Fluid Simulation with CUDA

Jonathan Cohen
NVIDIA Research

CFD Code Design Considerations

Legacy CFD codes in wide use

CFD community is cautiously exploring GPU computing

Major issue: How to handle legacy code?

- Accelerate vs. Rewrite?
- Is it worth it?

What about other approaches we may have completely missed? Rethink numerical methods?

Conceptual Map of CFD on CUDA

- Option 1: “Accelerate” existing code
- Option 2: Write new code from scratch
- Option 3: Rethink numerical methods

- Fermi results not available as of press time. [Notes in yellow indicate differences.]

Option 1: "Accelerator" Design

Case Study: FEAST from TU Dortmund

Finite Element Analysis and Solution Tools

Complex FE code for CFD and Structural Mechanics

Dominik Göddeke et al. accelerated using GPUs

Their approach: High level of abstraction

- *Minimally invasive co-processor integration*
- Identify and isolate "accelerable" parts of a computation
- Chunks must be large enough to amortise co-processor drawbacks

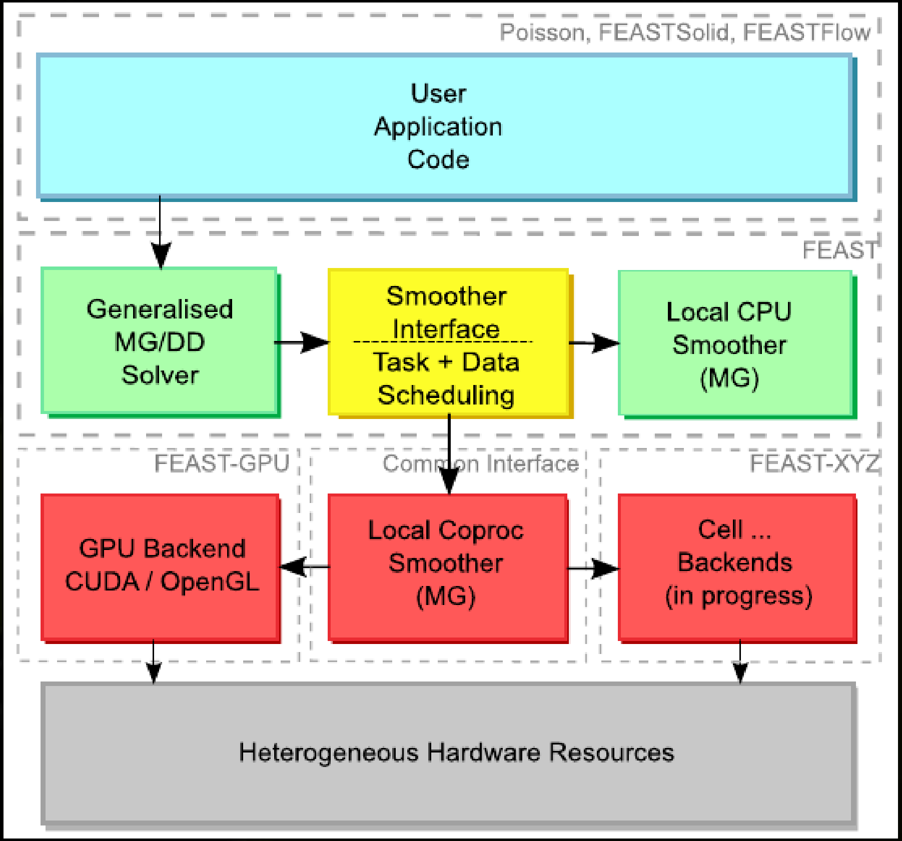
Portions of this slide courtesy Dominik Göddeke

FEAST-GPU Design Philosophy

FEAST-GPU Goal:

- **Integrate several co-processors into existing large-scale software package...**
- ***...without modifying application code***

- **NOT mapping single application to GPU / GPU Cluster**



Balance acceleration potential and acceleration effort

Portions of this slide courtesy Dominik Göddeke

FEAST-GPU Integration Results

Opteron 2214, 4 nodes

GeForce 8800 GTX

CUDA backend

18.8 M DOF

$$\begin{pmatrix} A_{11} & A_{12} & B_1 \\ A_{21} & A_{22} & B_2 \\ B_1^T & B_2^T & C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ p \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ g \end{pmatrix}$$

Accel. fraction R_{acc}:	75%
Local speedup S_{local}:	11.5x
Theoretical limit S_{max}:	4x
Global speedup S_{total}:	3.8x

```
fixed point iteration
solving linearised subproblems with
  global BiCGStab (reduce initial residual by 1 digit)
   Block-Schurcomplement preconditioner
    1) approx. solve for velocities with
       global MG (V 1+0), additively smoothed by
          for all $\Omega_i$: solve for $\mathbf{u}_1$ with
            local MG

          for all $\Omega_i$: solve for $\mathbf{u}_2$ with
            local MG
    2) update RHS: $\mathbf{d}_3 = -\mathbf{d}_3 + \mathbf{B}^T(\mathbf{c}_1, \mathbf{c}_2)^T$
    3) scale $\mathbf{c}_3 = (\mathbf{M}_p^L)^{-1}\mathbf{d}_3$
```

