GPU TECHNOLOGY CONFERENCE

Implementing 3D Finite Difference Codes on the GPU

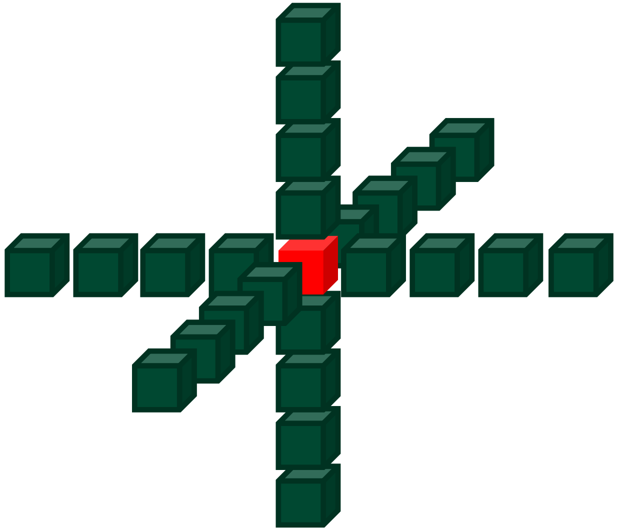
Tom Reed 07-14-2010

Outline

- **Single GPU Implementation**
  - 2-pass and 1-pass approaches
  - Performance evaluation
- **Multi-GPU Implementation**
  - Scalability on clusters

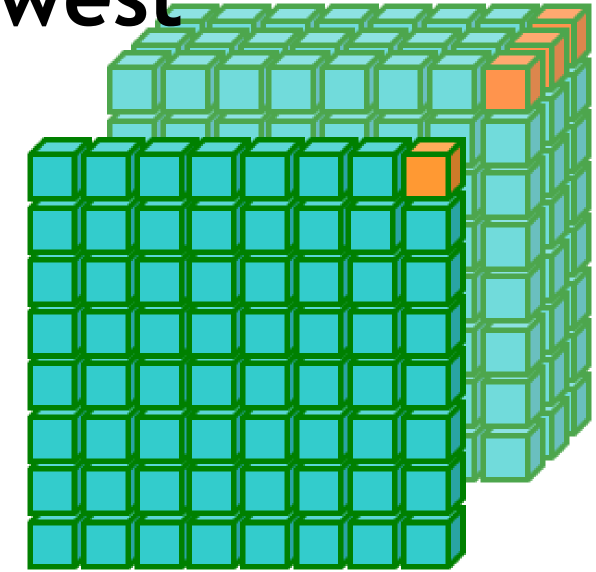
3D Finite Difference

- 25-point stencil (8th order in space)
- Isotropic: 5 distinct coefficients
- For each output element we need:
  - 29 flops
  - 25 input values
- Some applications:
  - FD of the wave equation (oil & gas exploration)



General Approach

- **Tile a 2D slice with 2D threadblocks**
  - Slice is 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
  - Also helps data reuse

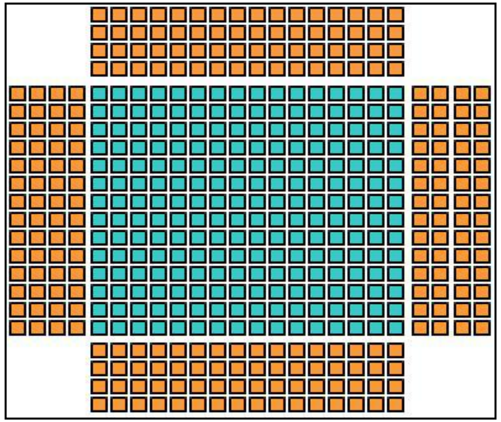


Naive Implementation

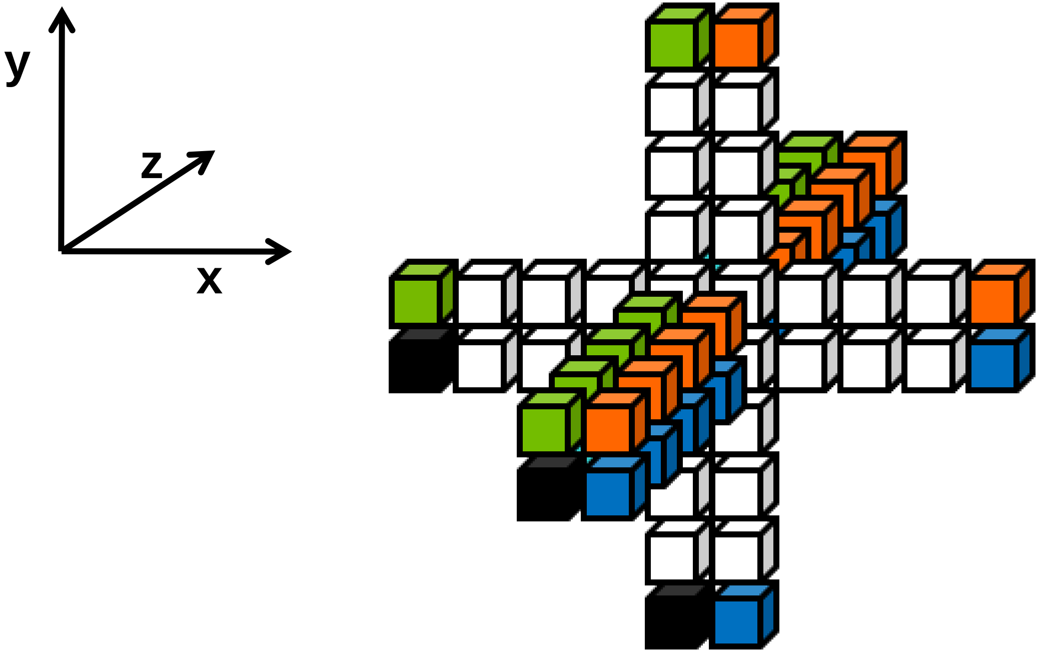
- **One thread per output element**
- **Fetch all data for every output element**
  - Redundant: input is read 25 times
  - Required bandwidth = 25 reads, 1 write (26x)
- ***Access Redundancy:***
  - Metric for evaluating implementations
  - Ratio between the elements accessed and the elements processed
    - *Appropriate for user-managed-cache architectures*
- **Optimization: share data among threads**
  - Use shared memory for data needed by many threads
  - Use registers for data not shared among threads

Using Shared Memory: 2 Passes

- **3DFD done with 2 passes:**
  - 2D-pass (2DFD)
  - 1D-pass (1DFD and output of the 2D-pass)
- **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
- **Volume accesses:**
  - Read/write for both passes
  - 16x16 subdomain tiles: 6.00 times
  - 32x32 subdomain tiles: 5.50 times
  - 56x56 subdomain tiles: 5.29 times



