AN OVERVIEW OF ACCELERATED PARALLELISM WITH C++ AMP
I’m NOT on the C++ AMP Team

I Just wrote the book
For Fun Not Money
What's it all about? …

Introduction
What About The Future?

- CPUs and GPUs coming closer together…
  - …rapidly evolving space

- C++ AMP is designed as a mainstream solution for data parallel kernels.

Not only for today, but also for tomorrow.
C++ AMP Platforms Goal
(not current reality)

- Windows Azure
- Windows Desktop
- Windows Server
- Windows HPC Server
- Windows Phone
- Windows RT
- Windows Embedded
- Xbox
C++ AMP

- STL-like library for multidimensional data
- Part of Visual C++
- Visual Studio integration
- Microsoft’s implementation builds on Direct3D
- An open specification
  - Intel’s Shevlin Park project implements a POC of C++ AMP on top of OpenCL
Portability

- Target GPUs with DirectX 11 drivers
  - NVIDIA GPUs
  - AMD GPUs (and APU s)
  - Intel GPUs (Ivy Bridge and later)
  - ARM GPUs from various IHVs (soon, e.g. see Mali design)
- Fallback to CPU when no capable GPU present
  - AMD and Intel CPUs (multi-core and SSE)
  - ARM CPUs (multi-core and NEON)
- Windows
  - HPC Server, Server, Desktop, Tablets (x86 and ARM)
- Other platforms/hardware through open specification
Performance

- Much faster than CPU multi-core solutions
  - Many samples in the book and online
- Comparable with other GPU approaches
Productivity

- Elements of productivity typically forgotten
  - Acquisition, Support, Deployment
- Lower the barrier to entry AND write less lines of code
  - Blur the line between “host” and “device”
    - One compiler for both, one code file for both, one outer function for both
    - Don’t have to manage data transfers explicitly
    - Don’t have to learn about accelerators until you need to…
  - Don’t have to learn about thread groups/blocks until you need to
    - …and then it is a seamless addition to learn and use
  - C++ AMP is modern C++!
    - No explicit resource management, No stateless global functions, No raw memory pointers, use of lambdas at the API surface
We’re developers so let’s focus on…

Productivity
Improving Time to Insight

Think

Run

Code
Improving Time to Insight

Think → Code → Run
Improving Time to Insight

Think

Run

Code
C++ AMP in five (ish) minutes…

The Quick Tour
Containers

array<T, N>
array_view<T, N>

Container descriptors
index<N>
extent<N>
extent<N> and index<N>

- **index<N>** - an N-dimensional point
- **extent<N>** - size of an N-dimensional space

```
index<1> i(2);
index<2> i(0,2);
index<3> i(2,0,1);
```

```
extent<1> e(6);
extent<2> e(3,4);
extent<3> e(3,2,2);
```

- rank N can be any number <=128
array<T, N>

- Multi-dimensional array of rank N with element type T
- Container whose storage lives on a specific accelerator
- Capture by reference [&] in the lambda
- Explicit copy

```cpp
vector<int> vec(8 * 12);
extent<2> ext(8, 12);
array<int, 2> arr(ext);
copy(vec.begin(), vec.end(), arr);
```
array_view<T, N>

- View on existing data on the CPU or GPU
- Dense in least significant dimension
- Of element T and rank N
- Requires extent
- Rectangular
- Access anywhere (implicit sync)
- Nearly identical interface to array_view<T, N>

```cpp
std::vector<int> vec(2 * 5);
extent<2> ext(2, 5);
array_view<int, 2> arr(ext, vec);
```
Keyword: restrict( . . . )

- Applies to functions (including lambdas)

- restrict(…) informs the compiler to enforce language restrictions
  - e.g., target-specific restrictions, optimizations, special code-gen

- In 1st release only implements two options:
  - cpu – the implicit default
  - amp – checks that the function conforms to C++ AMP restrictions
restrict(amp) restrictions

- Can only call other \textit{restrict(amp)} functions
- All functions must be inlinable
- Only amp-supported types
  - int, unsigned int, float, double, bool¹
  - structs & arrays of these types
- Pointers and References
  - Lambdas cannot capture by reference¹, nor capture pointers
  - References and single-indirection pointers supported only as local variables and function arguments
restrict(amp) restrictions

- **No**
  - recursion
  - 'volatile'
  - virtual functions
  - pointers to functions
  - pointers to member functions
  - pointers in structs
  - pointers to pointers
  - bitfields

- **No**
  - goto or labeled statements
  - throw, try, catch
  - globals or statics
  - dynamic_cast or typeid
  - asm declarations
  - varargs
  - unsupported types
    - e.g. char, short, long double
Algorithm(s)
parallel_for_each

