Accelerating FUN3D on Hybrid Many-Core Architectures

Austen C. Duffy

National Institute of Aerospace

August 3, 2010

This work is with

- Dana Hammond - NASA Langley Research Center, Computational Aerosciences Branch
- Eric Nielsen - NASA Langley Research Center, Computational Aerosciences Branch

Outline

Introduction

The Age of Hybrid Computing

High performance computing systems are undergoing a rapid shift towards the coupling of add-on hardware, such as graphics processing units (GPUs), Cell processors, and field programmable gate arrays (FPGAs), with traditional multicore CPU's.

These emerging systems are referred to as 'hybrid' or 'heterogeneous' computing architectures, the hardware components are referred to as 'accelerators' or 'application accelerators'

Pros and Cons

Hybrid architectures benefit from

- Additional fine grain parallelism at the node level
- Better price to performance ratios (1 TFLOP peak from a single GPU)
- Lower energy consumption per TFLOP

But software applications require

- Special (and perhaps more difficult) coding
- Abundant parallelism
- Efficient ways to share limited add on resources

Energy Efficiency

The top 8 computers on the June 2010 Green 500 list are all hybrid architectures

- Accelerator-based supercomputers on The Green500 List have average efficiency of 554 MFLOPS/W
- Traditional supercomputers average efficiency is at a much lower 181 MFLOPS/W.
- The top 3 use a combination of Cell processors and FPGAs, with no traditional CPUs

http://www.green500.org/lists.php

Emerging Challenges

- Developing efficient algorithms for hybrid clusters - Ground up approach
- Updating old software to make best use of newly available hardware - Accelerator Models
- Models for sharing a single GPU among multiple CPU cores

Notable Hybrid HPC's

As of June 2010, these hybrid computers ranked among the 10 fastest in the world.

- #2 Nebulae - National Supercomputing Center in Shenzen, China. 9280 Intel Xeon 6 core CPUs, 4640 Nvidia Tesla C2050 GPUs (numbers etimated)
- #3 Roadrunner - Los Alamos. 6,480 AMD Opteron Dual Core CPUs, 12,960 IBM PowerXCell 8i Cell Processors
- #7 Tianhe-1 - National SuperComputer Center in Tianjin, China. 6144 Intel Xeon dual core CPUs, 5120 ATI GPUs

Emerging Trends for Hybrid Many-Core Clusters

Note that the older Tianhe-1 has a CPU core to GPU ratio of 2:1, while the newer Nebulae is 12:1

As CPU manufacturers continue to add more cores per chip, the number of CPU cores per GPU in a hybrid cluster will undoubtedly continue to increase due to high cost and power consumption of computational GPUs.

- Cost: Intel Xeon X5670 6 core CPU costs about $1500, NVIDIA Tesla C2050 GPUs are about $2300 each
- Power: The X5670 requires 95W, each additional C2050 adds 250W

GPU Architectures

GPU architectures differ significantly from CPU architectures, allocating chip space to many identical SIMD type functional units optimized for graphics rendering, as opposed to many different special purpose units found on CPUs. Until recently, GPUs have suffered from

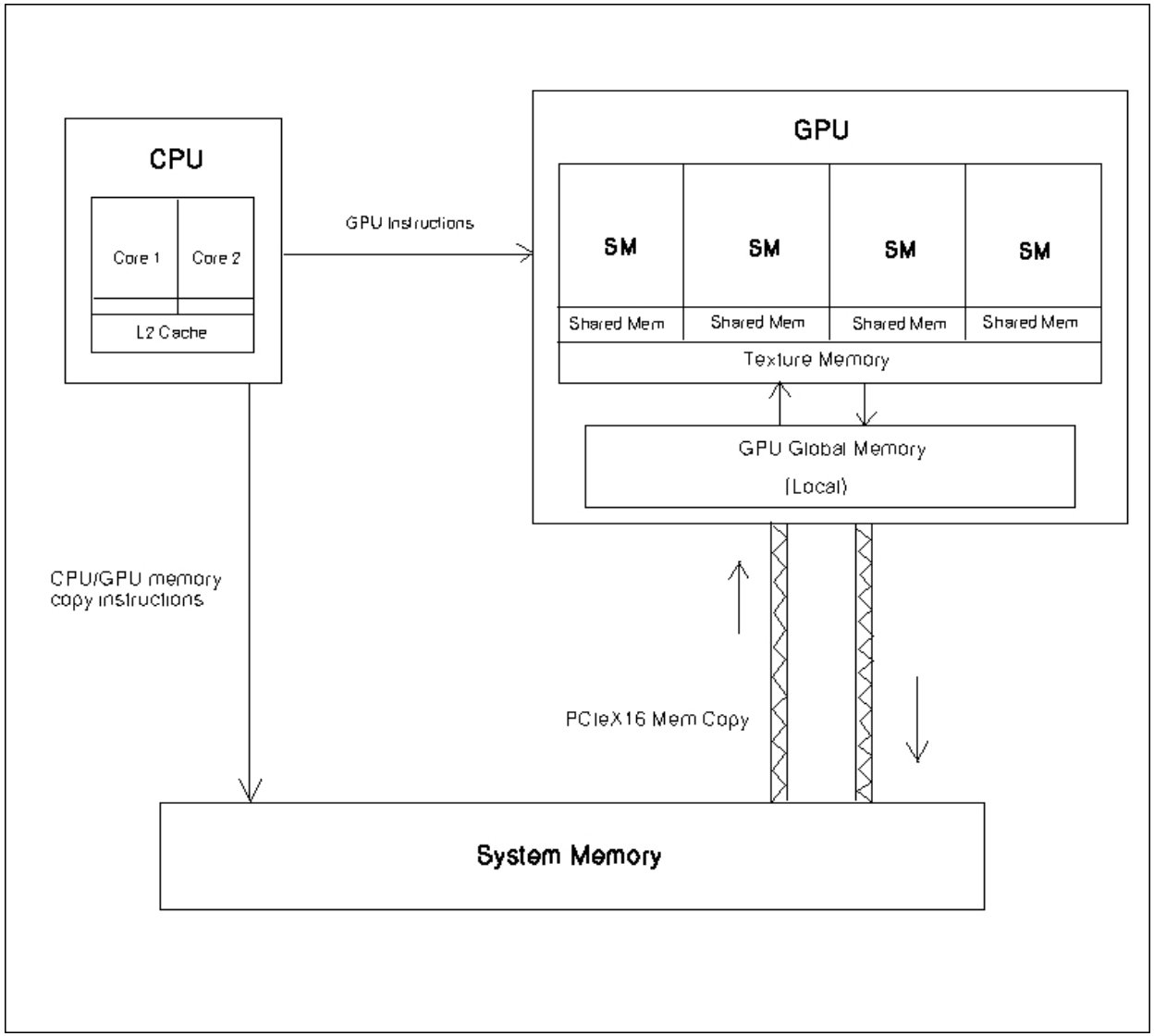
- Lack of true L1 and L2 caches
- Lack of double precision or slow DP when available
- Limited accuracy
- Serial kernel execution
- bit flip errors (no ECC support)

Memory Hierarchy

GPU memory layouts are slightly different than those in a cpu... in order of fastest access speed we have

- Registers: Fixed number per Streaming Multiprocessor (SM), spills go to local memory if true cache absent
- Constant Memory: Stored in Global, only fast when cached, difficult to use properly
- Shared Memory: Similar to L1 cache
- Texture: Similar to L2 cache
- Local: stored in global memory, just as slow
- Global: Big, slowest, but still much faster than CPU memory

GPU Memory Access



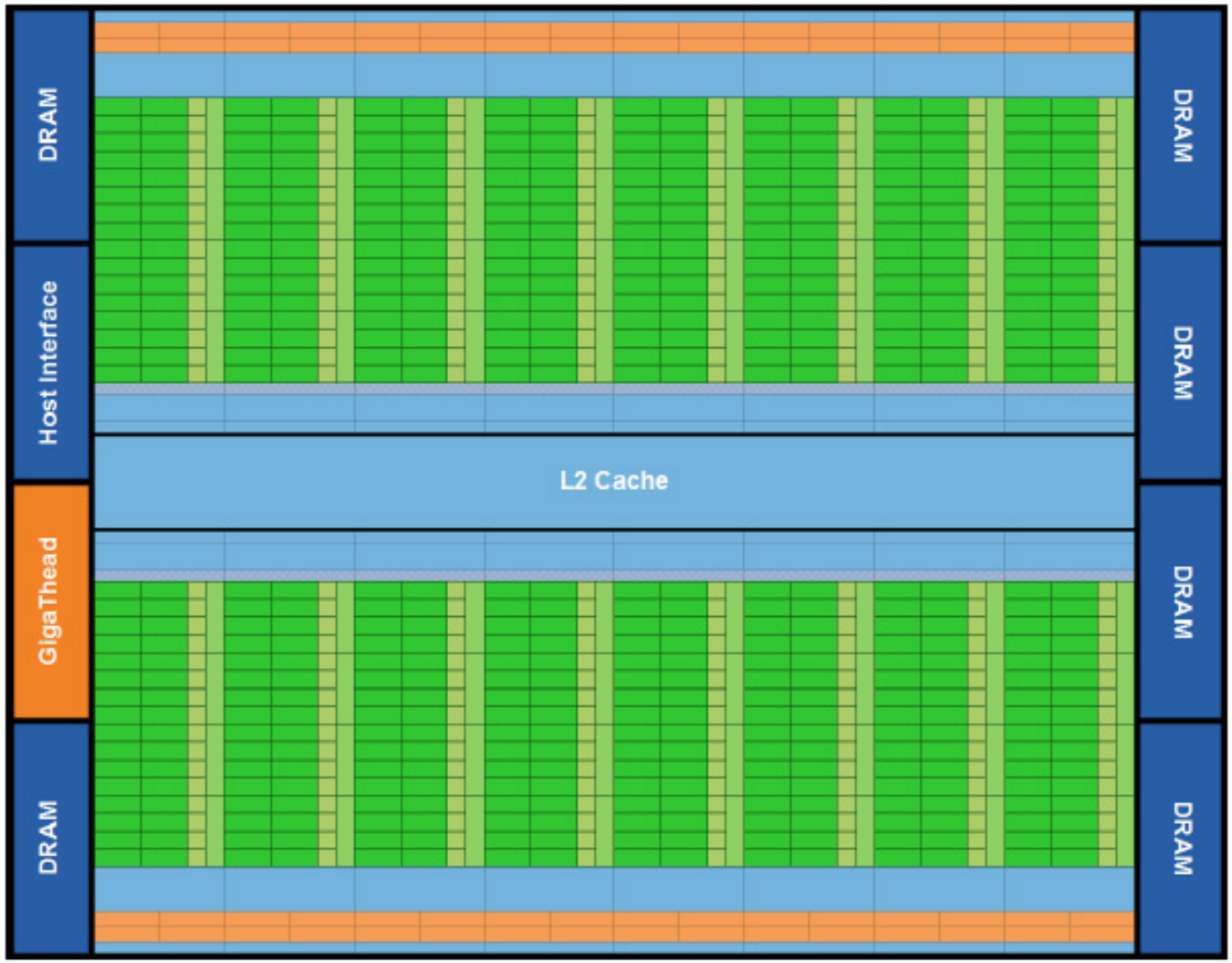
Latest Fermi Architecture

The latest generation GPU architecture from Nvidia, codenamed 'Fermi', makes significant leaps in GPU computing capabilities