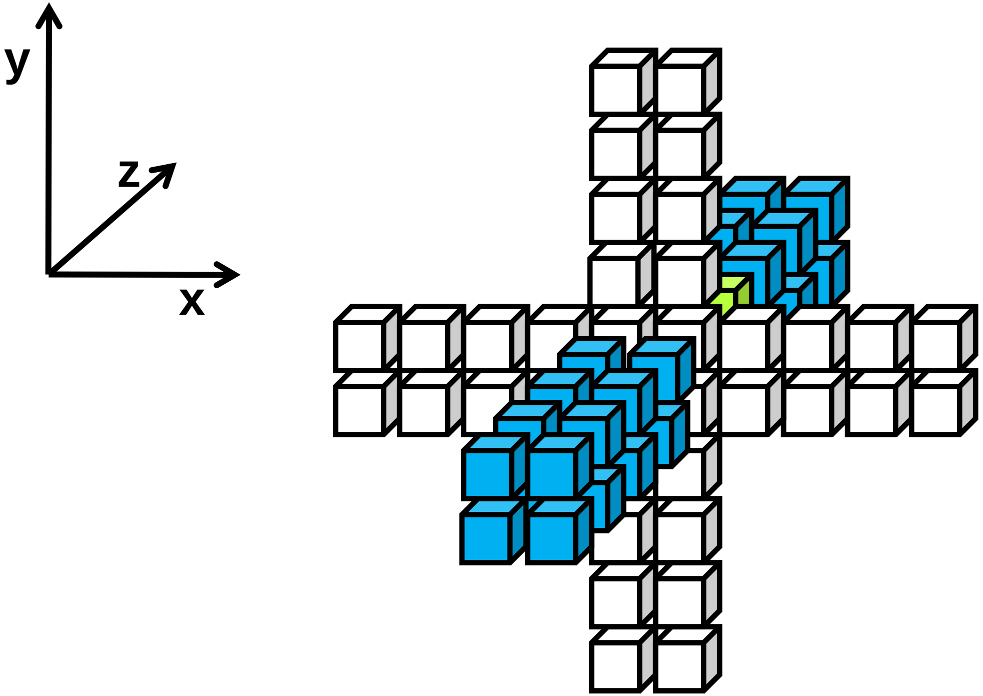
Input Reuse within a 2x2 Threadblock



- ⬜ Used by at least 2 threads
- 🟩 Used only by thread (0,0)
- 🟧 Used only by thread (0,1)
- ⬛ Used only by thread (1,0)
- 🟦 Used only by thread (1,1)

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

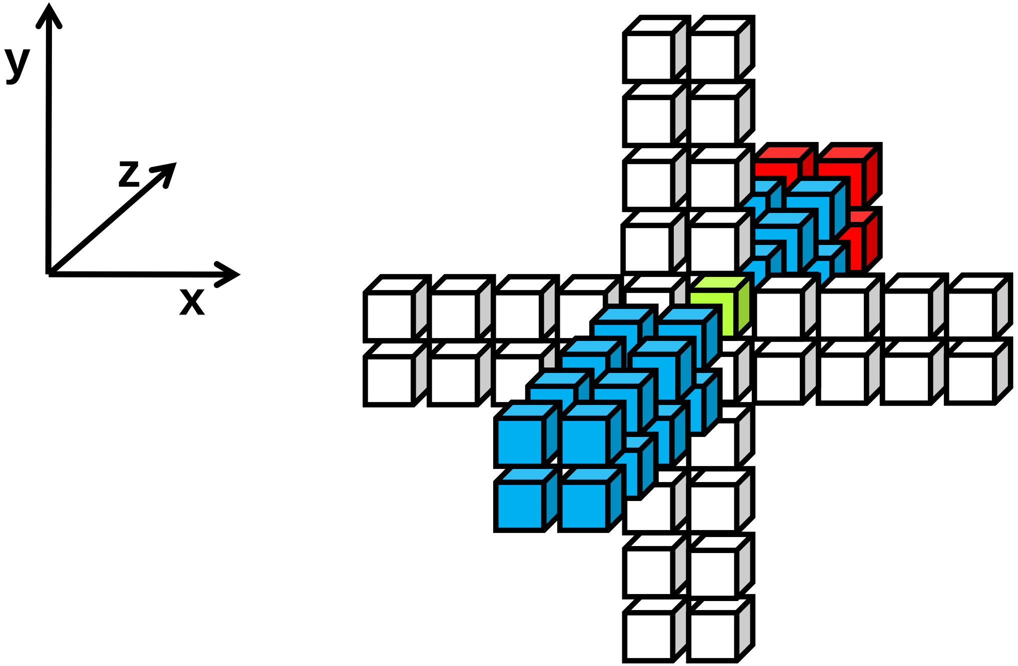
Process at *z* = *k*



- □ Stored in SMEM
- 🟦 Stored in registers
- 🟩 Value we “track”

GPU TECHNOLOGY CONFERENCE

Process at $z = k+1$



⬜ Stored in SMEM

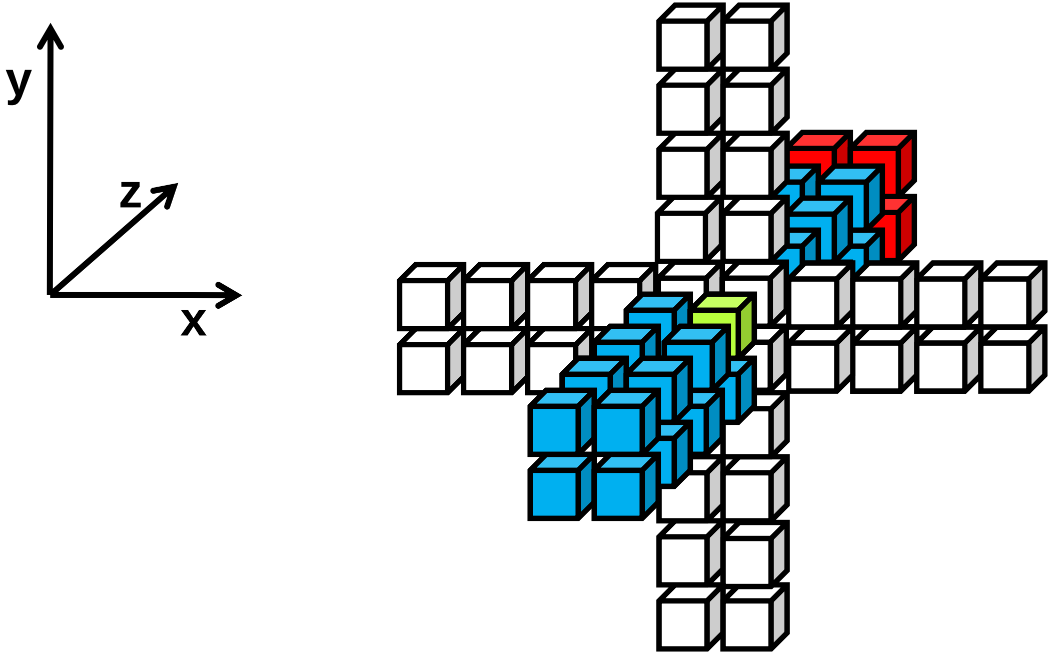
🟦 Stored in registers

🟩 Value we "track"

🟥 Newly-read value

NVIDIA.

Process at $z = k+2$



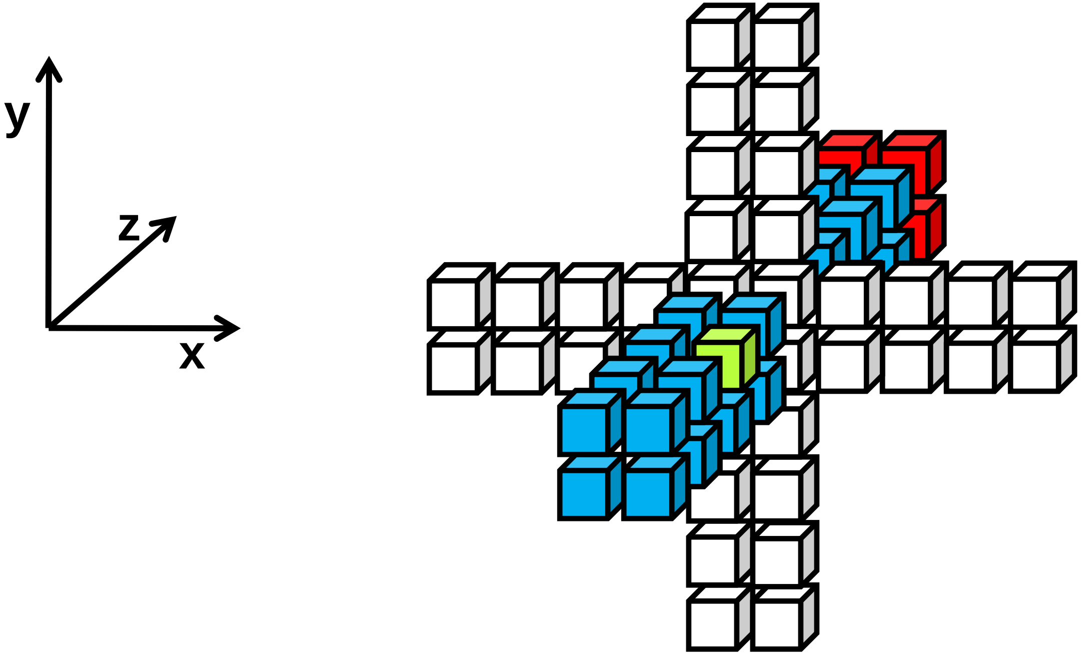
⬜ Stored in SMEM

🟦 Stored in registers

🟩 Value we “track”

