

**AMD Accelerated
Parallel Processing**
TECHNOLOGY

OpenCL Static C++ Kernel Language Extension

Document Revision: 04

Advanced Micro Devices

Authors:

Ofer Rosenberg, Benedict R. Gaster, Bixia Zheng, Irina Lipov

December 15, 2011

Contents

1	Overview	3
1.1	Supported Features.....	3
1.2	Unsupported Features	3
1.3	Relations with ISO/IEC C++	4
2	Additions and Changes to Section 6 - The OpenCL C Programming Language	5
2.1	Additions and Changes to Section 5.7.1 - Creating Kernel Objects	5
2.2	Passing Classes between Host and Device	5
3	Additions and Changes to Section 6 - The OpenCL C Programming Language	6
3.1	Building C++ Kernels.....	6
3.2	Classes and derived classes	6
3.3	Namespaces.....	7
3.4	Overloading	7
3.5	Templates	8
3.6	Exceptions.....	8
3.7	Libraries.....	9
3.8	Dynamic operation.....	9
4	Examples.....	10
4.1	Passing a class from the Host to the Device	10
4.2	Kernel overloading.....	11
4.3	Kernel Template.....	11
5	References.....	12

1 Overview

This extension defines OpenCL Static C++ kernel language, which is a form of the ISO/IEC Programming languages C++ specification [1]. The language supports some key features such as overloading and templates that can be resolved at compile time (hence static), while restricting the use of language features which requires dynamic/runtime resolving. At the same time, the language is extended to support most of the extended features described in Section 6 of OpenCL spec: new data types (vectors, images, samples, etc.), OpenCL Built-in functions and more.

1.1 Supported Features

The following list contains the major C++ features which are supported by this extension:

- Kernel and function overloading
- Inheritance
 - Strict inheritance
 - Friend classes
 - Multiple inheritance
- Templates:
 - Kernel templates
 - Member templates
 - Template default argument
 - Limited class templates (the “virtual” keyword is not exposed)
 - Partial template specialization
- Namespaces
- References
- ‘this’ operator

Note that supporting templates and overloading highly improves the efficiency of writing code, as it allows developers to avoid replication of code when not necessary.

Using kernel template & kernel overloading requires support from the Runtime API as well. AMD provides a simple extension to `clCreateKernel` which enables the user to specify the desired kernel.

1.2 Unsupported Features

The following list contains the major C++ features which are not supported by this extension:

- Virtual functions, i.e. methods marked with the `virtual` keyword.
- Abstract classes, i.e. a class defined only of pure virtual functions.
- Dynamic memory allocation, i.e. non-placement `new/delete` support is not provided.
- Exceptions, i.e. there is no support for `throw/catch`.
- The `::` operator
- STL and other standard C++ libraries are not supported.

The language specified in this extension can be easily expanded to support these features.

1.3 Relations with ISO/IEC C++

This extension focuses mainly on documenting the differences between the OpenCL Static C++ kernel language and the ISO/IEC Programming languages C++ specification [1]. Where possible, this extension leaves technical definitions to the ISO/IEC specification. The main motivation for this is simply that the C++ specification is a large and complicated definition and there seems little benefit in replicating it here.

2 Additions and Changes to Section 6 - The OpenCL C Programming Language

2.1 Additions and Changes to Section 5.7.1 - Creating Kernel Objects

In the static C++ kernel language, a kernel can be overloaded, templated, or both overloaded and templated. The syntax which explains how to do it is defined in chapters 3.4 & 3.5.

In order to support these cases, we added the following error codes that can be returned by *clCreateKernel*:

- `CL_INVALID_KERNEL_TEMPLATE_TYPE_ARGUMENT_AMD` if a kernel template argument is not a valid type, i.e. is neither a valid OpenCL C type or a user defined type in the same source file.
- `CL_INVALID_KERNEL_TYPE_ARGUMENT_AMD` if a kernel type argument, used for overloading resolution, is not valid a valid type, i.e. as above is neither a valid OpenCL C type or user defined type in the same source program.

2.2 Passing Classes between Host and Device

The extension allows a developer to pass Classes between the Host and the Device. The mechanism that is used to pass the class to the device and back is the existing buffer object APIs. The Class which is passed maintains its state (public and private members), and the Compiler implicitly changes the Class to use either the Host-side or Device-side methods.

