

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



Demo

Cartoonizer

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
- Mainstream programming with C++ AMP

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

Hello World: Array Addition

```
void AddArrays(int n, int * pA, int * pB,  
int * pSum)  
{
```

```
    for (int i=0; i<n; i++)
```

```
    {
```

```
        pSum[i] = pA[i] + pB[i];
```

```
    }
```

```
}
```

```
#include <amp.h>
```

```
using namespace concurrency;
```

```
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);
```

```
    for (int i=0; i<n; i++)
```

```
        sum.extent,
```

```
        [=](index<1> i) restrict(amp)
```

```
        {
```

```
            pSum[i] = a[i] + b[i];
```

```
        }
```

```
    );
```

```
}
```

Basic Elements of C++ AMP coding

```
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:
execute the lambda on
the accelerator once
per thread

```
    parallel_for_each(
        sum.extent,
        [=](index<1> i) restrict(amp)
        {
            sum[i] = a[i] + b[i];
        }
    );
```

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: wraps the data to
operate on the accelerator

restrict(amp): tells the compiler to
check that this code conforms to C++
AMP language restrictions

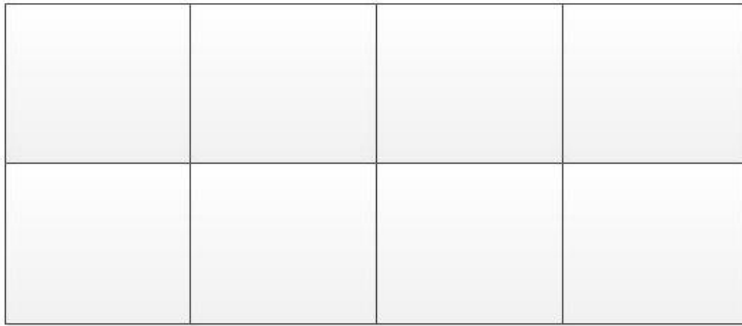
array_view variables captured and
associated data copied to
accelerator (on demand)

The lambda

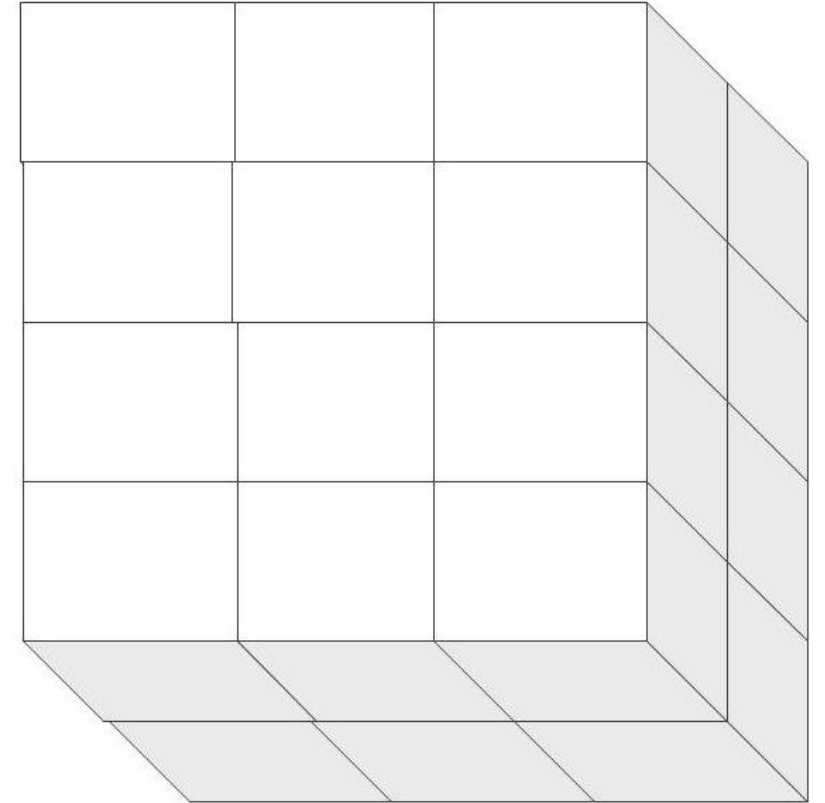
- Executes on the accelerator in parallel with whatever CPU code follows **parallel_for_each()** until a synchronization point is reached
- Synchronization:
 - Manually when calling **array_view::synchronize()**
 - **Good idea**, because you can handle exceptions gracefully
 - Automatically when CPU 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



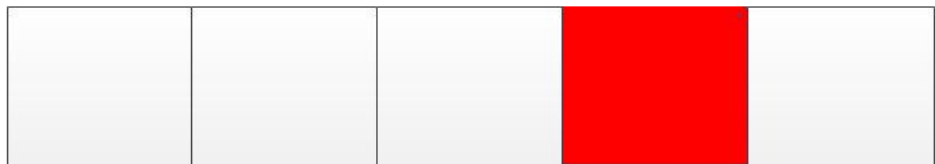
extent<1> e(5);



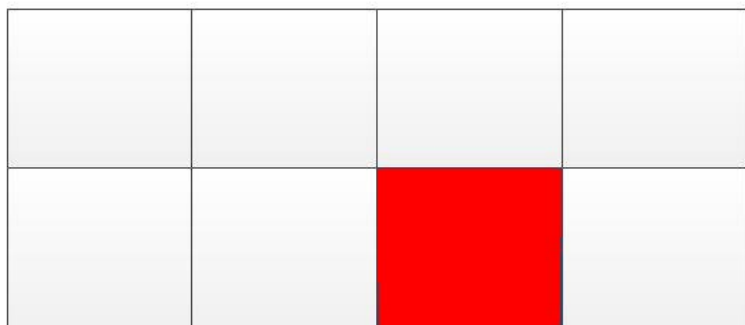
extent<2> f(2,4);



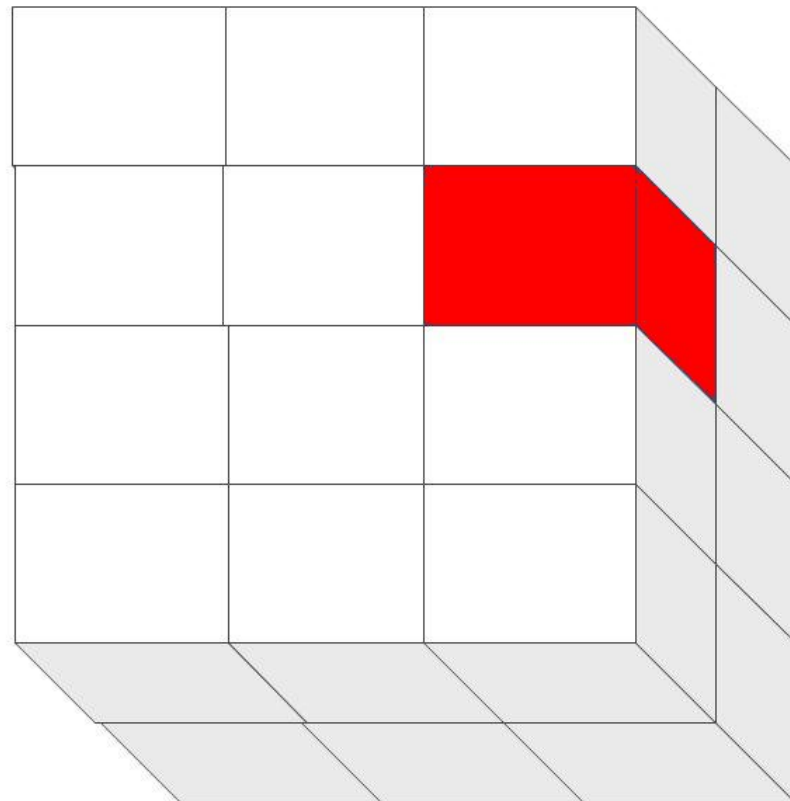
extent<3> g(2,4,3);



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)

```
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

M=2	W=4				N=6					
	A ₀₀	A ₀₁	A ₀₂	A ₀₃	B ₀₀	B ₀₁	B ₀₂	B ₀₃	B ₀₄	B ₀₅
	A ₁₀				B ₁₀					
					B ₂₀					
					B ₃₀					

$$C_{00} = A_{00} * B_{00} + A_{01} * B_{10} \\ + A_{02} * B_{20} + A_{03} * B_{30}$$

M=2	N=6					
	C ₀₀	C ₀₁	C ₀₂	C ₀₃	C ₀₄	C ₀₅
	C ₁₀	C ₁₁	C ₁₂	C ₁₃	C ₁₄	C ₁₅

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>

```
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());
```

