The Generic Address Space in OpenCL™ 2.0

By Adam Lake (Intel) (https://software.intel.com/en-us/user/334331), Robert I. (Intel) (https://software.intel.com/en-us/user/414915),

Updated February 6, 2015

Translate

- Introduction
- What is the Generic Address Space?
- Enabling the Generic Address Space
- Why Would I Want to Use the Generic Address Space?
- Performing Some Operations in a Specific Address Space
- Address Space Casting
- Performance Implications and How to Address Them
- A Working Example
- Future Work
- Acknowledgements
- About the Authors
- References
- Legal Information
- Download the Sample

Introduction

One of the new features of OpenCL 2.0 is the *generic address space*. Prior to OpenCL 2.0 the programmer had to specify an address space of what a pointer points to when that pointer was declared or the pointer was passed as an argument to a function. In OpenCL 2.0 the pointer itself remains in the private address space, but what the pointer points has changed its default to be *generic* meaning it can point to any of the named address spaces within the generic address space. This features requires you to set a flag to turn it on, so OpenCL C 1.2 programs will continue to compile with no changes.

To demonstrate this we show a brief of the new syntax on a function declaration and a variable declaration in the function. In OpenCL 1.2 we may have written this:

```
void foo(global unsigned int *bar)  // ‘global’ address space on bar,
works in both OCL 1.2 and OCL 2.0 with no additional flags to compile
{
    local unsigned int *temp = NULL;//’local’ address space on temp,
works in both OCL 1.2 and OCL 2.0 with no additional flags to compile
}
```

In OpenCL 2.0 the following code will work on pointers that point to the global or the local or private address spaces:

```
void foo(unsigned int *bar)  // OCL 2.0, no address space on bar
{
    unsigned int *temp = NULL;//OCL 2.0, no address space on temp
}
```

Remember to enable OpenCL 2.0 features by passing the flag “–cl-std=CL2.0” in the options of clBuildProgram() or clCompileProgram(); Otherwise, you would see the following error:

```
1:54:24: error: passing '__local unsigned int *' to parameter of type
```

```
'unsigned int *' changes address space of pointer
```

since OpenCL 2.0 defaults to compile programs as if they are OpenCL 1.2 programs by default for backwards compatibility.

What is the Generic Address Space?

The *generic* address space is an abstract address space that encapsulates the *local*, *global*, and *private* address spaces. For practical reasons it does not encompass the *constant* address space. The generic address space is inspired by the generic address space of the Embedded C Specification:

"In addition to the generic address space, an implementation may support other, named address spaces. Objects may be allocated in these alternate address spaces, and pointers may be defined that point to objects in these address spaces. It is intended that pointers into an address space only need be large enough to support the range of addresses in that address space."

On some architectures the pointers to the address spaces are of different sizes and may be different memory banks, so we don't want to get rid of the existing named address spaces. However, we do want a means to write programs that don't require specialization of functions when it isn't needed. Some of these use cases are documented below. For performance reasons we want to continue to support specialization to the respective address spaces but also enable programmers to write a single segment of code that will operate on different address spaces.

A few important points about the generic address space:

- Parameters to kernels called from the host must continue to be qualified with an address space.
- Function and kernel parameters remain in the private address space, it is their *points to* address that has changed. Read this three times, it is important and subtle! Many examples are in Chapter 6.5 of the OpenCL C 2.0 Specification, see the References.
- Null pointers from two different address spaces will evaluate to be equal as long as one of those pointers is generic.
- If a null pointer is converted from one address space to another the null pointer will now be a null pointer of that type.
- The address spaces are considered disjoint in the abstract sense but some implementations may treat them as if they are overlapping or part of the same physical memory. This is perfectly reasonable.
- global, local, private and constant can be used interchangeably with __global, __local, __private and __constant respectively. Generic is an unnamed address space and as such has no keyword in OpenCL 2.0.

Enabling the Generic Address Space

As stated in the introduction, to enable the OpenCL C 2.0 generic address space feature, the flag "-cl-std=CL2.0" must be passed to clBuildProgram() or clCompileProgram(). Otherwise, the program will continue to compile in OpenCL 2.0 as an OpenCL 1.2 program. This is to make it easier to move to the new OpenCL 1.2 runtime yet have older programs 'just work', migrating to the new OpenCL 2.0 features like shared virtual memory and generic address space incrementally.