🟥 Newly-read value

Process at $z = k+3$



□ Stored in SMEM

🟦 Stored in registers

🟩 Value we “track”

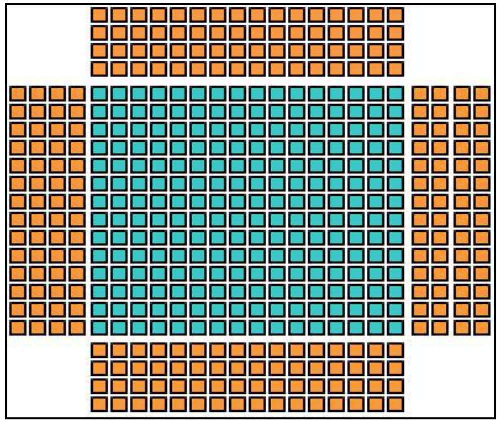
🟥 Newly-read value

Inner Loop of the 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 + block_dimy + radius][tx] = g_input[out_idx + block_dimy * dimx];
}
if( threadIdx.x<radius )    // left and right halo
{
   s_data[ty][threadIdx.x]                                      = g_input[out_idx – radius];
   s_data[ty][threadIdx.x + block_dimx + radius] = g_input[out_idx + block_dimx];
}
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;
```

Using Shared Memory: Single Pass

- **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)



25-point Stencil Performance

Implementation	naïve	2-pass	1-pass	1-pass
Tile dimensions	16x16	32x32	16x16	32x32
App. Redundancy	26.00	5.50	3.00	2.50
Hw. Redundancy	29.80	5.75	3.50	2.75
Gpoints/s	0.57	3.03	3.73	4.16

Performance for Various Stencil Sizes

order	Throughput				Redundancy		SM
	Gpoints/s	instr	GB/s, app	GB/s, hw	app	hw	threads
2	5.09	0.87	45.97	63.81	2.250	3.125	1024
4	4.97	0.82	49.88	64.79	2.500	3.250	768
6	4.03	0.81	44.60	54.67	2.750	3.375	768
8	3.73	0.88	45.08	52.54	3.000	3.500	512
10	3.50	0.79	45.93	51.18	3.250	3.625	512
12	3.22	0.69	45.55	48.77	3.500	3.750	512

16x16 tile, 640x640x400 volume

Multi-GPU Implementation

- **Hardware:**
  - 2 GPUs per cluster node
  - Infiniband SDR interconnect
- **Approach:**
  - Partition the volume among CPU threads
    - Each CPU thread drives its own GPU
    - Use standard APIs (MPI/OpenMP) to run multiple CPU threads
  - Partition along the slowest-varying dimension
    - Ghost regions will be contiguous in memory
- **Performance numbers: Gpoints/s for 3DFD**
  - Single GPU: 3 Gpoints/s (compared to 3.7 Gpoints/s for stencil-only)
  - 2 more reads and 4 more flops per output element, when compared to stencil only

Inner Loop of the 3DFD 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 + block_dimy + radius][tx] = g_input[out_idx + block_dimy * dimx];
}
if( threadIdx.x<radius )    // left and right halo
{
   s_data[ty][threadIdx.x]                                  = g_input[out_idx – radius];
   s_data[ty][threadIdx.x + block_dimx + radius] = g_input[out_idx + block_dimx];
}
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];
```

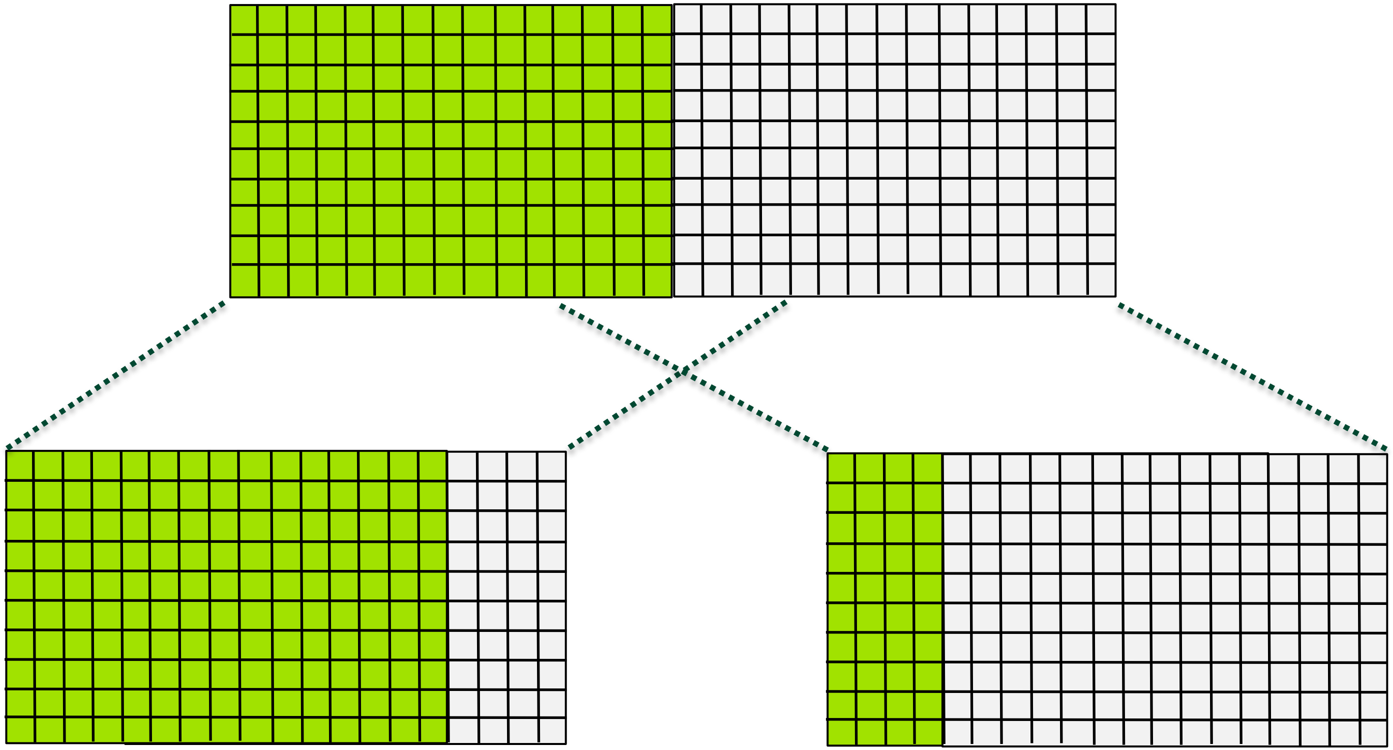
2 more GMEM reads
4 more FLOPS

Per output element:

- **33 FLOPS**
- **5 GMEM accesses (32bit)**

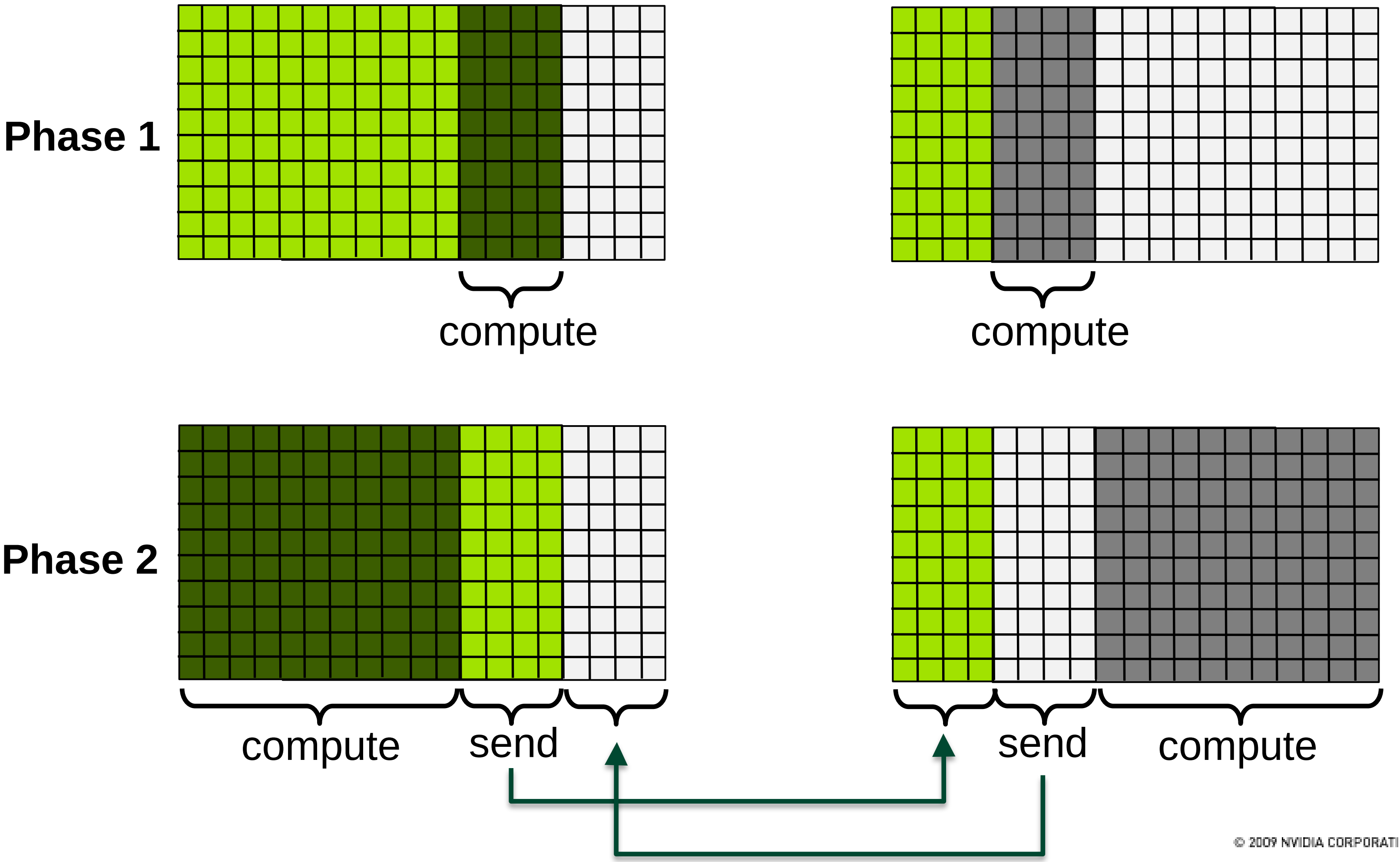
GPU TECHNOLOGY CONFERENCE

Volume Partitioning for 2 GPUs

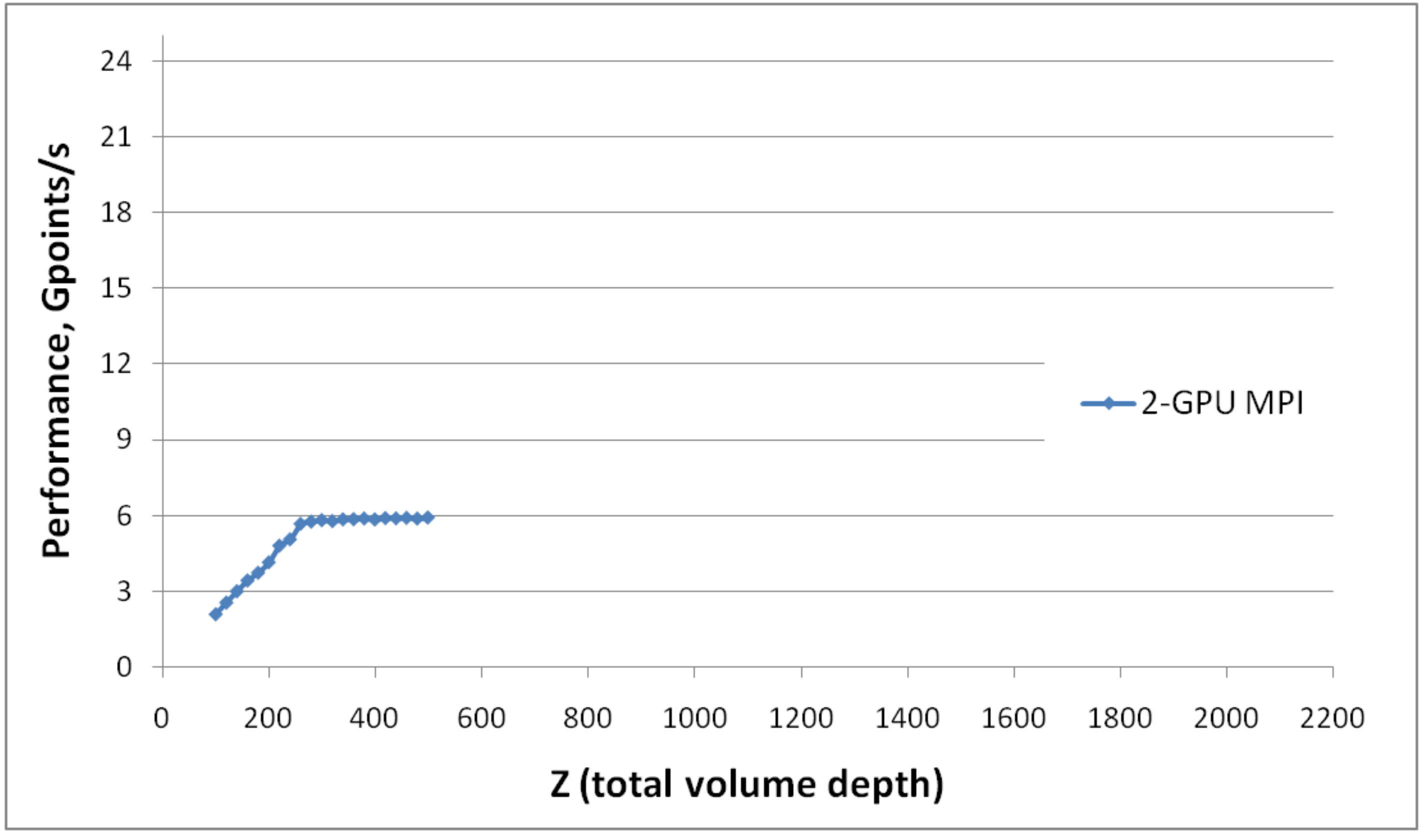


NVIDIA.