Portions of this slide courtesy Dominik Göddeke

Option 2: Rewrite

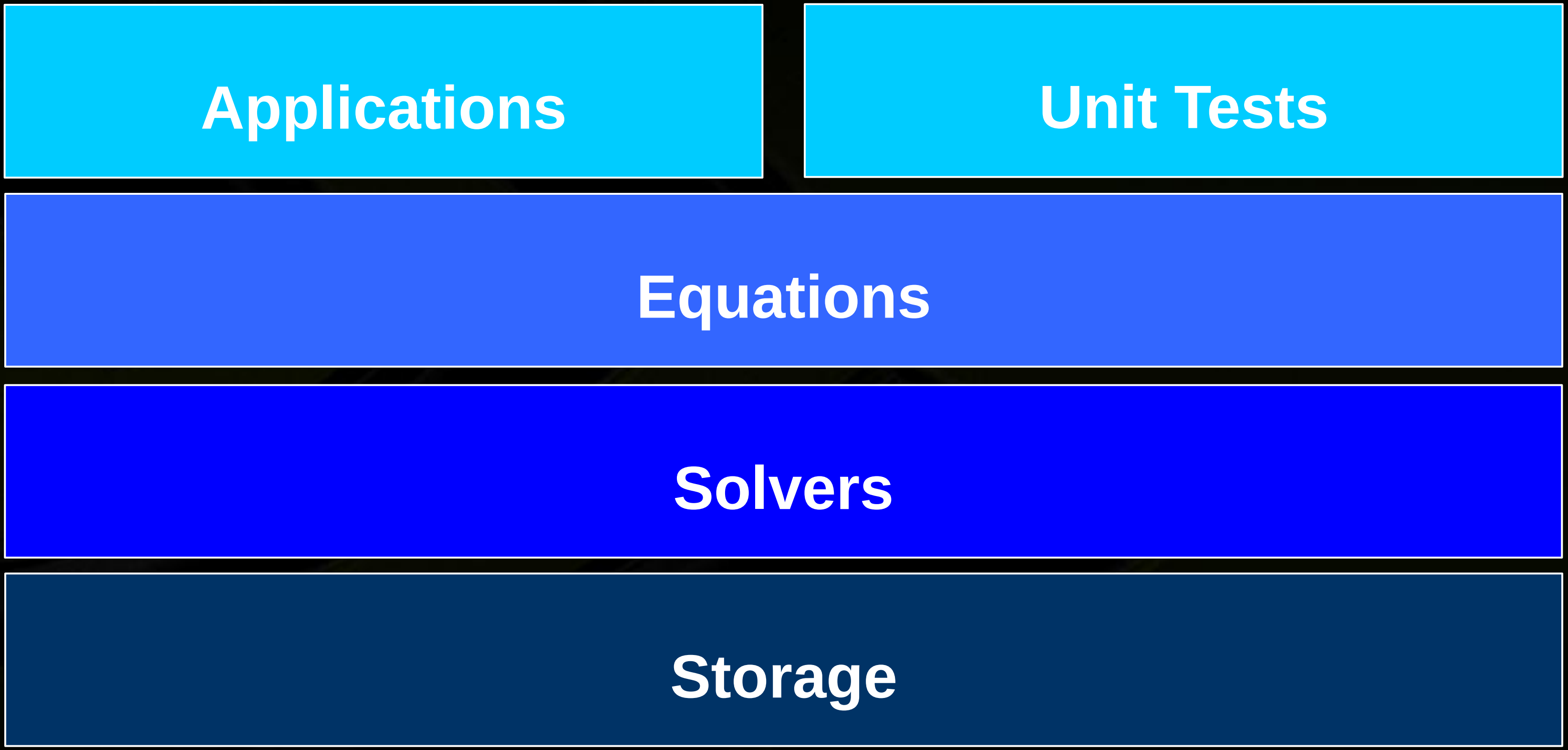
- If you were to attempt a rewrite:
  - What is a good overall design?
  - What global optimizations are possible?
  - What total application speedup can you get?
  - How does rewrite compare to “accelerator” design?

- Does 10x improvement on bottlenecks translate into 10x improvement for entire system?

- Challenge: Need a “fair” benchmark for comparison

OpenCurrent

Open Source, written by Jonathan Cohen
http://code.google.com/p/opencurrent/



Global Optimizations

- No serial bottlenecks (except IO)
- No unnecessary PCI-express transfers
- Small problems run on CPU
- Use of on-chip caches [Less important with Fermi]
- 3D array layout for maximum coalescing
- Congruent padding

Review: Memory Coalescing

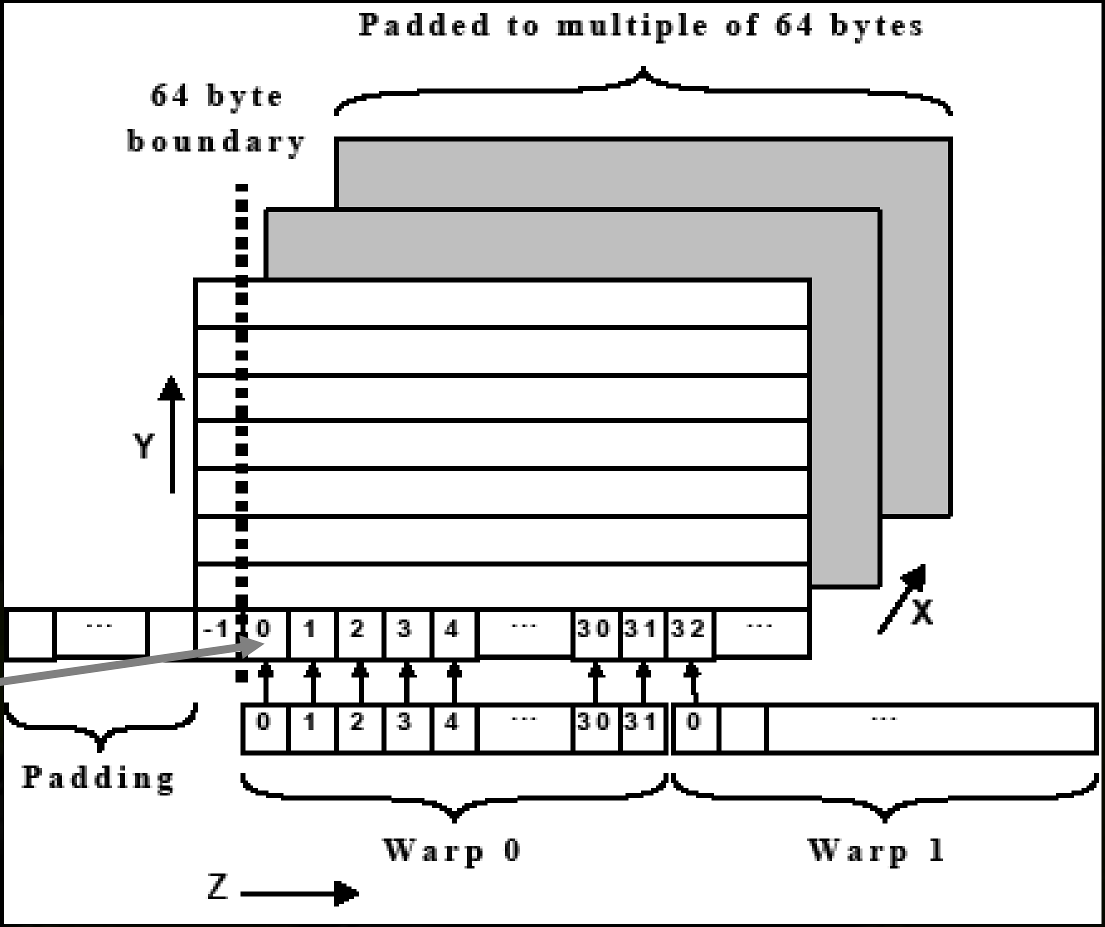
- 512-bit memory interface = 16 words per memclk
- Coalescer tries to batch simultaneous memory transactions into small number of 16 word batches
- Threads within half-warp loading bytes within

  ```
      [baseAddr, baseAddr + 16]
  ```

