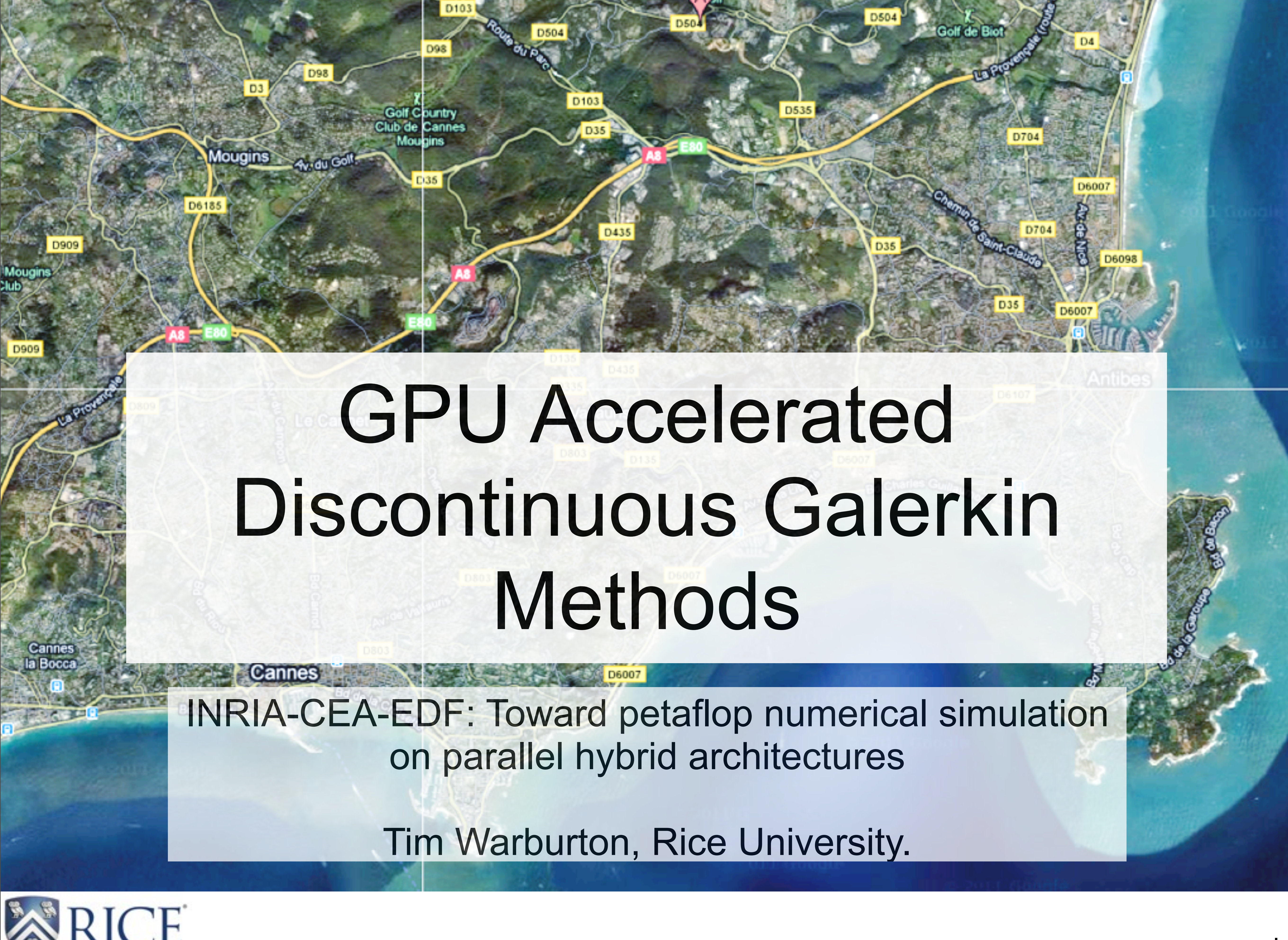


GPU Accelerated Discontinuous Galerkin Methods

INRIA-CEA-EDF: Toward petaflop numerical simulation on parallel hybrid architectures

Tim Warburton, Rice University.

RICE

With Thanks

- Jeffrey Bridge
- Carsten Burstedde
- Markus Clemens
- Omar Ghattas
- Nico Gödel
- Thomas Hagstrom
- Jan Hesthaven
- Andreas Klöckner
- Nigel Nunn
- Steffen Schomann
- Georg Stadler
- Lucas Wilcox

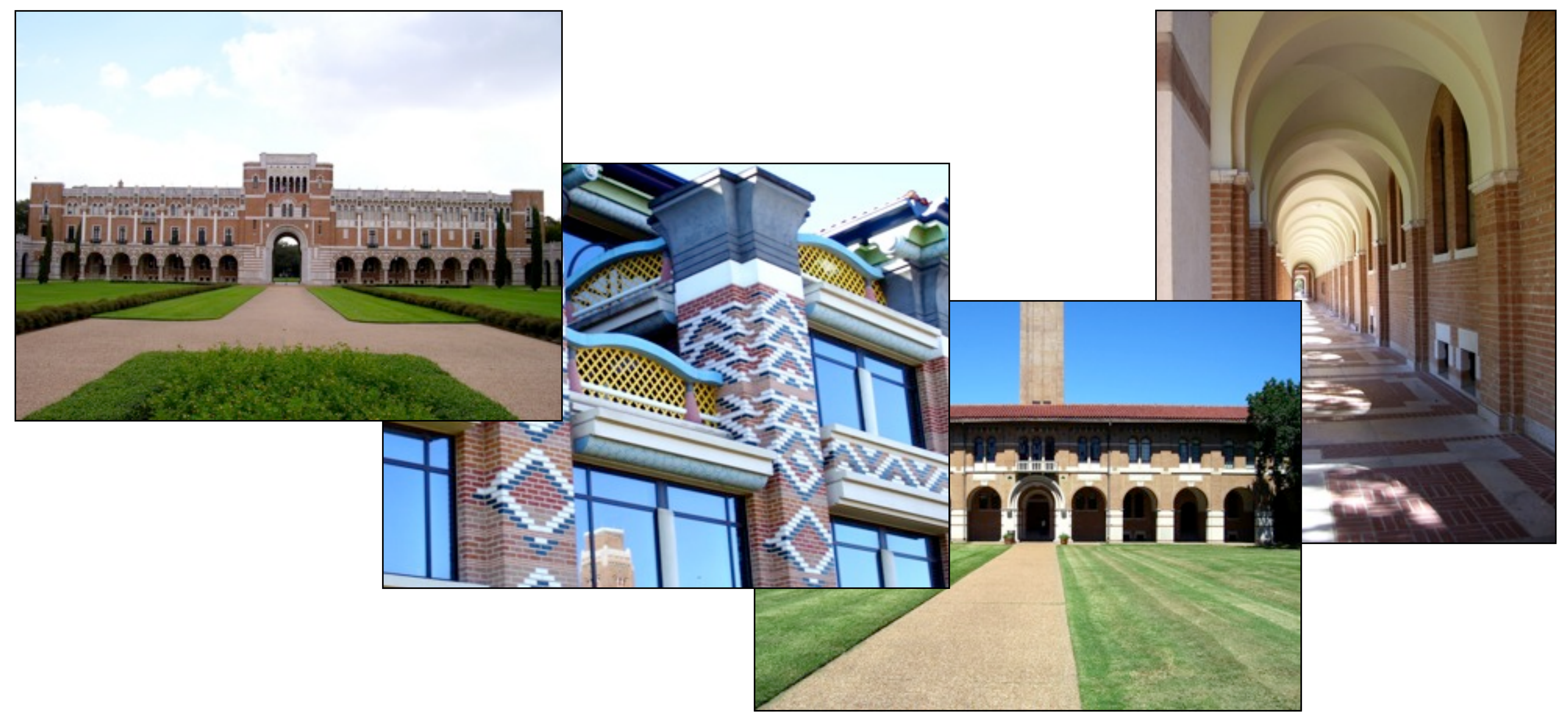
ARO
AFOSR
NSF

nvidia
AMD

Finite element meshes generated with CUBIT, Triangle, Netgen & distmesh.

Computational and Applied Math@Rice University

Rice University is a small private university in Houston, Texas.

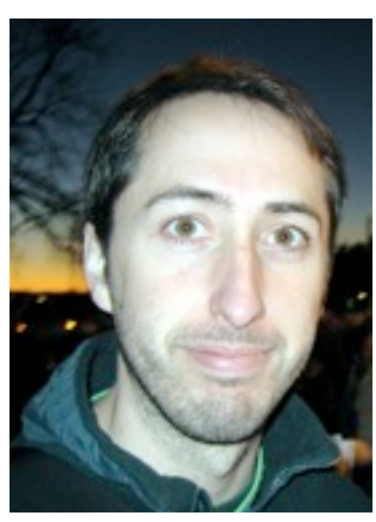


CAAM is a department with 12 faculty whose interests include:
PDE constrained optimization, inverse problems, numerical linear algebra, numerical PDEs, discrete optimization, neuroscience...

Current & Former *nudg* Team Members

Nigel Nunn
DG+GPU+
Frameworks

Dr Nico Gödel
DG+GPU+
Electromagnetics



Dr Lucas Wilcox
DG+GPU+
Multiphysics

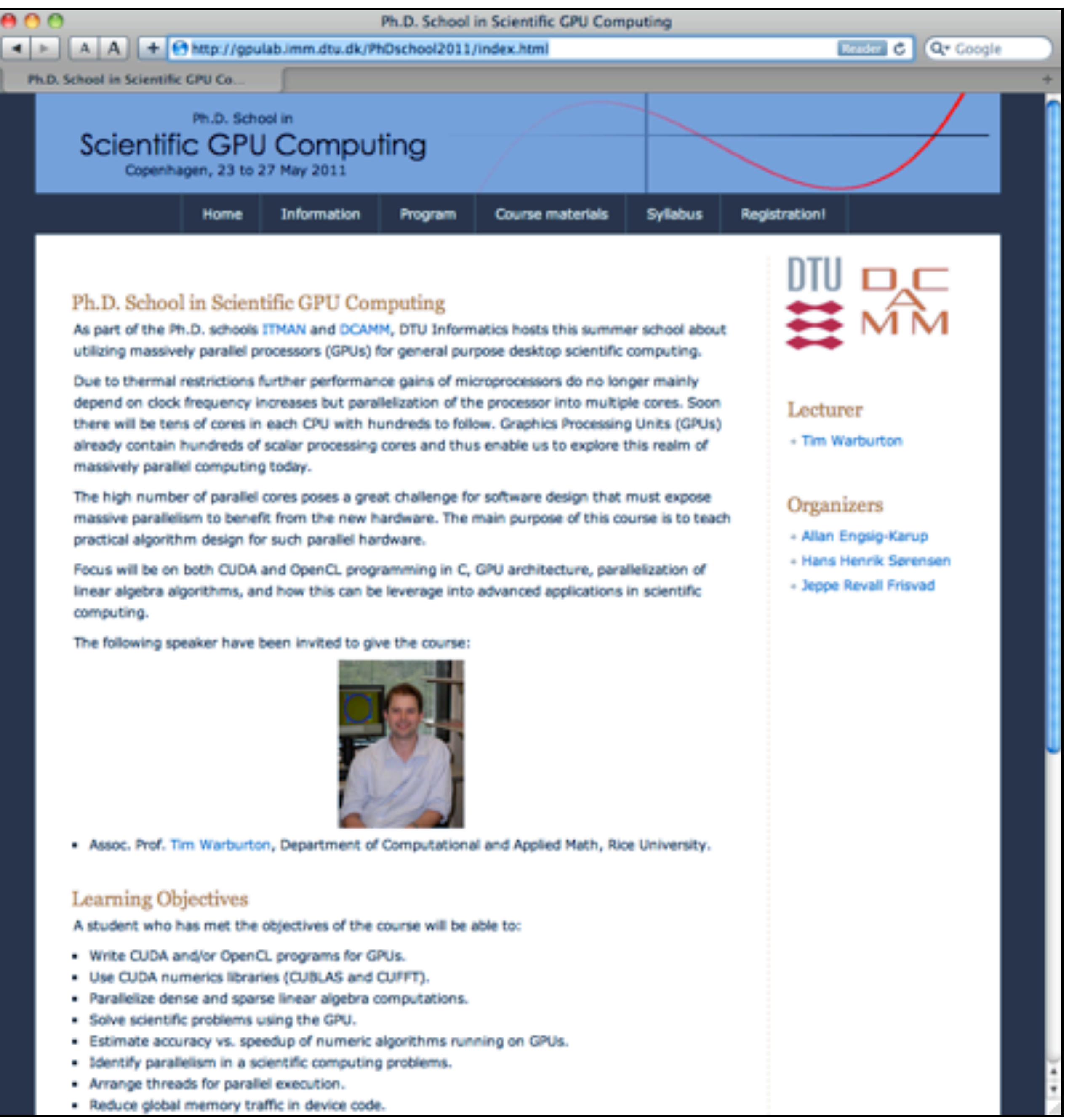
Dr Andreas Klöckner
DG+GPU+Python+
Shock Capturing

Reid Atcheson
DG + GPU + ?

Xin Wang
DG+GPU+
Seismic inversion

Rajesh Gandham
DG+GPU+
Supersonic flows

Denmark GPU Boot Camp 2011



Lecture notes on request.

Talk Overview

1. *Porting:* PU Accelerated Discontinuous Galerkin methods
    - Performance of DG time-domain Maxwell's solver.
    - CUDA v. OpenCL
    - Local time-stepping.
2. *Discretization:* Low Storage Curvilinear Discontinuous Galerkin methods:
    - GPU driven modification for curvilinear elements.
3. *Algorithms:* Global seismic modeling on 100s of GPUs
    - Performance of a linear elasticity code GPU kernels for DG on hexahedral elements.
    - Strong and weak scaling study.
4. *Physics:* Gas dynamics
    - Time stepping.
    - Artificial viscosity.

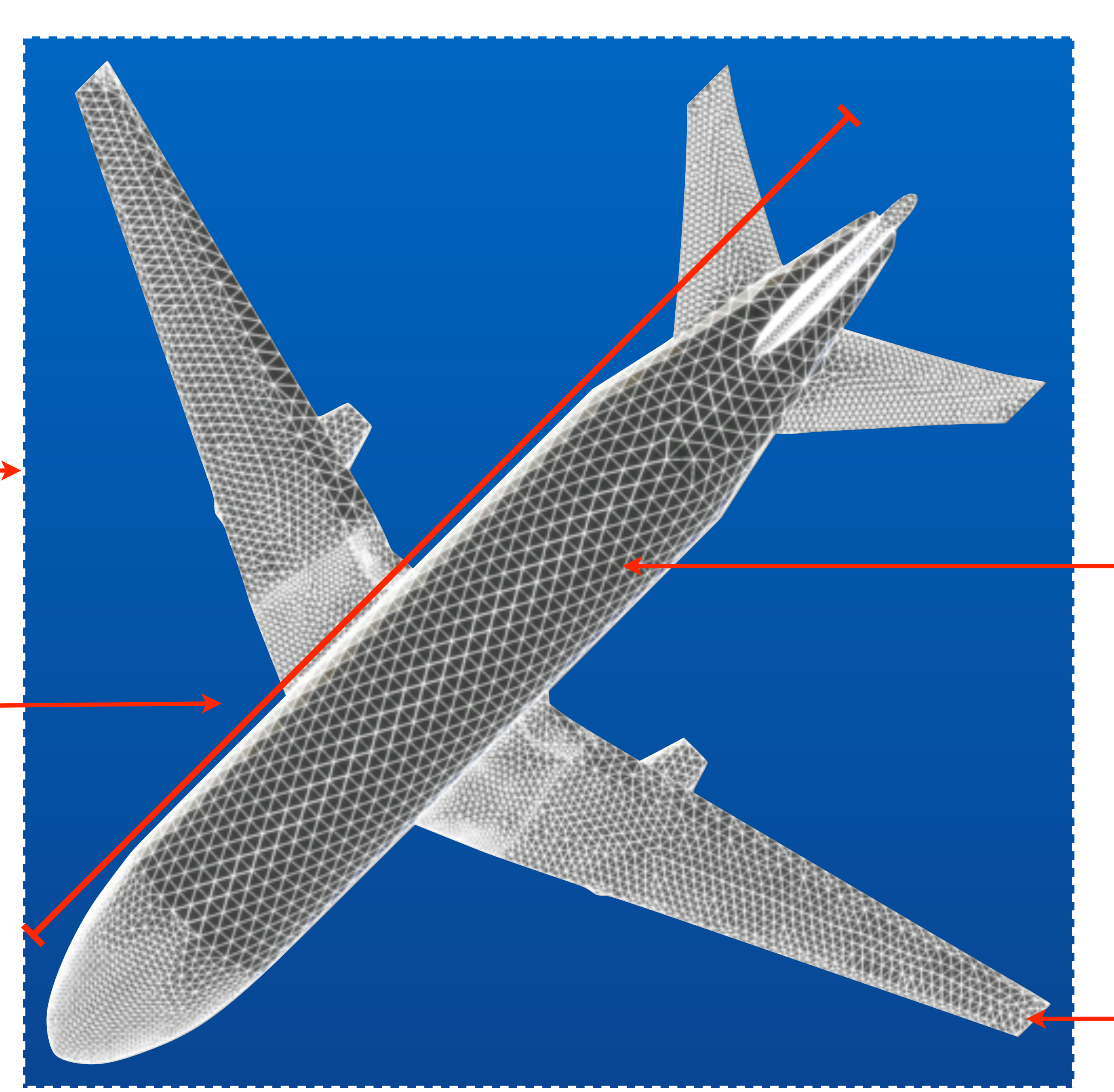
Challenge: numerical wave prop.

Time dependent volume wave propagation

Artificial absorbing BC

Many $\underline{\lambda}$

<u>Compute on a commodity workstation</u>



"High order" approximation of solution

Coarse curvilinear mesh where possible

Multirate time stepping for tiny elements

+ antennae + thin wires + thin material layers
+ more realistic geometry + <u>different physical models</u>

Discontinuous Galerkin Methods

Given the strong form of a PDE in conservation form:

$$\frac{\partial Q_i}{\partial t} + \frac{\partial F_{ij}}{\partial x_j} = 0$$

We mesh the domain and solve a weak form of the PDE in each element with boundary data supplied by its neighboring elements.

Find a weak solution $Q_i \in V^h$ that satisfies

$$0 = \underbrace{\left(\phi, \frac{\partial Q_i}{\partial t} + \frac{\partial F_{ij}}{\partial x_j}\right)_{D^k}}_{\text{Weak form of conservation law}} + \underbrace{\left(\phi, n_j\left(F_{ij}^* - F_{ij}\right)\right)_{\partial D^k}}_{\text{Distributional derivative contribution}}$$

for all ϕ in variational space V^h

We boil away the details on discretization to reveal a (possibly nonlinear) ODE:

$$\frac{d\mathbf{Q}}{dt} + \mathbf{F}(\mathbf{Q}) = 0$$

Linear PDEs (like Maxwell's)
$\mathbb{F}$ is sparse & block-dense.
$\mathbb{F}$ is large & never stored.

$$\frac{d\mathbf{Q}}{dt} = -\mathbf{F}\mathbf{Q} + \mathbf{S}$$

Discontinuous Galerkin Methods

Given the strong form of a PDE in conservation form:

$$\frac{\partial Q_i}{\partial t} + \frac{\partial F_{ij}}{\partial x_j} = 0$$

We mesh th... solve a wea... PDE in each... boundary d... its neighbor...

V^h that satisfies

$$n_j \left(F_{ij}^* - F_{ij} \right) \Big)_{\partial D^k}$$

...tional derivative contribution

...space V^h

We boil aw... discretizatio... (possibly no...

Linear PDEs (like Maxwell's)

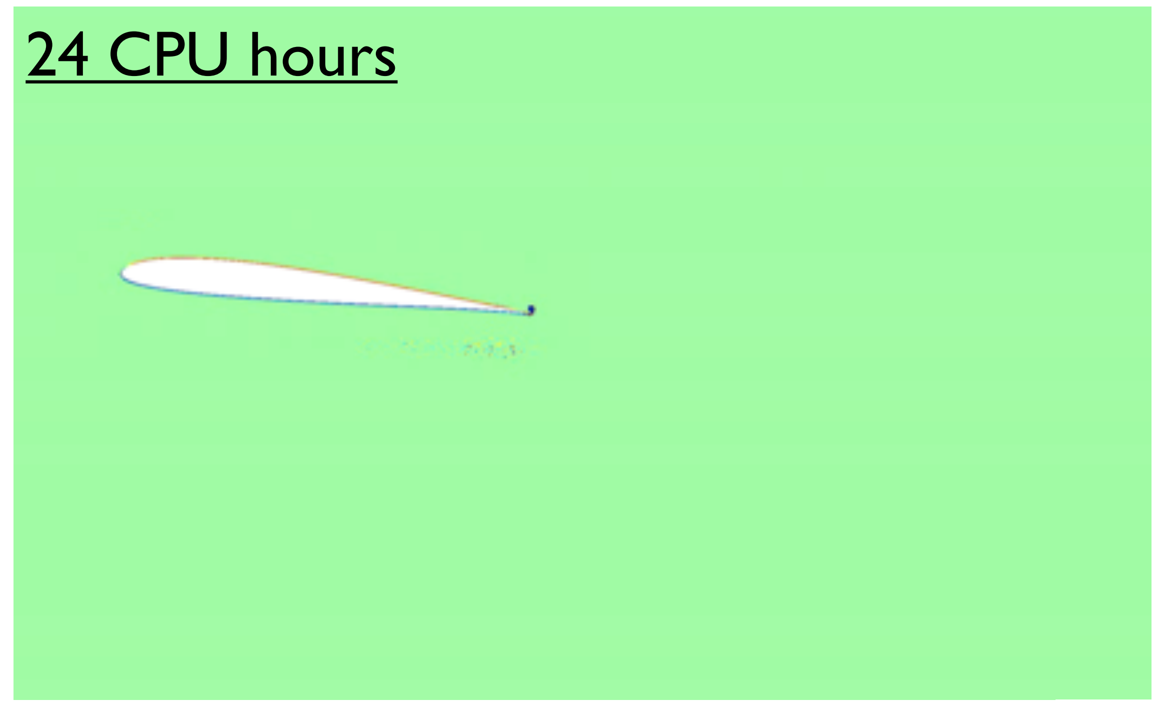
F is sparse & block-dense.

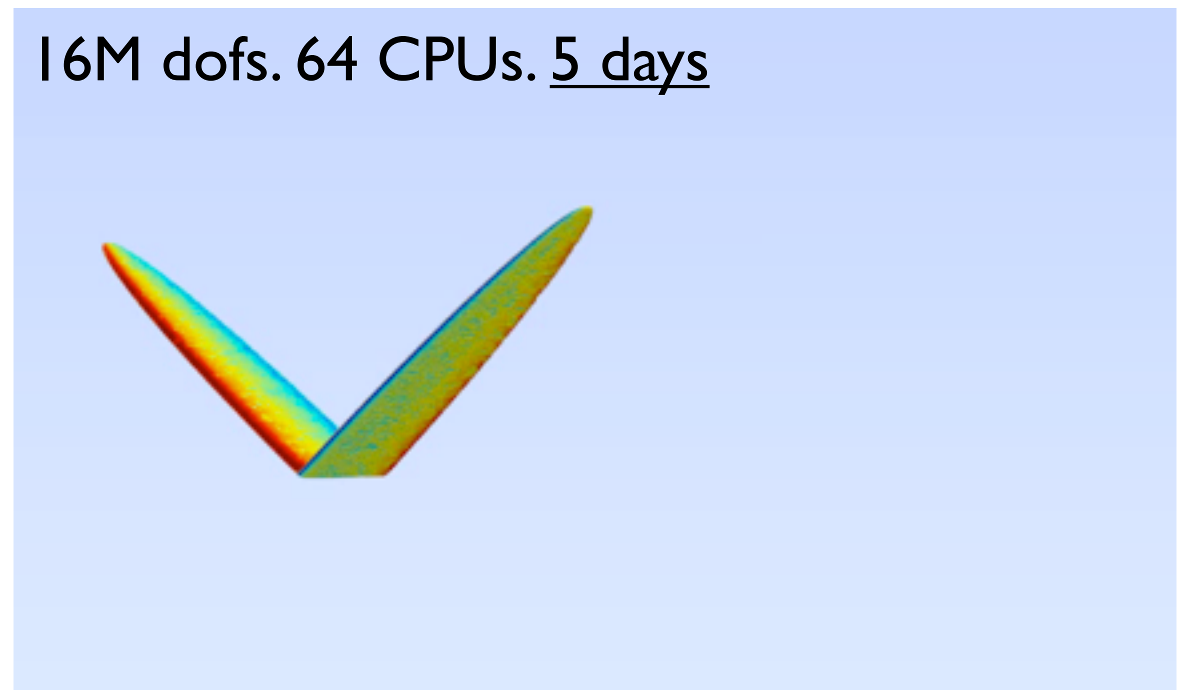
F is large & never stored.

$$\frac{d\mathbf{Q}}{dt} = -\mathbf{F}\mathbf{Q} + \mathbf{S}$$

Discontinuous Galerkin

DG solvers enable simulations of fluid flow, acoustics, electromagnetics ...
but the computations are demanding !!



