UC Berkeley 2012 Short Course on Parallel Programming

Parallel Programming on Windows and Porting CS267

Matej Ciesko
Technology Policy Group (TPG)
Microsoft

Agenda

- Overview of parallel programming landscape
- C++ AMP
- CS267 port to Windows

Overview to

PARALLEL COMPUTING LANDSCAPE

Welcome to the jungle

The free lunch is so over

1975-2005

Put a **computer** on every desk, in every home, in every pocket.

2005-2011

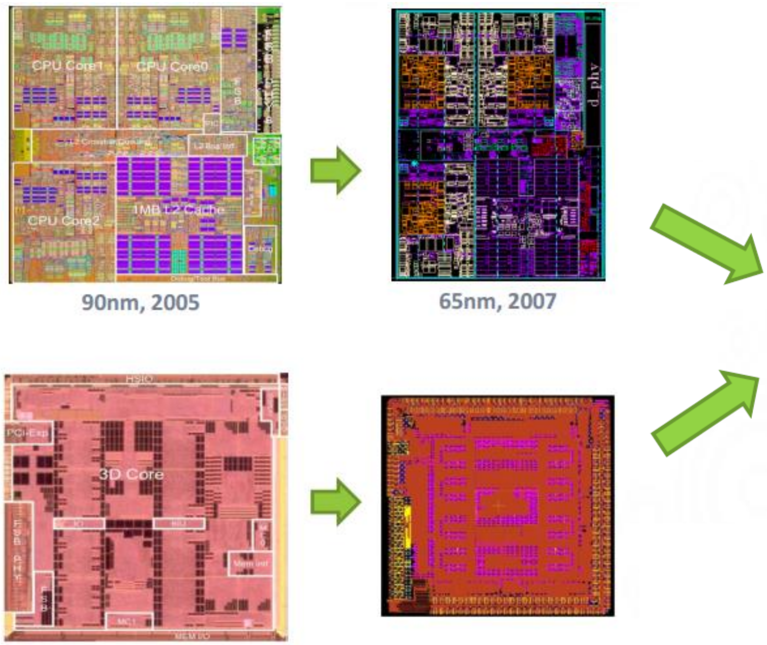
Put a **parallel supercomputer** on every desk, in every home, in every pocket.

2011-201x

Put a **massively parallel heterogeneous super-computer** on every desk, in every home, in every pocket.

Herb Sutter – “Welcome to the jungle”
David Callahan - AMP

Xbox360



CPU

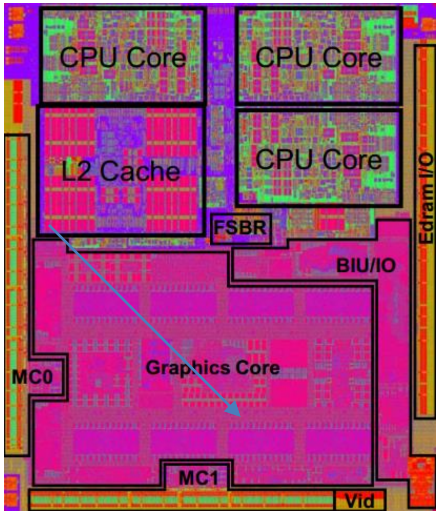
90nm, 2005

65nm, 2007

GPU

90nm, 2005

65nm, 2008



commercially available to millions

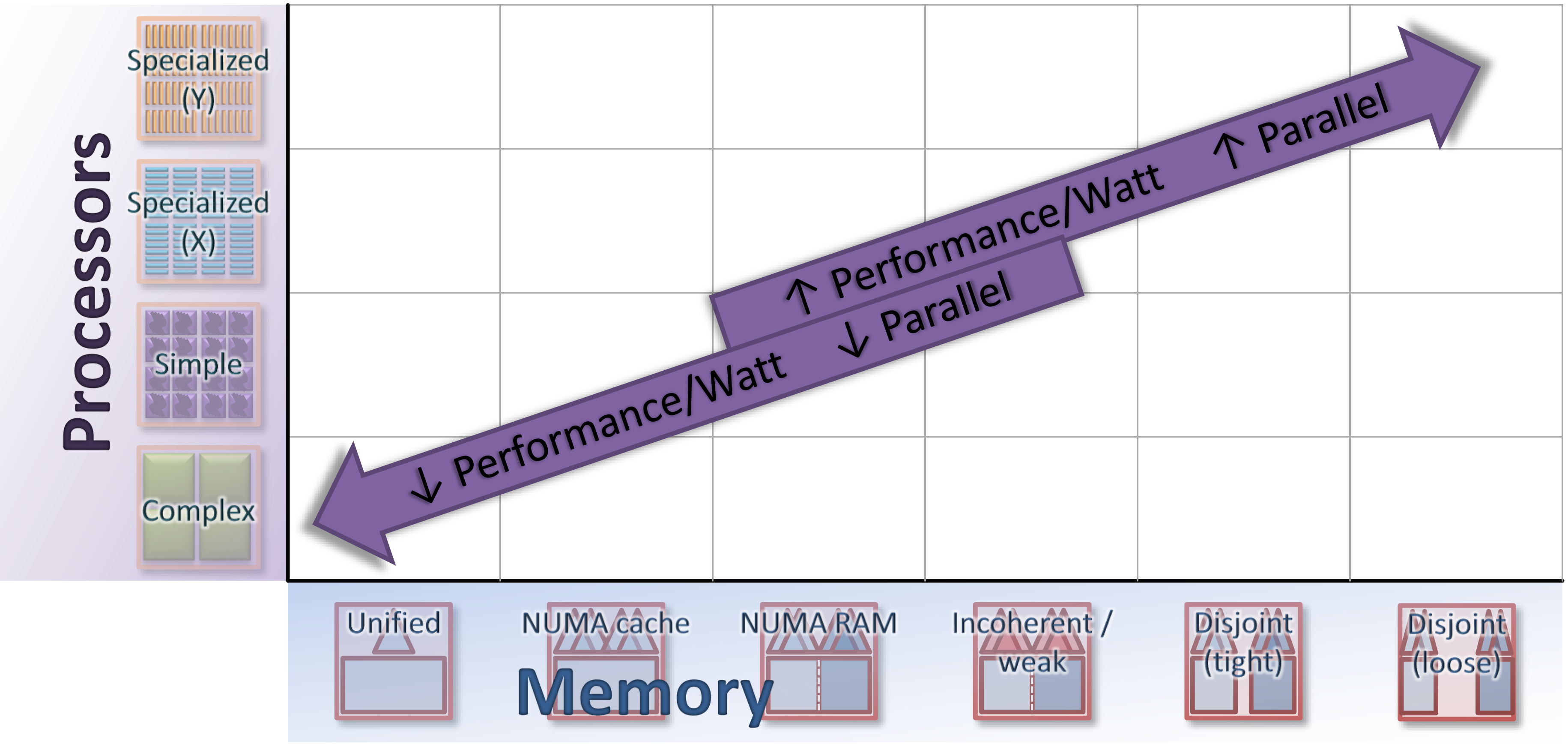
commercially affordable for millions

commercially programmable by millions

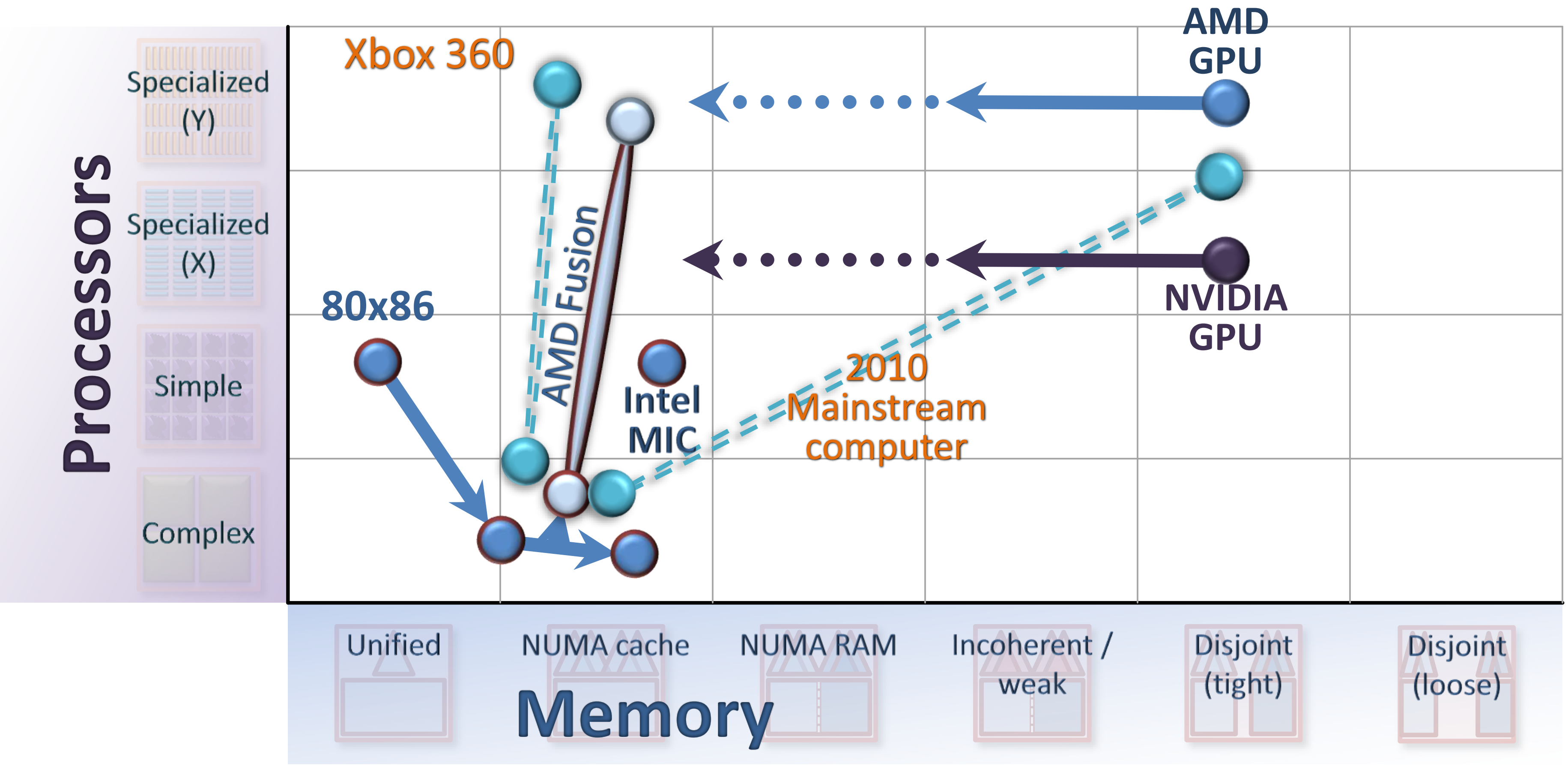
note: everything in "the mainstream" starts out in "the exotic"