  coalesced to single wide load
- E.g. this will be optimal:

  ```
  idx = blockIdx.x * blockDim.x + threadIdx.x;
  float x = global_array[idx];
  ```
- Up to 16x performance boost for memory-bound applications

Layout of 3D Array



"Pre-padding" so z=0 starts at 16-word boundary

Pad each row to multiple of 16 words

Optimal coalescing when each thread accesses corresponding array element:

```
__global__ void Process3DArray(
  double *baseptr,
  int xstride,
  int ystride,
  ...
  int blocksInY)
{
  unsigned int blockIdxz = blockIdx.y / blocksInY;
  unsigned int blockIdxy = blockIdx.y % blocksInY;
  unsigned int i = blockIdxz *blockDim.z + threadIdx.z;
  unsigned int j = blockIdxy *blockDim.y + threadIdx.y;
  unsigned int k = blockIdx.x*blockDim.x + threadIdx.x;

  int idx = i*xstride + j*ystride + k;

  // This will be optimally coalesced:
  double T_ijk = baseptr[idx];

  ...
}
```

Index Translation – The Problem

- Grids may naturally have different dimensions

  E.g. for staggered u,v,w grids on 32 x 32 x 32 mesh,

  ```
  u = 33 x 32 x 32
  v = 32 x 33 x 32
  w = 32 x 32 x 33
  ```

- Translate from (i,j,k) to memory location:

  ```
  ptr = base_ptr + i * ny * nz + j * nz + k;
  ```

  - Since nx, ny, nz are different for u,v,w, must calculate & store `ptr` 3 times per element
  - Serial code could calculate offsets for previous cells

    1 thread/element => offsets won't work
  - Cost of extra per-thread calculation & state adds up with millions of threads

Optimization: Congruent Padding

Grid A is congruent to Grid B iff

For all i,j,k:

```
    (&A[i,j,k] - &A[0,0,0]) = (&B[i,j,k] - &B[0,0,0])
```

Pad nx, ny, nz to enforce congruency

Also pad for memory alignment, ghost cells, etc.

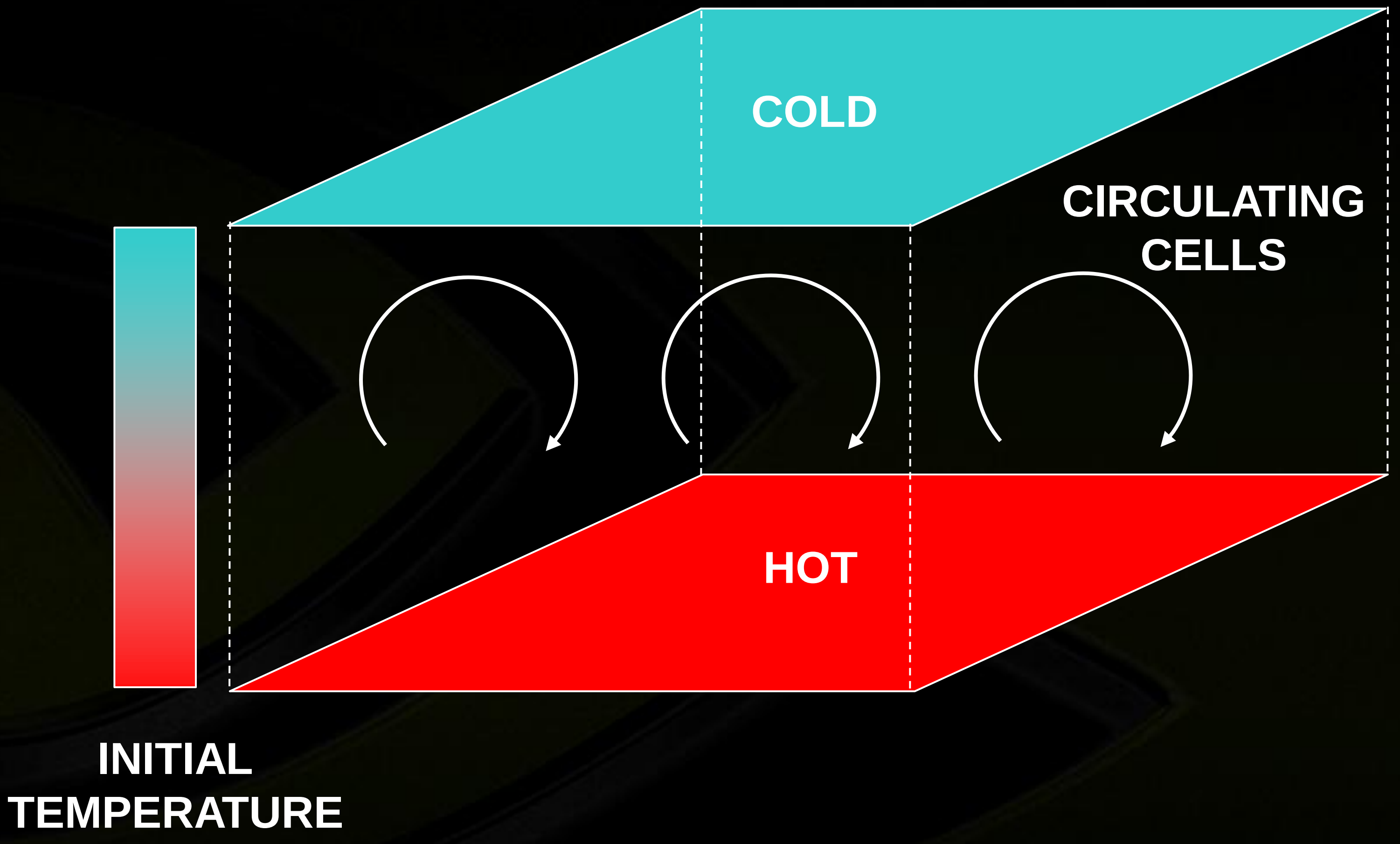
-1,2 18	0,2 19	1,2 20	2,2 21	Pad 22	Pad 23
-1,1 12	0,1 13	1,1 14	2,1 15	Pad 16	Pad 17
-1,0 6	0,0 7	1,0 8	2,0 9	Pad 10	Pad 11
-1,-1 0	0,-1 1	1,-1 2	2,-1 3	Pad 4	Pad 5