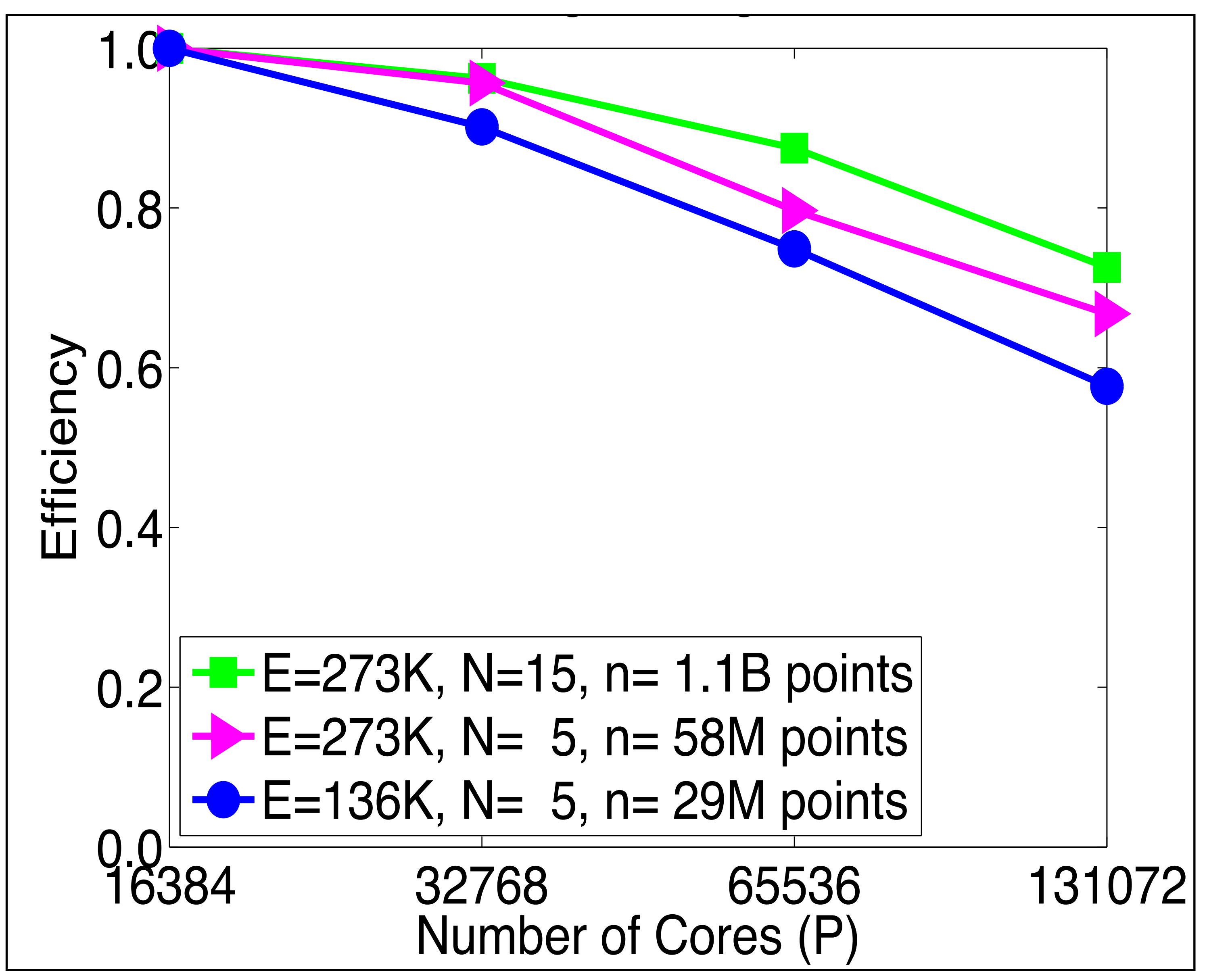


The DG simulations shown courtesy of Per-Olof Persson, UC Berkeley.

RICE

Discontinuous Galerkin: scalable

NekCEM is a DG based time-domain Maxwell's solver on hex elements.



Excellent strong scaling has been demonstrated on 130K+ cores.

Source: Min & Fischer,

Performance Analysis on the IBM Blue Gene/P for Wakefield Calculations.

NekCEM: Strong Scaling on IBM BG/P

The DG Grab Bag

Elements	**Spaces/Basis**	**Fluxes**	***Time stepping***	**Stabilization**
Quadrilaterals	Vector	Upwind	Adams-Bashforth	Filter
Triangles	Divergence free	Central	Runge-Kutta	TVB Limiter
Tetrahedra	Conforming	Lax-Friedrichs	Exponential	Moment Limiter
Hexahedra	Nodal	Multiscale(HMM)	Leap frog	Artificial Viscosity
Prisms	Non-polynomial	...	Multirate Adams	Shock Capturing
Pyramids	Monomials		Multirate Taylor	Hybrid Models
Polyhedral	Orthonormal		Symplectic	*hp* Adaptivity
Isoparametric	Vertex centered		Lax Wendroff	Shock Sensor
...	...		Space-time	...
			Parareal	
			JFNK	
			Implicit-explicit	
			...	

Choose 5 for an instant paper/talk/poster

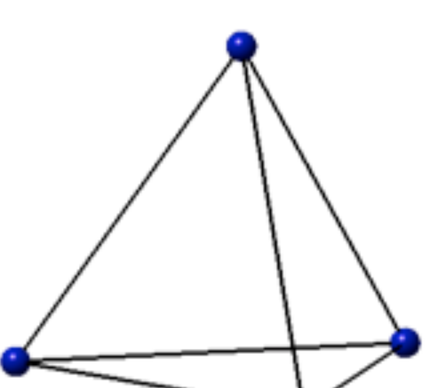
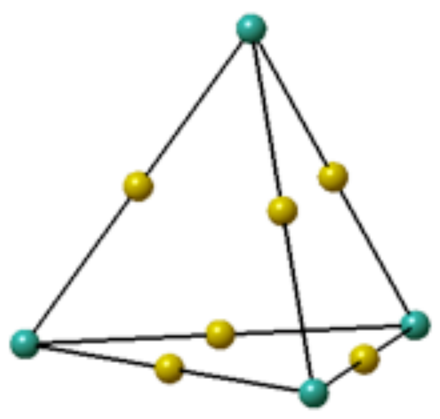
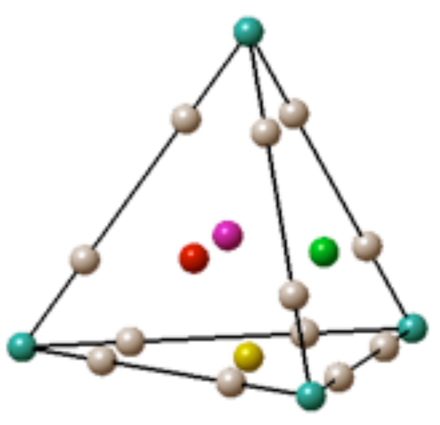
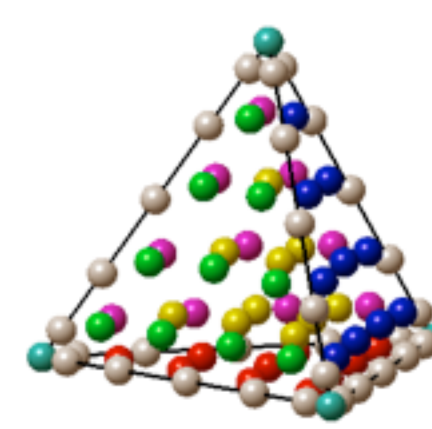
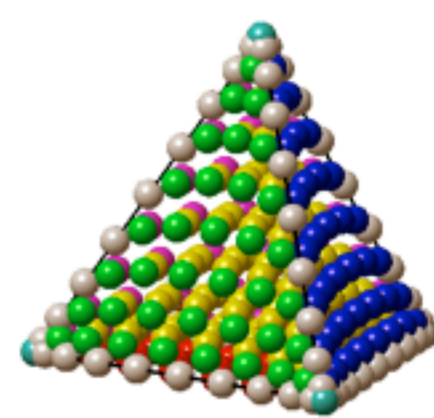
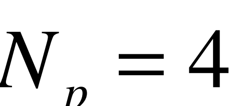
DG Grab Bag: this talk

Elements	**Spaces/Basis**	**Fluxes**	***Time stepping***	**Stabilization**
Quadrilaterals	Vector	Upwind	Adams-Bashforth	Filter
Triangles	Divergence free	Central	Runge-Kutta	TVB Limiter
Tetrahedra	Conforming	Lax-Friedrichs	Exponential	Moment Limiter
Hexahedra	Nodal	Multiscale(HMM)	Leap frog	Artificial Viscosity
Prisms	Non-polynomial	...	Multirate Adams	Shock Capturing
Pyramids	Monomials		Multirate Taylor	Hybrid Models
Polyhedral	Orthonormal		Symplectic	*hp* Adaptivity
Isoparametric	Vertex centered		Lax Wendroff	Shock Sensor
...	...		Space-time	...
			Parareal	
			Chebyshev RK	
			Implicit-explicit	
			...	

Seriously: should architecture inform DG design choices to maximize performance ?

Low Order versus High Order

The solution is represented on each element as a multivariate polynomial (interpolated at Warp & Blend nodes)

N=1 "Linears"	N=2 "Quadratics"	N=3 "Cubics"	N=6 "Sextics"	N=10 "Decics"
				
 $N_p = 4$	$N_p = 10$	$N_p = 20$	$N_p = 84$	$N_p = 220$

Approximation error $\sim h^{\min(N+1,\sigma)}$. Storage $\sim N^3 / h^3$. Element size $\sim h$

DG Options Element Type

Two common choices of elements in 3D are tetrahedra and hexahedra using *N'th* order polynomial approximation:

Element	Nodes/Element	FLOPS	FLOPS / LOADS
Tetrahedron	$N_p = \frac{(N+1)(N+2)(N+3)}{6}$	$O\left(\frac{N^6}{36}\right)$	$O\left(\frac{N^3}{6}\right)$
Hexahedron	$N_p = (N+1)^3$	$O(N^4)$	$O(N)$

Dimensional splitting on the hex yields a much lower op count per node for local elemental operations like interpolation or differentiation.

Q: Does the excess floating point capability of GPGPUs make this moot ?

GPU Accelerated Discontinuous Galerkin Methods

A. Klöckner, T. Warburton, J. Bridge, and J.S.Hesthaven, *High-Order Discontinuous Galerkin Methods on Graphics Processors,* Journal of Computational Physics, 2009.

Incomplete History of DG & Maxwell’s Equations

Progression of DG methods for time-domain electromagnetics:

- M. Remaki and L. Fezoui, *Une Méthode de Galerkin Discontinu pour la résolution des équations de Maxwell en milieu hétérogene*. Technical report RR-3501, INRIA, 1998.

- TW, *Application of the discontinuous Galerkin method to Maxwell’s equations using unstructured polymorphic hp-finite elements*, Lecture Notes in Computational Science and Engineering, 2000.

- D.A. Kopriva, S.L. Woodruff, and M.Y. Hussaini, *Discontinuous spectral element approximation of Maxwell’s Equations*, Lecture Notes in Computational Science and Engineering, 2000.

- M. Remaki, *A new finite volume scheme for solving Maxwell’s equations*. *COMPEL*, 2000.

- S. Piperno, M. Remaki and L. Fezoui, *A nondiffusive finite volume scheme for the three-dimensional Maxwell's equations on unstructured meshes*, SINUM 2002.

- J.S.Hesthaven and TW, *Nodal High-Order Methods on Unstructured Grids, JCP 2002.*

- ... ***many papers*** including advances in local time stepping... ***few*** commercial codes...

- N. Gödel, S. Schomann, T. Warburton, and M. Clemens, *Discontinuous Galerkin Methods for Electromagnetic Radio Frequency Problems*, IEEE, 2009.

- A. Klöckner, TW, J. Bridge, and J.S.Hesthaven, *High-Order Discontinuous Galerkin Methods on Graphics Processors*, JCP 2009.

DG Maxwell's Variational Equation

DG methods time march the solution locally on each element, with boundary data from neighbor elements.

For all triangles $\{D_k\}$ find $(H,E) \in X^h(D_k) \times Y^h(D_k)$ such that

$$0 = \left(\phi, \frac{\partial \mu H}{\partial t} + \nabla \times E\right)_{D_k} + \left(\phi, n \times \left(E^* - E\right)\right)_{\partial D_k}$$

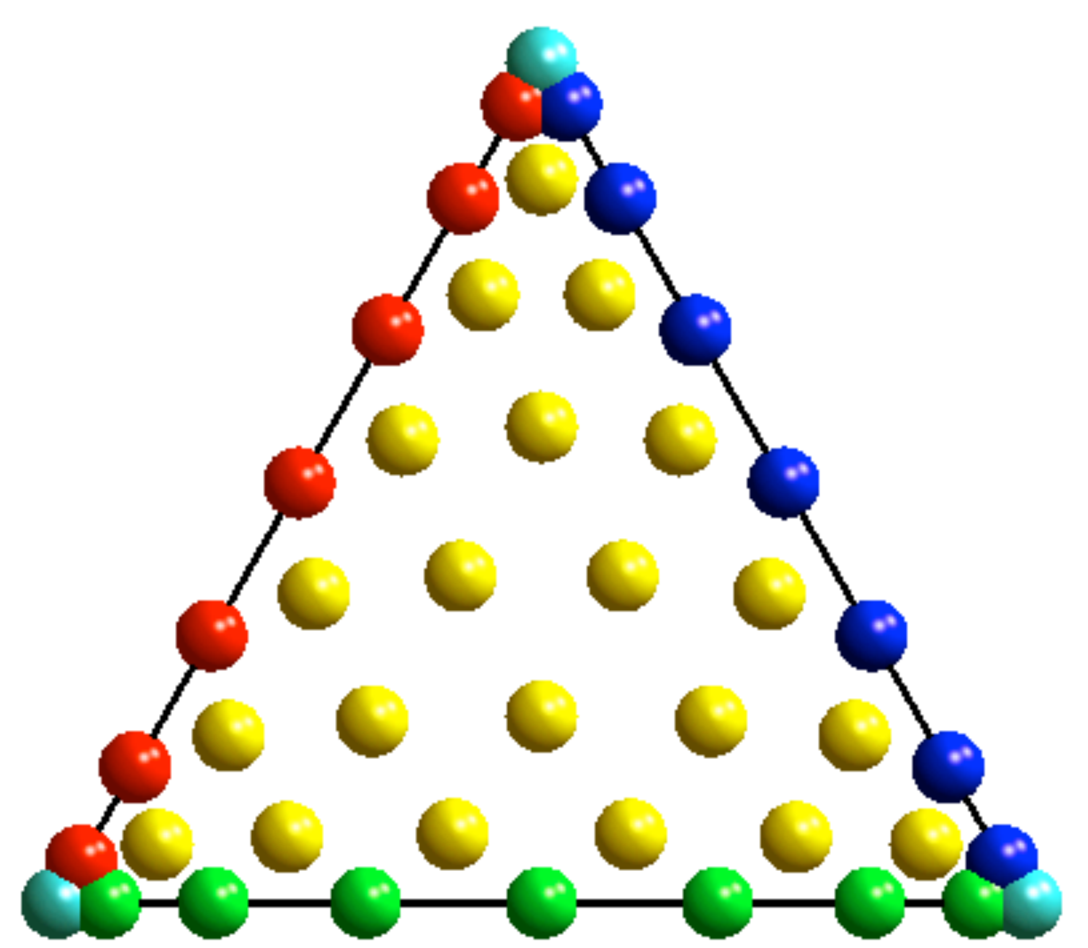
$$0 = \left(\psi, \frac{\partial \varepsilon E}{\partial t} - \nabla \times H\right)_{D_k} \underbrace{- \left(\psi, n \times \left(H^* - H\right)\right)_{\partial D_k}}_{\text{Distributional derivative contribution}}$$

holds for all $(\phi,\psi) \in X^h(D_k) \times Y^h(D_k)$

The H* and E* variables are the magnetic and electric fields obtained by upwinding at the element boundaries.

DG Grid Topology: element local

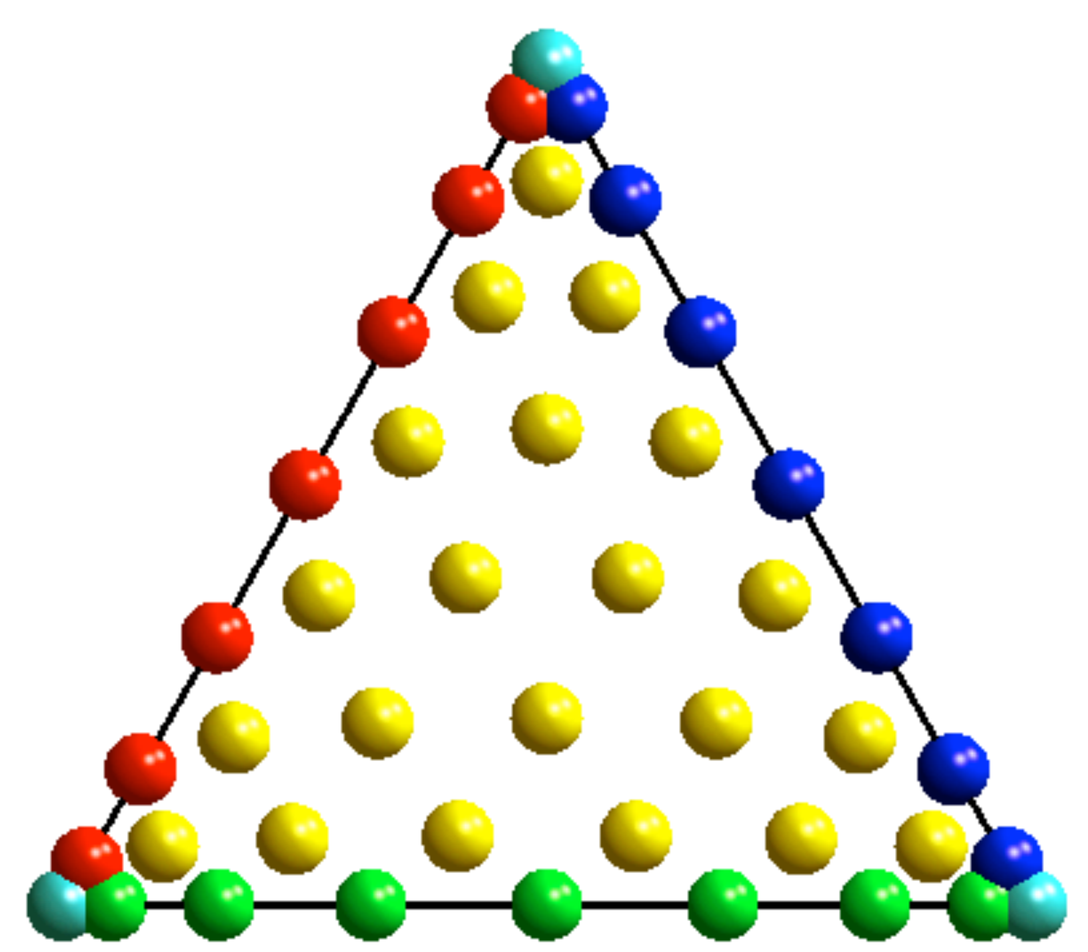
There is a common perception of a false dichotomy between unstructured and structured grids.



High order elements have regular internal node-node topology.

DG Grid Topology: inter element

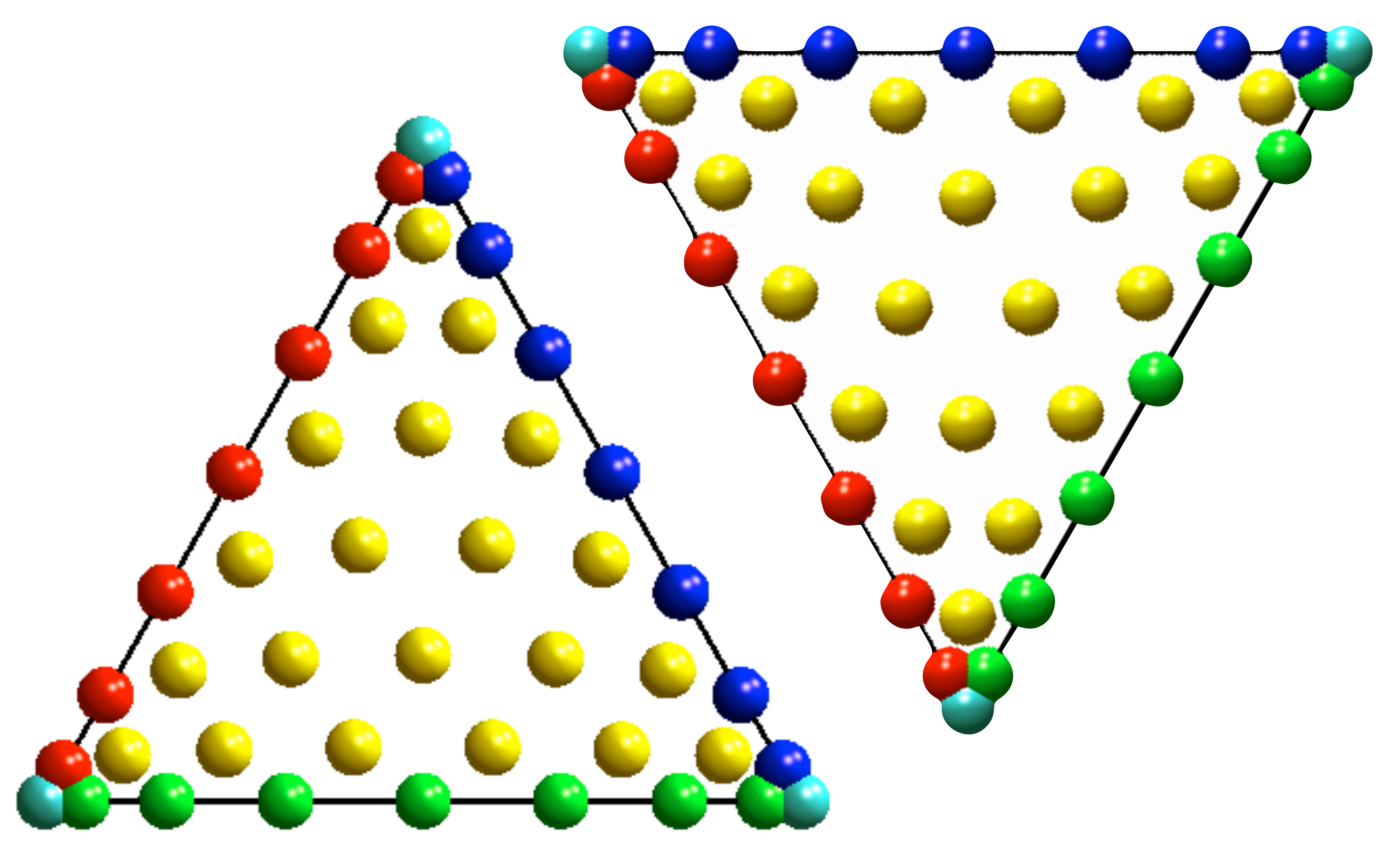
The connectivity between elements is by flux exchange.



Node-node connectivity is per boundary node per face.

DG Grid Topology: inter element

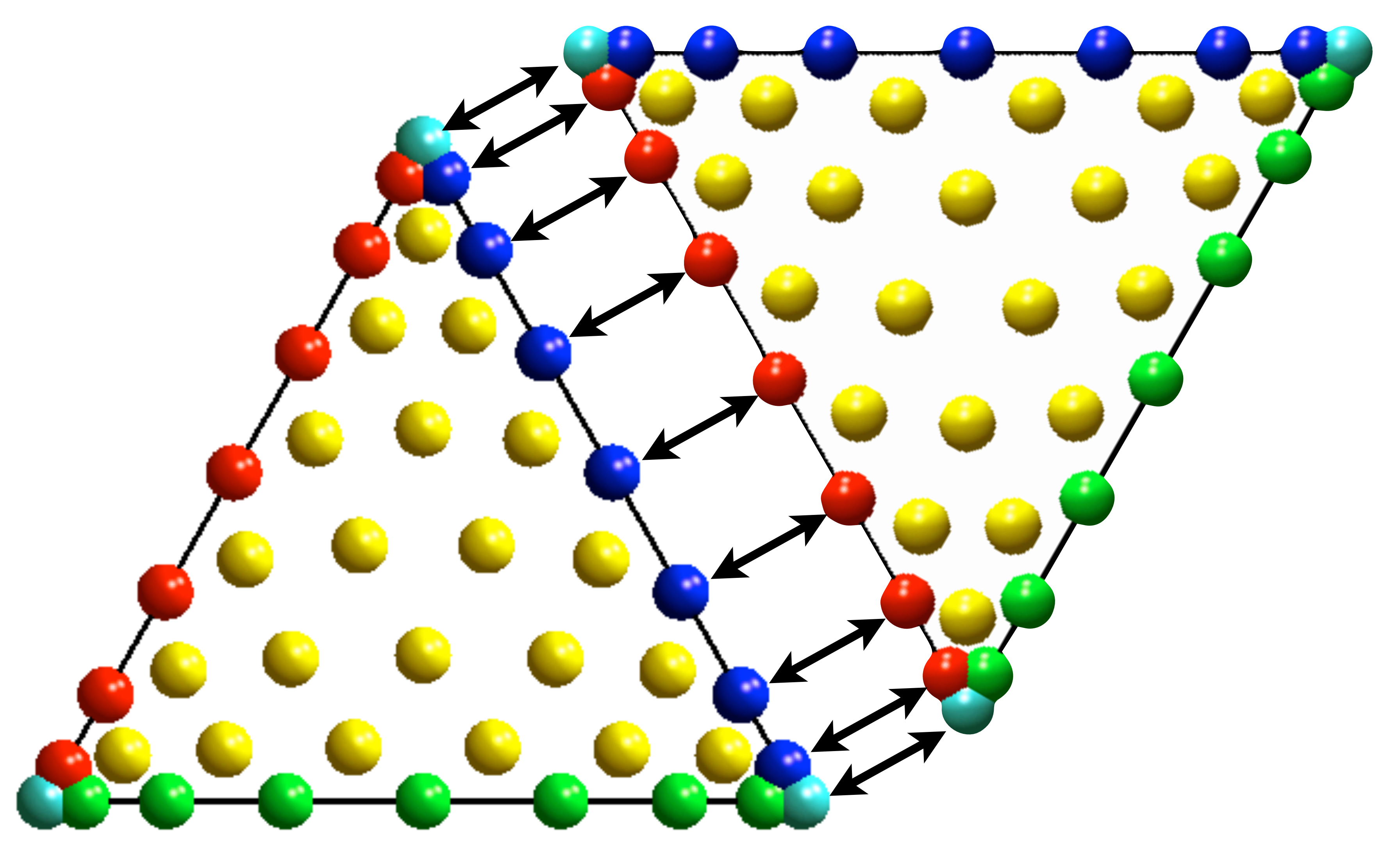
The connectivity between elements is by flux exchange.



Node-node connectivity is per boundary node per face.