GUIs ✓ objects ✓ parallelism ✓

Charting the Landscape

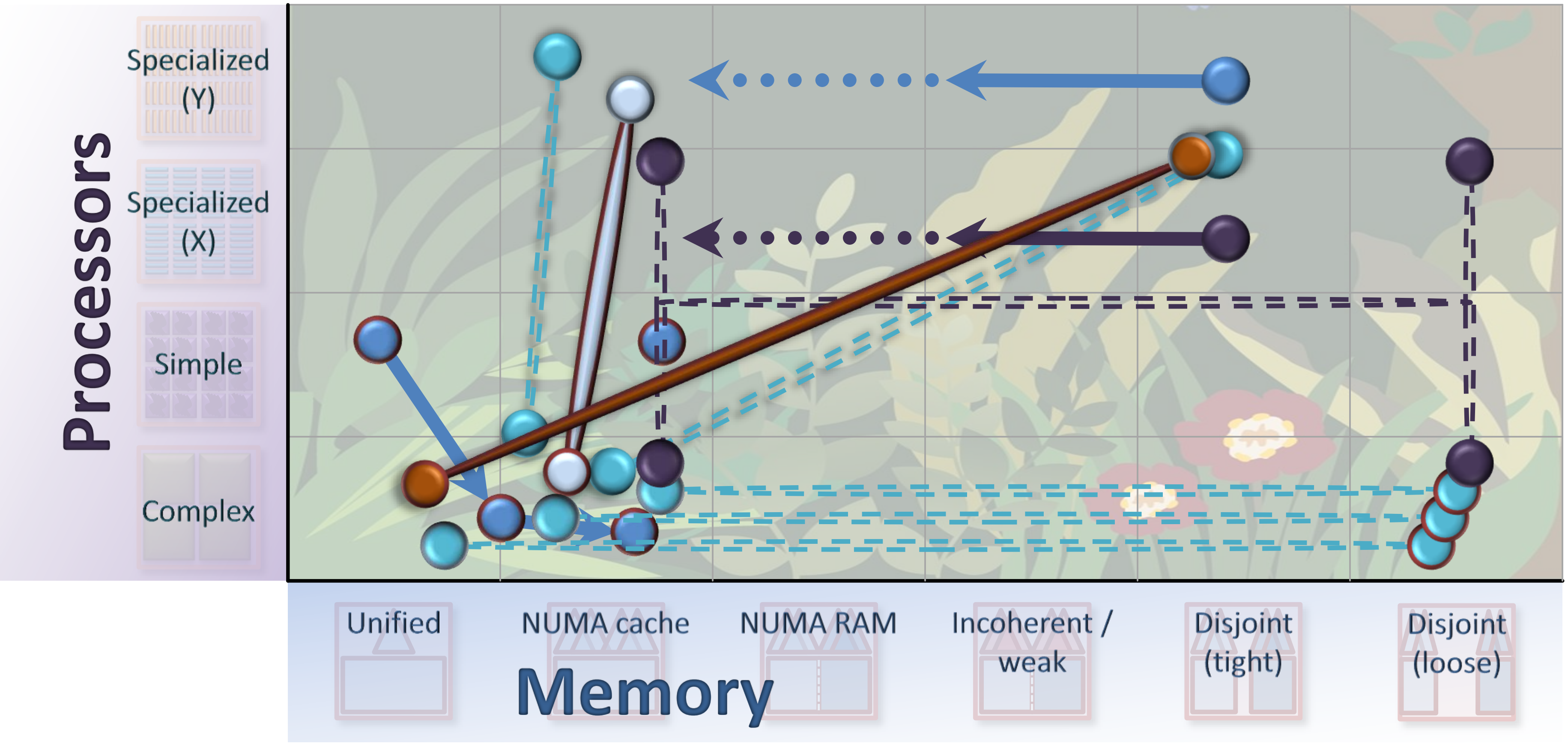


Hardware



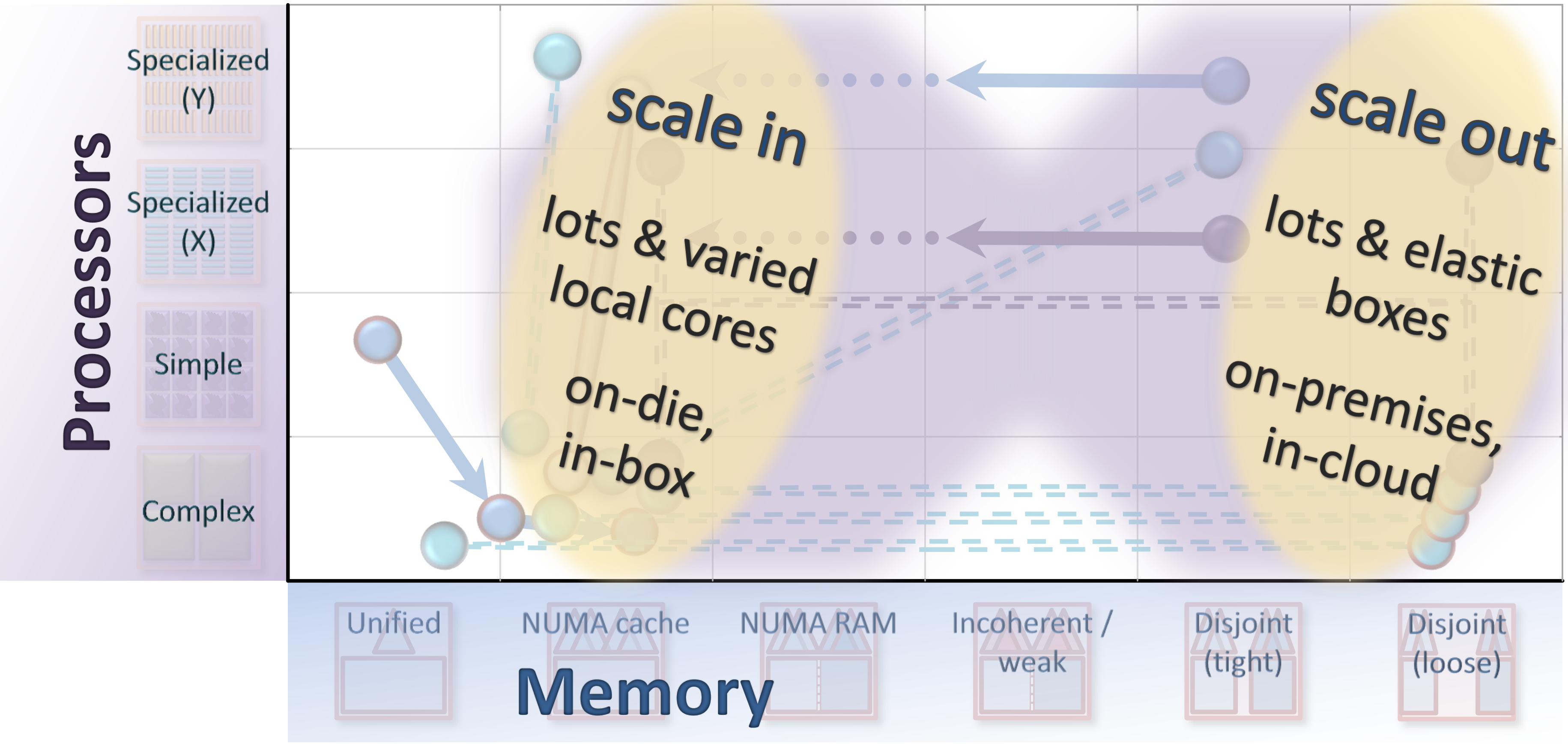
Processors
Specialized (Y)
Specialized (X)
Simple
Complex
Xbox 360
AMD Fusion
80x86
Intel MIC
AMD GPU
NVIDIA GPU
2010 Mainstream computer
Unified
NUMA cache
NUMA RAM
Incoherent / weak
Disjoint (tight)
Disjoint (loose)
Memory

The Jungle

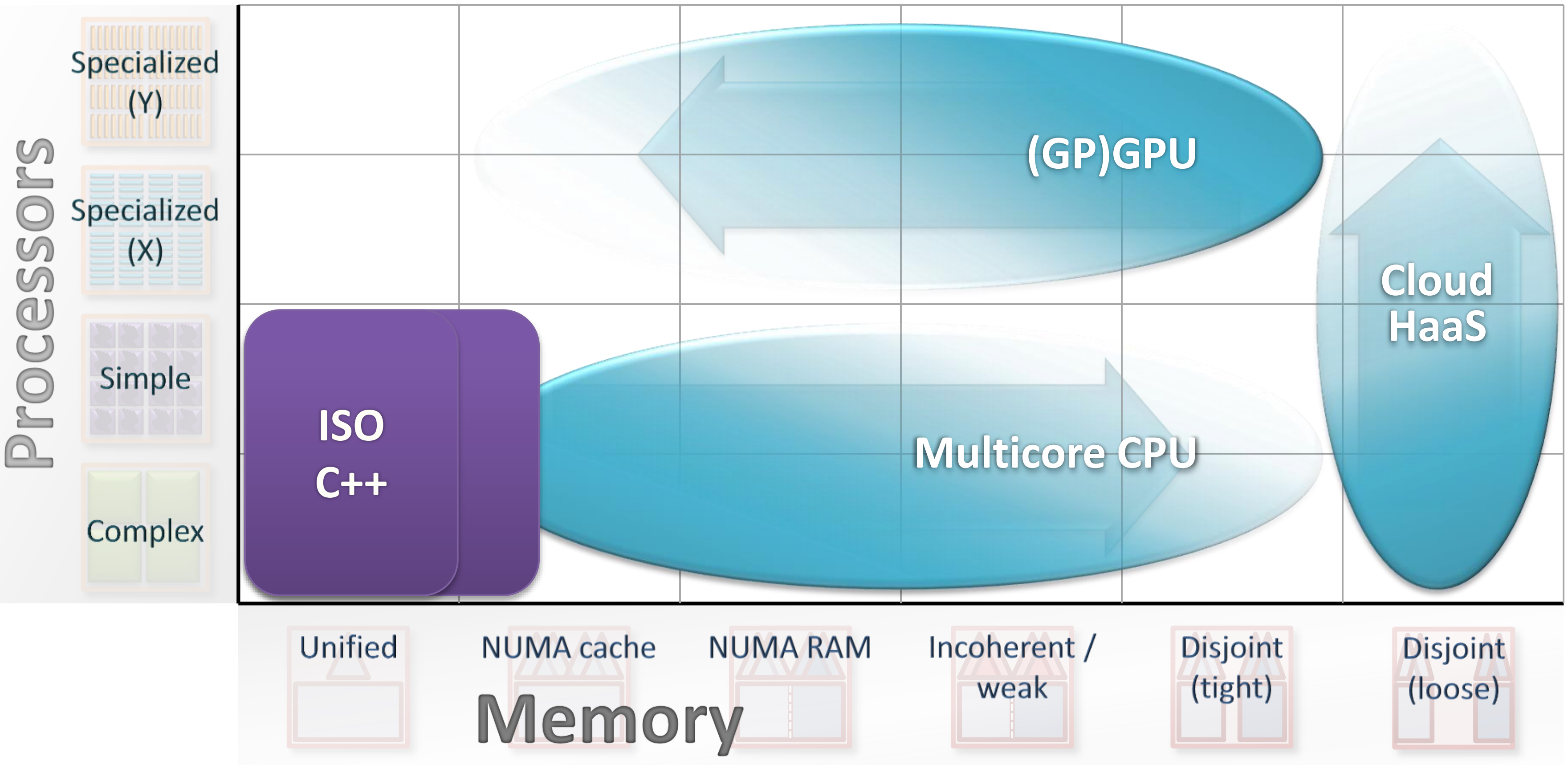


Processors
Specialized (Y)
Specialized (X)
Simple
Complex
Unified
NUMA cache
NUMA RAM
Incoherent / weak
Disjoint (tight)
Disjoint (loose)
Memory