Processing Phases for 2 GPUs

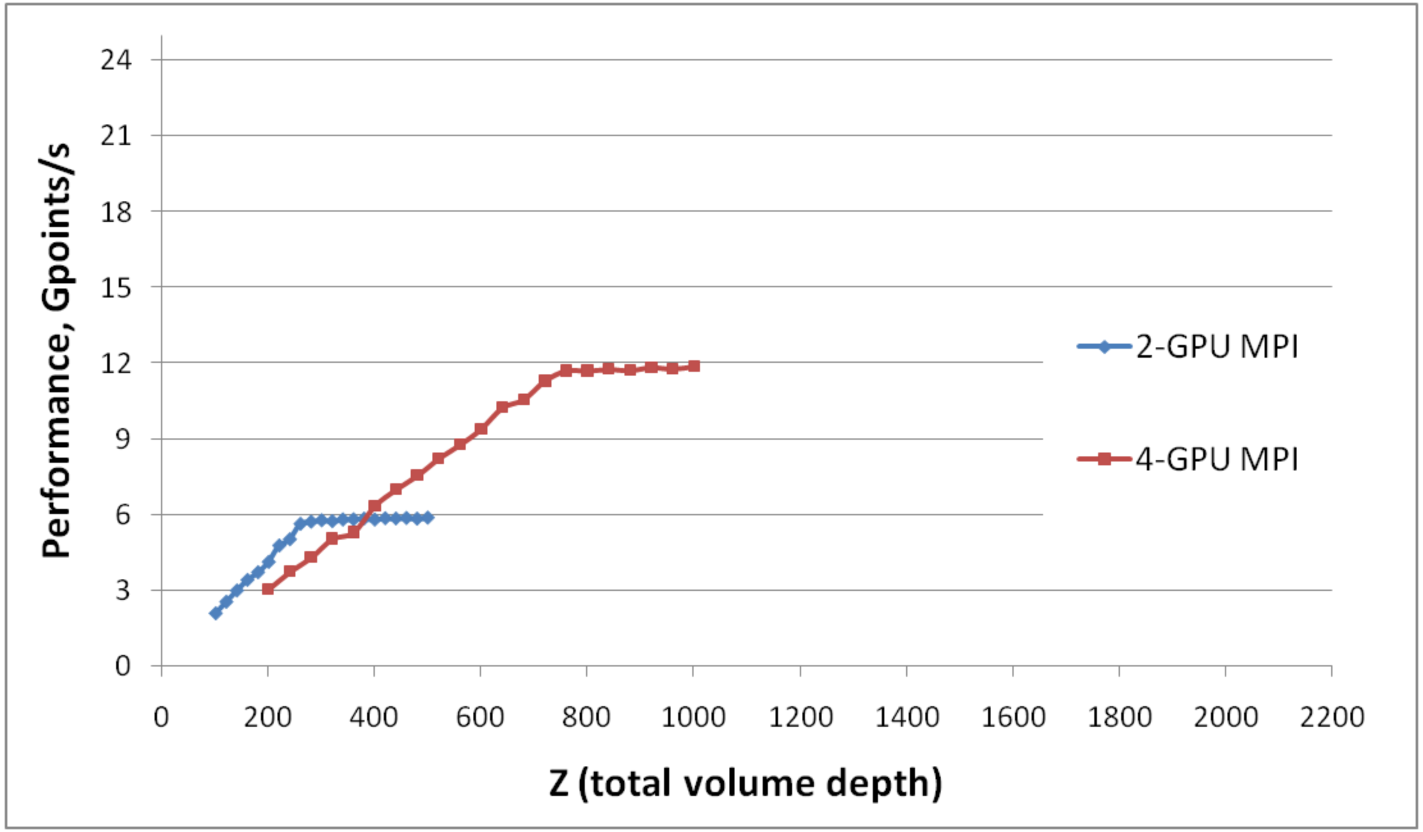


2-GPU Performance using MPI



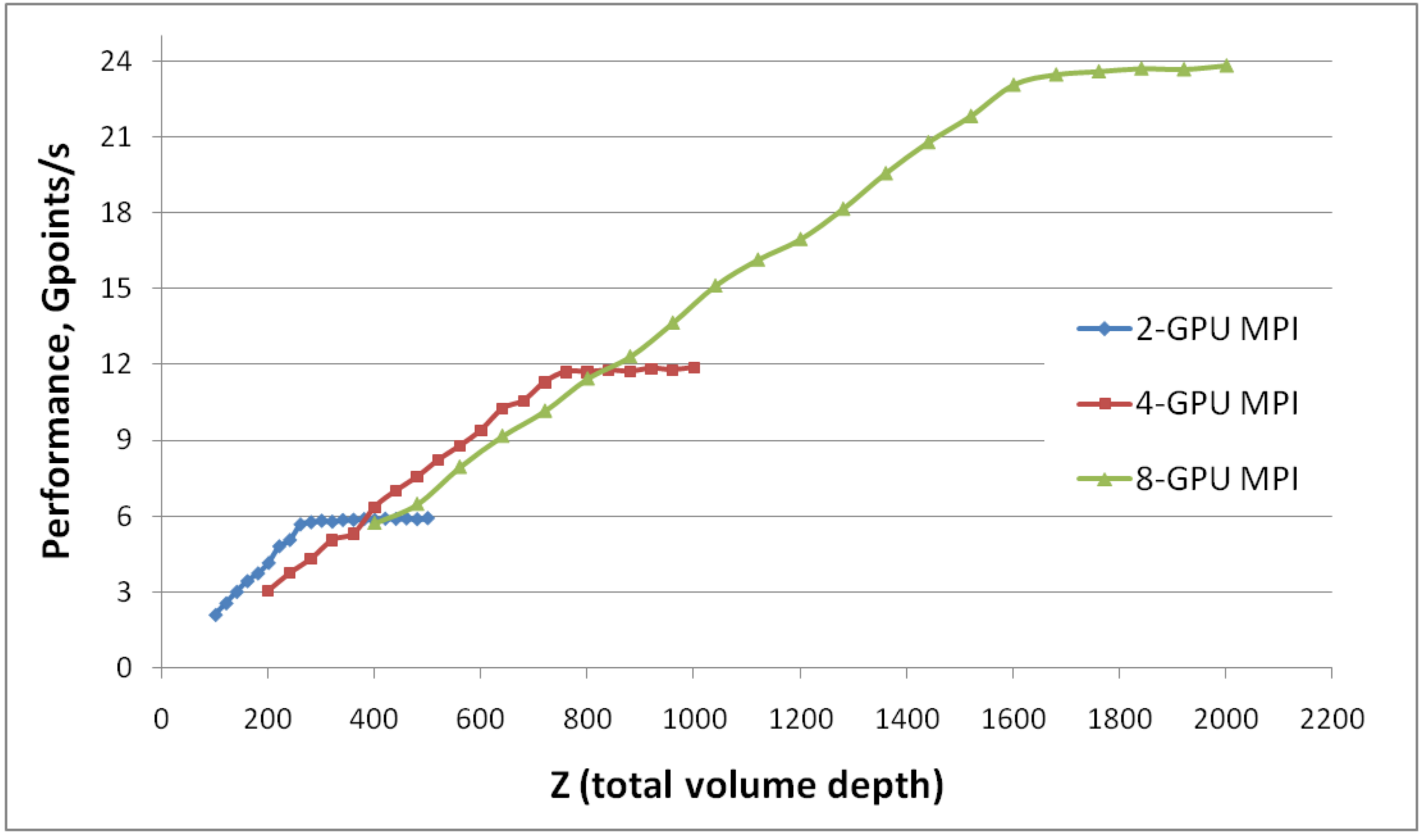
Communication per node: 4 PCIe memcopies, 1 MPI_Sendrecv

