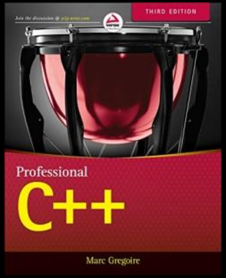
Introduction to C++ AMP
Accelerated **M**assive **P**arallelism

Marc Grégoire
marc.gregoire@nuonsoft.com
http://www.nuonsoft.com/
http://www.nuonsoft.com/blog/

Author of **Professional C++, 3rd Edition**



Supported by
Nikon Metrology

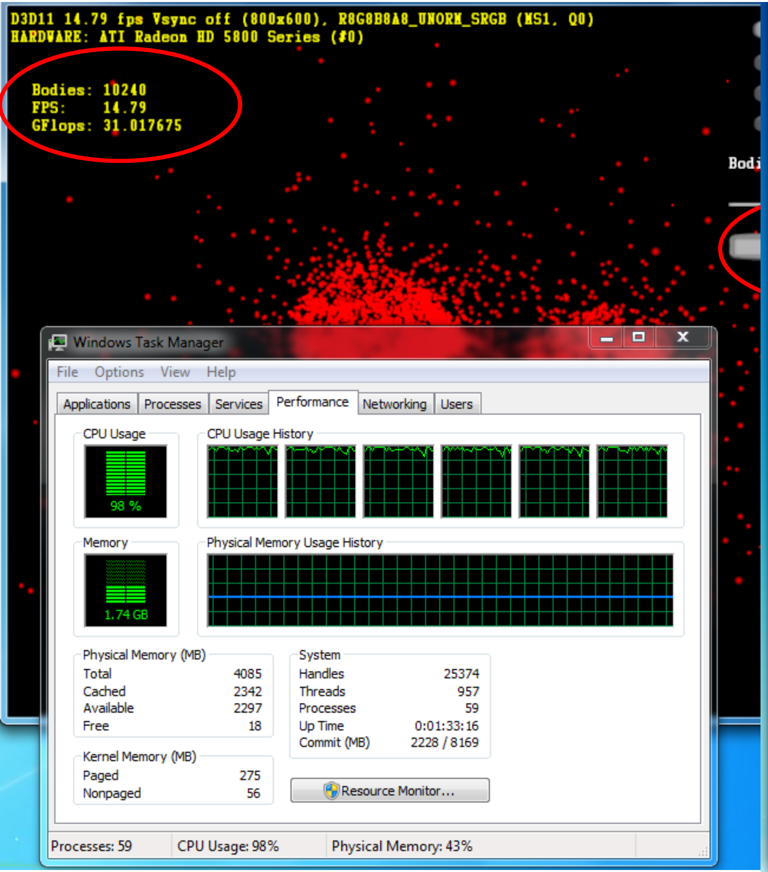
September 8th 2014

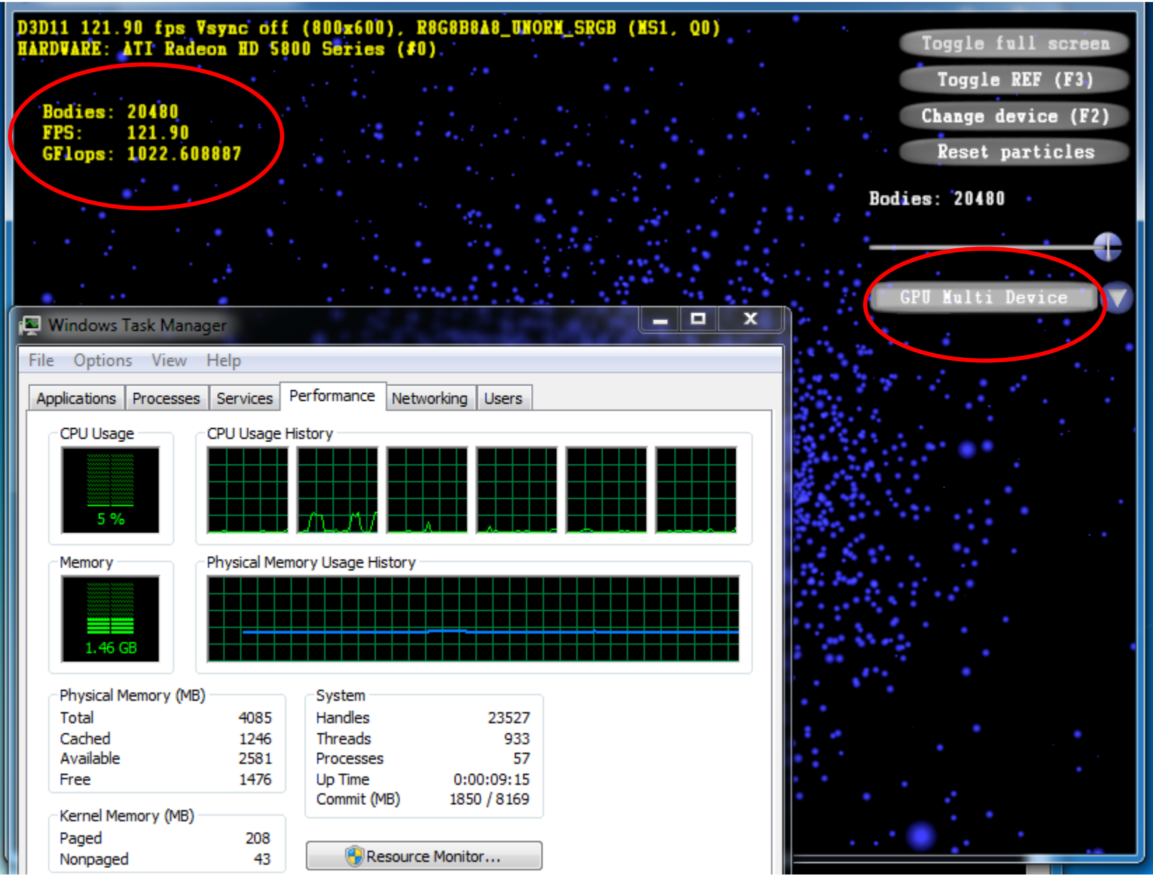
It is time to start taking advantage of the computing power of GPUs…

