

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**

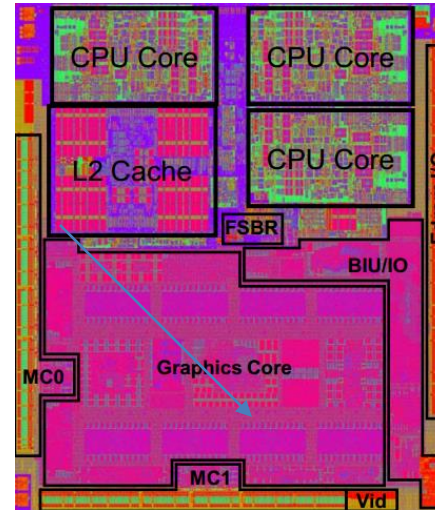
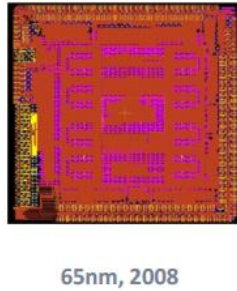
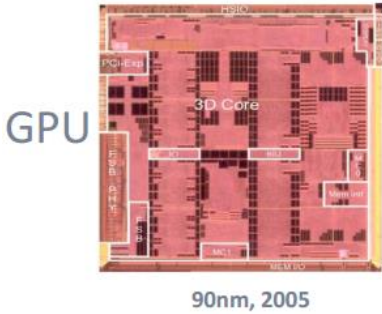
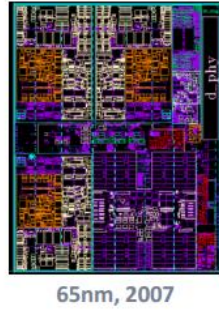
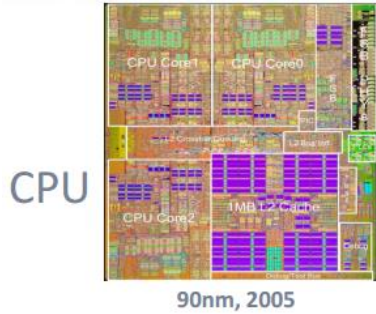
Put a **parallel supercomputer** on every desk, in every home, in every pocket.

**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

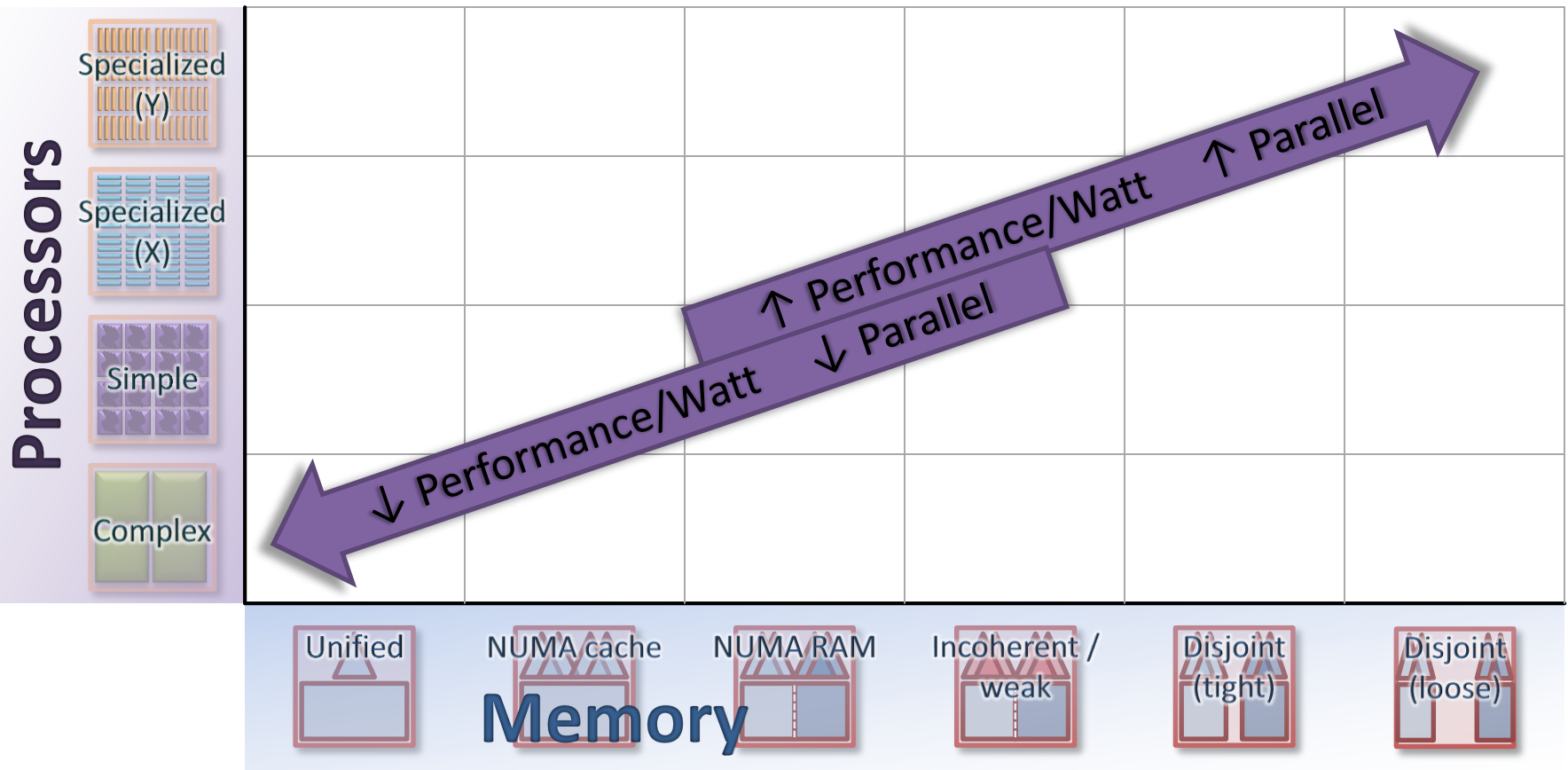


**commercially available  
to millions**

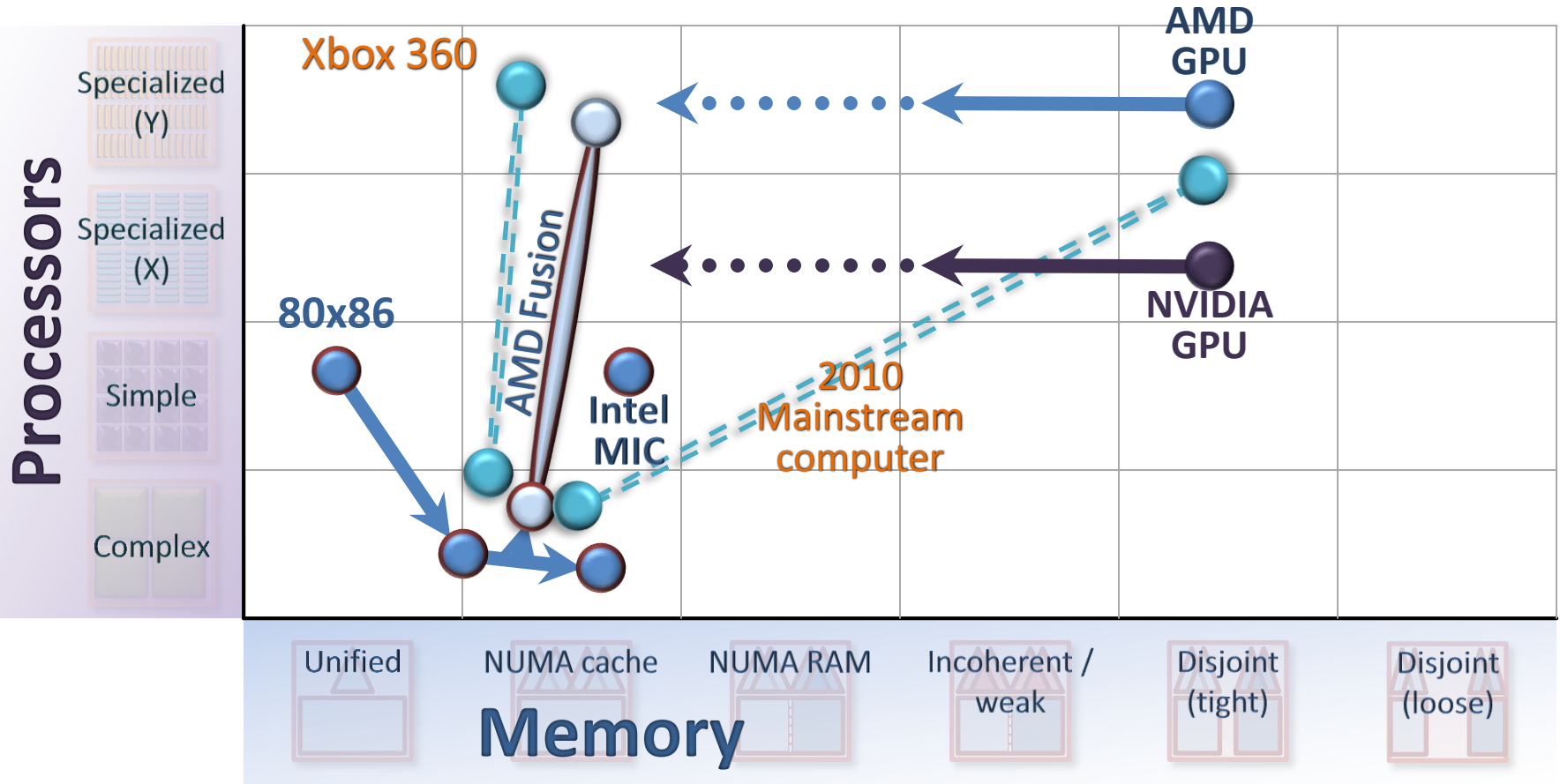
**commercially affordable  
for millions**