Why Would I Want to Use the Generic Address Space?

The generic address space makes writing OpenCL programs easier by removing the requirement of decorating all pointers with a *points to* address space when the programmer may not care or may want to use the same function regardless of the address space of the incoming pointer. For example, one can imagine incrementing the value of a histogram, adding or sorting a set of values in an array, or a set of operations on a per element basis. In all of these cases the OpenCL C 1.2 specification requires us to write a version of the function for each address space we expect to enter the function. This forces the developer to maintain multiple versions of the same function thus increasing the chances of making changes in one segment of code and not the other which can increase the risk of versioning issues. While the new generic address space eliminates the *requirement* for decorating all of the pointers with an address space, it does not *require* one to remove the address space, so all of your old OpenCL 1.2 code will continue to work as written. Programs that benefits from specifying the address space can continue to benefit from address spaces declared by the programmer.

Another reason one may choose to leverage the generic address space is to make it easier to cross compile a segment of C code on the CPU and the GPU, for example a data structure or set of core functions we use on both the host and device. Without the generic address space enabled code the programmer is forced to trick the host or device compiler to handle address space keywords by transforming address spaces to whitespace or other cleverness. It doesn't take care of all the issues but make it easier to compile the same code with different C compilers.

Performing Some Operations in a Specific Address Space

In some cases a programmer may want to write a function that operates in a generic fashion but there are some operations needed for a specific address space. In the case of a histogram, imagine incrementing a value in local memory may not require an atomic, but incrementing a value in a histogram shared globally among the workgroups would require an atomic operation on global memory. In such a case, there are built-ins to help for these portions of a function. The functions to_global(), to_local(), to_private() can be used to cast a pointer to the respective address space. If for some reason these functions are not able to cast a pointer to the respective address space they will return NULL. This allows the programmer to know if a pointer can be treated as if it points to the respective address space or not.

One might wonder why these functions do not return a Boolean based on the value of is_local(), is_global(), and is_private(), for example. The reason is that some implementations may treat one address space as another and it is better to allow them to return the pointer value if they can be treated as if they are in the requested address space. For example, CPUs may do this for all address spaces.

Address Space Casting

Casting from one named address space to another named address space is not allowed. The address space of a pointer to a named address space *can* be assigned to the generic address space but not the other way around. Also, a single generic pointer may be assigned to different named address spaces in the same code sequence. A variable that points to the constant address space is not convertible to any of the members of the generic address space.

This next example is legal. lp is a pointer to the local address space, g is a pointer in private memory pointing to the generic address space and is assigned a pointer that points to the local address space.

```
local int *lp;

int *g;

g = lp; //success!
```

However, the next example is not legal. The pointer is in private memory pointing to the local address space and this is an attempt to assign it a generic value, this is an error.

```
local *lp;

lp = p;  //error!
```

The OpenCL C specification Section 6.5 included in the References has numerous examples of legal and illegal casting.

Performance Implications and How to Address Them

In some implementations the performance of a function can be negatively impacted if the compiler cannot resolve the address space being pointed to at compile time. If this is an issue you can decorate the relevant pointer with a specific address space and the compiler does not have to generate any additional code to handle the generic address space. The generic address space in the cases when the address space cannot be resolved at compile time may have a small performance impact in the case of very small kernels. Ideally, compilers will give feedback to the programmer that the address space of a parameter was not able to be resolved but as far as I know no compiler yet publicly supports this feature. Also, most kernels should be many more instructions than the few additional instructions needed at runtime to resolve the address space so this cost is expected to amortize relative to the execution time of the kernel.