Demo…

N-Body Simulation

N-Body Simulation Demo





Demo...

Cartoonizer

Agenda

- Introduction
- Technical
  - The C++ AMP Technology
  - Coding Demo: Mandelbrot
- Visual Studio Integration
- Summary
- Resources

Introduction

- < 2005 → "Free Lunch"
  - Clock speed increased every year
  - Single threaded performance increased every year
- > 2005 → "Free Lunch" **is finished**
  - 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

Introduction

- Conclusion:

Scalable performance with future hardware?

↓

Parallelism (CPU, GPU, ...) is required!

Parallelism?

- On the CPU:
  - Vectorization (SIMD: SSE, AVX, ...)
  - Multithreading:
    - Microsoft PPL (Parallel Patterns Library)
    - Intel TBB (Threading Building Blocks) (compatible interface with PPL)
- Since Visual Studio 2012, auto-vectorization and auto-parallelization of your loops, if possible

Parallelism?

- On the GPU:
  - **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 of them are more C-like, and not truly C++ (so no type safety, genericity, …), only CUDA is becoming similar to C++
  - **Hard, you need to learn multiple technologies if you want to optimally target multiple devices…**

C++ AMP

- Solution for GPU's and other accelerators: **C++ AMP**
  - C++, not C, thus type safe and genericity using templates
  - It's an extension to C++, not a new language
  - C++ AMP is almost all library; only 2 keywords added to C++
    - tile_static
    - restrict
  - Included in vcredist
  - **Open standard!**

C++ AMP

- Vendor independent (NVidia, AMD, …)
- Abstracts “accelerators” (GPU’s, APU’s, …)
- Current version supports DirectX 11 GPU’s
- Fallback to WARP if no hardware GPU’s available
- In the future could support other accelerators like FPGA’s, off-site cloud computing…
- Support heterogeneous mix of accelerators!
  - Example: C++ AMP can use both an NVidia **and** AMD GPU in your system **at the same time** splitting the workload

Faster is not "just Faster"

- 2-3x faster is "just faster"
  - Do a little more, wait a little less
  - Doesn't change how you 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

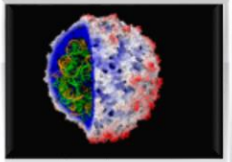
source

Power of Heterogeneous Computing

cppcon
the c++ conference

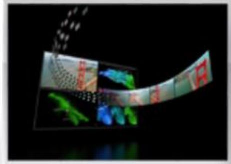
146X

Interactive visualization of volumetric white matter connectivity



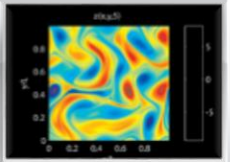
36X

Ionic placement for molecular dynamics simulation on GPU



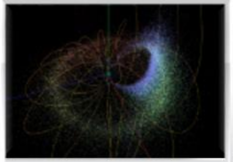
19X

Transcoding HD video stream to H.264



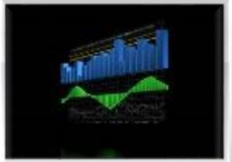
17X

Simulation in Matlab using .mex file CUDA function



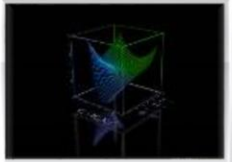
100X

Astrophysics N-body simulation



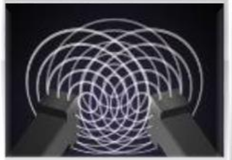
149X

Financial simulation of LIBOR model with swaptions



47X

GLAME@lab: An M-script API for linear Algebra operations on GPU

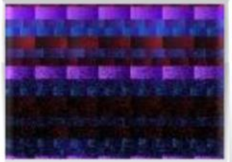


20X

Ultrasound medical imaging for cancer diagnostics

24X

Highly optimized object oriented molecular dynamics



30X

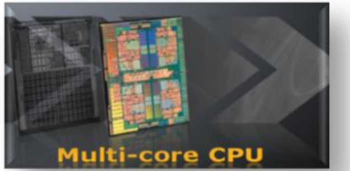
Cmatch exact string matching to find similar proteins and gene sequences