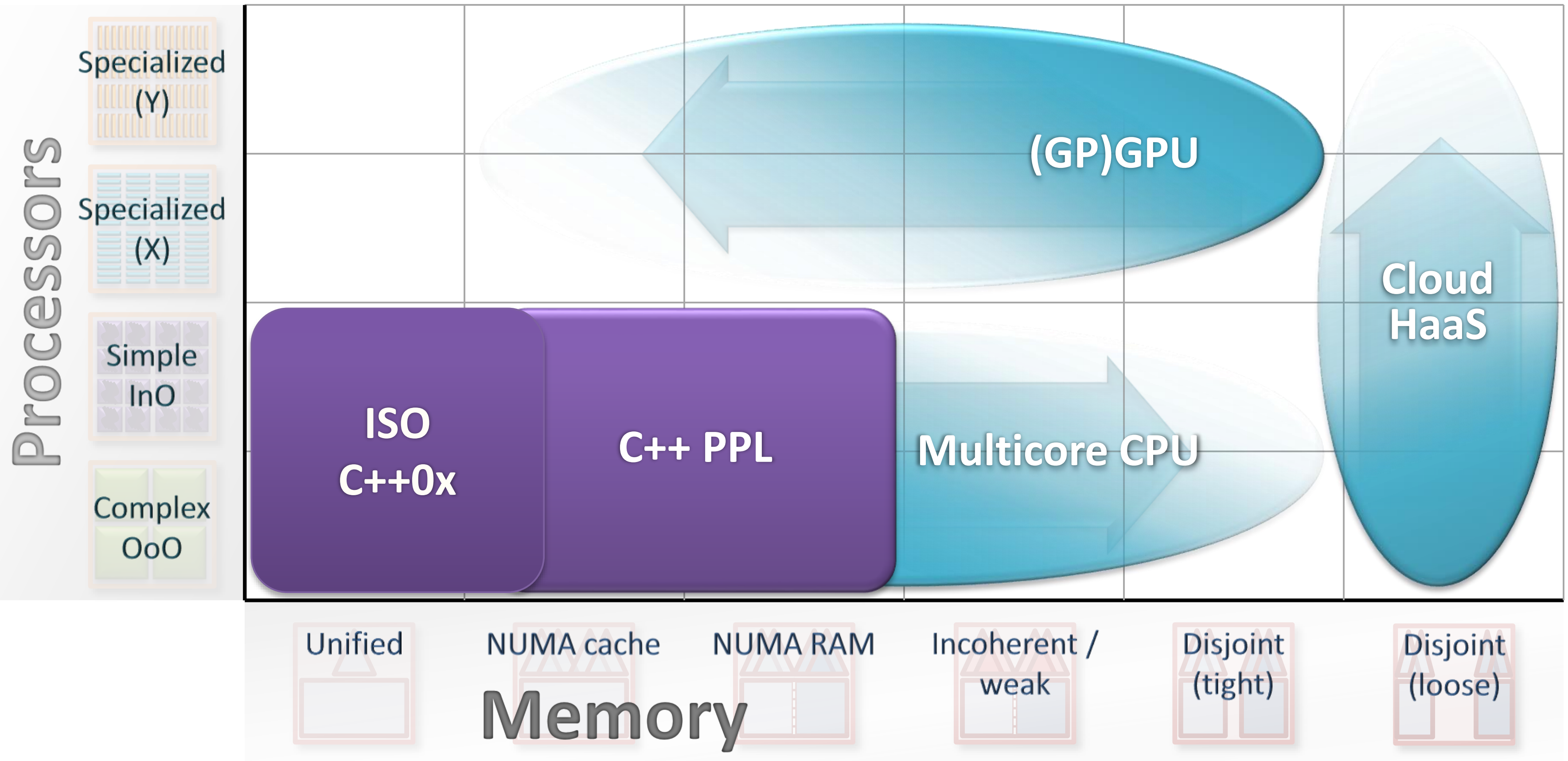
Hardware Evolution



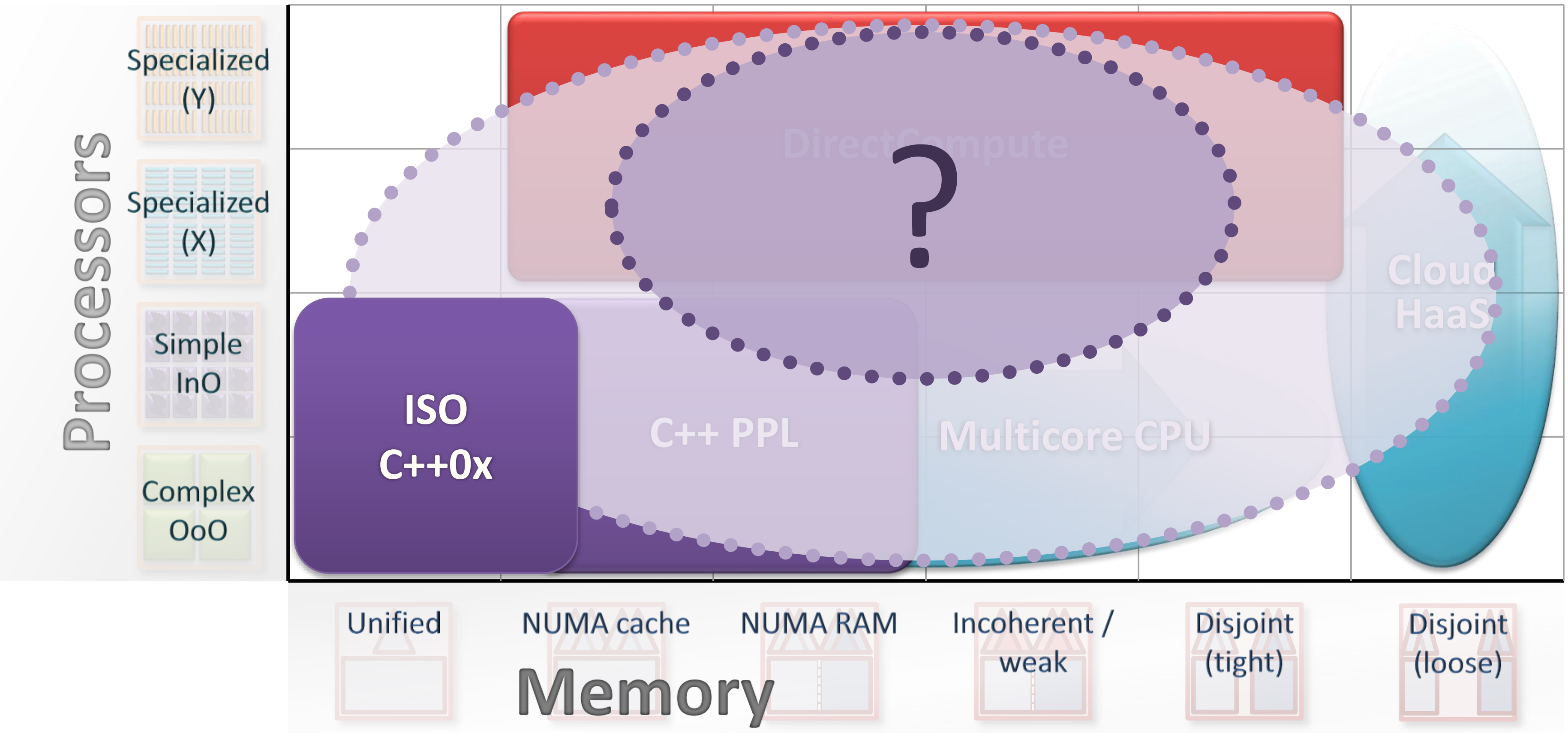
Programming Models & Languages



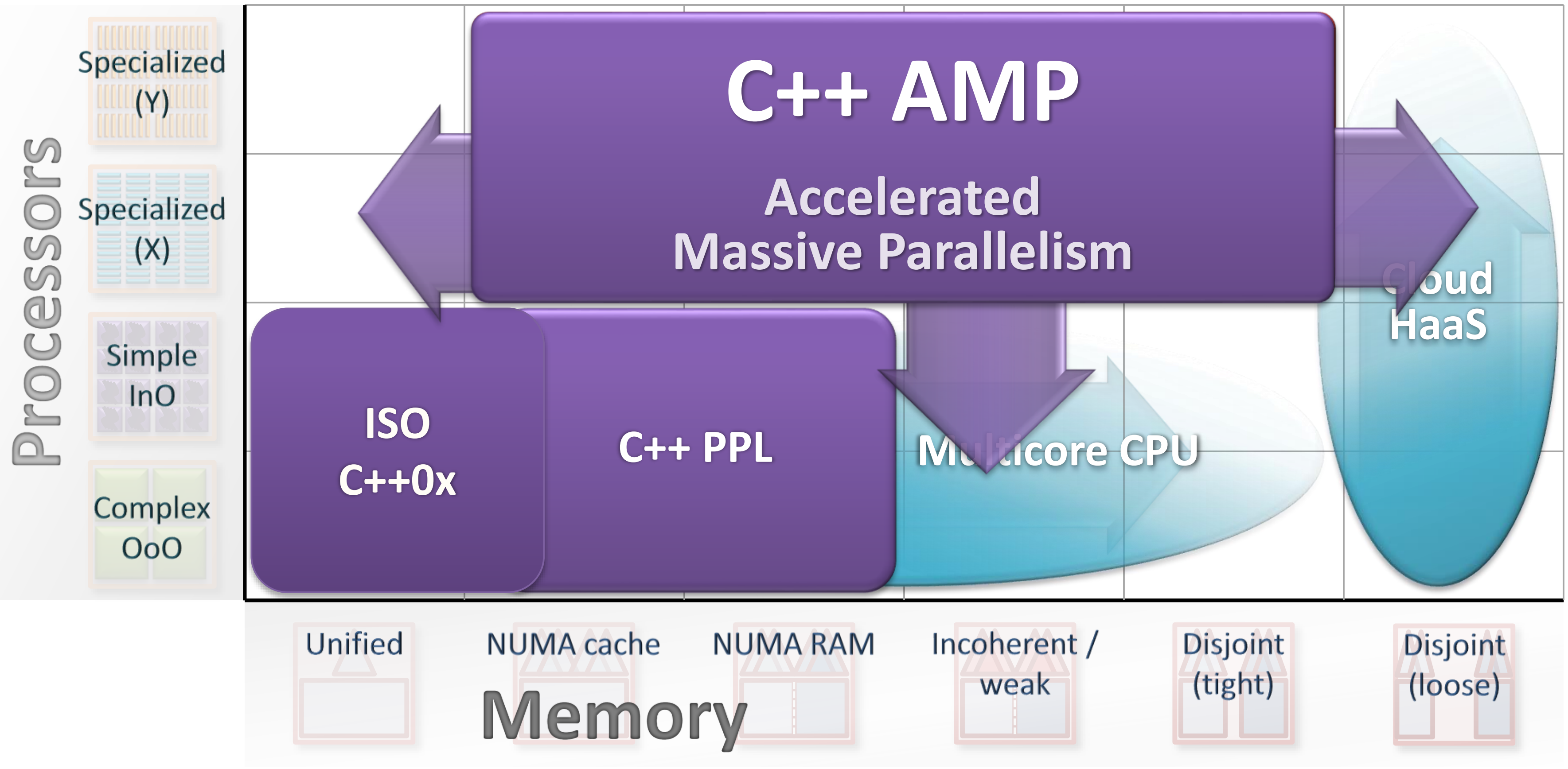
Programming Models & Languages



Programming Models & Languages



Programming Models & Languages



Matrix Multiply

Convert this (serial loop nest)

```
void MatrixMult( float* C, const vector<float>& A, const vector<float>& B,
                 int M, int N, int W )
{
  for (int y = 0; y < M; y++)
    for (int x = 0; x < N; x++) {
      float sum = 0;
      for(int i = 0; i < W; i++)
        sum += A[y*W + i] * B[i*N + x];
      C[y*N + x] = sum;
    }
}
```

Matrix Multiply

Convert this (serial loop nest)

```
void Mat

{
   for (int
      for (ir
         floa
         for(i
            su
         C[y*
      }
}
```

… to this (parallel loop, CPU or GPU)

```
void MatrixMult(  float* C, const vector<float>& A,  const vector<float>& B,
                  int M, int N, int W )
{
   array_view<const float,2> a(M,W,A), b(W,N,B);
   array_view<writeonly<float>,2> c(M,N,C);

