C++ AMP: Accelerated Massive Parallelism in Visual C++

Kate Gregory
Gregory Consulting
www.gregcons.com/kateblog, @gregcons
C++ is the language for performance

- If you need speed at runtime, you use C++
- Frameworks and libraries can make your code faster
  - Eg PPL: use all the CPU cores
  - With little or no change to your logic and code
- Experienced C++ developers are productive in C++
  - Don’t want to go back to C or C-like language
  - Enjoy the tool support in Visual Studio
- Many C++ developers value portability
  - Write standard C++, compile with anything
  - Use portable libraries, run anywhere
  - Even in all-Microsoft universe, simple deployment is important
What is C++ AMP?

• Accelerated Massive Parallelism
  • Run your calculations on one or more accelerators
    • Today, GPU is the accelerator you use
    • Eventually: other kinds of accelerators

• Write your whole application in C++
  • Not a “C-like” language or a separate resource you link in
  • Use Visual Studio and familiar tools
  • Speed up 20x, 50x, or more

• Basically a library
  • Comes with Visual Studio 2012 and 2013, included in vcredist
  • Spec is open – other platforms/compilers can implement it too
Agenda

• Why? Hardware Review
• C++ AMP Fundamentals
• A Few Details
• Debugging and Visualizing
• Call to Action
Wait, Why?

• Until 2005 “Free Lunch”
  • Clock speed increased every year
  • Single threaded performance increased every year
  • Apps got faster for free

• After 2005 “No More Free Lunch”
  • Clock speeds are not increasing that fast anymore
  • Instead, CPU’s get more powerful every year by adding more cores
  • Single threaded performance is now increasing much slower If at all

• Want to get faster?
  • Use more cores
CPUs vs GPUs today

**CPU**
- Low memory bandwidth
- Higher power consumption
- Medium level of parallelism
- Deep execution pipelines
- Random accesses
- Supports general code
- Mainstream programming

**GPU**
- High memory bandwidth
- Lower power consumption
- High level of parallelism
- Shallow execution pipelines
- Sequential accesses
- Supports data-parallel code
- Niche programming

images source: AMD
CPU Parallelism

• Vectorization (SIMD, SSE, AVX, ...)
  • Visual Studio 2012 and 2013 can auto-vectorize and auto-parallelize your loops

• Multithreading:
  • Microsoft PPL (Parallel Patterns Library)
  • Intel TBB (Threading Building Blocks) (compatible interface with PPL)
GPU Parallelism

- **CUDA**: If you want to optimally use NVidia GPUs
- **OpenCL**: If you want to optimally use AMD GPUs
- **DirectCompute**: Uses HLSL, looks like C
- All are C-like, not truly C++
  - no type safety, genericity, ...
  - only CUDA is becoming similar to C++

- **Hard**
  - need to learn multiple technologies to optimally target multiple devices...
Speed Changes Everything

- 2-3x faster is “just faster”
  - Do a little more, wait a little less
  - Doesn’t change how users really work
- 5-10x faster is “significant”
  - Worth upgrading
  - Worth re-writing (parts of) your applications
- 100x+ faster is “fundamentally different”
  - Worth considering a new platform
  - Worth re-architecting your applications
  - Makes completely new applications possible
C++ AMP

- Vendor independent (NVidia, AMD, ...)
- Abstracts “accelerators” (GPU’s, APU’s, ...)
- Only requirement: DirectX 11
  - Fallback to WARP if no hardware GPU’s available
- Future support for other accelerators
  - FPGA’s, off-site cloud computing...
- Support heterogeneous mix of accelerators!
  - Example: both an NVidia and AMD GPU in your system splitting a workload
C++ AMP is fundamentally a library

- Comes with Visual C++ 2012 and 2013
- `#include <amp.h>
- Namespace: concurrency
- New classes:
  - array, array_view
  - extent, index
  - accelerator, accelerator_view
- New function(s): `parallel_for_each()`
- New (use of) keyword: `restrict`
  - Asks compiler to check your code is ok for GPU (DirectX)
Agenda

• Why? Hardware Review
• C++ AMP Fundamentals
• A Few Details
• Debugging and Visualizing
• Call to Action
parallel_for_each