source

NVIDIA

CPU's vs GPU's 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 thanks to C++ AMP**

images source: AMD

C++ AMP

- Part of Visual C++ since VC++ 2012
- Complete Visual Studio integration (IntelliSense, GPU debugging, profiling, …)
- STL-like library for multidimensional data
- MS implementation builds on Direct3D

Agenda

- Introduction
- Technical
  - The C++ AMP Technology
  - Coding Demo: Mandelbrot
- Visual Studio Integration
- Summary
- Resources

Basics

- **#include <amp.h>**
- Everything is in the **concurrency** namespace
- Most important new classes:
  - **array, array_view**
  - **extent, index**
  - **accelerator, accelerator_view**
- New function: **parallel_for_each()**
- New keywords: **restrict / tile_static**

array

- **concurrency::array<type, dim>**
  - Allocates a buffer on an accelerator
  - Explicitly call **copy()** to copy data back from an accelerator to the CPU
- Example: A 1D array of 10 floats:
  - array<float, 1> arr(10)
- A 3D array of doubles:
  - array<double, 3> arr(3, 2, 1);

array_view

- **concurrency::array_view<type, dim>**
  - Wraps a user-allocated buffer so that C++ AMP can use it
- C++ AMP automatically transfers data between those buffers and memory on the accelerators
- Dense in least significant dimension

array_view

- Read/write buffer of given dimensionality, with elements of given type:
  - **array_view<type, dim> av(...);**
- Read-only buffer:
  - **array_view<const type, dim> 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<type, dim> av(...);**
    av.discard_data();
  - Only copies data from the accelerator to the CPU at the end, not to the accelerator at the start

extent<N> - size of an N-dim space

```
extent<1> e(5);
```

index<N> - an N-dimensional point

index<1> i(3);

parallel_for_each()

- **concurrency::parallel_for_each(*extent*, *lambda*);**
  - Basically, the entry point to C++ AMP
  - Takes number (and shape) of threads needed
  - Takes function or lambda to be executed by each thread
    - Must be **restrict(amp)**
  - Sends the work to the accelerator
    - Scheduling etc handled there
  - Returns – no blocking/waiting
  - Lambda must capture everything by value, except concurrency::array objects

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

Hello World: Array Addition

parallel_for_each: executes the lambda on the accelerator once per thread

```
void AddArrays(int n, int * pA, int * pB, int * pSum)
{
    array_view<const int,1> a(n, pA);
    array_view<const int,1> b(n, pB);
    array_view<int,1> sum(n, pSum);
    sum.discard_data();
    parallel_for_each(
            sum.extent,
            [a,b,sum](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

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.
Same dimensionality as the extent, so if extent is 2D, index is also 2D:
index<2> idx
Access the two dimensions as idx[0] and idx[1]

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

restrict(amp) restrictions

- Several restrictions apply to code marked as **restrict(amp)**:
  - Can only call other **restrict(amp)** functions
  - Function must be inlinable
  - Can only use
    - int, unsigned int, float, double, and bool
    - structs & arrays of these types

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
  - statics (use tile_static keyword instead)
  - dynamic_cast or typeid
  - asm declarations
  - varargs
  - unsupported types
    - e.g. char, short, long double

restrict()

- restrict() is really part of the signature
- Thus, can be overloaded on
- Example:
  - float foo(float) restrict(cpu, amp); // Code runs on both CPU and C++ AMP accelerators
  - float bar(float); // General code
    float bar(float) restrict(amp); // C++ AMP specific code

parallel_for_each() – lambda

- The lambda executes in parallel with CPU code that 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 observes the array_view
    - Not recommended, because you might lose error information if there is no try/catch block catching exceptions at that point
  - Automatically when for example array_view goes out of scope
    - Bad idea, errors will be ignored silently because destructors are not allowed to throw exceptions

accelerator / accelerator_view

- **accelerator** and **accelerator_view** can be used to query for information on installed accelerators
- **accelerator::get_all()** returns a vector of accelerators in the system

```
#include <iostream>
#include <amp.h>
using namespace std;
using namespace concurrency;
int main() {
  auto accelerators = accelerator::get_all();
  for (auto&& accel : accelerators) {
    wcout << accel.get_description() << endl;
  }
  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
- Because a tile of threads shares the programmable cache, you must prevent race conditions
  - Tile barrier can ensure a wait

Agenda

- Introduction
- Technical
  - The C++ AMP Technology
  - Coding Demo: Mandelbrot
- Visual Studio Integration
- Summary
- Resources