**commercially programmable  
by millions**

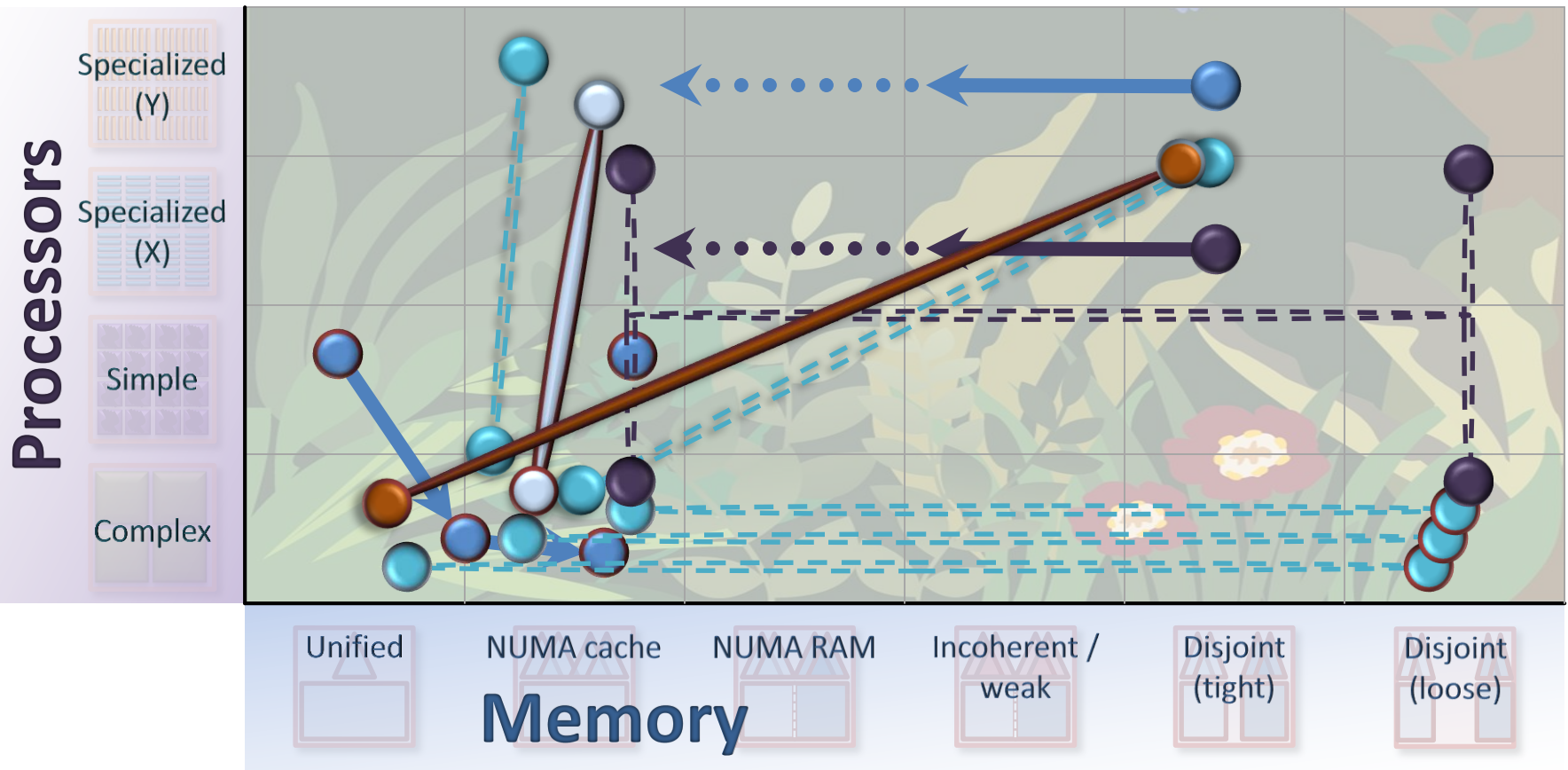
# Charting the Landscape



# Hardware

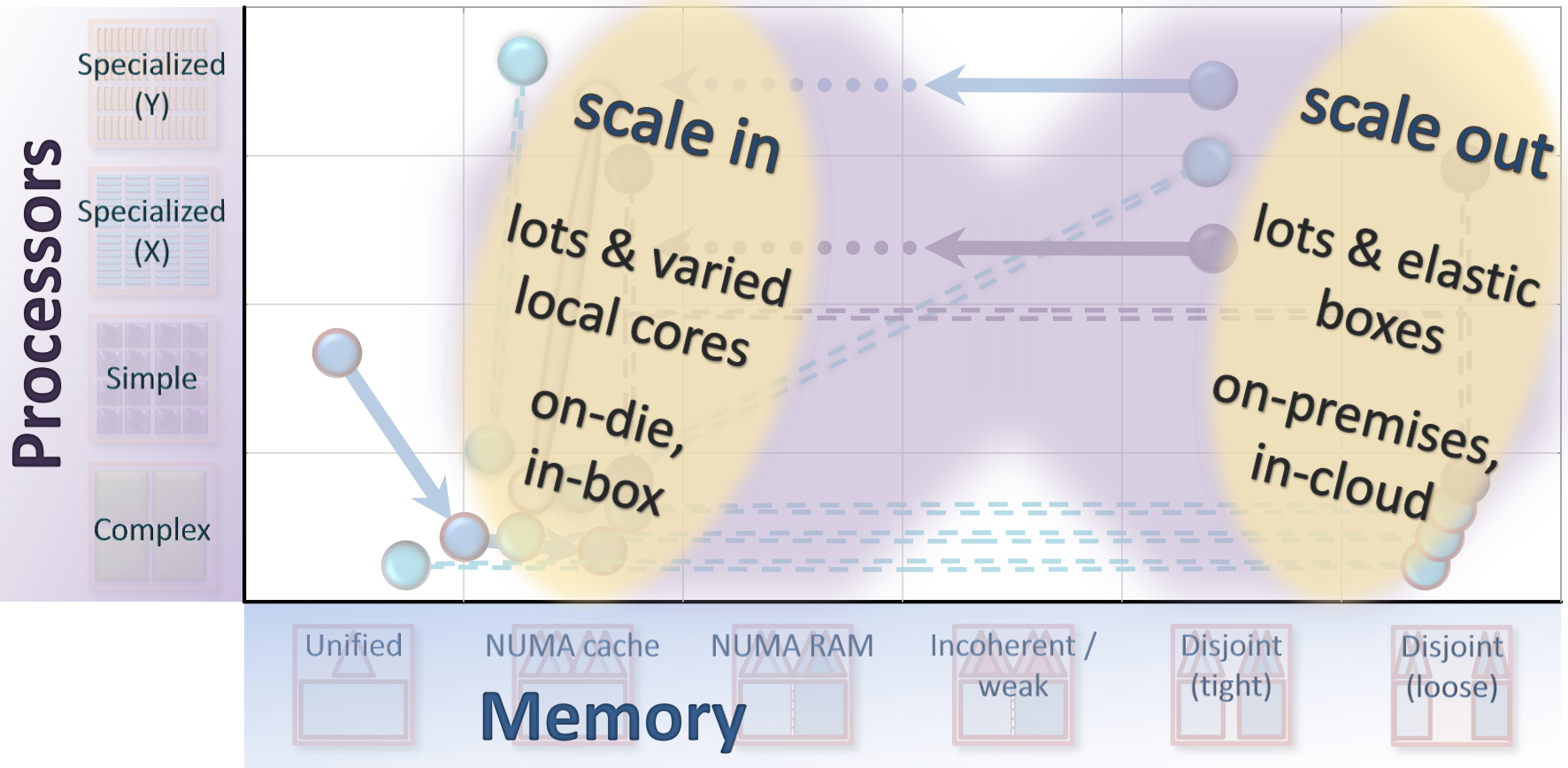


# The Jungle

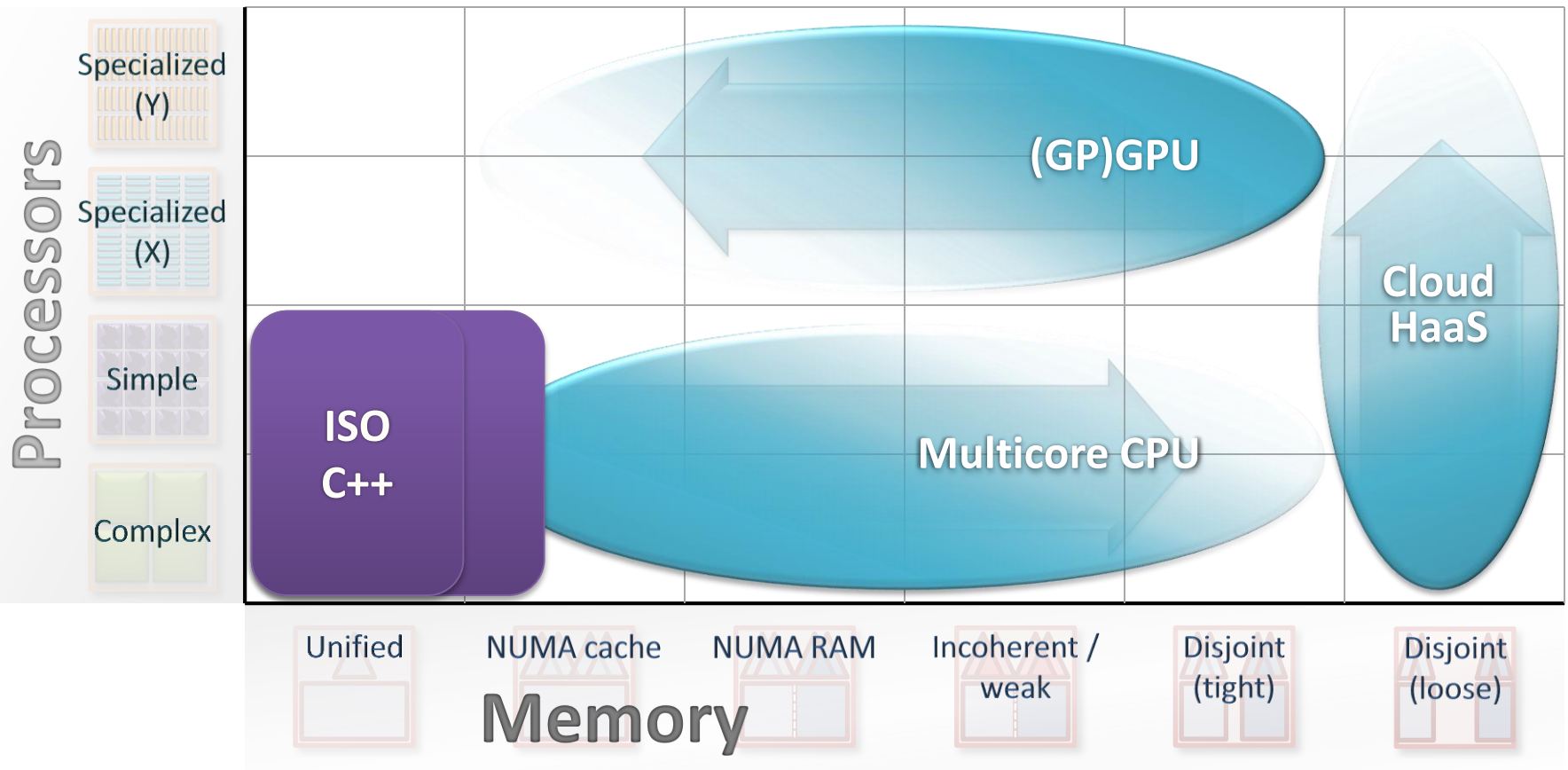