-1,2 18	0,2 19	1,2 20	2,2 21	3,2 22	4,2 23
-1,1 12	0,1 13	1,1 14	2,1 15	3,1 16	4,1 17
-1,0 6	0,0 7	1,0 8	2,0 9	3,0 10	4,0 11
-1,-1 0	0,-1 1	1,-1 2	2,-1 3	3,-1 4	4,-1 5

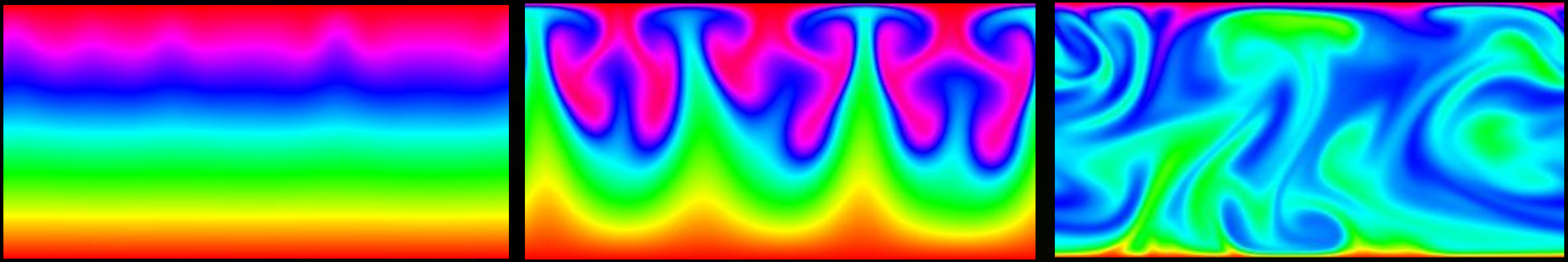
Congruent Padding - Results

- Passive Advection
  - 16 registers => 13 registers
  - 132 instructions => 116 instructions (12%)
- Restrict Residual (multigrid)
  - 14 registers => 13 registers
  - 75 instructions => 68 instructions (9%)
- Self-advection
  - 46 registers => 43 registers
  - 302 instructions => 264 instructions (13%)

Benchmark: Rayleigh-Bénard Convection



Rayleigh-Bénard Details



- Double precision, second order accurate
- 384 x 384 x 192 grid (max that fits in 4GB)
- Transition from stratified (left) to turbulent (right)
- Validated critical Rayleigh number against theory
- Validated / benchmarked more complex problems against published results & Fortran code

Benchmark Methodology

- Fortran code
  - Written by Jeroen Molemaker @ UCLA
  - 8 Threads (via MPI and OpenMP) on 8-core 2.5 GHz Xeon
  - Several oceanography pubs using this code, ~10 years of code optimizations. Code is small & fast.
- Per-step calculation time varies due to convergence rate of pressure solver
- Record time once # of v-cycles stabilizes
  - Point relaxer on GPU – 1 FMG + 7 v-cycles
  - Line relaxer on CPU – 1 FMG + 13 v-cycles

Benchmark Results

- CUDA (1 Tesla C1060) vs. Fortran (8-core 2.5 GHz Xeon)
- As “apples-to-apples” as possible ($ and manpower)
  - Equal price nodes (~$3k)
  - Skilled programmers in each paradigm

Resolution	CUDA time/step	Fortran time/step	Speedup
64 x 64 x 32	24 ms	47 ms	2.0x
128 x 128 x 64	79 ms	327 ms	4.1x
256 x 256 x 128	498 ms	4070 ms	8.2x
384 x 384 x 192	1616 ms	13670 ms	8.5x