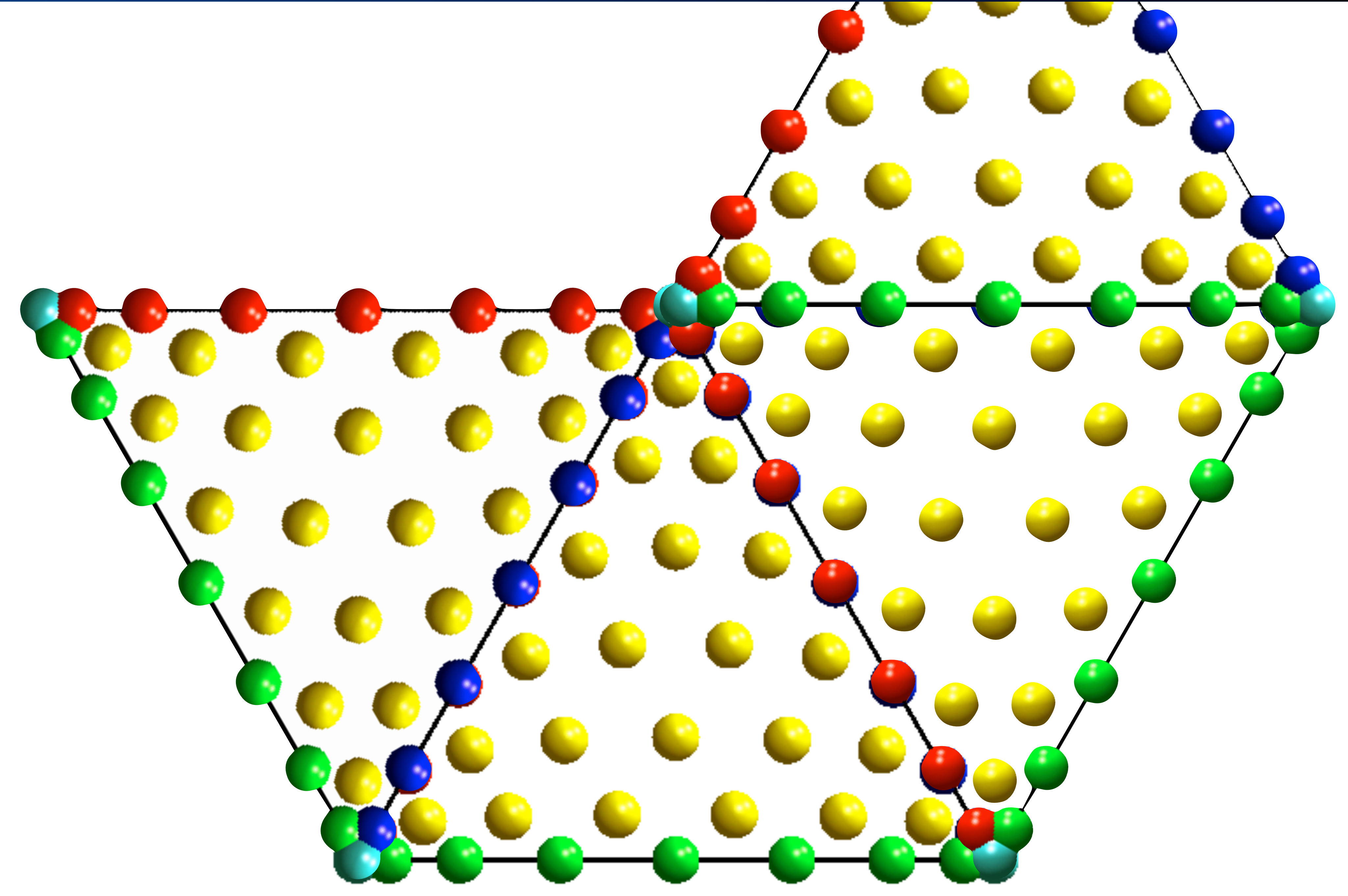
DG Grid Topology: inter element

The connectivity between elements is by flux exchange.



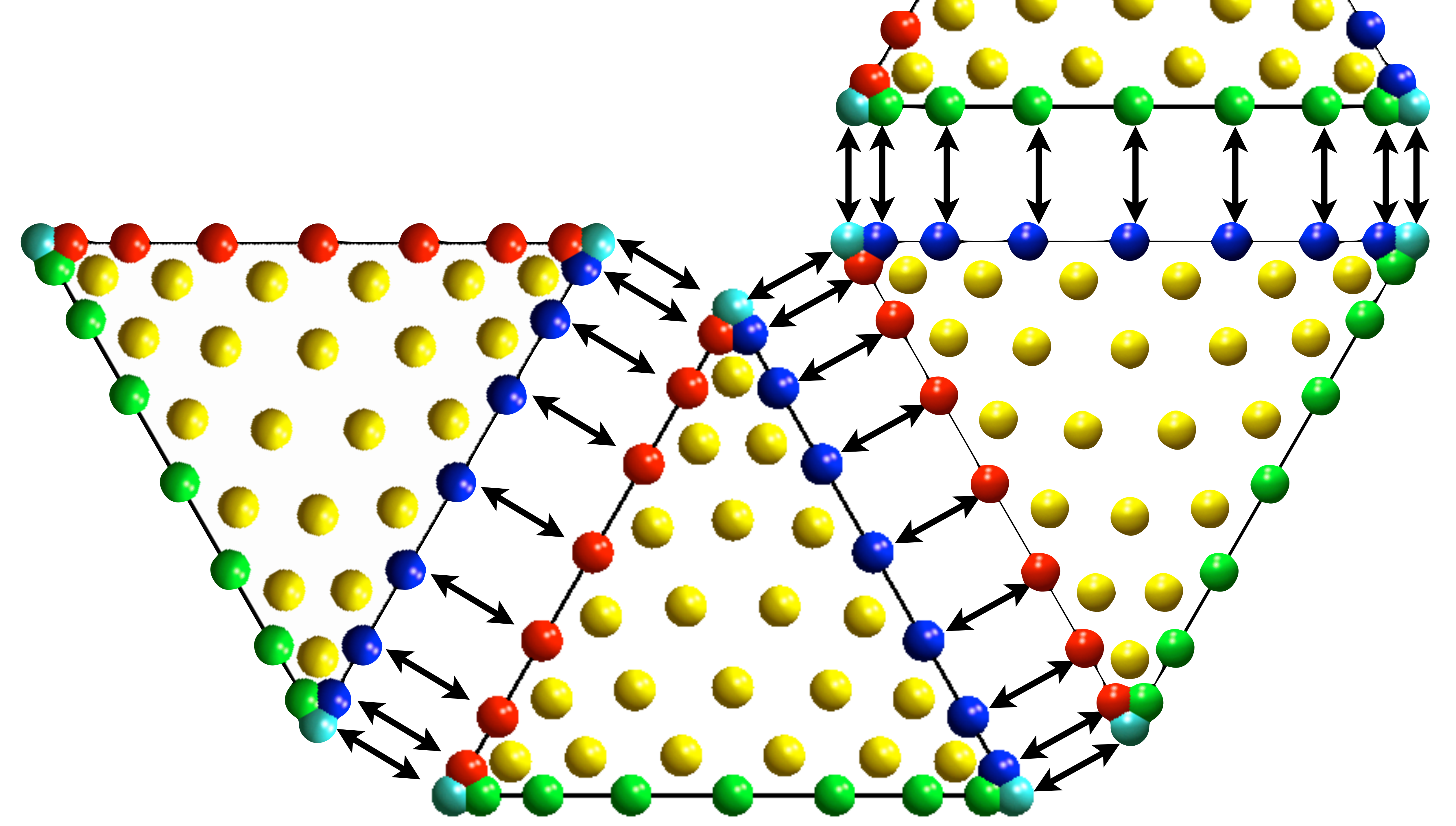
Node-node connectivity is per boundary node per face.

DG Grid Topology: inter element



Elements that just share a vertex are “disconnected”.

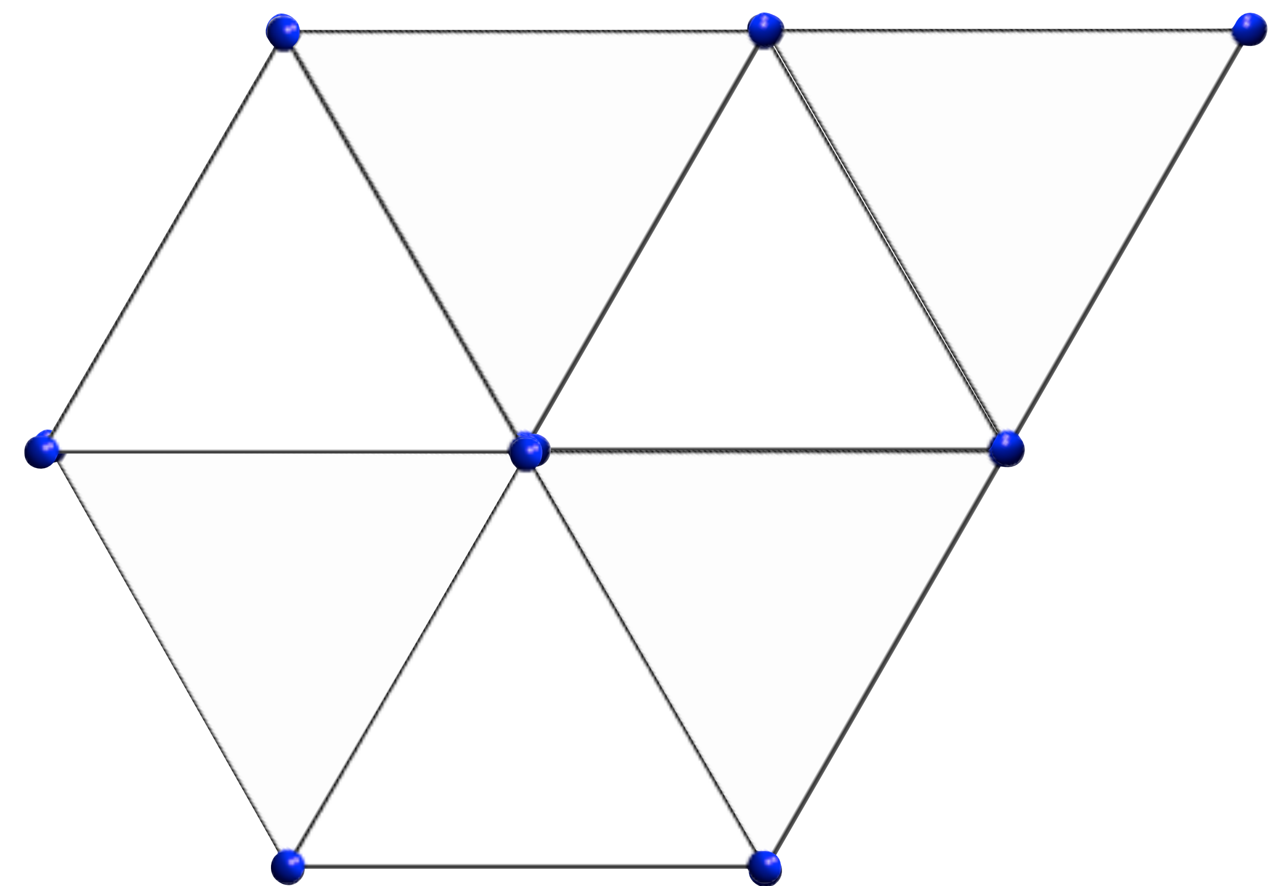
DG Grid Topology: inter element



The solution is multivalued at the element boundary nodes.
i.e. no need to maintain "coherency" with gather/scatter

P_1 FEM Connectivity

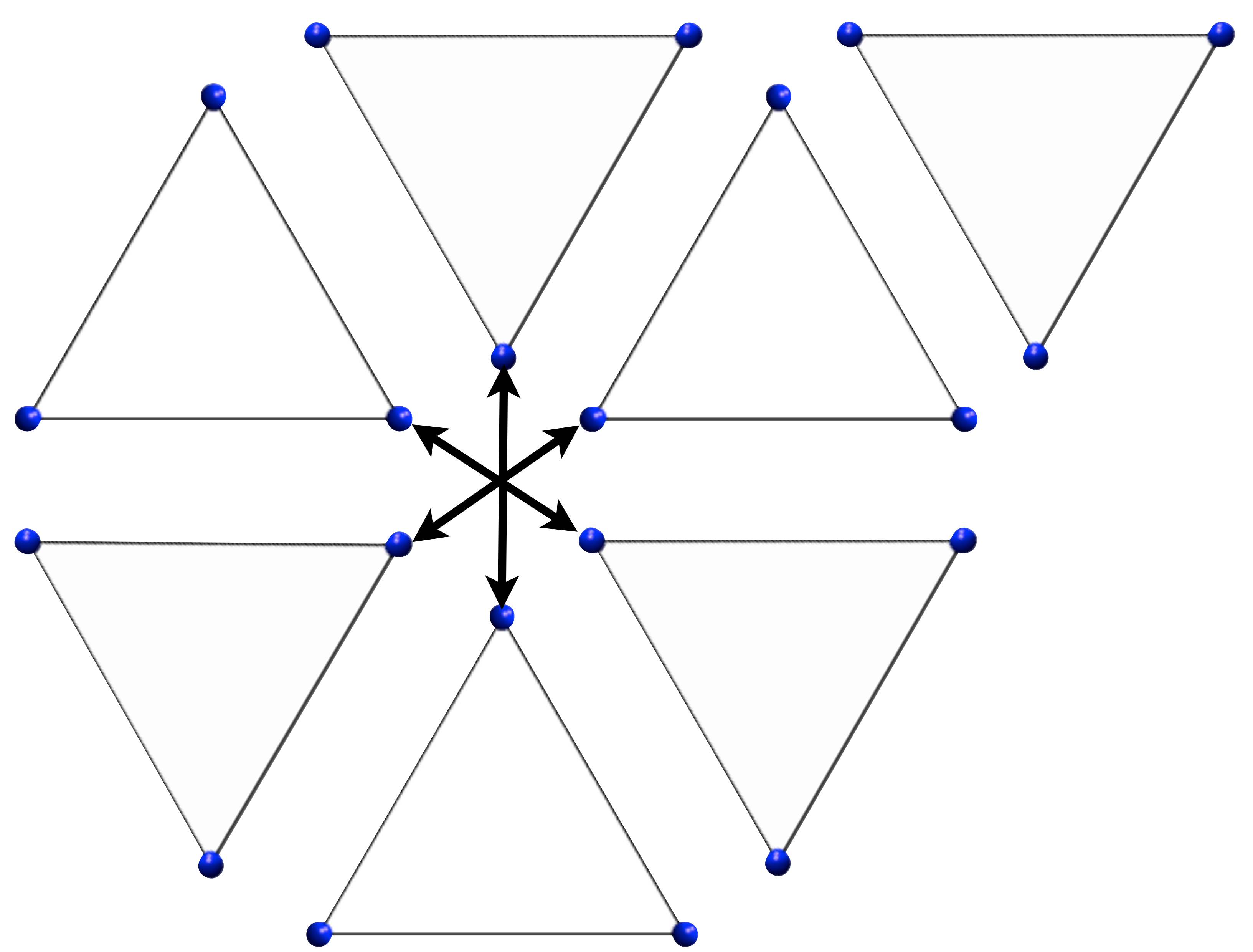
By contrast linear FEM topology involves all nodes



Node-node connectivity is per boundary node per face.

P_1 FEM Connectivity

More book keeping per dof & more complicated MPI.



Red-green ordering or similar => lower performance.
Similar issue with SEM assembly.

Learning Curve

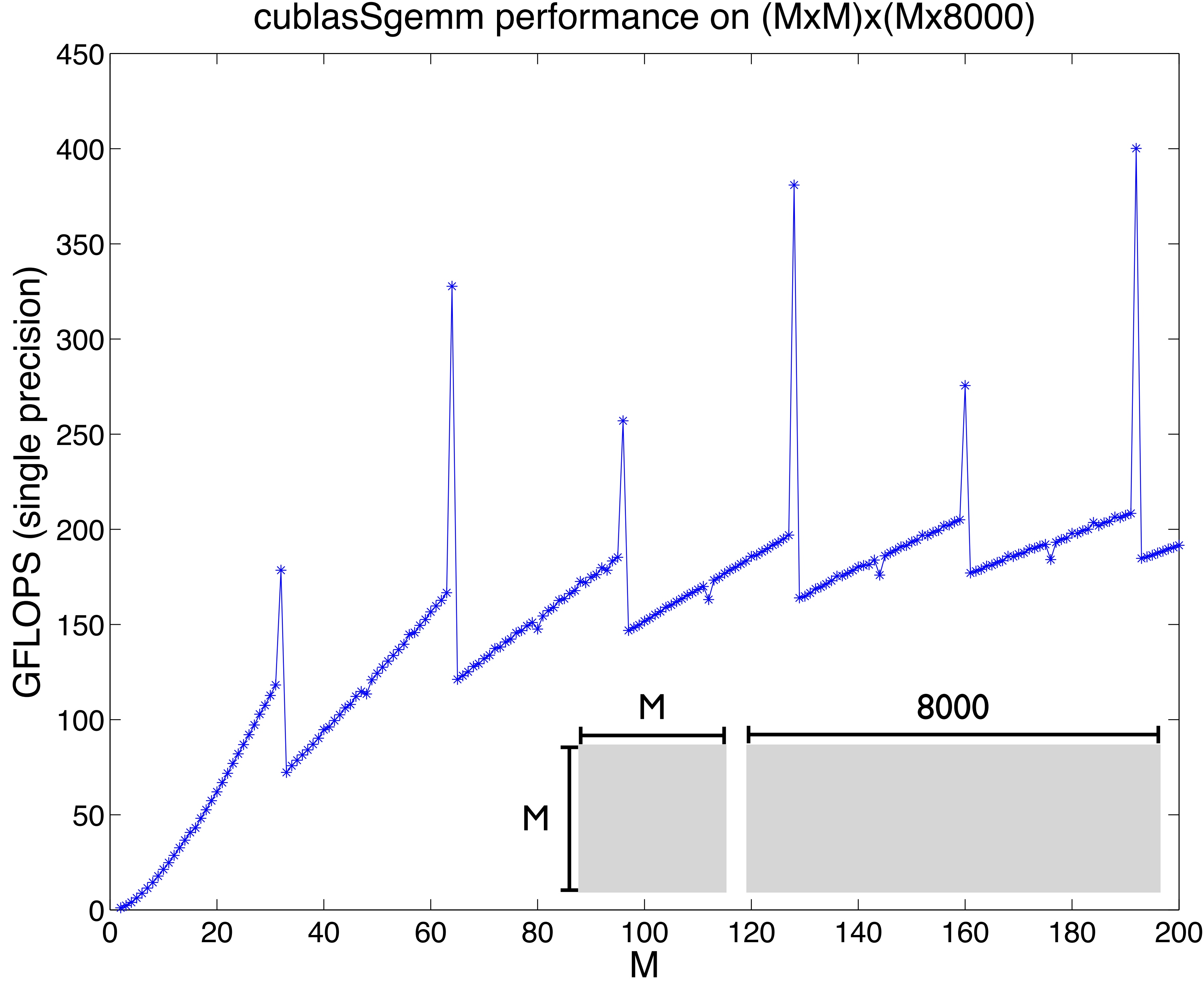
We spent 5 days during summer 2008
porting our core DGTD solvers using CUDA:

Day 1: ported using cuBLAS ☹☹☹
Day 2: wrote our first CUDA kernels ☺
Day 3: added shared memory for field data ☺☺
Day 4: used texture memory for operator matrix data ☺☺☺
Day 5: improved data layout patterns ☺☺☺☺

Overall speed up is 30x or more when compared with
SSE accelerated code on a single 3GHz Intel core

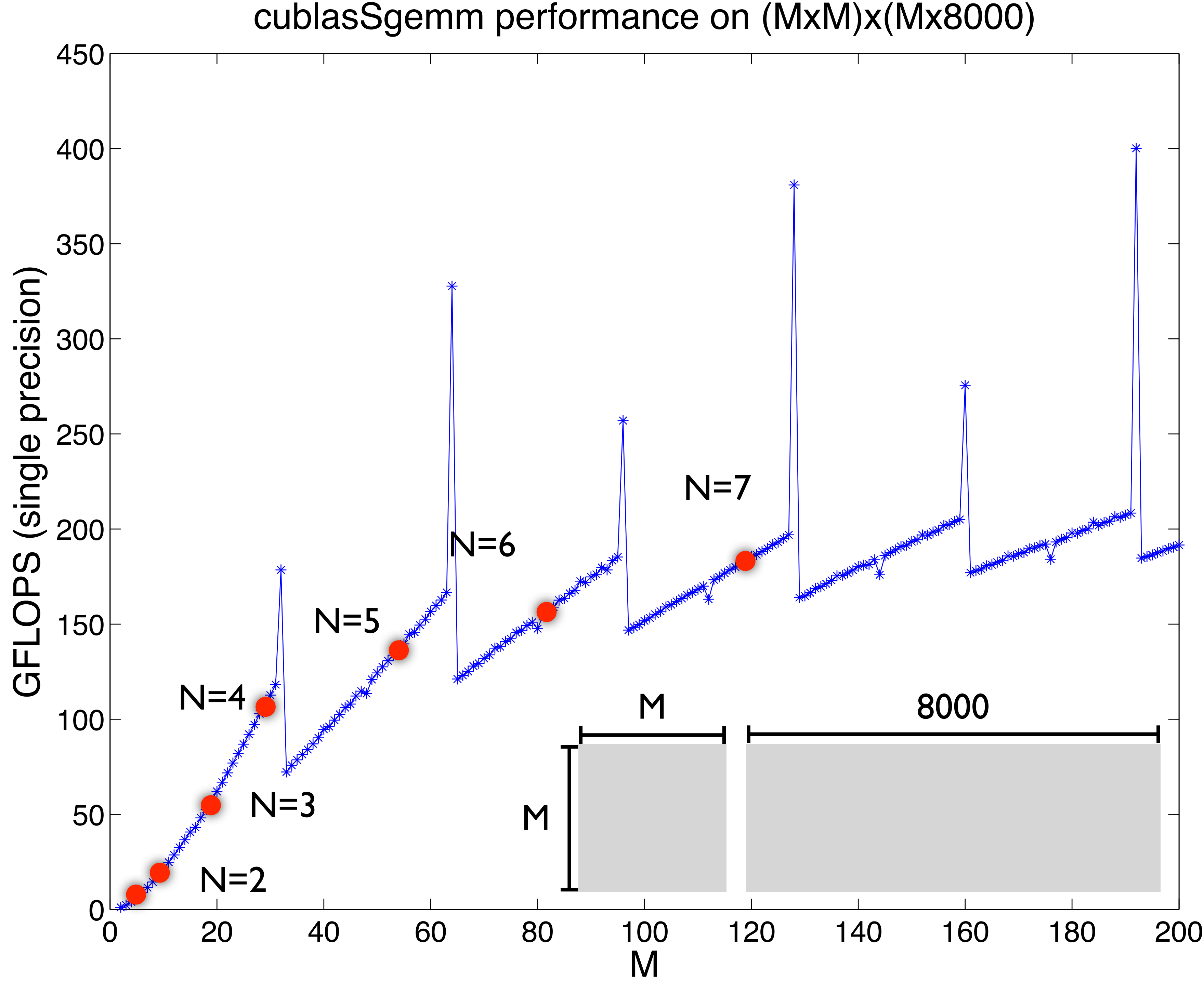
Additional time was spent enabling multiple GPUs
through pThreads and/or MPI...

Day 1: CUBLAS



The cublasSGEMM performance degrades for small block matrix multiplication.

Day 1: CUBLAS

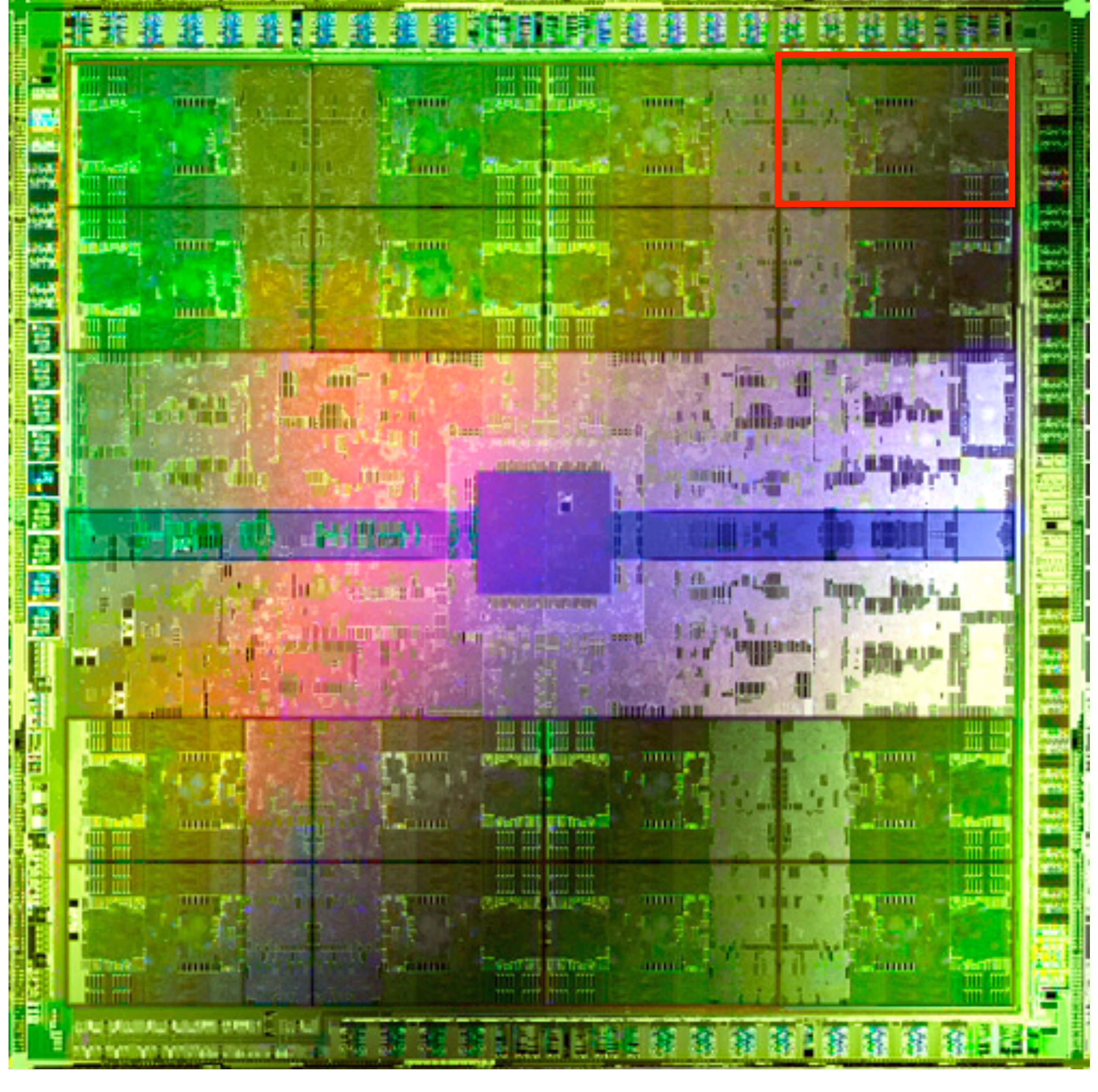


More problematic: cuBLAS achieves <30 GFLOPS for level 1 & 2 operations.

Day 2-5: Learning the NUMA

High memory latency hidden by simultaneous multithreading with 1000s of threads.