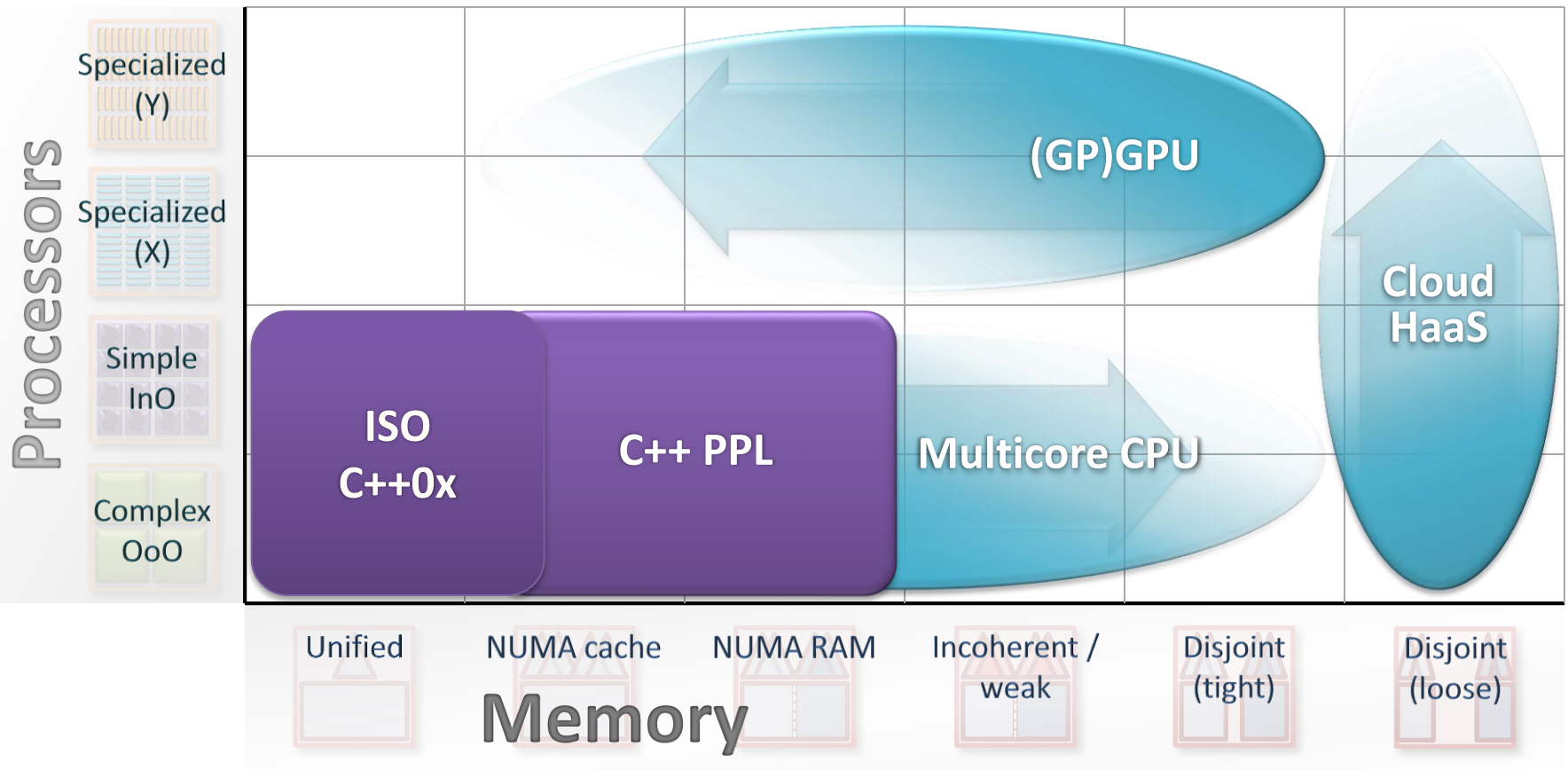
# Hardware Evolution



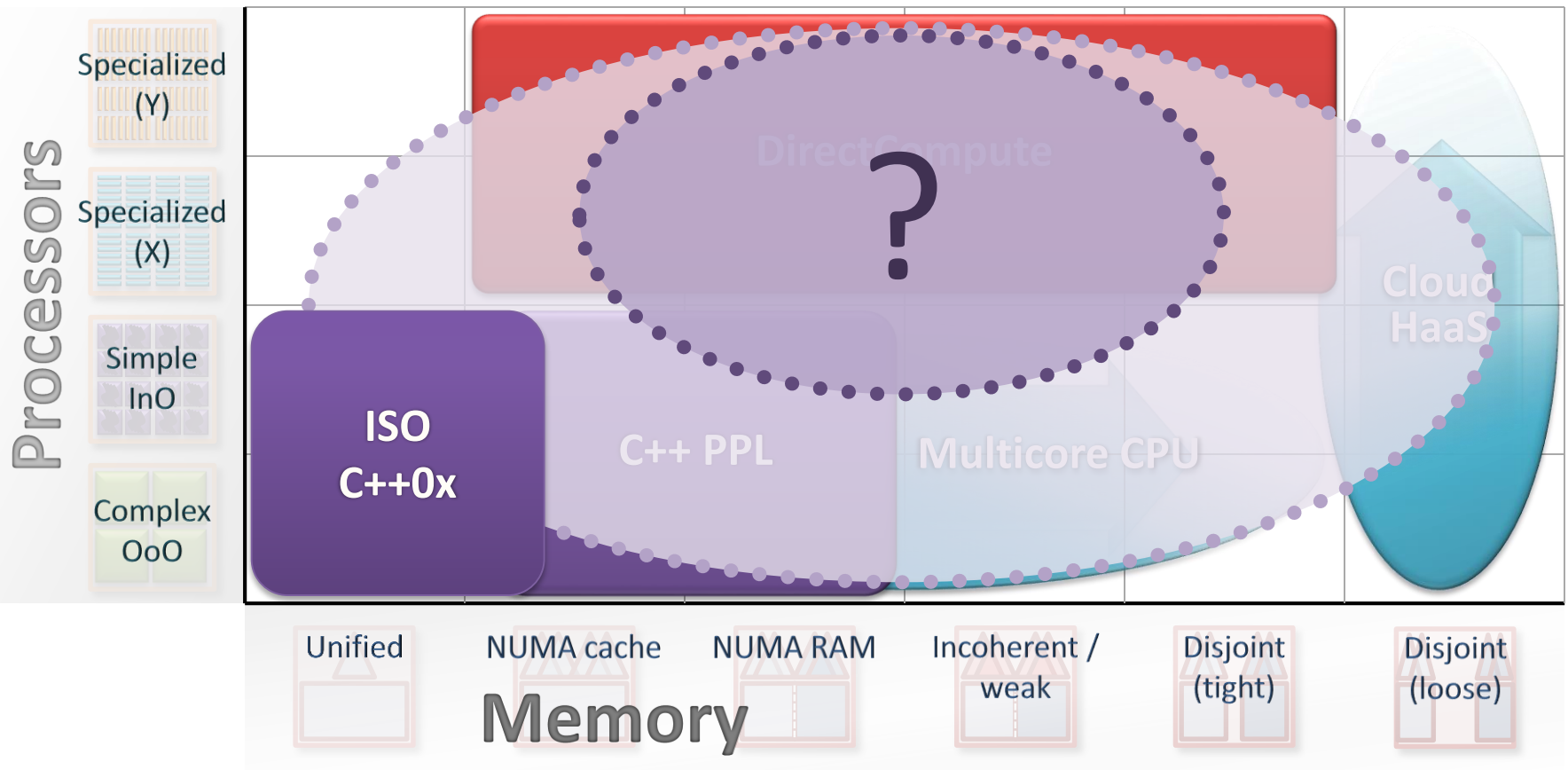
# Programming Models & Languages



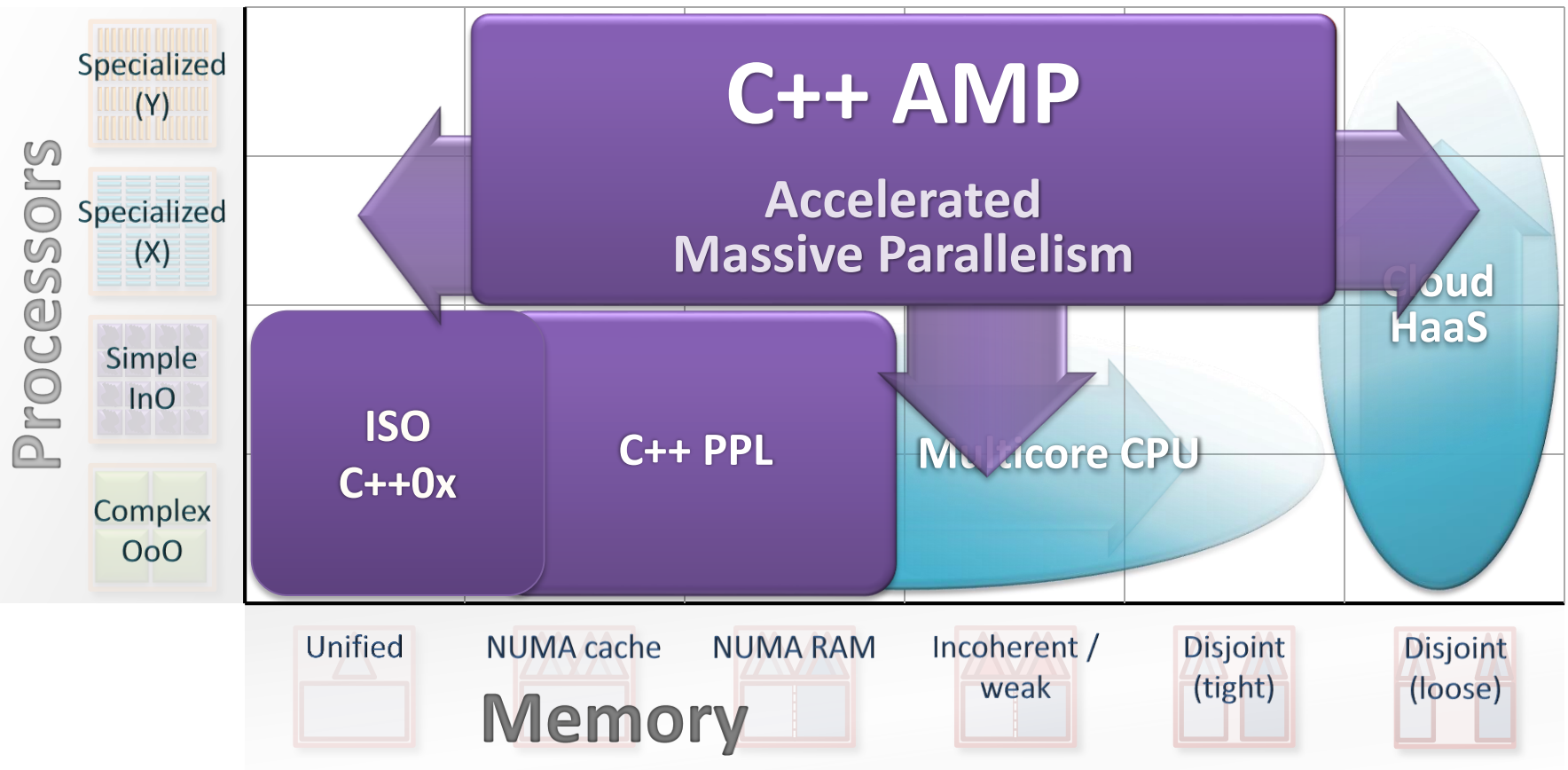
# Programming Models & Languages



# 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)