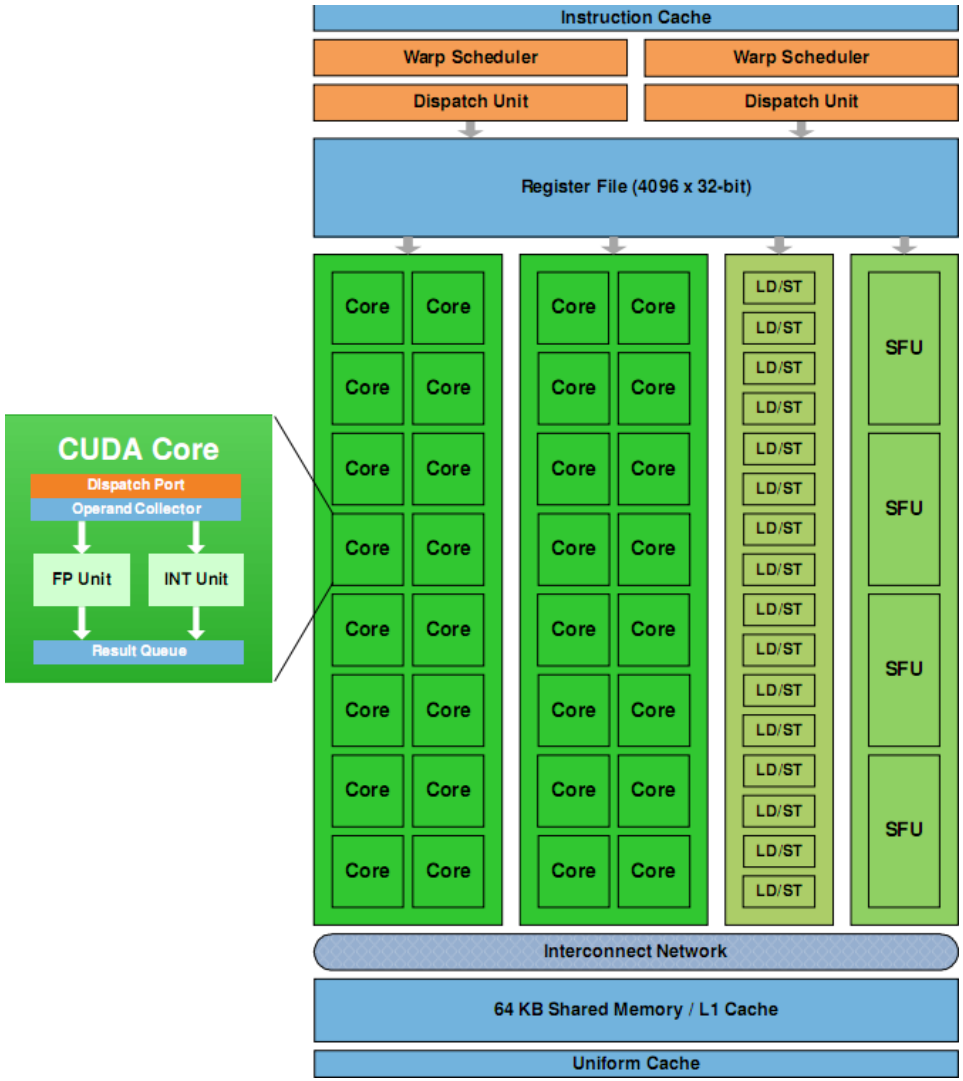
- Better IEEE 754-2008 Compliance = More Accurate Results
- Faster Double Precision (up to 8X)
- Introduction of true L1 and L2 Caches
- Concurrent Kernel Execution
- ECC memory support

NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper, NVIDIA Corp. 2009

NVIDIA Fermi Chip



Fermi Streaming Multiprocessor (SM)



Differences between the latest CPUs and Fermi GPUs

Fermi GPUs

- Cores: 448-480 (512 in the future)
- Execution Units: 1 FP, 1 INT
- Tasks: Data Parallel
- Threads: Thousands, lightweight, low overhead
- 32 cores per SM, which acts more like a CPU core

AMD and Intel CPUs

- Cores: 6-12
- Execution Units: Numerous, including FP, INT, ALU, SIMD
- Tasks: All, highly robust
- Threads: 6-12, heavy, significant overhead

Relevant Quote

"I know how to make 4 horses pull a cart - I don't know how to make 1024 chickens do it."

- *Enrico Clementi*

So how can a GPU accelerate my code?

Fine Grain Data Parallelism

- Most high end CFD codes are already optimized for large scale 'coarse grained' parallel computation
- Must seek fine grained data parallel tasks
- If these opportunities exist, the code can benefit from unprecedented levels of mass parallelism

GPU Programming Languages/Software

There have been numerous low level, difficult to program languages over the years that left GPU computing almost entirely in the hands of graphics programmers. Recent versions have brought GPU computing to the mainstream.

- `OpenGL` - Cross language graphics API, low level
- `CUDA C` - Nvidia, C language with GPU extensions
- `CUDA Fortran` - PGI wrapper for CUDA C
- `BrookGPU` - Stanford University graphics group compiler, C based w/ GPU extensions
- `Stream` - AMD, based on BrookGPU
- `OpenCL` - New cross platform language for GPUs and multicore CPUs

Fortran to CUDA

Since CUDA is just C with extensions for the GPU, the same mixed language rules apply (remember reverse indexing)

- Fortran `A(i,j,k)` $\rightarrow$ CUDA `A[k*nx*ny+j*nx+i]`
- Fortran `real(4),real(8)` $\rightarrow$ CUDA `float,double`
- Fortran `integer(4)` $\rightarrow$ CUDA `int`
- Fortran `function()` $\rightarrow$ CUDA `function_();`
- All arguments passed from Fortran are implicit pointers, but must be declared as such explicitly on the CUDA side

Simple CUDA Code Example: Vector Addition

Vector Addition in a parallel section of C code could look like

```
void vect_add(float *a, float *b, float *c, int N)
{
  for (int i=0; i<N; i++){
    c[i]=a[i]+b[i];
  }
}
```

But with CUDA, you could launch N threads, and each thread computes a single contribution in parallel on the GPU

```
__global__ void vect_add(float *a, float *b, float *c, int N)
{
  int i = threadIdx.x;
  c[i]=a[i]+b[i];
}
```

CUDA Kernels

A GPU function is referred to as a kernel. A kernel function is executed in parallel by lightweight threads that are easily created when the kernel starts and destroyed when it finishes. e.g. the kernel call

```
vect_add <<<1,512>>> (a,b,512);
```

Launches 1 block of 512 threads to compute the `vect_add` kernel for a 512 element array. We can have much more complex kernels, and can use 3D arrays mapped to 1D arrays with three dimensional thread indexing.

Advanced CUDA Kernel

A more complex example of what CUDA kernels are capable of can be found in Nvidia's sharedABMultiply

```
__global__ void shABmult(float *A, float *B, float *C, int N)
{
 __shared__ float aTile[TILE_DIM][TILE_DIM],
                  bTile[TILE_DIM][TILE_DIM];
 int row = blockIdx.y * blockDim.y + threadIdx.y;
 int col = blockIdx.x * blockDim.x + threadIdx.x;
 float sum =0.0f;
 aTile[threadIdx.y][threadIdx.x] = A[row*TILE_DIM+threadIdx.x];
 bTile[threadIdx.y][threadIdx.x] = B[threadIdx.y*N+col];
 __syncthreads();
 for (int i = 0; i < TILE_DIM; i++){
   sum += aTile[threadIdx.y][i] * bTile[i][threadIdx.x];
 }
 C[row*N+col] = sum;
}
```

NVIDIA CUDA Best Practices Guide. Version 3.0, 2/4/2010

CUDA Thread Blocks/Grids

Thread block sizes are 3 dimensional, grid sizes are 2 dimensional and are numbered with threadIdx and blockIdx identifiers. Dimensions are given by blockDim and gridDim

```
dim3 threads(8,8,4); // threadIdx.x --> 0-7, blockDim.x = 8
                     // threadIdx.y --> 0-7, blockDim.y = 8
                     // threadIdx.z --> 0-3, blockDim.z = 4

dim3 blocks(16,8,1); // blockIdx.x --> 0-15, gridDim.x = 16
                     // blockIdx.y --> 0-7,  gridDim.y = 8

kernel <<<blocks,threads>>> ();
```

CUDA Memory Copies

One must physically move data to the GPU with built in cudaMalloc and cudaMemcpy

```
// Allocate memory for a device copy of 'A' on GPU
cudaMalloc( (void **)&A_device, sizeof(float)*N );

// Make copy of 'A'
cudaMemcpy(A_device,A,sizeof(float)*N,cudaMemcpyHostToDevice);

...

// Copy back GPU updated 'A'
cudaMemcpy(A,A_device,sizeof(float)*N,cudaMemcpyDeviceToHost);

// Free 'A_device'
cudaFree(A_device);
```

CUDA Subroutine

Bringing it all together

```
extern C void fortran_to_cuda_(float *A, int *N)
{
  cudaMalloc( (void **)&A_device, sizeof(float)*N );
  cudaMemcpy(A_device,A,sizeof(float)*N,cudaMemcpyHostToDevice);

  dim3 threads(8,8,4);
  dim3 blocks(16,8,1);
  kernel <<<blocks,threads>>> (A,N);

  cudaMemcpy(A,A_device,sizeof(float)*N,cudaMemcpyDeviceToHost);
  cudaFree(A_device);
}
```

A Note on Realistic Speedup Expectations

GPU performance results have shown multiple orders of magnitude speedup for some problems. In a realistic high performance CFD setting these numbers will be rather difficult to achieve. Reasons for this include

- Quality CFD codes don't typically use dense linear algebra, slower converging embarrassingly parallel iterative solvers (e.g. Jacobi) or fully explicit solvers with small time step constraints. These algorithms will see the biggest performance gains on a GPU.
- CFD codes are typically run in parallel and hence require external communication, necessitating large numbers of costly CPU-GPU data transfers. This also means that complete iterative solves cannot be entirely completed on the GPU
- High end codes are highly optimized for a CPU, providing less opportunity for dramatic speedup.