Fermi has 16 streaming multiprocessors

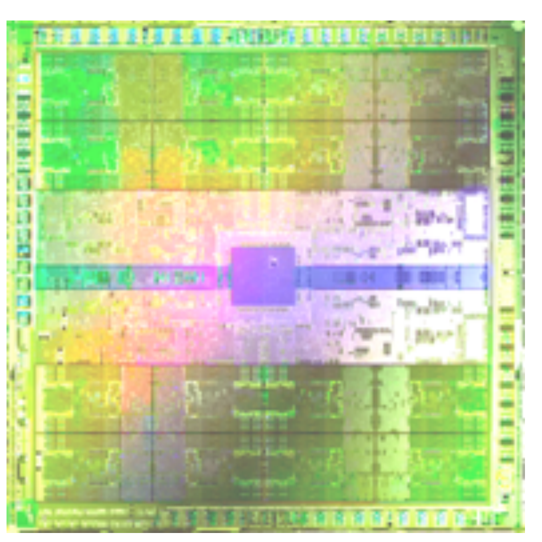


Each SM is a cluster of 32 "cores"

The GPU cores load/store ~50 GFLOATS per second
& perform a total of ~1.4 TFLOPS per second.
Theoretical peak performance requires >64 FLOPS per output float.

Day 2-5: GPGPU Programming Challenges

NVIDIA Fermi GPU



Up to 192GB/s bandwidth
~1.58 TFLOPS (SP)
up to 512 cores

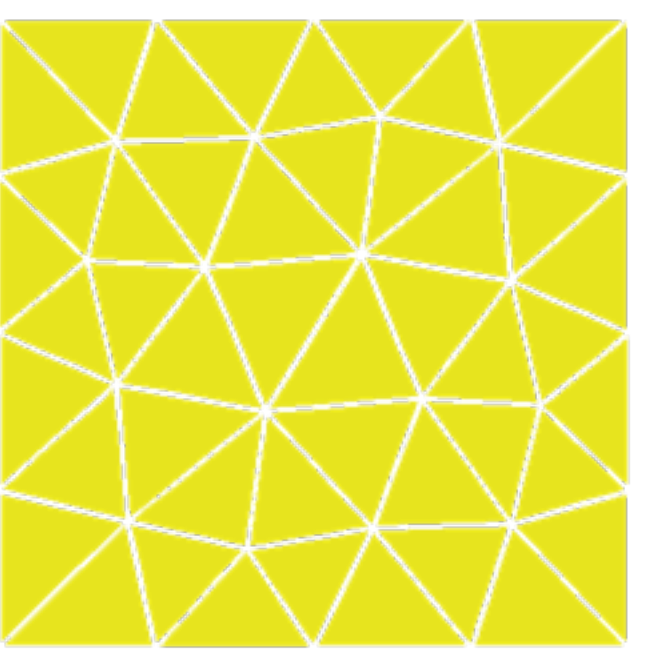
GPU programming is a balancing trick with competing demands to:

- minimize register pressure
- maximize occupancy
- hide high latency of global memory accesses
- maximize parallelism of data load/stores
  - i.e. avoid partition camping, inefficient bus utilization & local memory bank conflicts.
- reduce shared memory footprint
- avoid branching....

with only coarse grain profiling tools.

My initial interest was sparked in 2007 by the high device-memory bandwidth. Modern GPUs typically have an excess of floating point units.

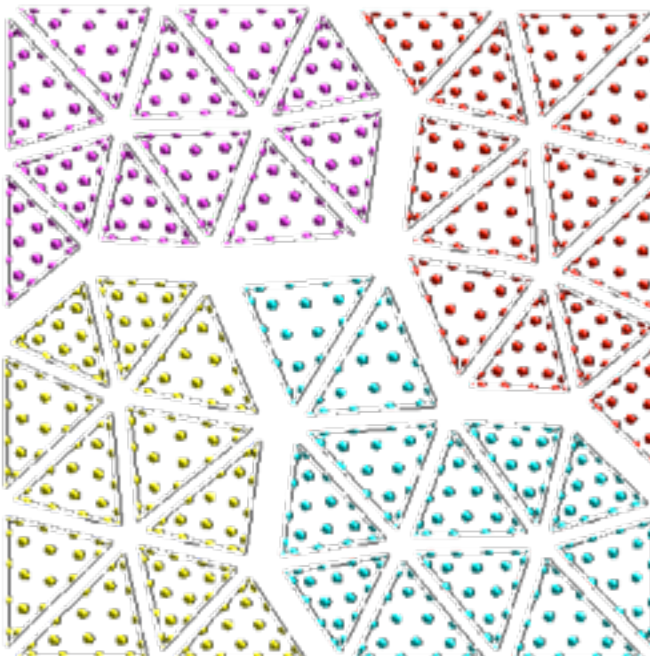
Parallel Partitioning Approach



FEM Mesh

ParMetis partitioning

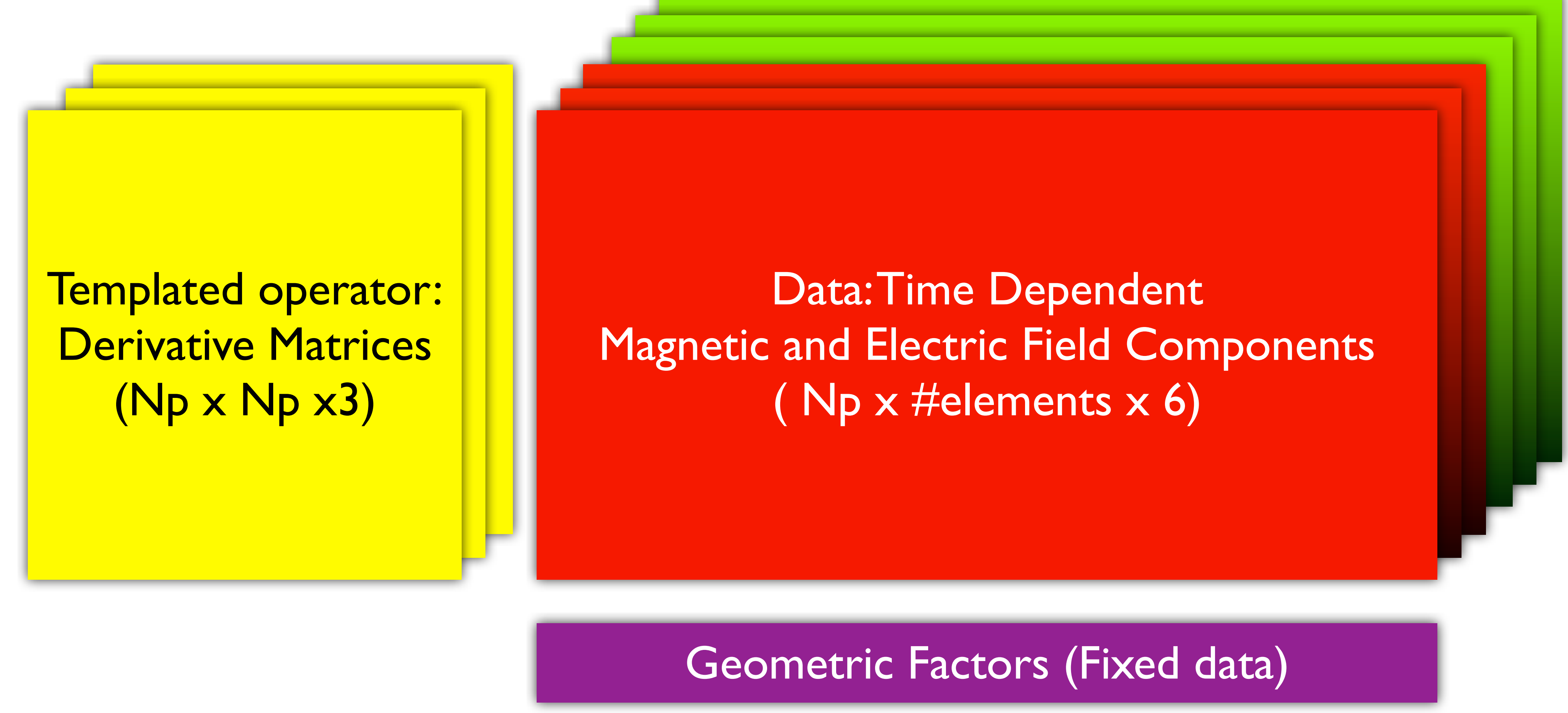
Each element is updated by a block of threads that share fast memory



Vector field updates at a node are done by a single CUDA thread

Template Curl Operation

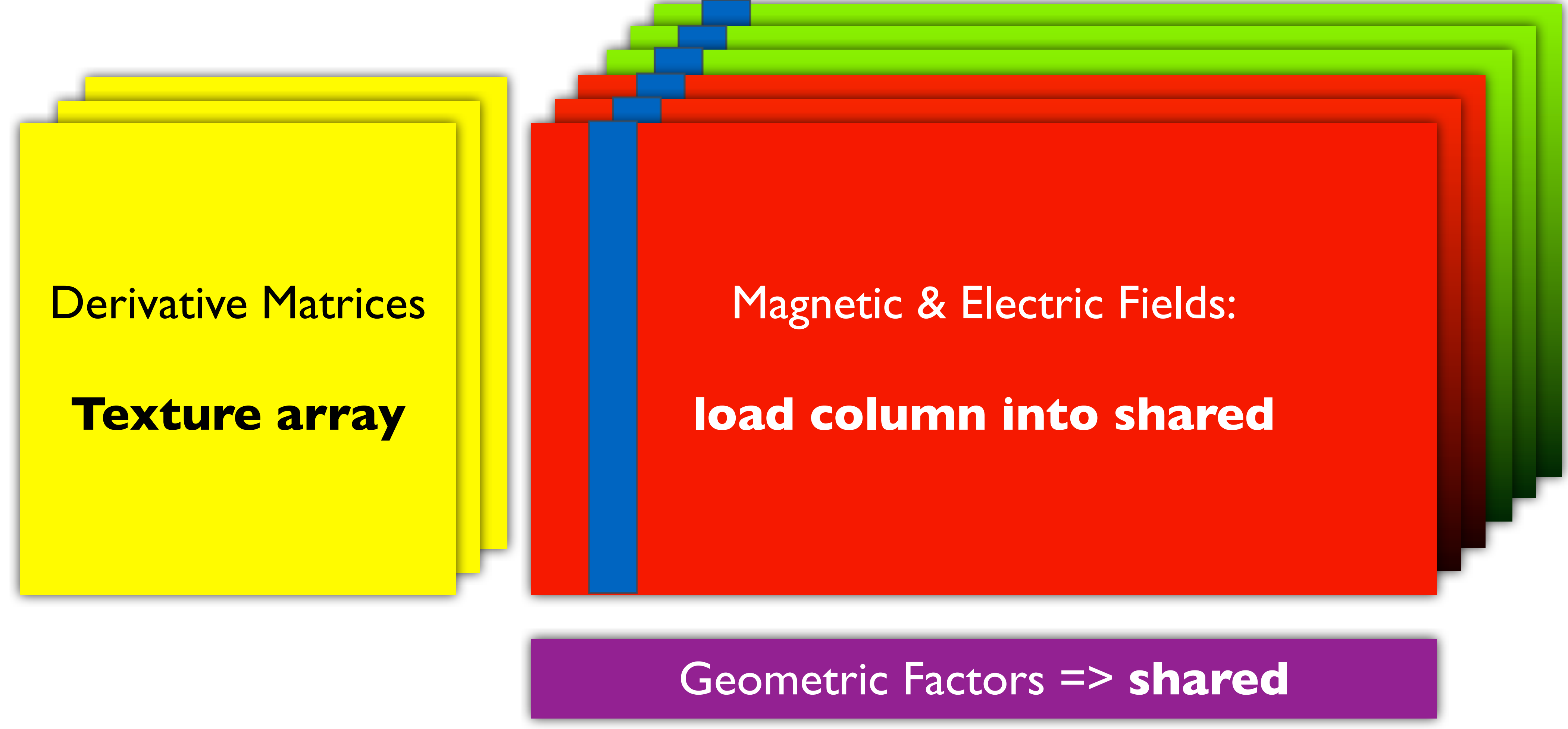
Discretizing the "curl" terms on straight sided tetrahedra yields templated derivative matrices that are dense and require lots of FLOPS.



One thread is responsible for computing the curl of H and E at one node

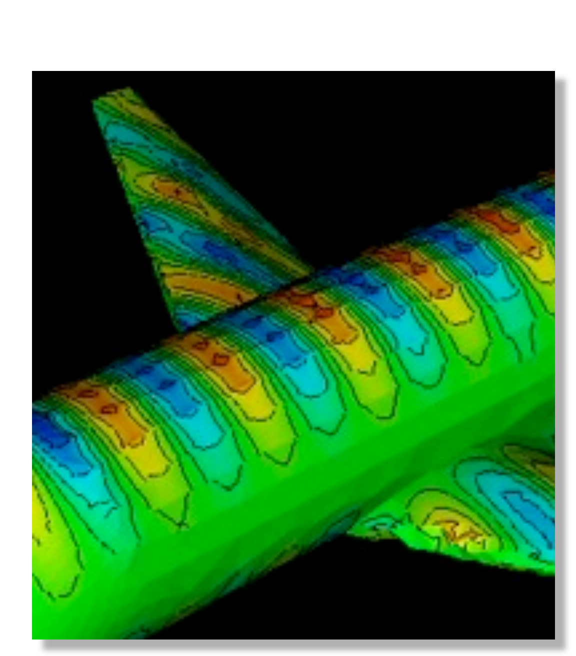
Hand Coded CUDA Implementation

After some experimentation our hand tuned kernel:
compute the elemental *curl* **H** and *curl* **E** to a thread block.



One thread is responsible for computing the curl of H and E at one node

Day 5: CUDA Based Maxwell's DGTD

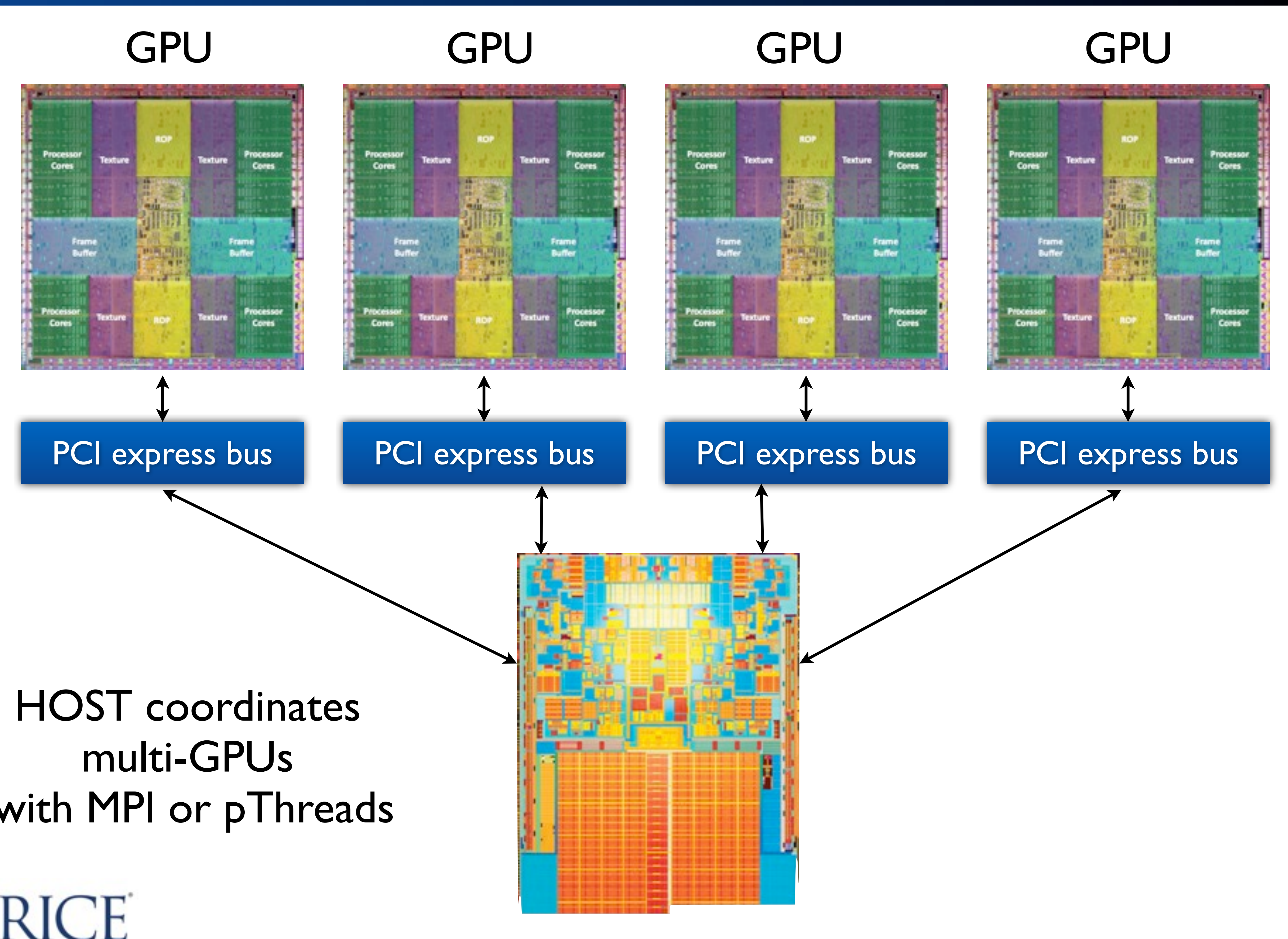


K=227,000 & 27M DOFs
cubic polynomial fields

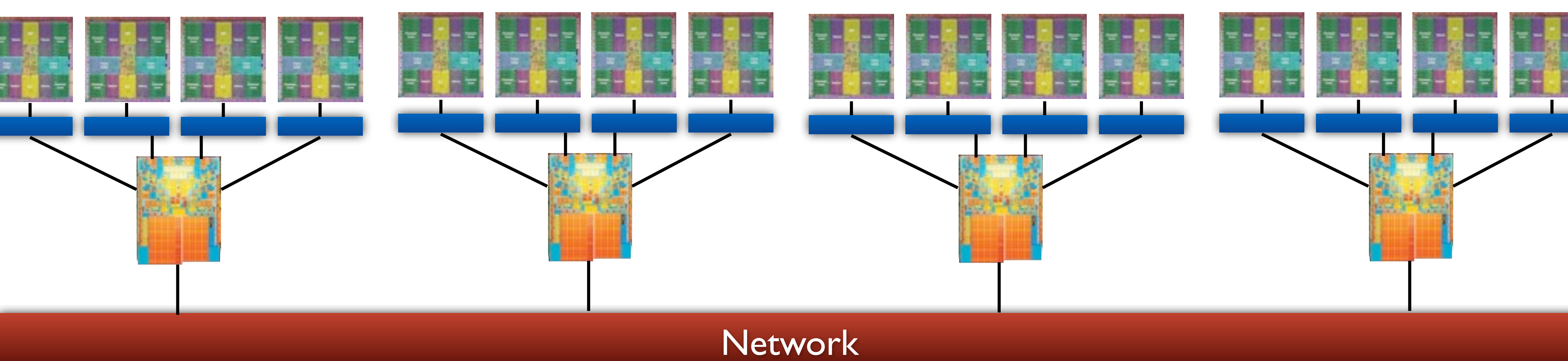
Computation: Nigel Nunn

Note the artifacts generated by a paneled representation of the toy plane

Day 6: CUDA+pThreads



Day 7: CUDA & pThreads & MPI



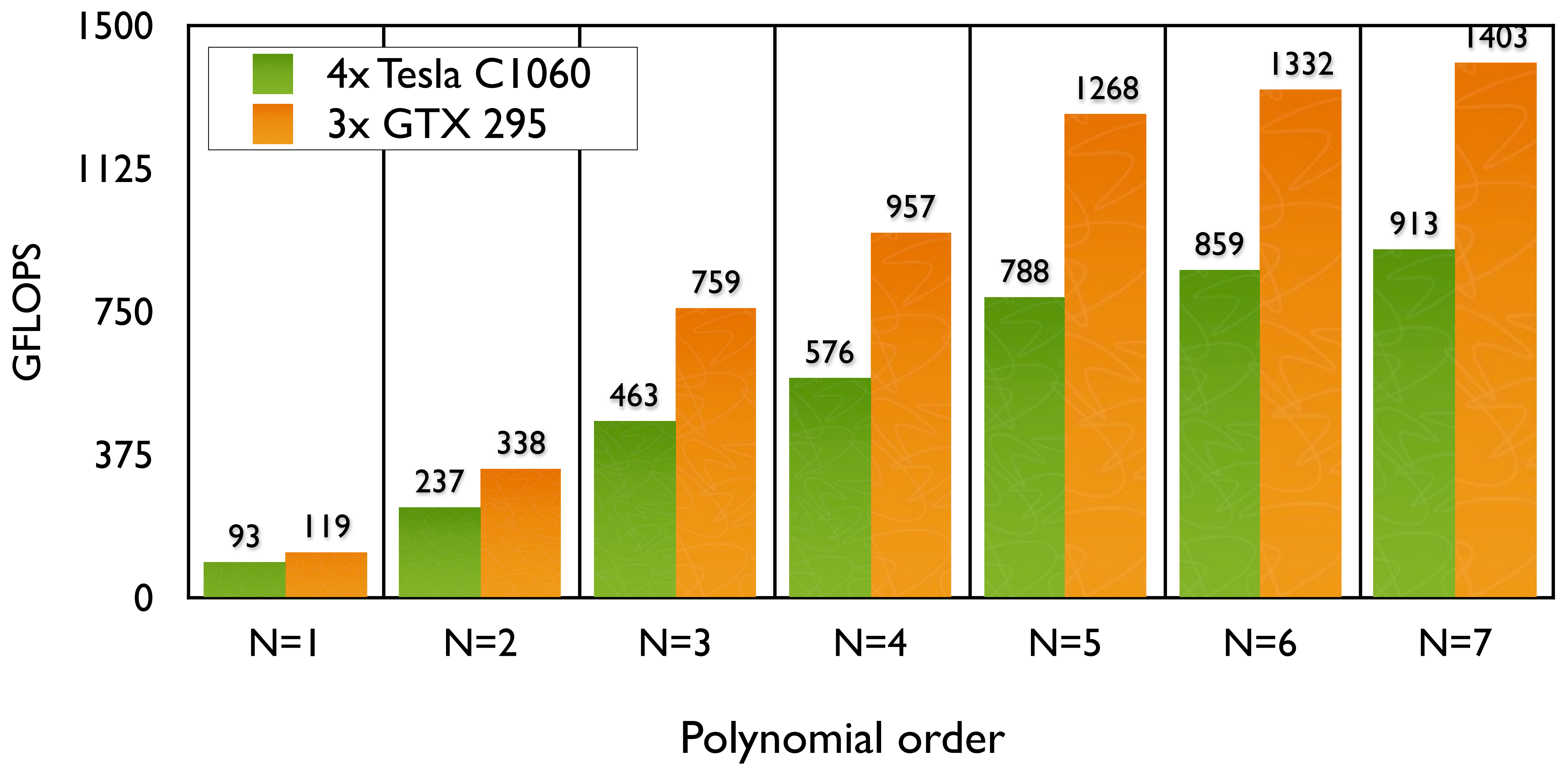
Communications:

GPU threads: CUDA

HOST threads: pThreads (optional)

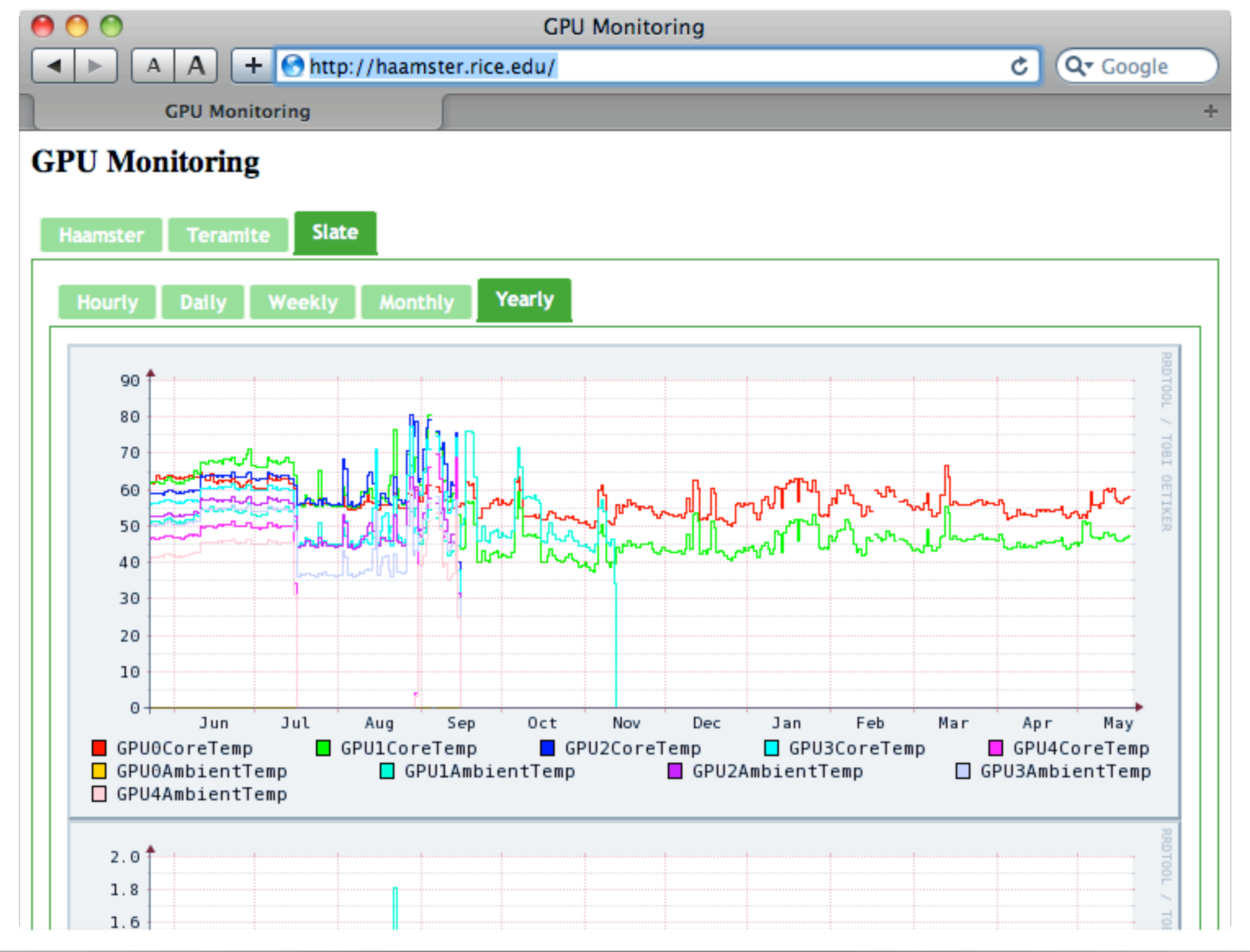
HOST to HOST: MPI

Day 7: Hand Coded CUDA DG Performance



Baseline: our SSE/CPU implementation achieves ~25 GFLOPS on a 2.66GHz Intel Yorkfield Quad Core Q9450 (similar vintage to the GPU)

Day 8: Thermal Monitoring



In the early days we caused unexpected computation failures... by baking the chips.