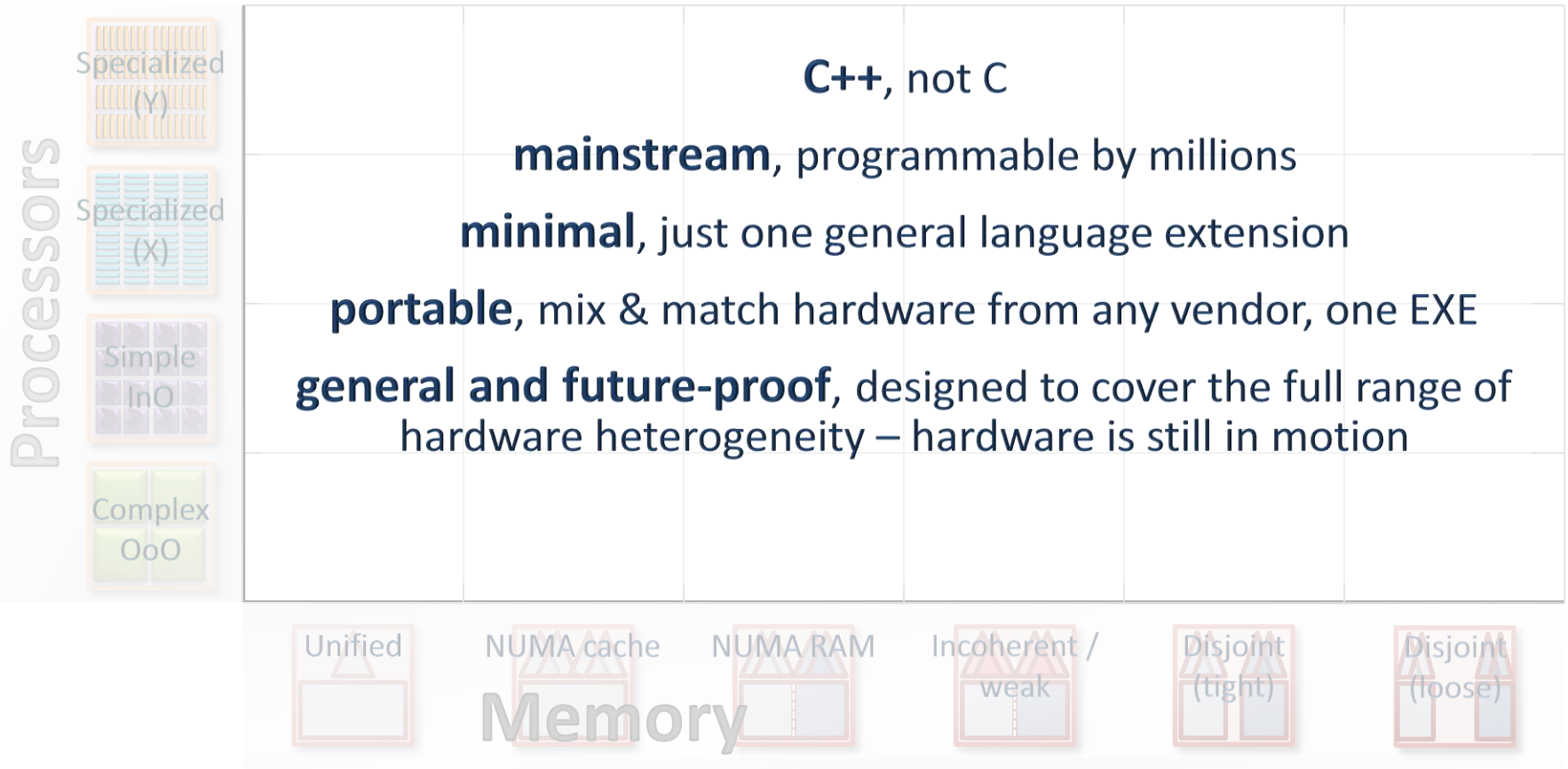
```
void MatMult(
{
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      float sum = 0;
      for (int k = 0; k < W; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
}
```

