C++ Accelerated Massive Parallelism

Yossi Levanoni
Principal Software Design Engineer
Developer Division
Microsoft Corporation

Agenda

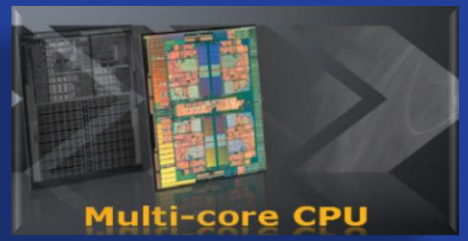
- Context
- Code
- IDE
- Summary

200x speedup

Unleashing GPU power using C++ AMP

CPUs vs GPUs today

CPU



- Medium level of parallelism
- Random accesses
- Lower memory bandwidth
- Higher clock frequency
- Higher power consumption

- **Supports general code**
- **Mainstream programming**

GPU

- High level of parallelism
- Streaming accesses
- High memory bandwidth
- Lower clock frequency
- Lower power consumption

- **Supports data-parallel code**
- **Niche programming**

images source: AMD

Industry trends…

- GPUs become more general purpose
- CPUs become more data parallel
- Fused SoC

- C++ AMP is designed to:
  - Take advantage of data parallel hardware TODAY

    …while…
  - Protecting your coding investments

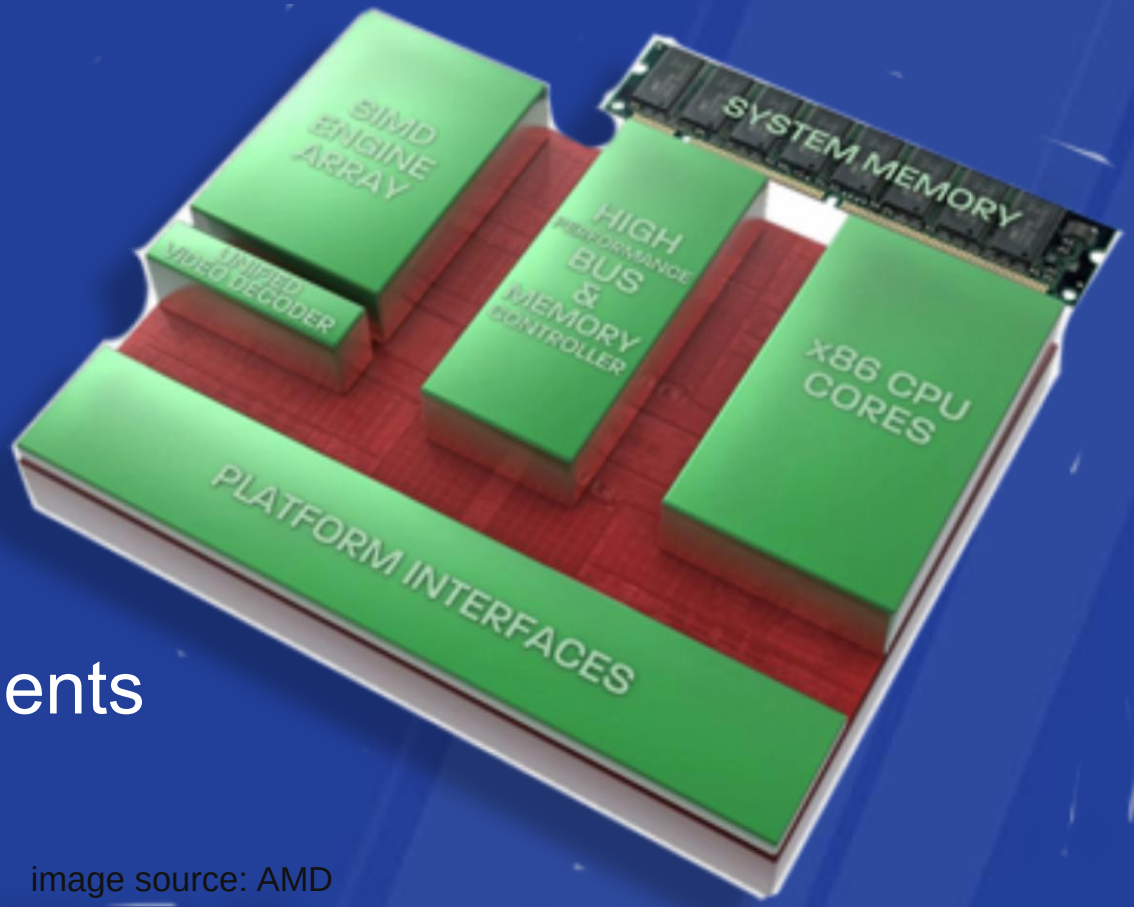
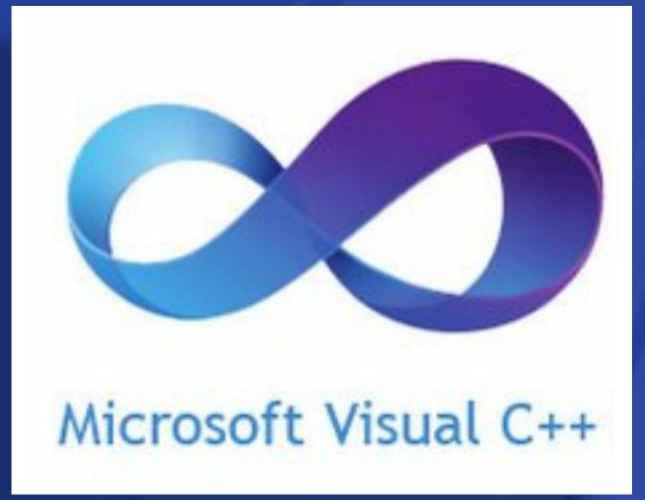


image source: AMD

C++ AMP

- Part of Visual C++
- Visual Studio integration
- STL-like library for multidimensional data
- Builds on Direct3D
- Intent is to make C++ AMP an open specification



performance

productivity

portability

Agenda checkpoint

- Context ✓
- Code
- IDE
- Summary

Hello World: Array Addition

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

How do we take the serial code on the left that runs on the CPU and convert it to run on an accelerator like the GPU?