Mandelbrot – Single-Threaded

```
for (int y = -halfHeight; y < halfHeight; ++y) {
    // Formula: zi = z^2 + z0
    float Z0_i = view_i + y * zoomLevel;
    for (int x = -halfWidth; x < halfWidth; ++x) {
        float Z0_r = view_r + x * zoomLevel;
        float Z_r = Z0_r;
        float Z_i = Z0_i;
        float res = 0.0f;
        for (int iter = 0; iter < maxiter; ++iter) {
            float Z_rSquared = Z_r * Z_r;
            float Z_iSquared = Z_i * Z_i;
            if (Z_rSquared + Z_iSquared > escapeValue) {
                // We escaped
                res = iter + 1 - log(log(sqrt(Z_rSquared + Z_iSquared))) * invLogOf2;
                break;
            }
            Z_i = 2 * Z_r * Z_i + Z0_i;
            Z_r = Z_rSquared - Z_iSquared + Z0_r;
        }
        unsigned __int32 grayValue = static_cast<unsigned __int32>(res * 50);
        unsigned __int32 result = grayValue | (grayValue << 8) | (grayValue << 16);
        pBuffer[(y + halfHeight) * m_nBuffWidth + (x + halfWidth)] = result;
    }
}
```

Mandelbrot – PPL

```
parallel_for(-halfHeight, halfHeight, 1, [&](int y) {
    // Formula: zi = z^2 + z0
    float Z0_i = view_i + y * zoomLevel;
    for (int x = -halfWidth; x < halfWidth; ++x) {
        float Z0_r = view_r + x * zoomLevel;
        float Z_r = Z0_r;
        float Z_i = Z0_i;
        float res = 0.0f;
        for (int iter = 0; iter < maxiter; ++iter) {
            float Z_rSquared = Z_r * Z_r;
            float Z_iSquared = Z_i * Z_i;
            if (Z_rSquared + Z_iSquared > escapeValue) {
                // We escaped
                res = iter + 1 - log(log(sqrt(Z_rSquared + Z_iSquared))) * invLogOf2;
                break;
            }
            Z_i = 2 * Z_r * Z_i + Z0_i;
            Z_r = Z_rSquared - Z_iSquared + Z0_r;
        }
        unsigned __int32 grayValue = static_cast<unsigned __int32>(res * 50);
        unsigned __int32 result = grayValue | (grayValue << 8) | (grayValue << 16);
        pBuffer[(y + halfHeight) * m_nBuffWidth + (x + halfWidth)] = result;
    }
});
```

Mandelbrot – C++ AMP

cppcon the c++ conference

```
array_view<unsigned __int32, 2> a(m_nBuffHeight, m_nBuffWidth, pBuffer);
a.discard_data();
parallel_for_each(a.extent, [=](index<2> idx) restrict(amp) {
    // Formula: zi = z^2 + z0
    int x = idx[1] - halfWidth; int y = idx[0] - halfHeight;
    float Z0_i = view_i + y * zoomLevel;
    float Z0_r = view_r + x * zoomLevel;
    float Z_r = Z0_r; float Z_i = Z0_i;
    float res = 0.0f;
    for (int iter = 0; iter < maxiter; ++iter) {
        float Z_rSquared = Z_r * Z_r;
        float Z_iSquared = Z_i * Z_i;
        if (Z_rSquared + Z_iSquared > escapeValue) {
            // We escaped
            res = iter + 1 – fast_log(fast_log(fast_sqrt(Z_rSquared + Z_iSquared))) * invLogOf2;
            break;
        }
        Z_i = 2 * Z_r * Z_i + Z0_i;
        Z_r = Z_rSquared - Z_iSquared + Z0_r;
    }
    unsigned __int32 grayValue = static_cast<unsigned __int32>(res * 50);
    unsigned __int32 result = grayValue | (grayValue << 8) | (grayValue << 16);
    a[idx] = result;
});
a.synchronize();
```

fast_math namespace for single precision
precise_math namespace for double precision

Mandelbrot – C++ AMP

- Wrap C++ AMP code inside a try-catch block to handle errors!

```
try
{
    array_view<unsigned __int32, 2> a(m_nBuffHeight, m_nBuffWidth, pBuffer);
    a.discard_data();
    parallel_for_each(a.extent, [=](index<2> idx) restrict(amp) {

        ...

    });
    a.synchronize();
}
catch (const Concurrency::runtime_exception& ex)
{
    MessageBoxA(nullptr, ex.what(), "Error", MB_ICONERROR);
}
```