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

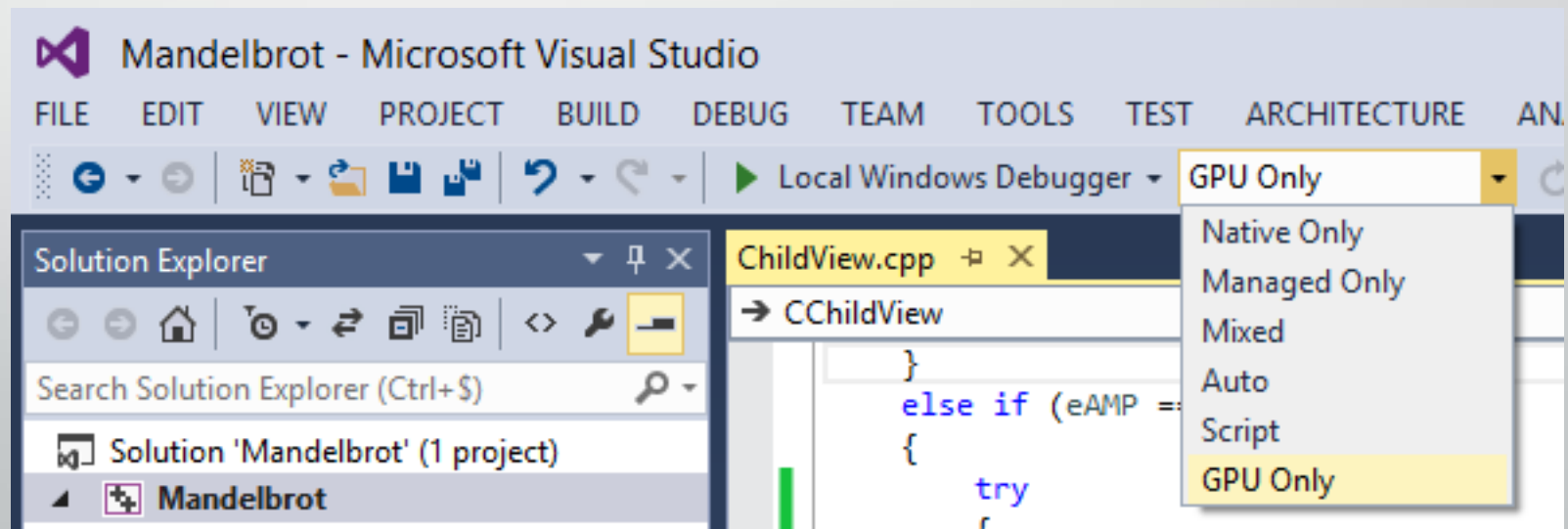
- Why? Hardware Review
- C++ AMP Fundamentals
- A Few Details
- Debugging and Visualizing
- Call to Action