Related Work

GPU acceleration has been extensively studied in numerous areas of computational science, engineering and finance. Some notes relevant here

- Most works to date have only considered single core - single GPU scenarios
- Many avoid the use of double precision
- Limited research has been done with high end CFD codes on GPU clusters
- No works to date have considered GPU sharing

Unstructured-Compressible

A. Corrigan, F. Camelli, R. Lohner, J. Wallin, Running unstructured grid based CFD solvers on modern graphics hardware, in: 19th AIAA CFD Conference, San Antonio, Texas, USA.

- Unstructured compressible Euler solver
- Explicit Runge-Kutta time stepping scheme
- Supersonic missile and wing test problems
- Averaged about 33X speedup single precision, 6X double with a Tesla C1060 over a single CPU core

Structured-Compressible

E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, J. Comp. Phys. 227 (2008) 10148-10161.

- Block structured compressible Euler, explicit Runge-Kutta
- Used BrookGPU with texture memory
- Hypersonic wing and vehicle test problems
- Achieved 15-40X speedup SP with -O2 optimization for 8800 GTX over a single CPU core

NASA OVERFLOW

D. C. Jesperson, Acceleration of a CFD Code with a GPU, NAS Technical Report NAS-09-003, NASA Ames Research Center, 2009.

- Structured RANS, Implicit SSOR
- Only considered 1 CPU core, 1 GPU, NO MPI
- Replaced 64 bit SSOR with a 32 bit Jacobi method on GPU
- Achieved 2.5-3X speedup (SP) using GTX 8800 and Tesla C1060 over a single CPU core

CFD on GPU Clusters

E. H. Phillips, Y. Zhang, R. L. Davis, J. D. Owens, Rapid aerodynamic performance prediction on a cluster of graphics processing units, in: Proceedings of the 47th AIAA Aerospace Sciences Meeting, Orlando, Florida, USA.

- Compressible structured MBFLO code
- 88X over 1 core for a compressible block structured solver on 16 GPU cluster

D. Jacobsen, J. Thibault, I. Senocak, An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters, in: 48th AIAA Aerospace Sciences Meeting, Orlando, Florida, USA.

- Incompressible, Jacobi iterative solver for projection step
- 130X speedup over 8 cpu cores on a 128 Tesla C1060 cluster

Finite Elements on GPU clusters

D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, S. Turek, Exploring weak scalability for FEM calculations on a GPU enhanced cluster, Parallel Computing 33 (2007) 685-699.

D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, S. Turek, Using GPUs to improve multigrid solver performance on a cluster, International Journal of Computational Science and Engineering 4 (2008) 36-55.

D. Göddeke, S. H. Buijssen, H. Wobker, S. Turek, GPU acceleration of an unmodified parallel finite element Navier-Stokes solver, in: High Performance Computing and Simulation 2009, Leipzig, Germany.

Outline

FUN3D Background

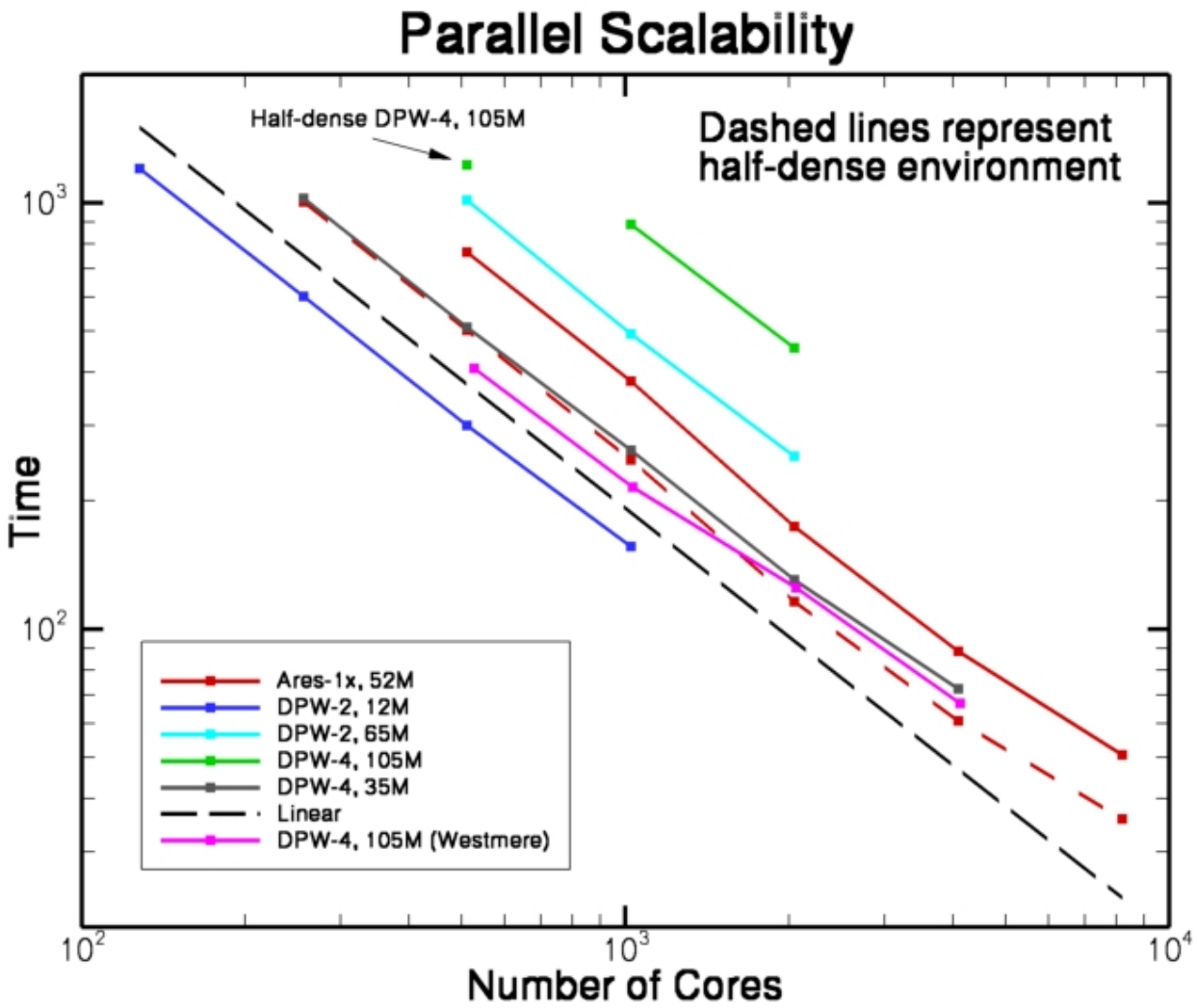
- Unstructured Navier-Stokes, all flow regimes, adjoint, design optimization
- Used by Boeing, Lockheed, DoD and SpaceX, among others
- Started out as an unstructured fluids research code in the late 1980's
- Has evolved into a production level status code
- http://fun3d.larc.nasa.gov

FUN3D Numerical Methods

- Node Based Finite Volume Scheme
- Colored Gauss-Seidel (GS) Point Implicit Solver
- Grid partitioning done with Metis and ParMetis [1]

[1] *G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Parallel Distrib. Comput. 48 (1998) 96-129.*

FUN3D Scaling



Accelerator Model w/ Distributed GPU Sharing

- Accelerator Model: Port a relatively small piece of CPU code containing a high level of data parallelism to GPU code
- We have inserted various CUDA C subroutines into the FUN3D `point_solve_5` (PS5) subroutine which replace the CPU computation when an appropriate GPU is available.
- These new routines are capable of sharing a GPU which receives distributed work in parallel from each thread.
- Our distributed GPU sharing model is a novel concept

Colored GS Iteration

The colored GS `point_solve` iterations generally resemble

for *color* = 1 to *maxcolor_sweeps* **do**
  // Do RHS 5x5 solve on color
  $b(:) = residual(:)$
  for $n = start$ to end **do**
    1. Compute off diagonal contributions
    for $j = istart$ to $iend$ **do**
      $b_i = b_i - \Sigma_{i \neq j} A_{i,j} * x_{icol}$ (SP)
    end for
    2. Compute 5x5 forward solve (DP)
    3. Compute 5x5 back solve (DP)
    4. Compute sum contribution, update RHS (MP)
  end for
  // End color, do MPI communication
  MPI xfer
end for

GPU Mapping

To map PS5 to the GPU, we simply let n parallel threads replace the sequential RHS solve via a kernel function

for *color* = 1 to *maxcolor_sweeps* **do**
  // Launch n/8+1 blocks of 8 threads
  `ps5_kernel <<<n/8+1,8>>> ();`
  // End color, do MPI communication
  MPI xfer
end for

We have replaced a sequential loop of n iterations with n parallel thread computations. n is in the 10,000's to 100,000's for our problems.

CUDA Subroutines

- `gs_gpu` : The first subroutine is for a straightforward 1 core - 1 GPU setup where the entire GS solve is done in CUDA.
- `gs_mpi0` : This case is identical to that above in terms of the GPU kernel, however, it assumes that an MPI communication is required at the end of each color.
- `gs_mpi1` : This subroutine is the first to consider the effects of GPU sharing, by attempting to reduce the kernel size and distribute a small portion of the workload to the CPU.
- `gs_mpi2` : The final subroutine was designed to execute with many threads sharing a single GPU. It shares a substantial amount of computation with the CPU, and also eliminates all double precision calculations from the GPU side.

