M02: High Performance Computing with CUDA

Optimizing CUDA

Paulius Micikevicius

Outline

- Memory Coalescing
- Staging Coefficients
- Streams and Asynchronous API
- Sample: 3D Finite Difference

MEMORY COALESCING

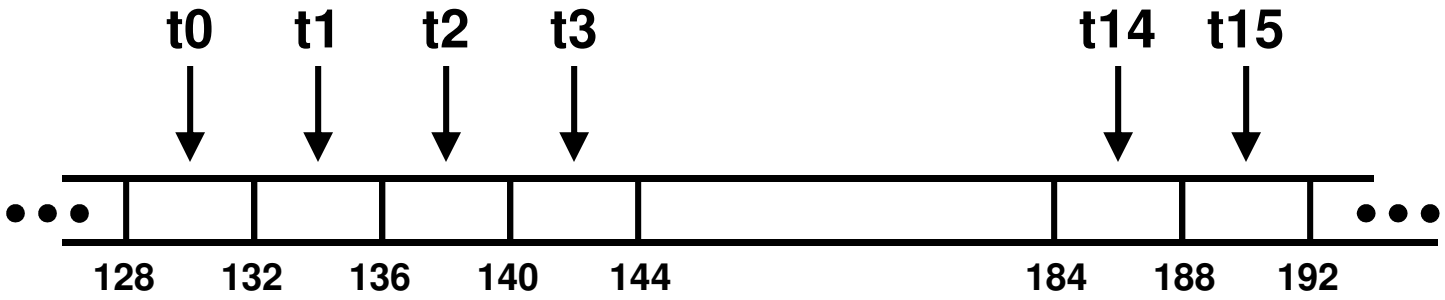
Memory Performance

- **To maximize global memory bandwidth:**
  - Minimize the number of bus transactions
    - Coalesce memory accesses
- **Coalescing**
  - Memory transactions are per half-warp (16 threads)
  - In best cases, one transaction will be issued for a half-warp
  - Latest hardware relaxes coalescing requirements
    - Compute capability 1.2 and later

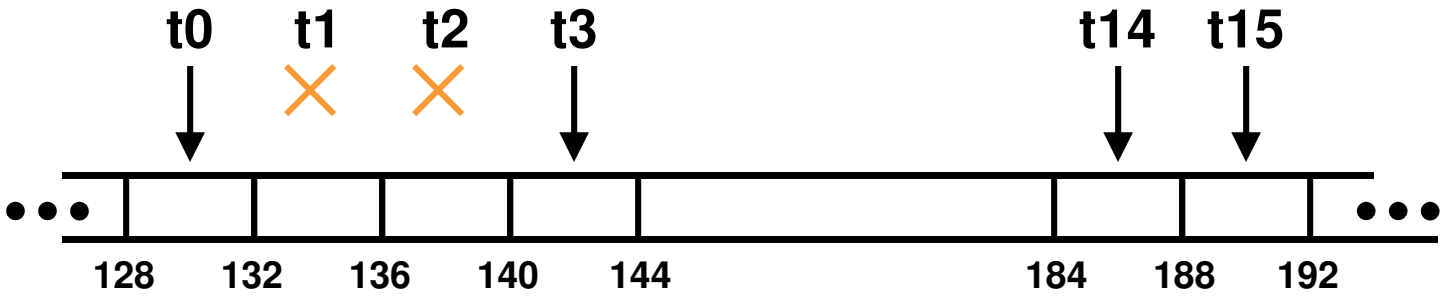
Coalescing: Compute Capability < 1.2

- **A coordinated read by a half-warp (16 threads)**
- **A contiguous region of global memory:**
  - 64 bytes - each thread reads a word: int, float, …
  - 128 bytes - each thread reads a double-word: int2, float2
  - 256 bytes – each thread reads a quad-word: int4, float4, …
- **Additional restrictions:**
  - Starting address must be a multiple of region size
  - The k^{th} thread in a half-warp must access the k^{th} element in a block being read
- **Exception: not all threads must be participating**
  - Predicated access, divergence within a halfwarp