Visual Studio 2013 AMP Support

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

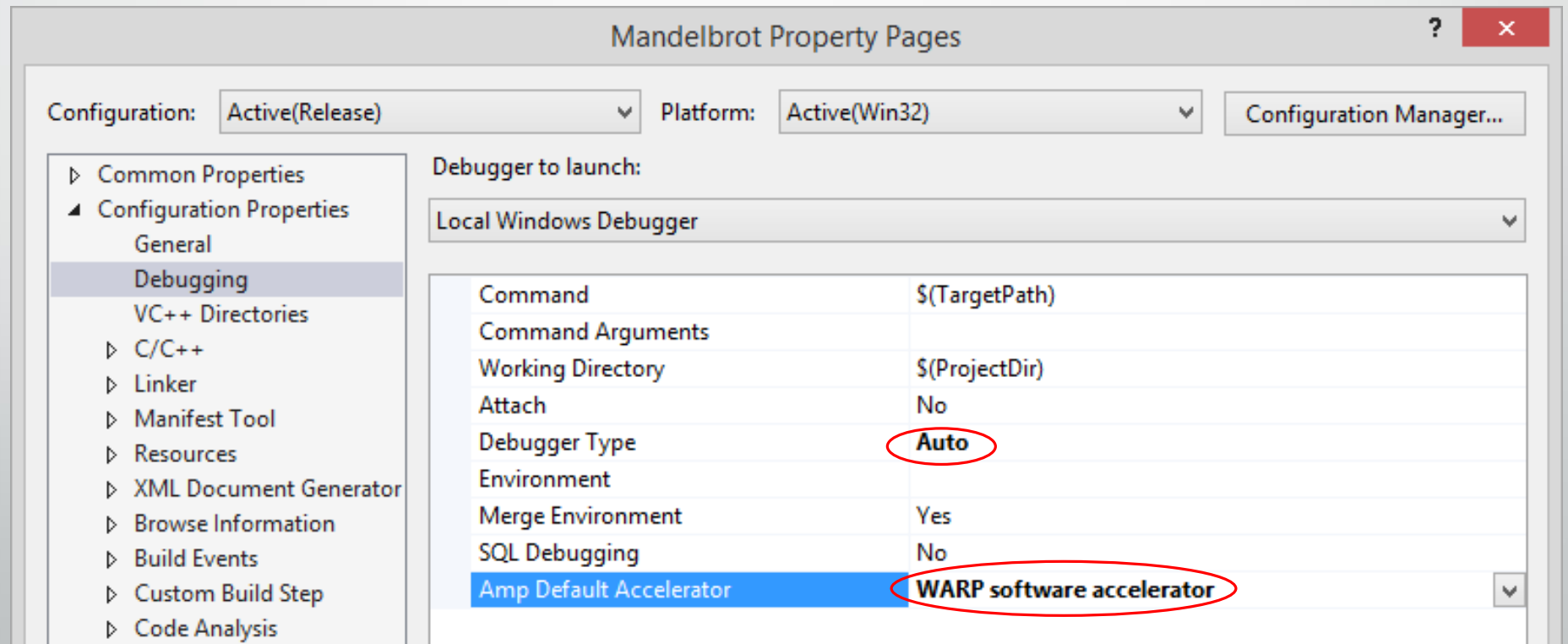
Debugging

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

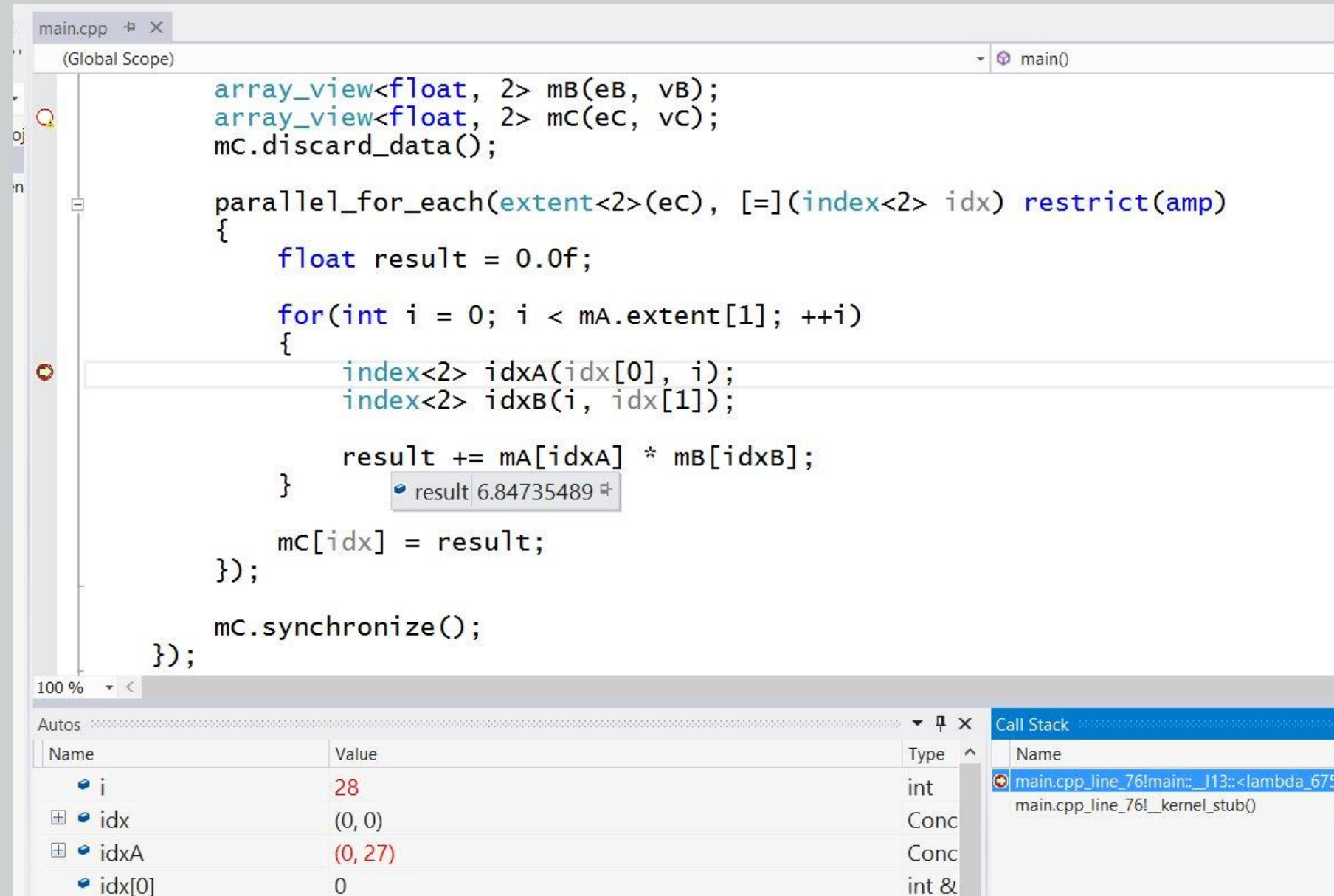


Debugging

- Windows 8.1 and VC++2013 support simultaneous CPU/GPU debugging:
 - Uses the WARP accelerator



Values, Call Stacks, etc



The screenshot shows a C++ IDE with a file named `main.cpp`. The code is in the `(Global Scope)` and `main()` context. The code defines two `array_view<float, 2>` objects, `mB` and `mC`, and a `parallel_for_each` loop. The loop iterates over an `extent<2>(ec)` with a lambda function that calculates a result. A debugger overlay shows the current state of the loop: `i` is 28, `idx` is (0, 0), `idxA` is (0, 27), and `idx[0]` is 0. The result of the calculation is 6.84735489. The call stack shows the current function is `main.cpp_line_76!main::_113::<lambda_675>` and the caller is `main.cpp_line_76!__kernel_stub0`.

```
main.cpp [X]
(Global Scope) [main()]

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];
    }
    mC[idx] = result;
});
mC.synchronize();
});
```