4-GPU Performance using MPI



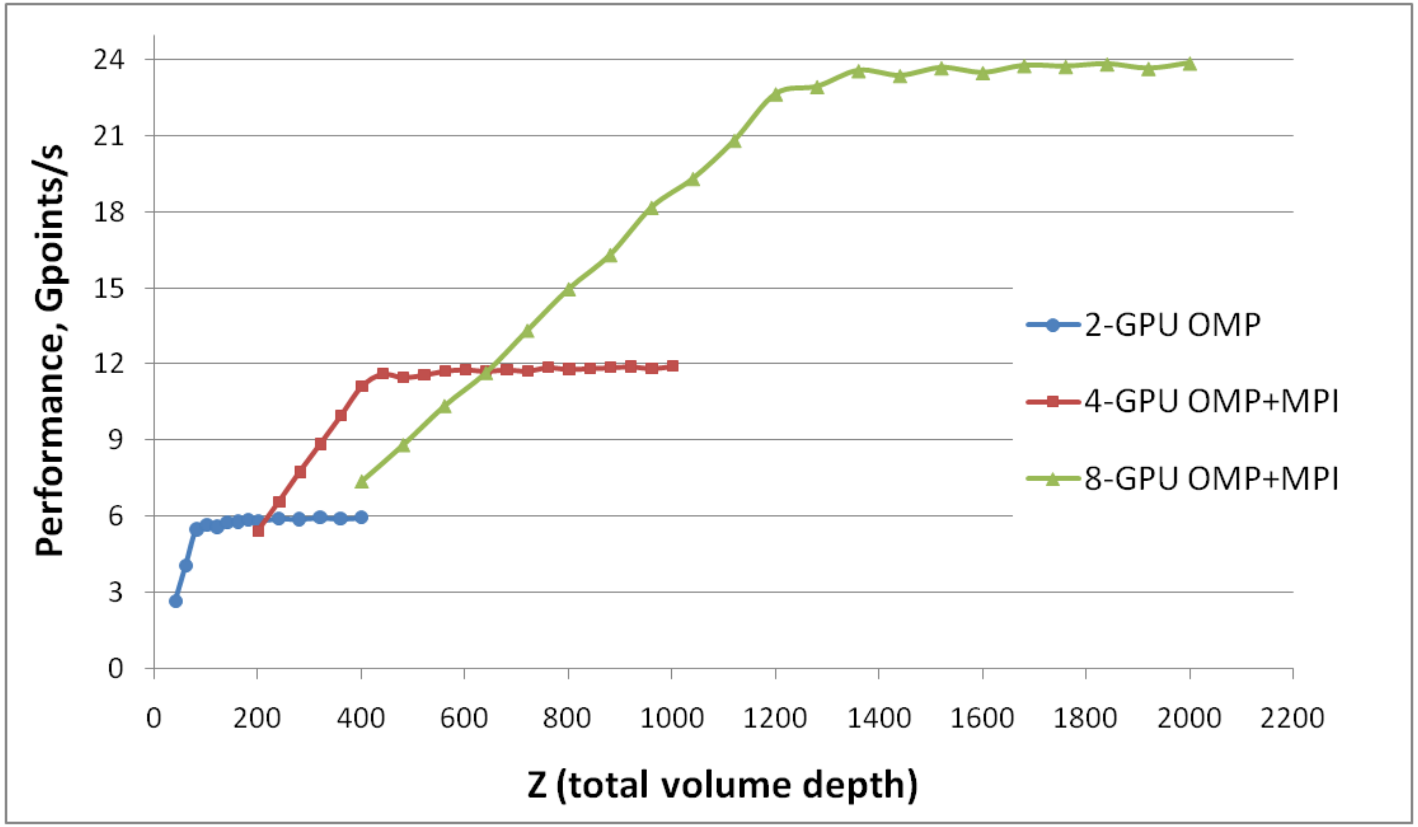
Communication per node: 8 PCIe memcopies, 2 MPI_Sendrecv