Coalesced Access: Reading floats

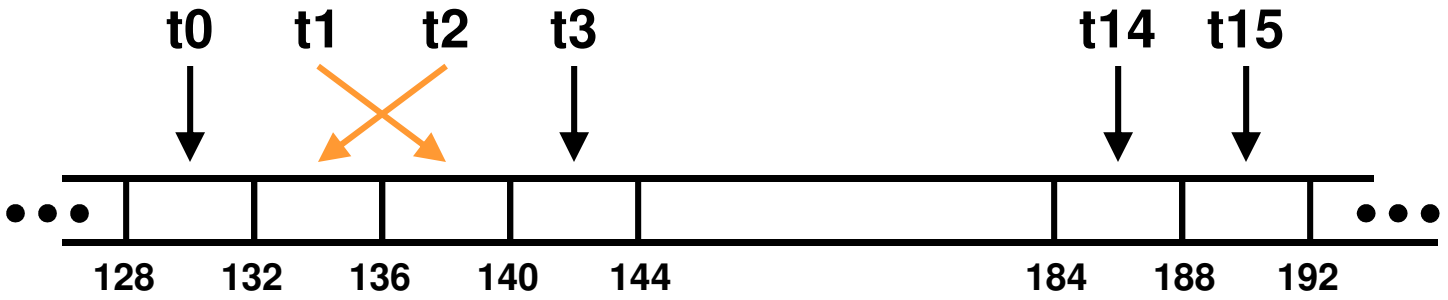


All threads participate

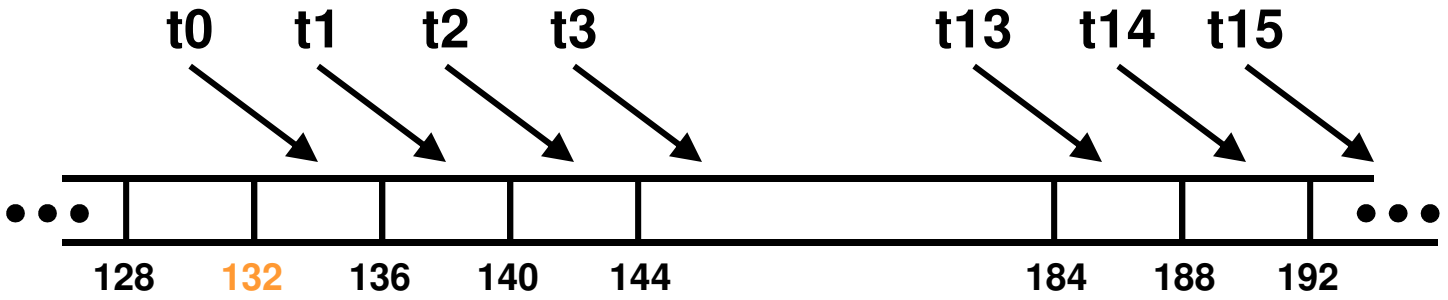


Some Threads Do Not Participate

Uncoalesced Access: Reading floats



Permuted Access by Threads



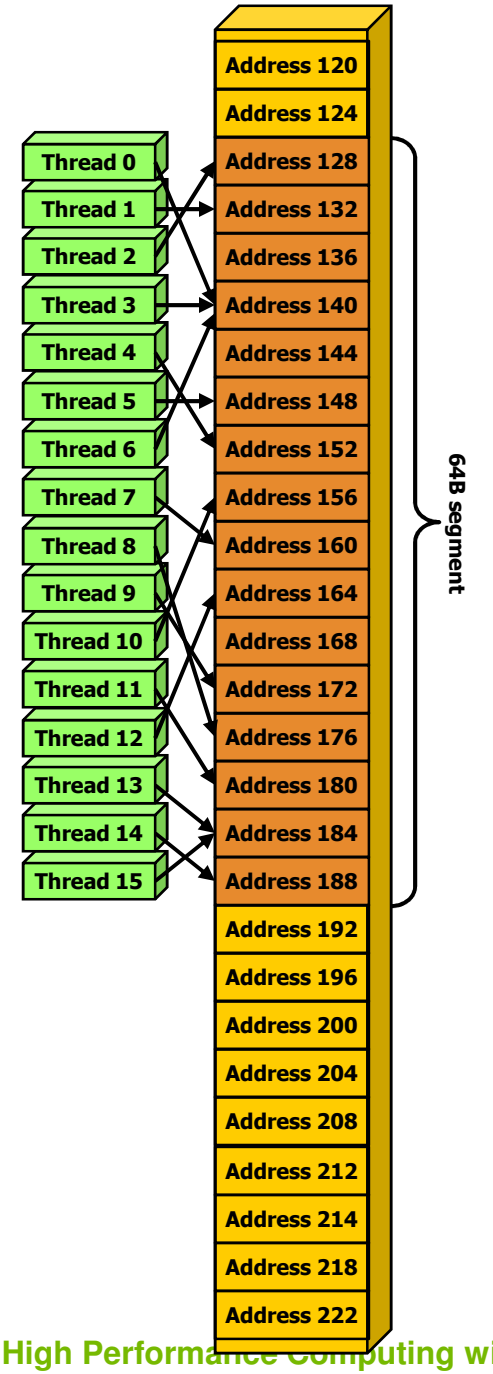
Misaligned Starting Address (not a multiple of 64)