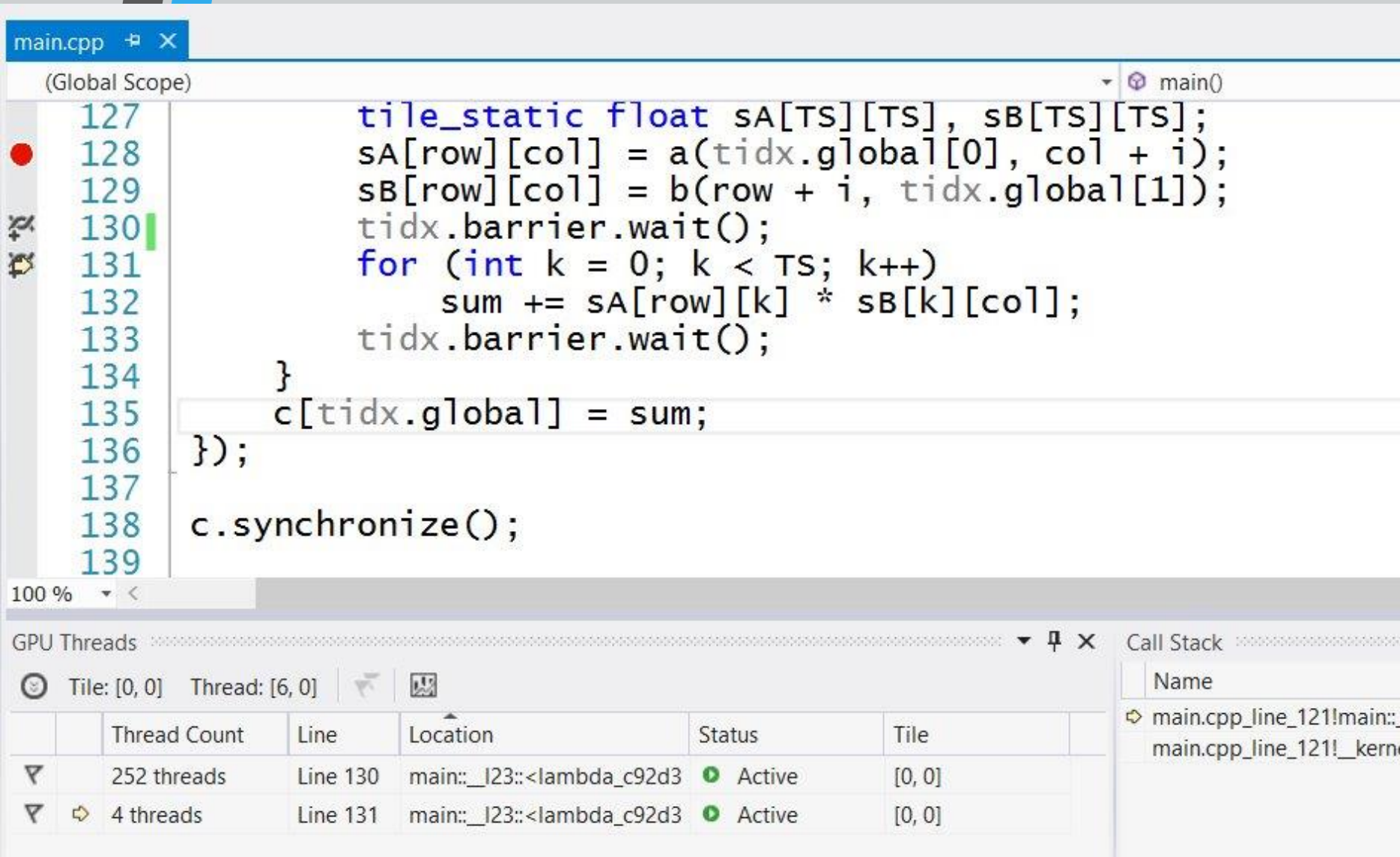
100 %

Name	Value	Type
i	28	int
idx	(0, 0)	Conc
idxA	(0, 27)	Conc
idx[0]	0	int &

result 6.84735489

Name
main.cpp_line_76!main::_113::<lambda_675>
main.cpp_line_76!__kernel_stub0

GPU Threads Window



The screenshot displays a code editor window with a file named `main.cpp`. The code is in C++ and shows a function `main()` with a loop over `i` from 0 to `TS-1`. Inside the loop, there is a `tile_static` block containing a loop over `k` from 0 to `TS-1`. The code calculates the sum of products of elements from two matrices `sA` and `sB` and stores it in `c[tidx.global]`. The `c` array is then synchronized.

```
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 }
135 c[tidx.global] = sum;
136 });
137
138 c.synchronize();
139
```

Below the code editor, the **GPU Threads** window is visible. It shows the current thread configuration: `Tile: [0, 0]` and `Thread: [6, 0]`. The window contains a table with the following data:

	Thread Count	Line	Location	Status	Tile
▼	252 threads	Line 130	main::_l23::<lambda_c92d3	Active	[0, 0]
▼	4 threads	Line 131	main::_l23::<lambda_c92d3	Active	[0, 0]

To the right of the table is a **Call Stack** window showing the current call stack:

- main.cpp_line_121!main::_
- main.cpp_line_121!__kerne

- Shows progress through the calculation

Parallel Watch

- Shows values across multiple threads

Chapter4 (Debugging) - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP

Process: [3884] Chapter4.exe Thread: [0, 0][0, 4] Stack Frame: main::_l23::<lambda_c92d372ddc6bd558>

main.cpp

```
126 {
127     tile_static float sA[TS][TS], sB[TS][TS];
128     sA[row][col] = a(tididx.global[0], col + i);
129     sB[row][col] = b(row + i, tididx.global[1]);
130     tididx.barrier.wait();
131     for (int k = 0; k < TS; k++)
132         sum += sA[row][k] * sB[k][col];
133     tididx.barrier.wait();
134 }
135 c[tididx.global] = sum;
136 });
137
138 c.synchronize();
```