   parallel_for_each( c.grid, [=](index<2> idx) restrict(direct3d) {
      float sum = 0;
      for(int i = 0; i < a.x; i++)
         sum += a(idx.y, i) * b(i, idx.x);
      c[idx] = sum;
   } );
}
```

Why C++ AMP?

C++, not C

mainstream, programmable by millions

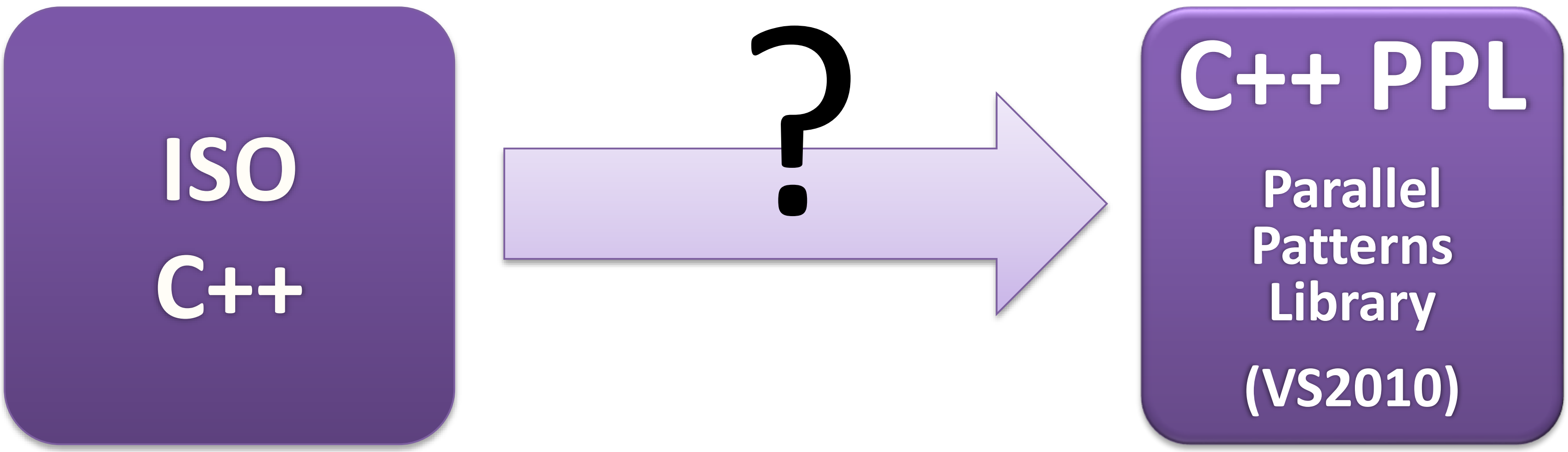
minimal, just one general language extension

portable, mix & match hardware from any vendor, one EXE

general and future-proof, designed to cover the full range of hardware heterogeneity – hardware is still in motion

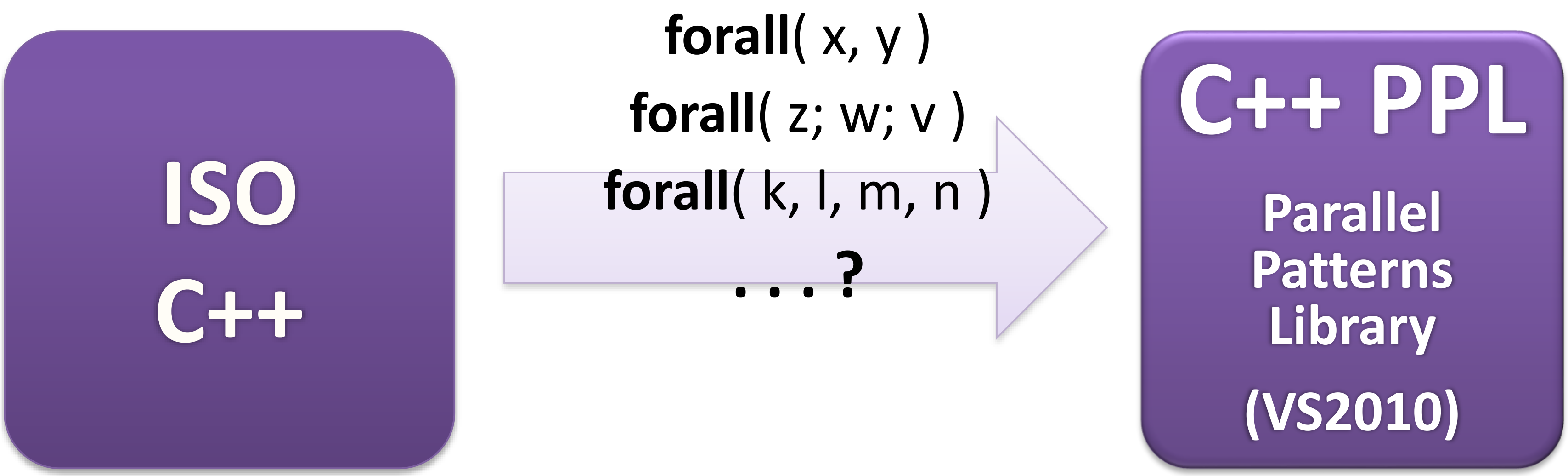
Language Design: Parallelism Phase 1

Single-core to multi-core



Language Design: Parallelism Phase 1

Single-core to multi-core



Language Design: Parallelism Phase 1

Single-core to multi-core

ISO C++0x

C++ PPL

Parallel Patterns Library

(VS2010)

```
parallel_for_each(
   items.begin(), items.end(),
   [=]( Item e )
{
   … your code here …
} );
```

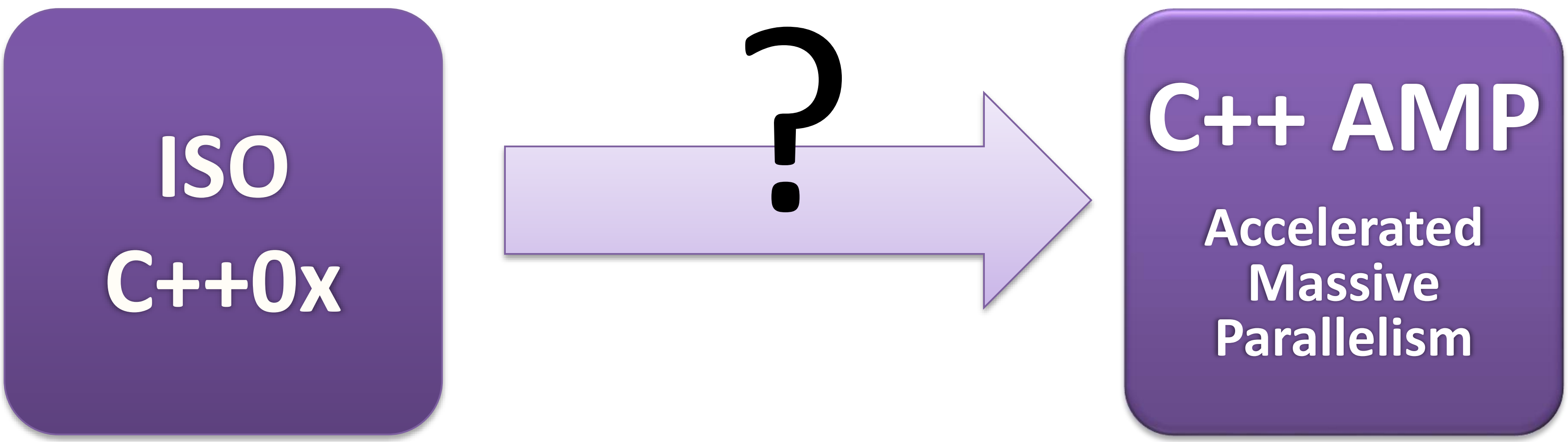
1

language feature for multicore

and STL, functors, callbacks, events, ...

