If you are familiar with OpenCL, this guide will provide a gentle introduction to C++ AMP using familiar terminology and concepts.

OpenCL draws a sharp distinction between the code that runs on the host (typical C or C++) and the code that runs on the device (C-like OpenCL kernel code). The host code and the device code typically reside in different files usually with different filename extensions: .c or .cpp versus .cl. Moreover, if the OpenCL kernel code is compiled offline, there are two binaries that must be produced. In C++ AMP, the distinction is still there, but the code is all written in C++, it can all exist in a single file, and a single binary is produced by a single compiler.

C++ AMP extends C++ with two new language features: a new storage class, tile_static, which corresponds to OpenCL’s __local 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 an OpenCL implementation of a well-known algorithm for matrix multiplication in C++ AMP. In the second part, we’ll present several tables that map the most common functionality in OpenCL to equivalent functionality in C++ AMP.

From OpenCL to C++ AMP: Tiled Matrix Multiplication

In this section, we will take an OpenCL 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 OpenCL via the code 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.

Following convention, the OpenCL implementation is divided into two files: mm_opencl.cpp and mm.cl. On the other hand, the C++ AMP is contained in the single file mm_amp.cpp.

When porting from OpenCL 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.

```cpp
#include <CL\opencl.h>
```
Let's say we wrap matrix multiplication in a single function named `mm`. We'll create the same function in C++ AMP. There is no change necessary here.

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

In OpenCL, the next step is to select a device, load a kernel, and otherwise initialize the program state. In C++ AMP, the computation is written entirely in C++ so we don’t have to explicitly compile or load a kernel. The kernel code is compiled along with the C++ and the binary is embedded in the C++ binary. Since there is a default accelerator that turns out to be the most available GPU, we also don’t have to explicitly select one. Thus this start-up code can be omitted when implementing the C++ AMP code.

Note that we’re referring to a variable `source` that contains the OpenCL kernel source code as a string. This is typically read from a file containing the OpenCL kernel code. We’ve omitted the code to load the source into the string for brevity. Alternatively, the OpenCL kernel can be compiled separately and the binary can be loaded in its place.

```cpp
cl_int err;
cl_context context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, &err);
size_t devicesSize;
err = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &devicesSize);
cl_device_id *devices = (cl_device_id*) malloc(devicesSize);
err = clGetContextInfo(context, CL_CONTEXT_DEVICES, devicesSize, devices, NULL);
cl_command_queue commandQueue = clCreateCommandQueue(context, devices[0], 0, &err);
cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, &err);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "mm", &err);
```

The next step is to create buffers on the GPU and copy data to these buffers from the CPU. We can replace OpenCL buffers with either `array` or `array_view` objects in C++ AMP. 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 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 OpenCL 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 should take care to build the kernel in OpenCL only once, and we may wish to optimize how and when the data is copied from the host to the device in both OpenCL and C++ AMP.

```cpp
cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
size*size*sizeof(float), (void*)A, &err);
cl_mem d_B = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
size*size*sizeof(float), (void*)B, &err);
cl_mem d_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size*size*sizeof(float), NULL, &err);
```
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();

In OpenCL, the buffers and the variable size must be explicitly sent to the kernel. In C++ AMP, as we will see below, the array_view objects and the variable size are implicitly captured from the outer scope by value.

err = clSetKernelArg(kernel, 0, sizeof(int), (void*)&size);
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&d_A);
err = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&d_B);
err = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void*)&d_C);

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 global work size. We tile it to specify the local work size. The kernel is executed on the default GPU by default, but we could have also specified one. 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.

size_t localWorkSize[] = {16, 16};
size_t globalWorkSize[] = {size, size};
err = clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL, globalWorkSize, localWorkSize, 0,
                                   NULL, NULL);
parallel_for_each(d_C.extent.tile<16, 16>(),

In OpenCL, the following code exists in the file mm.cl, but we’ll show it here to match the intuitive program flow.

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.

Like the kernel in OpenCL, the lambda in C++ AMP will be called for each index in the extent (the equivalent of the global work space). The tiled_index provides equivalent functionality to the OpenCL functions like get_local_id and get_global_id. Similarly, the tile_static qualifier and wait method in C++ AMP respectively provide equivalent functionality to the __local qualifier and barrier function in OpenCL.