Parallel Watch 1 - ...3b1fe1a>::operator()

	[Tile]	[Thread]	sum
▼	[0, 0]	[0, 0]	5.29245663
▼	[0, 0]	[0, 1]	7.22255516
▼	[0, 0]	[0, 2]	7.24177313
▼	[0, 0]	[0, 3]	7.36989260
▼	[0, 0]	[0, 4]	6.04233170
▼	[0, 0]	[0, 5]	6.45951271
▼	[0, 0]	[0, 6]	8.30030918
▼	[0, 0]	[0, 7]	6.32688856

GPU Threads

Tile: [0, 0] Thread: [0, 4]

	Thread Count	Line	Location	Status	Tile
▼	4 threads	Line 133	main::_l23::<lambda_c92d3	Blocked	[0, 0]
▼	4 threads	Line 133	main::_l23::<lambda_c92d3	Active	[0, 0]
▼	248 threads	Line 130	main::_l23::<lambda_c92d3	Active	[0, 0]

GPU Threads Autos Locals Threads Modules Watch 1

Ready Ln 127 Col 1 Ch 1 INS



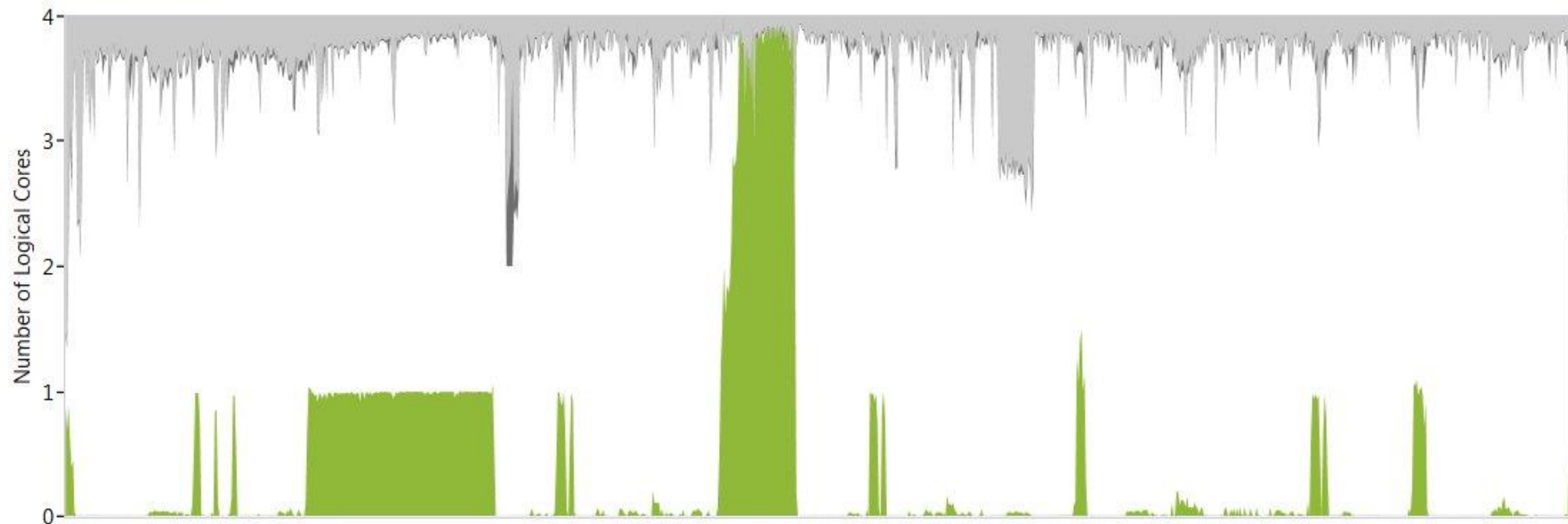
And more!