Scaling on 3 Generations of GPUs

GFLOPS	Tesla C870 (128 cores)	Tesla C1060 (240 cores)	Tesla C2050 (448 cores)	GTX 480 (480 cores)	GTX 590 (2x512 cores)
N=3	45	115	137	196	255
N=4	85	144	237	290	428
N=5	125	197	328	460	735
N=6	125	214	332	454	890

Solver performance has improved through three generations with efficiencies gained from increases core count, core frequency, bus size, & caching.

Kernel Performance on Fermi (2010)

29 GFLOPS
145 GB/s

RK

Curl

Surface

341 GFLOPS
151 GB/s

642 GFLOPS
"284" GB/s

Tet Aggregate:
460 GFLOPS

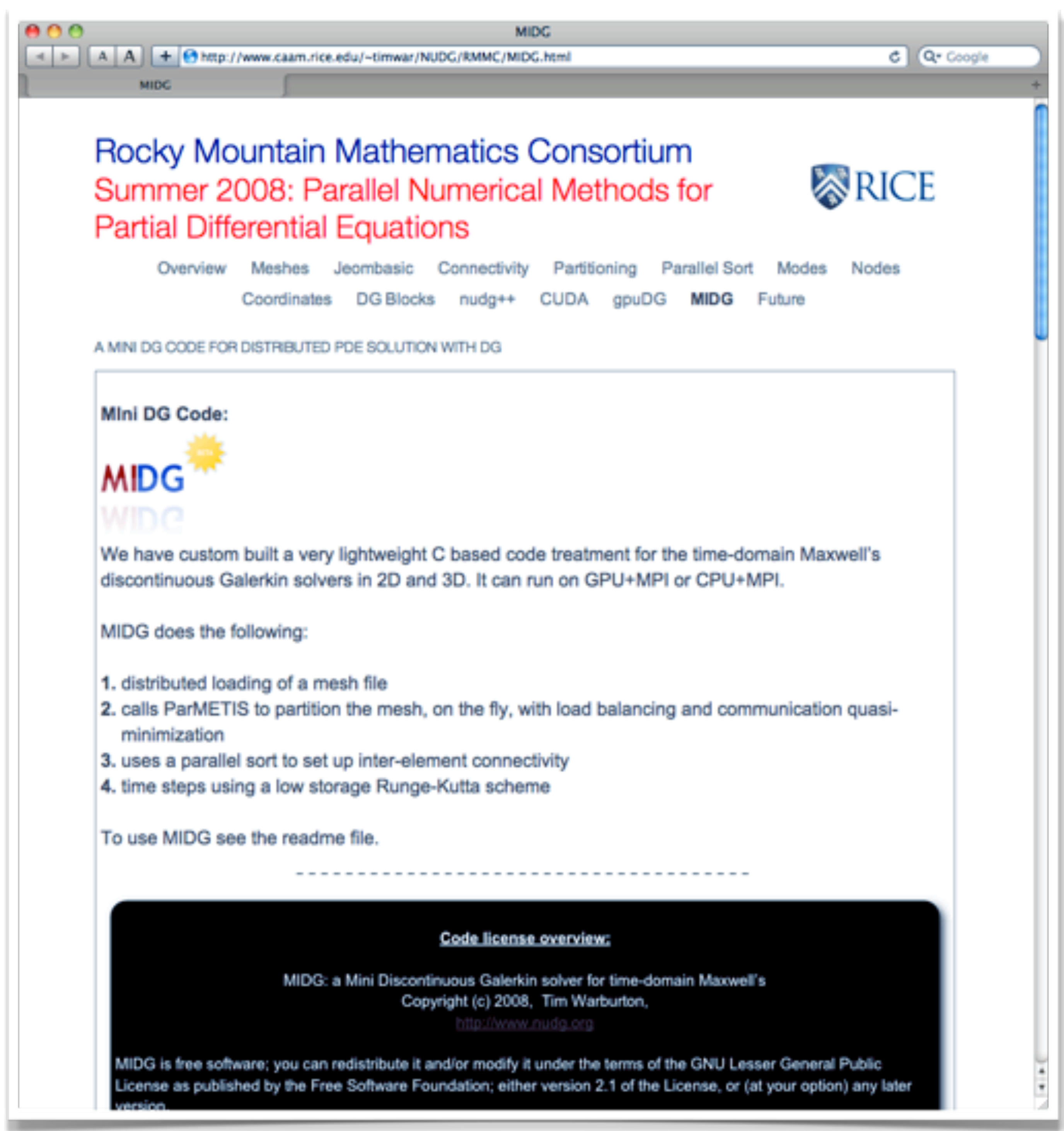
The three CUDA DG kernels are achieving a respectable fraction of peak memory bandwidth (177GB/s).

MiDG Multi-GPU DG Code

We have released a simple DGTD electromagnetic solver:

MIDG implements the high order nodal DGTD method for 2D & 3D electromagnetic cavities on multiple GPUs using CUDA and MPI.

3 KLOC in C.



Download: http://www.caam.rice.edu/~timwar/NUDG/RMMC/MIDG.html

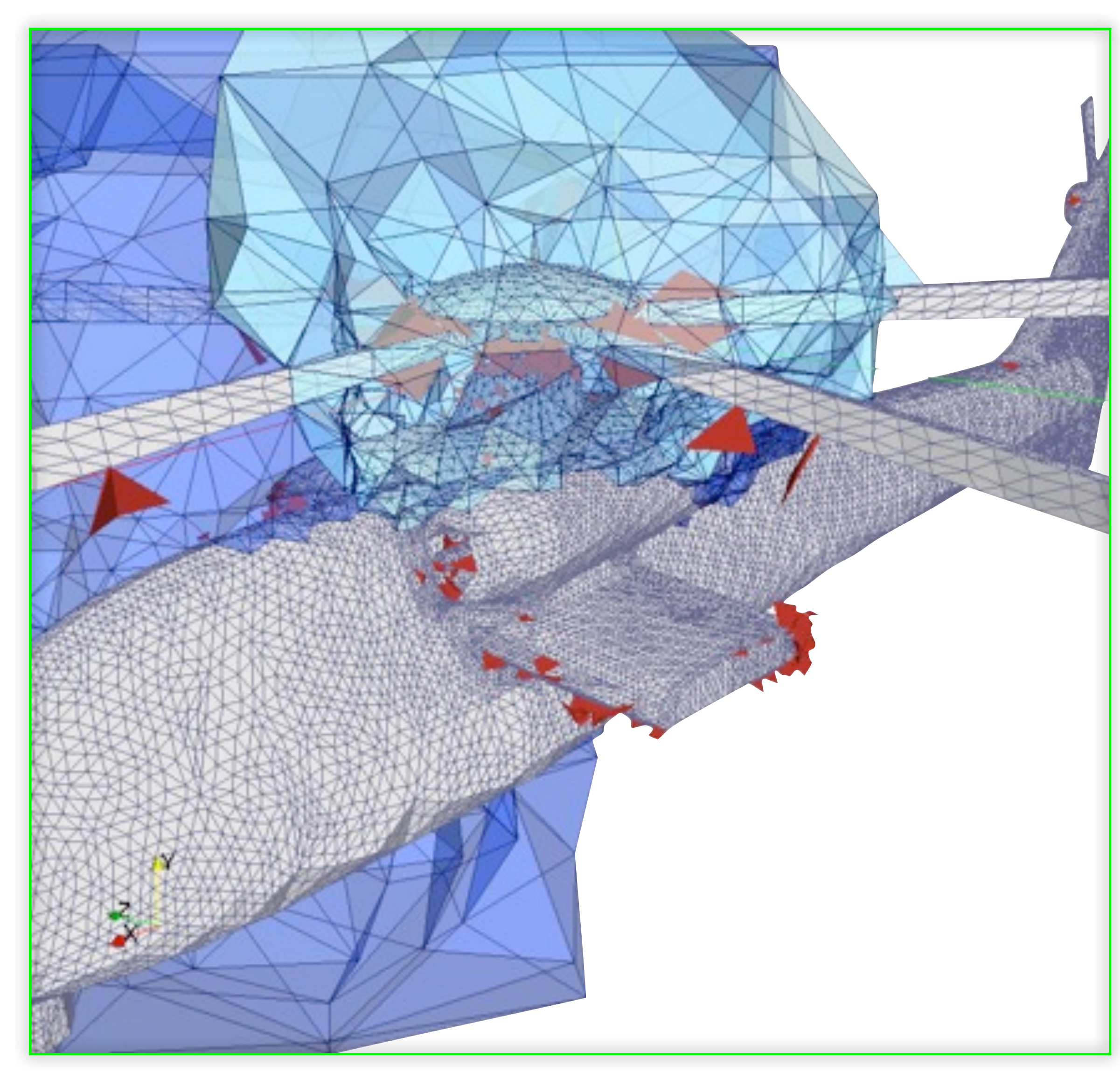
Local dt + Multiple GPUs

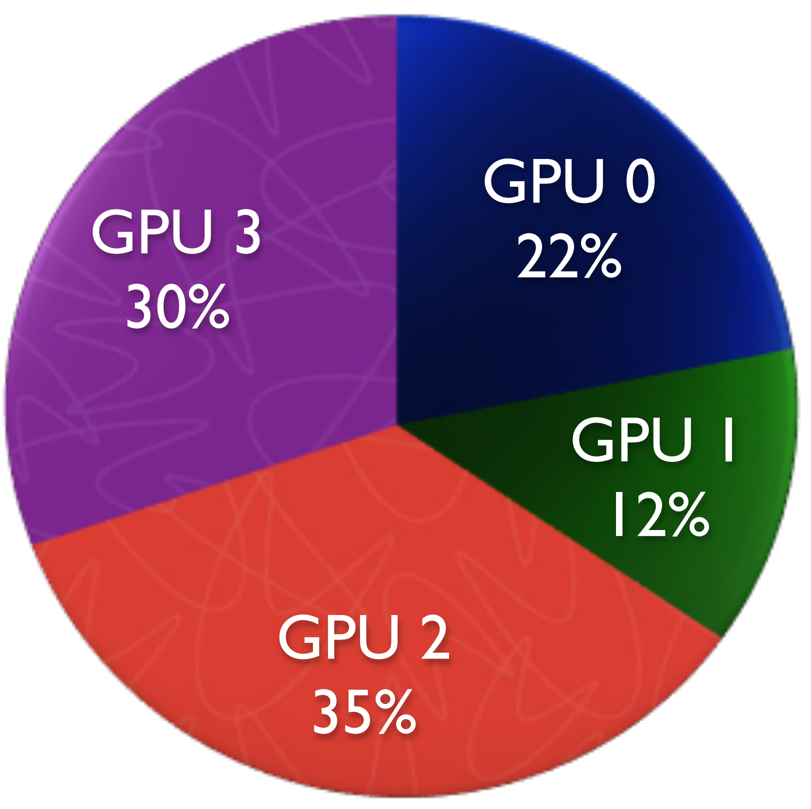
Mesh generation of this model helicopter gives
~ 4% pathologically small tetrahedra

Local time stepping is an absolute necessity.
[local time stepping is not as trivial for SEM or FEM]

Local dt + Multiple GPUs

It is tricky to load-balance and minimize communication for local time-stepping
Approach: use ParMetis with lumped weighted graph to avoid splitting fine clusters

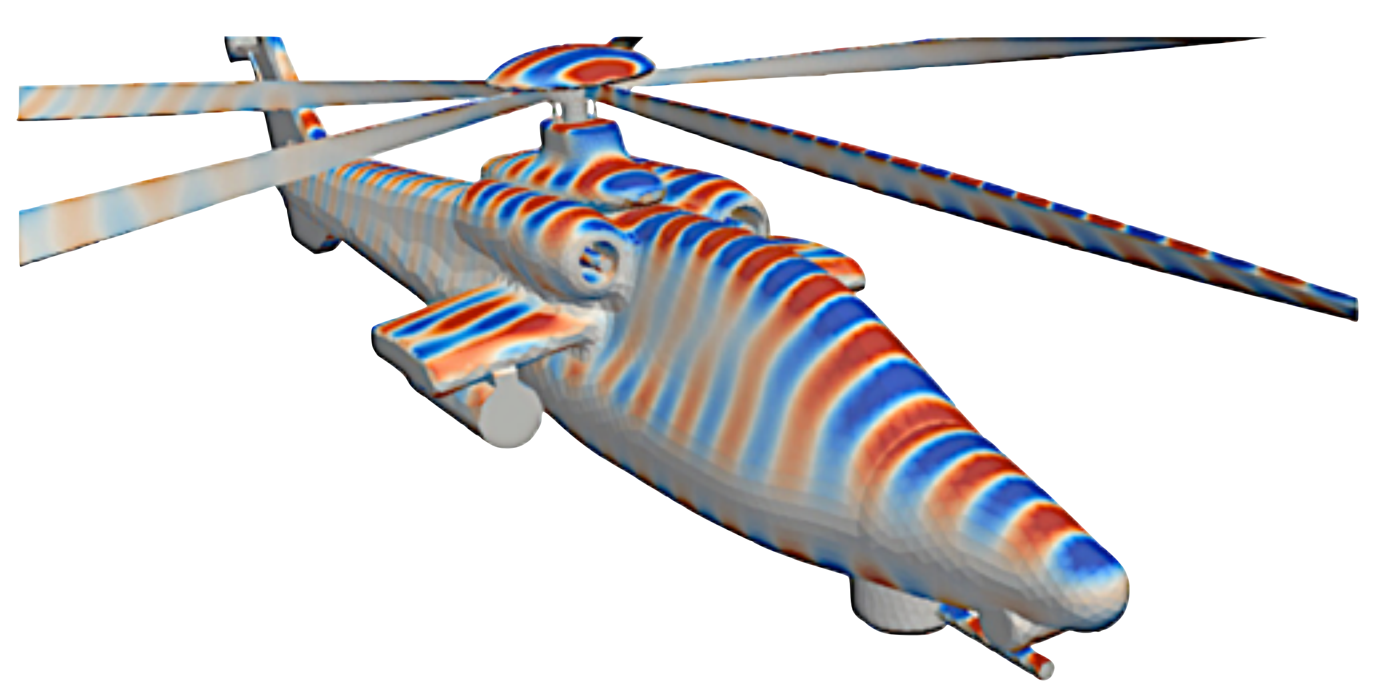




Each GPU owns isolated clusters of fine elements. Work load is balanced.

Local Time Stepping & Multi GPUs

Small elements handled with Gear & Wells' multirate linear multistep method



375K tetrahedral elements
4th order polynomial approximation
78M degrees of freedom

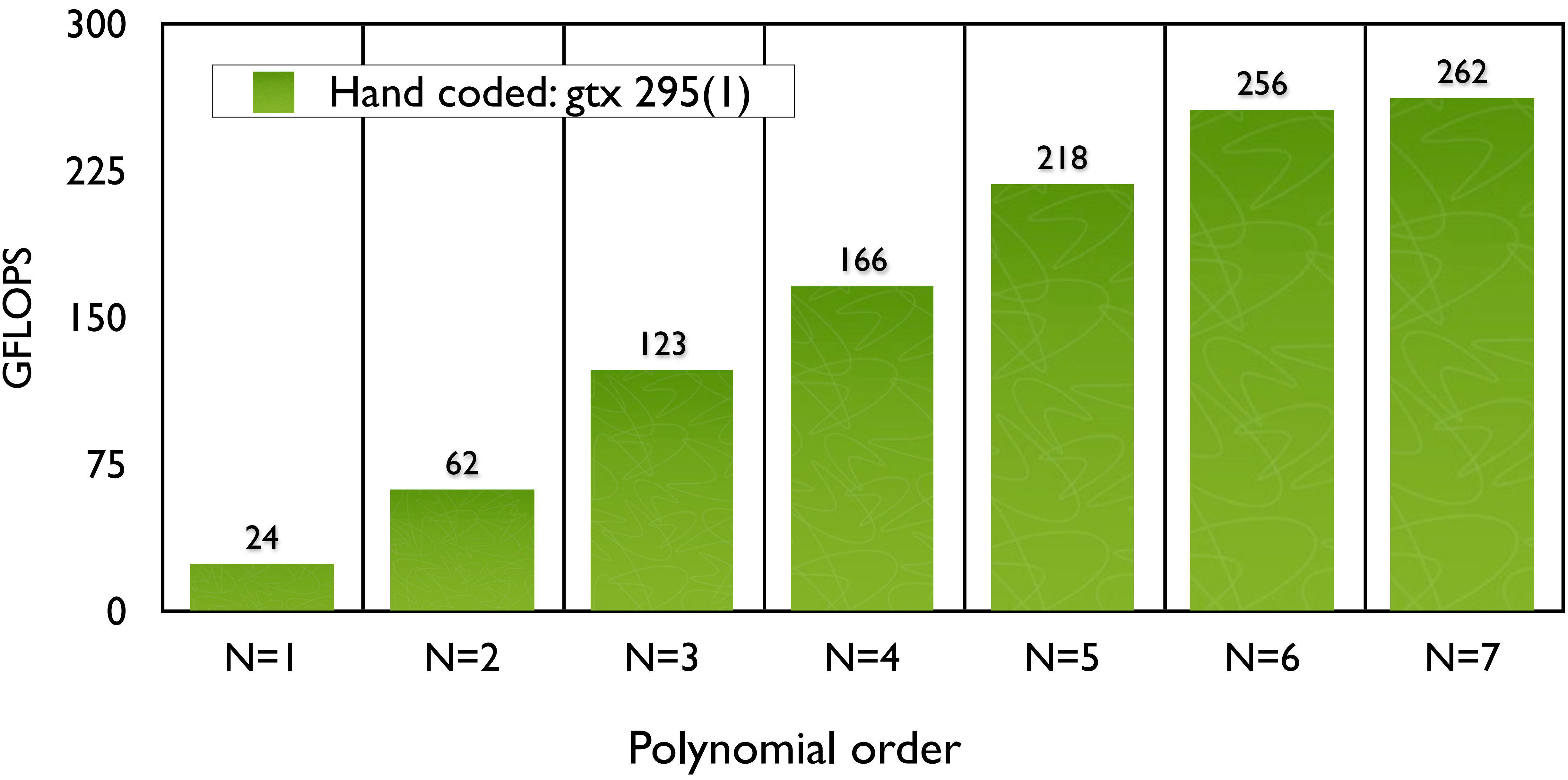
1:8 coarse to fine time step ratio
Weighted graph partitioning by Metis

Method	Platform	Compute Time	Speed Up
Global Adams-Bashforth	4x AMD Opteron 8356 Quad Cores	29h 55m	1
Global Adams-Bashforth	4x NVIDIA Tesla C1060	2h 31	11.88
Two-level AB	4x NVIDIA Tesla C1060	53m	33.87

Nico Gödel, Steffen Schomann, TW, and Markus Clemens, *GPU Accelerated Adams-Bashforth Multirate Discontinuous Galerkin Simulation of High Frequency Electromagnetic Fields*, IEEE Transactions on Magnetics, 2010.

Are we Doing ok ?

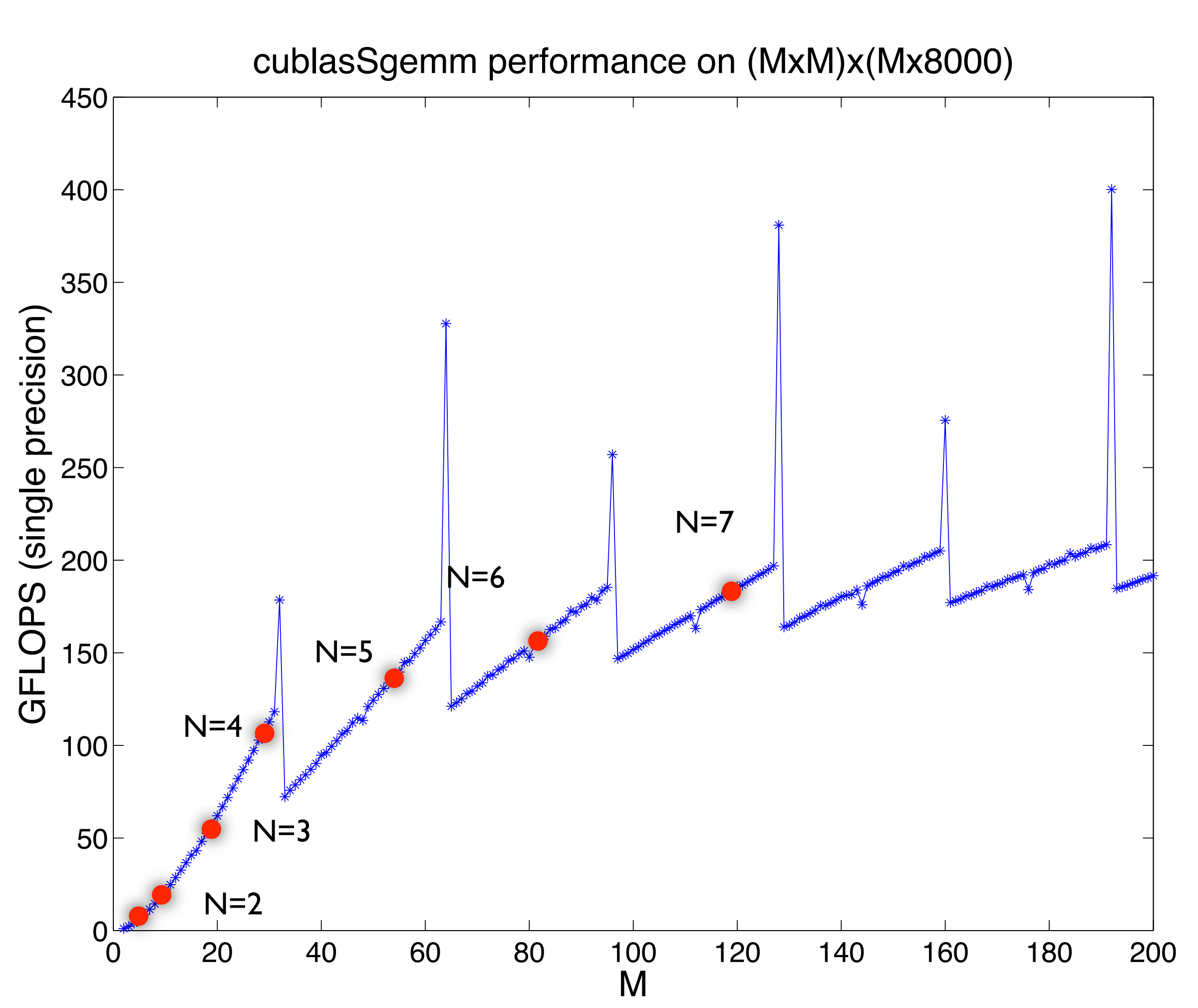
There is still a lingering doubt: how good is the hand coded implementation ?

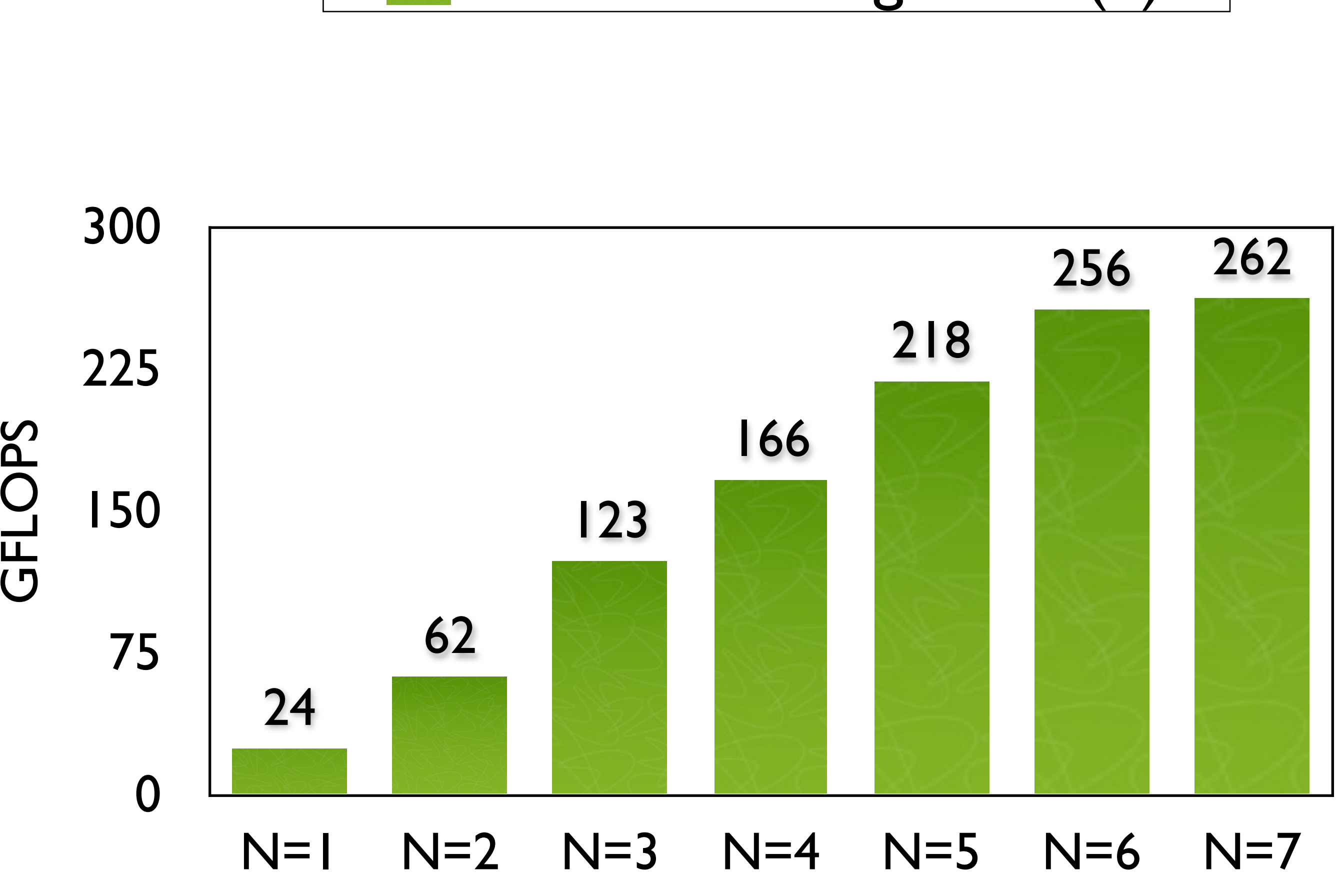


Improved performance at low polynomial order was obtained with Andreas Klöckner's Python DG domain specific language Hedge (built on PyCUDA with autotuning)

Why was CUBLAS Disappointing ?