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]).

mm.cl
__kernel __attribute__((reqd_work_group_size(16, 16, 1)))
void mm(int size, const __global float *d_A, const __global float *d_B, __global float *d_C) {
    int row = get_local_id(1);
    int col = get_local_id(0);
    __local float local_a[16][16];
    __local float local_b[16][16];
    float sum = 0.0f;
    for (int i = 0; i < size; i += 16) {
        local_a[row][col] = d_A[get_global_id(1) * size + i + col];
        local_b[row][col] = d_B[(i + row) * size + get_global_id(0)];
        barrier(CLK_GLOBAL_MEM_FENCE);
        for (int k = 0; k < 16; k++)
        {
            sum += local_a[row][k] * local_b[k][col];
        }
    }
barrier(CLK_GLOBAL_MEM_FENCE);
}
d_C[get_global_id(1) * size + get_global_id(0)] = 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 clean up the buffers and other state. 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. The rest of the cleanup is also implicit in C++ AMP since the destructors are called when the scope is closed.

```
err = clEnqueueReadBuffer(commandQueue, d_C, CL_TRUE, 0, size*size*sizeof(float), C, 0, NULL, NULL);
err = clReleaseMemObject(d_A);
err = clReleaseMemObject(d_B);
err = clReleaseMemObject(d_C);
err = clReleaseKernel(kernel);
err = clReleaseProgram(program);
err = clReleaseCommandQueue(commandQueue);
err = clReleaseContext(context);
```

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:

```
parallel_for_each(d_C.extent, [=] (index<2> idx) restrict(amp)
{
    float sum = 0.0f;
    for (int i = 0; i < size; i++)
    {
        sum += d_A(idx[0], i) * d_B(i, idx[1]);
    }
    d_C[idx] = sum;
});
```
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.

From OpenCL to C++ AMP: Mapping Terms and Concepts
OpenCL and C++ AMP define similar concepts but sometimes use different terminology and names. For example, work items in OpenCL are called threads in C++ AMP; work groups in OpenCL are called tiles in C++ AMP. This section contains a number of tables that will map OpenCL terms and concepts to C++ AMP terms and concepts.

Buffers
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.

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_mem</td>
<td>array&lt;type,rank&gt;</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>array_view&lt;type,rank&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 in C++ AMP.

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>__kernel (in OpenCL source)</td>
<td>restrict(amp)</td>
</tr>
<tr>
<td>Default in OpenCL source</td>
<td>restrict(amp)</td>
</tr>
<tr>
<td>Default in C++ source</td>
<td>restrict(cpu) (default)</td>
</tr>
<tr>
<td>No equivalent for reusing code between host and device.</td>
<td>restrict(cpu,amp)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
</table>

5
Kernel Indexing

OpenCL kernels have access to three functions \( \text{get\_local\_id}, \text{get\_group\_id}, \text{and get\_global\_id} \) that provide functionality for indexing into the computation space. C++ AMP provides the same functionality through a single parameter of type \( \text{tiled\_index} \) (typically named \( t\_idx \)).

It is important to note that the numerical order of an index’s components is reversed between OpenCL 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 \( d_\text{C}(t\_idx\_global) \), which can be written as \( d_\text{C}(t\_idx\_global[0], t\_idx\_global[1]) \), is equivalent to

\[
d_\text{C}(\text{get\_global\_id}(1) \times \text{size} + \text{get\_global\_id}(0))
\]

in OpenCL.

For 1D, 2D, and 3D tiled indices, the following tables show the C++ AMP equivalent for OpenCL code given the parameters above and assuming access to a captured \( \text{tiled\_extent} \) variable named \( t\_ext \) and a captured \( \text{extent} \) variable named \( ext \):

### 1D

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_local_id(0)</td>
<td>( t_idx_local[0] )</td>
</tr>
<tr>
<td>get_group_id(0)</td>
<td>( t_idx_tile[0] )</td>
</tr>
<tr>
<td>get_global_id(0)</td>
<td>( t_idx_global[0] )</td>
</tr>
<tr>
<td>get_global_id(0)-get_local_id(0)</td>
<td>( t_idx_tile_origin[0] )</td>
</tr>
<tr>
<td>get_local_size(0)</td>
<td>( t_ext_tile_dim0 )</td>
</tr>
<tr>
<td>get_num_groups(0)</td>
<td>( t_ext[0] )</td>
</tr>
<tr>
<td>get_global_size(0)</td>
<td>( ext[0] )</td>
</tr>
</tbody>
</table>