... 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;
  });
}
```

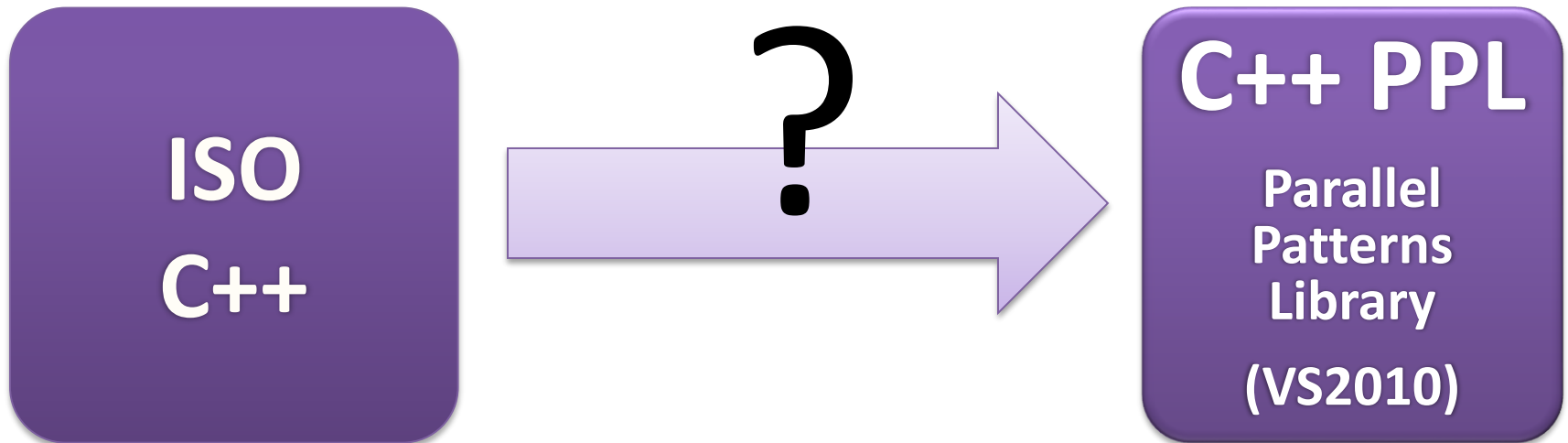
# Why C++ AMP?





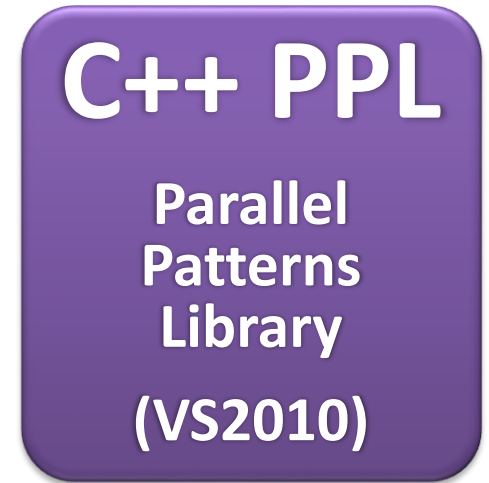
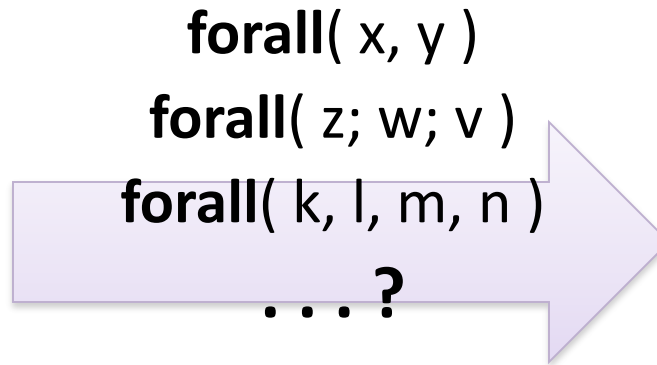
# Language Design: Parallelism Phase 1

*Single-core to multi-core*



# Language Design: Parallelism Phase 1

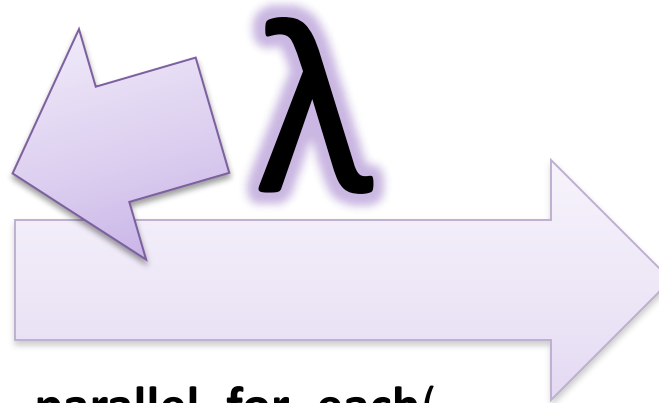
*Single-core to multi-core*



# Language Design: Parallelism Phase 1

*Single-core to multi-core*

ISO  
C++0x



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

C++ PPL  
Parallel  
Patterns  
Library  
(VS2010)

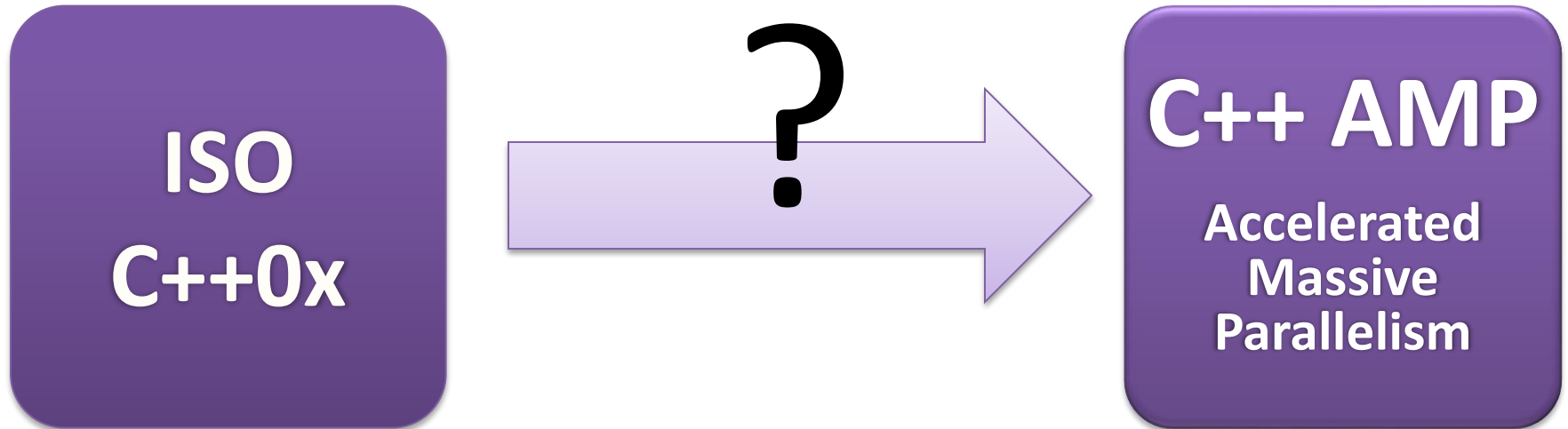
# 1

language feature for multicore

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

# Language Design: Parallelism Phase 2

*Multi-core to hetero-core*

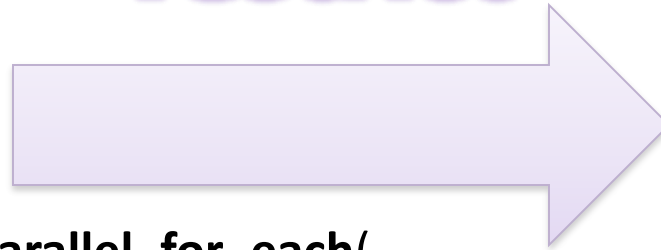


# Language Design: Parallelism Phase 2

*Multi-core to hetero-core*

ISO  
C++0x

**restrict**



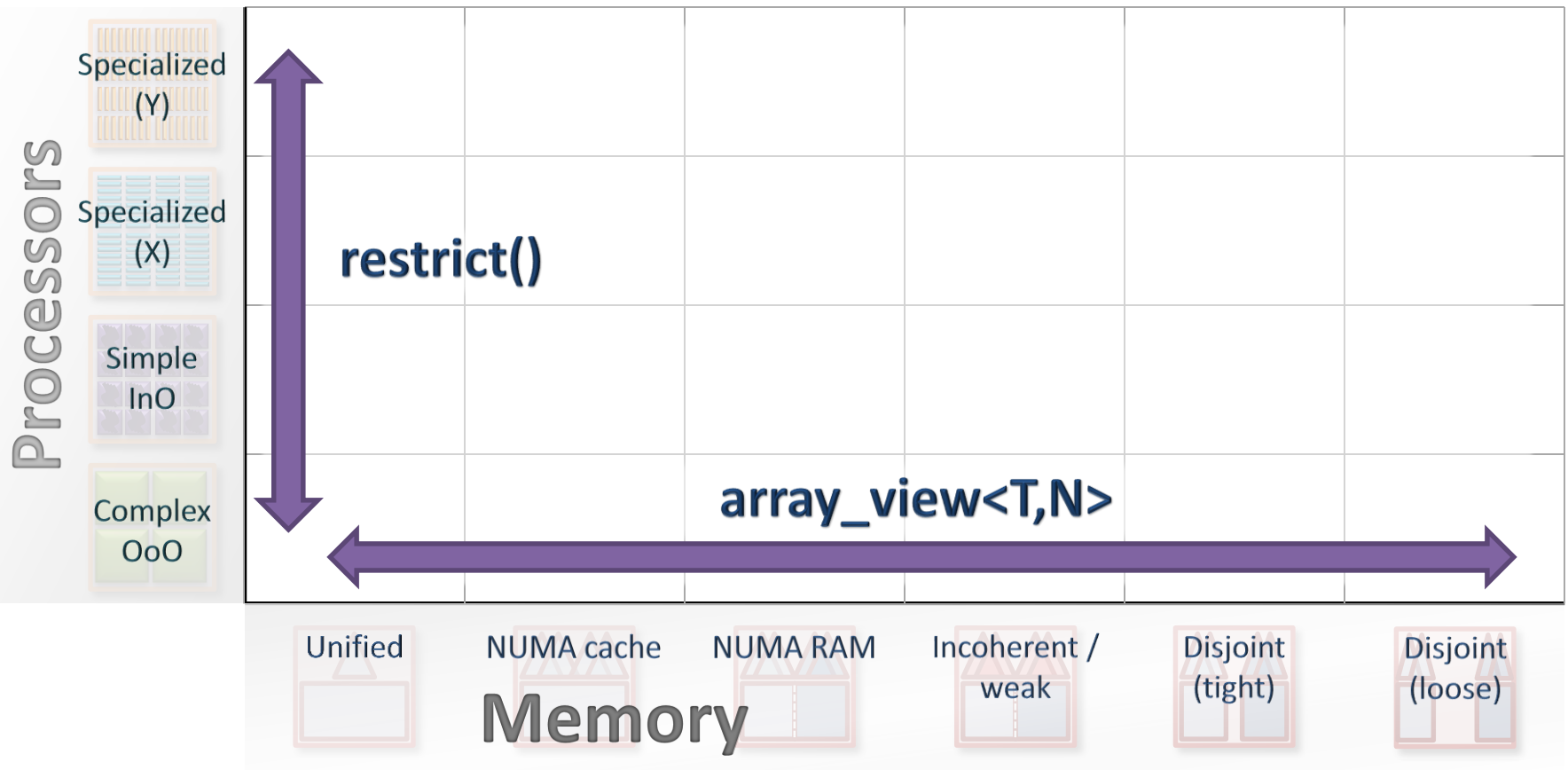
C++ AMP  
Accelerated  
Massive  
Parallelism