Demo…

Mandelbrot

Agenda

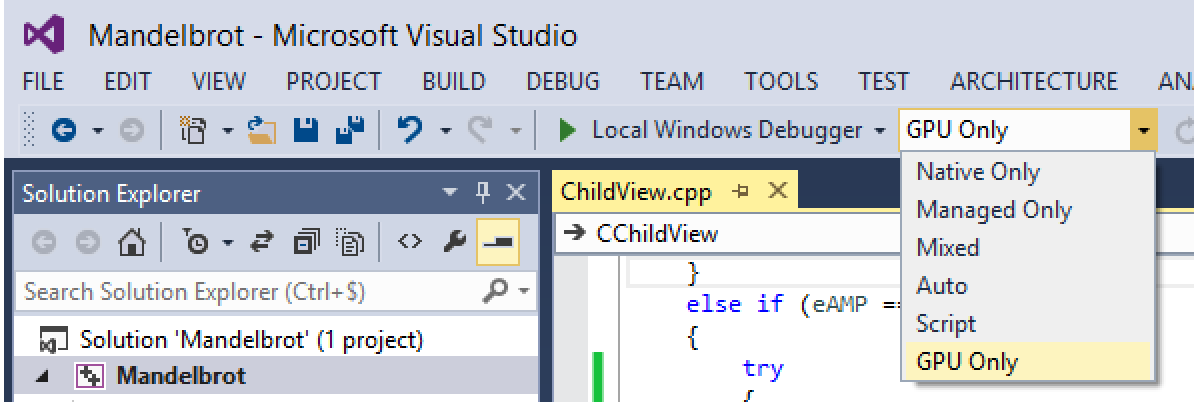
- Introduction
- Technical
  - The C++ AMP Technology
  - Coding Demo: Mandelbrot
- Visual Studio Integration
- Summary
- Resources

Visual Studio

- C++ AMP is deeply integrated into VC++ >= 2012
- Debugging
  - CPU/GPU breakpoints (even simultaneously)
  - GPU threads
  - Parallel Stacks
  - Parallel Watch
- Concurrency Visualizer

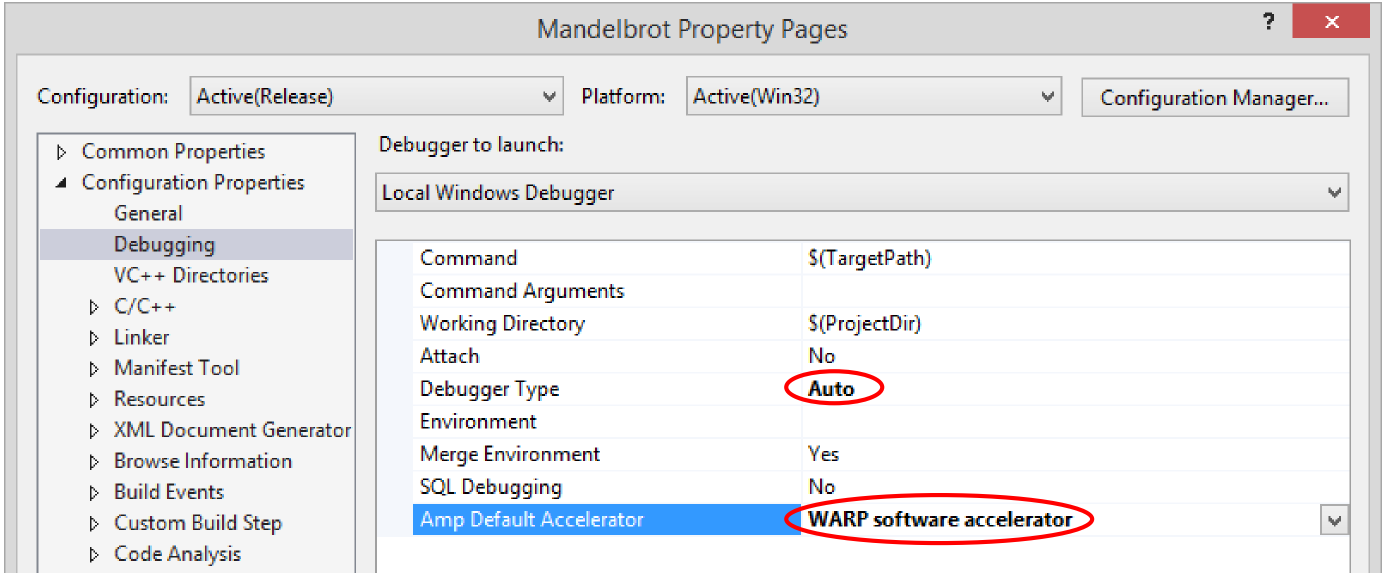
Debugging

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



Debugging

- Simultaneous CPU/GPU debugging:
  - Requires Windows 8.1 and at least VC++2013
  - Requires the WARP accelerator

