile` qualifiers.

```
__kernel void foo()
{
  __local int i;
  constexpr int c = 1;

  __constant auto cai = c; // type of 'cai' is '__constant int' (no deduction).

  auto aii = cai; // type of 'aii' is '__private int' (regular deduction).

  auto *ptr = &i; // type of 'ptr' is '__local int * __private'
                  // (addr space of a pointer is deduced regularly,
                  // addr space of its pointee is taken from 'i').

  auto *&refptr = ptr; // type of 'refptr' is '__local int * generic & __private'
                       // (addr space of a reference and type of referencing object
                       // is deduced regularly,
                       // addr space of a pointee is taken from the pointee of 'ptr').
}
```

2.3.4. 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;
  __private 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 addr spaces.
}
```

2.3.5. Lambda function

The address space qualifier can be optionally added for lambda expressions after the attributes. Similar to method qualifiers, they will alter the default address space of lambda call operator that has generic address space by default.

```
__kernel void foo() {
  auto priv1 = []() __private {};
  priv1();
  auto priv2 = []() __global {};
  priv2(); // error: lambda object and its expression have mismatching address space.
  __constant auto const3 = []() __constant{};
  const3();

  [&] () __global {} (); // error: lambda temporary is in __private address space.

  [&] () mutable __private {} ();
  [&] () __private mutable {} (); // error: mutable specifier should precede address
space.
}
```

2.3.6. Implicit special members

The prototype for implicit special members (default, copy or move constructor, copy or move assignment, destructor) has the generic address space for an implicit object pointer and reference parameters (see also *Member function qualifier*).

```
class C {
  // Has the following implicitly defined member functions.

  // void C(); /* implicit 'this' parameter is a pointer to */
               /* object in generic address space. */

  // void C(const C & par); /* 'this'/'par' is a pointer/reference to */
                            /*  object in generic address space. */

  // void C(C && par); /* 'this'/'par' is a pointer/r-val reference to */
                       /* object in generic address space. */

  // operator= C &(const C & par); /* 'this'/'par'/return value is */
                                   /* a pointer/reference/reference to */
                                   /* object in generic address space. */

  // operator= C &(C && par)'; /* 'this'/'par'/return value is */
                               /* a pointer/r-val reference/reference to */
                               /* object in generic address space. */
};
```

2.3.7. Builtin operators

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

2.3.8. 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.

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

 __global int g;
void bar(){
  foo(&g); // error: template instantiation failed as function scope variable 'var'
           // appears to be declared in __global address space (see line 3).
}
```

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
                       // for 'var', '__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 'var' cannot be declared
                       // in '__global' address space.
}
```

2.3.9. Temporary materialization

All temporaries are materialized in `__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).
}
```

2.3.10. Construction, initialization and destruction

Construction, initialization and destruction of objects in `__private` and `__global` address space follow the general principles of C++. For program scope objects or static objects in the function scope with non-trivial constructors and destructors, the implementation defines an ABI format for runtime initialization and destruction of global objects before/after all kernels are enqueued.

Non-trivial destructors for global objects are not required to be supported by all implementations. The macro `__opencl_cpp_global_destructor`, which is defined if and only if such destructors are supported by the implementation, can be used to check whether this functionality is available when compiling kernel code.

Objects in `__local` address space can not have initializers in declarations and therefore a constructor can not be called. All objects created in the local address space have undefined state at the point of their declaration. Developers are free to define a special member function that can initialize local address space objects after their declaration. Any default values provided for the initialization of members in a class declaration are ignored when creating the local address space objects. Since the initialization is performed after the variable declaration, special handling is required for classes with data members that are references because their values can not be overwritten trivially. Destructors of local address space objects are not invoked automatically. They can be called manually.

```
class C {
  int m;
};

kernel void foo() {
  __local C locobj{}; // error: local address space objects can't be initialized
  __local C locobj; // uninitialised object
  locobj = {}; // calling copy assignment operator is allowed
  locobj.~C(); // local address space object destructors are not invoked
automatically.
}
```

User defined constructors are not allowed to construct objects in `__constant` address space. Such objects can be initialized using literals and initialization lists if they do not require any user defined conversions.