```
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;  
        }  
    }  
}
```

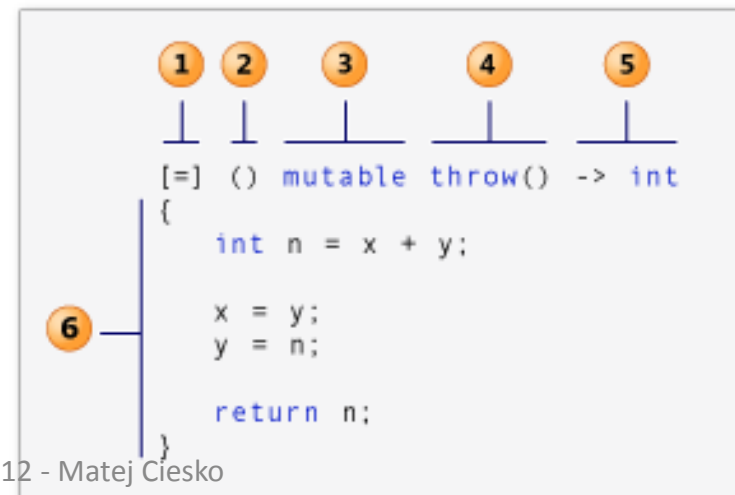
```
void MatrixMultiplyAMP( vector<float>& vC,  
    const vector<float>& vA,  
    const vector<float>& vB, int M, int N, int W )  
{  
    array_view<const float,2> a(M,W,vA),b(W,N,vB);  
    array_view<float,2> c(M,N,vC);  
    c.discard_data();  
    parallel_for_each(c.extent,  
        [=](index<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<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;  
  }  
);
```

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



# Explicit Caching

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

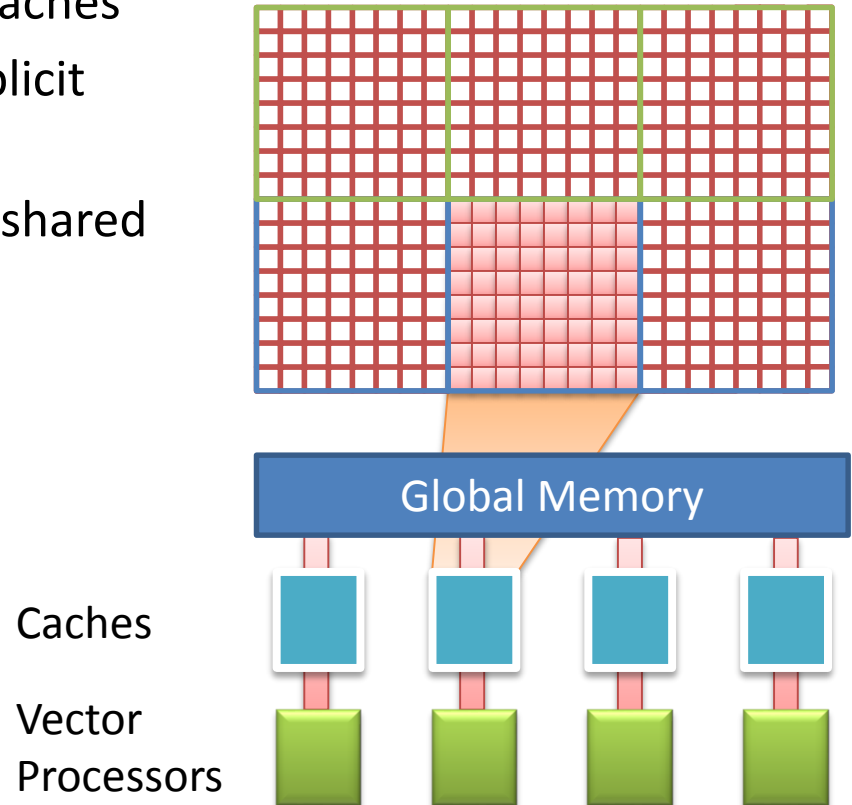
Key uses:

Capture cross-thread data reuse

Fast communication

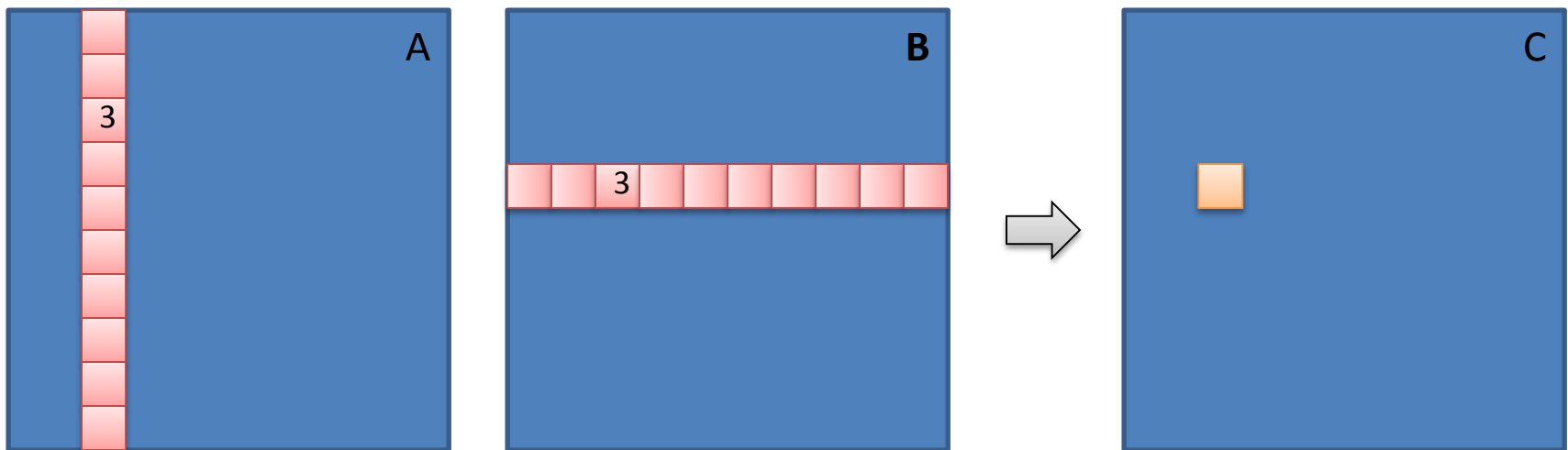
Reductions, Scans

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

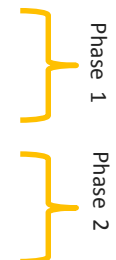
```
void MatrixMultTiled(vector<float>& vC, const vector<float>& vA,
const vector<float>& vB, int M, int N, int W )
{
    static const int TS = 16;
    array_view<const float,2> a(M, W, vA), b(W, N, vB);
    array_view<float,2> c(M,N,vC); c.discard_data();
    parallel_for_each(c.extent.tile< TS, TS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

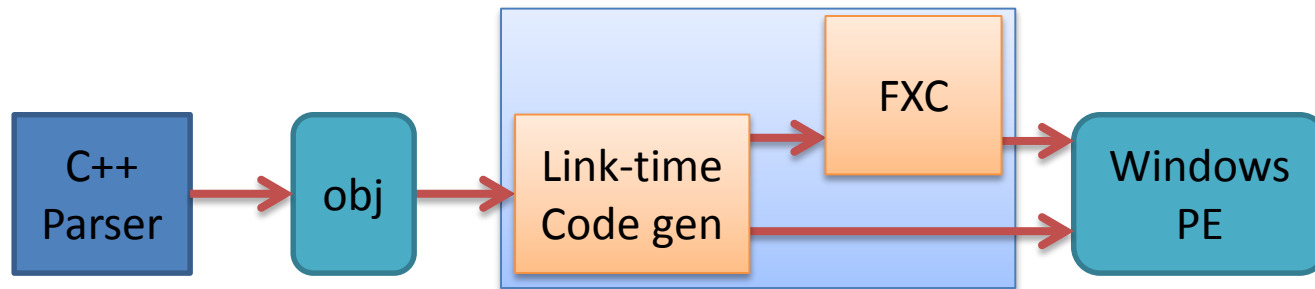
```
void MatrixMultTiled(vector<float>& vC, const vector<float>& vA,
const vector<float>& vB, int M, int N, int W )
{
    static const int TS = 16;
    array_view<const float,2> a(M, W, vA), b(W, N, vB);
    array_view<float,2> c(M,N,vC); c.discard_data();
    parallel_for_each(c.extent.tile< TS, TS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

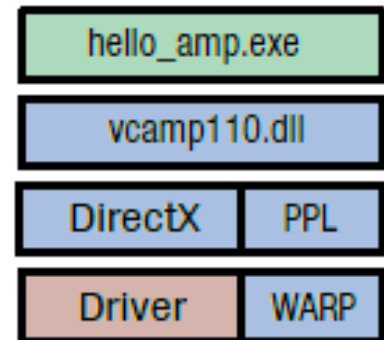
# C++ compilation & deployment



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

The screenshot displays the Visual Studio IDE during a GPU debugging session. The main window shows the source code for a matrix multiplication kernel. The code includes a lambda function that is being debugged. The Parallel Stacks window shows the call stack, highlighting the lambda function and the kernel stub. The GPU Threads window shows the execution of 4 threads, with 256 GPU threads being spawned. The Parallel Watch window shows the values of the variables idx, idxA, and idxB for each thread.