Language Design: Parallelism Phase 2

Multi-core to hetero-core



Language Design: Parallelism Phase 2

Multi-core to hetero-core

ISO C++0x

restrict

C++ AMP

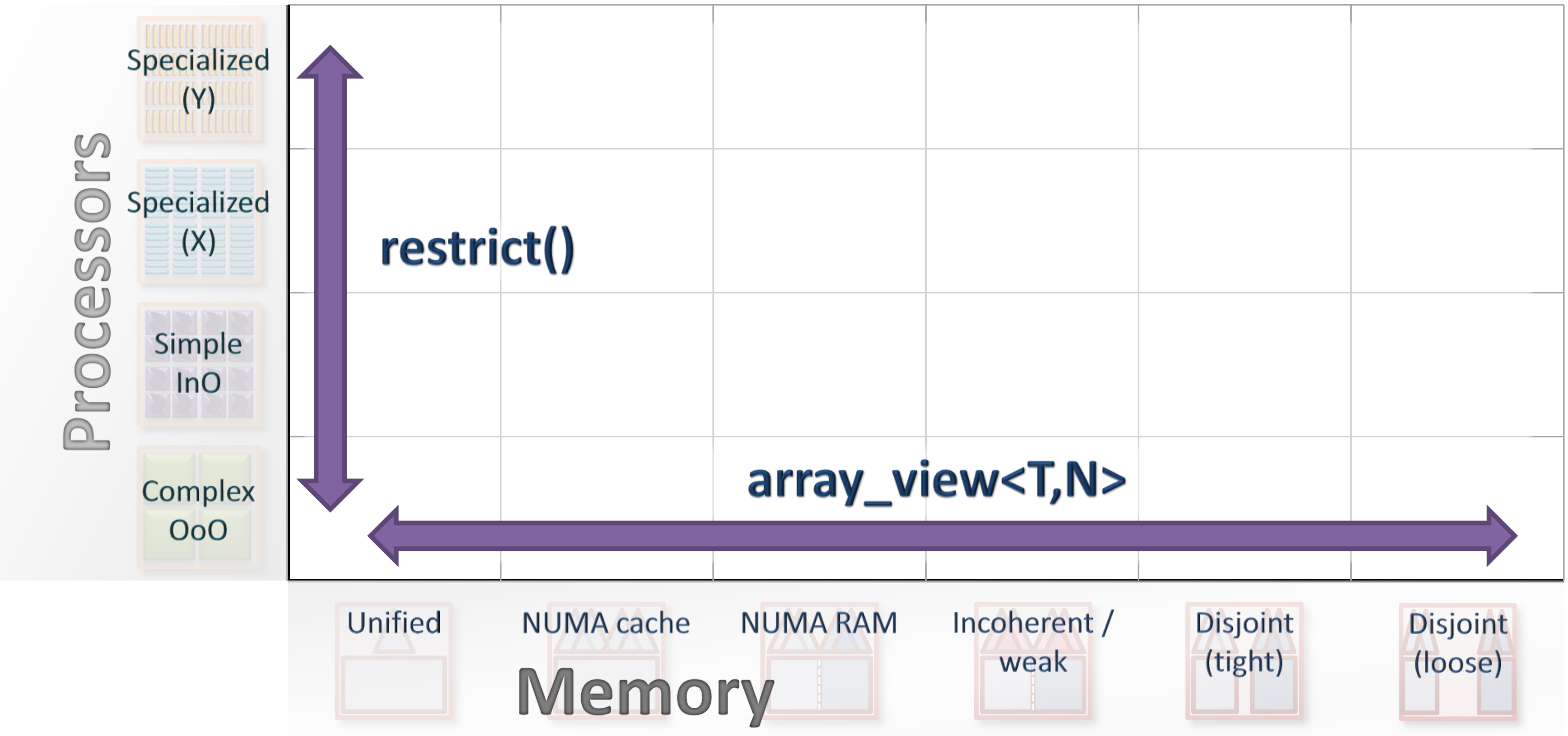
Accelerated Massive Parallelism

```
parallel_for_each(
   items.grid,
   [=](index<2> i) restrict(direct3d)
{
   … your code here …
} );
```

1

language feature for heterogeneous cores

C++ AMP at a Glance



restrict()

- **Problem:** Some cores don't support the entire C++ language.
- **Solution:** General restriction qualifiers enable expressing language subsets within the language. Direct3d math functions in the box.

Example

```
double sin( double ) restrict(cpu,direct3d);          // 1: same code for either

double cos( double );                                 // 2a: general code
double cos( double ) restrict(direct3d);              // 2b: specific code

parallel_for_each( c.grid, [=](index<2> idx) restrict(direct3d) {
   …
   sin( data.angle );  // ok
   cos( data.angle );  // ok, chooses overload based on context
   …
});
```

restrict()

- Initially supported restriction qualifiers:
  - **restrict(cpu)**: The implicit default.
  - **restrict(direct3d)**: Can execute on any DX11 device via DirectCompute.
    - Restrictions follow limitations of DX11 device model (e.g., no function pointers, virtual calls, goto).

- Potential future directions:
  - **restrict(pure)**: Declare and enforce a function has no side effects. Great to be able to state declaratively for parallelism.
  - General facility for language subsets, not just about compute targets.

Functionally Restricted Processors

```
int  myFunction( int x, int y) restrict(amp)
{
   // only call similarly restricted functions
   // no partial word data types
   // no exceptions, try or catch
   // no goto
   // no indirect function calls
   // no varargs routines
   //  no new/delete
   // very restricted pointer usage
   // no static variables
}
```

```
// overload resolution
int myFunction(int x, int y) restrict(amp);
int myFunction(int x, int y) restrict(cpu);
int myFunc(int x) restrict(cpu,amp);
```

```
// evolution over time
int myFunction(int x, int y) restrict(amp:2);
```

array_view

- **Problem:** Memory may be flat, nonuniform, incoherent, *and/or* disjoint.
- **Solution:** Portable view that works like an N-dimensional “iterator range.”
  - Future-proof: No explicit .copy()/.sync(). As needed by each actual device.

Example

```
void MatrixMult( float* C, const vector<float>& A, const vector<float>& B,
                 int M, int N, int W )
{
  array_view<const float,2> a(M,W,A), b(W,N,B); // 2D view over C++ std::vector
  array_view<writeonly<float>,2> c(M,N,C);      // 2D view over C array

  parallel_for_each( c.grid, [=](index<2> idx) restrict(direct3d) {
    …
  } );
}
```

Example Matrix Multiplication

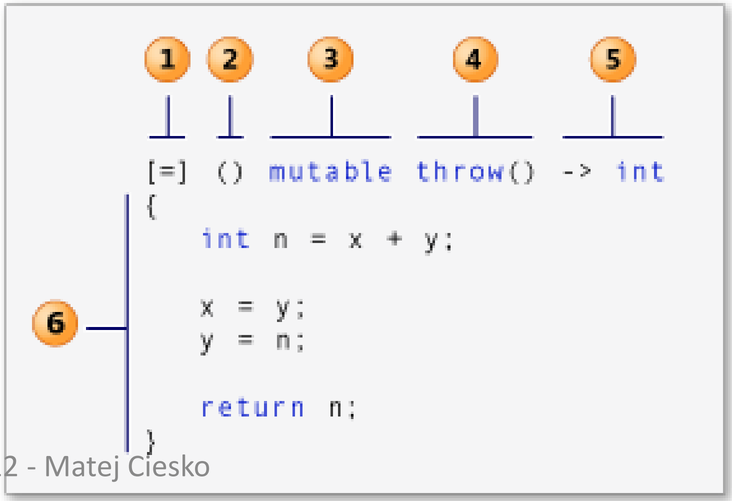
```
void MatrixMultiplySerial( vector<float>& vC,
   const vector<float>& vA,
   const vector<float>& vB, int M, int N, int W )
{




  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<float,2> c(M,N,vC);
  c.discard_data();
  parallel_for_each(c.extent,
    [=](index<2> idx) restrict(amp) {
        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;
    }
   );
   c.synchronize();
}
```

Disjoint Address Space, One Name Space

- array_view<> enables copy-as-needed semantics
- Build on C++ lambda capture rules
- Explicit allocation via array<> & accelerators
- → Future-proofing for shared memory architectures

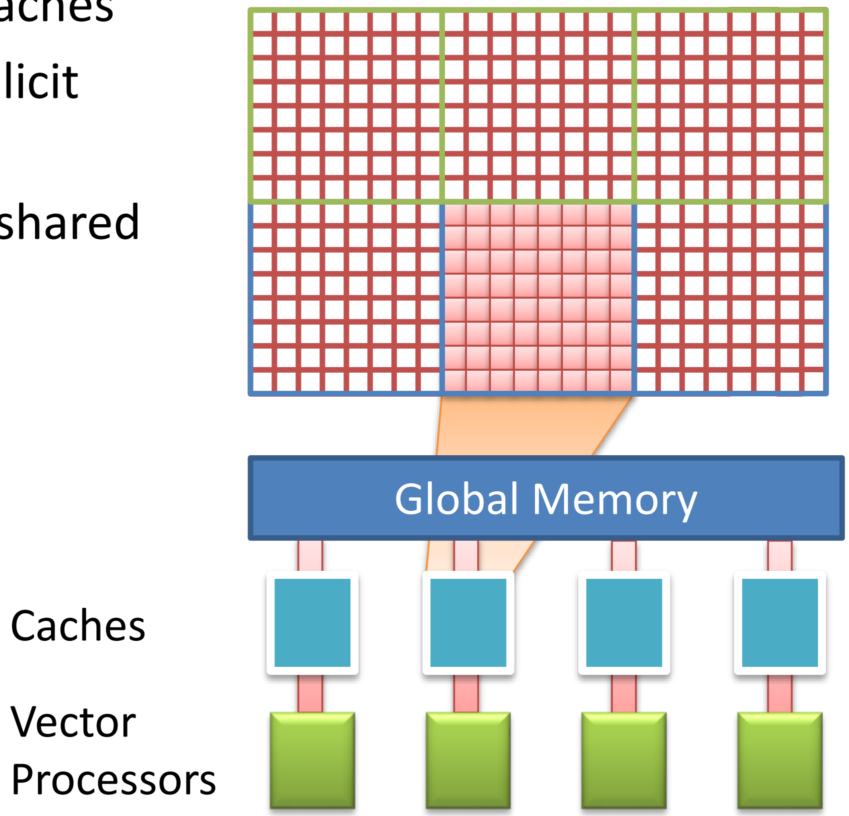
```
array_view<const float,2> a(M,W,vA), b(W,N,vB);
array_view<float,2> c(M,N,vC);

parallel_for_each(c.extent,
    [=](index<2> idx) restrict(amp) {
            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;
    }
 );
```



Explicit Caching

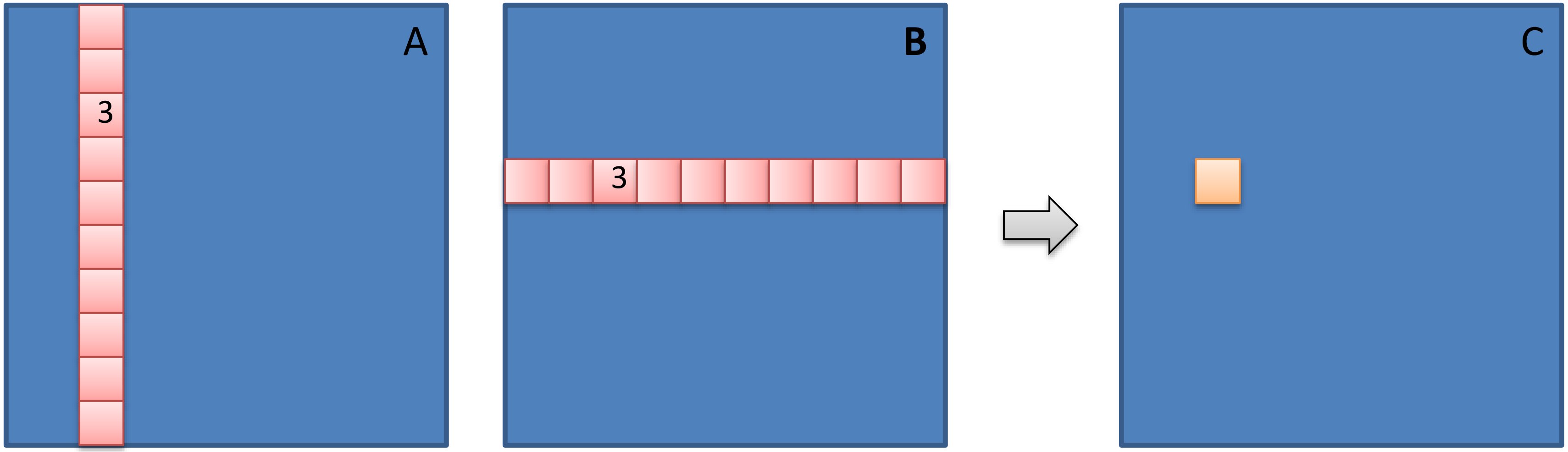
- Embrace a multi-core with private caches
- Extend data-parallel model with explicit tiling
- Within tiles, barrier coordination + shared storage