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

Concurrency Visualizer

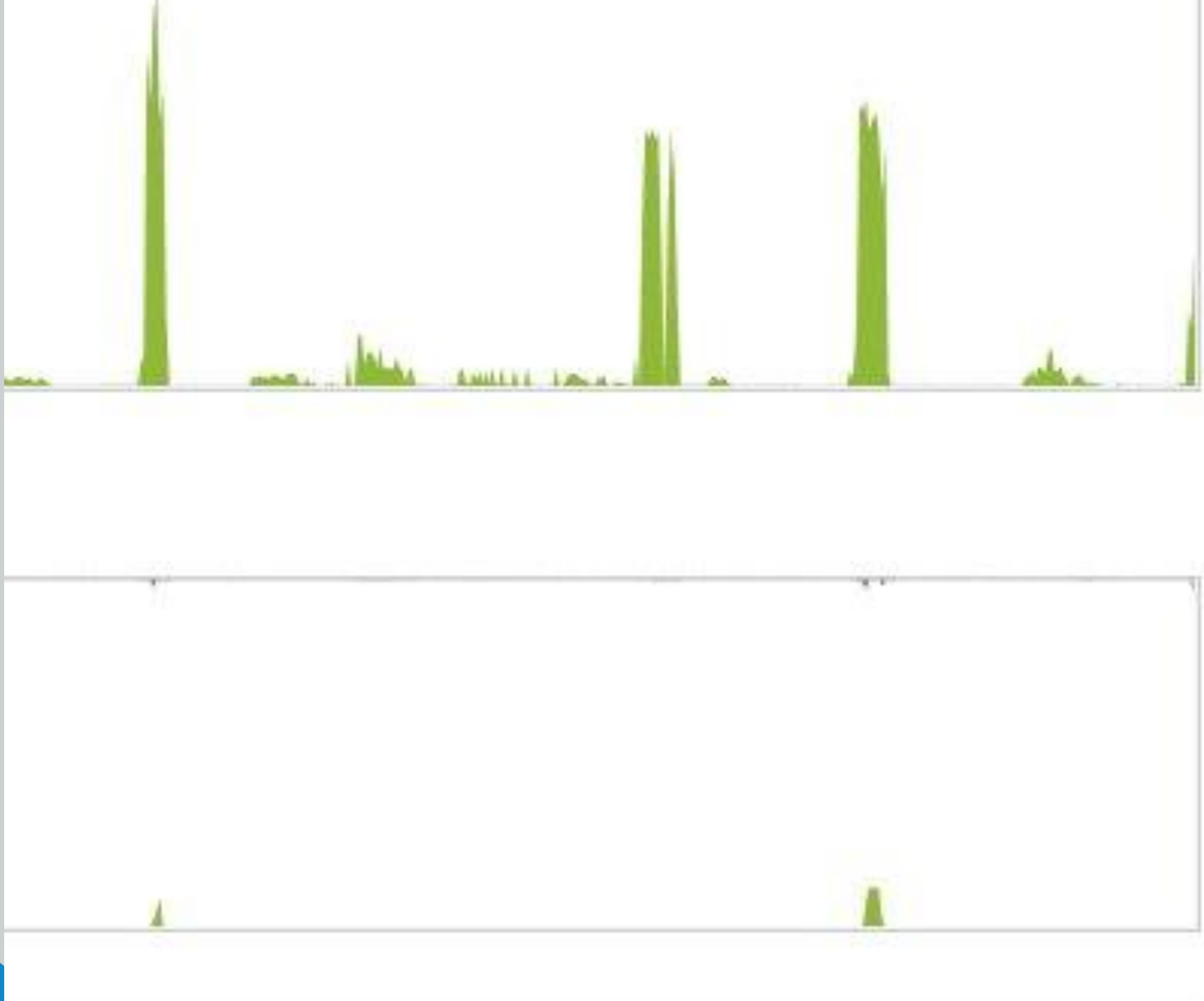
- Shows activity on CPU and GPU
- Can highlight relative times for specific parts of a calculation
- Or copy times to/from the accelerator
- Comes with Visual Studio 2012
- For Visual Studio 2013, shipped as a free extension
 - Search www.visualstudiogallery.com or use Extension Manager inside Visual Studio

CPU Utilization



GPU Activity (DirectX)





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



- book <http://www.gregcons.com/cppamp/>
- training <http://www.acceleware.com/cpp-amp-training>
- videos <http://channel9.msdn.com/Tags/c++-accelerated-massive-parallelism>
- articles <http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/05/c-amp-articles-in-msdn-magazine-april-issue.aspx>
- samples <http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx>
- guides <http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/11/c-amp-for-the-cuda-programmer.aspx>
- spec <http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx>
- forum <http://social.msdn.microsoft.com/Forums/en/parallelcppnative/threads>

<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