In each case our hand coded application performance exceeds the most optimistic CUBLAS performance.





Realism: every Level 1,2 CUBLAS call drops performance closer to 30GFLOPS i.e. "Modularity" can cause low performance !

Are we Doing ok ?

I was later in a friendly competition with Andreas Klöckner...

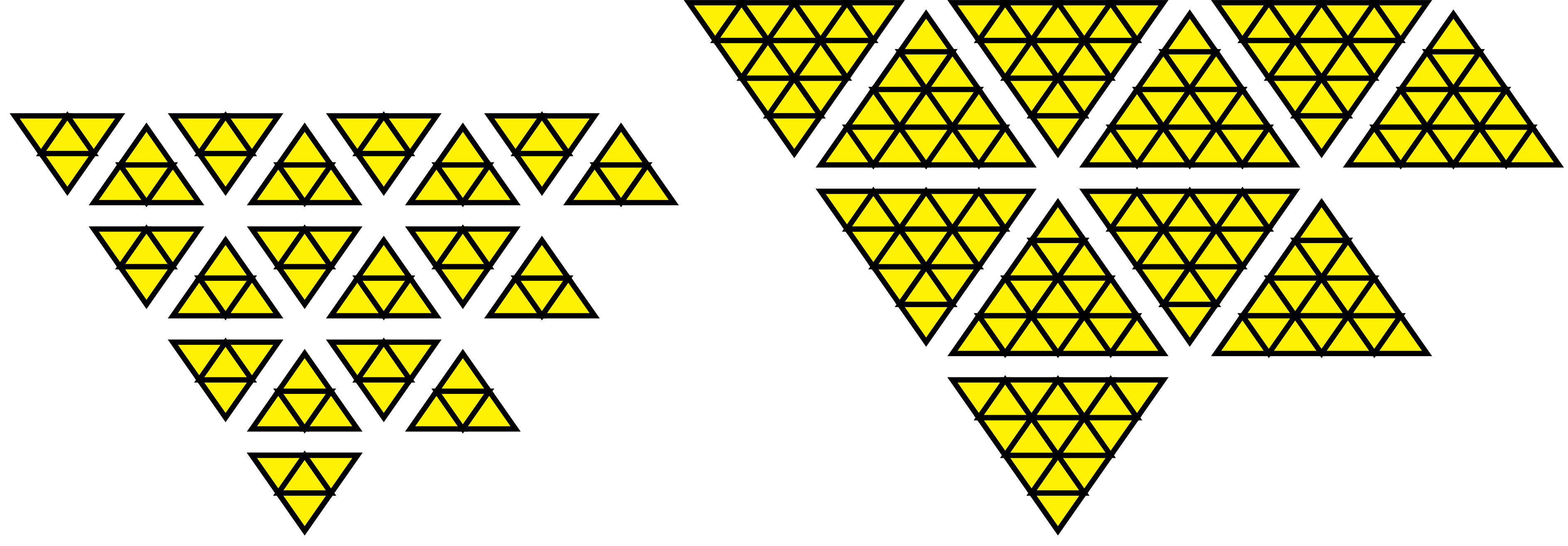
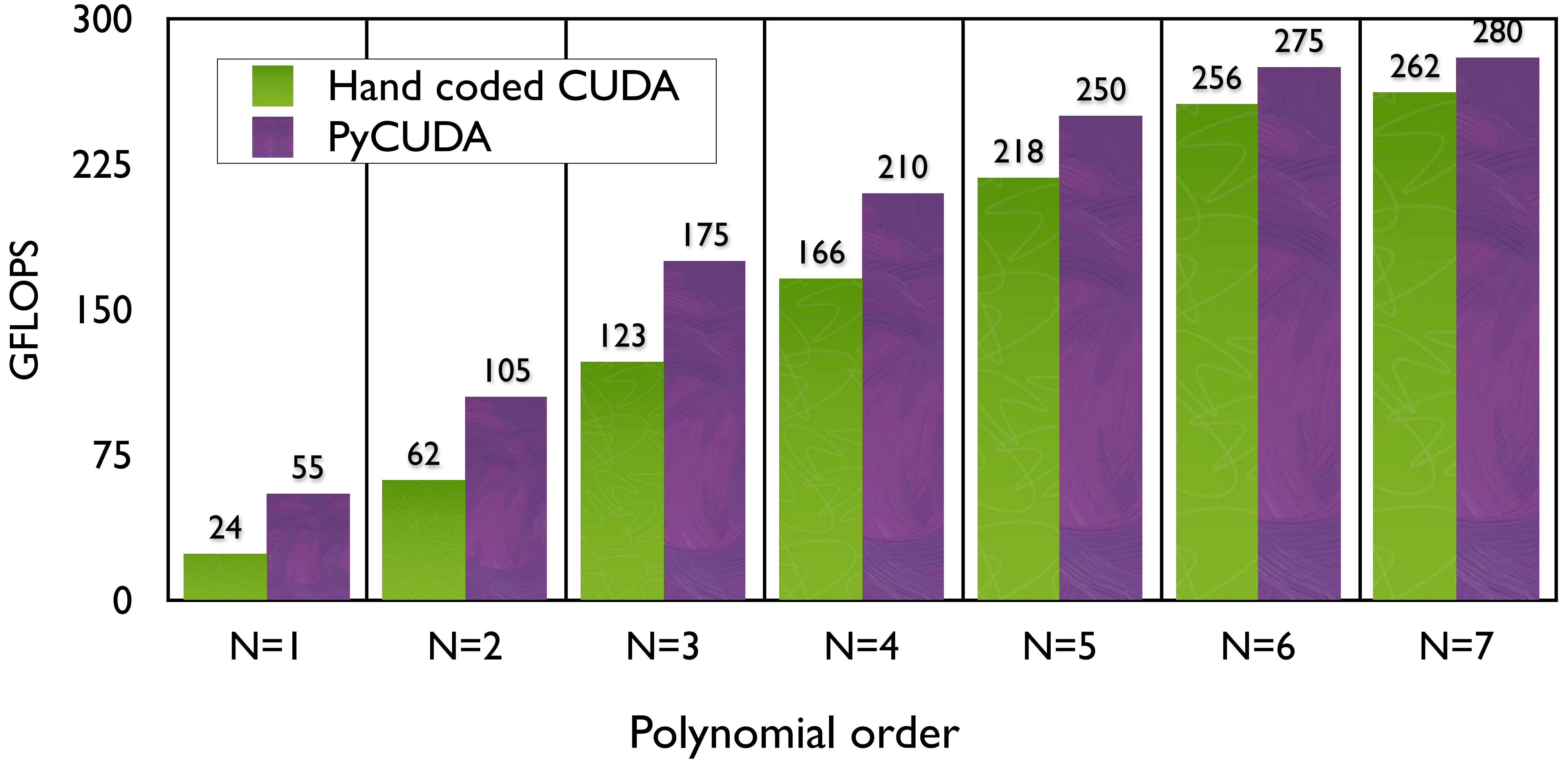


Fig. 1.5 Multiple granularities for inter-element flux computation. Obviously, larger blocks lead to more data reuse as fewer face pairs are split.

Approach: lightweight kernels with auto-tuning to automatically select:
i. the number of elements processed in a thread-block.
ii. the (shared memory) cache blocking strategy in mXm.
iii. run-time code generation in Python (simultaneously creating PyCUDA)

Are we Doing ok ?

The high-order performance is quite similar with two very different approaches:



Substantial improved performance at low polynomial order was obtained with Andreas Klöckner's Python DG domain specific language Hedge.

Slides by Andreas Klöckner:
http://mathema.tician.de/software/pycuda

Scripting for GPUs

Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

$\rightarrow$ complement each other

Scripting for GPUs

Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

$\rightarrow$ complement each other

- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough

Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks (~1000/sec)
  - Scripting fast enough
- Python + CUDA = **PyCUDA**

PyCUDA: Example code

Whetting your appetite

```
1  import pycuda.driver as cuda
2  import pycuda.autoinit
3  import numpy
4
5  a = numpy.random.randn(4,4).astype(numpy.float32)
6  a_gpu = cuda.mem_alloc(a.nbytes)
7  cuda.memcpy_htod(a_gpu, a)
```

[This is `examples/demo.py` in the PyCUDA distribution.]

PyCUDA: Example code

```
1   mod = cuda.SourceModule("""
2        __global__  void  twice( float  *a)
3       {
4          int  idx  = threadIdx.x + threadIdx.y*4;
5          a[idx]  *= 2;
6       }
7       """)
8
9   func  = mod.get_function("twice")
10  func(a_gpu,  block=(4,4,1))
11
12  a_doubled  = numpy.empty_like(a)
13  cuda.memcpy_dtoh(a_doubled, a_gpu)
14  print  a_doubled
15  print  a
```

PyCUDA: Example code

```
1   mod = cuda.SourceModule("""
2     __global__ void twice( float *a)
3     {
4       int idx = threadIdx.x + threadIdx.y*4;
5       a[idx] *= 2;
6     }
7     """)
8
9   func = mod.get_function("twice")
10  func(a_gpu, block=(4,4,1))
11
12  a_doubled = numpy.empty_like(a)
13  cuda.memcpy_dtoh(a_doubled, a_gpu)
14  print a_doubled
15  print a
```

Compute kernel

PyCUDA

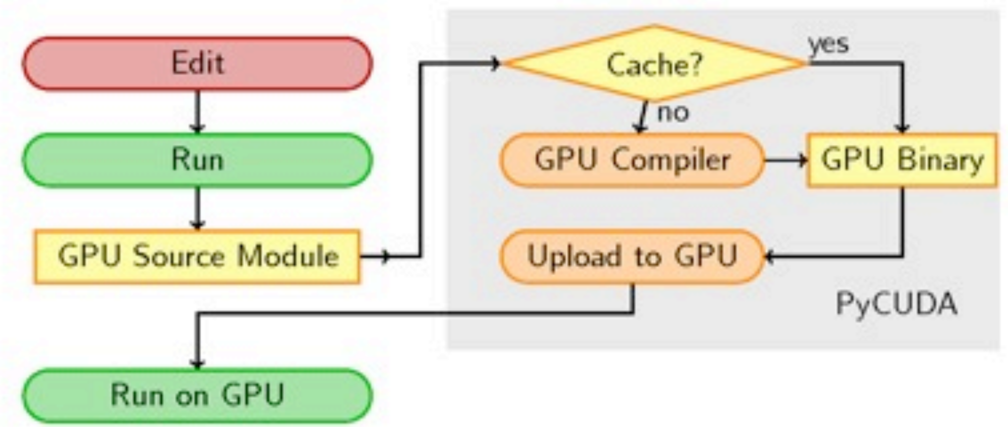
Whetting your appetite, Part II

```
1  import numpy
2  import pycuda.autoinit
3  import pycuda.gpuarray as gpuarray
4
5  a_gpu = gpuarray.to_gpu(
6      numpy.random.randn(4,4).astype(numpy.float32))
7  a_doubled = (2*a_gpu).get()
8  print a_doubled
9  print a_gpu
```

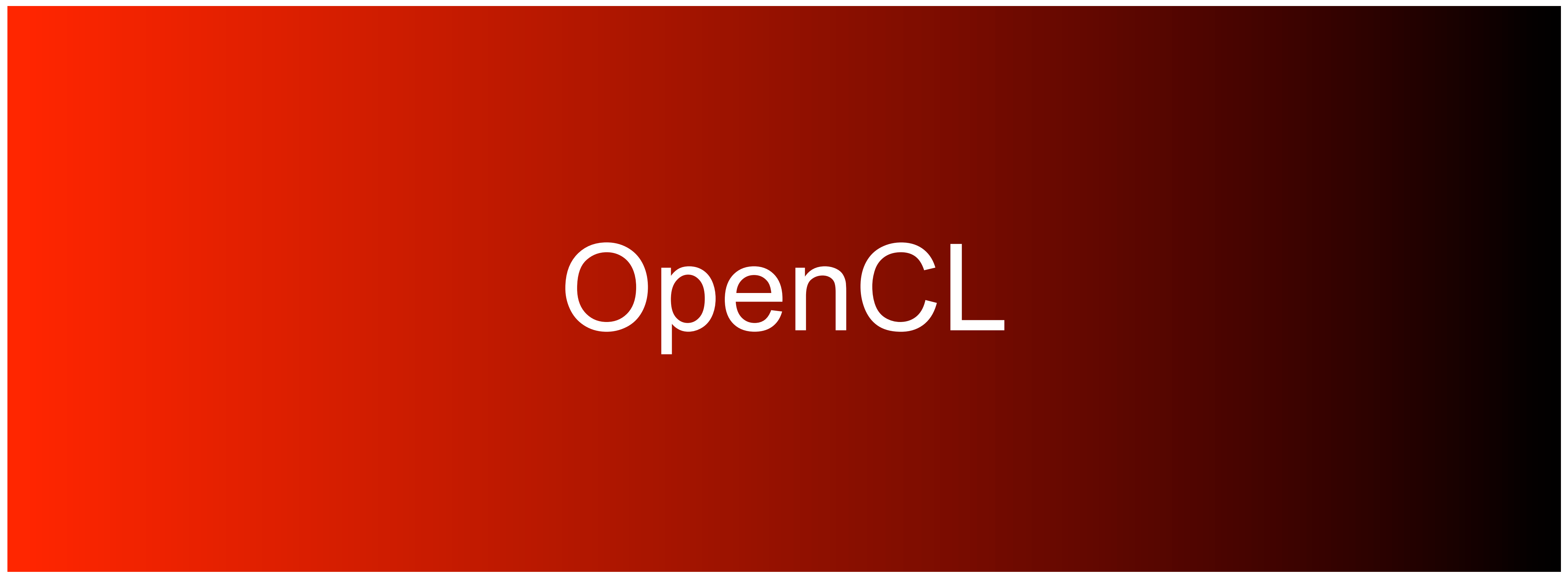
Python + CUDA = PyCUDA

- All of CUDA in a modern scripting language
- Full Documentation
- Free, open source (MIT)
- Also: PyOpenCL

- CUDA C Code = Strings
- Generate Code Easily
  - Automated Tuning
- Batteries included: GPU Arrays, RNG, …
- Integration: `numpy` arrays, Plotting, Optimization, …



http://mathema.tician.de/software/pycuda


OpenCL

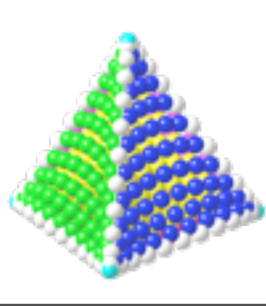
RICE

OpenCL: cross-platform

GFLOPS

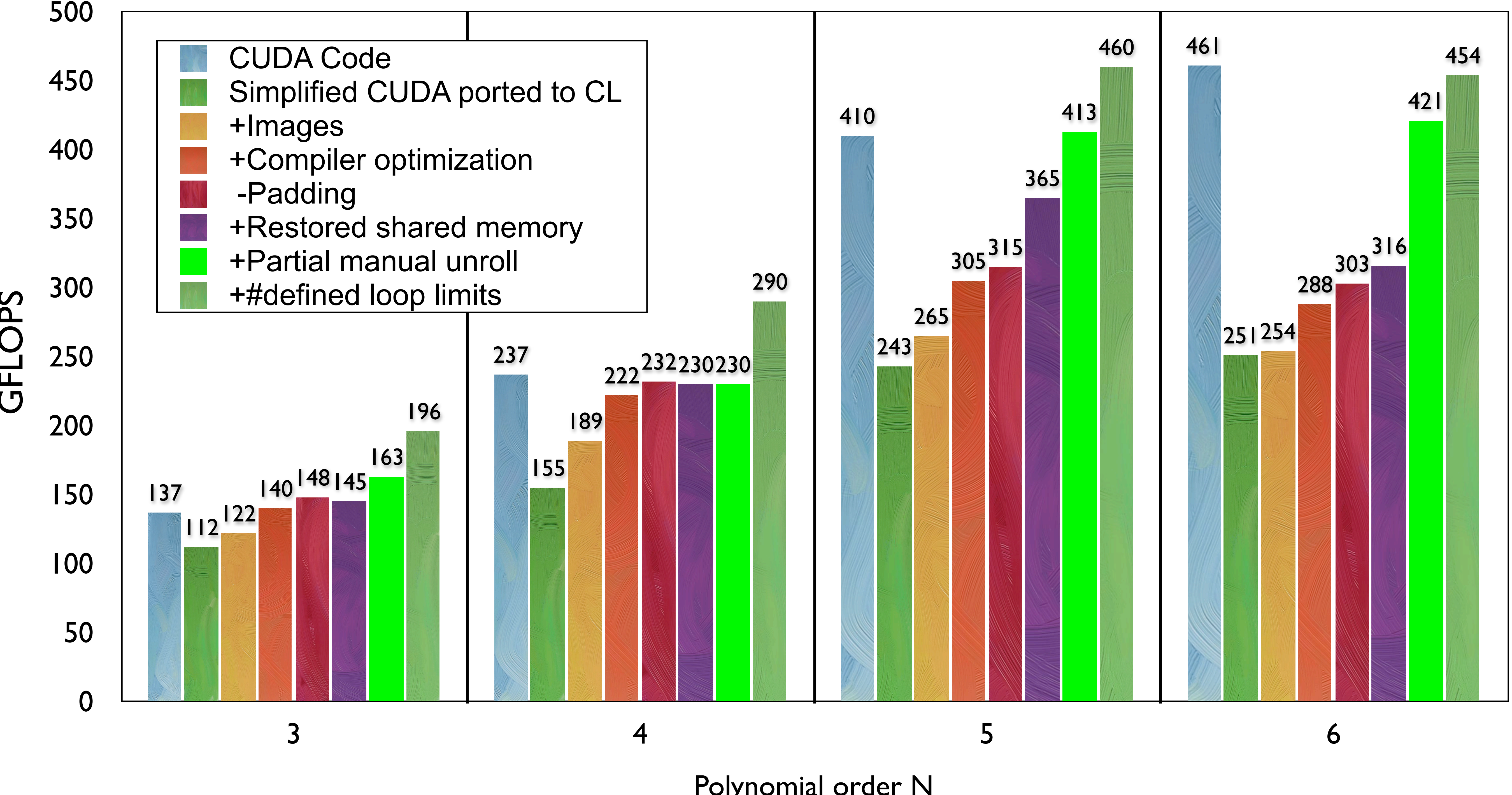
	Tesla C870	Tesla C1060	AMD 5970 (1 chip)	Tesla C2050	GTX 480
N=3	45	115	107	137	196
N=4	85	144	199	237	290
N=5	125	197	332	328	460
N=6	125	214	0	332	454

We were pleasantly surprised that the code ported easily from CUDA to OpenCL and maintained performance on GPUs from AMD and NVIDIA



OpenCL: Kernel Hand Tuning Process

Porting our CUDA code to OpenCL we tracked the performance on fermi.



For higher order individual kernels exceed 600GFLOPS (~50% peak)

ptx Code

```
tex.2d.v4.f32.f32 { %f122, %f123, %f124, %f125 },
add.s32         %r17, %r16, 1;
ld.shared.f32   %f47, [%r15+1120];
ld.shared.f32   %f48, [%r15+896];
ld.shared.f32   %f49, [%r15+672];
ld.shared.f32   %f50, [%r15+448];
ld.shared.f32   %f51, [%r15+224];
ld.shared.f32   %f55, [%r15];
add.s32         %r18, %r15, 4;
mad.f32         %f37, %f124, %f47, %f19;
mad.f32         %f36, %f123, %f47, %f18;
mad.f32         %f35, %f122, %f47, %f17;
mad.f32         %f34, %f124, %f48, %f16;
mad.f32         %f33, %f123, %f48, %f15;
mad.f32         %f32, %f122, %f48, %f14;
mad.f32         %f31, %f124, %f49, %f13;
mad.f32         %f30, %f123, %f49, %f12;
mad.f32         %f29, %f122, %f49, %f11;
mad.f32         %f28, %f124, %f50, %f10;
mad.f32         %f27, %f123, %f50, %f9;
mad.f32         %f26, %f122, %f50, %f8;
mad.f32         %f25, %f124, %f51, %f7;
mad.f32         %f24, %f123, %f51, %f6;
mad.f32         %f23, %f122, %f51, %f5;
mad.f32         %f22, %f124, %f55, %f2;
mad.f32         %f21, %f123, %f55, %f3;
```

OpenCL

```
tex.1d.v4.f32.s32 {%f54,%f55,%f56,%f57};
.loc    14      325     0
mov.f32         %f58, %f54;
mov.f32         %f59, %f55;
mov.f32         %f60, %f56;
.loc    14      329     0
ld.shared.f32   %f61, [%rd7+0];
fma.rn.f32      %f53, %f61, %f58, %f53;
fma.rn.f32      %f52, %f61, %f59, %f52;
fma.rn.f32      %f51, %f61, %f60, %f51;
.loc    14      330     0
ld.shared.f32   %f61, [%rd7+224];
fma.rn.f32      %f50, %f61, %f58, %f50;
fma.rn.f32      %f49, %f61, %f59, %f49;
fma.rn.f32      %f48, %f61, %f60, %f48;
.loc    14      331     0
ld.shared.f32   %f61, [%rd7+448];
fma.rn.f32      %f47, %f61, %f58, %f47;
fma.rn.f32      %f46, %f61, %f59, %f46;
fma.rn.f32      %f45, %f61, %f60, %f45;
.loc    14      333     0
ld.shared.f32   %f61, [%rd7+672];
fma.rn.f32      %f44, %f61, %f58, %f44;
fma.rn.f32      %f43, %f61, %f59, %f43;
fma.rn.f32      %f42, %f61, %f60, %f42;
....
```

CUDA

The intermediate ptx "assembler" code reveals some differences in the compiler output

PyOpenCL

- There is also a Python API for OpenCL.
- This provides a very clean and intuitive interface.
- For further details see: pyopencl.pdf on the K2I summer institute resources web page.
- Web pages:
  - http://mathema.tician.de/software/pyopencl
  - http://documen.tician.de/pyopencl/

PyOpenCL: Example

```
import pyopencl as cl

import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()

print la.norm(a_plus_b - (a+b))
```

Source Code Modification Based Profiling in OpenCL

As a side project we used the just in time compilation feature of OpenCL to perform source code modification based profiling.

Material sourced from:
Paulius Micikevicius' Presentation on Source Code Analysis

OpenCL Profiling: source modification

Analysis with Modified Source Code

- **Time memory-only and math-only versions of the kernel**
  - Easier for codes that don't have data-dependent control-flow or addressing
  - Gives you good estimates for:
    - Time spent accessing memory
    - Time spent in executing instructions
- **Comparing the times for modified kernels**
  - Helps decide whether the kernel is mem or math bound
  - Shows how well memory operations are overlapped with arithmetic
    - Compare the sum of mem-only and math-only times to full-kernel time

© NVIDIA 2010

9

From: Paulius Micikevicius Presentation on Source Code Analysis

OpenCL Profiling: source modification

Some Example Scenarios

time

	mem	math	full

Memory-bound

Good mem-math overlap: latency not a problem

(assuming memory throughput is not low compared to HW theory)

Math-bound

Good mem-math overlap: latency not a problem

(assuming instruction throughput is not low compared to HW theory)

Balanced

Good mem-math overlap: latency not a problem

(assuming memory/instr throughput is not low compared to HW theory)

Memory and latency bound

Poor mem-math overlap: latency is a problem

NVIDIA 2010

13

From: Paulius Micikevicius Presentation on Source Code Analysis

OpenCL Profiling: source modification

Source Modification

- **Memory-only:**
  - Remove as much arithmetic as possible
    - Without changing access pattern
    - Use the profiler to verify that load/store instruction count is the same
- **Store-only:**
  - Also remove the loads
- **Math-only:**
  - Remove global memory accesses
  - Need to trick the compiler:
    - Compiler throws away all code that it detects as not contributing to stores
    - Put stores inside conditionals that always evaluate to false
      - Condition should depend on the value about to be stored (prevents other optimizations)
      - Condition should not be known to the compiler

From: Paulius Micikevicius Presentation on Source Code Analysis

OpenCL Profiling: source modification

Source Modification for Math-only

```
__global__ void fwd_3D( ..., int flag)
{
    ...
  value = temp + coeff * vsq;
  if( 1 == value * flag )
    g_output[out_idx] = value;
}
```

If you compare only the flag, the compiler may move the computation into the conditional as well

15

From: Paulius Micikevicius Presentation on Source Code Analysis

OpenCL Profiling: source modification

Source Modification and Occupancy

- **Removing pieces of code is likely to affect register count**
  - This could increase occupancy, skewing the results
  - See slide 27 to see how that could affect throughput
- **Make sure to keep the same occupancy**
  - Check the occupancy with profiler before modifications
  - After modifications, if necessary add shared memory to match the unmodified kernel's occupancy

```
kernel<<< grid, block, smem, ...>>>(...)
```

From: Paulius Micikevicius Presentation on Source Code Analysis

OpenCL Profiling: source modification

We wrote a source code analysis tool that:

i. labels each line of a kernel according to the type of actions

a. math
b. smem memory read/write
c. device memory read/write
d. control flow
e. integer or floating point arithmetic

ii. Math: Then we scan the kernel source code expression by expression and automatically generate a kernel with individual math operation turned off. We then build the OpenCL kernel in each case, and time the kernels.

iii. Memory: we repeat the sweep selectively turning off each type of memory access where possible [without radically altering the control flow]

This is all done automatically and we output a html heat map of the source code.

Fine Grain Kernel Profiling

Q: is it possible to measure fine grain profiling info for GPGPU kernels by automatic code dissection ?

A: maybe.

So far we are able to automatically detect "interesting" expressions and generate/compile/benchmark kernels by selectively switching expressions on and off.

