C++ AMP for the CUDA Programmer

April 11, 2012—The most-up-to-date version of this guide can be found on the C++ AMP team blog:

http://blogs.msdn.com/b/nativeconcurrency/

If you are familiar with Nvidia's CUDA programming language for targeting NVIDIA hardware, this guide will provide a gentle introduction to C++ AMP using familiar terminology and concepts.

C++ AMP extends C++ with two new language features: a new storage class, tile_static, which corresponds to CUDA's *__shared__* qualifier, and a new language feature, a non-extensible set of restriction specifiers for restricting the content and behavior of functions. The *restrict(amp)* specifier limits functions so that they are amenable to execution on a typical GPU. The rest of C++ AMP is implemented as a library mostly in the *concurrency* namespace and the <amp.h> header file.

This guide is divided into two parts. In the first part, we'll rewrite a CUDA implementation of a well-known algorithm for matrix multiplation in C++ AMP. In the second part, we'll present several tables that map the most common functionality in CUDA to equivalent functionality in C++ AMP.

From CUDA to C++ AMP: Tiled Matrix Multiplication

In this section, we will take a CUDA implementation of the classic tiled (blocked) algorithm for matrix multiplication and rewrite it, step-by-step, in C++ AMP. For conciseness, code to handle errors in CUDA via the *CudaError_t* return value *err* is omitted. In C++ AMP, errors surface as exceptions, and we will omit the very minimal code necessary to catch these exceptions. In addition, and also for conciseness, we have made the simplifying assumptions that the matrix width is a multiple of 16 and equal to the matrix height.

When porting from CUDA to C++ AMP, the first step is to include a different header file. The functionality for using C++ AMP is contained in the *concurrency* namespace in the header file *amp.h*.

mm_cuda.cu

```
#include <cuda_runtime.h>
```

mm_amp.cpp

```
#include <amp.h>
using namespace concurrency;
```

Let's say we wrap matrix multiplication in a single function named *mm*. We'll create the same function in C++ AMP. In CUDA, we'll also define a function that executes the kernel computation on the GPU; in C++ AMP, we'll implement the same computation at the point where it is invoked. To make the comparison easier, we'll show the CUDA code later, leaving the lines blank for now.

```
__global__ void mm_kernel(const float *d_A, const float *d_B, float *d_C, int size)
```

```
{
  // The body of this function (18 lines) will be shown below with the C++ AMP equivalent...
}

void mm(const float * A, const float * B, float * C, int size)
{
```

```
void mm(const float * A, const float * B, float * C, int size)
{
```

The next step in CUDA is to allocate arrays on the GPU and copy the data from the CPU, using calls to cudaMalloc and cudaMemcpy respectively. In C++ AMP, we use either *array* or *array_view* objects for the same purpose. In this example, we'll use the *array_view* type. An *array_view* is an array-like data structure that provides a view of the data in some other structure. In the code below, the views *d_A*, *d_B*, and *d_C* alias the data in *A*, *B*, and *C* respectively. When these array views are used on a GPU, the data is implicitly copied to the GPU. The data is implicitly copied back when the views are destroyed, when the data is accessed, or when explicitly synchronized. The constant type qualifier (const) on the element types of *d_A* and *d_B* ensures that the data of these views will not be copied back from the accelerator. Similarly, the call to the *discard_data* method ensures that the data of *d_C* is not copied to the accelerator.