### 2D

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_local_id(1) get_local_id(0)</td>
<td>( t_idx_local[0] )</td>
</tr>
<tr>
<td>get_local_id(0) get_local_id(0)</td>
<td>( t_idx_local[1] )</td>
</tr>
<tr>
<td>get_group_id(1) get_group_id(0)</td>
<td>( t_idx_tile[0] )</td>
</tr>
<tr>
<td>get_group_id(0) get_group_id(0)</td>
<td>( t_idx_tile[1] )</td>
</tr>
<tr>
<td>get_global_id(1) get_global_id(0)</td>
<td>( t_idx_global[0] )</td>
</tr>
<tr>
<td>get_global_id(0) get_global_id(0)</td>
<td>( t_idx_global[1] )</td>
</tr>
<tr>
<td>get_global_id(1)-get_local_id(1) get_local_id(0)</td>
<td>( t_idx_tile_origin[0] )</td>
</tr>
<tr>
<td>get_global_id(0)-get_local_id(0) get_local_id(0)</td>
<td>( t_idx_tile_origin[1] )</td>
</tr>
</tbody>
</table>
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. For non-tiled indices, the following tables show the simplified C++ AMP equivalent for OpenCL code given an index parameter named `idx` and assuming access to a captured `extent` variable named `ext`:

### 3D

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_local_id(2)</code></td>
<td><code>t_idx.local[0]</code></td>
</tr>
<tr>
<td><code>get_local_id(1)</code></td>
<td><code>t_idx.local[1]</code></td>
</tr>
<tr>
<td><code>get_local_id(0)</code></td>
<td><code>t_idx.local[2]</code></td>
</tr>
<tr>
<td><code>get_group_id(2)</code></td>
<td><code>t_idx.tile[0]</code></td>
</tr>
<tr>
<td><code>get_group_id(1)</code></td>
<td><code>t_idx.tile[1]</code></td>
</tr>
<tr>
<td><code>get_group_id(0)</code></td>
<td><code>t_idx.tile[2]</code></td>
</tr>
<tr>
<td><code>get_global_id(2)</code> - <code>get_local_id(2)</code></td>
<td><code>t_idx.tile_origin[0]</code></td>
</tr>
<tr>
<td><code>get_global_id(1)</code> - <code>get_local_id(1)</code></td>
<td><code>t_idx.tile_origin[1]</code></td>
</tr>
<tr>
<td><code>get_global_id(0)</code> - <code>get_local_id(0)</code></td>
<td><code>t_idx.tile_origin[2]</code></td>
</tr>
<tr>
<td><code>get_local_size(2)</code></td>
<td><code>t_ext.tile_dim0</code></td>
</tr>
<tr>
<td><code>get_local_size(1)</code></td>
<td><code>t_ext.tile_dim1</code></td>
</tr>
<tr>
<td><code>get_local_size(0)</code></td>
<td><code>t_ext.tile_dim2</code></td>
</tr>
<tr>
<td><code>get_num_groups(2)</code></td>
<td><code>t_ext[0]</code></td>
</tr>
<tr>
<td><code>get_num_groups(1)</code></td>
<td><code>t_ext[1]</code></td>
</tr>
<tr>
<td><code>get_num_groups(0)</code></td>
<td><code>t_ext[2]</code></td>
</tr>
<tr>
<td><code>get_global_size(2)</code></td>
<td><code>ext[0]</code></td>
</tr>
<tr>
<td><code>get_global_size(1)</code></td>
<td><code>ext[1]</code></td>
</tr>
<tr>
<td><code>get_global_size(0)</code></td>
<td><code>ext[2]</code></td>
</tr>
</tbody>
</table>

### 1D

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_global_id(0)</code></td>
<td><code>idx[0]</code></td>
</tr>
<tr>
<td><code>get_global_size(0)</code></td>
<td><code>ext[0]</code></td>
</tr>
</tbody>
</table>