Hello World: Array Addition

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




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


   {
         pC[i] = pA[i] + pB[i];
   }

}
```

```
#include <amp.h>
using namespace concurrency;

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

   parallel_for_each(
      sum.grid,
      [=](index<1> i) restrict(direct3d)
      {
         sum[i] = a[i] + b[i];
      }
    );
}
```

Basic Elements of C++ AMP coding

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

    parallel_for_each(
            sum.grid,
            [=](index<1> idx) restrict(direct3d)
            {
                sum[idx] = a[idx] + b[idx];
            }
    );
```

parallel_for_each: execute the lambda on the accelerator once per thread

restrict(direct3d): tells the compiler to check that this code can execute on Direct3D hardware (aka accelerator)

array_view: wraps the data to operate on the accelerator

grid: the number and shape of threads to execute the lambda

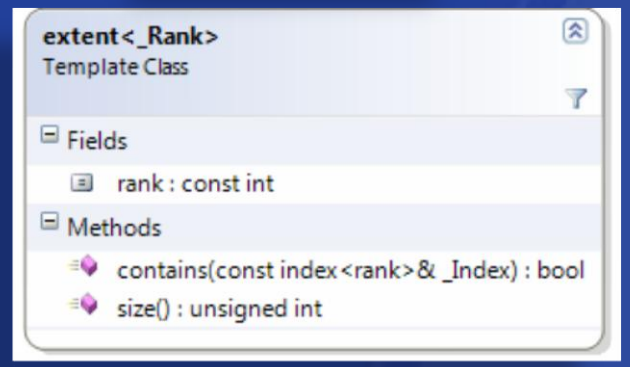
array_view variables captured and associated data copied to accelerator (on demand)

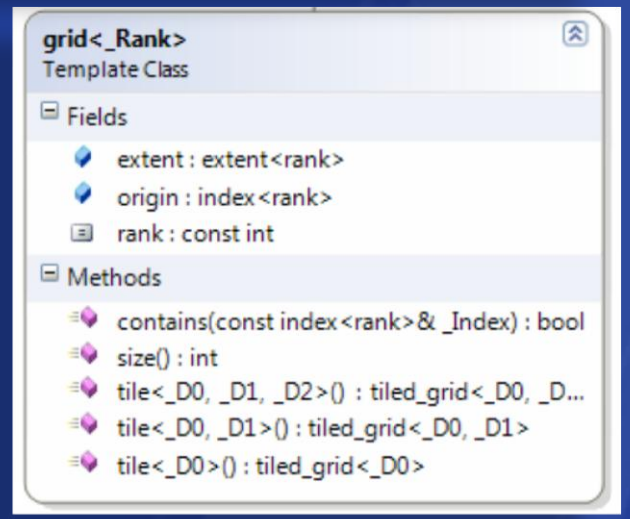
index: the thread ID that is running the lambda, used to index into data

grid<N>, extent<N>, and index<N>

- index<N>
  - represents an N-dimensional point
- extent<N>
  - number of units in each dimension of an N-dimensional space
- grid<N>
  - origin (index<N>) plus extent<N>

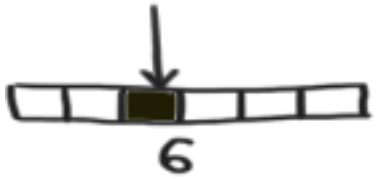
- N can be any number





Examples: grid, extent, and index

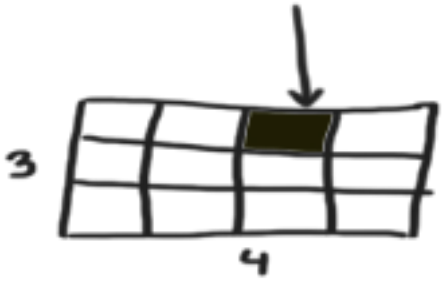
index<1> i(2);



extent<1> e(6);

grid<1> g(e);

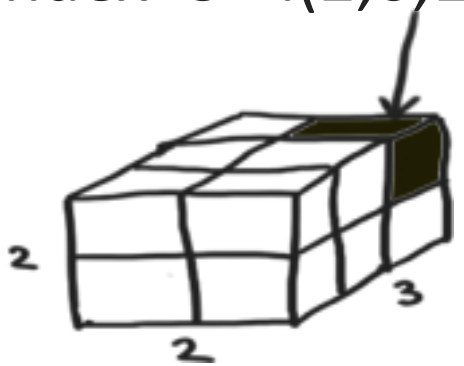
index<2> i(0,2);



extent<2> e(3,4);

grid<2> g(e);

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



extent<3> e(3,2,2);

grid<3> g(e);

array<T,N>

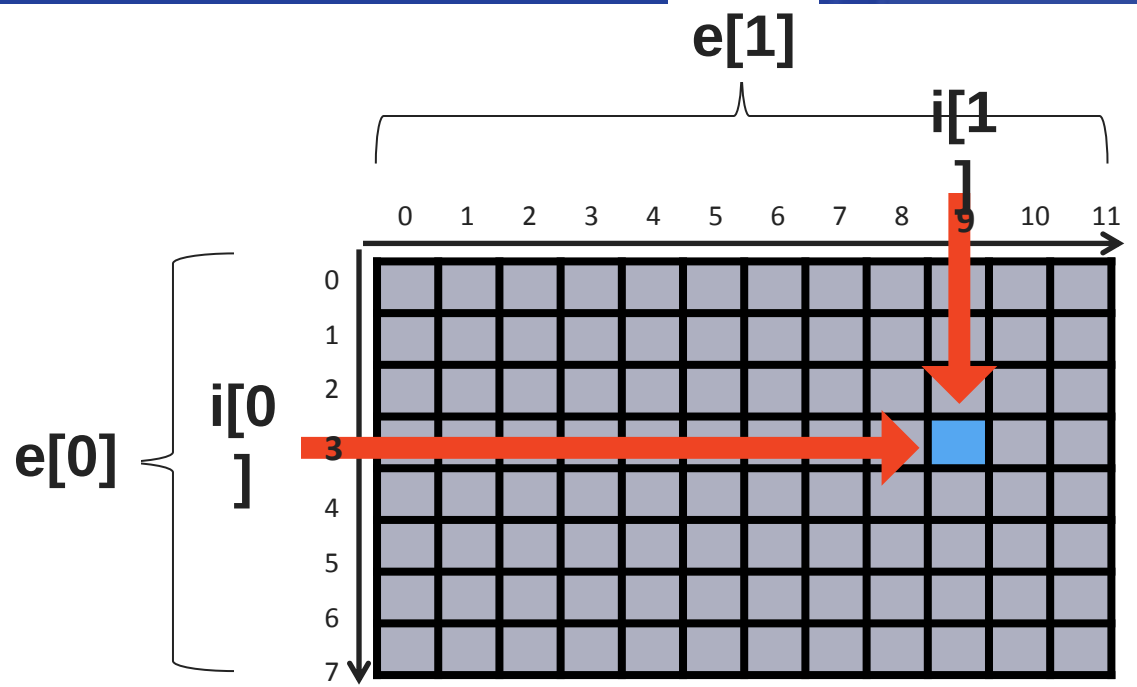
- Multi-dimensional array of rank N with element T
- Storage lives on accelerator