- Executes the kernel for each point in the extent
- As-if synchronous in terms of visible side-effects

```cpp
std::vector<int> arr(100000);
array_view<int, 1> arr_av(input.size(), input);
parallel_for_each(arr_av.extent, [ ](index<N> idx)
    restrict(amp)
{
    // kernel code ...}
```
```cpp
#include <ppl.h>
using namespace concurrency;

static float Func(float val)
{
    ... 
}

std::vector<float> arr(10000);
std::iota(begin(arr), end(arr), 1.0f);

parallel_for_each(begin(arr), end(arr),
    [=](float& v)
    {
        v = Func(v);
    });
```
#include <amp.h>
using namespace concurrency;

static float Func(float val) restrict(cpu, amp) 
{ ... }

// Initialize arr ...
array_view<float> arr_av(arr.size(), arr);

parallel_for_each(arr_av.extent, [=](index<1> idx) restrict(amp) 
{ 
    arr_av[idx] = Func(arr_av[idx]);
});

for(int i = 0; i < arr_av.extent[0]; ++i)
    std::cout << arr_av[i] << std::endl;
```cpp
#include <amp.h>
using namespace concurrency;

static float Func(float val) restrict(cpu, amp)
{ ... }

// Initialize arr ...
array_view<float> arr_av(arr.size(), arr);

parallel_for_each(arr_av.extent, [=](index<1> idx) restrict(amp)
{
    arr_av[idx] = Func(arr_av[idx]);
});

for(int i = 0; i < arr_av.extent[0]; ++i)
    std::cout <= arr_av[i] <= std::endl;
```
Using Tiled Memory to Improve Performance
Keyword: tile\_static

- The tile\_static storage class
  - Second addition to the C++ language
  - Reflects hardware memory hierarchy

- Within the tiled parallel\_for\_each lambda use
  - tile\_static for local variables
    - indicates that the variable is allocated in fast cache memory
      - i.e. shared by each thread in a tile of threads
    - only applicable in restrict(amp) functions
array_view<const float> arr_av(arr.size(), arr);
std::vector<float> avg(arr.size() - 2);
array_view<float> avg_av(avg.size(), avg); extent<1>

avg_av.discard_data();
parallel_for_each(avg_av.extent, [=](index<1> idx)
    restrict(amp)
{
    const int cIdx = idx[0] + 1;

    avg_av[cIdx - 1] = (arr_av[idx] +
                        arr_av[idx + 1] +
                        arr_av[idx + 2]) / 3;
});
static const int tileSize = 4; // 256

tiled_extent<tileSize> computeDomain = avg_av.extent;
computeDomain = computeDomain.pad();

parallel_for_each(computeDomain,
    [](tiled_index<tileSize> idx) restrict(amp)
    {
        const int gIdx = idx.global[0];
        const int tIdx = idx.local[0];

        // ...
    });
parallel_for_each(computeDomain, 
    [=](tiled_index<tileSize> idx) restrict(amp) 
    {
        const int gIdx = idx.global[0];
        const int tIdx = idx.local[0];

        tile_static float local[tileSize + 2];

        local[tIdx + 1] = PaddedRead(arr_av, gIdx);
        if (tIdx == 0)
            local[0] = PaddedRead(arr_av, gIdx - 1);
        if (tIdx == (tileSize - 1))
            local[tileSize + 1] = 
                PaddedRead(arr_av, gIdx + 1);

        idx.barrier.wait();
        // ...
parallel_for_each(computeDomain,
    [=](tiled_index<tileSize> idx) restrict(amp)
    {
        const int gIdx = idx.global[0];
        const int tIdx = idx.local[0];

        // ...

        float val = (local[tIdx] +
                     local[tIdx + 1] +
                     local[tIdx + 2]) / 3;
        PaddedWrite(avg_av, gIdx - 1, val);
    });
template <typename T>
T PaddedRead(const array_view<const T, 1>& A, int idx)
restrict(cpu, amp)
{
}

template <typename T>
void PaddedWrite(const array_view<T, 1>& A, int idx, T val)
restrict(cpu, amp)
{
    if (A.extent.contains(index<1>(idx))) A[idx] = val;
}
Usual Performance Rules Apply

- Measure and understand your goals
- Consider your whole algorithm
- Use off the shelf libraries – C++ AMP Algorithms Library
- Minimize or overlap memory transfers to and from the GPU
- Coalesce global memory accesses
- Take Advantage of tile_static (local memory)
- Avoid bank conflicts in tile_static memory
- Avoid branching within kernels
Using GPUs and CPUs together

Braided Parallelism
Cartoonizer Demo

- Real time video processing on multiple GPUs
- Edge detection and color smoothing
- Pipelined approach using both CPU and GPU at different stages
Video Processing Pipeline
parallel_for_each(begin(m_processors), end(m_processors),
  [=](std::shared_ptr<IFrameProcessor>& proc) {
    ImageInfoPtr pInfo = nullptr;
    do {
      pInfo = receive(m_inputBuffer);
      CartoonizeImage(pInfo, proc, ...);
      asend(..., pInfo);
    } while (nullptr != pInfo);
  });

extent<2> computeDomain(...);
parallel_for_each(computeDomain,
  [=, &srcFrame, &destFrame](index<2> idx) restrict(amp) {
  SimplifyIndex(srcFrame, destFrame, idx, ...);
});
C++ AMP Tools

- Visual Studio 2012
  - Visual C++ with C+11 support
  - CPU and GPU Debugger
  - CPU and GPU Profiler
  - IDE support; Intellisense
Visual Studio IDE
Learn C++ AMP...

Book / Source Code / Blogs:
http://www.gregcons.com/cppamp

Courses:
http://www.acceleware.com/
- Apr 23 - C++ AMP in Seattle, WA
- Sep 10 - C++ AMP in Boston, MA
More C++ AMP

C++ AMP Team Blog
http://blogs.msdn.com/b/nativeconcurrency/

C++ AMP Forum

Open Specification:
http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx
Questions?