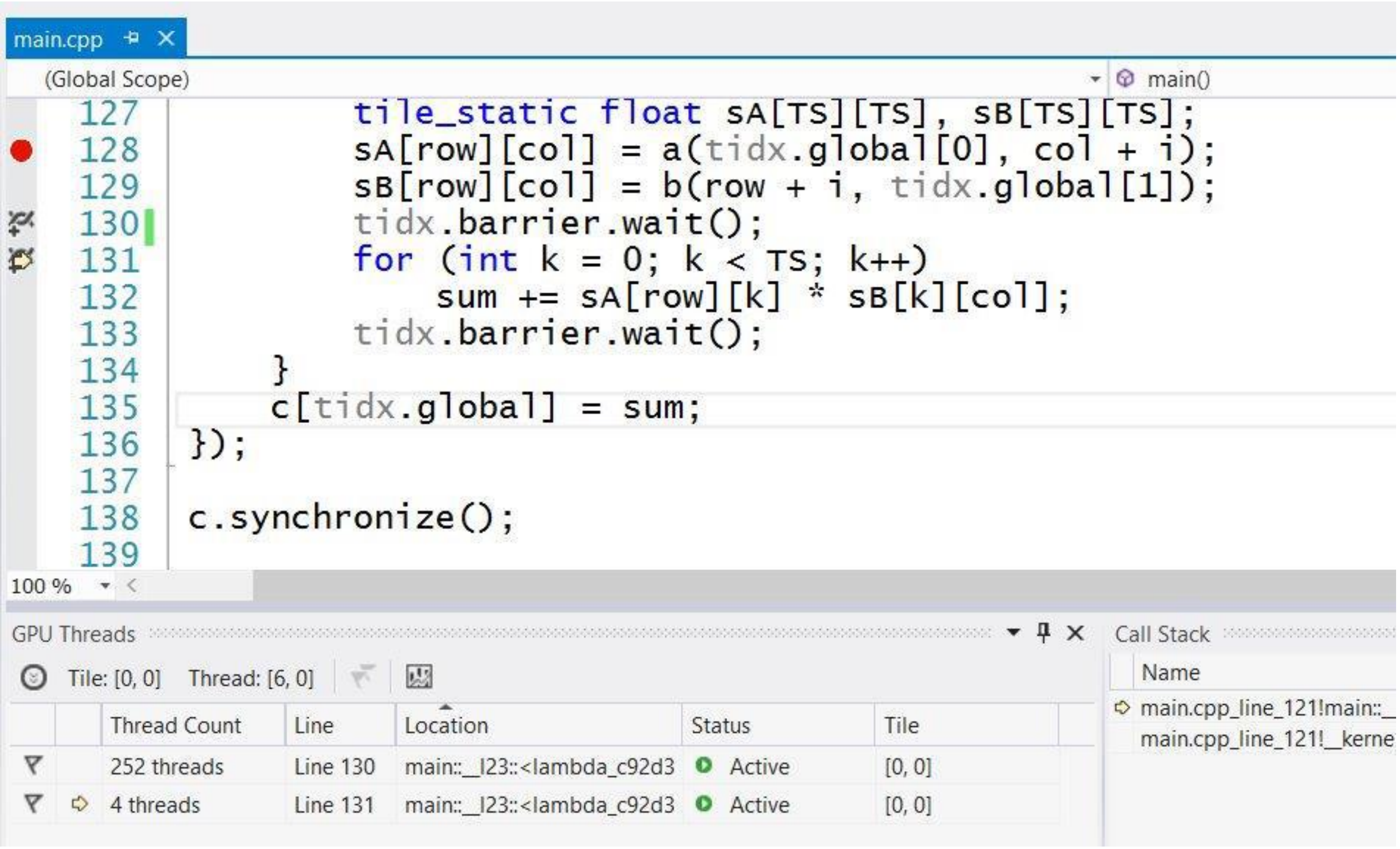


Debugging

```
else if (eAMP == m_renderMode)
{
    try
    {
        array_view<unsigned __int32, 2> a(m_nBuffHeight, m_nBuffWidth, pBuffer);
        a.discard_data();
        parallel_for_each(a.extent, [=](index<2> idx) restrict(amp)
        {
            // Formula: zi = z^2 + z0
            int x = idx[1] - halfWidth; int y = idx[0] - halfHeight;
            float Z0_i = view_i + y * zoomLevel;
            float Z0_r = view_r + x * zoomLevel;
```