array<_Value_type, _Rank>
Template Class

Fields
- accelerator_view : accelerator_view
- associated_accelerator_view : accelerator_view
- grid : grid<_Rank>
- rank : const int

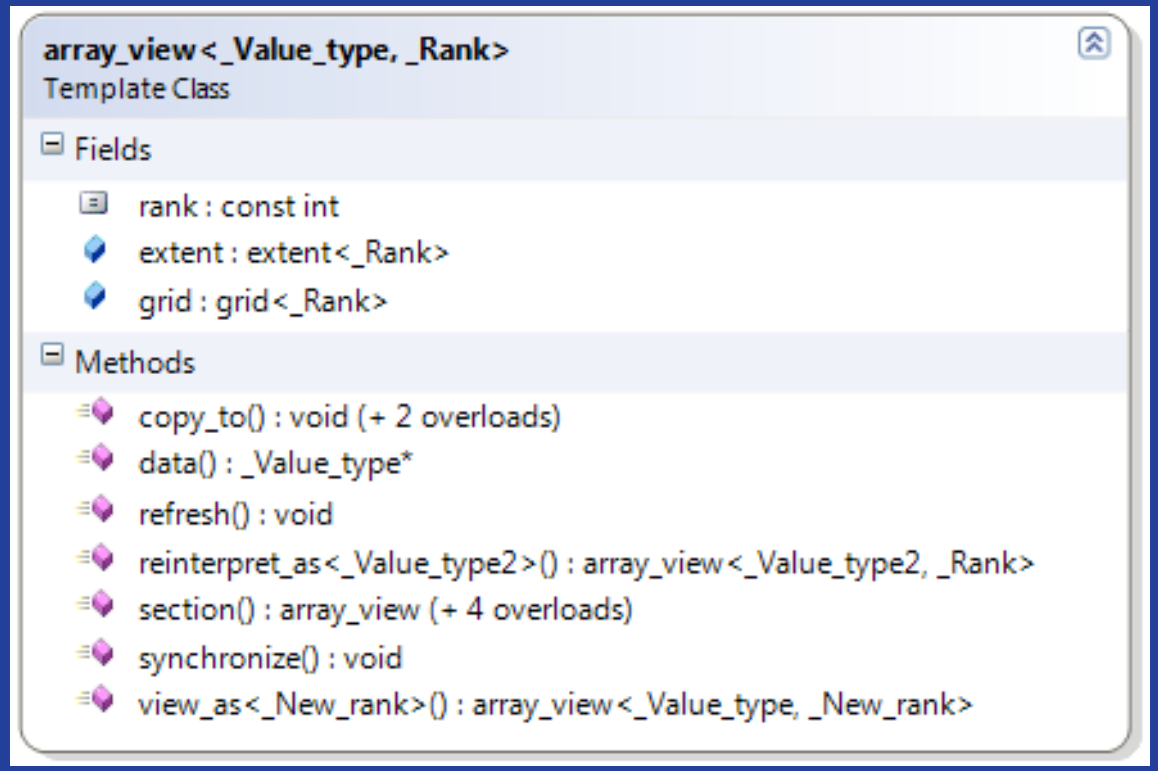
Methods
- copy_to() : void (+ 1 overload)
- data() : _Value_type* (+ 1 overload)
- reinterpret_as<_Value_type2>() : array_view<_Value_type2, ...
- section() : array_view<_Value_type, _Rank> (+ 9 overloads)
- view_as<_K>() : array_view<_Value_type, _K> (+ 1 overload)