### 2D

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_global_id(1)</code></td>
<td><code>idx[0]</code></td>
</tr>
<tr>
<td><code>get_global_id(0)</code></td>
<td><code>idx[1]</code></td>
</tr>
</tbody>
</table>
Kernel Synchronization

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

### OpenCL

- `barrier(CLK_LOCAL_MEM_FENCE)`
- `barrier(CLK_GLOBAL_MEM_FENCE)`
- `mem_fence(CLK_LOCAL_MEM_FENCE)`
- `mem_fence(CLK_GLOBAL_MEM_FENCE)`

### C++ AMP

- `t_idx.barrier.wait()` or `t_idx.barrier.wait_with_all_memory_fence()`
- `t_idx.barrier.wait_with_tile_static_memory_fence()`
- `all_memory_fence(t_idx.barrier)`
- `tile_static_memory_fence(t_idx.barrier)`
- `global_memory_fence(t_idx.barrier)`

In addition, both OpenCL and C++ AMP provide functions to implement the following atomic operations: (C++ AMP uses a single name for both 32- and 64-bit operations.)

### OpenCL

- `atomic_add` or `atom_add`
- `atomic_sub` or `atom_sub`
- `atomic_inc` or `atom_inc`
- `atomic_dec` or `atom_dec`
- `atomic_xchg` or `atom_xchg`
- `atomic_cmpxchg` or `atom_cmpxchg`
- `atomic_max` or `atom_max`
- `atomic_min` or `atom_min`

### C++ AMP

- `atomic_fetch_add`
- `atomic_fetch_sub`
- `atomic_fetch_inc`
- `atomic_fetch_dec`
- `atomic_exchange`
- `atomic_compare_exchange`
- `atomic_fetch_max`
- `atomic_fetch_min`
atomic_and or atom_and  atomic_fetch_and
atomic_or or atom_or  atomic_fetch_or
atomic_xor or atom_xor  atomic_fetch_xor

**Host Functionality**

OpenCL 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

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_context</td>
<td>accelerator_view</td>
</tr>
<tr>
<td>cl_device_id</td>
<td>accelerator</td>
</tr>
<tr>
<td>cl_command_queue</td>
<td>Automatic mode of accelerator_view provides some of the functionality.</td>
</tr>
<tr>
<td>cl_program</td>
<td>No equivalent is necessary. The kernel code is part of the C++ source.</td>
</tr>
<tr>
<td>cl_kernel</td>
<td>No equivalent is necessary. The kernel code is part of the C++ source.</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>C++ AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>clCreateContextFromType</td>
<td>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.</td>
</tr>
<tr>
<td>clGetContextInfo</td>
<td>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.</td>
</tr>
<tr>
<td>clCreateProgramWithSource</td>
<td>No equivalent is necessary. The kernel code is part of the C++ source.</td>
</tr>
<tr>
<td>clCreateProgramWithBinary</td>
<td>No equivalent is necessary. The kernel code is part of the C++ source.</td>
</tr>
<tr>
<td>clBuildProgram</td>
<td>No equivalent is necessary. The kernel code is part of the C++ source.</td>
</tr>
<tr>
<td>clCreateKernel</td>
<td>No equivalent is necessary. The kernel code is part of the C++ source.</td>
</tr>
<tr>
<td>clCreateCommandQueue</td>
<td>Use features of accelerator_view for similar functionality.</td>
</tr>
<tr>
<td>clCreateBuffer</td>
<td>Use the array or array_view constructor. See notes on buffers above.</td>
</tr>
<tr>
<td>clEnqueueReadBuffer, clEnqueueWriteBuffer</td>
<td>Use the copy function, or the synchronize method to synchronize an array_view with the host. Implicit synchronization is often sufficient.</td>
</tr>
<tr>
<td>clSetKernelArg</td>
<td>No equivalent is necessary. The kernel code is part of the C++ source. Normal C++ capture rules apply where only arrays and textures can be captured by reference; everything else must be captured by value.</td>
</tr>
<tr>
<td>clEnqueueNDRangeKernel</td>
<td>parallel_for_each</td>
</tr>
<tr>
<td>clRelease*</td>
<td>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.</td>
</tr>
</tbody>
</table>
Closing Thoughts

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

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