Outline

Test Problem 1

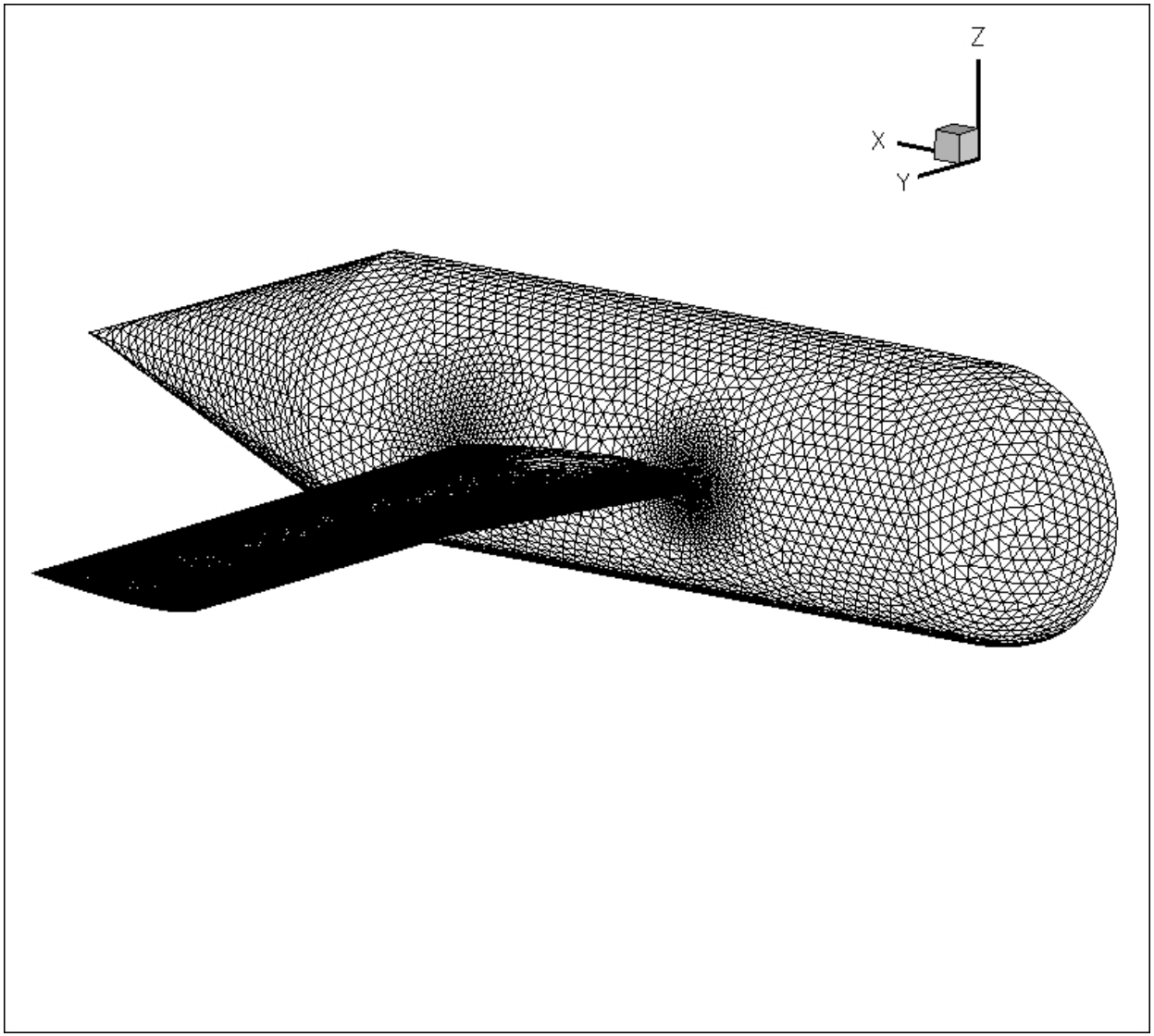
Simple wing-body geometry originating from the LaRC vgrid/tetruss training. It is a pure tetrahedral grid, with triangular boundary faces.

- 339206 Nodes
- 1995247 Cells
- 37834 Boundary Faces

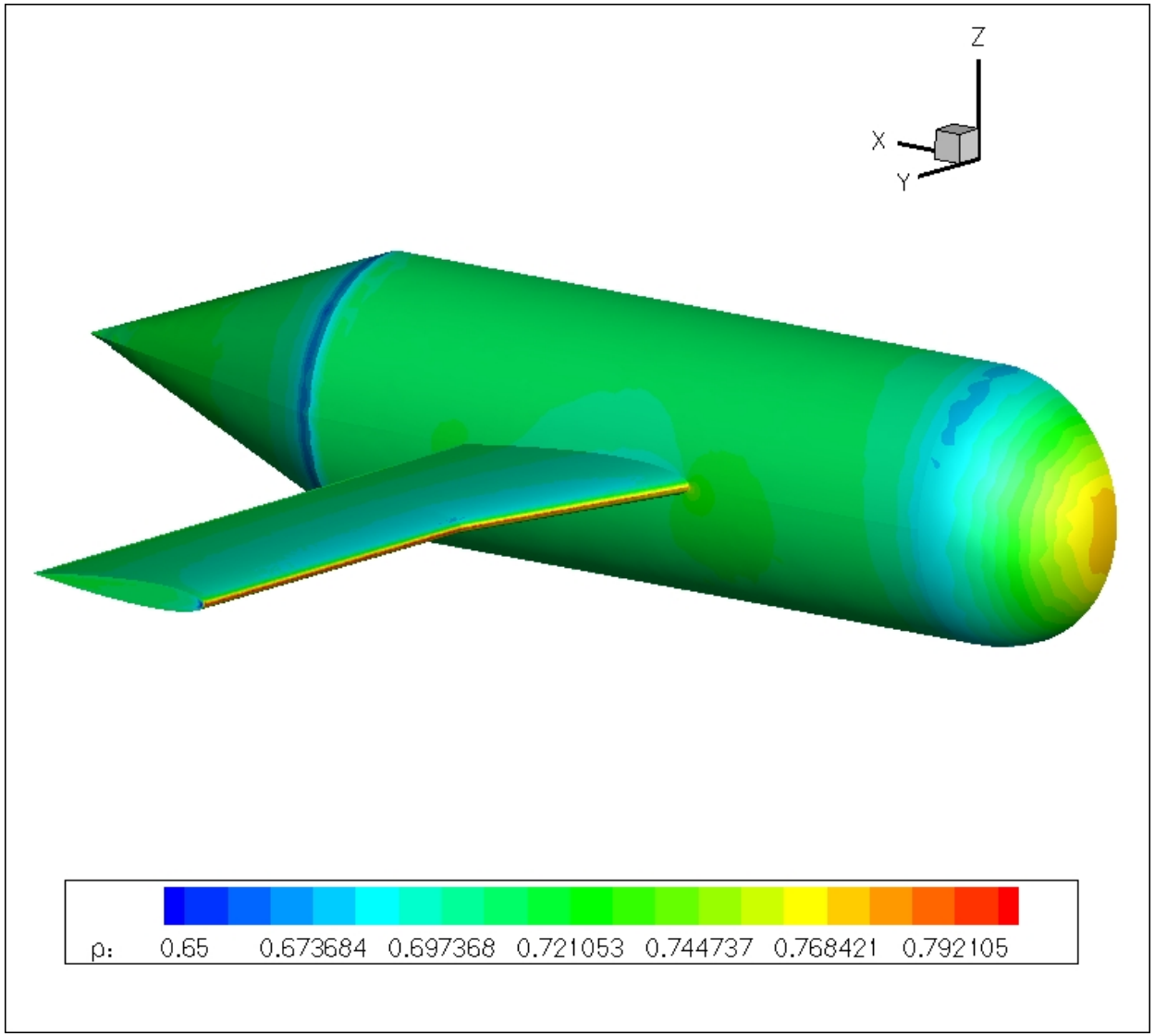
Node Partitioning:

Proc #	1 Core	2 Cores	4 Cores	8 Cores
0	339206	168161	85339	42824
1		171045	84626	40608
2			84861	42828
3			84380	42622
4				42555
5				42525
6				42726
7				42518

Test Problem 1 - Grid



Test Problem 1 - GPU Pres Solution



Test Problems 2 and 3

These test problems are used strictly for their node sizes in order to test PS5 run times, limited information will be given.

- Test Problem 2: DLR-F6, 650,000 Nodes
- Test Problem 3: Wing Leading-Edge, 900,000 Nodes