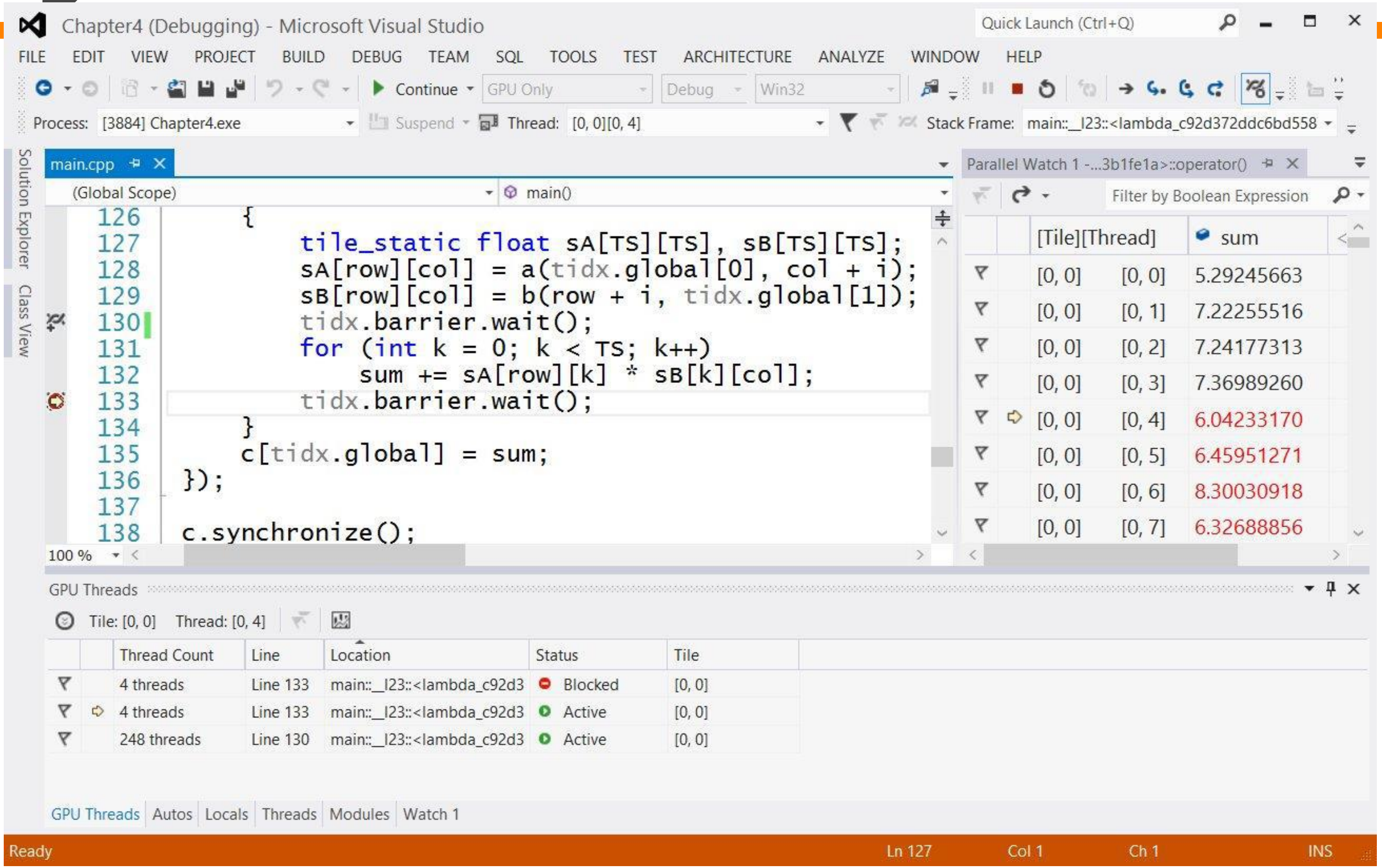
Debugging

- GPU Threads



Debugging

- Parallel Watch
- Shows values across multiple threads



Debugging

- Other things supported:
  - Help with race condition detection
  - Flagging, filtering, grouping
  - Freezing, thawing
  - Run tile to cursor

Debugging

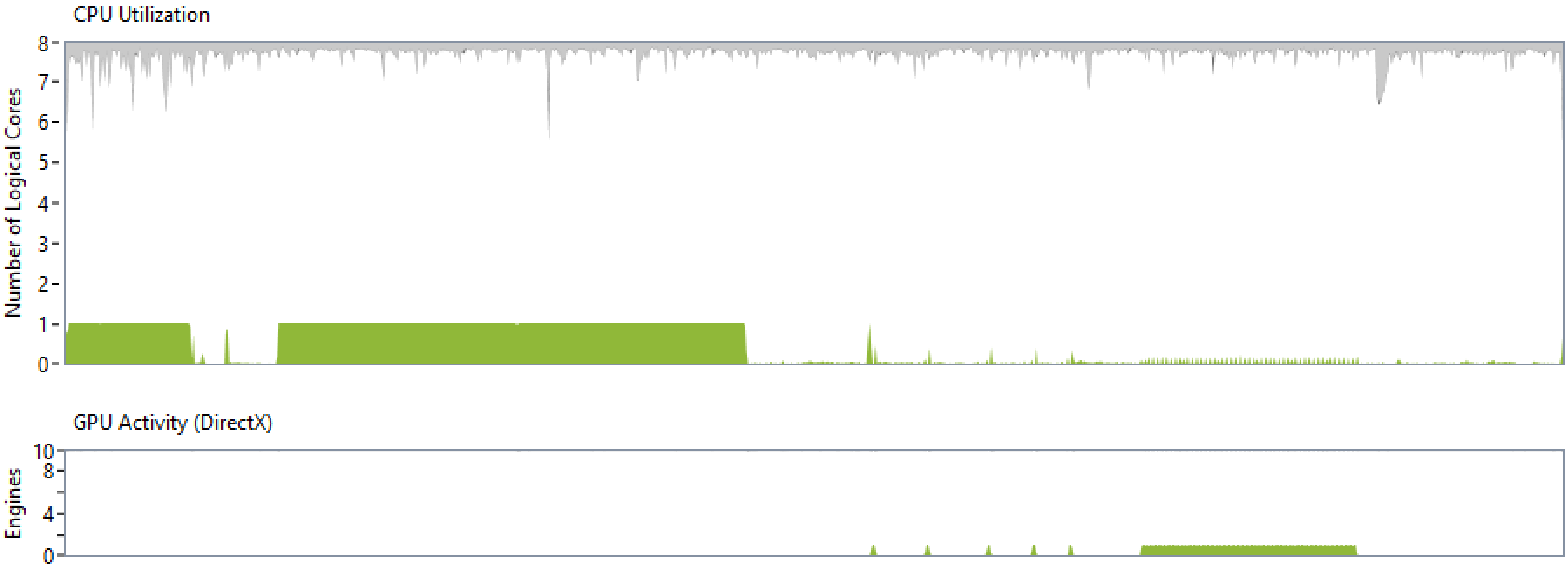
- Concurrency Visualizer is not included with VC++2013 anymore
- Download and install it from: http://visualstudiogallery.msdn.microsoft.com/24b56e51-fcc2-423f-b811-f16f3fa3af7a