Key uses:
Capture cross-thread data reuse
Fast communication
Reductions, Scans
Nearest-neighbor



Improve Matrix Multiply with tiling

In block form, each block-level multiply-accumulate can be done in cache



Explicit Caching: Tiles

```
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<float,2> c(M,N,vC);  c.discard_data();
   parallel_for_each(c.extent,
      [=] (index<2> idx) restrict(amp)
      {
         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<float,2> c(M,N,vC);  c.discard_data();
   parallel_for_each(c.extent.tile< TS, TS >(),
      [=] (tiled_index< TS, TS> t_idx) restrict(amp)
      {
         int row = t_idx.global[0];
         int col =   t_idx.global[1];

         float sum = 0.0f;
         for(int k = 0; k < W; k++)
            sum += a(row, k) * b(k, col);

         c[t_idx.global] = sum;
      } );
}
```

Explicit Caching: Coordination

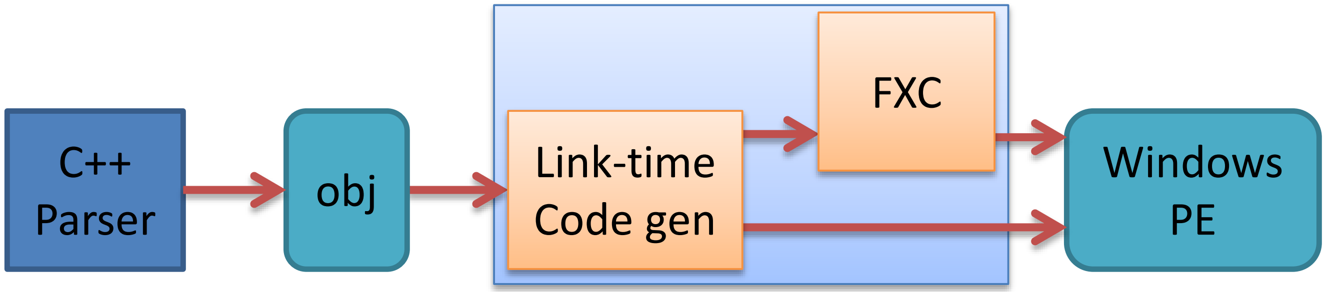
```
void MatrixMultTiled(vector<float>& vC, const vector<float>& vA,
const vector<float>& vB, int M, int N, int W )
{
   static const int TS = 16;
   array_view<const float,2> a(M, W, vA), b(W, N, vB);
   array_view<float,2> c(M,N,vC);  c.discard_data();
   parallel_for_each(c.extent.tile< TS, TS >(),
      [=] (tiled_index< TS, TS> t_idx) restrict(amp)  {

         int row = t_idx.global[0]; int col = t_idx.global[1];
         float sum = 0.0f;




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


         c[t_idx.global] = 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<float,2> c(M,N,vC);  c.discard_data();
   parallel_for_each(c.extent.tile< TS, TS >(),
      [=] (tiled_index< TS, TS> t_idx) restrict(amp
         tile_static float locA[TS][TS], locB[TS][TS];
         int row = t_idx.local[0]; int col = t_idx.local[1];
         float sum = 0.0f;
         for (int i = 0; i < W; i += 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;
      } );
}
```

Phase 1

Phase 2

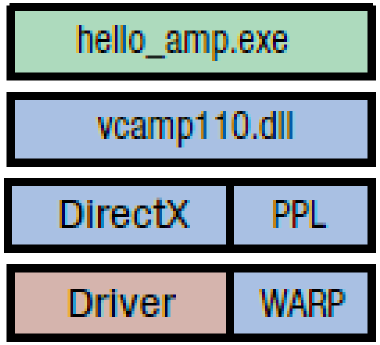
C++ compilation & deployment



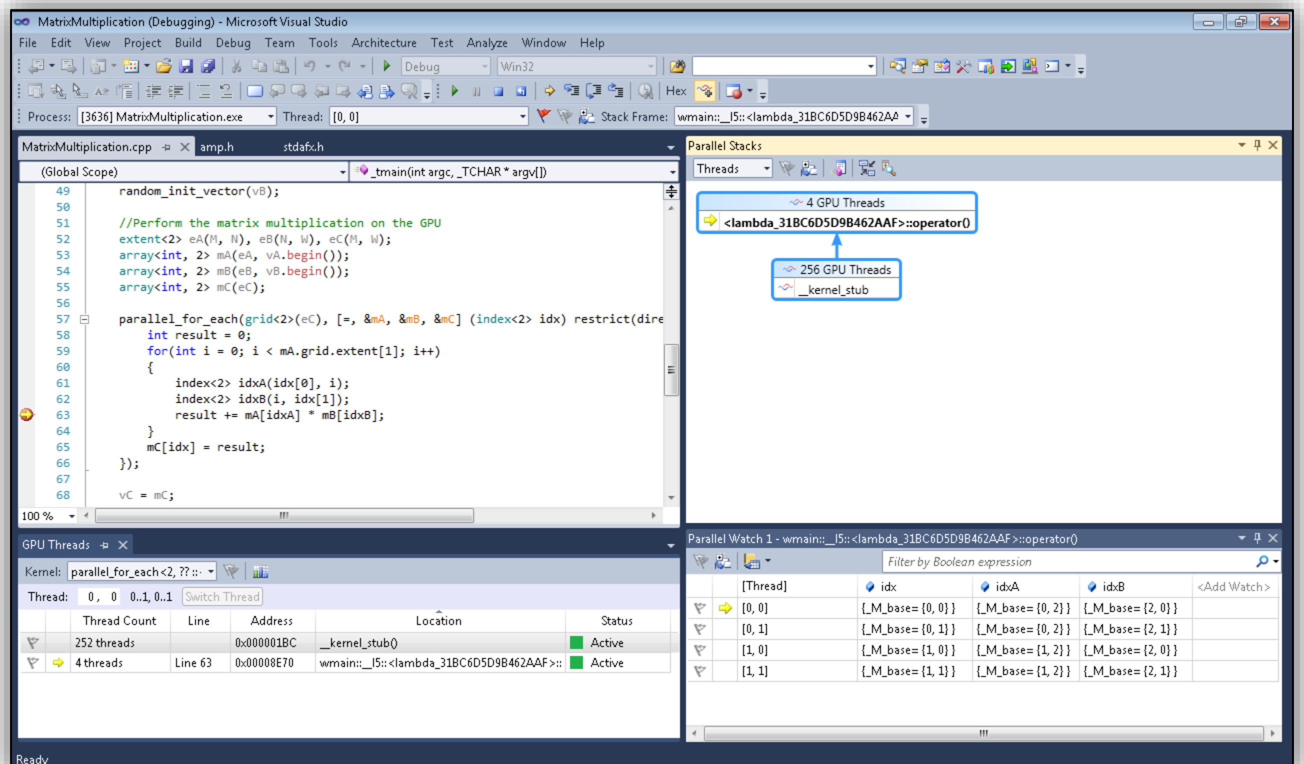
C++ compilation units, separate until linking