On the Host side, the Application creates the Class and an equivalent memory object with the same size (using the `sizeof` function). It can then use the class methods to set or change values of the class members. When the class is ready, the Application uses a standard buffer API to move the class to the device – either `Unmap` or `Write`, then sets the buffer object as the adequate kernel argument and enqueues the kernel for execution. When the kernel finished the execution, the Application can map back (or read) the buffer object into the Class, and continue working on it.

3 Additions and Changes to Section 6 - The OpenCL C Programming Language

3.1 Building C++ Kernels

In order to compile a program which contains C++ kernels and functions, the Application must add following compile option to `clBuildProgramWithSource`

```
-x language
```

where `language` is defined as one of the following:

- `clc` – in this case the source language is considered to be OpenCL C as defined in the “The OpenCL Programming Language version 1.1” [2].
- `clc++` - in this case the source language is considered to be OpenCL C++ as defined in the following sections of the this document.

3.2 Classes and derived classes

OpenCL C is extended to support classes and derived classes as per Sections 9 and 10 [1], with the limitation that virtual functions and abstracts classes are not supported. The `virtual` keyword is reserved and the OpenCL C++ compiler is required to report a compile time error if it is used in the input program.

This limitation restricts class definitions to be fully statically defined. There is nothing prohibiting a future version of OpenCL C++ from relaxing this restriction, pending performance implications.

A class definition should not contain any address space qualifier, either for members or for methods:

```
class myClass{
    public:
        int myMethod1(){ return x;}
        void __local myMethod2(){x = 0;} // illegal
    private:
        int x;
        __local y; // illegal
};
```

The class invocation inside a kernel, however, can be either in private or local address space:

```
__kernel void myKernel()
{
    myClass c1;
    __local myClass c2;
    ...
}
```

Classes can be passed as arguments to kernels, by defining a buffer object at the size of the class, and using it. The device will invoke the adequate device-specific methods, and will access the class members passed from the host.

OpenCL C kernels (defined with `__kernel`) may not be applied to a class constructor, destructor, or method, except in the case that the class method is defined static and thus does not require object construction to be invoked.

3.3 Namespaces

Namespaces are support without change as per [1].

3.4 Overloading

As defined in [1] when two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types are called overloaded declarations. Only kernel and function declarations can be overloaded; object and type declarations cannot be overloaded.

As per [1] there are a number of restrictions to how functions can be overloaded and these are defined formally in Section 13 [1]; we note here that kernels and functions cannot be overloaded by return type. Also the rules for well-formed programs as defined by Section 13 [1] are lifted to apply to both kernel and function declarations.

Overloading resolution is per [1], Section 13.1, but extended to account for vector types. The algorithm for "best viable function", Section 13.3.3 [1], is extended for vector types by inducing a partial-ordering as a function of the partial-ordering of its elements. Following the existing rules for vector types in OpenCL 1.1 [2], explicit conversion between vectors is not allowed. (This reduces the number of possible overloaded functions with respect to vectors but we do not expect this to be a particular burden to developers as explicit conversion can always be applied at the point of function evocation).

For overloaded kernels, the following syntax is used as part of the kernel name:

```
foo(type1, ..., typen)
```

where `type1, ..., typen` must be either OpenCL scalar or vector type, or can be a user defined type that is allocated in the same source file as the kernel `foo`.

In order to allow overloaded kernels, the user should use the following syntax:

```
__attribute__((mangled_name(myMangledName)))
```

The kernel mangled name is used as a parameter to passed to `clCreateKernel()` API. This mechanism is needed to allow overloaded kernels without changing the existing OpenCL kernel creation API.

3.5 Templates

OpenCL C++ provides support, without restriction, for C++ templates as defined in Section 14 [1]. The arguments to templates are extended to allow for all OpenCL base types, including vectors and pointers qualified with OpenCL C address spaces, i.e. `__global`, `__local`, `__private`, and `__constant`.

OpenCL C++ kernels (defined with `__kernel`) can be templated and can be called from within an OpenCL C (C++) program or as an external entry point, i.e from the host.

For kernel templates the following syntax is used as part of the kernel name (assuming a kernel called *foo*):

```
foo<type1, . . . , typen>
```

where `type1, . . . , typen` must be either OpenCL scalar or vector type, or can be a user defined type that is allocated in the same source file as the kernel *foo*.