Debugging

cppcon the c++ conference

- Concurrency Visualizer
  - Shows activity on CPU and GPU
  - Locate performance bottlenecks
  - Copy times to/from the accelerator
  - CPU underutilization
  - Thread contention
  - Cross-core thread migration
  - Synchronization delays
  - DirectX activity

Debugging



CPU Utilization
Number of Logical Cores
8
7
6
5
4
3
2
1
0
GPU Activity (DirectX)
Engines
10
8
4
0

Agenda

- Introduction
- Technical
  - The C++ AMP Technology
  - Coding Demo: Mandelbrot
- Visual Studio Integration
- Summary
- Resources

Summary

- C++ AMP makes heterogeneous computing mainstream and allows anyone to make use of parallel hardware
  - Easy-to-use
  - High-level abstractions in C++ (*not* C)
  - Excellent integration of C++ AMP in VS, including the debugger
  - Abstracts multi-vendor hardware
- C++ AMP is an open specification ☺

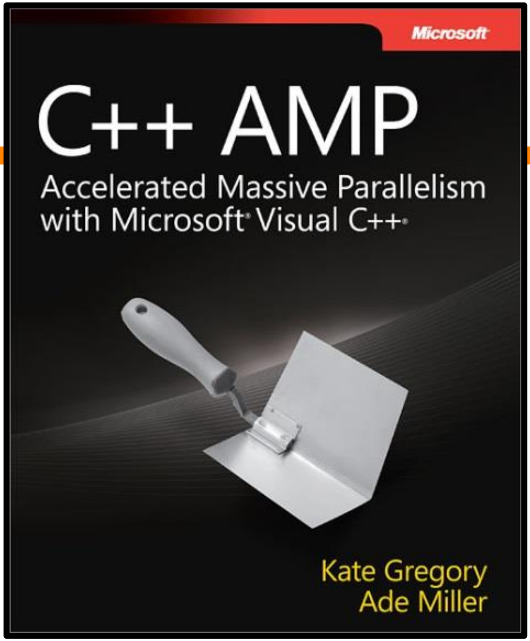
Other Presentations

- Tuesday, September 9 • 9:00am - 10:00am:
  “Writing Data Parallel Algorithms on GPUs”
  Ade Miller
- Tuesday, September 9 • 2:00pm - 3:00pm:
  “Another fundamental shift in Parallelism Paradigm? OpenMP 4.0 for GPU/Accelerators and other things”
  Michael Wong
- Tuesday, September 9 • 3:15pm - 4:15pm:
  “Decomposing a Problem for Parallel Execution”
  Pablo Halpern

The C++ AMP Book

Book / Source Code / Blogs:

http://www.gregcons.com/cppamp



- Written by Kate Gregory & Ade Miller, two experienced C++ programmers
- Covers all the C++ AMP features in detail, 350 pages
- Source code for each chapter and all three case studies available online
- eBook also available form Amazon or O'Reilly Books

Resources

- MSDN Native parallelism blog (team blog)
  - http://blogs.msdn.com/b/nativeconcurrency/
- Samples (36 at the time of this presentation)
  - 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
- Videos
  - http://channel9.msdn.com/Tags/c++-accelerated-massive-parallelism
- Daniel Moth's blog (previous PM of C++ AMP), interesting C++ AMP posts
  - http://www.danielmoth.com/Blog/
- MSDN Dev Center for Parallel Computing
  - http://msdn.com/concurrency
- MSDN Forums to ask questions
  - http://social.msdn.microsoft.com/Forums/en/parallelcppnative/threads

Microsoft Contact Person

- **Sharma Raman** (Program Manager C++ AMP)
- Sharma.Raman@microsoft.com

Questions

?