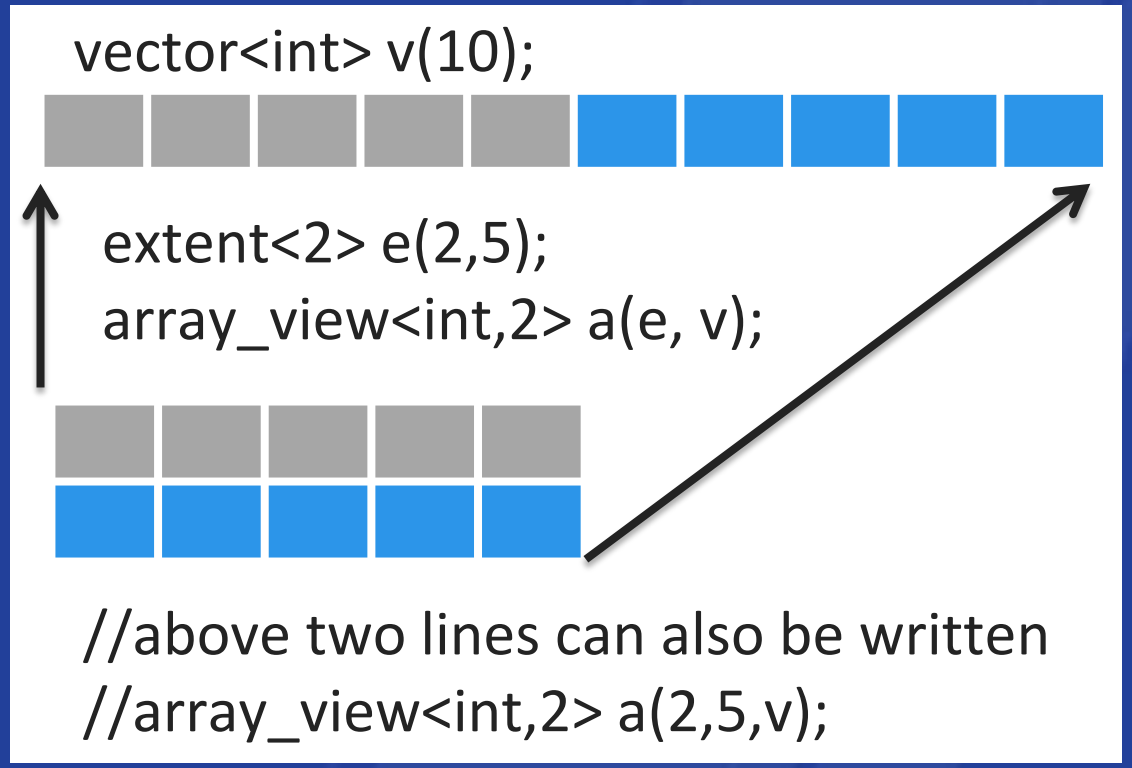
```
vector<int> v(96);
extent<2> e(8,12);
array<int,2> a(e, v.begin(), v.end());

// in the body of my lambda
index<2> i(3,9);
int i1 = a[i];
int i2 = a(i[0], i[1]);
```



array_view<T,N>

- View on existing data on the CPU or GPU
  - array_view<T,N>
  - array_view<const T,N>





Data Classes Comparison

array<T,N>

- Container for data
- Specific location
- Explicit copy
- Dense
- Capture by reference [&]

- Rank and element at compile time
- extent<N> at runtime
- Access elements with index<N>

array_view<T,N>

- Wrapper for data
- Access anywhere
- Implicit copy
- Dense in one dimension
- Capture by value [=]

- Rank and element at compile time
- extent<N> at runtime
- Access elements with index<N>

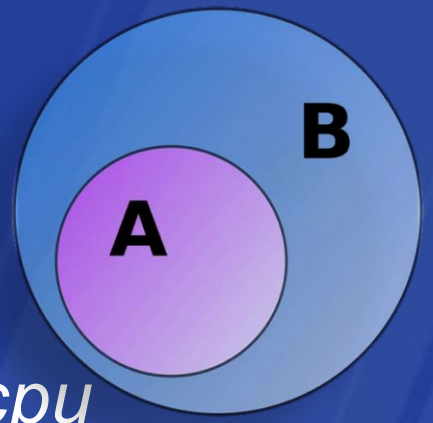
Kernels

- parallel_for_each
  - Executes the lambda for each point in the grid
  - As-if synchronous in terms of visible side-effects

```
1.      parallel_for_each(
2.          g,    //g is of type grid<N>
3.          [ ](index<N> idx) restrict(direct3d)
            {
                // kernel code
            }
1.      );
```

restrict(…)

- Applies to functions (including lambdas)
- Why restrict
  - Target-specific language restrictions
  - Optimizations or special code-gen behavior
- Functions can have multiple restrictions
  - In 1st release we are implementing *direct3d* and *cpu*
  - *cpu* – the implicit default



restrict(direct3d) restrictions

- Can only call other *restrict(direct3d)* functions
- All functions must be inlinable
- Only direct3d-supported types
  - int, unsigned int, float, double, bool
  - structs & arrays of these types
- Pointers and References
  - Lambdas cannot capture by reference[1], nor capture pointers
  - References and single-indirection pointers supported only as local variables and function arguments

restrict(direct3d) restrictions

- No
  - recursion
  - 'volatile'
  - virtual functions
  - pointers to functions
  - pointers to member functions
  - pointers in structs
  - pointers to pointers
- No
  - goto or labeled statements
  - throw, try, catch
  - globals or statics
  - dynamic_cast or typeid
  - Inline asm
  - varargs
  - unsupported types
    - e.g. char, short, long double

Example: restrict overloading

```
double bar( double ) restrict(cpu,direc3d);    // 1: same code for both
double cos( double );                          // 2a: general code
double cos( double ) restrict(direct3d);       // 2b: specific code

void SomeMethod(array<double> c) {
   parallel_for_each( c.grid, [=](index<2> idx) restrict(direct3d)
   {
      //…
      double d1 = bar(c[idx]);    // ok
      double d2 = cos(c[idx]);    // ok, chooses direct3d overload
      //…
   });
}
```

