Microsoft announced new technology for GPGPU called C++ Accelerated Massive Parallelism - C++ AMP (http://blogs.msdn.com/b/nativeconcurrency/archive/2011/09/13/c-amp-in-a-nutshell.aspx).

It accelerates the execution of C++ code by taking advantage of the GPUs present on video cards with DirectX11 support. The main structure of C++ AMP is influenced by its main competitors like OpenCL and CUDA. But unlike OpenCL and CUDA, that are more oriented in C code, C++ AMP looks like STL library with new C++11 support. It is available in Visual Studio 2011 Beta and become a part of the existing concurrency namespace (Parallel Patterns Library –PPL and its Concurrency Runtime – ConcRT).

This paper assumes that readers have some experience with GPGPU programming. First, we will compare basic primitive data types and restrictions of both technologies. Then we will present the results of the GPU performance analysis in application to handling specific algorithms of the Point Cloud Library (PCL). (More on PCL at pointcloud.org)

1. Data types and functions

The following chart shows the data types supported by both CUDA and C++ AMP (see Table 1.)

Code primitives	CUDA	C++ AMP
Basic structures (data of type T that can be transferred to GPU)	T*	array<T> array_view<T>
Indexing (technique that give you control of accessing data in threads)	blockDim blockIdx gridDim threadIdx	idx.global idx.local idx.tile idx.tile_origin
Shared memory(fast memory on the chip)	__shared__	tile_static
Synchronization primitives	__syncthreads();	idx.barrier.wait();

Table 1

In addition to the basic data types C++ AMP has a library of mathematical functions that resides in two namespaces precise_math and fast_math. The chart below compares the capabilities of CUDA and C++ AMP in regards to the usage of the mathematical function:

N	CUDA	C++ AMP
1	can only call functions that have the ***__device__*** clause. Execution of kernel code has special syntax ***kernel<<<N>>>();***	can only call functions that have ***restrict(amp)*** clause. Execution of kernel code is done using ***parallel_for_each*** function.
2	Kernel can be function or functor, but not lambda.	Kernel can be function, functor or lambda.
3	References and multiple-indirection pointers are supported.	References and single-indirection pointers are supported only as local variables and functions.
4	Recursion is supported.	Recursion is not supported.
5	Doesn't support STL containers.	Can be easily integrated with STL.
6	Non-POD types, such as classes with virtual functions, are supported.	Everything must be of a POD type.
7	Exceptions are not allowed.	
8	No asm declarations.	

Table 2

The items 2 and 3 (see Table 2) are not critical, since C++ AMP has own wrappers (array, array_view) around user data, so there is no need in pointer to pointer declarations.

2. Tasks and syntax

Every problem that can be parallelized on GPU always passes following phases:

1. Transferring data from CPU to GPU.
2. Calling kernel for each thread.
3. Transferring data from GPU to CPU.

The table below shows a few basic operations to perform GPGPU computations using CUDA and C++ AMP.

Phase	CUDA	C++ AMP
Copying data to GPU	***cudaMemcpy***	***copy***
Calling kernel/shader	***kernel<<<…>>>(…)*** ***cudaDeviceSynchronize***	***parallel_for_each*** ***flush*** ***wait***
Copying results back to CPU	***cudaMemcpy***	***copy***

Table 3

To compare the syntax let's write simple program for vector sum computation.

CUDA

```
__global__ void add_kernel(int *A, int *B, int *C)
{
        int tid = blockIdx.x;
        C[tid] = A[tid] + B[tid];
}

void add(const std::vector<int>& v1, const
std::vector<int>& v2, std::vector<int>& v3)
{
        int *dev_A = nullptr;
        int *dev_B = nullptr;
        int *dev_C = nullptr;

        size_t N = v1.data();
        v3.resize(N);

        cudaMalloc((void**)&dev_A,N*sizeof(int) );
        cudaMalloc((void**)&dev_B,N*sizeof(int) );
        cudaMalloc((void**)&dev_C,N*sizeof(int) );

        cudaMemcpy(dev_A,v1.data(),N*sizeof(int),cuda
MemcpyHostToDevice);
        cudaMemcpy(dev_B,v2.data(),N*sizeof(int),cuda
MemcpyHostToDevice);

        add_kernel<<<M>>>(dev_A,dev_B,dev_C);
        cudaMemcpy(v3.data(),dev_C,N*sizeof(int),cuda
MemcpyDeviceToHost);

        cudaFree(dev_A);
        cudaFree(dev_B);
        cudaFree(dev_C);
}
```

AMP C++

```
void add(const std::vector<int>& v1, const std::vector<int>&
v2, std::vector<int>& v3)
{
        array_view<const int> av1(v1.size(), v1);
        array_view<const int> av2(v2.size(), v2);
        array_view<int> av3(v3.size(), v3);

        parallel_for_each(av3.grid, [=] (index<1> idx)
restrict(amp){av3[idx] = av1[idx] + av2[idx]; });

        av3.synchronize();
}
```

Table 4

Both languages extend C++ with special keywords. For CUDA the syntax is "<<< >>>" and AMP C++ is using the keyword restrict(amp). It is obvious that code written on AMP C++ is less cluttered and hence easier to read than on CUDA. Moreover, AMP C++ can be easily integrated with STL. But how fast is it?

3. Performance comparison

To compare performance between CUDA and C++ AMP we are going to use PCL, which has already some parts parallelized using CUDA technology. We selected the algorithm that is particularly heavy on the floating-point computation. As input, the algorithm takes point cloud data and returns triangle normals as the output. Existing implementation on CUDA can be found at PCL\gpu\cuda\features directory.

Test environment

GeForce GT 520, Asus P5KPL-VM, Intel Core2 Duo E6550 @ 2.33, 4 GB DDR2-800 @400 MHz, Windows 7 Professional 64-bit OS

Results

The following are the time measurements for the phases described in table 3.

Copying to GPU:

N	CUDA (ms)	AMP C++ (ms)
10000	77.3	67.2
50000	76.9	68.2
100000	75.7	70.6
200000	76.2	70
400000	77.6	73.4

Calling kernel:

N	CUDA (ms)	AMP C++ (ms)
10000	0.075	26.5
50000	0.34	25.8
100000	0.33	24.3
200000	0.36	27.9
400000	0.41	33.3

Copying to CPU:

N	CUDA (ms)	AMP C++ (ms)
10000	2.02	3.4
50000	6.4	5
100000	8.4	8.4
200000	14	14.3
400000	25.8	25.6

4. Conclusions

In order to get maximum performance using GPU, developers need to understand how the task can be mapped into GPUs and how to program using API. C++ AMP is relatively new technology, which is built around DirectX11 DirectCompute and has some performance lags compare to 5-years old CUDA technology. We hope that the performance issues will be resolved with Visual Studio 11 RTM. As for the syntax, Microsoft did a great job on C++ extension by introducing STL like classes and STL integration, and only one additional keyword- restrict(amp). Developers can easily rewrite existing code to improve performance of the execution of their algorithms.

5. References

1) Wong, H., M. Papadopoulou, et al. (2010). Demystifying GPU microarchitecture through microbenchmarking. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2010, IEEE.
2) Microsoft, “C++ AMP: language and Programming Model”