Note that in both CUDA and C++ AMP, a different code organization could be more optimal if we wish to call this function multiple times or call other functions on the GPU that access the same data. For example, we may wish to optimize how and when the data is copied from the host to the device in both CUDA and C++ AMP.

```
  cudaError_t err;
  float *d_A, *d_B, *d_C;
  err = cudaMalloc(&d_A, size * size * sizeof(float));
  err = cudaMemcpy(d_A, A, size * size * sizeof(float), cudaMemcpyHostToDevice);
  err = cudaMalloc(&d_B, size * size * sizeof(float));
  err = cudaMemcpy(d_B, B, size * size * sizeof(float), cudaMemcpyHostToDevice);
  err = cudaMalloc(&d_C, size * size * sizeof(float));
```

```
  array_view<const float, 2> d_A(size, size, A);
  array_view<const float, 2> d_B(size, size, B);
  array_view<float, 2> d_C(size, size, C);
  d_C.discard_data();
```

To launch the computation in C++ AMP, we use a *parallel_for_each* looping construct similar in form to *std::for_each*. An *extent*, a rectangular zero-based index set, that is derived from *d_C* specifies the total number of threads and the 2D shape. We tile it to specify the number of threads per tile. In contrast, the number of thread blocks and the number of threads per block is specified in the CUDA.

In C++ AMP, the second argument to the *parallel_for_each* is the code that will execute on the accelerator in the form of a C++ lambda. We'll see that next.

```
  dim3 grid(size/16, size/16);
  dim3 threads(16, 16);
  mm_kernel<<<grid, threads>>>(d_A, d_B, d_C, size);
```

```
  parallel_for_each(d_C.extent.tile<16, 16>(),
```

In C++ AMP, we use a lambda to write the kernel in the *parallel_for_each* call expression. Think of the lambda as an anonymous function object that can implicitly access the variables in its enclosing scope. In CUDA, the kernel is declared in its own function, but we'll show it here to match the intuitive program flow.

Like the kernel in CUDA, the lambda in C++ AMP will be called for each index in the *extent*, and each call will be

executed by a different thread. The *tiled_index* provides equivalent functionality to the CUDA variables *blockIdx*, *blockDim*, and *threadIdx*. Similarly, the *tile_static* qualifier and *wait* method in C++ AMP respectively provide equivalent functionality to the *__shared__* qualifier and *syncThreads* function in CUDA. The C++ AMP code does not require the size or the arrays to be passed as parameters since they are captured from the outer scope by value.

Indexing in C++ AMP is multi-dimensional. In line 28, notice the direct indexing into the array with the index object. This is equivalent to *d_C(t_idx.global[0], t_idx.global[1])*.

```
__global__ void mm_kernel(int size, const float *d_A, const float *d_B, float *d_C)
{
  int row = threadIdx.y;
  int col = threadIdx.x;
  __shared__ float local_a[16][16];
  __shared__ float local_b[16][16];
  float sum = 0.0f;
  for (int i = 0; i < size; i += 16)
  {
    local_a[row][col] = d_A[(blockIdx.y*blockDim.y+threadIdx.y) * size + i + col];
    local_b[row][col] = d_B[(i + row) * size + (blockIdx.x*blockDim.x+threadIdx.x)];
    __syncthreads();
    for (int k = 0; k < 16; k++)
    {
      sum += local_a[row][k] * local_b[k][col];
    }
    __syncthreads();
  }
  d_C[(blockIdx.y*blockDim.y+threadIdx.y) * size + blockIdx.x*blockDim.x+threadIdx.x] = sum;
}
```

```
    [=] (tiled_index<16, 16> t_idx) restrict(amp)
    {
      int row = t_idx.local[0];
      int col = t_idx.local[1];
      tile_static float local_a[16][16];
      tile_static float local_b[16][16];
      float sum = 0.0f;
      for (int i = 0; i < size; i += 16)
      {
        local_a[row][col] = d_A(t_idx.global[0], i + col);
        local_b[row][col] = d_B(i + row, t_idx.global[1]);
        t_idx.barrier.wait();
        for (int k = 0; k < 16; k++)
        {
          sum += local_a[row][k] * local_b[k][col];
        }
        t_idx.barrier.wait();
      }
      d_C[t_idx.global] = sum;
    });
```