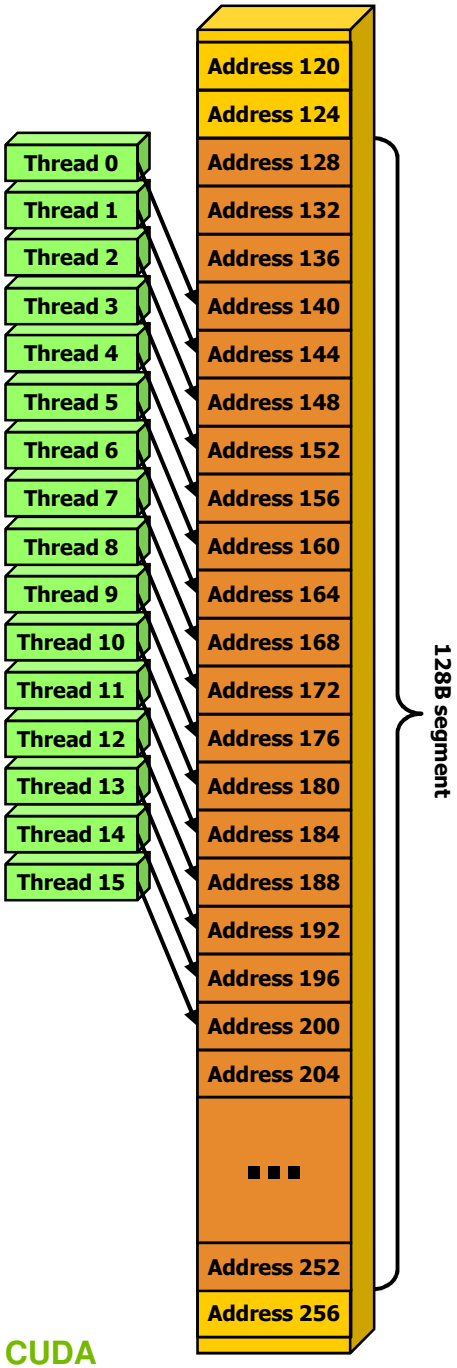
Coalescing: Timing Results

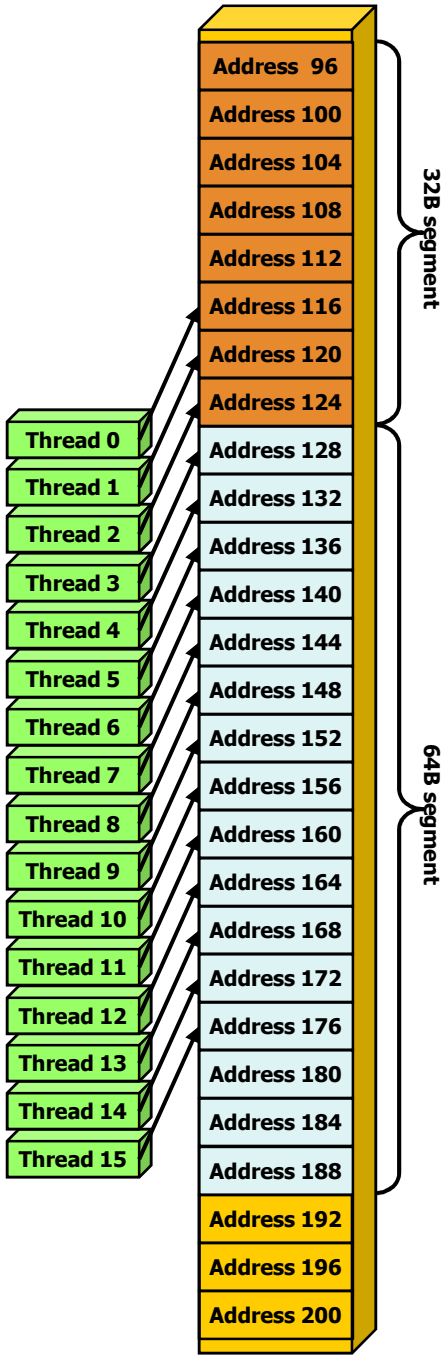
- **Experiment:**
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- **12K blocks x 256 threads:**
  - 356µs – coalesced
  - 357µs – coalesced, some threads don't participate
  - 3,494µs – permuted/misaligned thread access

Coalescing: Compute Capability ≥ 1.2

- **Possible bus transaction sizes:**
  - 32B, 64B, or 128B
  - Memory segment must be aligned
    - First address = multiple of segment size
- **Hardware coalescing for each half-warp:**
  - Carry out the smallest possible number of transactions
  - Reduce transaction size when possible


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 208
Address 212
Address 214
Address 218
Address 222
64B segment


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 252
Address 256
128B segment


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Address 96
Address 100
Address 104
Address 108
Address 112
Address 116
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
32B segment
64B segment

Coalescing Algorithm

- **Find the memory segment that contains the address requested by the lowest numbered active thread:**
  - 32B segment for 8-bit data
  - 64B segment for 16-bit data
  - 128B segment for 32, 64 and 128-bit data.
- **Find all other active threads whose requested address lies in the same segment**
- **Reduce the transaction size, if possible:**
  - If size == 128B and only the lower or upper half is used, reduce transaction to 64B
  - If size == 64B and only the lower or upper half is used, reduce transaction to 32B
- **Carry out the transaction, mark threads as inactive**
- **Repeat until all threads in the half-warp are serviced**

Comparing Compute Capabilities

- **Compute capability < 1.2**
  - Requires threads in a half-warp to:
    - Access a single aligned 64B, 128B, or 256B segment
    - Threads must issue addresses in sequence
  - If requirements are not satisfied:
    - Separate 32B transaction for each thread
- **Compute capability ≥ 1.2**
  - Does not require sequential addressing by threads
  - Perf degrades gracefully when a half-warp addresses multiple segments

STAGING COEFFICIENTS

General Use Case

- **Kernel contains a loop:**
  - For a given iteration, all threads read the same value
  - Different values for different iterations
- **Implementation choices:**
  - Each thread reads in every iteration:
    - From global memory
    - From constant memory (cached)
    - From 1D texture (cached)
  - Threads *stage* reads through shared memory:
    - Threads collectively place coefficients into smem
    - Each thread reads from smem in every iteration

Experiment

- **Threads iterate through a 3D volume along z**
  - For a given z-iteration, threads write the same coefficient
  - Different coefficients for every z

```
__global__ void gmem_bcast( float *g_data, float *g_coeff, int dimx, int dimy, int dimz )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    int stride = dimx*dimy;

    for(int iz=0; iz<dimz; iz++)
    {
        g_data[idx] = g_coeff[iz];
        idx += stride;
    }
}
```


Kernel with Staged Coefficients:

- **Number of coefficients ≤ threads per block**

```
__global__ void gmem_staged( float *g_data, float *g_coeff, int dimx, int dimy, int dimz )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    int stride = dimx*dimy;

    __shared__ float s_coeff[NUM_COEFF];
    int thread_id = threadIdx.y*blockDim.x + threadIdx.x;
    if( thread_id < NUM_COEFF)
        s_coeff[thread_id] = g_coeff[thread_id];
    __syncthreads();

    for(int iz=0; iz<dimz; iz++)
    {
        g_data[idx] = s_coeff[iz];
        idx += stride;
    }
}
```


Kernel with Staged Coefficients

- **Any number of coefficients that fits into shared mem**

```
__global__ void gmem_staged( float *g_data, float *g_coeff, int dimx, int dimy, int dimz )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    int stride = dimx*dimy;

    __shared__ float s_coeff[NUM_COEFF];
    int thread_id = threadIdx.y*blockDim.x + threadIdx.x;
    int num_threads = blockDim.x*blockDim.y;
    for(int i=thread_id; i<NUM_COEFF; i+=num_threads)
        s_coeff[i] = g_coeff[i];
    __syncthreads();

    for(int iz=0; iz<dimz; iz++)
    {
        g_data[idx] = s_coeff[iz];
        idx += stride;
    }
}
```


Experimental Results

- 800x800x400 data

Method	Time (ms)
gmem bcast	19
cmem bcast	9
texture bcast	9
gmem staged	9

Bcast = each thread reads the same value from specified memory

Conclusion: Just Use Constant Mem


```
__constant__  float c_coeff[ NUM_COEFFICIENS];
...

__global__ void gmem_bcast( float *g_data, float *g_coeff, int dimx, int dimy, int dimz )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    int stride = dimx*dimy;

    for(int iz=0; iz<dimz; iz++)
    {
        g_data[idx] = c_coeff[iz];
        idx += stride;
    }
}

...

cudaMemcpyToSymbol( c_coeff,  ... );
```


STREAMS AND ASYNC API

Streams and Async API

- **Default API:**
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls block on GPU
    - Serialized by the driver
- **Streams and async functions provide:**
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
- **Stream = sequence of operations that execute in order on GPU**
  - Operations from different streams can be interleaved
  - A kernel and memcopy from different streams can be overlapped

Overlap kernel and memory copy

- **Requirements:**
  - D2H or H2D memcopy from *pinned* memory
  - Device with compute capability ≥ 1.1 (G84 and later)
  - Kernel and memcopy in different, non-0 streams
- **Code:**