Single Precision vs Double Precision

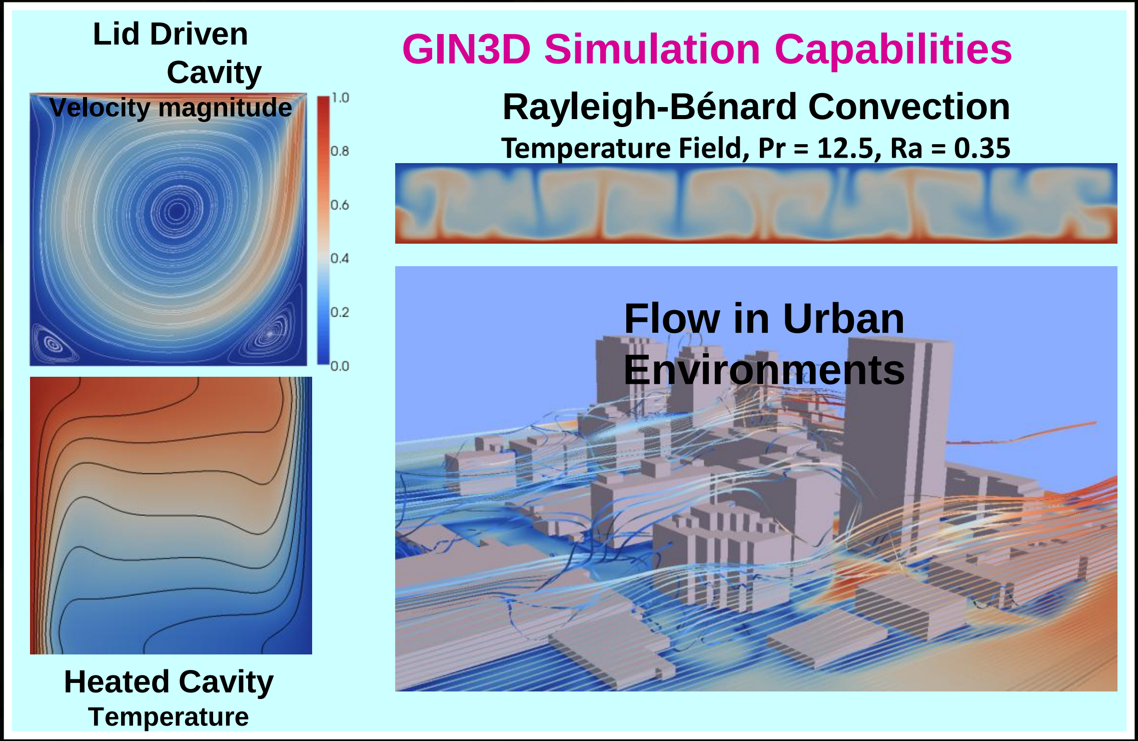
- Identical simulation, only difference is precision of buffers & math routines
- fp64 incurs penalty of 46% - 68% (far less than 12x)
- [Fermi fp64 results are different]

Resolution	fp64 time/step	fp32 time/step	Ratio
64^3	0.020912	0.014367	1.46x
128^3	0.077741	0.046394	1.68x
256^3	0.642961	0.387173	1.66x

Related Work: GIN3D

GIN3D: GPU accelerated **I**ncompressible **N**avier-Stokes **3D** solver for emerging massively parallel **multi-GPU clusters**

From Dana Jacobsen and İnanç Şenocak @ Boise State


Lid Driven Cavity
Velocity magnitude
1.0
0.8
0.6
0.4
0.2
0.0
Heated Cavity
Temperature
GIN3D Simulation Capabilities
Rayleigh-Bénard Convection
Temperature Field, Pr = 12.5, Ra = 0.35
Flow in Urban Environments

Portions of this slide courtesy Dana Jacobsen

GIN3D: Combining CUDA with MPI

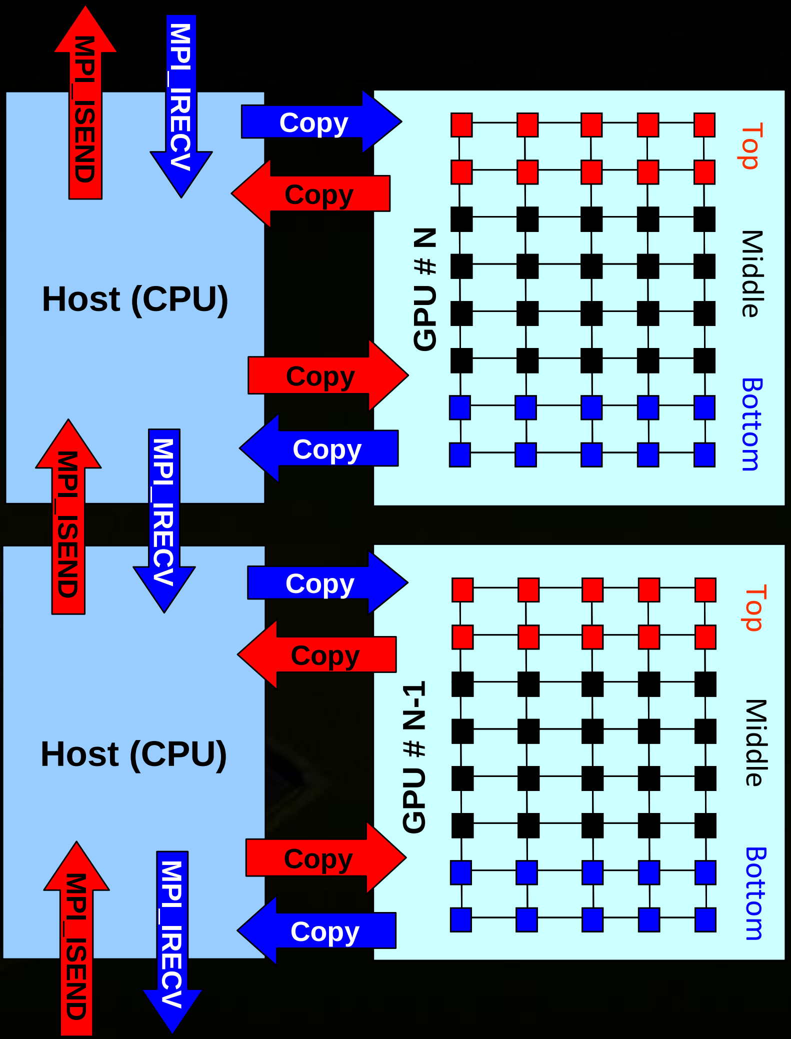
- NCSA Lincoln Cluster: 32 nodes, 64 GPUs
- Staggered uniform grid, 2nd order in time and space
- MPI exchange interleaved with CUDA kernels

```
for (t = 1 .. nTimesteps) {
   temperature<<<dimGrid,dimBlock>>>(u,v,w,phi)
   Copy_Exchange_Ghost_Cells(phi)
   momentum<<< dimGrid,dimBlock >>>(u,v,w,phi)
   Copy_Exchange_Ghost_Cells(u,v,w)
   divergence<<< dimGrid,dimBlock >>>(u,v,w,div)
   for (n =  1 .. nIterations) {
      pressure<<< dimGrid,dimBlock >>>(div,p)
      Copy_Exchange_Ghost_Cells(p)
   }
   velocity_correct<<< dimGrid,dimBlock >>> (u,v,w,p)
   Copy_Exchange_Ghost_Cells(u,v,w)
}
```