Multi-GPU Performance using MPI

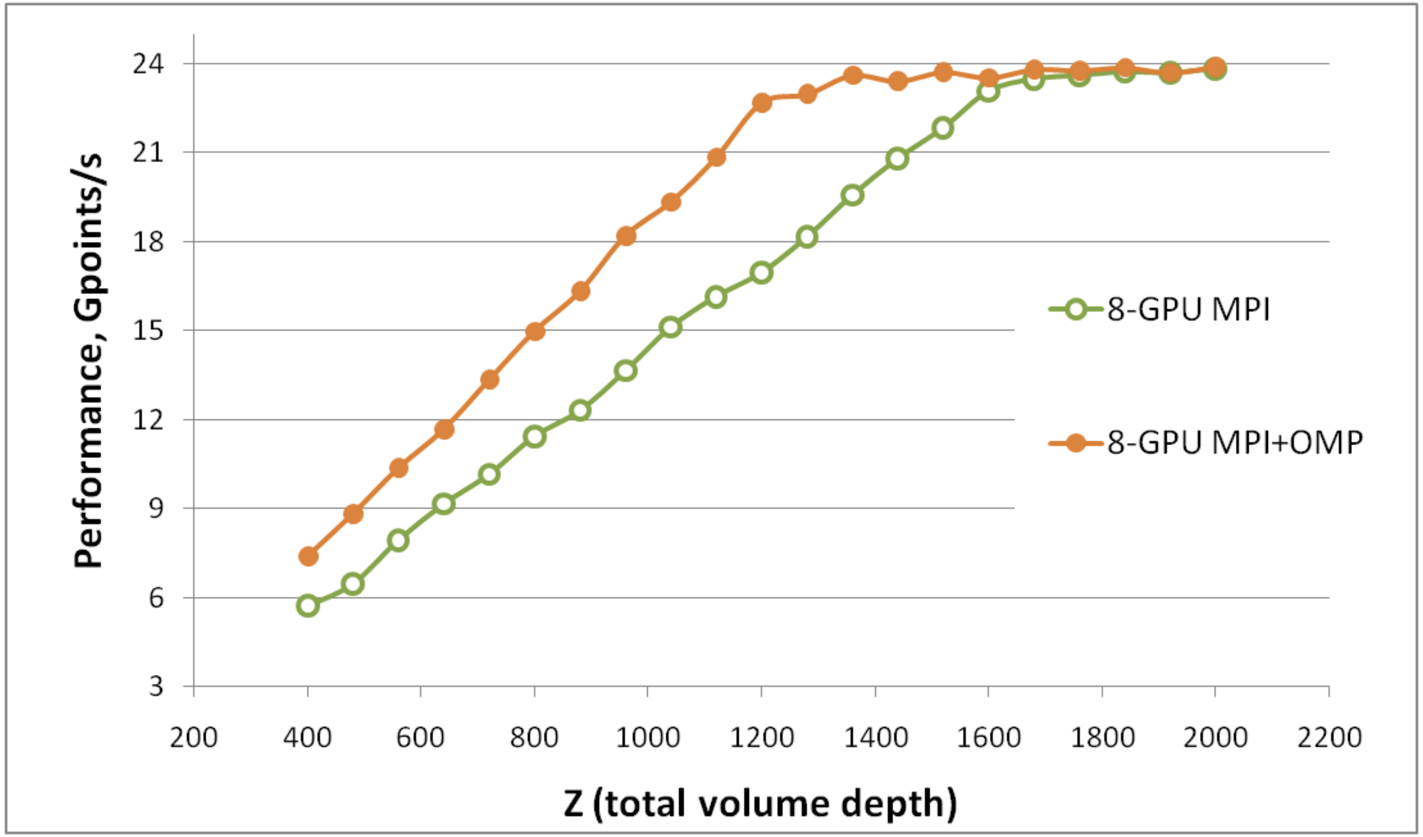


Communication per node: 8 PCIe memcopies, 3 MPI_Sendrecv

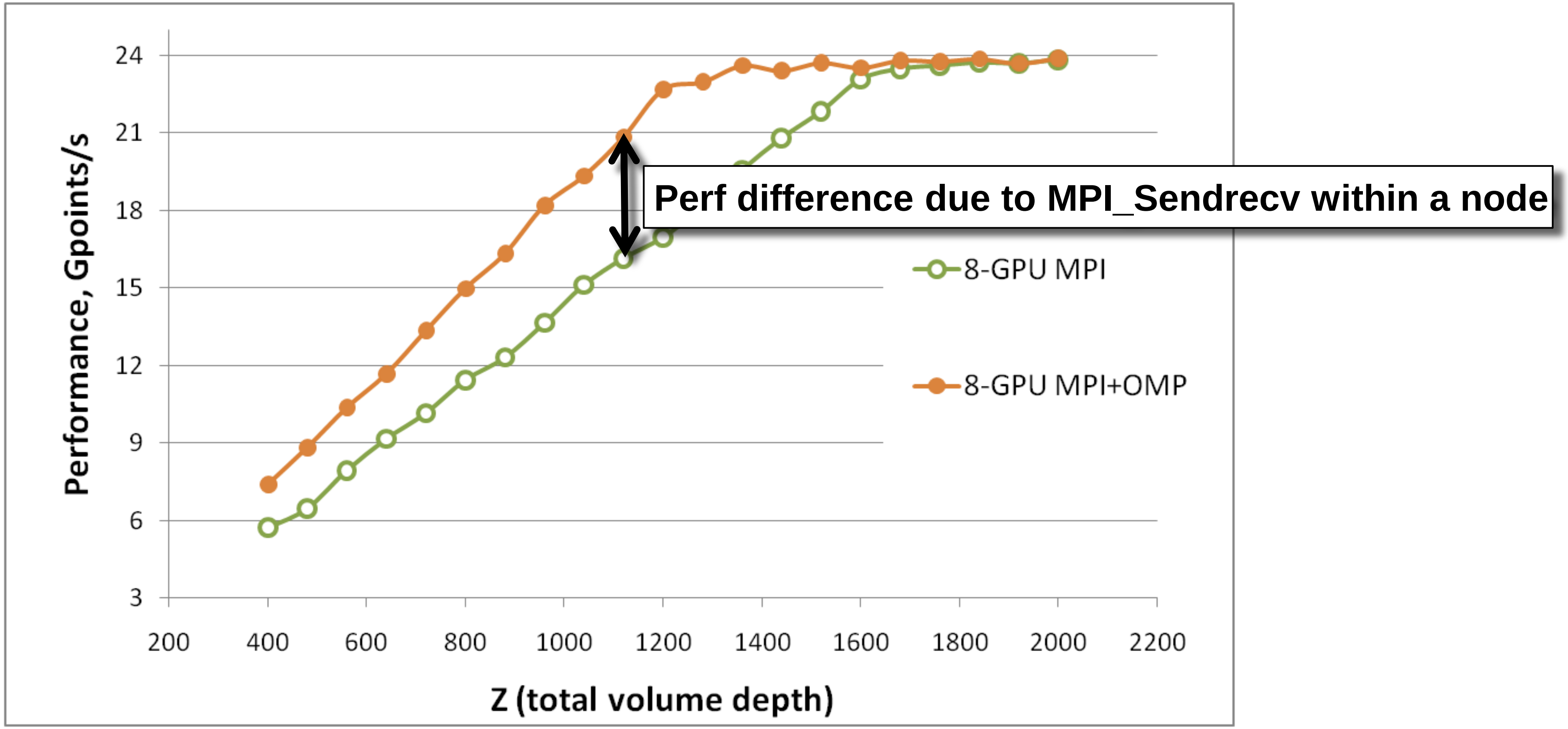
Multi-GPU Performance Using MPI+OpenMP



Comparing MPI and MPI+OMP performance

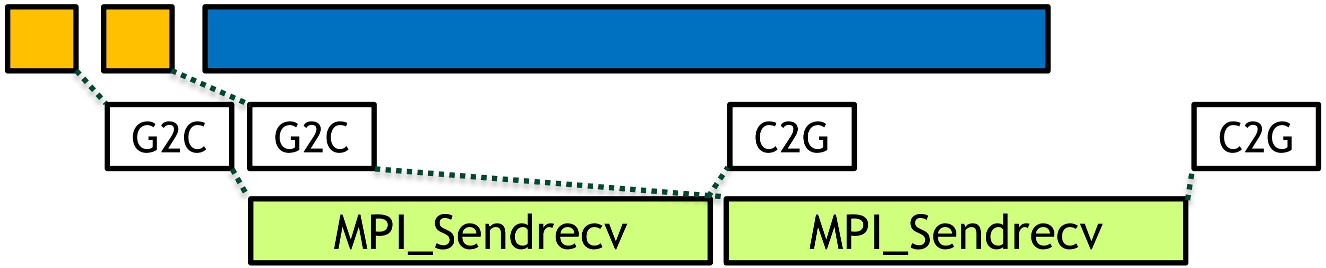


Comparing MPI and MPI+OMP performance



Perf difference due to MPI_Sendrecv within a node

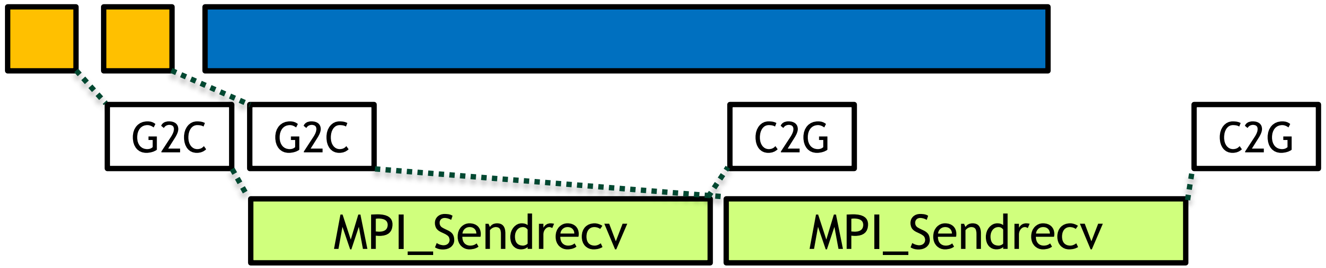
Communication and Computation Overlap



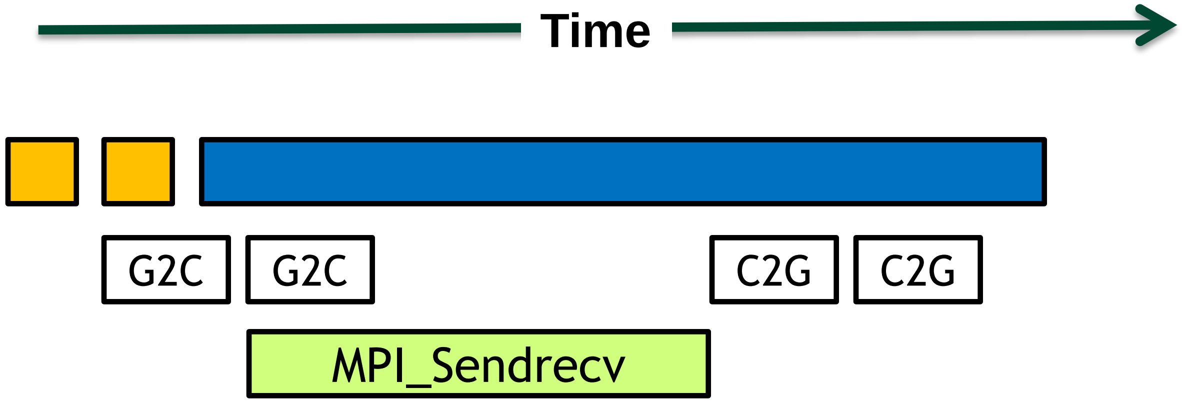
Communication and computation when using only MPI

- 🟧 Ghost node computation
- 🟦 Internal node computation
- ⬜ PCIe communication (CPU-GPU)
- 🟩 MPI communication (CPU-CPU)