Node partitioning distributions are comparable to problem 1

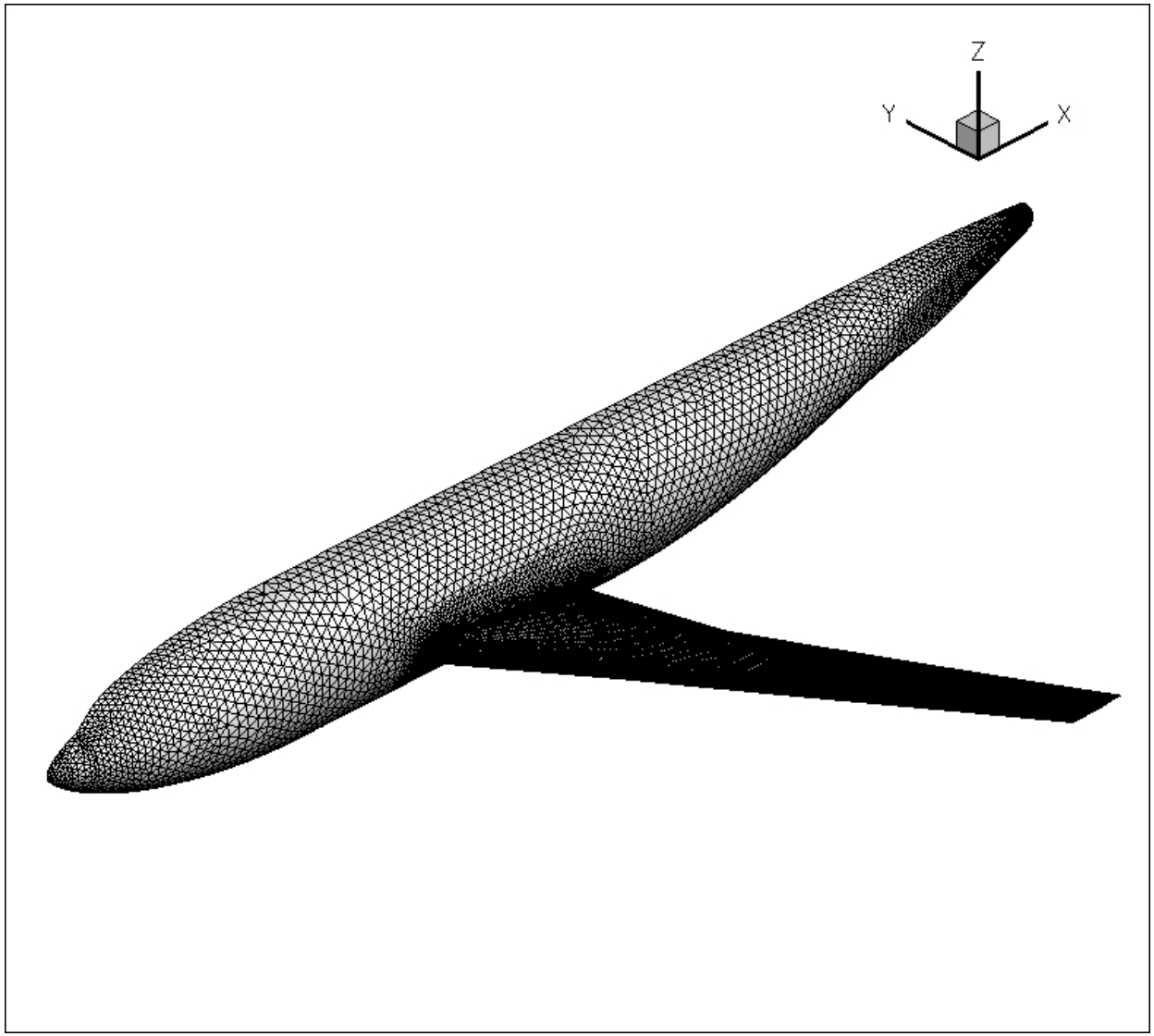
Test Problem 4

DLR-F6 Wing-Body Configuration from the second international AIAA Drag Prediction Workshop (DPW-II)

- 1,121,301 Nodes
- 6,558,758 Cells
- 674,338 Viscous Nodes
- 3,826,019 Viscous Cells

E. Lee-Rausch, N. Frink, D. Mavriplis, R. Rausch, W. Milholen, Transonic drag prediction on a DLR-F6 transport configuration using unstructured grid solvers, Computers & Fluids 38 (2009) 511-532.

Test Problem 4 - Grid



Test Machines

GTX 480 WS: Intel Xeon 5080 Workstation, NVIDIA GTX 480

- CPU: 2 x 2 core @ 3.73 GHz, 2 MB L2
- GPU: 1 x 480 core @ 1.4 GHz, 1.6 GB Global Memory

GTX 470 BC: AMD Athlon II Beowulf Cluster, NVIDIA GTX 470

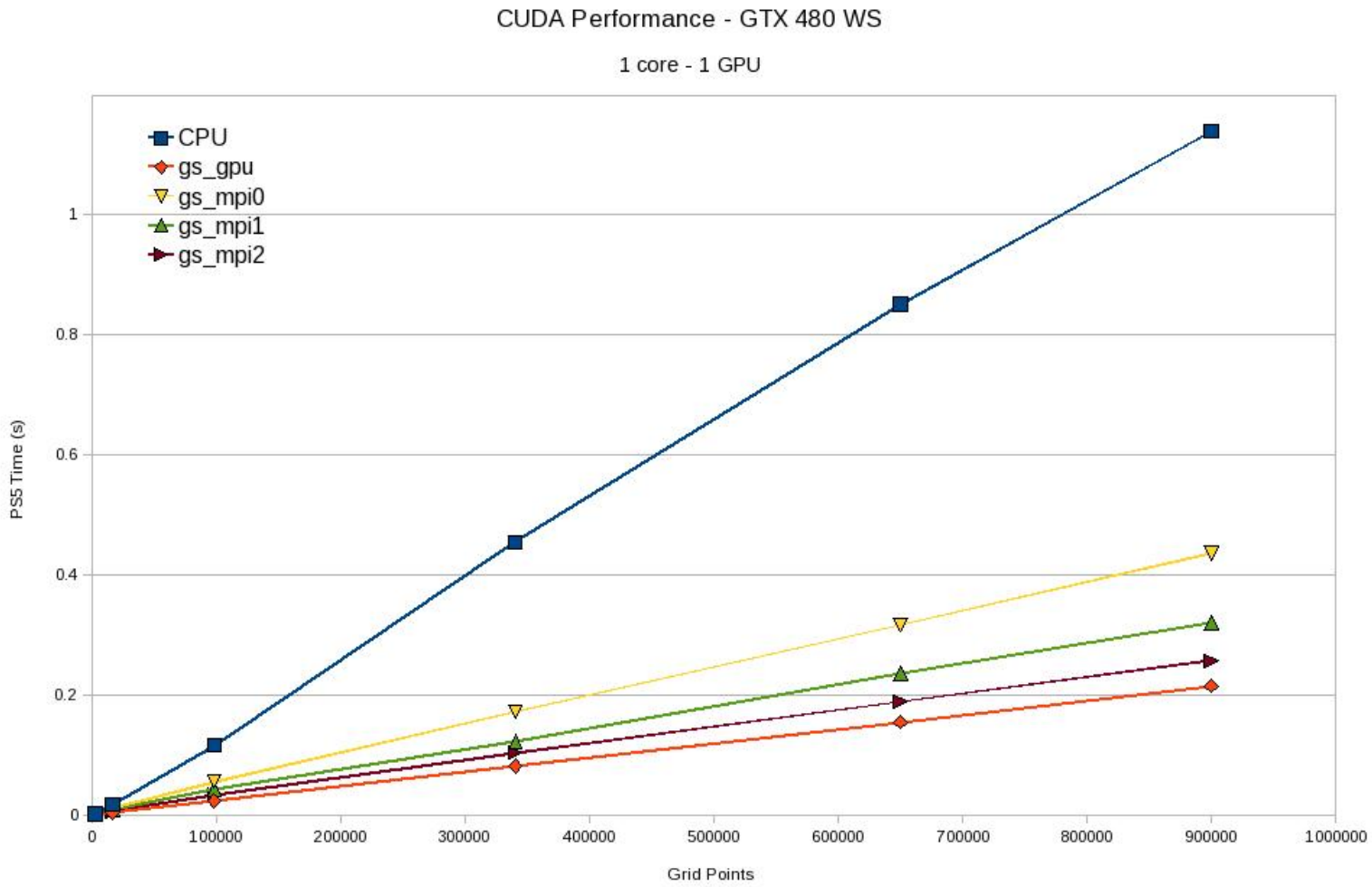
- CPU: 2 x 4 core @ 2.6 GHz, 2 MB L2
- GPU: 2 x 448 core @ 1.25 GHz, 1.26 GB Global Memory

Outline

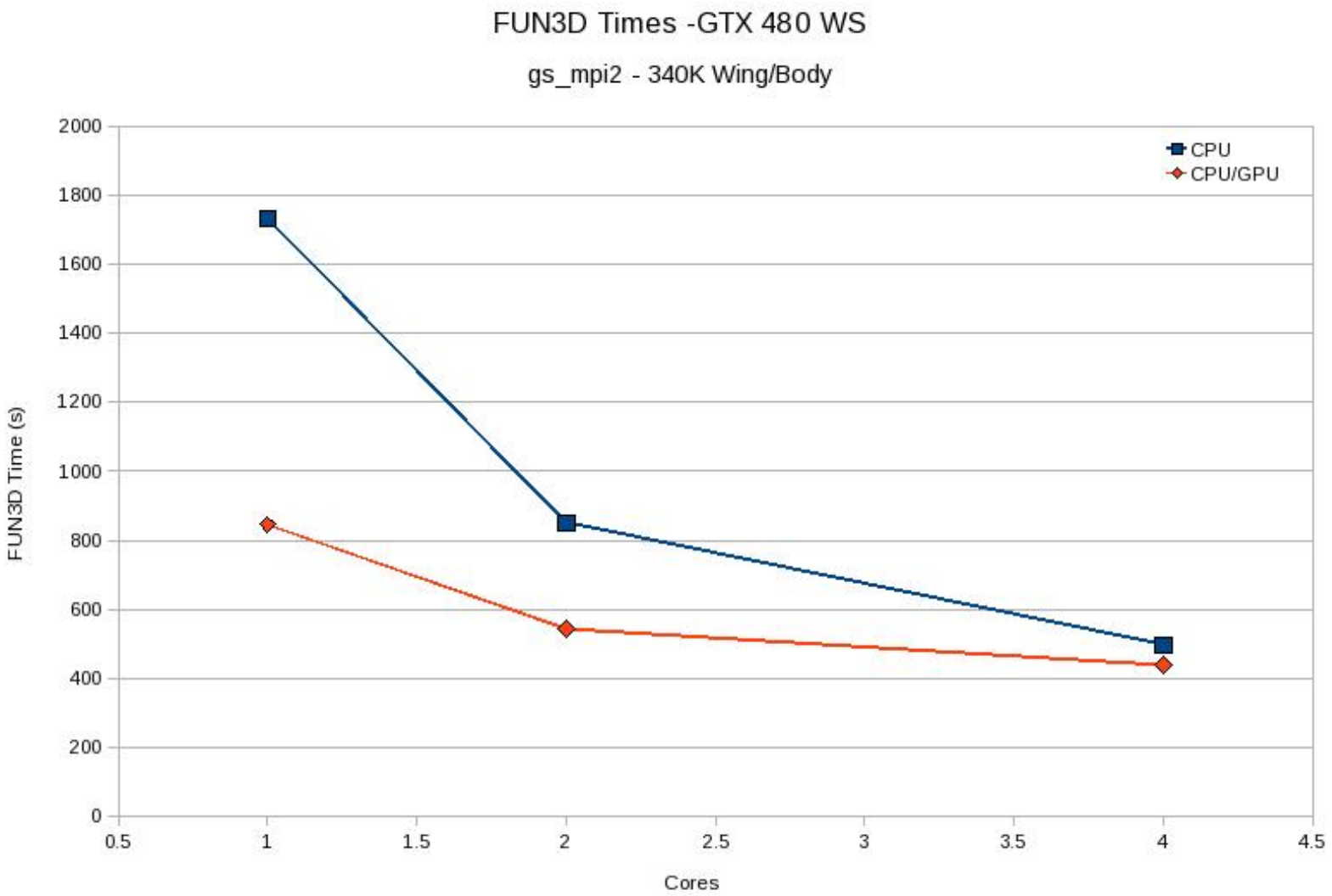
Considerations For The Following Timing Results