In the case a kernel is both overloaded and templated:

```
foo<type1, . . . , typen>(typen+1, . . . , typem)
```

Note that in this case overloading resolution is done by first matching non-templated arguments in order they appear in the definition and then substituting template parameters. This allows for the intermixing of template and non-template arguments in the signature.

In order to support template kernels, the same mechanism for kernel overloading is used. The user should use the following syntax:

```
__attribute__((mangled_name(myMangledName)))
```

The kernel mangled name is used as a parameter to passed to `clCreateKernel()` API. This mechanism is needed to allow template kernels without changing the existing OpenCL kernel creation API. An implementation is not required to detect name collision with user specified kernel mangled names involved.

3.6 Exceptions

Support for exceptions, as per Section 15 [1] is not support. The keywords:

- `try`
- `catch`
- `throw`

are reserved and it is required that the OpenCL C++ compiler produce a static compile time error if they are used in the input program.

3.7 Libraries

Support for the general utilities library as defined in Sections 20-21 [1] is not provided. The standard C++ libraries and STL library are not supported.

3.8 Dynamic operation

Features related to dynamic operation are not supported:

- “virtual” modifier not allowed
 - OpenCL C++ prohibits the use of “virtual” modifier. As a result, virtual member functions and virtual inheritance are not supported.
- `Dynamic_cast` that requires runtime check is not allowed
- Dynamic storage allocation and deallocation

4 Examples

4.1 Passing a class from the Host to the Device

Here's an example for passing a Class from the Host to the device and back.

The class definition needs to be the same on the Host code and Device code, besides the members' type in the case of vectors. If the class includes vector data types, the definition needs to be according to the table which appears on Section 6.1.2 (Corresponding API type for OpenCL Language types)

Kernel code:

```
Class Test
{
    setX (int value);
    private:
    int x;
}

__kernel foo (__global Test* InClass, ...)
{
    If (get_global_id(0) == 0)
        InClass->setX(5);
}
```

Host Code:

```
Class Test
{
    setX (int value);
    private:
    int x;
}

MyFunc ()
{
    tempClass = new(Test);
    ... // Some OpenCL startup code - create context, queue, etc.
    cl_mem classObj = clCreateBuffer(context, CL_USE_HOST_PTR,
                                    sizeof(Test), &tempClass, event);
    clEnqueueMapBuffer(..., classObj, ...);
    tempClass.setX(10);
    clEnqueueUnmapBuffer(..., classObj, ...); //class is passed to the Device
    clEnqueueNDRange(..., fooKernel, ...);
    clEnqueueMapBuffer(..., classObj, ...); //class is passed back to the Host
}
```

4.2 Kernel overloading

Here's an example for defining and used mangled name for kernel overloading, and choosing the right kernel rom the host code. Assume the following kernels are defined:

```
__attribute__((mangled_name(testAddFloat4))) kernel void
testAdd(global float4 * src1, global float4 * src2, global float4 * dst)
{
    int tid = get_global_id(0);
    dst[tid] = src1[tid] + src2[tid];
}

__attribute__((mangled_name(testAddInt8))) kernel void
testAdd(global int8 * src1, global int8 * src2, global int8 * dst)
{
    int tid = get_global_id(0);
    dst[tid] = src1[tid] + src2[tid];
}
```

The names “testAddFloat4” and “testAddInt8” are the external names for the two kernel instants. When calling `clCreateKernel`, passing one of these kernel names will lead to the right overloaded kernel.

4.3 Kernel Template

The following example defines a kernel template, `testAdd`. It also defines two explicit instants of the kernel template, `testAddFloat4` and `testAddInt8`. The names “testAddFloat4” and “testAddInt8” are the external names for the two kernel template instants that must be used as parameters when calling to `clCreateKernel` API.

```
template <class T>
kernel void testAdd(global T * src1, global T * src2, global T * dst)
{
    int tid = get_global_id(0);
    dst[tid] = src1[tid] + src2[tid];
}

template __attribute__((mangled_name(testAddFloat4))) kernel void
testAdd(global float4 * src1, global float4 * src2, global float4 * dst);

template __attribute__((mangled_name(testAddInt8))) kernel void
testAdd(global int8 * src1, global int8 * src2, global int8 * dst);
```

5 References

[1] Programming languages C++. International Standard ISO/IEC 14881, 1998.

[2] The OpenCL Programming Language 1.2. Rev15 Khronos 2011.