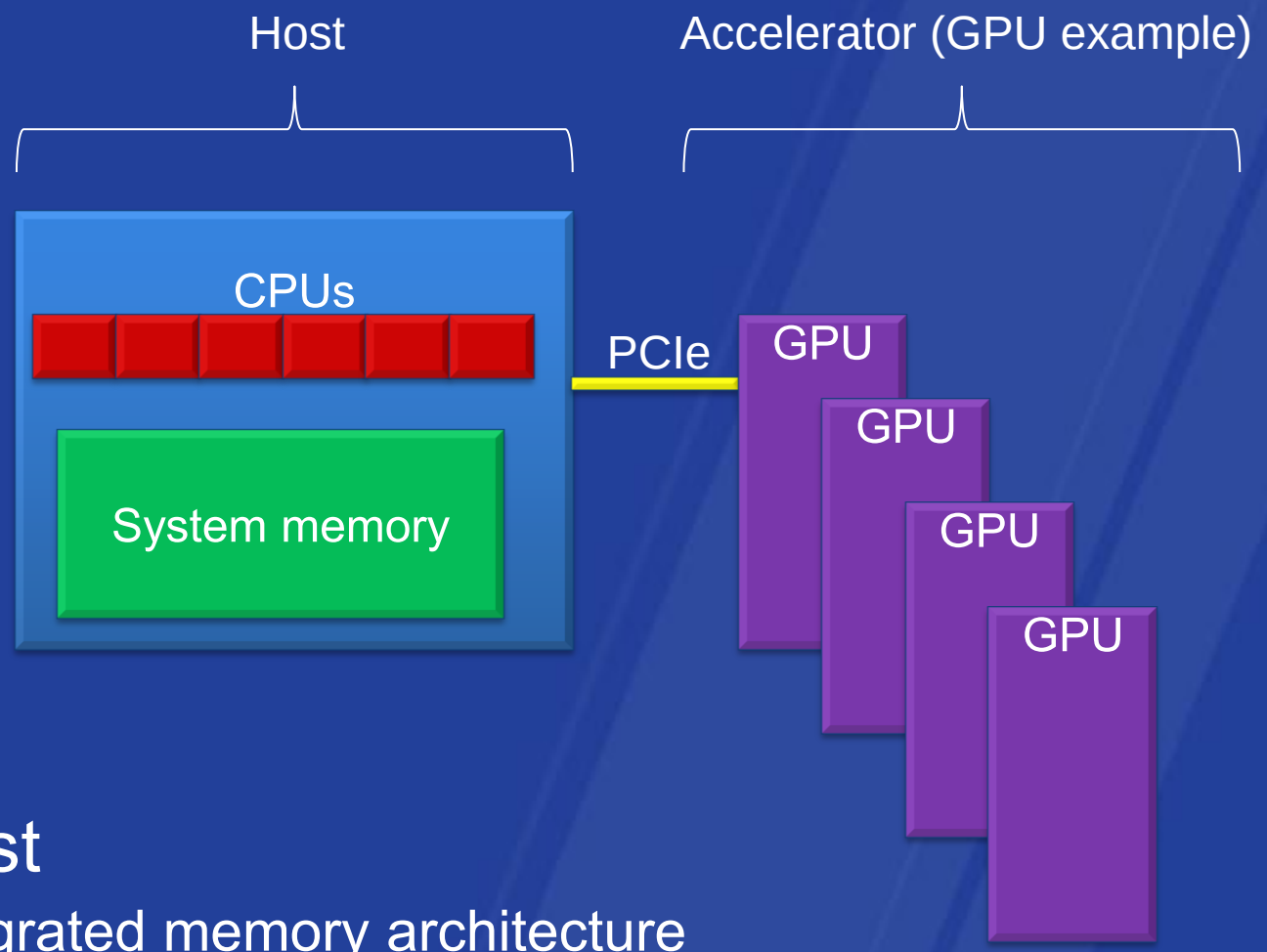
Example: Matrix Multiplication

```
void MatrixMultiplySerial( vector<float>& vC,
    const vector<float>& vA,
    const vector<float>& vB, int M, int N, int W )
{



   for (int row = 0; row < M; row++) {
      for (int col = 0; col < N; col++){
         float sum = 0.0f;
         for(int i = 0; i < W; i++)
            sum += vA[row * W + i] * vB[i * N + col];
         vC[row * N + col] = sum;
      }
   }
}
```

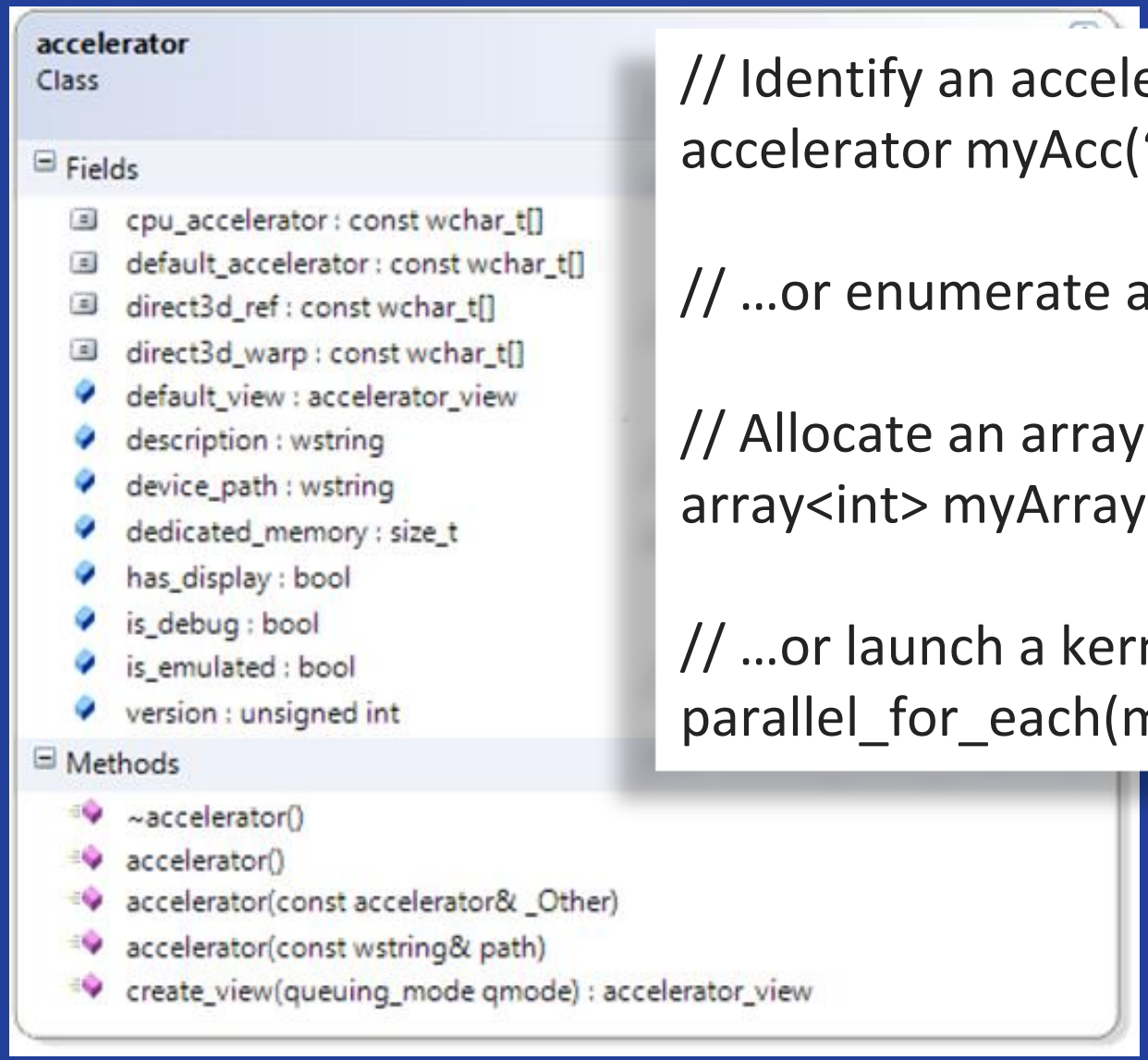
```
void MatrixMultiplyAMP( vector<float>& vC,
    const vector<float>& vA,
    const vector<float>& vB, int M, int N, int W )
{
   array_view<const float,2> a(M,W,vA),b(W,N,vB);
   array_view<writeonly<float>,2> c(M,N,vC);
   parallel_for_each(c.grid,
      [=](index<2> idx) restrict(direct3d) {
         int row = idx[0]; int col = idx[1];
         float sum = 0.0f;
         for(int i = 0; i < W; i++)
            sum += a(row, i) * b(i, col);
         c[idx] = sum;
      }
    );
}
```