- Limited GPU memory size prevents testing with large problem sizes on a single GPU, must distribute across multiple
- For GTX 470 BC, scaling comparisons should be made in a node to node fashion, as cache space and GPU:CPU ratio are preserved
- There are GPU-CPU memory copy costs external to the CUDA C subroutines. These are included in the overall FUN3D time.
- GeForce series cards have detuned double precision capabilities, substantially slowing down `gs_gpu`, `mpi0`, and `mpi1`
- Both Fortran and CUDA compiled with -O2 optimizations

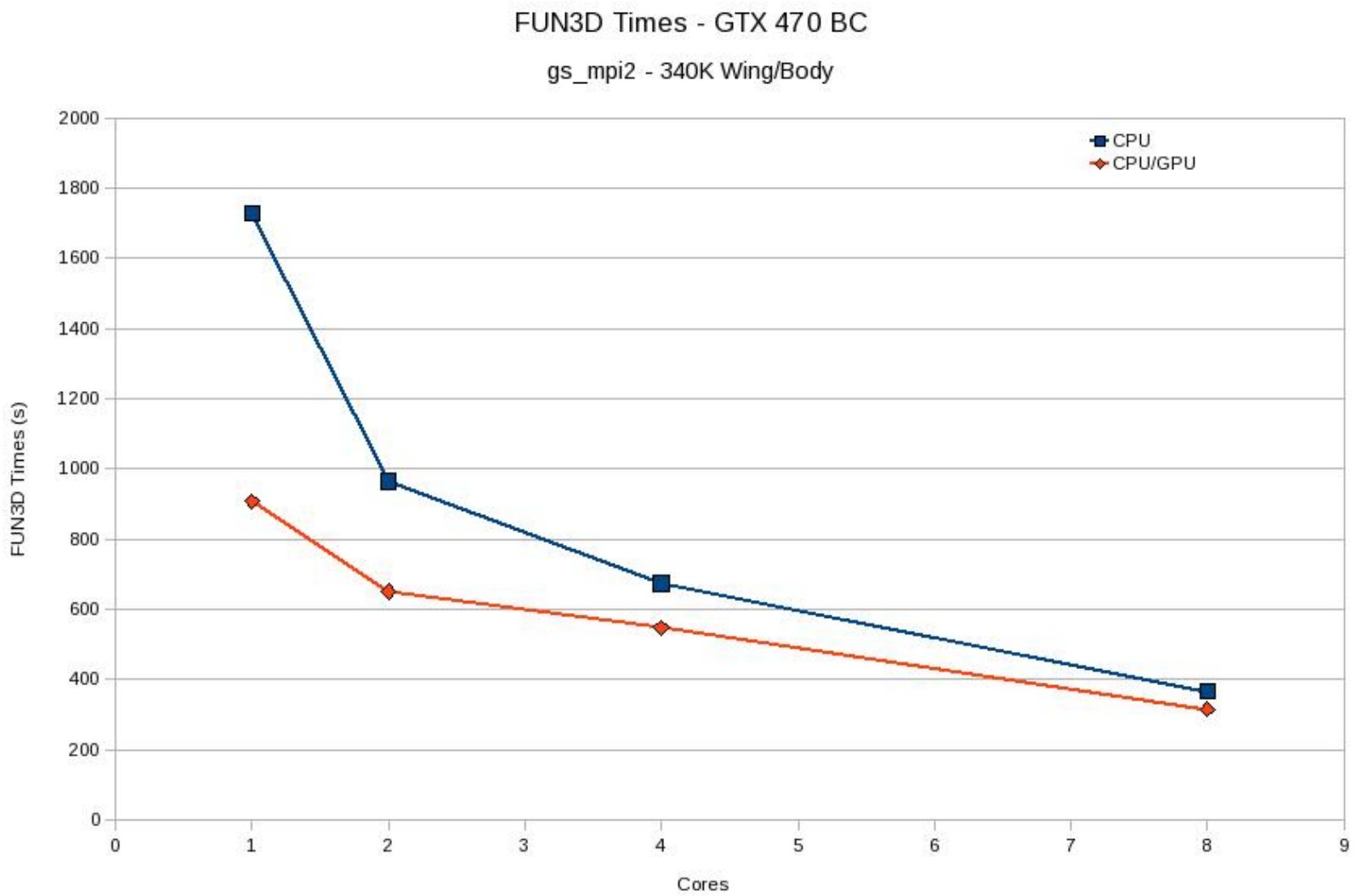
FUN3D Times



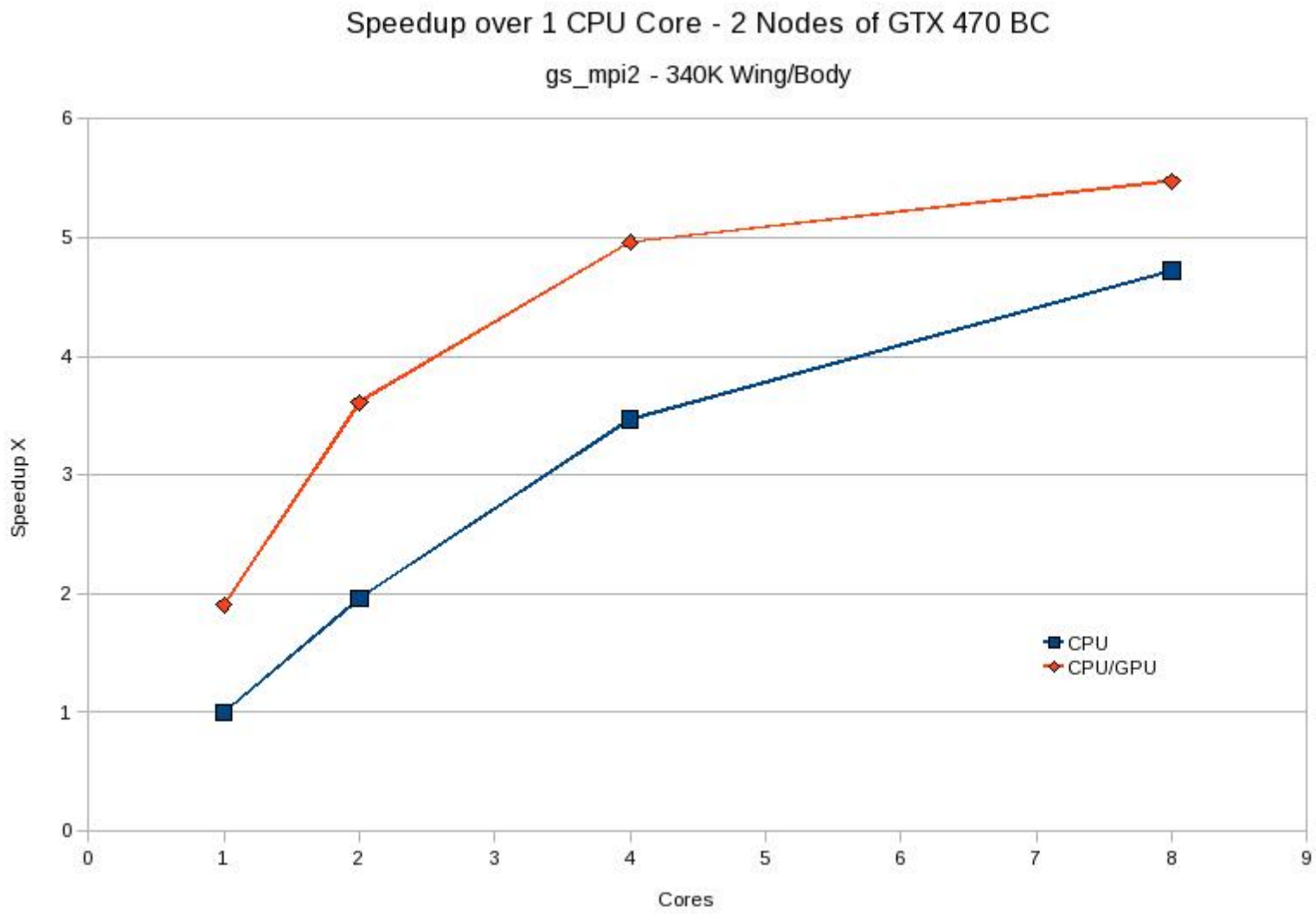
FUN3D Times



FUN3D Times



FUN3D Speedup



Test Problem 1 - gs_gpu

GTX 480 WS

gs_gpu	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D	1	1732.8 s	794.1 s	2.182X
PS5	1	0.455 s	0.082 s	5.549X

GTX 470 BC

gs_gpu	CPU Cores	CPU Time	CPU/GPU Time	Speedup
FUN3D	1	1728.9 s	860.5 s	2.001X
PS5	1	0.437 s	0.092 s	4.750X

FUN3D = overal FUN3D wall clock time
PS5 = CPU time for 1 sweep of PS5 solver

Test Problem 1 - gs_mpi0

GTX 480 WS - `gs_mpi0`	Cores	CPU	CPU/GPU	Speedup
FUN3D	1	1732.8 s	1026.5 s	1.688X
PS5	1	0.455 s	0.170 s	2.676X
FUN3D	2	850.6 s	744.5 s	1.143X
PS5	2	0.216 s	0.164 s	1.317X
FUN3D	4	496.0 s	648.1 s	0.758X
PS5	4	0.125 s	0.167 s	0.727X

FUN3D = overal FUN3D wall clock time
PS5 = CPU time for 1 sweep of PS5 solver

Test Problem 1 - gs_mpi0 - GTX 470 BC

1 Node (1 GPU)

gs_mpi0	Cores	CPU	CPU/GPU	Speedup
FUN3D	1	1728.9 s	1137.1 s	1.568X
PS5	1	0.437 s	0.197 s	2.218X
FUN3D	2	963.9 s	882.15 s	1.093X
PS5	2	0.240 s	0.193 s	1.244X
FUN3D	4	674.3 s	786.5 s	0.857X
PS5	4	0.172 s	0.197 s	0.873X