```
49 random_init_vector(vB);
50
51 //Perform the matrix multiplication on the GPU
52 extent<2> eA(M, N), eB(N, W), eC(M, W);
53 array<int, 2> mA(eA, vA.begin());
54 array<int, 2> mB(eB, vB.begin());
55 array<int, 2> mC(eC);
56
57 parallel_for_each(grid<2>(eC), [=, &mA, &mB, &mC] (index<2> idx) restrict(direct)
58 int result = 0;
59 for(int i = 0; i < mA.grid.extent[1]; i++)
60 {
61     index<2> idxA(idx[0], i);
62     index<2> idxB(i, idx[1]);
63     result += mA[idxA] * mB[idxB];
64 }
65 mC[idx] = result;
66 });
67 vC = mC;
68
```

Parallel Stacks:

- 4 GPU Threads
- <lambda\_31BC6D5D9B462AAF>::operator()
- 256 GPU Threads
- \_kernel\_stub

GPU Threads:

Thread	Thread Count	Line	Address	Location	Status
0, 0	0..1, 0..1				
	252 threads		0x000001BC	_kernel_stub0	Active
	4 threads	Line 63	0x00008E70	wmain::_J5::<lambda_31BC6D5D9B462AAF>::	Active

Parallel Watch 1 - wmain::\_J5::<lambda\_31BC6D5D9B462AAF>::operator()

[Thread]	idx	idxA	idxB	<Add Watch>
[0, 0]	{_M_base={0, 0}}	{_M_base={0, 2}}	{_M_base={2, 0}}	
[0, 1]	{_M_base={0, 1}}	{_M_base={0, 2}}	{_M_base={2, 1}}	
[1, 0]	{_M_base={1, 0}}	{_M_base={1, 2}}	{_M_base={2, 0}}	
[1, 1]	{_M_base={1, 1}}	{_M_base={1, 2}}	{_M_base={2, 1}}	

Bring CPU  
debugging  
experience  
to the GPU

# GPU Debugging

```
57 parallel_for_each(grid<2>(eC), [=, &mA, &mB, &mC] (index<2> idx) restrict(dire
58     int result = 0;
59     for(int i = 0; i < mA.grid.extent[1]; i++)
60     {
61         index<2> idxA(idx[0], i);
62         index<2> idxB(i, idx[1]);
63         result += mA[idxA] * mB[idxB];
64     }
65     mC[idx] = result;
66 });
67
```

Thread Count	Line	Address	Location	Status
252 threads	63	0x000001BC	__kernel_stub0	Active
4 threads	65	0x00008E70	wmain::_J5::<lambda_31BC6D5D9B462AAF>::	Active

Parallel Stacks

Threads

- 4 GPU Threads
  - <lambda\_31BC6D5D9B462AAF>::operator()
- 256 GPU Threads
  - \_\_kernel\_stub

**Bring CPU debugging experience to the GPU**

# GPU Debugging

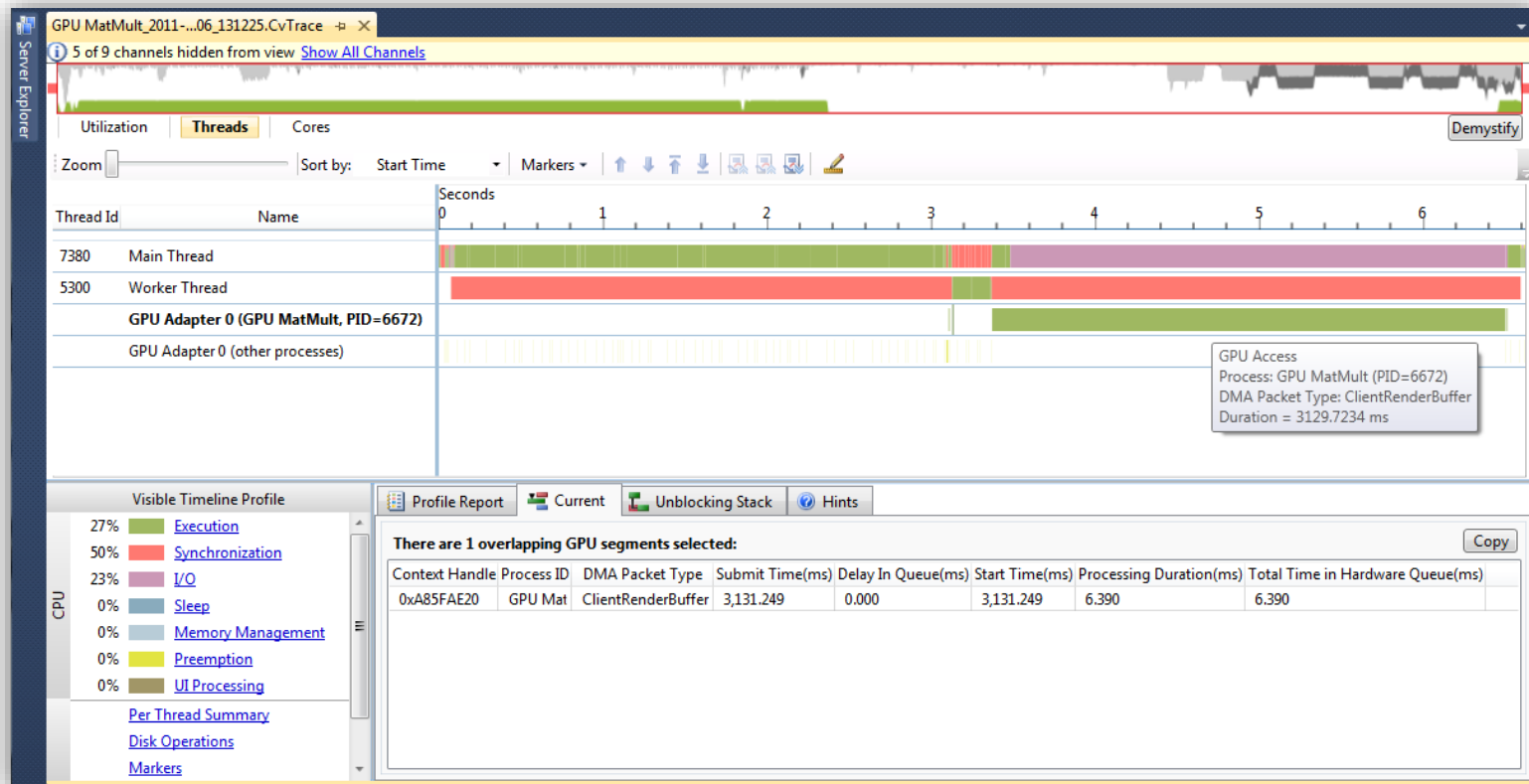
GPU Threads

Kernel: parallel\_for\_each<16,16, ...

Tile: 0, 0 0..7, 0..15 | Thread: 0, 0 0..15, 0..15 Switch Thread

	Thread Count	Line	Address	Location	Status	Tile
<b>Thread Group: [0, 0] (256 Threads)</b>						
▼	252 threads		0x000127D8	_52C1BEEA_388D_48FC_86DE_7D17D68D02A2_	Active	[0, 0]
▼	2 threads	Line 22	0x00012A88	ObtainAbsoluteIndex	Diverged	[0, 0]
▼	2 threads	Line 13	0x00012BAC	CalculateAbsoluteIndex	Active	[0, 0]
<b>Thread Group: [0, 1] (256 Threads)</b>						
▼	252 threads	1 warp; 1 diverged warp: warp 0		_52C1BEEA_388D_48FC_86DE_7D17D68D02A2_	Active	[0, 1]
▼	2 threads				Diverged	[0, 1]
▼	2 threads				Active	[0, 1]
<b>Thread Group: [0, 10] (256 Threads)</b>						
▼	252 threads		0x000127D8	_52C1BEEA_388D_48FC_86DE_7D17D68D02A2_	Active	[0, 10]
▼	2 threads	Line 22	0x00012A88	ObtainAbsoluteIndex	Diverged	[0, 10]
▼	2 threads	Line 13	0x00012BAC	CalculateAbsoluteIndex	Active	[0, 10]
<b>Thread Group: [0, 11] (256 Threads)</b>						
▼	252 threads		0x000127D8	_52C1BEEA_388D_48FC_86DE_7D17D68D02A2_	Active	[0, 11]
▼	2 threads	Line 22	0x00012A88	ObtainAbsoluteIndex	Diverged	[0, 11]
▼	2 threads	Line 13	0x00012BAC	CalculateAbsoluteIndex	Active	[0, 11]

# GPU Profiling



# PORTING CS 267



# Assignment 1

## Serial Performance

- **Assignment target:** Single thread performance

- **Learning goal:**

- Identify performance bottlenecks
- Use libraries
- Optimize kernel

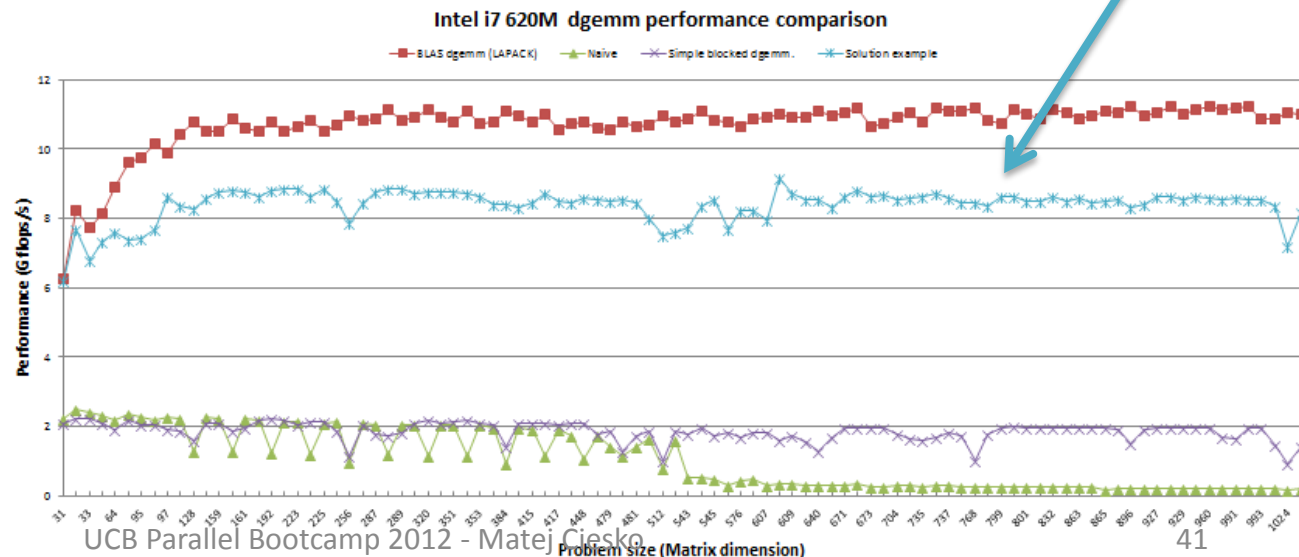
- **Problem:**  $C \leftarrow A \times B + C \quad A, B, C \in \mathbb{R}^{n \times n}$

- **Solution:**

Optimizations

- Blocking
- Loop unrolling
- Prefetching
- Vectorization
- Repacking data
- [...]

**Performance:**



# Assignment 1 (Cont.)

The screenshot shows the Visual Studio IDE with a 'Function Code View' window on the left and a 'Performance Wizard' dialog box in the center. The 'Function Code View' displays a list of functions with their respective CPU usage percentages. The 'Performance Wizard' dialog box is titled 'Specify the profiling method' and offers four options: CPU sampling (recommended), Instrumentation, .NET memory allocation (sampling), and Resource contention data (concurrency). A blue arrow points from the text 'Using Visual Studio student can identify performance bottlenecks.' to the 'Performance Wizard' dialog box.

**Function Code View**

Line	Function	CPU %
1	#	
2	#	
3	C	
4	v	
5	{	
6		
7	< 0.1 %	
8		
9	2.0 %	
10	8.1 %	
11	87.6 %	
12	< 0.1 %	
13	< 0.1 %	
14	}	
15	#	
16		
17		

**Performance Wizard -- Page 1 of 3**

**Specify the profiling method**

Profiling your application can help diagnose performance problems and identify the most common expensive methods in your application. To begin, choose a profiling method from the options below.

**What method of profiling would you like to use?**

- CPU sampling (recommended)**  
Monitor CPU-bound applications with low overhead
- Instrumentation**  
Measure function call counts and timing
- .NET memory allocation (sampling)**  
Track managed memory allocation
- Resource contention data (concurrency)**  
Detect threads waiting for other threads

[Read more about profiling methods](#)

< Previous    Next >    **Finish**    Cancel

**Process List**

Process	Private Bytes	Working Set	Private Bytes	Private Bytes	Private Bytes
msvcr110.dll	0.00				
Matmul.exe	1.52				
Matmul.exe	0.04				
Matmul.exe	0.03				
Matmul.exe	0.00				

# 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:

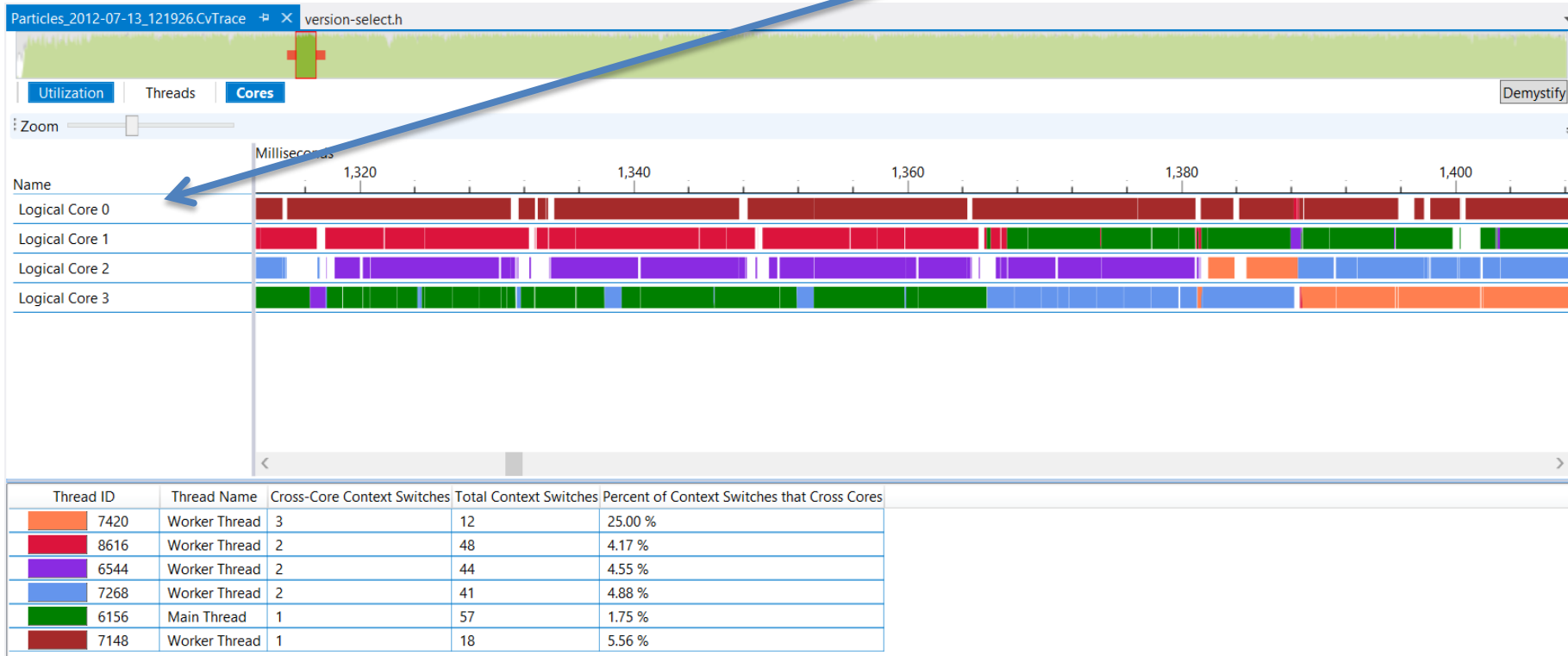
- Native threads
  - OpenMP, PPL
  - MPI
  - C++ AMP
- Performance comparison: (initial/ trivial solution, n = 1000)