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

kernel<<<grid, block, 0, stream1>>>(…);
cudaMemcpyAsync( dst, src, size, dir, stream2 );
```

potentially overlapped

CUDA Events

- **Events are inserted into streams of CUDA calls**
  - cudaEventRecord(event, stream)
- **Event is *recorded* when the GPU reaches it in a stream**
  - Record = assign a timestamp (GPU clocktick)
- **Useful for timing, querying/syncing with GPU**
- **Stream/event queries:**
  - Do not block the CPU thread
  - cudaStreamQuery(stream)
    - Indicates whether the stream is idle
  - cudaEventQuery(event)
    - Indicates whether the event has been recorded

CPU/GPU Synchronization

- **Three ways to synchronize CPU-thread and GPU:**
  - cudaThreadSynchronize()
    - Blocks until all previously issued CUDA calls complete
  - cudaStreamSynchronize(stream)
    - Blocks until all CUDA calls issued to the given stream complete
  - cudaEventSynchronize(event)
    - Blocks until the given event is recorded on GPU
- **Any CUDA call to stream-0 blocks until previous calls complete**
  - No CUDA calls can be overlapped with a stream-0 call

Timing with CUDA Events

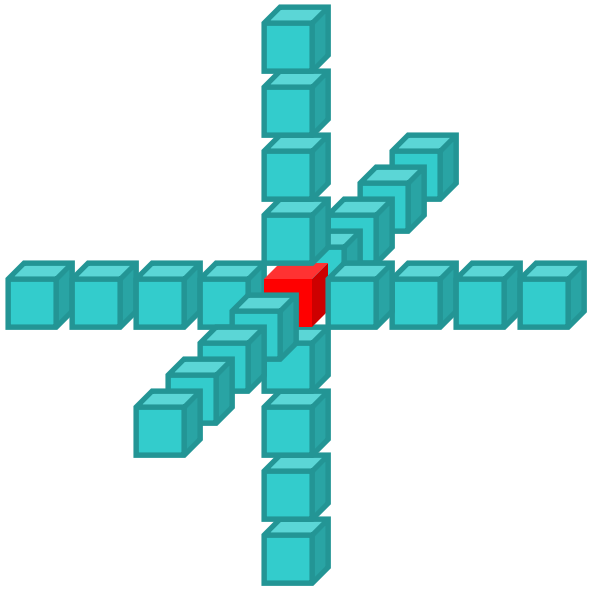
- **Timer resolution ~ GPU clock period**
- **Remember that timing is done on GPU**

```
float elapsed_time_ms = 0.0f;
cudaEvent_t  start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );

cudaEventRecord( start, 0 );
some_kernel<<<....>>>(...);
cudaEventRecord( stop, 0 );