The final step is to copy the data back to the CPU and then free the arrays. In C++ AMP, the data in the *array_view* object is copied back implicitly when the destructor is called, though note that no copy back is performed for array_view objects with constant data.

```
  err = cudaMemcpy(C, d_C, size * size * sizeof(float), cudaMemcpyDeviceToHost);
  err = cudaFree(d_A);
  err = cudaFree(d_B);
  err = cudaFree(d_C);
}
```

```
}
```

C++ AMP can be simplified further if the algorithm does not require local or tiled indexing, barriers, or *tile_static* memory. As an example, let's look at rewriting matrix multiplication without the tiling optimization. In C++ AMP, we can simplify the 21 lines (9—29) of mm_amp.cpp with the following 9 lines of code:

```
9   parallel_for_each(d_C.extent, [=] (index<2> idx) restrict(amp)
10    {
11      float sum = 0.0f;
12      for (int i = 0; i < size; i++)
13      {
14        sum += d_A(idx[0], i) * d_B(i, idx[1]);
15      }
16      d_C[idx] = sum;
17    });
```

Lines 1—8 and 30 remain unchanged. The computation is still parallelized using multiple tiles, but the mapping from indices to tiles is determined by the compiler and runtime, and this mapping is invisible to the user. There is no requirement that the extent bounds be evenly divisible by any computed tile size.

The C++ AMP indexing and kernel launch is also greatly simplified to use a non-tiled extent and non-tiled indices. Non-tiled indices are simple tuples of integer values. There is no equivalent to non-tiled indices in CUDA.

From CUDA to C++ AMP: Mapping Terms and Concepts

CUDA and C++ AMP define similar concepts but sometimes use different terminology and names. For example, thread blocks in CUDA are called tiles in C++ AMP. This section contains a number of tables that will map CUDA terms and concepts to C++ AMP terms and concepts.

Arrays

C++ AMP provides two class templates, *array* and *array_view*, for defining data buffers that can be accessed on the GPU, and you can choose to use either one of them. They can both be used at the same time, but you do not need to use both of them. These are parameterized over an element type and a constant number of dimensions (rank) for type safety and ease of use.

CUDA	C++ AMP	Notes
`type* (allocated on GPU)`	`array<type,rank>`	The *array* class is a multidimensional array that supports indexing and other array-like functionality. The memory in a given array resides on a specific accelerator.
	`array_view<type,rank>`	The *array_view* class is a multidimensional view of a container that supports the same functionality as an array, but which is implicitly mapped to accelerators as needed.

Kernel Code

In C++ AMP, code that runs on the GPU must have the amp restriction specified. The default restriction for C++ is cpu. Code that can be executed on both a GPU and a CPU can have both restrictions. The distinction between code that can be called from the host and code that can only be called from the device is not made explicit with these qualifiers as is

done in CUDA, and the same function can be shared between the host and the device. However, a lambda or functor acting as the kernel entry function must take a single argument of *index* or *tiled_index* type.

CUDA	C++ AMP
`__global__`	`restrict(amp)`
`__device__`	`restrict(amp)`
`__host__` *(default)*	`restrict(cpu)` *(default)*
`__host__ __device__`	`restrict(cpu,amp)`

Note that C++ AMP allows overloading of functions based on restrictions. This makes it easier to share code between the host and the device since such code can call functions with the same name that have different implementations.

Kernel Memory

Both CUDA and C++ AMP introduce qualifiers to map data to memory on the GPU. The following table describes the equivalent qualifiers:

CUDA	C++ AMP
`__shared__`	`tile_static`
`__device__`	*Unnecessary/implicit*
`__constant__`	*Unnecessary/implicit*

Kernel Indexing

CUDA kernels have access to four global variables (*threadIdx*, *blockIdx*, *blockDim*, and *gridDim*) that provide functionality for indexing into the computation space. C++ AMP provides the same functionality through a single parameter of type *tiled_index* (typically named *t_idx*).