Communication and Computation Overlap



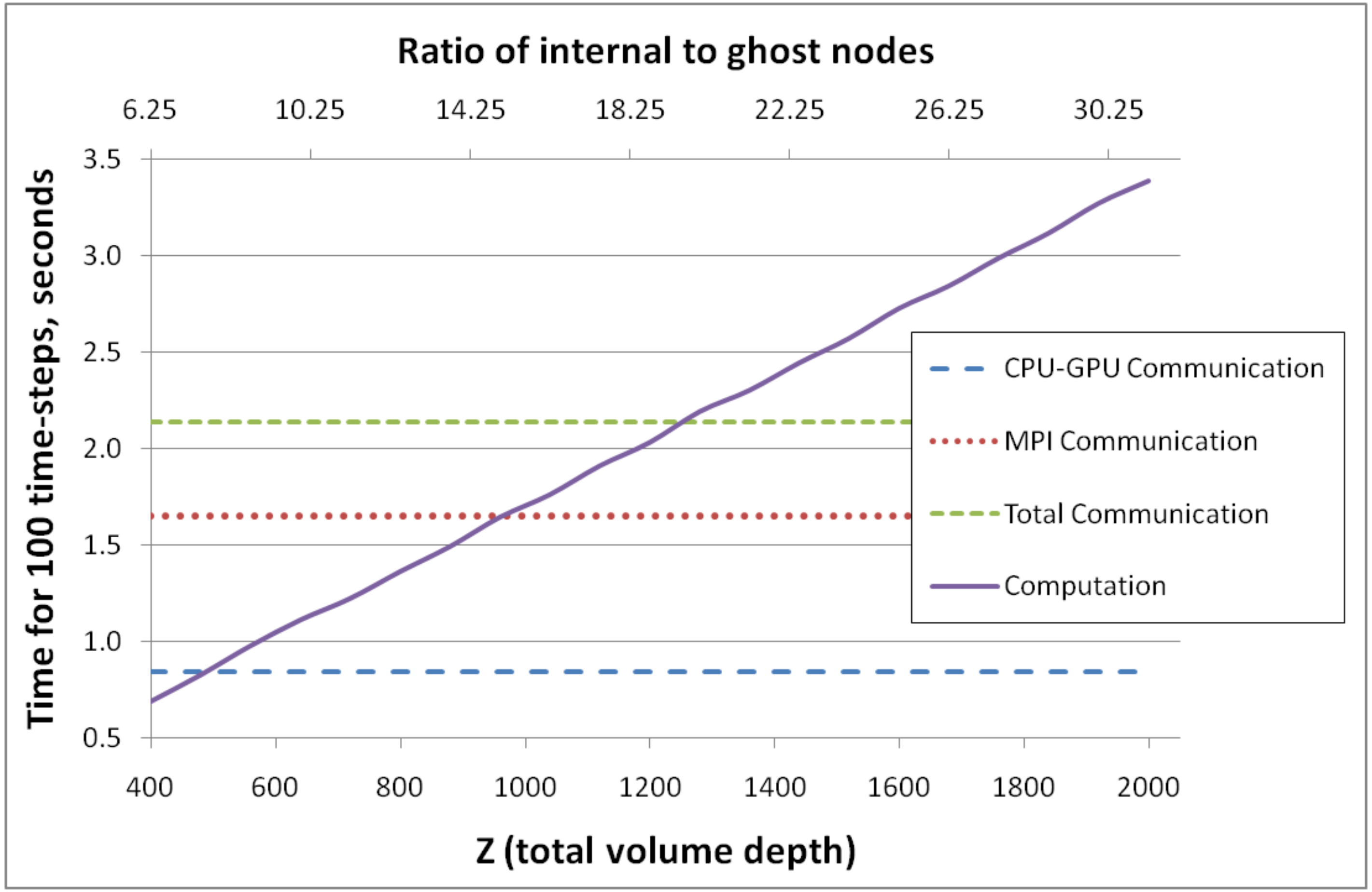
Communication and computation when using only MPI



Communication and computation when using MPI and OpenMP

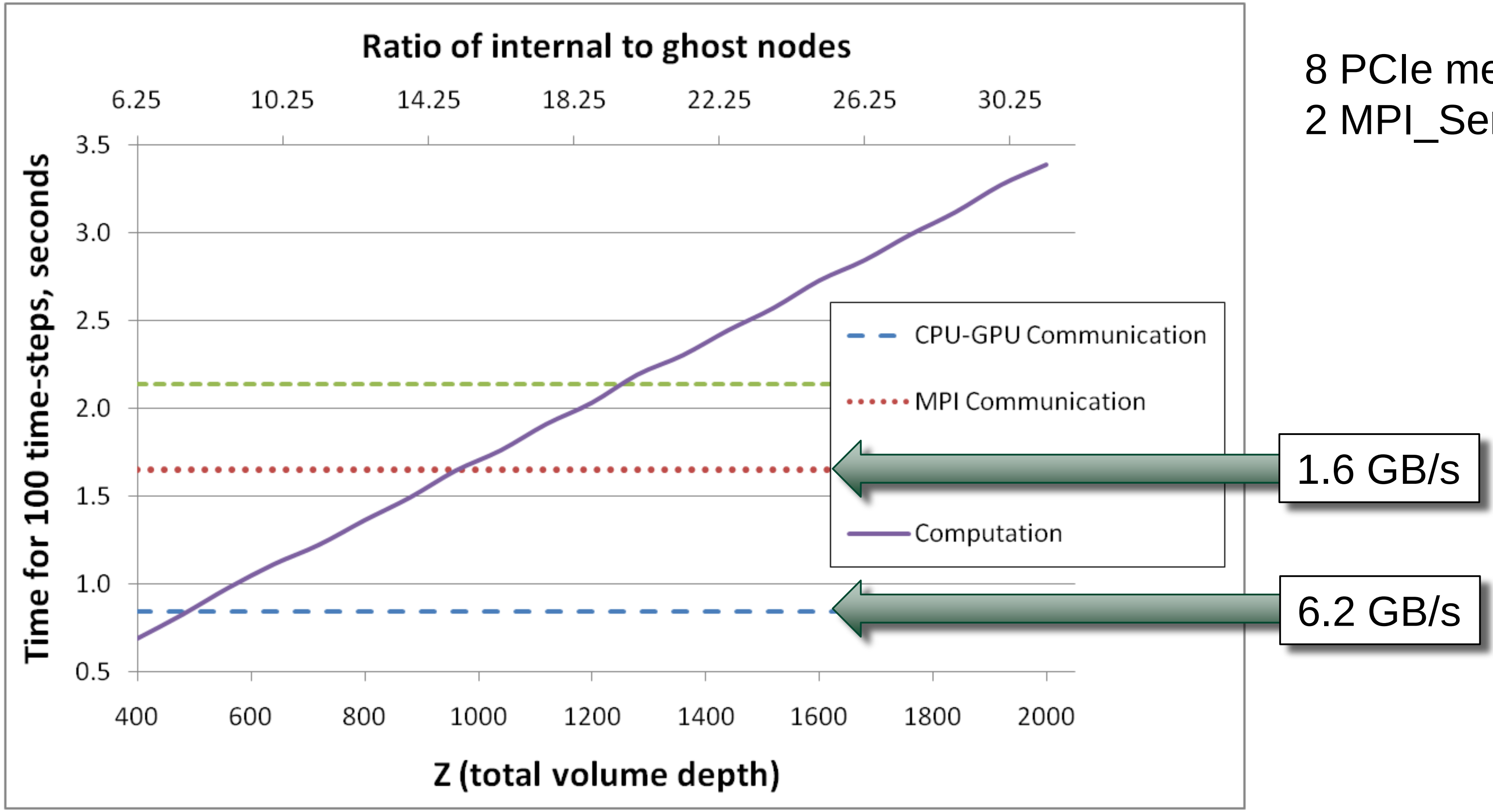
- 🟧 Ghost node computation
- 🟦 Internal node computation
- □ PCIe communication (CPU-GPU)
- 🟩 MPI communication (CPU-CPU)

Computation and Communication Costs per Node



8 PCIe memcopies
2 MPI_Sendrecv

Computation and Communication Costs per Node



1.6 GB/s

6.2 GB/s

8 PCIe memcopies
2 MPI_Sendrecv

Conclustions

- **GPU achieves high throughput for 3DFD over regular grids**
  - Bandwidth and instruction rate advantage over CPUs
- **Multi-GPU 3DFD performance scales linearly**
  - For large enough problems, communication is hidden by computation on internal data
  - “large enough” depends on stencil size and cluster configuration
    - Number of network and PCIe transfers
- **Multi-GPU scaling can be reliably modeled**
  - Using throughput rates for the GPU, network, and PCIe bus

Questions?