[extending ideas from Micikevicius]

Red is bad

```
__kernel void MaxwellsVolume2d(int K,
                 read_only __global float *g_Q,
                 __global float *g_rhsQ,
                 read_only image2d_t t_DrDs,
                 read_only __global float *g_vgeo,
                 __local float *s_Q,
                 __local float *s_vgeo){

#define p_Np (((p_N+1)*(p_N+2))/2)

  const int p_Nfields = 3;

  const int BSIZE = (4*((p_Np+3)/4));

  /* LOCKED IN to using Np threads per block */
  const int n = get_local_id(0);
  const int k = get_group_id(0);

  float dHxdr=0; float dHxds=0;
  float dHydr=0; float dHyds=0;
  float dEzdr=0; float dEzds=0;
  float Q;

  if(k>=K) return;

  int m = n+k*p_Nfields*BSIZE;
  int id = n;

  s_Q[id] = g_Q[m]; m+=BSIZE; id+=BSIZE;
  s_Q[id] = g_Q[m]; m+=BSIZE; id+=BSIZE;
  s_Q[id] = g_Q[m];
```

```
  dHxdr=0; dHxds=0;
  dHydr=0; dHyds=0;
  dEzdr=0; dEzds=0;

  sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE;
  for(m=0;p_Np-m;){
    float4 D= read_imagef(t_DrDs, sampler, (int2)(n,m));

    id = m;
    Q = s_Q[id]; dHxdr += D.x*Q; dHxds += D.y*Q; id += BSIZE;
    Q = s_Q[id]; dHydr += D.x*Q; dHyds += D.y*Q; id += BSIZE;
    Q = s_Q[id]; dEzdr += D.x*Q; dEzds += D.y*Q;

    ++m;

  }

  m = n+p_Nfields*BSIZE*k;

  const float drdx= s_vgeo[0];
  const float drdy= s_vgeo[1];
  const float dsdx= s_vgeo[2];
  const float dsdy= s_vgeo[3];

  g_rhsQ[m] = -(drdy*dEzdr+dsdy*dEzds); m += BSIZE;
  g_rhsQ[m] =  (drdx*dEzdr+dsdx*dEzds); m += BSIZE;
  g_rhsQ[m] =  (drdx*dHydr+dsdx*dHyds - drdy*dHxdr-dsdy*dHxds);

}
```

Tentatively we can spot intensive expressions (soon groups of expressions)
More reliable information can be gleaned from the OpenCL profiler.

Red is bad

```
__kernel void MaxwellsVolume2d(int K,
                  read_only __global float *g_Q,
                  __global float *g_rhsQ,
                  read_only image2d_t t_DrDs,
                  read_only __global float *g_vgeo,
                  __local  float *s_Q,
                  __local  float *s_vgeo){

#define p_Np  (((p_N+1)*(p_N+2))/2)

  const int p_Nfields = 3;

  const int BSIZE = (4*((p_Np+3)/4));

  /* LOCKED IN to using Np threads per block */
  const int n = get_local_id(0);
  const int k = get_group_id(0);

  float dHxdr=0;  float dHxds=0;
  float dHydr=0;  float dHyds=0;
  float dEzdr=0;  float dEzds=0;
  float Q;

  if(k>=K) return;

  int m = n+k*p_Nfields*BSIZE;
  int id = n;

  s_Q[id] = g_Q[m];  m+=BSIZE; id+=
  s_Q[id] = g_Q[m];  m+=BSIZE; id+=BS
  s_Q[id] = g_Q[m];
```

```
dHxdr=0;  dHxds=0;
dHydr=0;  dHyds=0;
dEzdr=0;  dEzds=0;

sampler_t sample                    ZED_COORDS_FALSE;
for(m=0;p_Np
  float4 D                 sampler, (int2)(n,m));

  id
              += D.x*Q;  dHxds += D.y*Q;  id += BSIZE;
             r += D.x*Q;  dHyds += D.y*Q;  id += BSIZE;
            Ezdr += D.x*Q;  dEzds += D.y*Q;

  = n+p_Nfields*BSIZE*k;

  const float drdx= s_vgeo[0];
  const float drdy= s_vgeo[1];
  const float dsdx= s_vgeo[2];
  const float dsdy= s_vgeo[3];

  g_rhsQ[m] = -(drdy*dEzdr+dsdy*dEzds);  m += BSIZE;
  g_rhsQ[m] =  (drdx*dEzdr+dsdx*dEzds);  m += BSIZE;
  g_rhsQ[m] =  (drdx*dHydr+dsdx*dHyds - drdy*dHxdr-dsdy*dHxds);

}
```

Work in progress

Tentatively we can spot intensive expressions (soon groups of expressions)
More reliable information can be gleaned from the OpenCL profiler.

OpenCL or CUDA: DGTD

- DGTD achieves a reasonable percentage of peak with both OpenCL and CUDA on NVIDIA cards.
- DGTD achieves a reasonable percentage of peak with both PyOpenCL and PyCUDA on NVIDIA cards.
- There is a definite bump in performance with OpenCL but this can likely be achieved with CUDA.
- CUDA is currently limited to x86 and NVIDIA cards.
- OpenCL is infinitely faster than CUDA on AMD GPUs.
- PyOpenCL is a to get similar performance for DGTD via auto-tuning.

RICE

OpenCL or CUDA: May's law

" First, a disappointment. It is widely accepted that hardware efficiency doubles ever 18 months, following Moore's law. Let me now introduce you to May's law:

software efficiency halves every 18 months, compensating Moore's law!

It's not clear what has caused this, but the tendency to add features, programming using copy-paste techniques, and programming by 'debugging the null-program' - starting with a debugger and an empty screen and debugging interactively until the desired program emerges - have probably all contributed. "

David May
CSP, occam, and Transputers, Communicating Sequential Processes, pp. 75-84, 2004.

OpenCL or CUDA: my view

Long term:

- Fingers crossed compilers make all this irrelevant.

Short/medium term:

- Shielding "domain specialists" from parallelism may add complexity
  => use very thin wrappers on the HOST.

- CUDA/OpenCL force us to address task level parallelism and data locality
  => encourages programmer focus and simplicity on the DEVICE.

- OpenCL is a nearly uniform way to program devices
  => obliges us to code with at least simple auto-tuning in mind.

- New devices will be heterogeneous
  => need a general interface.

- **We are migrating our "legacy" CUDA codes to OpenCL and all new codes are OpenCL or PyOpenCL.**

Low-Storage Curvilinear Discontinuous Galerkin Time-Domain Method

Question: how can we handle curvilinear elements without saturating memory capacity and bandwidth

We adopted the "flops-for-free philosophy"

Isoparameteric Transform Jacobians

$$J = \left| \frac{\partial \mathbf{x}}{\partial r} \times \frac{\partial \mathbf{x}}{\partial s} \right| \text{ appears in inner products: } (u,v)_{D^k} = \iint_{\hat{D}} u(r,s)v(r,s)J(r,s)\,dr\,ds$$

FEM Mesh

Curvilinear Mesh

Piecewise polynomial determinant of the Jacobian plotted vertically

Discrete DG Equations

Testing with a basis for the function spaces

$$\sum_m (\mu\phi_n, \phi_m)_{D_k} \frac{dH_{mk}}{dt} = \sum_m (\phi_n, \nabla \times \psi_m)_{D_k} E_{mk} \quad + \left(\phi_n, n \times \left(E^* - E\right)\right)_{\partial D_k}$$

$$\sum_m (\varepsilon\psi_n, \psi_m)_{D_k} \frac{dE_{mk}}{dt} = -\sum_m (\nabla \times \psi_n, \phi_m)_{D_k} H_{mk} \quad - \left(\psi, n \times H^*\right)_{\partial D_k}$$

for $n = 1,\ldots,N_p$, k=1,...,K.

In matrix form we have

$$\mathbf{M}_\mu \frac{d\mathbf{H}}{dt} = \mathbf{CE} \quad + \text{fluxes}$$

$$\mathbf{M}_\varepsilon \frac{d\mathbf{E}}{dt} = -\mathbf{C}^T\mathbf{H} \quad - \text{fluxes}$$

The mass matrices are block diagonal with dense blocks.

Standard DG Mass Matrices

Unfortunately the local mass matrices are unique to each element with curvature or non-constant coefficients.

$$\mathbf{M}_{\mu} \frac{d\mathbf{H}}{dt} = \mathbf{CE} \quad + \text{fluxes}$$

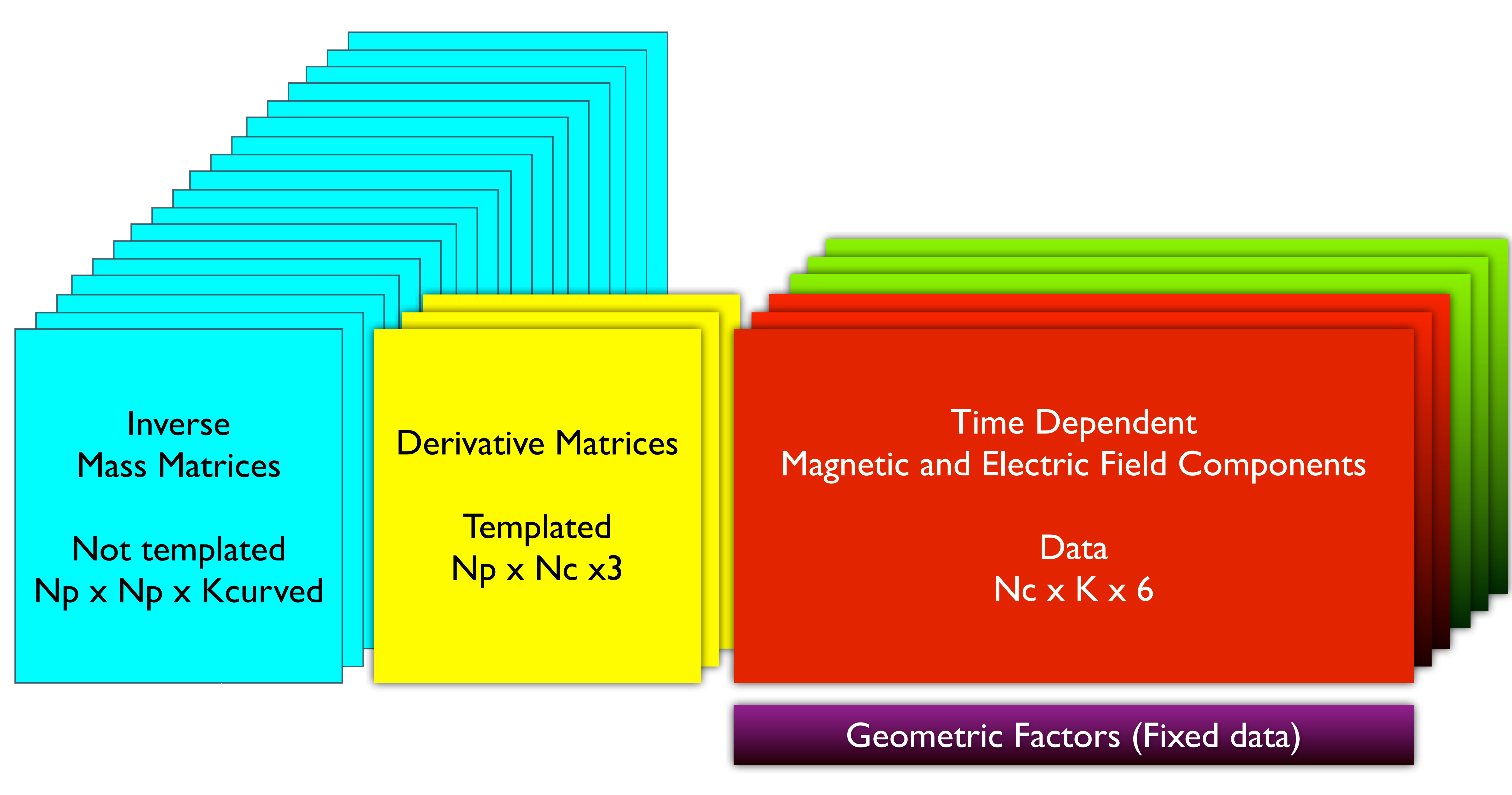
$$\mathbf{M}_{\varepsilon} \frac{d\mathbf{E}}{dt} = -\mathbf{C}^{T}\mathbf{H} \quad - \text{fluxes}$$

For a specific element's mass matrix:

$$\left(M_{\varepsilon}^{k}\right)_{nm} = \left(\varepsilon\phi_n, \phi_m\right)_{D_k} = \int_{D_k} \varepsilon\phi_n\phi_m = \int_{\hat{D}} \varepsilon\phi_n\phi_m J$$

Where the electric permittivity and the determinant of the coordinate transform Jacobian matrix, J, can be non-constant.

Curvilinear Matrix Structures



Each curved element requires its own mass matrix, occupying expensive memory and bandwidth

Geometry Free Mass Matrices

The geometric dependence of the mass matrices...

$$\left(M_{\varepsilon}^{k}\right)_{nm} = \int_{\hat{D}} \varepsilon \phi_n \phi_m J_k$$

.. suggests local elemental test and trial spaces of the form ...

$$V_k = \left\{ \phi : \phi = \frac{\tilde{\phi}}{\sqrt{J_k}} \text{ for some } \tilde{\phi} \in P^N\left(\hat{D}\right) \right\}$$

Adopting a basis for the numerator space P^N we see that the resulting mass matrix is independent of the element.

$$\left(M_{\varepsilon}^{k}\right)_{nm} = \int_{\hat{D}} \varepsilon \phi_n \phi_m J_k = \int_{\hat{D}} \varepsilon \frac{\tilde{\phi}_n}{\sqrt{J_k}} \frac{\tilde{\phi}_m}{\sqrt{J_k}} J_k = \int_{\hat{D}} \varepsilon \tilde{\phi}_n \tilde{\phi}_m$$

Low Storage Curvilinear DG

The variational equation becomes:

$$0 = \left(\tilde{\phi}, \frac{\partial \mu \tilde{H}}{\partial t}\right)_{\hat{D}} + \left(\tilde{\phi}, \nabla \times \tilde{E}\right)_{\hat{D}} + \left(\frac{\tilde{\phi}}{\sqrt{J_k}}, n \times \left(E^* - E\right)\right)_{\partial D_k} - \left(\tilde{\phi}, \frac{\nabla J_k}{2J_k} \times \tilde{E}\right)_{\hat{D}}$$

$$0 = \underbrace{\left(\tilde{\psi}, \frac{\partial \varepsilon \tilde{E}}{\partial t}\right)_{\hat{D}} - \left(\nabla \times \tilde{\psi}, \tilde{H}\right)_{\hat{D}}}_{\text{Maxwell's equations on reference element}} \underbrace{- \left(\frac{\tilde{\psi}}{\sqrt{J_k}}, n \times H^*\right)_{\partial D_k}}_{\text{Distributional derivative contribution}} \underbrace{+ \left(\tilde{\psi}, \frac{\nabla J_k}{2J_k} \times \tilde{H}\right)_{\hat{D}}}_{\text{Transform terms}}$$

Semi-discrete stability is automatic:

$$\frac{d}{dt}\left\{\left\|\frac{\tilde{H}}{\sqrt{J}}\right\|^2 + \left\|\frac{\tilde{E}}{\sqrt{J}}\right\|^2\right\} = -\frac{1}{2}\sum_{k=1}^{k=K}\sum_{f=1}^{4}\sum_{i=1}^{i=N_c} w_i J_i^{s,k,f}\left(\left|n \times \left[\frac{\tilde{H}}{\sqrt{J_k}}\right]\right|^2 + \left|n \times \left[\frac{\tilde{E}}{\sqrt{J_k}}\right]\right|^2\right)\Bigg|_{x_i^{k,f}}$$

No need for custom mass matrices for the price of extra FLOPS

2D Convergence Test

Resonant mode of concentric cylinder cavity

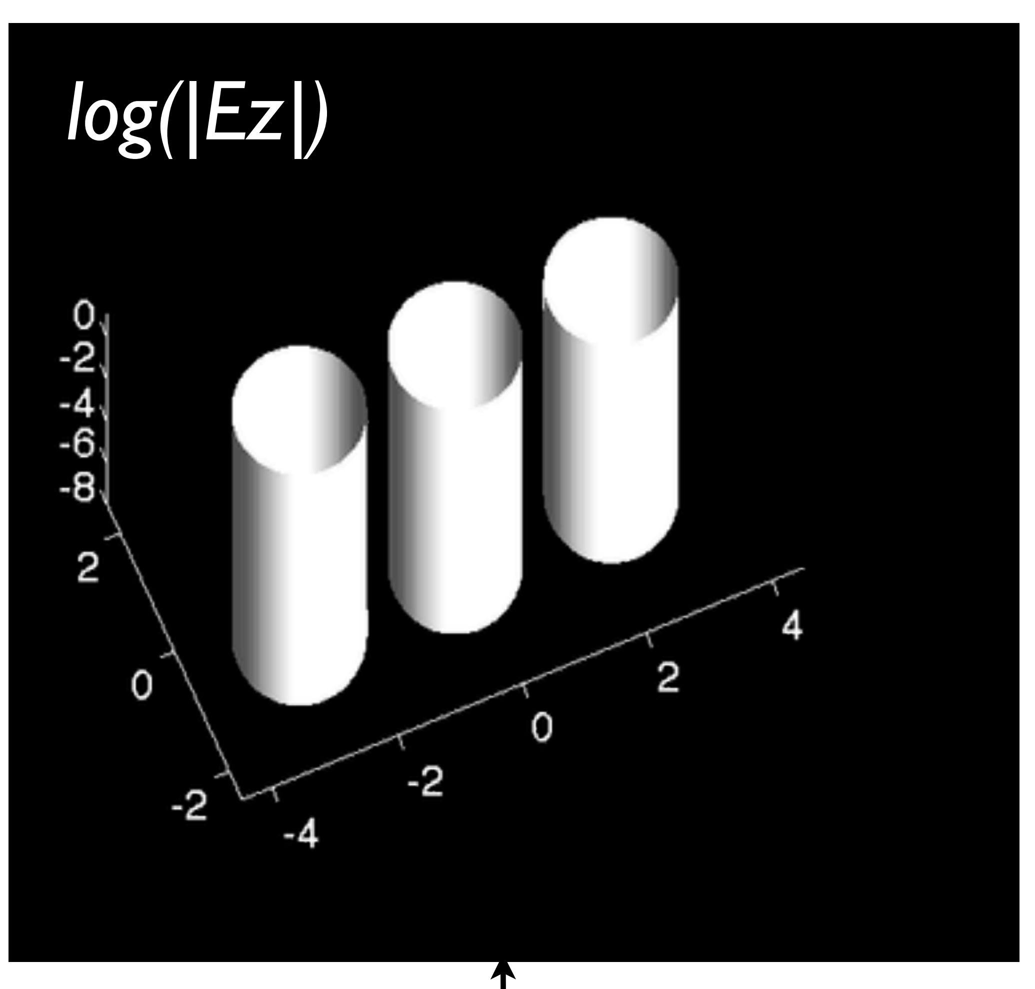
$$
\begin{aligned}
E_z &= \cos(\omega t + \theta)(J_1(\omega r) + aY_1(\omega r)) \ , \\
H_x &= -\frac{1}{2}\sin(\omega t + \theta)\sin(\theta)(J_0(\omega r) - J_2(\omega r) + a(Y_0(\omega r) - Y_2(\omega r))) \\
&\quad -\frac{\cos(\theta)}{\omega r}\cos(\omega t + \theta)(J_1(\omega r) + aY_1(\omega r)) \ , \\
H_y &= \frac{1}{2}\sin(\omega t + \theta)\cos(\theta)(J_0(\omega r) - J_2(\omega r) + a(Y_0(\omega r) - Y_2(\omega r))) \\
&\quad -\frac{\sin(\theta)}{\omega r}\cos(\omega t + \theta)(J_1(\omega r) + aY_1(\omega r)) \ ,
\end{aligned}
$$

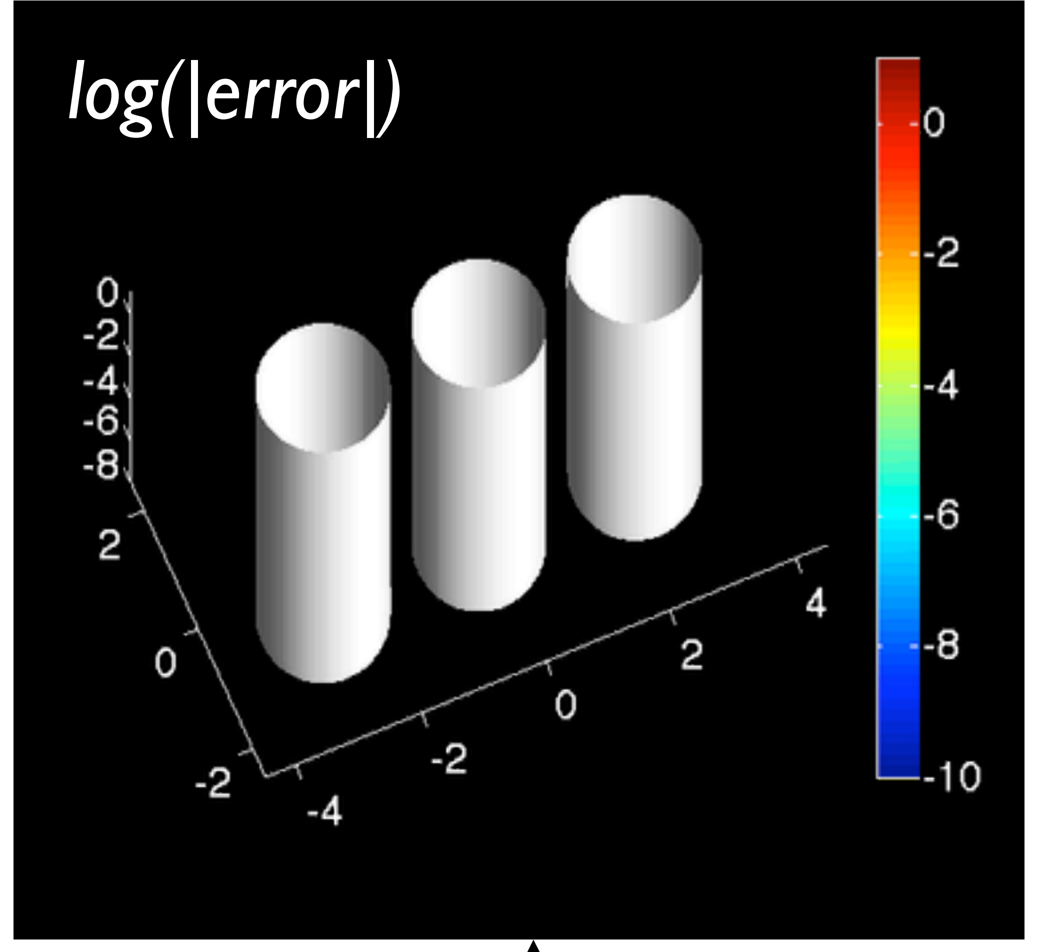
Method	N	h	h/2	h/4	h/8	Est. Order
DGTD	5	2.45E-04	8.06E-06	2.56E-05	5.24E-09	5.61
	6	4.31E-05	1.43E-06	2.52E-08	2.81E-10	6.49
Low storage	5	2.44E-04	8.03E-06	2.55E-05	5.22E-09	5.61
	6	4.29E-05	1.43E-06	2.52E-08	2.79E-10	6.50

Negligible change in accuracy !!!