• Entry point to the library
• Takes number (and shape) of threads needed
• Takes function or lambda to be done by each thread
  • Must be restrict(amp)
  • Lambda must capture everything by value, except concurrency::array objects
• Sends the work to the accelerator
  • Scheduling etc handled there
• Returns – no blocking/waiting
#include <amp.h>

using namespace concurrency;

void AddArrays(int n, int * pA, int * pB, int * pSum)
{
    for (int i=0; i<n; i++)
    {
        pSum[i] = pA[i] + pB[i];
    }
}

void AddArrays(int n, int * pA, int * pB, int * pSum)
{
    array_view<int, 1> a(n, pA);
    array_view<int, 1> b(n, pB);
    array_view<int, 1> sum(n, pSum);

    parallel_for_each(
        sum.extent,
        [=](index<1> i) restrict(amp) {
            sum[i] = a[i] + b[i];
        });
}
Basic Elements of C++ AMP coding

```cpp
void AddArrays(int n, int * pA, int * pB, int * pSum)
{
    array_view<int,1> a(n, pA);
    array_view<int,1> b(n, pB);
    array_view<int,1> sum(n, pSum);
    parallel_for_each(
        sum.extent,
        [=](index<1> i) restrict(amp)
        {
            sum[i] = a[i] + b[i];
        });
}
```

- **array_view**: wraps the data to operate on the accelerator
- **restrict(amp)**: tells the compiler to check that this code conforms to C++ AMP language restrictions
- **parallel_for_each**: execute the lambda on the accelerator once per thread
- **extent**: the number and shape of threads to execute the lambda
- **index**: the thread ID that is running the lambda, used to index into data
- **array_view variables captured and associated data copied to accelerator (on demand)**
The lambda

• Executes on the accelerator in parallel with whatever CPU code follows `parallel_for_each()` until a synchronization point is reached

• Synchronization:
  • Manually when calling `array_view::synchronize()`
    • Good idea, because you can handle exceptions gracefully
  • Automatically when CPU code uses structure wrapped by `array_view`
    • Not recommended, because you might lose error information if there is no try/catch block catching exceptions at that point
  • Automatically when `array_view` goes out of scope
    • Dangerous, errors will be ignored silently because destructors are not allowed to throw exceptions
extent<1> e(5);

extent<2> f(2,4);

extent<3> g(2,4,3);
index<1> i(3);

index<2> j(1,2);

index<3> k(0,1,2);
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)

```cpp
vector<int> v(10);
extent<2> e(2,5);
array_view<int,2> a(e, v);

//above two lines can also be written
//array_view<int,2> a(2,5,v);
index<2> i(1,3);
int o = a[i]; // or a[i] = 16;
//or int o = a(1, 3);
```
array_view

- Read-only buffer:
  - `array_view<const int, 2> av(...);`
  - Only copies data from the CPU to the accelerator at the start, not back to the CPU at the end

- Write-only buffer:
  - `array_view<int, 2> av(...);
    av.discard_data();`
  - Only copies data from the accelerator to the CPU at the end, not to the accelerator at the start
Demo

Matrix Multiplication
Matrix Multiplication

\[
C_{00} = A_{00} \cdot B_{00} + A_{01} \cdot B_{10} + A_{02} \cdot B_{20} + A_{03} \cdot B_{30}
\]
Agenda

• Why? Hardware Review
• C++ AMP Fundamentals
• A Few Details
• Debugging and Visualizing
• Call to Action
restrict(amp) restrictions

• Can only call other restrict(amp) functions
• All functions must be inlinable
• Only amp-supported types
  • int, unsigned int, float, double, bool
  • structs & arrays of these types
• Pointers and References
  • Lambdas cannot capture by reference, 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
restrict()

• restrict() is really part of the signature
  • Can differentiate overloads

• Compare:
  • float func1(float) restrict(cpu, amp);
    • Can run on both CPU and C++ AMP accelerators
  • float func2(float);
    • General code – not ok to call from parallel_for_each
  • float func2(float) restrict(amp);
    • AMP-specific code – ok to call from parallel_for_each
array<T,N>