Portions of this slide courtesy Dana Jacobsen

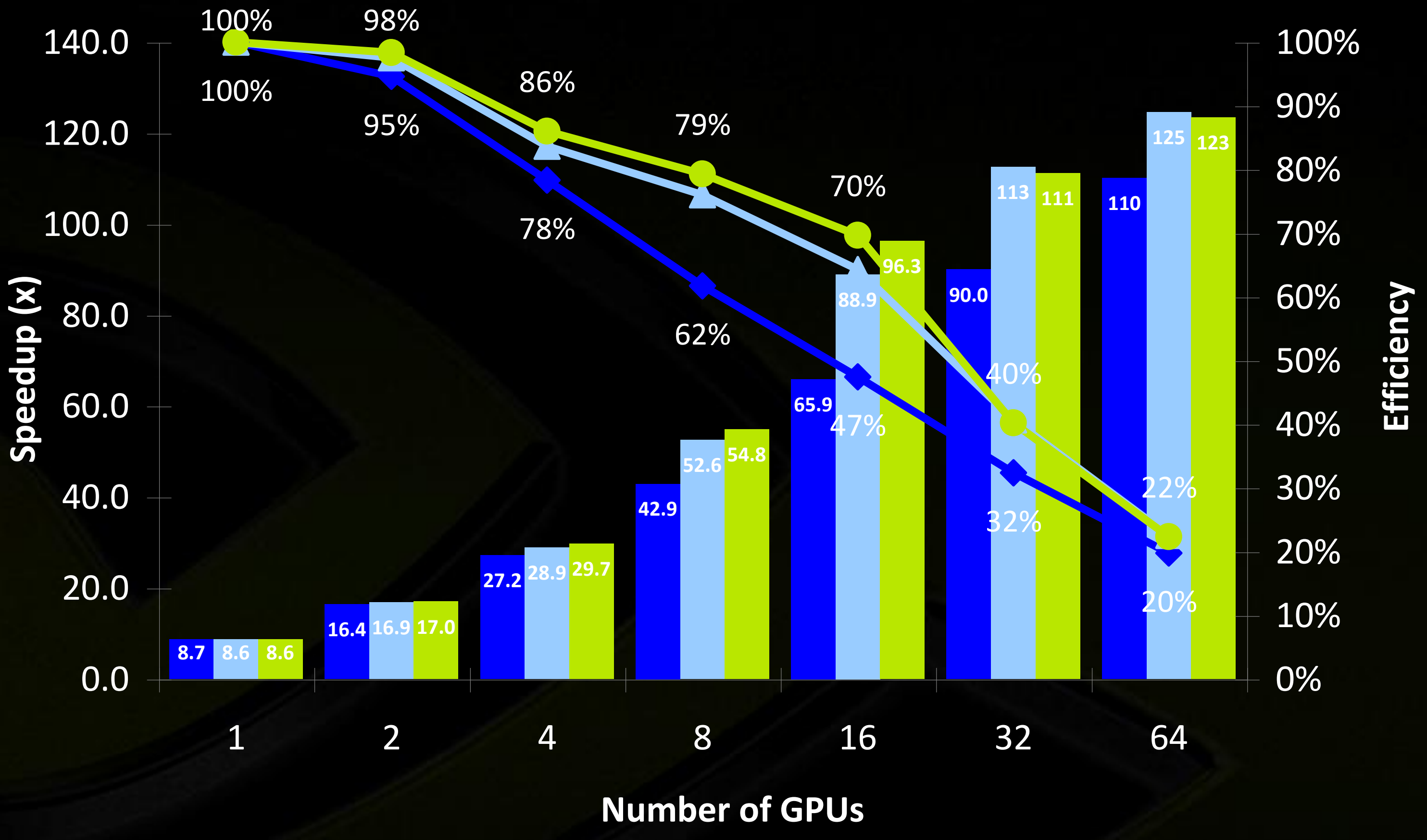
Overlap MPI with CUDA

No Overlap	Overlap	CUDA Streams
•Compute full domain •Copy and send top/bottom using MPI async calls •Finish MPI receives, copy top/bottom to device • No overlap of computation • Interleaves MPI and host/device copies	•Compute top •Compute bottom •Copy top/bottom to host •Send top/bottom using MPI async calls •Compute middle •Finish MPI receives •Copy top/bottom to device •ThreadSync to ensure middle is done • Overlaps computation with exchange	•Compute top •Compute bottom •ThreadSync to complete computations •Use CUDA Streams to simultaneously copy top/bottom from device while computing middle •Exchange top/bottom using MPI async calls •Async copy top/bottom to device •ThreadSync to ensure all work is done • Overlaps computation with copies and exchanges