Single “fat” binary with native CPU code and DX11 bytecode

After launch, hardware-specific JIT, same as for graphics

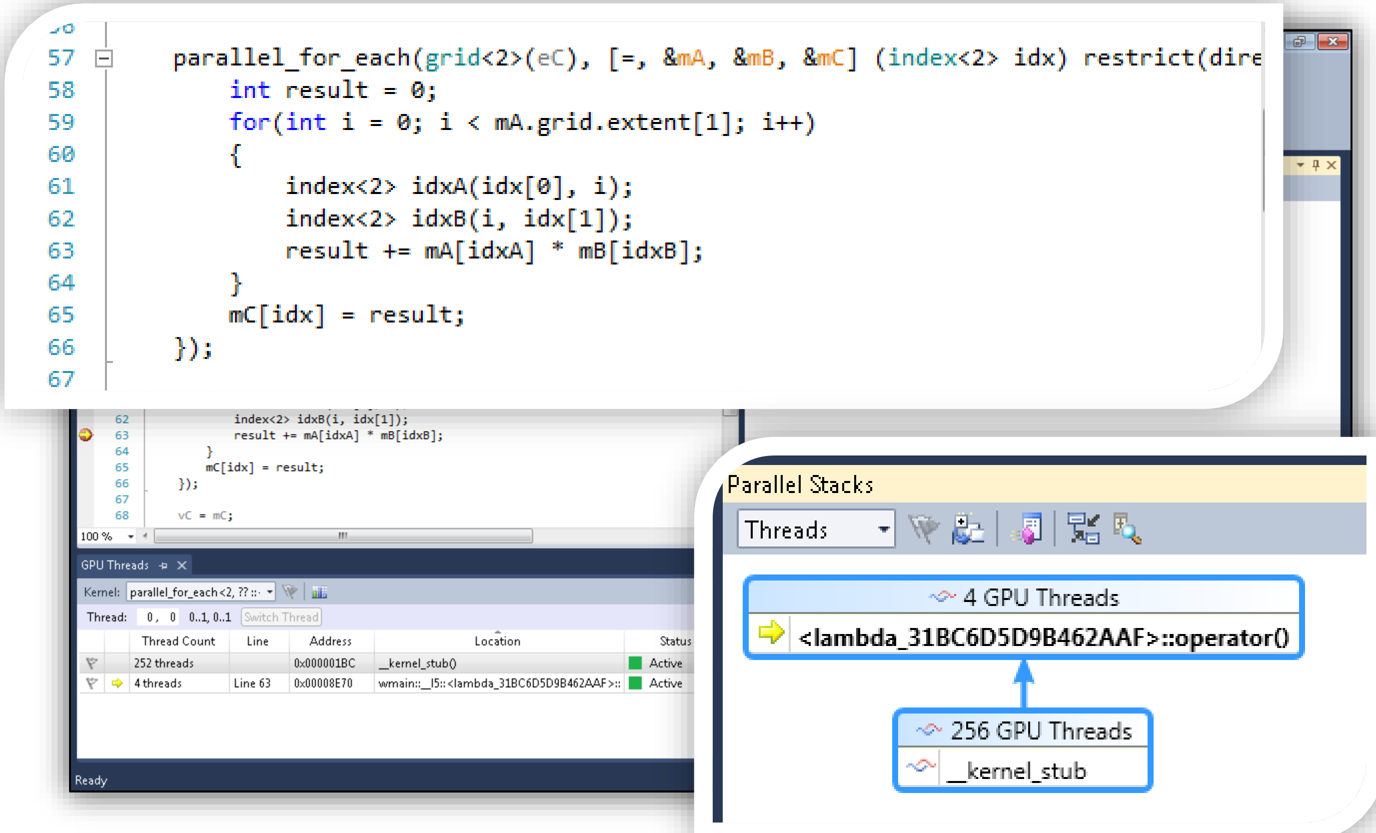


GPU Debugging



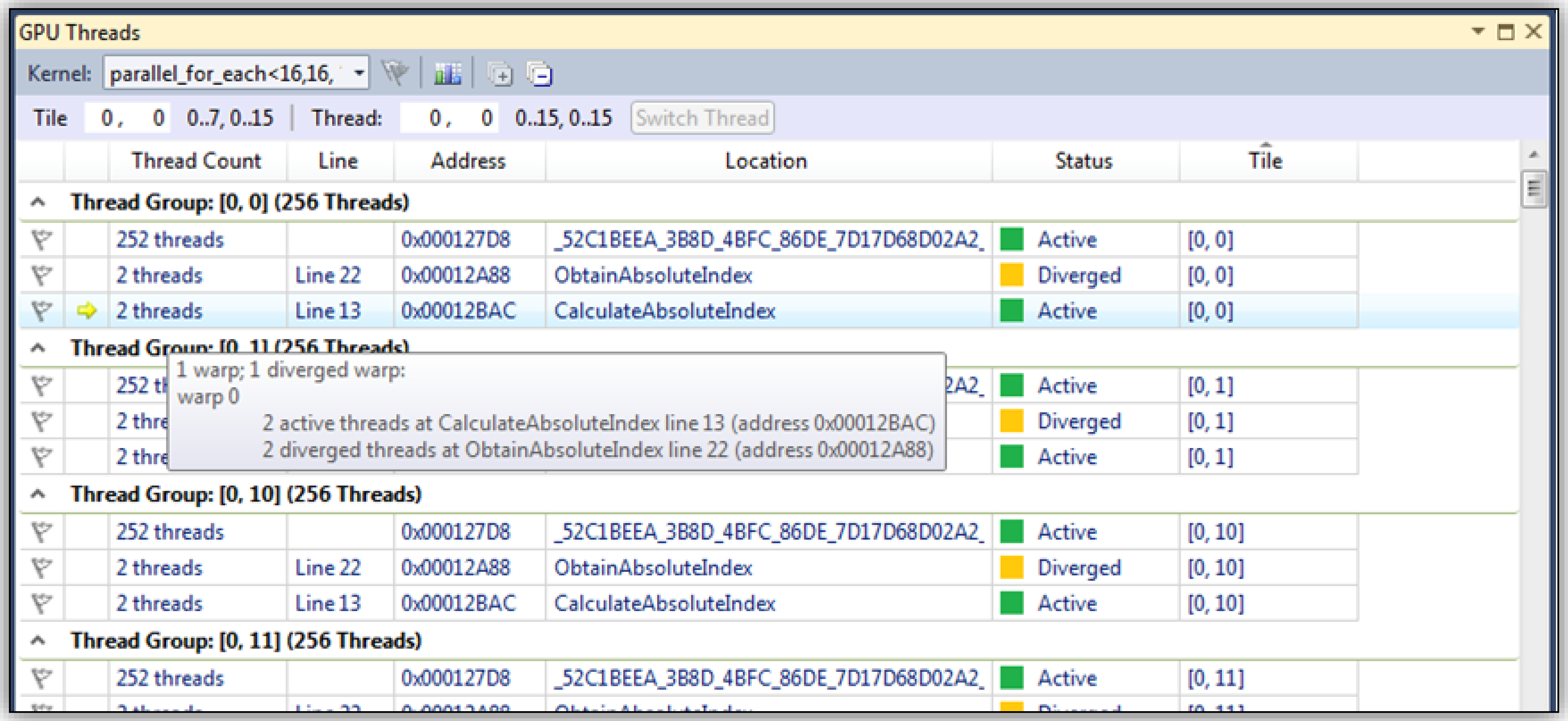
Bring CPU debugging experience to the GPU