```
#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();
}
```

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

# Assignment 2 (Cont.)

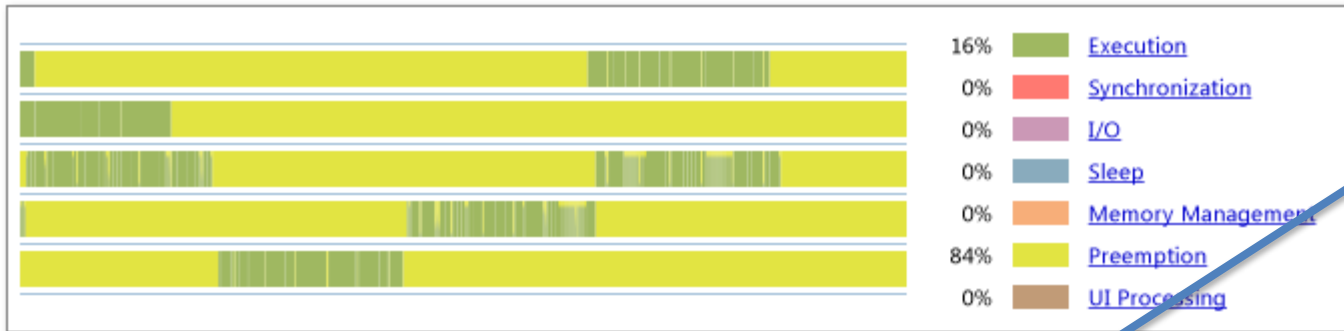
Using Visual Studio students can deep dive into synchronization patterns and timing.



# Assignment 2 (Cont.)

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

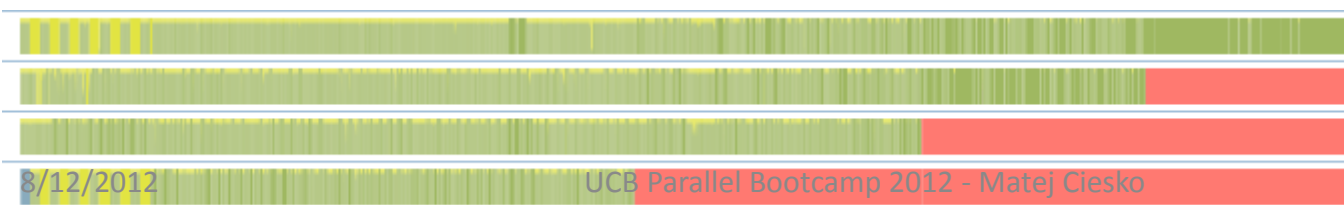
- Oversubscription



- 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!