Portions of this slide courtesy Dana Jacobsen

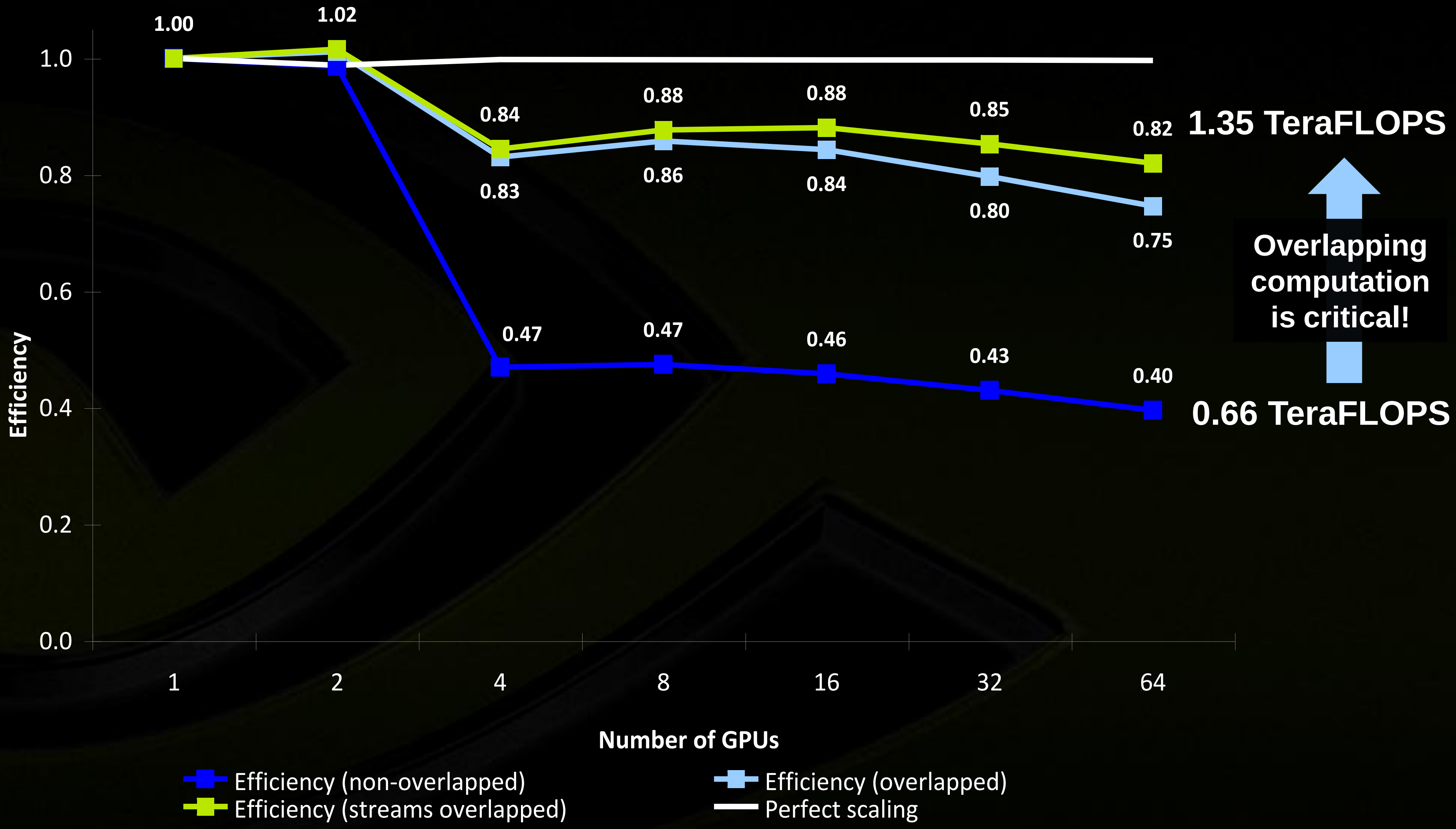
Strong Scaling: 1024x64x1024 Grid



GPU Speedup relative 8 cores (non-overlapped)
GPU Speedup relative 8 cores (overlapped)
GPU Speedup relative 8 cores (streams overlapped)
GPU Cluster effficiency (non-overlapped)
GPU Cluster effficiency (overlapped)
GPU Cluster effficiency (streams overlapped)

Portions of this slide courtesy Dana Jacobsen

Weak Scaling: 3.8 GB per GPU



Portions of this slide courtesy Dana Jacobsen

Option 3: Rethink the Numerics

- Numerical methods have co-evolved with programming languages, compilers, & architectures
  - Not coincidental that popular methods are easy to express in popular languages

- Maybe new (parallel) architectures require new numerics?

- Find methods that inherently map well to GPUs
  - Maybe we overlooked something in the past because it was impractical

Example: Nodal Discontinuous Galerkin Methods

Work from Andreas Klöckner et al @ Brown & Rice

Solve conservation laws over unstructured grids

$$u_t + \quad \cdot F(u) = 0$$

DG on GPUs: Why?

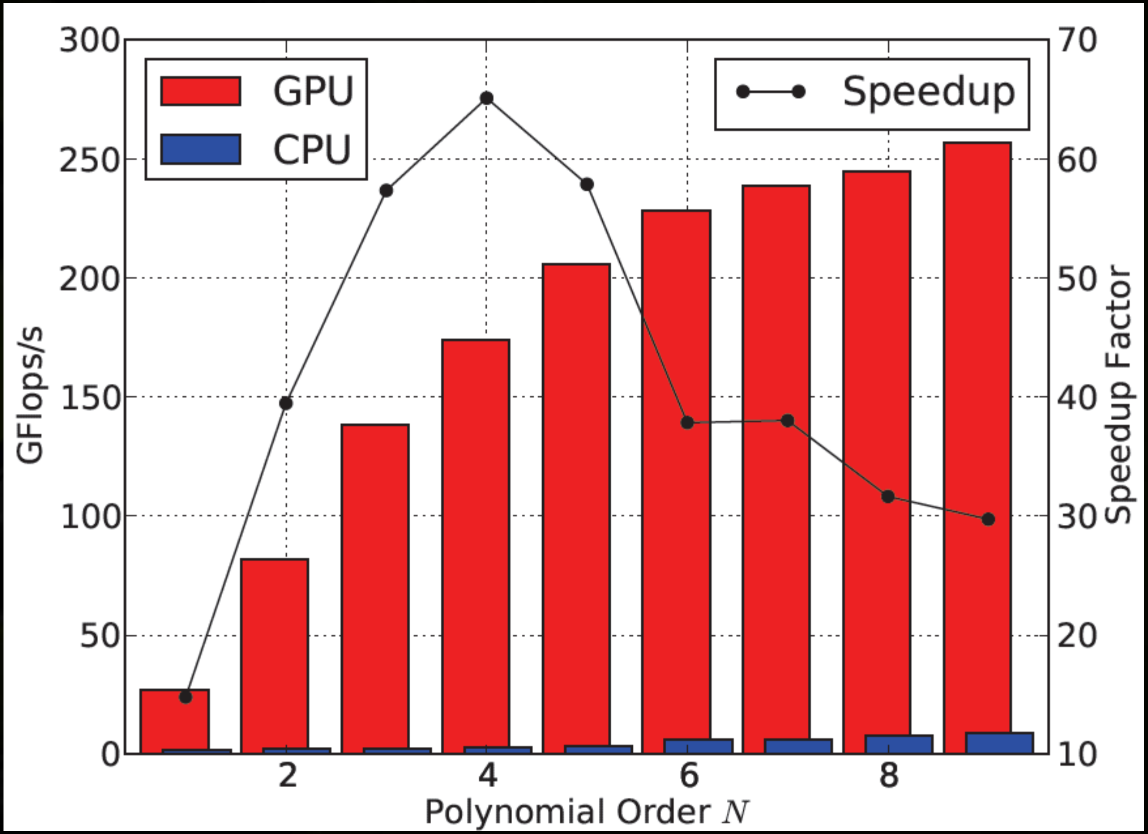
- GPUs have deep memory hierarchy
  - The majority of DG is local.
- Compute Bandwidth >> Memory Bandwidth
  - DG is arithmetically intense.
- GPUs favor dense data.
  - Local parts of the DG operator are dense.