- Multi-dimensional array of rank N with element T
- Container whose storage lives on a specific accelerator
- Capture by reference [&] in the lambda
- Explicit copy
- Nearly identical interface to array_view<T,N>

```cpp
class vector
{
public:
    vector(int n); // constructor
    // other member functions...
private:
    int* data; // storage
};

class extent
{
public:
    extent(int n); // constructor
    // other member functions...
private:
    int size; // extent
};

class accelerator
{
public:
    accelerator(); // constructor
    // other member functions...
private:
    // implementation details...
};

class array
{
public:
    array(extent e, accelerator acc); // constructor
    // other member functions...
private:
    // implementation details...
};

int main()
{
    vector<int> v(8 * 12);
    extent<2> e(8, 12);
    accelerator acc = ...
    array<int,2> a(e,acc.default_view);
    copy_async(v.begin(), v.end(), a);
    parallel_for_each(e, [&](index<2> idx)
    restrict(amp)
    {
        a[idx] += 1;
    });
    copy(a, v.begin());
    return 0;
}
```
Tiling

• Rearrange algorithm to do the calculation in tiles
• Each thread in a tile shares a programmable cache
  • tile_static memory
  • Access 100x as fast as global memory
  • Excellent for algorithms that use each piece of information again and again
• Overload of parallel_for_each that takes a tiled extent
Race Conditions in the Cache

• Because a tile of threads shares the programmable cache, you must prevent race conditions
  • Tile barrier can ensure a wait

• Typical pattern:
  • Each thread does a share of the work to fill the cache
  • Then waits until all threads have done that work
  • Then uses the cache to calculate a share of the answer
Agenda

• Why? Hardware Review
• C++ AMP Fundamentals
• A Few Details
• Debugging and Visualizing
• Call to Action
Visual Studio 2013 AMP Support

• Debugging
  • Everything you had before, plus:
    • GPU Threads
    • Parallel Stacks
    • Parallel Watch
• Visualizing
Debugging

- GPU breakpoints are supported
- On Windows 8 and 7, no CPU/GPU simultaneous debugging possible
- Choose the GPU Only debugging option
Debugging

- Windows 8.1 and VC++2013 support simultaneous CPU/GPU debugging:
  - Uses the WARP accelerator
Values, Call Stacks, etc

```cpp
array_view<float, 2> mB(eB, vB);
array_view<float, 2> mC(eC, vC);
mC.discard_data();

parallel_for_each(extent<2>(eC), [=](index<2> idx) restrict(amp)
{
    float result = 0.0f;
    for(int i = 0; i < mA.extent[1]; ++i)
    {
        index<2> idxA(idx[0], i);
        index<2> idxB(i, idx[1]);

        result += mA[idxA] * mB[idxB];
    }
    result += mA[idx] * mB[idxB];
    mC[idx] = result;
});
mC.synchronize();
```
GPU Threads Window

- Shows progress through the calculation

```
127 tile_static float sA[TS][TS], sB[TS][TS];
128 sA[row][col] = a(tidx.global[0], col + i);
129 sB[row][col] = b(row + i, tidx.global[1]);
130 tidx.barrier.wait();
131 for (int k = 0; k < TS; k++)
132     sum += sA[row][k] * sB[k][col];
133 tidx.barrier.wait();
134 c[tidx.global] = sum;
135 });
136 c.synchronize();
```
Parallel Watch

- Shows values across multiple threads
And more!

- Race Condition Detection
- Parallel Stacks
- Flagging, Filtering, and Grouping
- Freezing and Thawing
- Run Tile to Cursor
Concurrency Visualizer

• Shows activity on CPU and GPU
• Can highlight relative times for specific parts of a calculation
• Or copy times to/from the accelerator

• Comes with Visual Studio 2012
• For Visual Studio 2013, shipped as a free extension
  • Search [www.visualstudiogallery.com](http://www.visualstudiogallery.com) or use Extension Manager inside Visual Studio
Agenda

• Why? Hardware Review
• C++ AMP Fundamentals
• A Few Details
• Debugging and Visualizing
• Call to Action
C++ AMP is...

- C++
  - The language you know
  - Excellent productivity
  - The language you choose when performance matters
- Implemented as (mostly) a library
  - Variety of application types
- Well supported by Visual Studio
  - Debugger
  - Concurrency Visualizer
  - Everything else you already use
- Can be supported by other compilers and platforms
  - Open spec
Learn C++ AMP

- training http://www.acceleware.com/cpp-amp-training
- samples http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx
- spec http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx

http://blogs.msdn.com/nativeconcurrency/
Call to Action

- Get Visual Studio 2013
- Download some samples
- Play with debugger and other tools
- Try writing a C++ AMP application of your own
  - Console (command prompt)
  - Windows
  - Metro style for Windows 8
- Measure your performance and see the difference