accelerator, accelerator_view

- accelerator
  - e.g. DX11 GPU, REF
  - e.g. CPU
- accelerator_view
  - a context for scheduling and memory management
- Data transfers
  - between accelerator and host
    - could be optimized away for integrated memory architecture



Example: accelerator



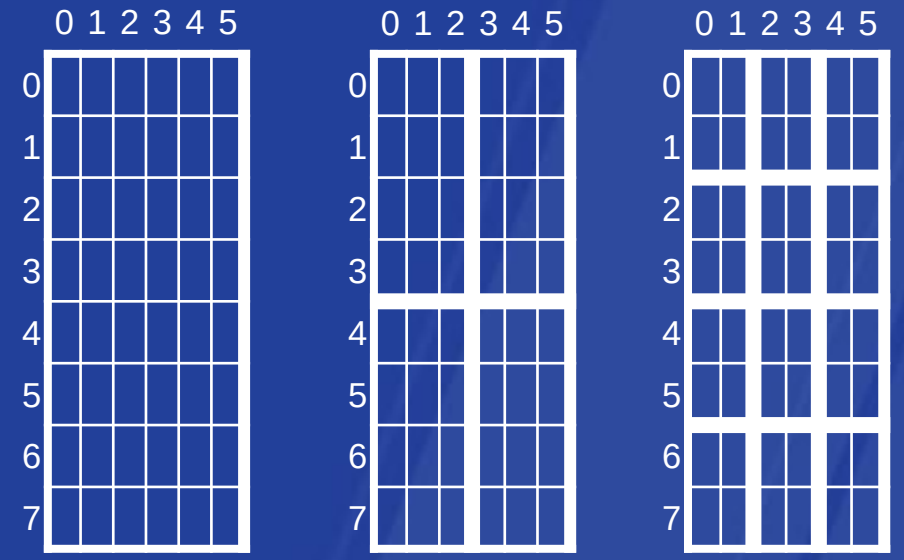

```
// Identify an accelerator based on Windows device ID
accelerator myAcc("PCI\\VEN_1002&DEV_9591&CC_0300");

// …or enumerate all accelerators (not shown)

// Allocate an array on my accelerator
array<int> myArray(10, myAcc.default_view);

// …or launch a kernel on my accelerator
parallel_for_each(myAcc.default_view, myArrayView.grid, ...);
```

C++ AMP at a Glance (so far)

- restrict(direct3d, cpu)
- parallel_for_each
- class array<T,N>
- class array_view<T,N>
- class index<N>
- class extent<N>, grid<N>
- class accelerator
- class accelerator_view

Achieving maximum performance gains

- Schedule threads in tiles
  - Avoid thread index remapping
  - Gain ability to use tile static memory



- parallel_for_each overload for tiles accepts
  - **tiled_grid<D0>** or **tiled_grid<D0, D1>** or **tiled_grid<D0, D1, D2>**

… a lambda which accepts …

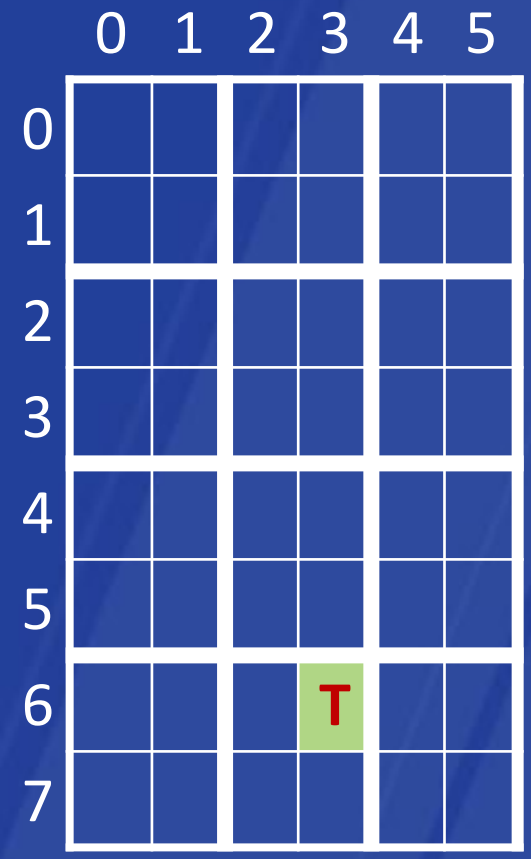
  - **tiled_index<D0>** or **tiled_index<D0, D1>** or **tiled_index<D0,**

tiled_grid, tiled_index

- Given

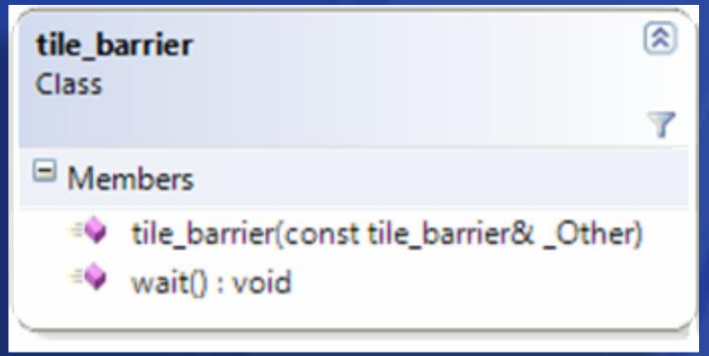
```
array_view<int,2> data(8, 6, p_my_data);
parallel_for_each(
   data.grid.tile<2,2>(),
   [=] (tiled_index<2,2> t_idx)… { … });
```