cudaEventSynchronize( stop );
cudaEventElapsedTime( &elapsed_time_ms, start, stop );
```


3D FINITE DIFFERENCE

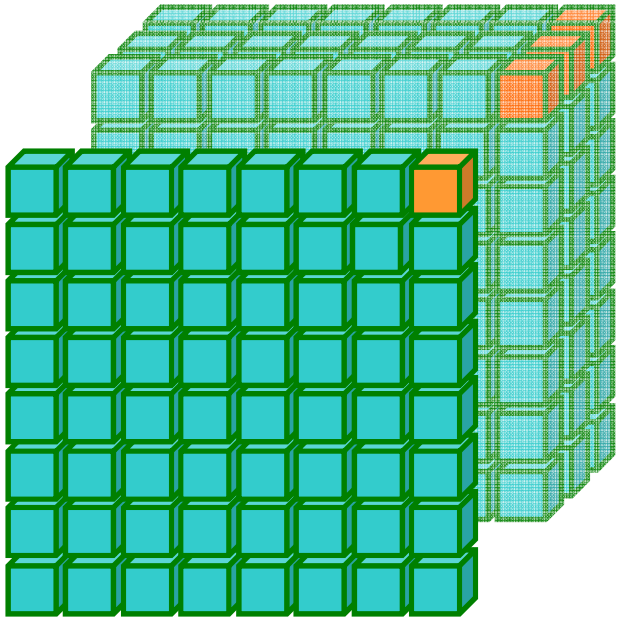
3D Finite Difference

- 25-point stencil (8th order in space)
- Isotropic: 5 distinct coefficients
- For each output element's stencil we need:
  - 29 flops
  - 25 input values
- FD of the wave equation
  - More details on application in Scott Morton's *Seismic Imaging* talk at 1:30



General Approach

- **Tile a 2D slice with 2D threadblocks**
  - Slice in the two fastest dimensions: x and y
- **Each thread iterates along the slowest dimension (z)**
  - Each thread is responsible for one element in every slice
  - Only one kernel launch
  - Also helps data reuse



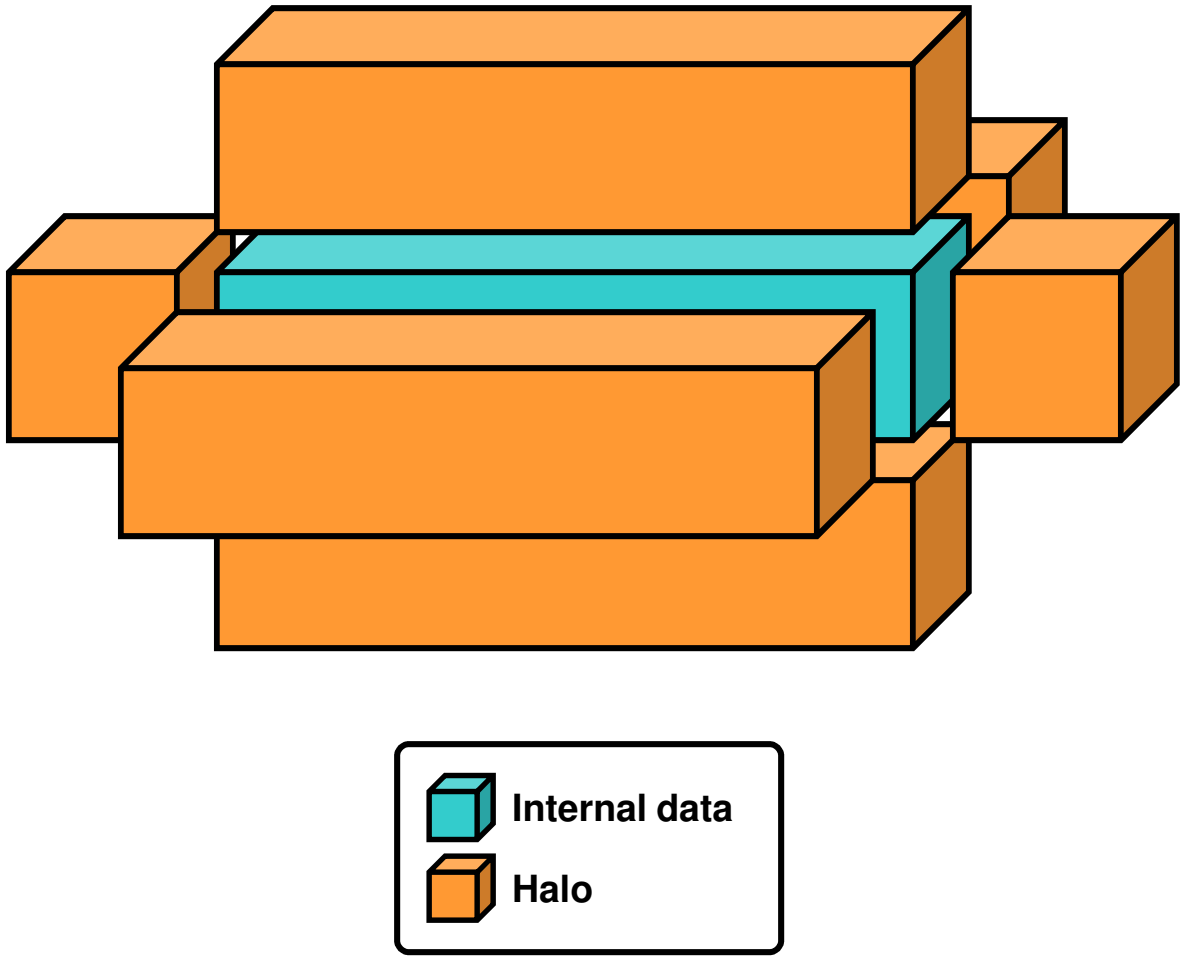
Naive Implementation

- **One thread per output element**
- **Fetch all data for every output element**
  - Redundant: input is read 25 times
  - Required bandwidth = 25 reads, 1 write (26x)
- **Optimization: share data among threads**
  - Use shared memory for data needed by many threads
  - Use registers for data needed not shared among threads

Using Shared Memory: First Take

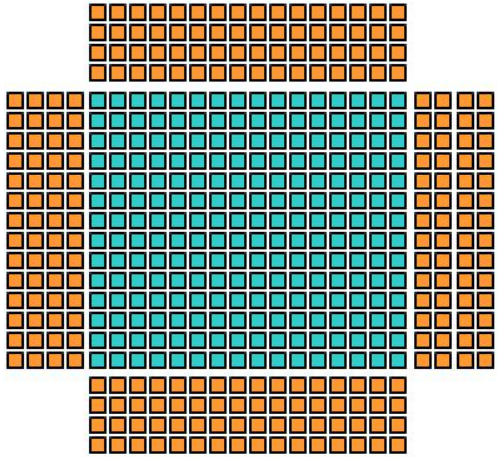
- **Read a 3D subdomain from gmem into smem**
  - Compute from smem
- **Limited by amount of smem (16KB)**
  - Need 4-element halos in each direction:
    - (dimx+8)×(dimy+8)×(dimz+8) storage for dimx×dimy×dimz subdomain
    - dimx should be multiple of 16 for max bandwidth (coalescing)
  - What would fit (4-byte elements):
    - 24x14x12 storage (16x6x4 subdomain)
    - Only 9.5% of storage is not halo (could be improved to 20%)
- **Requires bandwidth for 5.8x data size**
  - 4.83x read, 1 write
  - Better than 26x but still redundant

3D Subdomain in Shared Memory



Using Shared Memory: Second Take

- **SMEM is sufficient for 2D subdomains**
  - Square tiles require the smallest halos