It is important to note that the order (alphabetical or numerical) of an index's components is reversed between CUDA and C++ AMP. The order in C++ AMP increases from left-to-right when indexing into data. In the tiled matrix multiplication example above, recall that *d_C(t_idx.global)*, expanding to *d_C(t_idx.global[0], t_idx.global[1])*, is equivalent to

*d_C[(blockIdx.y*blockDim.y+threadIdx.y) * size + blockIdx.x*blockDim.x+threadIdx.x]*

in CUDA.

For 1D, 2D, and 3D tiled indices, the following tables show the C++ AMP equivalent for CUDA code given the parameters above and assuming access to a captured *tiled_extent* variable named *t_ext* and a captured *extent* variable named *ext*:

1D	CUDA	C++ AMP
	`threadIdx.x`	`t_idx.local[0]`
	`blockIdx.x`	`t_idx.tile[0]`

blockDim.x * blockIdx.x + threadIdx.x	t_idx.global[0]
blockDim.x * blockIdx.x	t_idx.tile_origin[0]
blockDim.x	t_ext.tile_dim0
gridDim.x	t_ext[0]
gridDim.x * blockDim.x	ext[0]

2D	CUDA	C++ AMP
	threadIdx.y threadIdx.x	t_idx.local[0] t_idx.local[1]
	blockIdx.y blockIdx.x	t_idx.tile[0] t_idx.tile[1]
	blockDim.y * blockIdx.y + threadIdx.y blockDim.x * blockIdx.x + threadIdx.x	t_idx.global[0] t_idx.global[1]
	blockDim.y * blockIdx.y blockDim.x * blockIdx.x	t_idx.tile_origin[0] t_idx.tile_origin[1]
	blockDim.y blockDim.x	t_ext.tile_dim0 t_ext.tile_dim1
	gridDim.y gridDim.x	t_ext[0] t_ext[1]
	gridDim.y * blockDim.y gridDim.x * blockDim.x	ext[0] ext[1]

3D	CUDA	C++ AMP
	threadIdx.z threadIdx.y threadIdx.x	t_idx.local[0] t_idx.local[1] t_idx.local[2]
	blockIdx.z blockIdx.y blockIdx.x	t_idx.tile[0] t_idx.tile[1] t_idx.tile[2]
	blockDim.z * blockIdx.z + threadIdx.z blockDim.y * blockIdx.y + threadIdx.y blockDim.x * blockIdx.x + threadIdx.x	t_idx.global[0] t_idx.global[1] t_idx.global[2]
	blockDim.z * blockIdx.z blockDim.y * blockIdx.y blockDim.x * blockIdx.x	t_idx.tile_origin[0] t_idx.tile_origin[1] t_idx.tile_origin[2]
	blockDim.z blockDim.y blockDim.x	t_ext.tile_dim0 t_ext.tile_dim1 t_ext.tile_dim2
	gridDim.z gridDim.y	t_ext[0] t_ext[1]

gridDim.x	t_ext[2]
gridDim.z * blockDim.z gridDim.y * blockDim.y gridDim.x * blockDim.x	ext[0] ext[1] ext[2]

If a kernel does not use *t_idx.local* or *t_idx.tile*, then you can simplify the equivalent code in C++ AMP by using the non-tiled *index* type. There is no equivalent to non-tiled indices in CUDA. For non-tiled indices, the following tables show the simplified C++ AMP equivalent for CUDA code given an index parameter named *idx* and assuming access to a captured *extent* variable named *ext*:

1D	CUDA	C++ AMP
	blockDim.x * blockIdx.x + threadIdx.x	idx[0]
	gridDim.x * blockDim.x	ext[0]

2D	CUDA	C++ AMP
	blockDim.y * blockIdx.y + threadIdx.y blockDim.x * blockIdx.x + threadIdx.x	idx[0] idx[1]
	gridDim.y * blockDim.y gridDim.x * blockDim.x	ext[0] ext[1]

3D	CUDA	C++ AMP
	blockDim.z * blockIdx.z + threadIdx.z blockDim.y * blockIdx.y + threadIdx.y blockDim.x * blockIdx.x + threadIdx.x	idx[0] idx[1] idx[2]
	gridDim.z * blockDim.z gridDim.y * blockDim.y gridDim.x * blockDim.x	ext[0] ext[1] ext[2]

Kernel Synchronization

Within a tile, CUDA and C++ AMP provide similar functionality to synchronize between the threads of that tile.