- When the lambda is executed by **T**
  - t_idx.global = index<2> (6,3)
  - t_idx.local = index<2> (0,1)
  - t_idx.tile = index<2> (3,1)
  - t_idx.tile_origin = index<2> (6,2)



tile_static, tile_barrier

- Within the tiled parallel_for_each lambda we can use
  - tile_static storage class 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(direct3d) functions
  - class tile_barrier
    - synchronize all threads within a tile
    - e.g. t_idx.barrier.wait();



Example: Matrix Multiplication (tiled)

```
void MatrixMultSimple(vector<float>& vC, const vector<float>& vA,
const vector<float>& vB, int M, int N, int W )
{

   array_view<const float,2> a(M, W, vA), b(W, N, vB);
   array_view<writeonly<float>,2> c(M,N,vC);
   parallel_for_each(c.grid,
      [=] (index<2> idx) restrict(direct3d)  {
         int row = idx[0]; int col = idx[1];
         float sum = 0.0f;









         for(int k = 0; k < W; k++)
            sum += a(row, k) * b(k, col);



         c[idx] = sum;
      } );
}
```

```
void MatrixMultTiled(vector<float>& vC, const vector<float>& vA,
const vector<float>& vB, int M, int N, int W )
{
   static const int TS = 16;
   array_view<const float,2> a(M, W, vA), b(W, N, vB);
   array_view<writeonly<float>,2> c(M,N,vC);
   parallel_for_each(c.grid.tile< TS, TS >(),
      [=] (tiled_index< TS, TS> t_idx) restrict(direct3d)  {
         int row = t_idx.local[0]; int col = t_idx.local[1];
         float sum = 0.0f;
         for (int i = 0; i < W; i += TS) {
            tile_static float locA[TS][TS], locB[TS][TS];
            locA[row][col] = a(t_idx.global[0], col + i);
            locB[row][col] = b(row + i, t_idx.global[1]);
            t_idx.barrier.wait();

            for (int k = 0; k < TS; k++)
               sum += locA[row][k] * locB[k][col];
            t_idx.barrier.wait();
          }
         c[t_idx.global] = sum;
      } );
}
```

C++ AMP at a Glance

- restrict(direct3d, cpu)
- parallel_for_each
- class array<T,N>
- class array_view<T,N>
- class index<N>
- class extent<N>, grid<N>
- class accelerator
- class accelerator_view
- tile_static storage class
- class tiled_grid< , , >
- class tiled_index< , , >
- class tile_barrier

Not Covered

1. Math library
   - e.g. acosf
2. Atomics library
   - e.g. atomic_fetch_add
   - Fences
3. Direct3D Interop
   - get_device, create_accelerator_view, make_array, get_buffer
4. Direct3D debugging intrinsics
   - direct3d_printf

- Graphics (textures and short vector types)
  - Coming up in the next Beta release