  - Up to 64x64 storage (56x56 subdomain)
    - 76.5% of storage is not halo
- **3D FD done with 2 passes:**
  - 2D-pass
  - 1D-pass
- **Volume accesses:**
  - Read/write for both passes
    - 2D-pass reads original, halo, and 1D-pass output
  - 16x16 subdomain tiles: 6.00 times
  - 32x32 subdomain tiles: 5.50 times
  - 56x56 subdomain tiles: 5.29 times



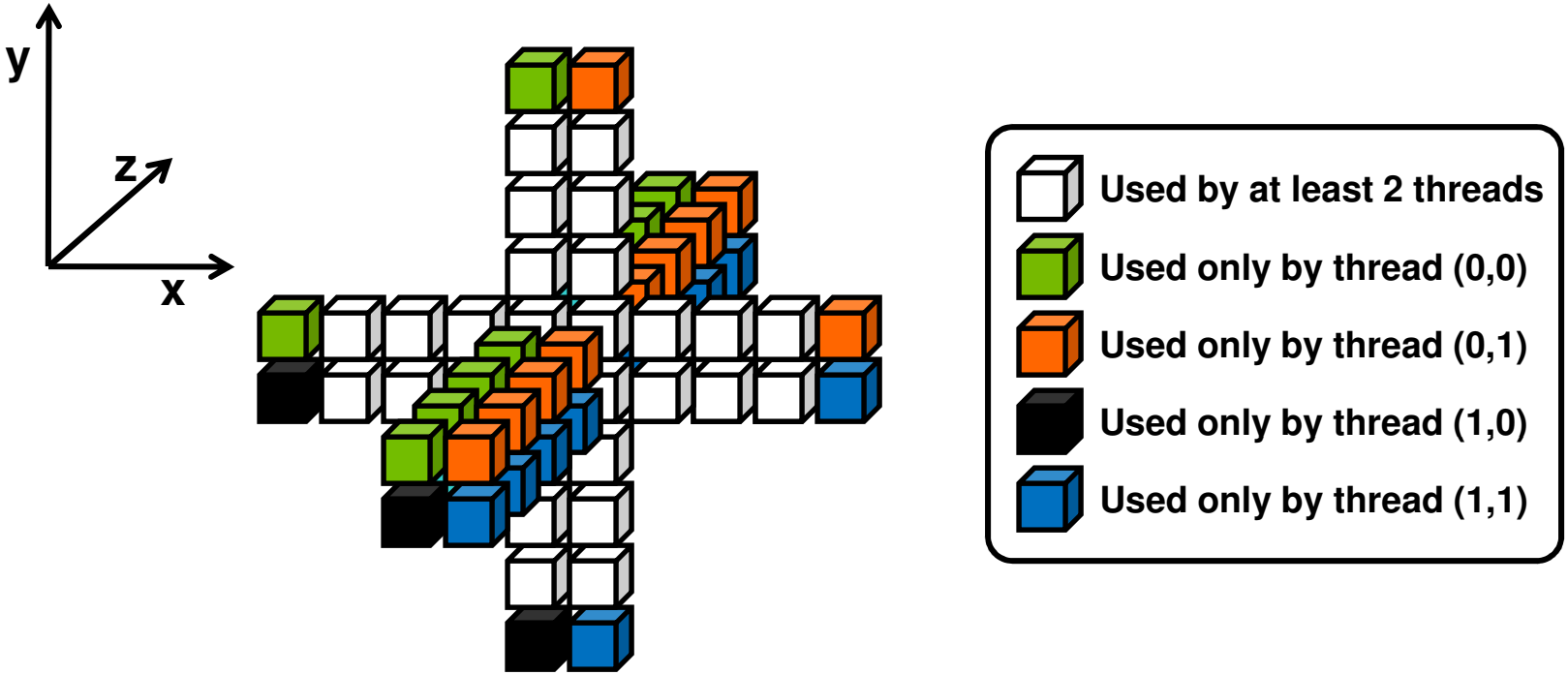
Two-Pass Stencil-Only Performance

- **Hardware: Tesla C1060 (4GB, 240 SPs)**
- **2D-pass (32x32 tile):**
  - 544x512x200: 5,811 Mpoints/s
  - 800x800x800: 5,981 Mpoints/s
- **1D-pass (3 gmem accesses / point):**
  - 544x512x200: 6,547 Mpoints/s
  - 800x800x800: 6,307 Mpoints/s
- **Combined:**
  - 544x512x200: 3,075 Mpoints/s
  - 800x800x800: 3,071 Mpoints/s

Using Shared Memory: Third Take

- **Combine the 2D and 1D passes**
  - 1D pass needs no SMEM: keep data in registers

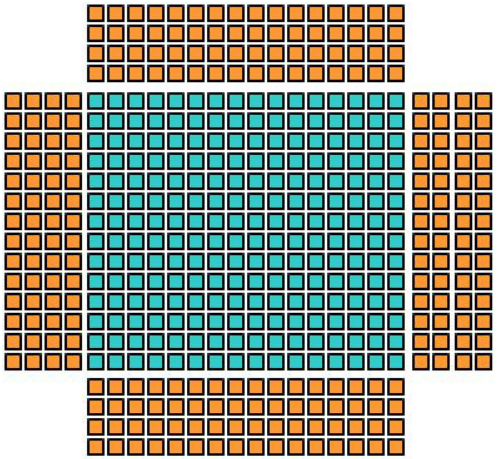
Input Re-use by 2x2 Threads



- Store the xy-slice in SMEM
- Each thread keeps its 8 z-elements in registers
  - 4 “infront”, 4 “behind”

Using Shared Memory: Third Take

- **Combine the 2D and 1D passes**
  - 1D pass needs no SMEM: keep data in registers
- **16x16 2D subdomains**
  - 16x16 threadblocks
  - 24x24 SMEM storage (2.25KB) per threadblock
    - 44% of storage is not halo
    - Volume is accessed 3 times (2 reads, 1 write)
      - 2 reads due to halo



Using Shared Memory: Third Take

- **Combine the 2D and 1D passes**
  - 1D pass needs no SMEM: keep data in registers
- **16x16 2D subdomains**
  - 16x16 threadblocks
  - 24x24 SMEM storage (2.25KB) per threadblock
    - 44% of storage is not halo
    - Volume is accessed 3 times (2 reads, 1 write)
- **32x32 2D subdomains**
  - 32x16 threadblocks
  - 40x40 SMEM storage (6.25KB) per threadblock
    - 64% of storage is not halo
    - Volume is accessed 2.5 times (1.5 reads, 1 write)

Inner Loop of 16x16-tile stencil kernel