A Working Example

To demonstrate this functionality we have written a short code sequence that behaves correctly in OpenCL 1.2 and the same code sequence using the generic address space for OpenCL 2.0. We use a simple synthetic kernel that mimics a memcpy() from a buffer segment in either the global or local address space to a global buffer. It is easy to imagine such a kernel doing a set of math operations on the value before writing it back to the global memory buffer on a per element basis. For example the conversion from an RGB image to YUV which would require several multiplies and adds on the input values. The same set of operations would take place on the data the only difference would be the address spaces on the input buffer and this is a good candidate for the use of the generic address space as one may have some machines that perform better by leaving the buffer in global memory before the RGB to YUV conversation and other implementations that may choose to tile the image into local memory before doing any additional operations.

A simple memcpy() function written in OpenCL 1.2 that loads from global memory and writes back to another location in global memory can be written:

```
void GlobalXToY_internal(__global unsigned char *in, __global unsigned char *out, unsigned int startFrom, unsigned int startTo, unsigned int length)
{
       for(int i = startFrom; i < startFrom + length;)
       {
              out[startTo++] = in[i++];
       }
}
```

To get the equivalent functionality in OpenCL 2.0 that can accept as input a value in the global or local address space is written as follows. Note the elimination of the keyword __global on the first function argument:

```
void GenericXToY_internal(unsigned char *in, unsigned char *out, unsigned int startFrom, unsigned int startTo, unsigned int length)
{
       for(int i = startFrom; i < startFrom + length;)
       {
              out[startTo++] = in[i++];
       }
}
```

The sample compiles and runs on any Intel Processor with Intel Processor Graphics supporting OpenCL 2.0. It has not been tested on other implementations at this time.

Future work

In the future we could augment this memcpy() example with a more complicated workload, for example tiled memory operations or a more complicated matrix multiplication. Also, we would like to see more OpenCL 2.0 implementations and we could enable the sample on those platforms. Additionally, doing analysis to verify the cost of using the generic address space would give us greater assurance to the performance implications.

Acknowledgements

We would like to thank Dillon Sharlet who was a close collaborator on the initial generic address space proposal, as well as Ben Ashbaugh, Stephen Junkins, Ben Gaster, and others who made significant contributions and clarifications during its development. Also thanks to the other vendors in Khronos who worked to do the proper analysis of the feature before inclusion into the OpenCL specification.

About the Authors

Adam Lake works in the Visual Products Group as a Senior Graphics Architect and Voting Representative to the Khronos OpenCL Standards Body. He has worked on GPGPU programming for 12+ years. Previously he has worked in VR, 3D, graphics, and stream programming language compilers.

Robert Ioffe is a Technical Consulting Engineer at Intel's Software and Solutions Group. He was heavily involved in Khronos standards work, focusing on prototyping the latest features and making sure they can run well on Intel architecture.

You might also be interested in the following articles:

[Optimizing Simple OpenCL Kernels: Modulate Kernel Optimization (https://software.intel.com/en-us/videos/optimizing-simple-opencl-kernels-modulate-kernel-optimization)](https://software.intel.com/en-us/videos/optimizing-simple-opencl-kernels-modulate-kernel-optimization)

Optimizing Simple OpenCL Kernels: Sobel Kernel Optimization (https://software.intel.com/en-us/videos/optimizing-simple-opencl-kernels-sobel-kernel-optimization)

GPU-Quicksort in OpenCL 2.0: Nested Parallelism and Work-Group Scan Functions (https://software.intel.com/en-us/articles/gpu-quicksort-in-opencl-20-using-nested-parallelism-and-work-group-scan-functions)

Sierpiński Carpet in OpenCL 2.0 (https://software.intel.com/en-us/articles/sierpinski-carpet-in-opencl-20)

References

Embedded C: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf)

Shared Virtual Memory Sample: https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-code-sample (https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-code-sample)

Additional samples available at: https://software.intel.com/en-us/intel-opencl-support/product-library (https://software.intel.com/en-us/intel-opencl-support/product-library)

Intel® SDK for OpenCL™ Applications: https://software.intel.com/en-us/intel-opencl (https://software.intel.com/en-us/intel-opencl)

OpenCL API Specification: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf (https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf)

OpenCL C Specification: https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf (https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf)

Legal Information

Download the Sample

Attachment	Size
generic address space.zip (https://software.intel.com/sites/default/files/managed/01/62/generic%20address%20space.zip)	15.5 KB