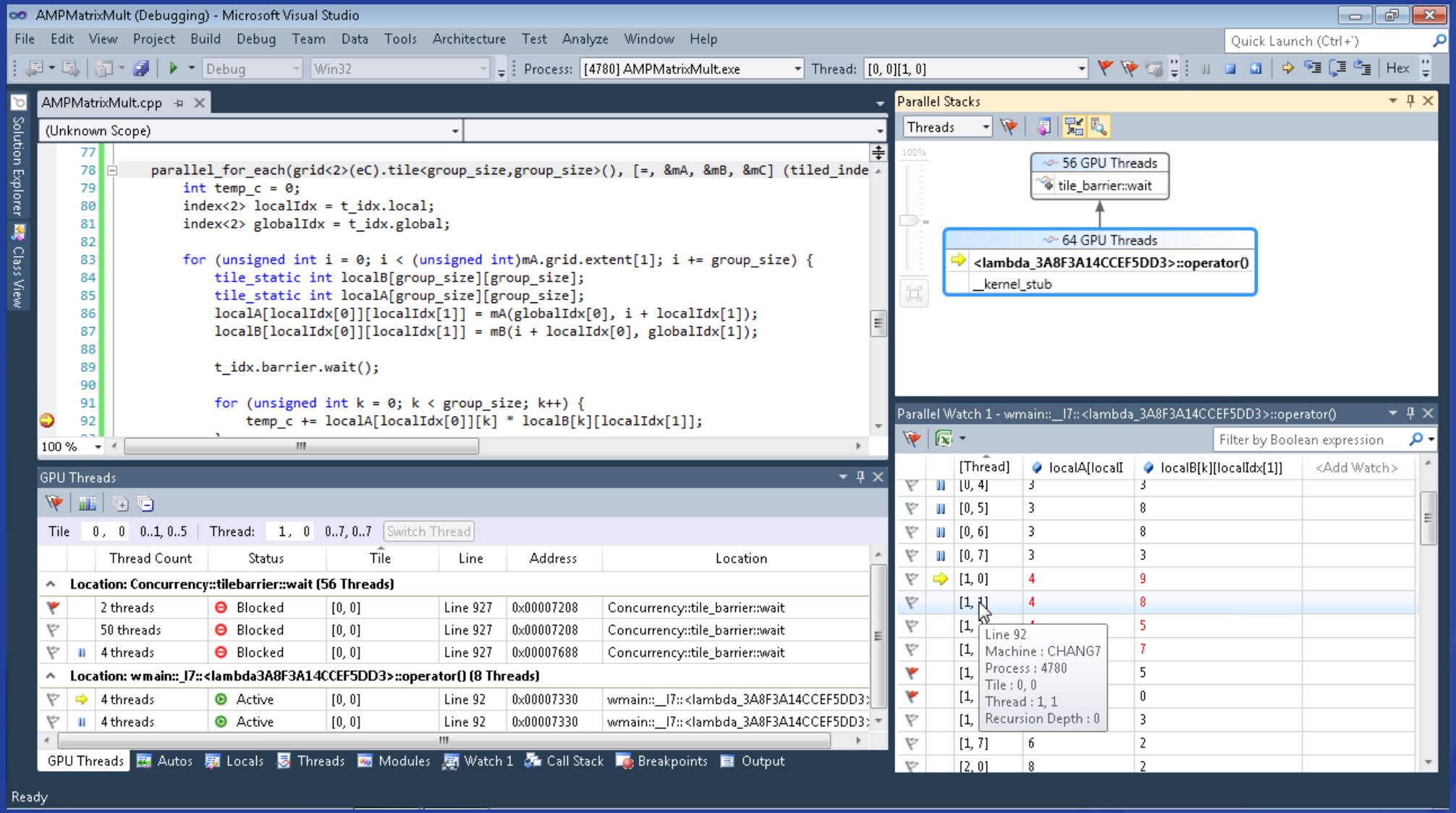
Agenda checkpoint

- Context
- Code
- IDE
- Summary

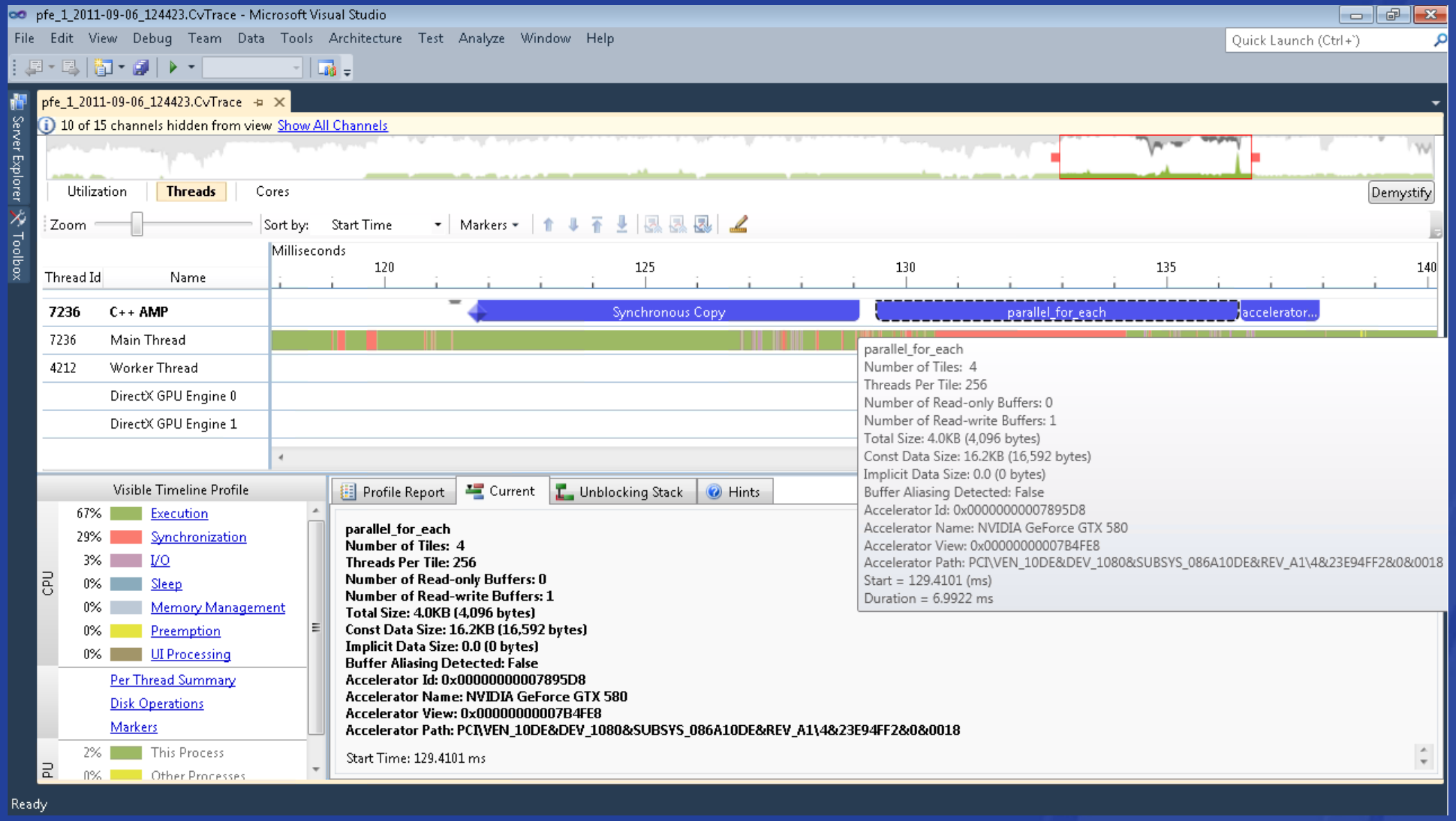
Visual Studio 11

- Organize
- Edit
- Design
- Build
- Browse
- Debug
- Profile

GPU Debugging



Concurrency Visualizer for the GPU



Agenda checkpoint

- Context
- Code
- IDE
- Summary

Summary

- Democratization of parallel hardware programmability
  - Performance for the mainstream
  - High-level abstractions in C++ (*not* C)
  - State-of-the-art Visual Studio IDE
  - Hardware abstraction platform
- Intent is to make C++ AMP an open specification

Questions?

MSDN Forums to ask questions

- http://social.msdn.microsoft.com/Forums/en/parallelcppnative/threads

MSDN Native parallelism blog (team blog)

- http://blogs.msdn.com/b/nativeconcurrency/

Daniel Moth's blog (PM of C++ AMP)

- http://www.danielmoth.com/Blog/

Microsoft®