```
// ------- advance the slice (move the thread-front) --------------------
behind4  = behind3;
behind3  = behind2;
behind2  = behind1;
behind1  = current;
current   = infront1;
infront1  = infront2;
infront2  = infront3;
infront3  = infront4;
infront4  = g_input[in_idx];

in_idx    += stride;
out_idx  += stride;
__syncthreads( );

// ------- update the data slice in smem  ---------------------------------
if( threadIdx.y<radius )   // top and bottom halo
{
   s_data[threadIdx.y][tx]                  = g_input[out_idx – radius * dimx];
   s_data[threadIdx.y+16+radius][tx]  = g_input[out_idx + 16 * dimx];
}
if( threadIdx.x<radius )   // left and right halo
{
   s_data[ty][threadIdx.x]                  = g_input[out_idx – radius];
   s_data[ty][threadIdx.x+16+radius] = g_input[out_idx + 16];
}
s_data[ty][tx] = current;   // 16x16 “internal” data
__syncthreads( );

// compute the output value  ------------------------------------------------

float div  = c_coeff[0] * current;
div += c_coeff[1] * ( infront1 + behind1 + s_data[ty-1][tx]+ s_data[ty+1][tx]+ s_data[ty][tx-1]+ s_data[ty][tx+1] );
div += c_coeff[2] * ( infront2 + behind2 + s_data[ty-2][tx]+ s_data[ty+2][tx]+ s_data[ty][tx-2]+ s_data[ty][tx+2] );
div += c_coeff[3] * ( infront3 + behind3 + s_data[ty-3][tx]+ s_data[ty+3][tx]+ s_data[ty][tx-3]+ s_data[ty][tx+3] );
div += c_coeff[4] * ( infront4 + behind4 + s_data[ty-4][tx]+ s_data[ty+4][tx]+ s_data[ty][tx-4]+ s_data[ty][tx+4] );
g_output[out_idx] = div;
```


Inner Loop of 16x16-tile FD kernel


```
// ------- advance the slice (move the thread-front) --------------------
behind4  = behind3;
behind3  = behind2;
behind2  = behind1;
behind1  = current;
current   = infront1;
infront1  = infront2;
infront2  = infront3;
infront3  = infront4;
infront4  = g_input[in_idx];

in_idx    += stride;
out_idx  += stride;
__syncthreads( );

// ------- update the data slice in smem  ---------------------------------
if( threadIdx.y<radius )   // top and bottom halo
{
   s_data[threadIdx.y][tx]                  = g_input[out_idx – radius * dimx];
   s_data[threadIdx.y+16+radius][tx]  = g_input[out_idx + 16 * dimx];
}
if( threadIdx.x<radius )   // left and right halo
{
   s_data[ty][threadIdx.x]                  = g_input[out_idx – radius];
   s_data[ty][threadIdx.x+16+radius] = g_input[out_idx + 16];
}
s_data[ty][tx] = current;   // 16x16 “internal” data
__syncthreads( );

// compute the output value  ------------------------------------------------
float temp = 2.f * current - g_next[out_idx];
float div  = c_coeff[0] * current;
div += c_coeff[1] * ( infront1 + behind1 + s_data[ty-1][tx]+ s_data[ty+1][tx]+ s_data[ty][tx-1]+ s_data[ty][tx+1] );
div += c_coeff[2] * ( infront2 + behind2 + s_data[ty-2][tx]+ s_data[ty+2][tx]+ s_data[ty][tx-2]+ s_data[ty][tx+2] );
div += c_coeff[3] * ( infront3 + behind3 + s_data[ty-3][tx]+ s_data[ty+3][tx]+ s_data[ty][tx-3]+ s_data[ty][tx+3] );
div += c_coeff[4] * ( infront4 + behind4 + s_data[ty-4][tx]+ s_data[ty+4][tx]+ s_data[ty][tx-4]+ s_data[ty][tx+4] );
g_output[out_idx] = temp + div * g_vsq[out_idx];
```

2 more GMEM reads
4 more FLOPS

Per output element:

- 33 FLOPS
- 5 GMEM accesses (32bit)

Based on FD code by Scott Morton, Hess

32x32 Tiles

- **32x32 tile is divided into upper and lower halves**
  - 32x16 threadblocks
  - Each thread is responsible for 2 output elements
- **Register pressure is an issue**
  - Each output element requires 8 registers (z-values)
  - For 32x16 threadblocks (512 threads) must use 32 or fewer registers per thread
    - Use *–maxrregcount=32* compiler flag

Single-Pass Stencli-Only Performance for Various Orders in Space, MPoints/s

Data Dimensions	4	6	8	10	12
480 × 480 × 400	4,774	4,455	4,269	3,435	2,885
544 × 544 × 400	4,731	4,389	4,214	3,347	2,816
640 × 640 × 400	4,802	4,168	3,932	3,331	2,802
800 × 800 × 400	3,611	3,807	3,717	3,717	3,236

16x16 Tiles (subdomains)

Smaller orders can be optimized further by doing multiple time-steps out of smem

Measured on: Tesla C1060

Single-Pass 3D Finite Difference Performance in MPoints/s (8^{th} order in space, 2^{nd} order in time)

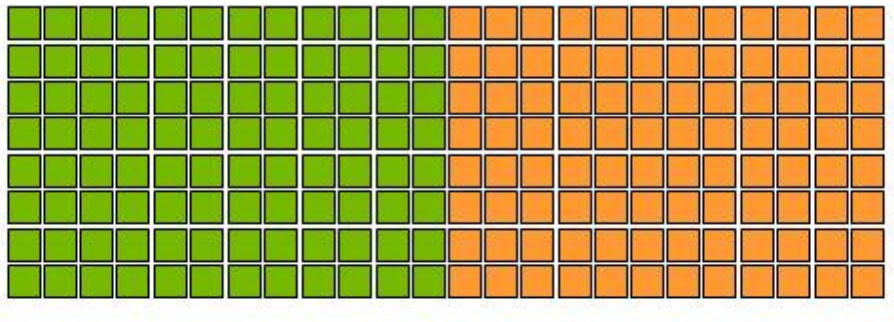
Data Dimensions	16×16 Tiles	32×32 Tiles
480 × 480 × 400	3,077.8	3,081.7
544 × 544 × 544	2,797.9	3,181.2
640 × 640 × 640	2,558.5	3,106.4
800 × 800 × 400	2,459.0	3,256.9

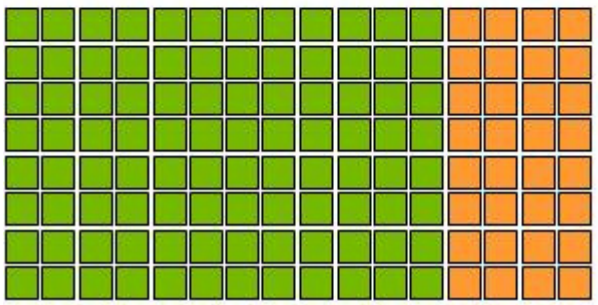
Read: 25-point stencil, velocity, and previous time step value
(forward solve of the RTM, though boundary conditions are ignored)

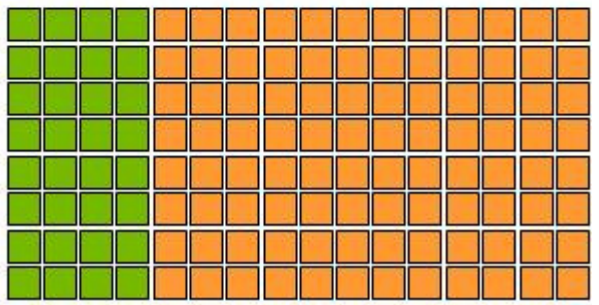
Measured on: Tesla C1060

Multi-GPU Approach (8th order in space)

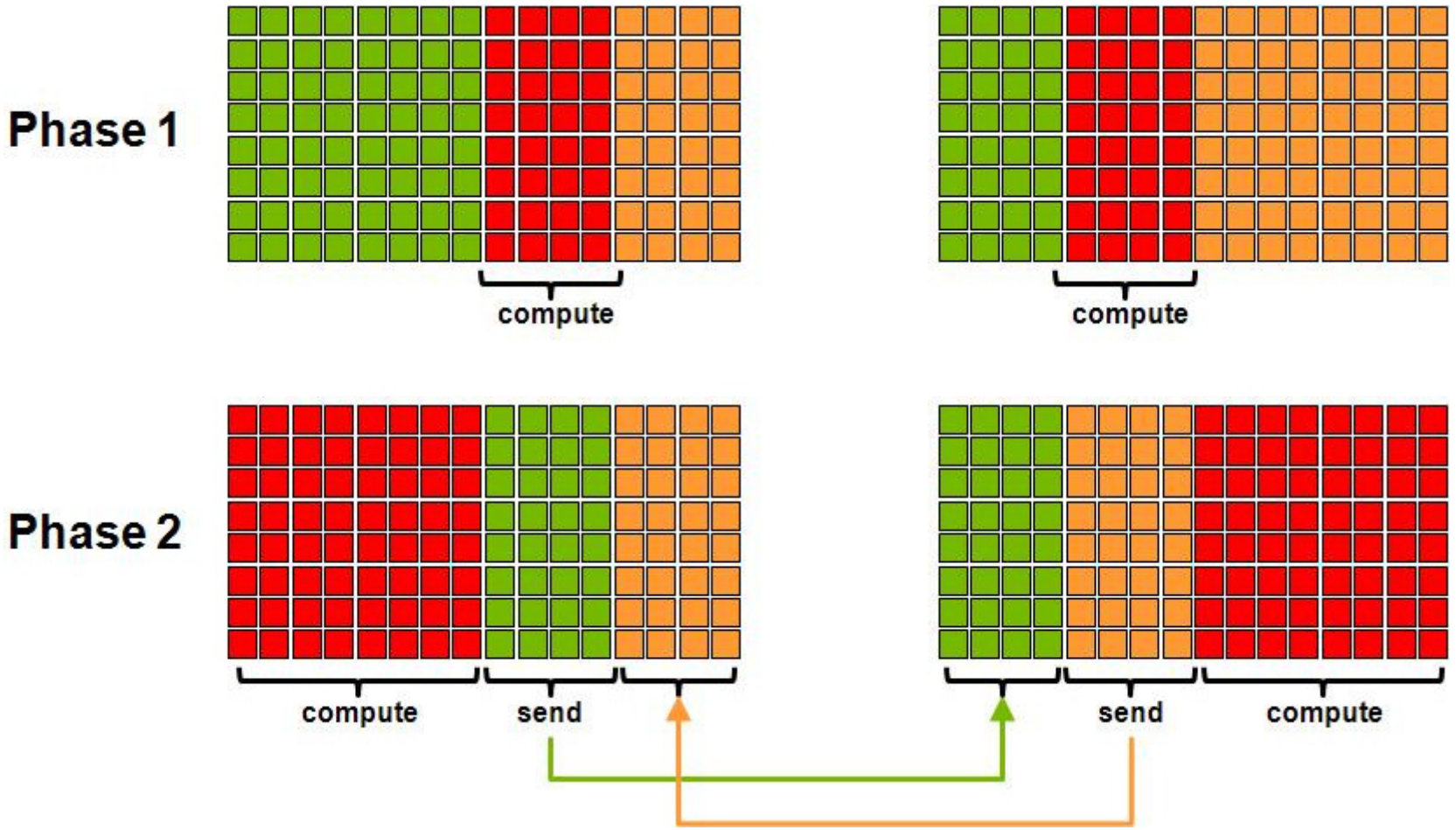
- **Test with 2 GPUs:**
  - Split the data volume between 2 GPUs
  - Split along the slowest-varying dimension
  - Each GPU gets (dimz+4) slices







Every Time Step



Streams and async memcopies are used to overlap computation and communication in Phase 2

Stencil-only 2-GPU Communication Code