2 Nodes (2 GPUs)

gs_mpi0	Cores	CPU	CPU/GPU	Speedup
FUN3D	2	888.1 s	596.2 s	1.490X
PS5	2	0.221 s	0.104 s	2.125X
FUN3D	4	498.8 s	482.9 s	1.033X
PS5	4	0.121 s	0.104 s	1.163X
FUN3D	8	366.0 s	441.6 s	0.829X
PS5	8	0.088 s	0.109 s	0.807X

Test Problem 1 - gs_mpi1

GTX 480 WS - `gs_mpi1`	Cores	CPU	CPU/GPU	Speedup
FUN3D	1	1732.8 s	912.0 s	1.900X
PS5	1	0.455 s	0.128 s	3.555X
FUN3D	2	850.6 s	626.4 s	1.358X
PS5	2	0.216 s	0.118 s	1.831X
FUN3D	4	496.0 s	528.1 s	0.939X
PS5	4	0.125 s	0.123 s	1.016X

FUN3D = overal FUN3D wall clock time
PS5 = CPU time for 1 sweep of PS5 solver

Test Problem 1 - gs_mpi1 - GTX 470 BC

1 Node (1 GPU)

gs_mpi1	Cores	CPU	CPU/GPU	Speedup
FUN3D	1	1728.9 s	1011.7 s	1.709X
PS5	1	0.437 s	0.149 s	2.933X
FUN3D	2	963.9 s	754.7 s	1.277X
PS5	2	0.240 s	0.142 s	1.690X
FUN3D	4	674.3 s	669.0 s	1.008X
PS5	4	0.172 s	0.148 s	1.162X

2 Nodes (2 GPUs)

gs_mpi1	Cores	CPU	CPU/GPU	Speedup
FUN3D	2	881.1 s	522.2 s	1.687X
PS5	2	0.221 s	0.080 s	2.763X
FUN3D	4	498.8 s	405.2 s	1.231X
PS5	4	0.121 s	0.079 s	1.532X
FUN3D	8	366.0 s	369.3 s	0.991X
PS5	8	0.088 s	0.082 s	1.073X

Test Problem 1 - gs_mpi2

GTX 480 WS - `gs_mpi2`	Cores	CPU	CPU/GPU	Speedup
FUN3D	1	1732.8 s	845.7 s	2.049X
PS5	1	0.455 s	0.104 s	4.375X
FUN3D	2	850.6 s	543.5 s	1.565X
PS5	2	0.216 s	0.087 s	2.483X
FUN3D	4	496.0 s	438.3 s	1.132X
PS5	4	0.125 s	0.092 s	1.359X

FUN3D = overal FUN3D wall clock time
PS5 = CPU time for 1 sweep of PS5 solver

Test Problem 1 - gs_mpi2 - GTX 470 BC

1 Node (1 GPU)

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
FUN3D	1	1728.9 s	908.1 s	1.904X
PS5	1	0.437 s	0.113 s	3.867X
FUN3D	2	963.9 s	651.4 s	1.480X
PS5	2	0.240 s	0.105 s	2.286X
FUN3D	4	674.3 s	548.3 s	1.230X
PS5	4	0.172 s	0.109 s	1.578X

2 Nodes (2 GPUs)

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
FUN3D	2	881.1 s	478.8 s	1.840X
PS5	2	0.221 s	0.061 s	3.623X
FUN3D	4	498.8 s	348.4 s	1.432X
PS5	4	0.121 s	0.059 s	2.051X
FUN3D	8	366.0 s	315.8 s	1.159X
PS5	8	0.088 s	0.063 s	1.397X

Test Problem 2 - gs_gpu

GTX 480 WS:

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	1	0.851 s	0.155 s	5.490X

GTX 470 BC:

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	1	0.822 s	0.174 s	4.724X

PS5 = CPU time for 1 sweep of PS5 solver

Test Problem 2 - gs_mpi2

GTX 480 WS:

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	1	0.851 s	0.189 s	4.503X
PS5	2	0.402 s	0.152 s	2.645X
PS5	4	0.237 s	0.155 s	1.529X

PS5 = CPU time for 1 sweep of PS5 solver

Test Problem 2 - gs_mpi2 - GTX 470 BC

1 Node (1 GPU)

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	1	0.822 s	0.215 s	3.823X
PS5	2	0.449 s	0.198 s	2.268X
PS5	4	0.332 s	0.197 s	1.685X

2 Nodes (2 GPUs)

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	2	0.407 s	0.120 s	3.392X
PS5	4	0.223 s	0.112 s	1.991X
PS5	8	0.165 s	0.115 s	1.435X

PS5 = CPU time for 1 sweep of PS5 solver

Test Problem 3 - gs_mpi2

GTX 480 WS:

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	1	1.140 s	0.257 s	4.436X
PS5	2	0.558 s	0.213 s	2.620X

GTX 470 BC - 2 Nodes (2 GPUs)

gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	2	0.564 s	0.213 s	2.648X
PS5	4	0.316 s	0.179 s	1.765X
PS5	8	0.230 s	0.168 s	1.369X

* M1: GPU Memory is insufficient for sharing among 4 threads, M2: Problem must be split across two GPUs

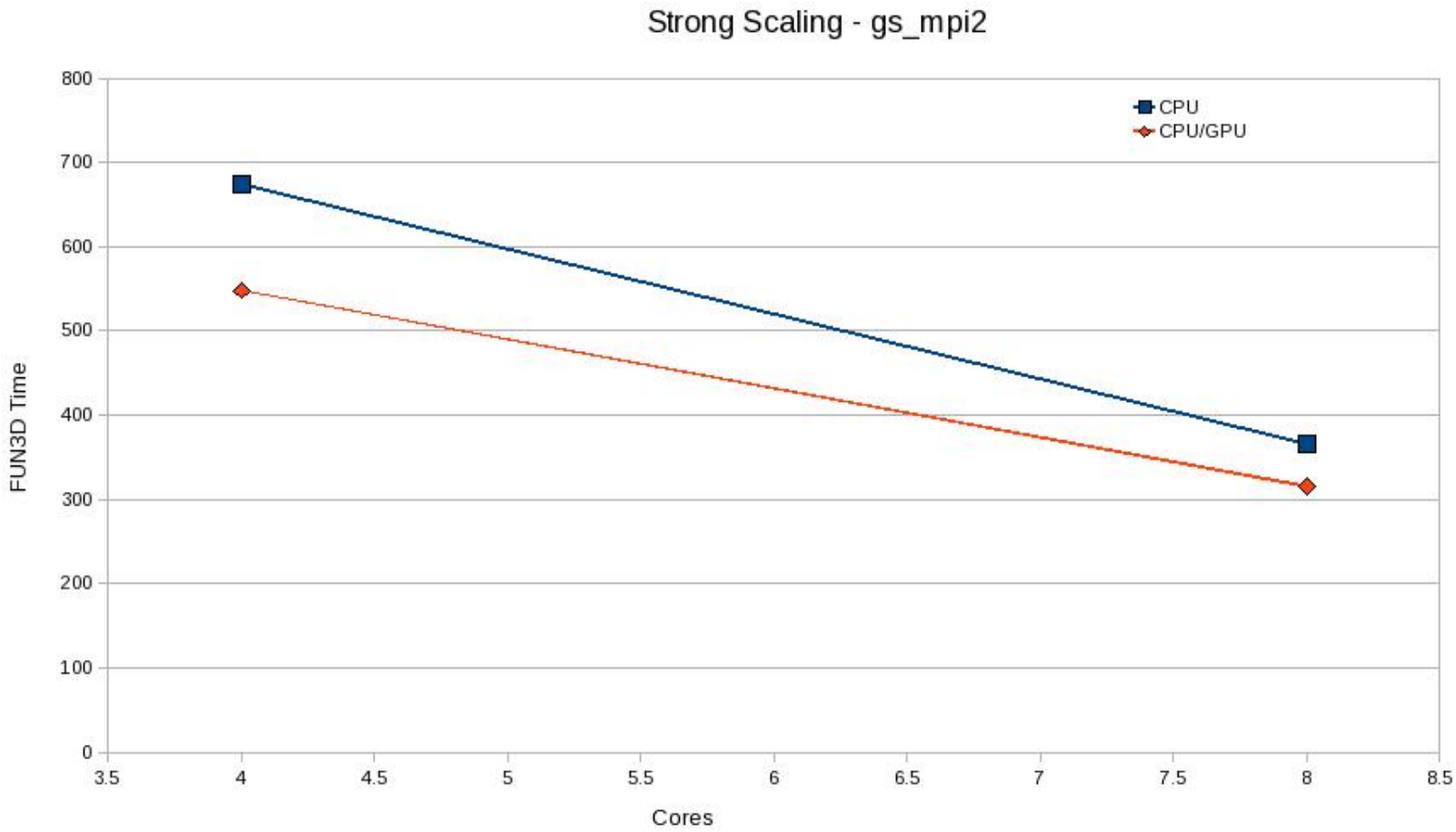
Test Problem 3 - gs_mpi2

GTX 480 WS - 2 Nodes:

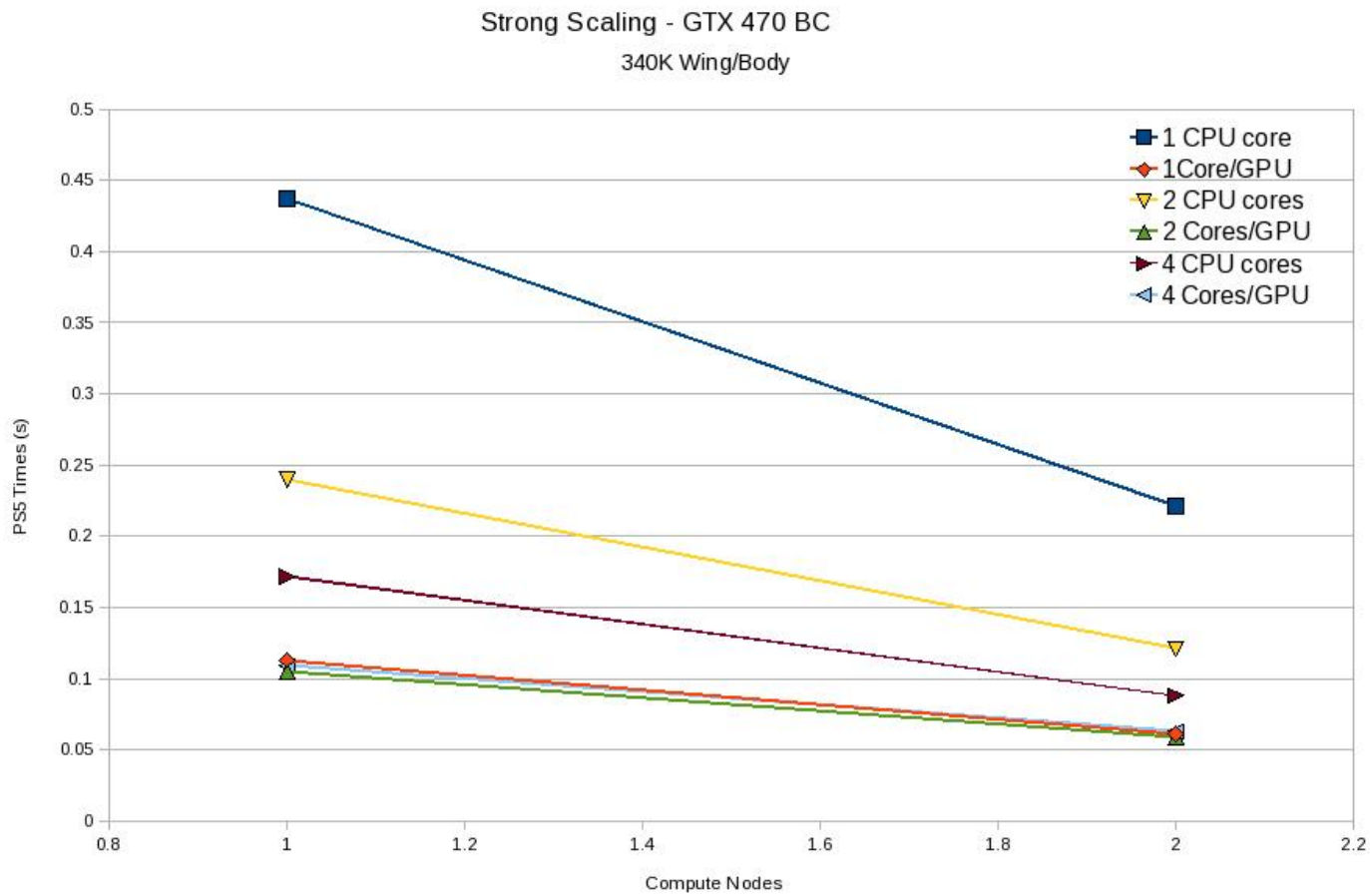
gs_mpi2	Cores	CPU	CPU/GPU	Speedup
PS5	4	0.411 s	0.195 s	2.108X
PS5	8	0.299 s	0.189 s	1.582X

PS5 = CPU time for 1 sweep of PS5 solver

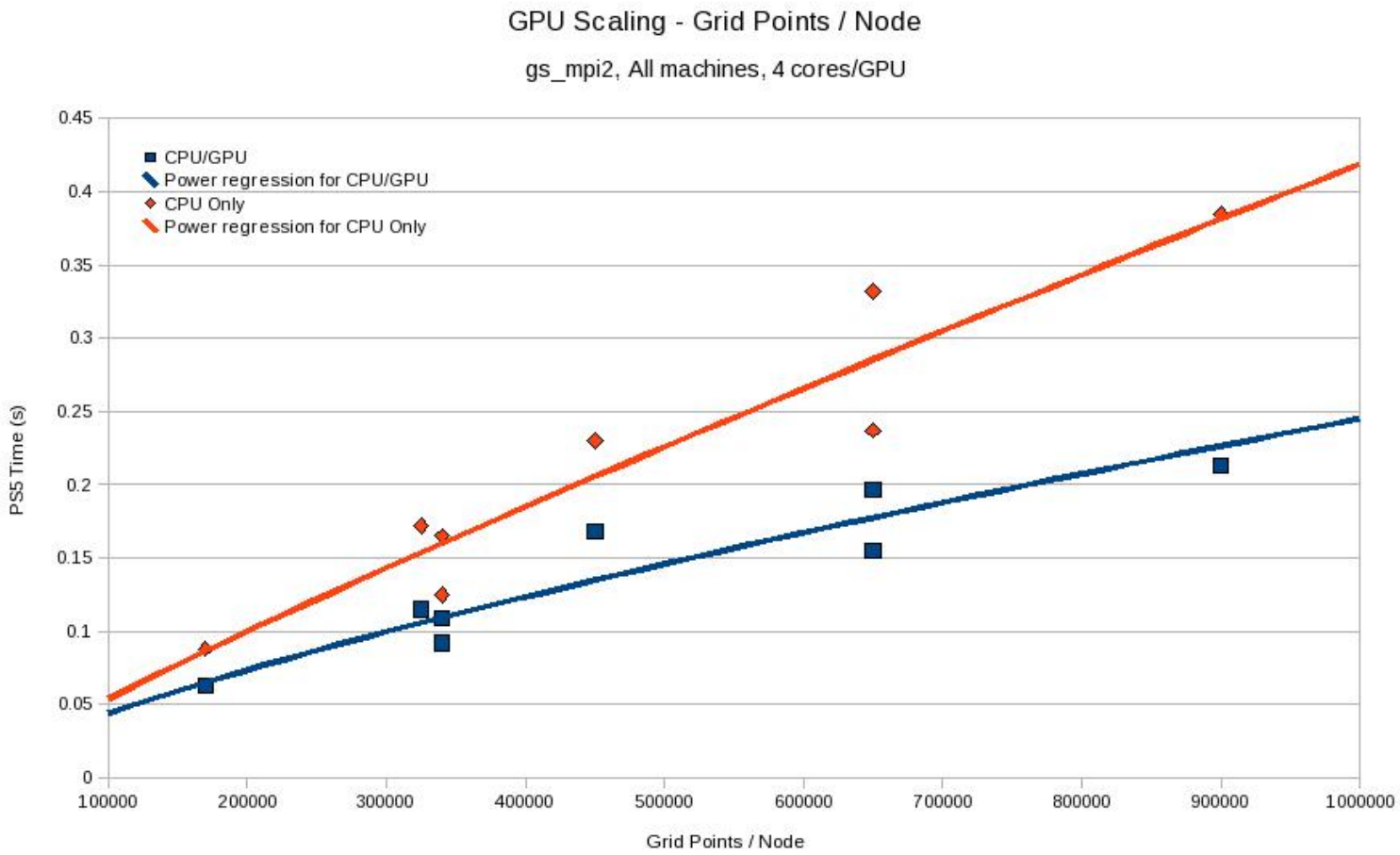
Strong Scaling - 340,000 Nodes



Strong Scaling - All



Pseudo-Weak Scaling - Points per node



Discussion

- FUN3D can be accelerated with up to 4 processor cores sharing a single low price GeForce series GPU
- We expect better acceleration with more cores per processor on a Tesla C2050 (much faster DP)
- For each additional core, there is a reduction in available memory of approximately 66 MB
- For problem 3, a GTX 480 runs out of memory when attempting to go from 2 to 4 cores per GPU with `gs_mpi2`
- Lack of a fast double precision GPU will require one to use the `gs_mpi2` routine in a parallel setting

Discussion

- The gs_gpu CUDA routine could benefit from moving the sweeping steps to inside the PS5 subroutine, resulting in only 1 data transfer per iteration. We would expect to see a speed increase of up to 7X for test problem 1, but it would only benefit the single core case.
- We have also tested problem 1 with gs_gpu on a GT 240 card on the headnode of machine 2, which could only achieve 1.27X speedup for PS5 in single precision
- With data from only two nodes of the GTX 470 BC machine, it appears that CPU/GPU times do not scale as well in the strong sense as strict CPU times, but it looks like weak scaling may be better.

Discussion

What stops PS5 from realizing 100X speedup? We can use tens of thousands of threads in a highly parallel fashion but...

- Communication requirement forces additional CPU/GPU memory transfers between colors and PS5 iterations - We need much faster transfer speeds, either from improved PCI or some new technology.
- Unable to use fast shared memory - We will likely never overcome this due to the unstructured nature of this code, and relatively small shared memory space.
- Need more cache space - GPU is working with 768 KB L2 total, CPU has 512-1024 KB per core. GPUs can quickly catch up here.

Outline

Numbers Summary

- PS5 speedup of 4.5X to 5.5X with 1 GeForce GTX 480 GPU per core
- Overall FUN3D code speedup of 2X
- Could be substantially higher with 20x20 block solves (higher order methods)

Conclusions

- We have shown that distributed GPU sharing can be used as an effective model when scaling on hybrid clusters
- The need to communicate across processors forces additional CPU-GPU data transfers, which severely limit performance gains
- GPU memory size can be a limiting factor in particular applications
- Larger L1 and L2 cache sizes in future generations will improve performance for unstructured codes

Future Generation GPUs

In the next generation of CPUs and GPUs, we may see much more substantial speedup for this code

- GPU core growth should outpace that of CPU cores. GPU chips are substantially larger than CPU chips and have more room to shrink.
- CPU cores will likely have lower clock speeds when they break out into many-cores (see AMD's 12 core Opteron)
- CPUs will likely maintain current cache/core ratios, GPUs could exceed theirs (next slide)
- And ... hopefully we will have faster data transfer technology (new PCI?) and/or global memory access (GDDR6?).

Future Generation GPUs

Each generation of GPUs has provided more cores, larger 'cache' sizes and faster memory access.

Arch.	Cores	L1	L2	Mem Acc.
G80	112	16 KB [1]	128 KB [2]	57.6 GB/s GDDR3
GT200	240	24 KB [1]	256 KB [2]	102 GB/s GDDR3
Fermi	448	48/16 KB [3]	768 KB	144 GB/s GDDR5

- G80 - Geforce 8800 GT
- GT200 - Tesla C1060
- Fermi - Tesla C2050
- [1] = shared memory, [2] = texture cache, [3] = configurable L1/Shared memory

Future Work - FUN3D

- Testing on a Tesla C2050 card
- Testing on a large hybrid cluster
- Porting the other solvers (PS4, PS6, etc.) to CUDA (easy)
- OpenCL porting (more difficult)
- Move Jacobian computation to GPU (much more difficult)
- Other routines, other accelerators (Cell Processors) ???

Questions?