CUDA	C++ AMP
__syncthreads()	t_idx.barrier.wait() ***or*** t_idx.barrier.wait_with_all_memory_fence()
No equivalent	t_idx.barrier.wait_with_tile_static_memory_fence()
No equivalent	t_idx.barrier.wait_with_global_memory_fence()
__threadfence_block()	all_memory_fence(t_idx.barrier)
No equivalent	tile_static_memory_fence(t_idx.barrier)

No equivalent	global_memory_fence(t_idx.barrier)
__threadfence()	No equivalent
__threadfence_system()	No equivalent

There are subtle differences between the semantics of memory fences in CUDA and in C++ AMP. Note that the C++ AMP semantics of fences are more similar to OpenCL.

In addition, both CUDA and C++ AMP provide functions to implement the following atomic operations:

CUDA	C++ AMP
atomicAdd	atomic_fetch_add
atomicSub	atomic_fetch_sub
atomicInc	atomic_fetch_inc
atomicDec	atomic_fetch_dec
atomicExch	atomic_exchange
atomicCAS	atomic_compare_exchange
atomicMax	atomic_fetch_max
atomicMin	atomic_fetch_min
atomicAnd	atomic_fetch_and
atomicOr	atomic_fetch_or
atomicXor	atomic_fetch_xor

Host Functionality

CUDA provides several types and a large number of functions to coordinate on the host with the device. The following tables summarize how this functionality maps to C++ AMP.

Data structures

CUDA	C++ AMP
CUcontext	accelerator_view
CUdevice	accelerator
CUmodule	No equivalent is necessary. The kernel code is part of the C++ source.
CUfunction	No equivalent is necessary. The kernel code is part of the C++ source.

Functions

CUDA	C++ AMP

cudaGetDeviceCount cudaGetDeviceProperties	The accelerator::get_all() function returns a list of all of the available accelerator objects. Properties and members of accelerator objects can be used to query information about an accelerator.
cudaMalloc	Use the array or array_view constructor. See notes on arrays above.
cudaHostAlloc	Use an array on the CPU accelerator.
cudaMemcpy	Use the copy function, or the synchronize method to synchronize an array_view with the host. Implicit synchronization is often sufficient.
Function call syntax with <<< and >>>	parallel_for_each
cudaFree	No equivalent is necessary. There is rarely a need to release resources or free memory explicitly as resources are released and memory is freed implicitly when objects are destroyed. As always, remember to free any arrays or other objects that are allocated on the heap.

Closing Thoughts

C++ AMP is a powerful and portable way to elegantly and succinctly program GPUs entirely in C++. For CUDA programmers, it should be easy to learn. When coming to C++ AMP from CUDA, keep these following thoughts in mind:

- C++ AMP makes it easy to write a function that can be called from both the host and the accelerator by marking the function as *restrict(amp,cpu)*.
- C++ AMP lets you overload functions with the same signature that differ only by the restriction specifier. These functions can then be called from code that runs either on the host or the accelerator, *e.g.*, from within a function marked as *restrict(cpu,amp)*.
- Templates, the *auto* keyword, and many other high-level C++ features can be used in *restrict(amp)* code to make for easy-to-maintain programs.
- There are many more features and functions available in both C++ AMP and CUDA. In particular, this guide has omitted all discussion of C++ AMP's math libraries, graphics libraries (including support for textures, norms, and short vectors), interoperability, and error handling. If you have questions, please ask them in our MSDN Forum: http://social.msdn.microsoft.com/Forums/en-US/parallelcppnative/threads

A good starting place to learn more about C++ AMP is the C++ AMP team blog:

http://blogs.msdn.com/b/nativeconcurrency/