```
for(int i=0; i<num_time_steps; i++)
{
        launch_kernel( d_output+offset1, d_input+offset1, dimx,dimy,12, stream1);

        launch_kernel( d_output+offset2, d_input+offset2, dimx,dimy,dimz, stream2 );
        cudaMemcpyAsync( h_ghost_own, d_ghost_own, num_ghost_bytes, cudaMemcpyDeviceToHost, stream1 );
        cudaStreamSynchronize( stream1 );
        MPI_Sendrecv(  h_ghost_own,      num_ghost_elmnts,  MPI_REAL,  partner,  i,
                       h_ghost_partner,  num_ghost_elmnts,  MPI_REAL,  partner,  i,
                       MPI_COMM_WORLD, &status );
        cudaMemcpyAsync( d_ghost_partner, h_ghost_partner, num_ghost_bytes, cudaMemcpyHostToDevice, stream1 );

        cudaThreadSynchronize();
}
```


Performance Scaling with 2 GPUs

16×16 Tile Finite Difference Kernel

Data Dimensions	Scaling
480 × 480 × 200	1.51
480 × 480 × 300	1.93
480 × 480 × 400	2.04
544 × 544 × 544	2.02
640 × 640 × 640	2.26
800 × 800 × 400	2.04

Using 2 GPUs (half of Tesla S1070)

3DFD Scaling with Multiple GPUs (8^{th} order in space, 2^{nd} order in space)

Data Dimensions	1 GPU	2 GPUs	4 GPUs
480 × 480 × 800	1.00	1.99	3.90
480 × 480 × 1200	1.00	2.00	4.05
640 × 640 × 640	1.00	2.17	3.62

Each GPU communicates with 2 neighbors:
twice the communication cost

Using 2 Tesla S1070s (connected to 4 CPU nodes)