Objects in `__constant` address space can be initialized using:

- Literal expressions;
- Uniform initialization syntax `{}`;
- Using implicit constructors.

```
struct C1 {
  int m;
};

struct C2 {
  int m;
  C2(int init) __constant {};
};

kernel void k() {
  __constant C1 cobj1 = {1};
  __constant C1 cobj2 = C1();
  __constant C2 cobj3 = {1}; // error: user defined constructor can't be used
}
```

2.3.11. Nested pointers

C++ for OpenCL does not allow implicit address space conversions in nested pointers even with compatible address spaces. The following rules apply when converting between address spaces in nested pointers:

- Implicit conversions of address spaces in nested pointers are disallowed.
- Any address space conversion in nested pointers with safe casts (e.g. `const_cast`, `static_cast`, `addrspace_cast`) is disallowed.
- Any address space conversion in nested pointers can be done using low level C-style or `reinterpret_cast`. No compatibility check is performed for address spaces in nested pointers.

```
local int * * locgenptr;
constant int * * cnstgenptr;
int * * gengenptr;
gengenptr = const_cast<int**>(locgenptr); // illegal.
gengenptr = static_cast<int**>(cnstgenptr); // illegal.
gengenptr = addrspace_cast<int**>(cnstgenptr); // illegal.
gengenptr = reinterpret_cast<int**>(locgenptr); // legal.
gengenptr = reinterpret_cast<int**>(cnstgenptr); // legal.
```

2.4. C++ casts

C++ has 3 casts in addition to C-style casts. `static_cast` and `const_cast` function the same way as in C++, but `reinterpret_cast` has some additional functionality:

- Conversion between vectors and scalars are allowed.
- Conversion between OpenCL types are disallowed.

2.4.1. Vectors and scalars

`reinterpret_cast` reinterprets between integral types like integers and pointers. In C++ for openCL this also includes vector types, and so using `reinterpret_cast` between vectors and scalars is also possible, as long as the size of the vectors are the same.

```
int i;
short2 s2 = reinterpret_cast<short2>(i); // legal.
int2 i2 = reinterpret_cast<int2>(i); // illegal.

short8 s8;
int4 i4 = reinterpret_cast<int4>(s8); // legal.
long l4 = reinterpret_cast<long>(s8); // illegal.
```

2.4.2. OpenCL types

Some of the OpenCL types are the same size as integers, or can be implemented as integers, but since they are not conceptually integral, they can not be used with `reinterpret_cast`. Therefore these are all illegal conversions:

```
reserve_id_t id;
reserve_id_t id2 = reinterpret_cast<reserve_id_t>(id); // illegal.
int i = reinterpret_cast<int>(r); // illegal.
long l = reinterpret_cast<long>(r); // illegal.
```

2.5. Kernel functions

Kernel functions have implicit C linkage (C++17 `[dcl.link]`) which means that C++ specific features are not supported. Therefore, the kernel functions:

- Can not be class members (C++17 `[class.mfct]`);
- Can not be overloaded (C++17 `[over]`);
- Can not be function templates (C++17 `[temp.fct]`).

Moreover the types used in parameters of the kernel functions must be:

- Trivial and standard-layout types C++17 `[basic.types]` (plain old data types) for parameters passed by value;
- Standard layout types for pointer parameters. The same applies to references[2] if an implementation supports them in kernel parameters.

These are additional restrictions to the list detailed in `OpenCL C v2.0 section 6.9`.

[1] The macro belongs to the list of C++20's feature test macros.

[2] Whether C++ features (e.g references) can be used in functions with C linkage is implementation-defined (C++17 [dcl.link]).

Chapter 3. Normative References

1. "The OpenCL Specification, Version 2.0", https://www.khronos.org/registry/OpenCL/.
2. "The OpenCL C Specification, Version 2.0", https://www.khronos.org/registry/OpenCL/.
3. "ISO/IEC 14882:2017 - Programming languages — C++", https://www.iso.org/standard/68564.html. References are to sections of this specific version, referred to as the "The C++17 Specification", although other versions exist.
4. "ISO/IEC TR 18037:2008 Programming languages - C - Extensions to support embedded processors", https://www.iso.org/standard/51126.html. References are to sections of this specific version, referred to as the "The Embedded C Specification", although other versions exist.

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
- Stuart Brady, Arm
- Ole Strøhm, Arm
- Justas Janickas, Arm