Portions of this slide courtesy Andreas Klöckner

DG Results – Single GPU

Nvidia GTX280 vs. single core of Intel E8400

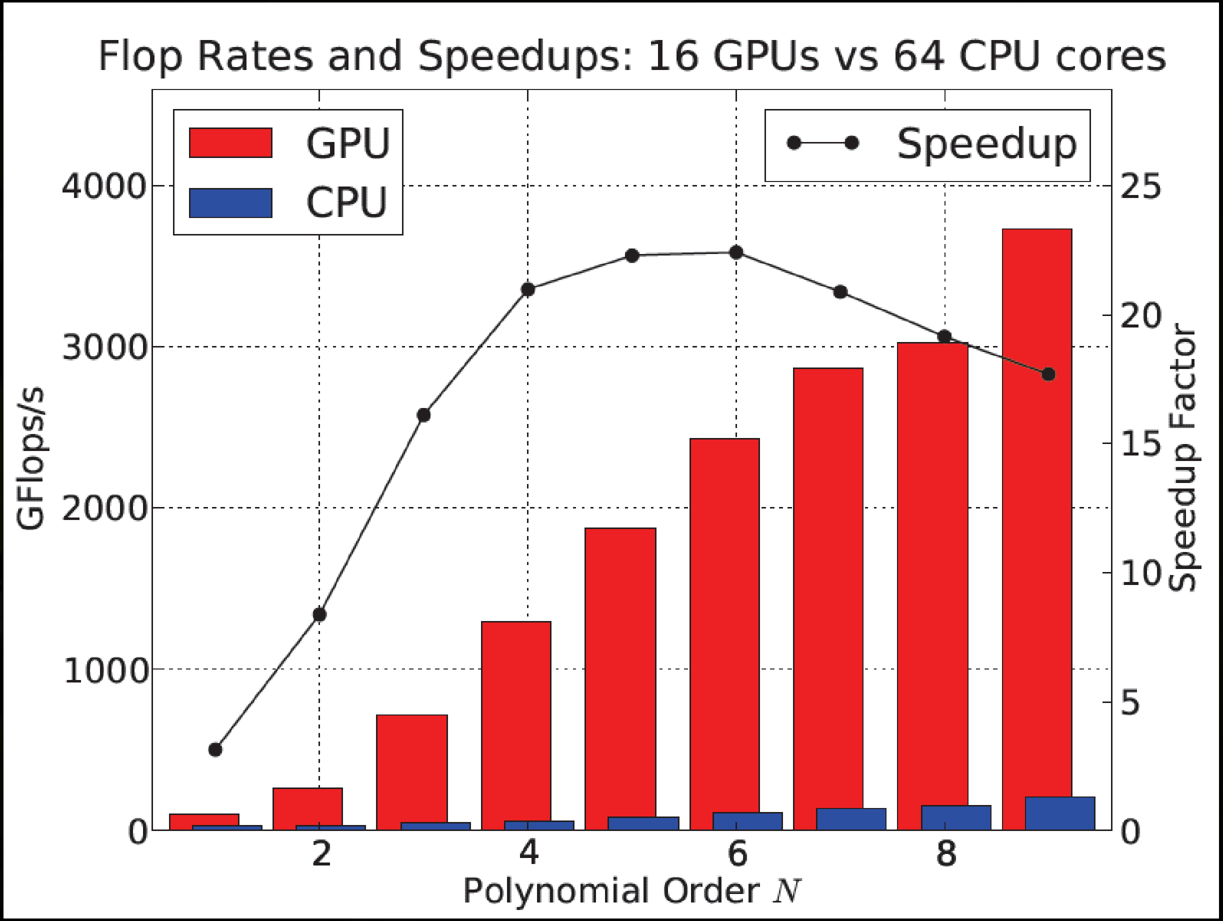
Maxwell's Equations



Portions of this slide courtesy Andreas Klöckner

DG Results – GPU Cluster

16 T10s vs. 64 Xeon E5472



Portions of this slide courtesy Andreas Klöckner

Conclusion: Three Options

- "Accelerate" Legacy Codes
  - Appropriate in some cost/benefit regime
- Rewrite From Scratch
  - Worth it for many applications
  - Double Precision performance is pretty good, getting better
- Rethink Numerical Methods
  - Potential for biggest performance advantage
  - Exciting time to be computational scientist!

Thanks

- Andreas Klöckner
- Dominik Göddeke
- Dana Jacobsen
- Jeroen Molemaker

Cited Work

- GIN3D

  Jacobsen, D. and Senocak, I. “Massively Parallel Incompressible Navier-Stokes Computations on the NCSA Lincoln Tesla Cluster,” GPU Technology Conference, 2009.

  Thibault, J. and Senocak, I. “CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows,” 47th AIAA Aerospace Sciences Meeting, paper no: AIAA-2009-758, 2009.

- Nodal DG Methods

  Andreas Klöckner, Tim Warburton, Jeff Bridge, Jan Hesthaven, “Nodal Discontinuous Galerkin Methods on Graphics Processors,” J. of Comp. Physics 2009.

- FEAST-GPU

  http://www.feast.uni-dortmund.de/publications.html

For more info

- OpenCurrent

  http://code.google.com/p/opencurrent/

  J. Cohen and M. Molemaker, “A Fast Double Precision CFD Code using CUDA,” Proceedings of ParCFD 2009.

- CFD applications in CUDA

  http://www.nvidia.com/object/computational_fluid_dynamics.html

- NVIDIA Research

  http://www.nvidia.com/research