GPU Debugging

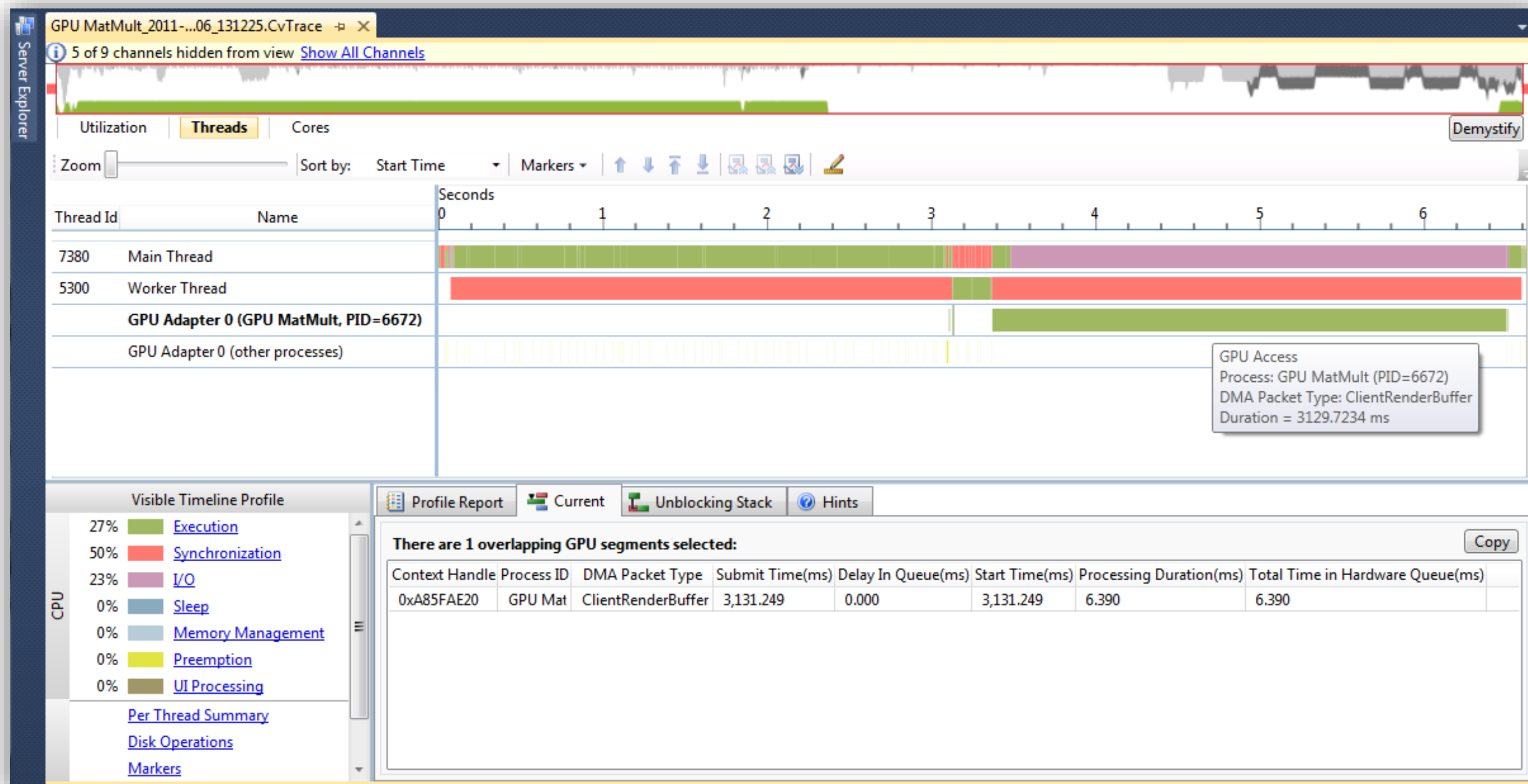


Bring CPU debugging experience to the GPU

GPU Debugging



GPU Profiling



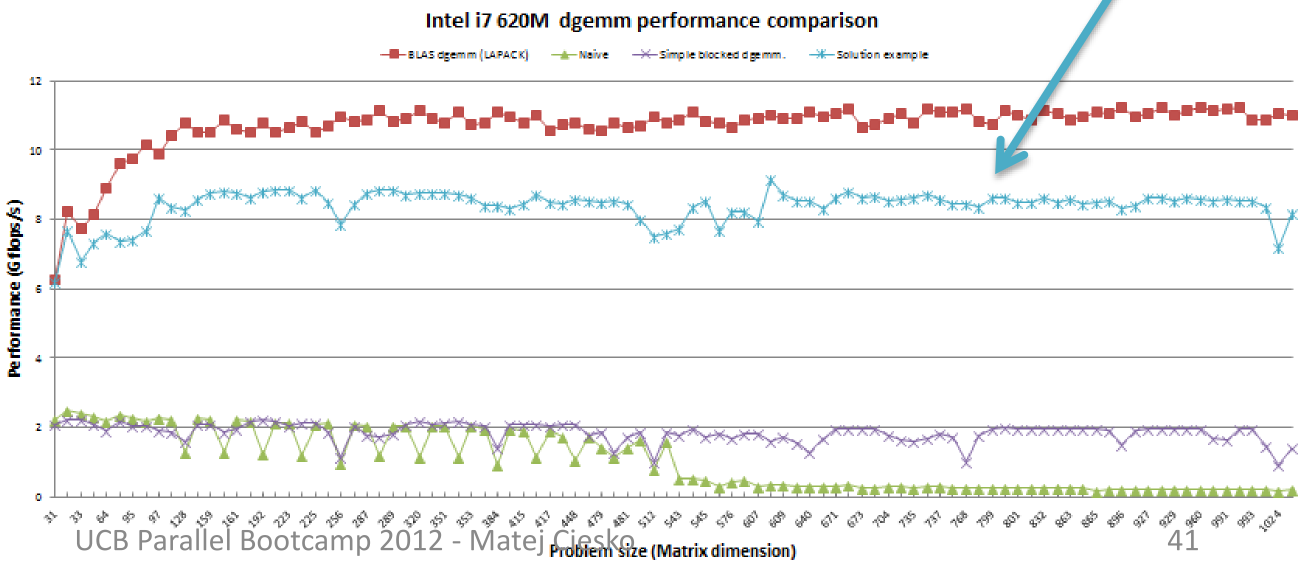
PORTING CS 267

Assignment 1
Serial Performance

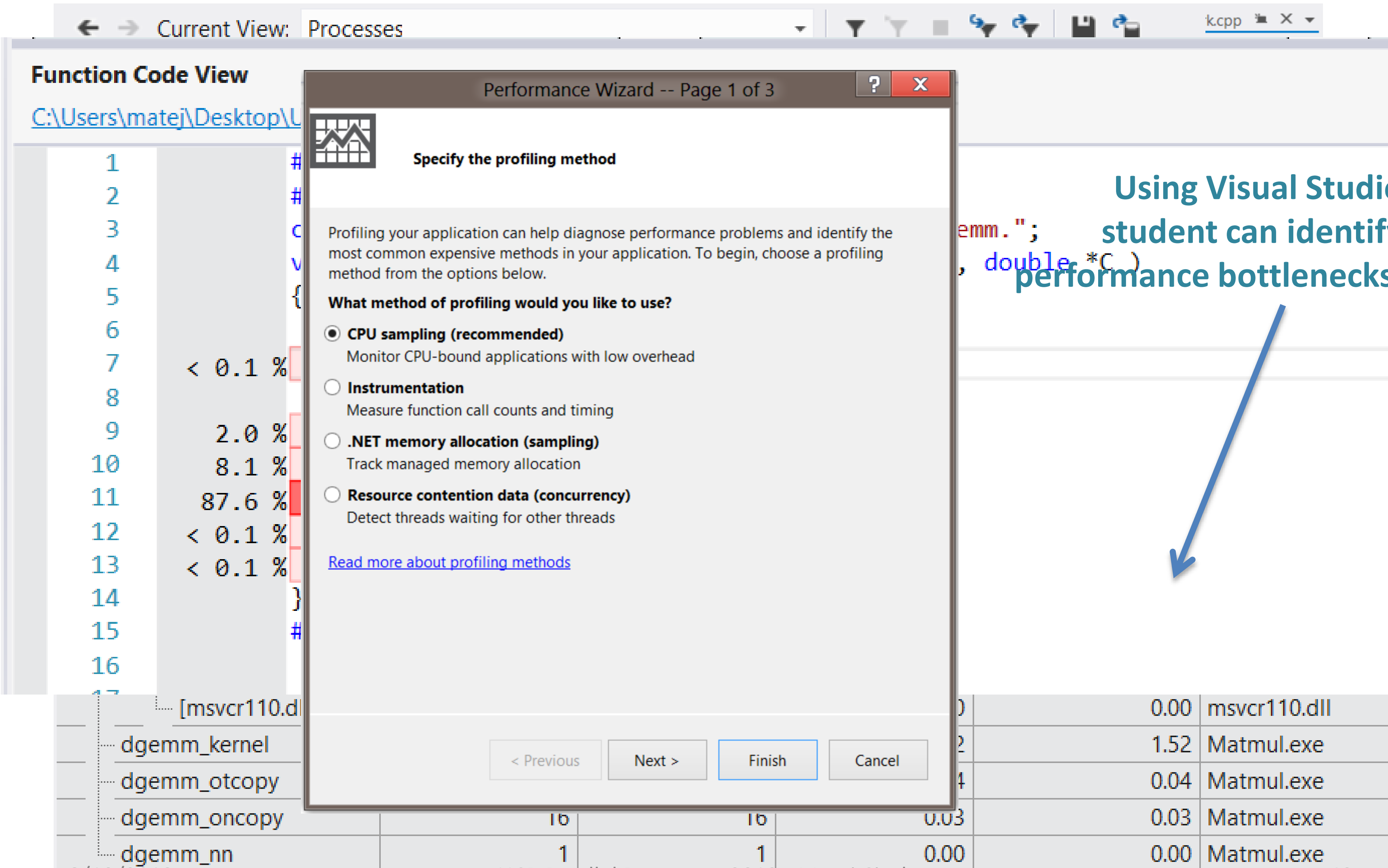
- **Assignment target:** Single thread performance
- **Learning goal:**
  - Identify performance bottlenecks
  - Use libraries
  - Optimize kernel
- **Problem:** $C \leftarrow A \times B + C \qquad A, B, C \in \mathbb{R}^{n \times n}$
- **Solution:**
  Optimizations
  - Blocking
  - Loop unrolling
  - Prefetching
  - Vectorization
  - Repacking data
  - [...]

Performance:

Using Visual Studio students can achieve near optimal performance by applying standard optimization techniques.



Assignment 1 (Cont.)



Using Visual Studio student can identify performance bottlenecks.

Assignment 2
Parallel Programming

Using Visual Studio students can easily write parallel code in native threads, OpenMP, MPI, C++ AMP.

- **Assignment target & learning goal:**
  - Parallelize given problem
  - Distributed & shared memory parallelism
  - Accelerators (CUDA)
- **Problem:** n-body

  On Windows:
  - Native threads
  - OpenMP, PPL
  - MPI
  - C++ AMP

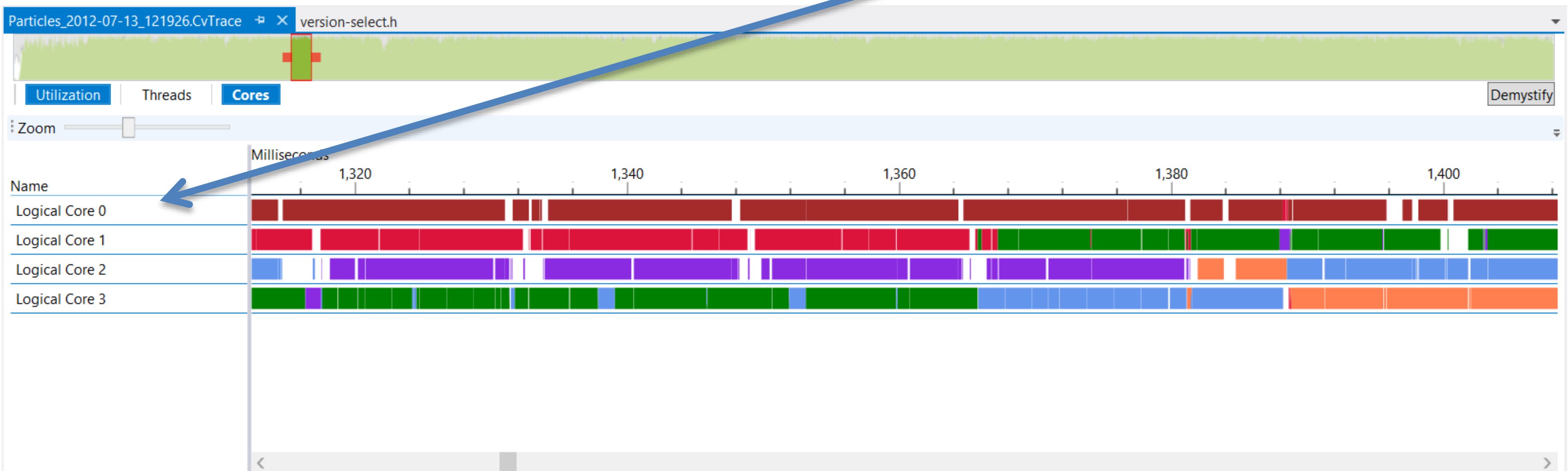
```
#include <amp.h>
array_view<particle_t, 1>  particles_av(n, particles);
for( int step = 0; step < NSTEPS; step++ )
{
  parallel_for_each(particles_av.extent, [=] (index<1> i) restrict(amp){
    particles_av[i].ax = 0;
    particles_av[i].ay = 0;
    for (int j = 0; j < n; j++ ){
      apply_force( particles_av[i], particles_av[j] );
    }
  });
  parallel_for_each(particles_av.extent, [=] (index<1> i) restrict(amp){
      move( particles_av[i], size );
  });
  particles_av.synchronize();
}
```

- Performance comparison: (initial/ trivial solution, n = 1000)

Threading	**OpenMP**	**PPL**	**MPI**	**C++ AMP***
3.49 sec	2.21 sec	2.33 sec	2.70 sec	2.41 sec

Assignment 2 (Cont.)

Using Visual Studio students can deep dive into synchronization patterns and timing.

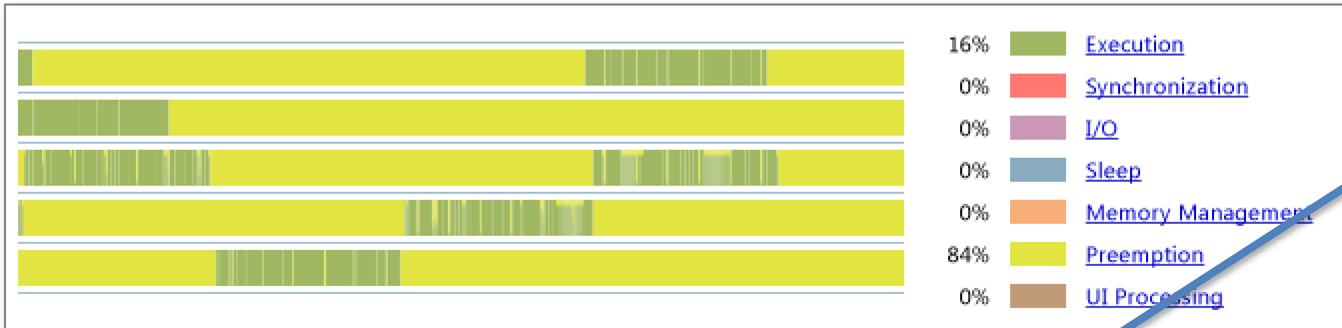


Thread ID	Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
7420	Worker Thread	3	12	25.00 %
8616	Worker Thread	2	48	4.17 %
6544	Worker Thread	2	44	4.55 %
7268	Worker Thread	2	41	4.88 %
6156	Main Thread	1	57	1.75 %
7148	Worker Thread	1	18	5.56 %

Assignment 2 (Cont.)

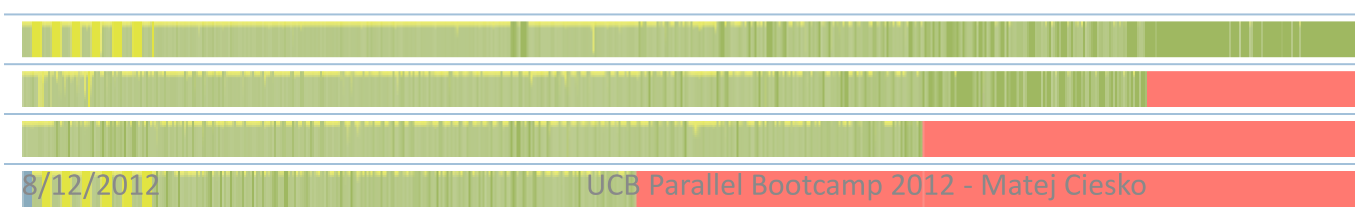
Using Visual Studio students can deep dive into synchronization patterns and timing.

- Oversubscription



- Lock Convoys

- Uneven Workload distribution



Lab

- Get hands-on experience using Visual Studio profiling and parallel debugging functionality
- Experiment with C++ AMP

Sources

- **AMP blog:** http://blogs.msdn.com/b/nativeconcurrency/
- **H. Sutter blog:** http://herbsutter.com/
- **NsfPPC:** http://www.nsfppc.net

Thank you for your attention!