LSC-DG & CRBC Radiation Boundary Conditions

We integrated the Low Storage Curvilinear DG scheme with the Complete radiation boundary conditions





Low storage curvilinear DGTD

Discrepancy with full integration DGTD

3D Cavity Convergence Test

$$E_z = J_0(k_c\rho)\cos\left(k(z+1)\right)$$

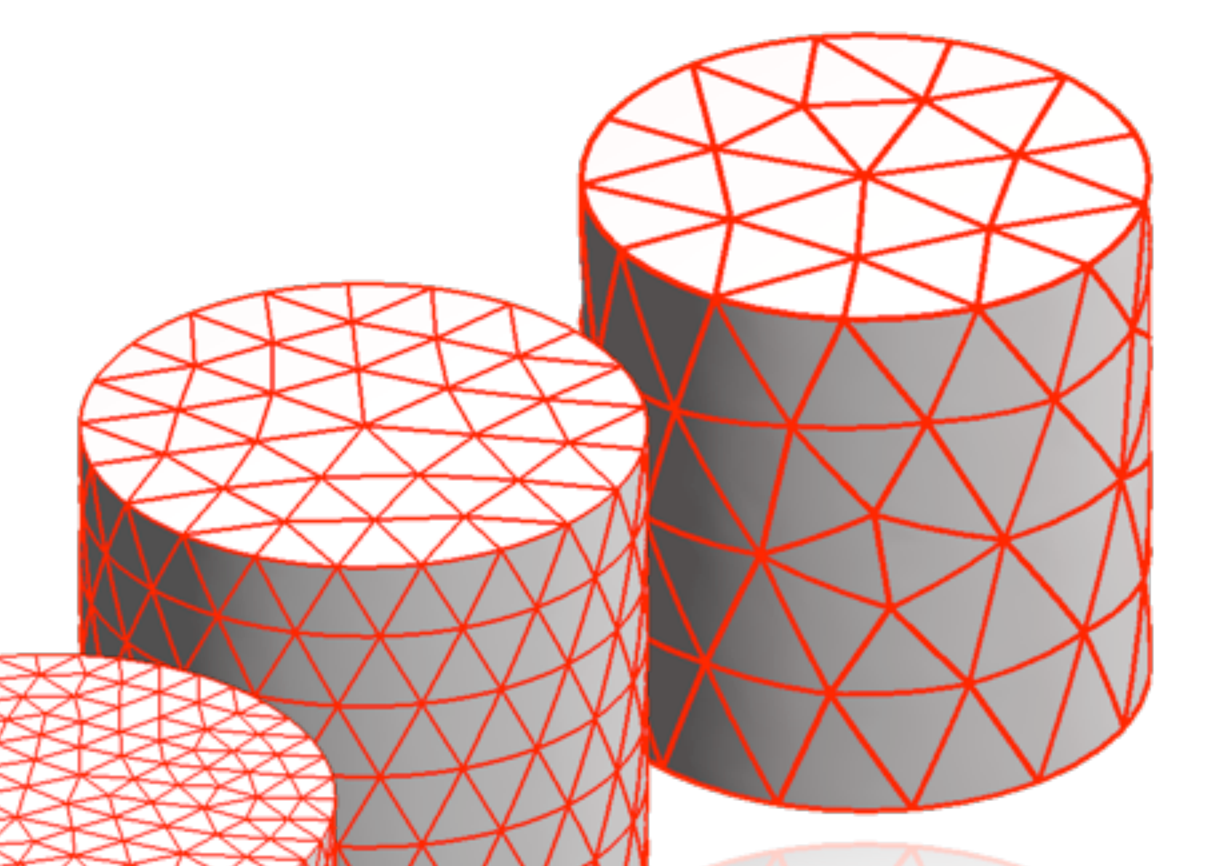
$$E_\rho = -\frac{k}{k_c} J_0'(k_c\rho)\sin\left(k(z+1)\right)$$

$$H_\theta = -\frac{i\omega}{k_c} J_0'(k_c\rho)\cos\left(k(z+1)\right)$$

N	h	h/2	h/4	Est.Order
1	1.87E-01	5.75E-02	1.09E-02	2.03
2	1.59E-02	2.07E-03	1.24E-04	3.43
3	2.53E-03	1.65E-04	6.13E-06	4.01
4	2.03E-04	7.69E-06	2.70E-08	6.89
5	2.29E-05	4.72E-07	7.55E-10	7.85

RICE

3D Cavity Convergence Test



$$E_z = J_0(k_c\rho)\cos(k(z+1))$$

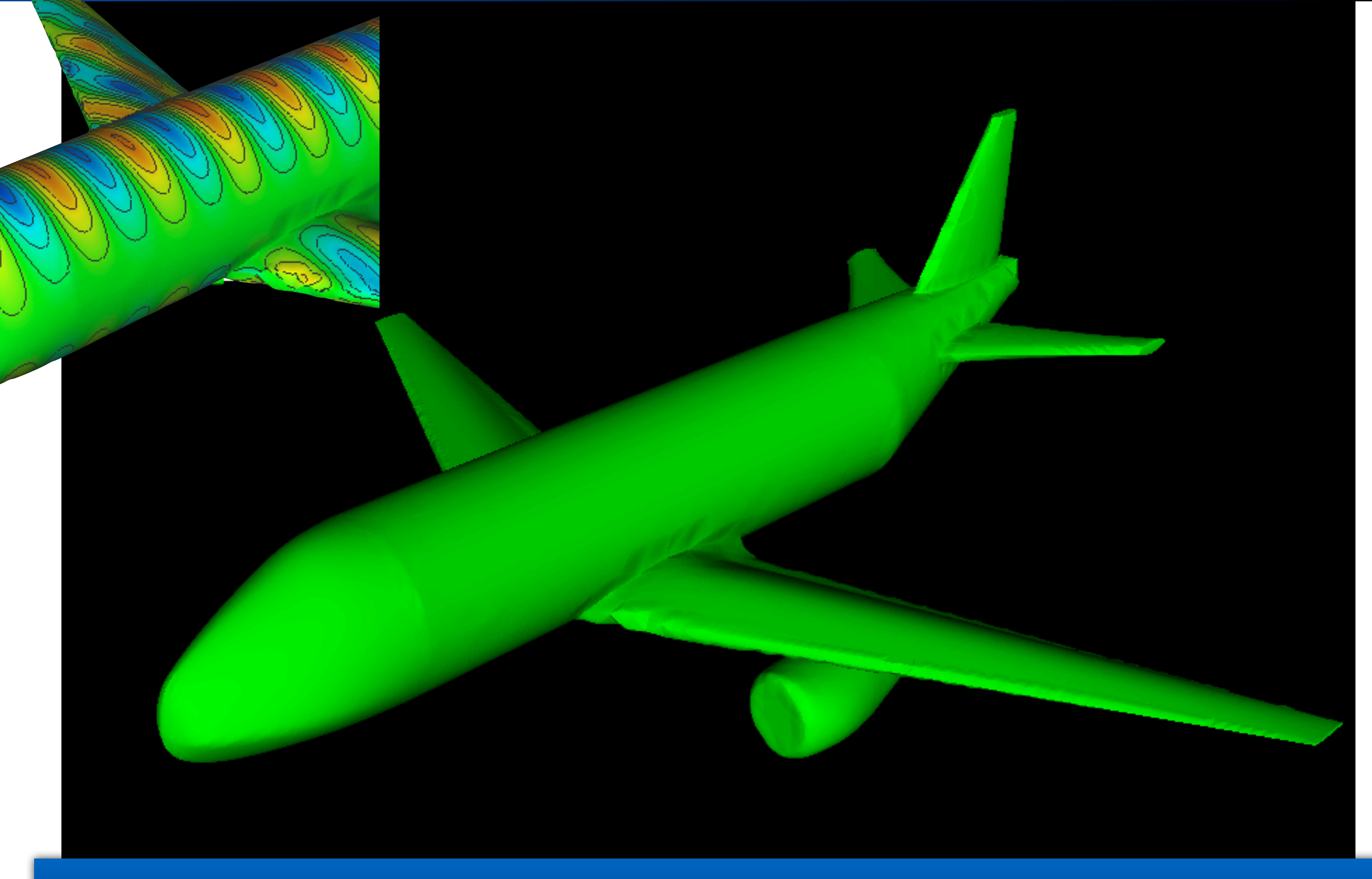
$$E_\rho = -\frac{k}{k_c} J_0'(k_c\rho)\sin(k(z+1))$$

$$H_\theta = -\frac{i\omega}{k_c} J_0'(k_c\rho)\cos(k(z+1))$$

N	h	h/2	h/4	Est.Order
1	1.87E-01	5.75E-02	1.09E-02	2.03
2	1.59E-02	2.07E-03	1.24E-04	3.43
3	2.53E-03	1.65E-04	6.13E-06	4.01
4	2.03E-04	7.69E-06	2.70E-08	<u>6.89</u>
5	2.29E-05	4.72E-07	7.55E-10	<u>7.85</u>

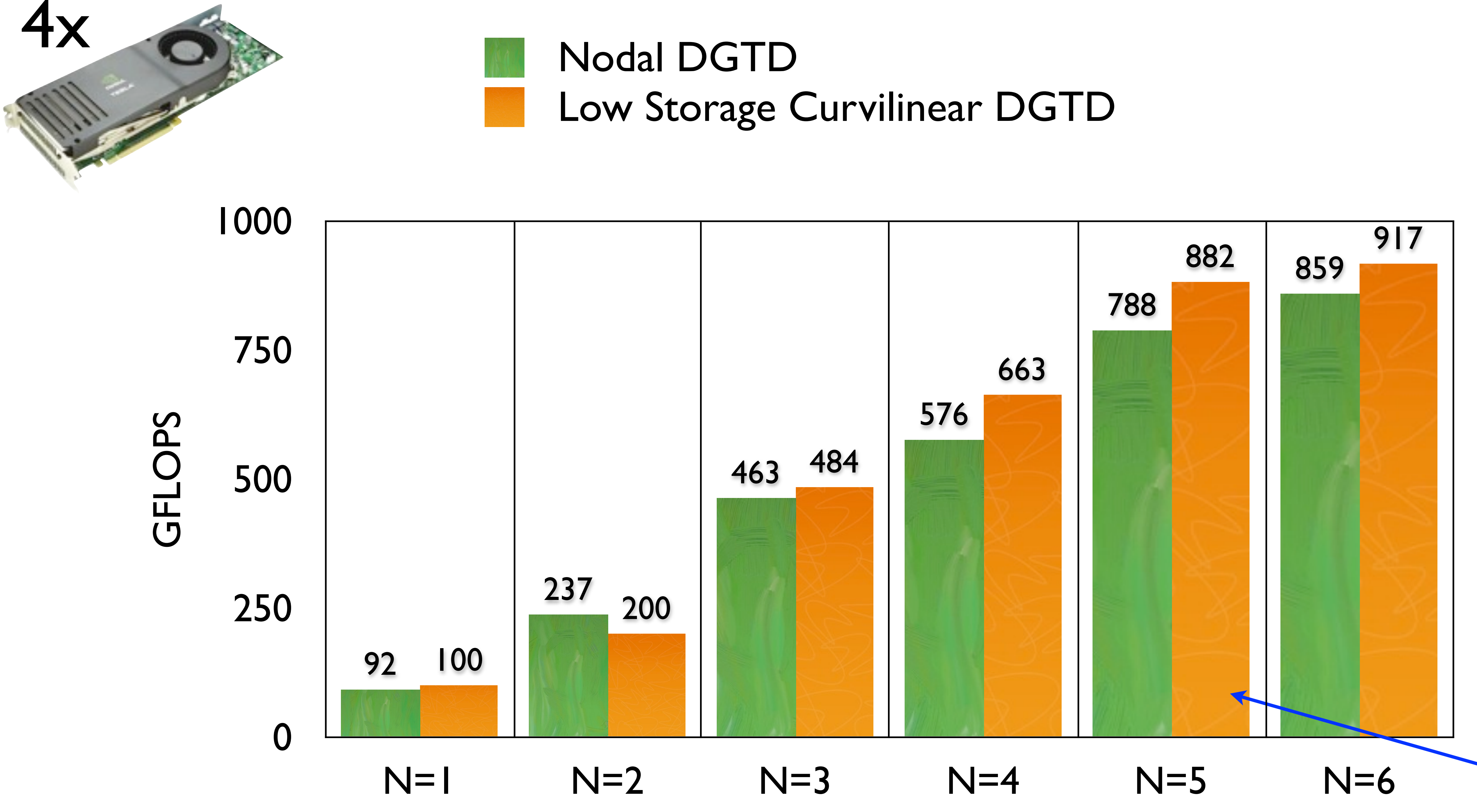
RICE

Low-storage Curvilinear DGTD



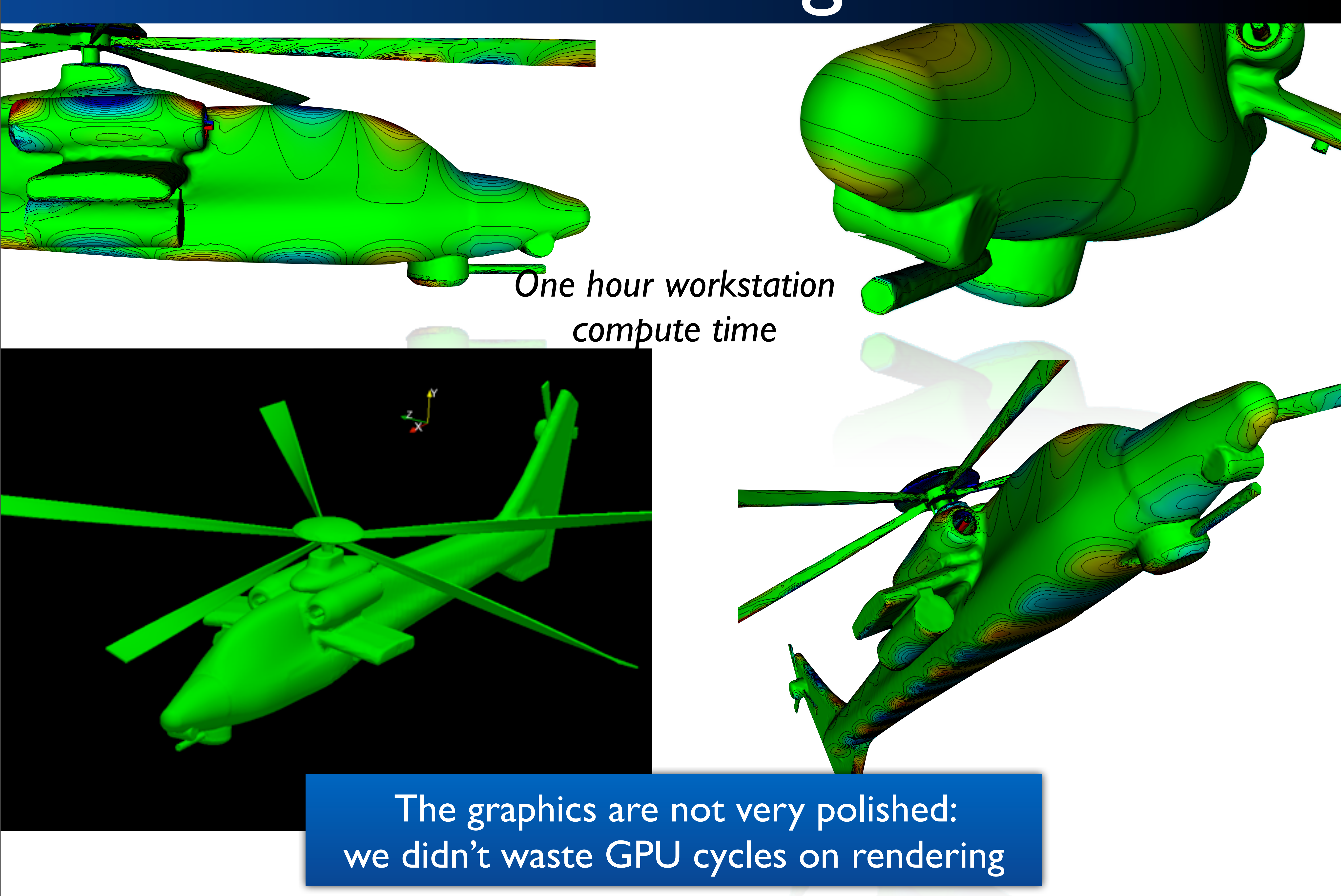
Model airplane curved with cubic Bezier spline surfaces (PN triangles)

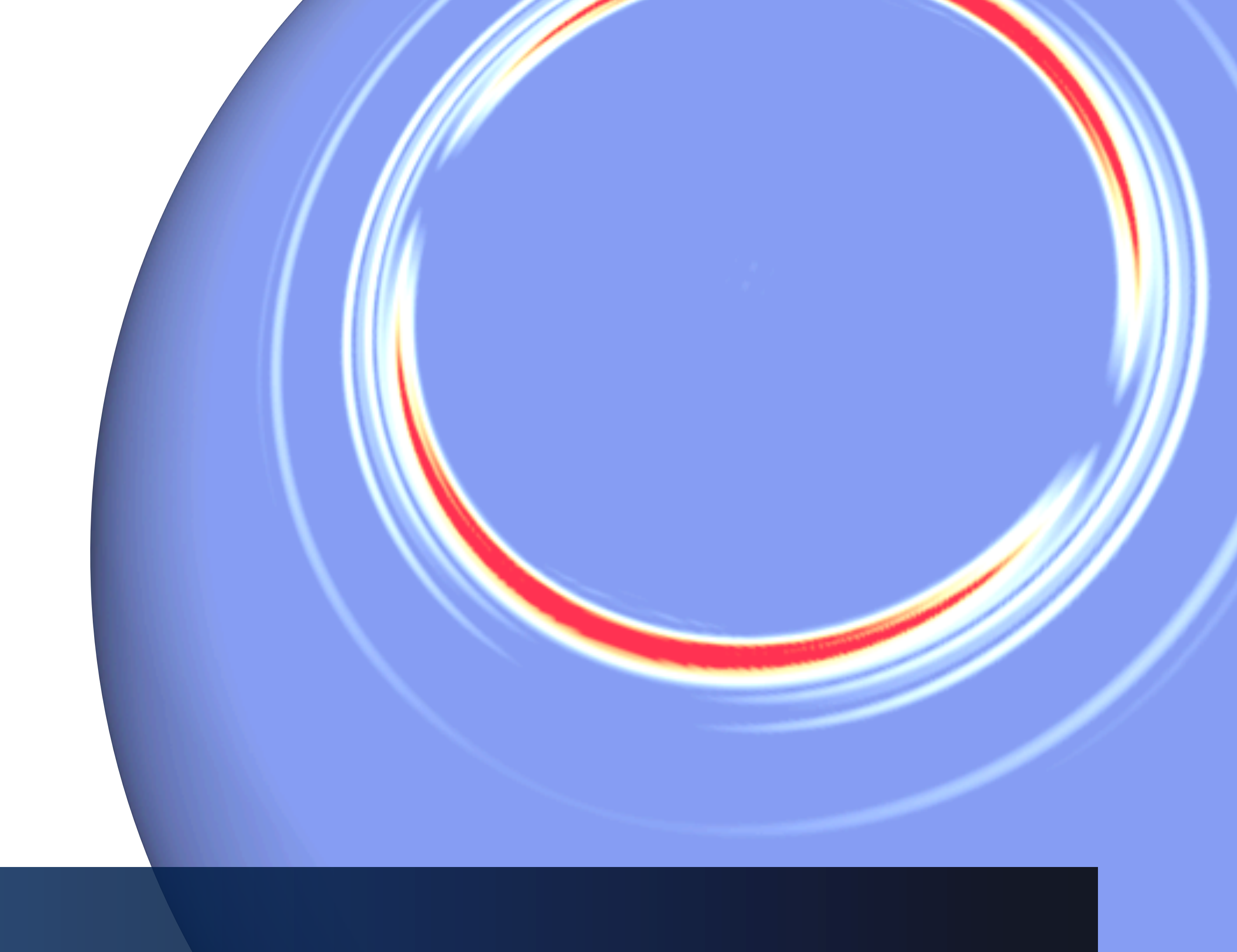
Low Storage Curvi DGTD Performance



The Low Storage Curvilinear DGTD trades some extra algebraic operations for much reduced storage

Colorful Electro-Magnetics





Multi-GPU Scaling Study for a Global Seismic Simulator

[similar to work by Komatitsch et al discussed by Dominik Goeddeke]

Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Andreas Klöckner, Georg Stadler, TW, and Lucas C. Wilcox, *Extreme-Scale AMR.* ACM/IEEE Super Computing Conference Series, 2010. Gordon-Bell finalist paper.

Velocity-Stress Formulation

We start with a first order velocity-stress formulation for elasticity:

$$\frac{\partial S_{ij}}{\partial t} = \lambda\left(\frac{\partial v_k}{\partial x_k}\right)\delta_{ij} + \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)$$

$$\rho\frac{\partial v_i}{\partial t} = \frac{\partial S_{ij}}{\partial x_j} + f_i$$

We express this in conservation law form:

$$\frac{\partial Q_i}{\partial t} = \frac{\partial F_i(Q)}{\partial x_k} + f_i$$

We then express this in weak conservation law form on the local element D^k

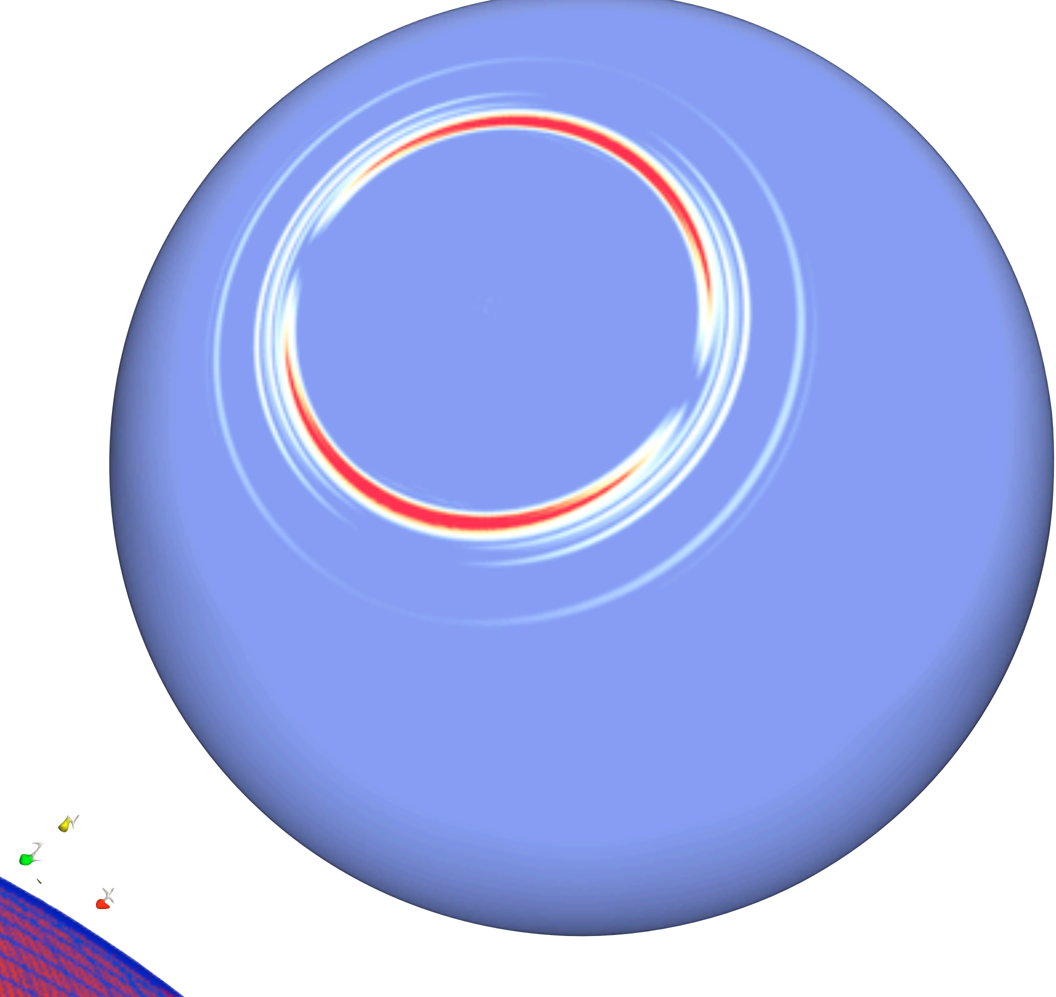
$$\left(\phi, \frac{\partial Q_i}{\partial t}\right)_{D^k} = \left(\phi, \frac{\partial F_{ij}(Q)}{\partial x_j} + f_i\right)_{D^k} + \left(\phi, n_j\left(F_{ij}\left(Q^* - Q\right)\right)_i\right)_{\partial D^k}$$

Hexahedra are typically the element of choice in geophysics.

DGTD Elasticity

Magnitude of velocity after some time.

PREM: Preliminary Reference Earth Model
(Dziewonski & Anderson, 1981)



Example grid and simulation.
Note: non-conforming mesh
enabled trivially with DG.

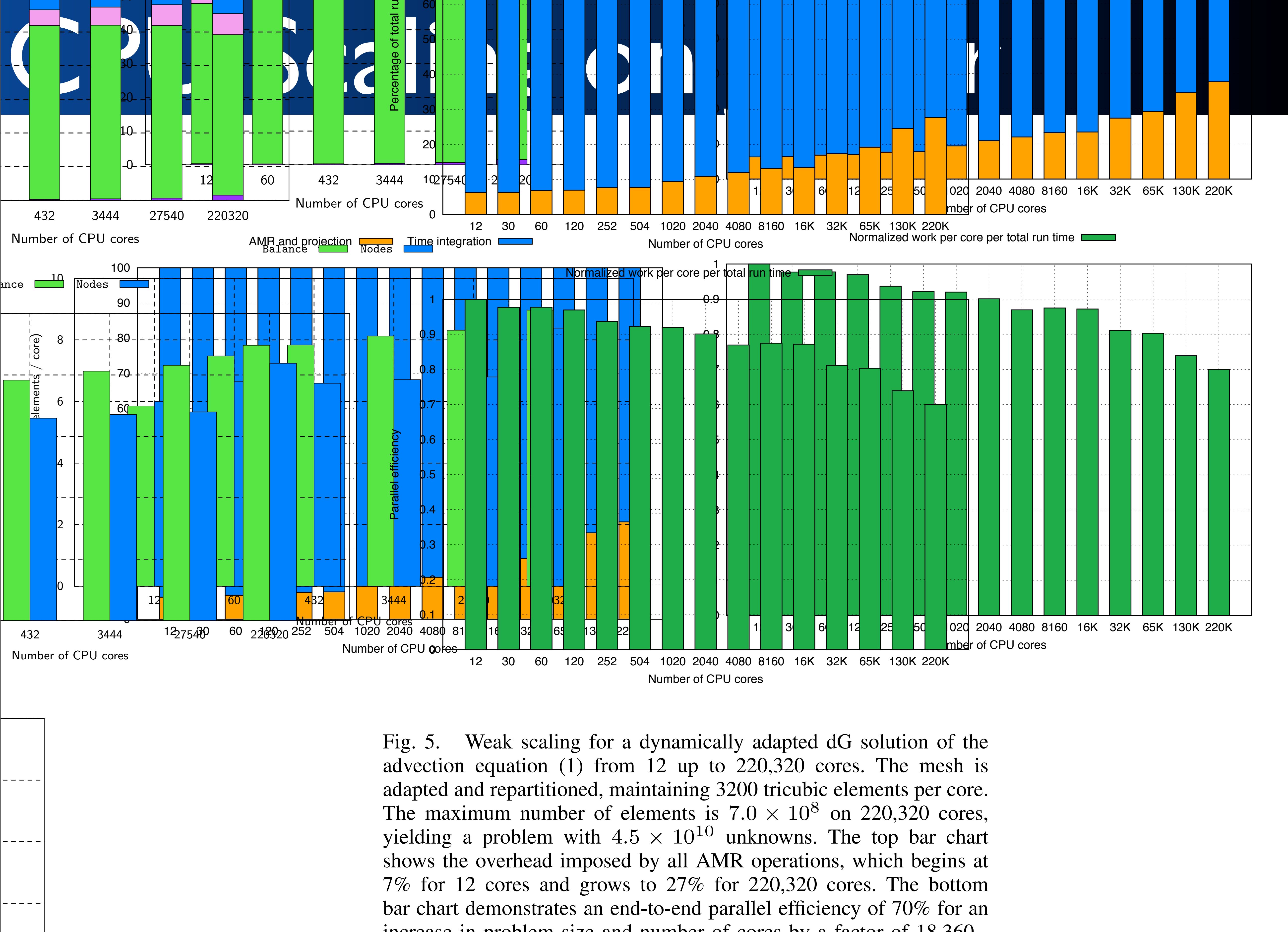
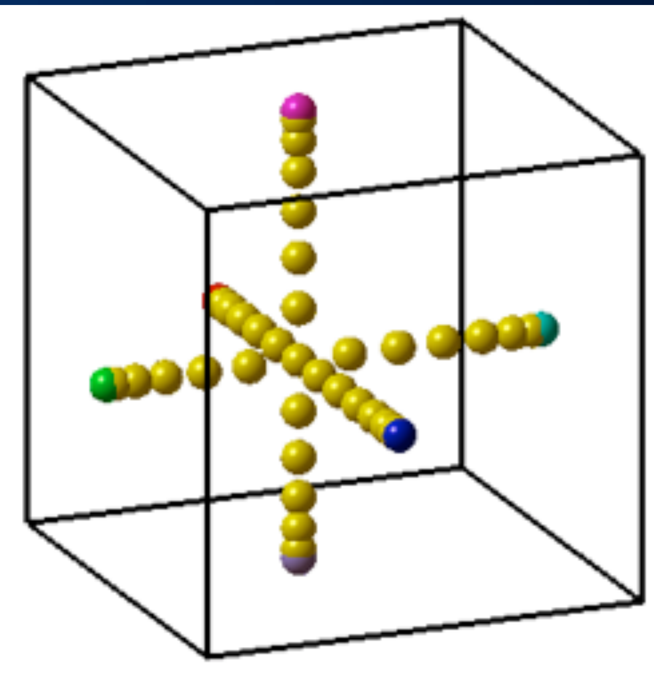
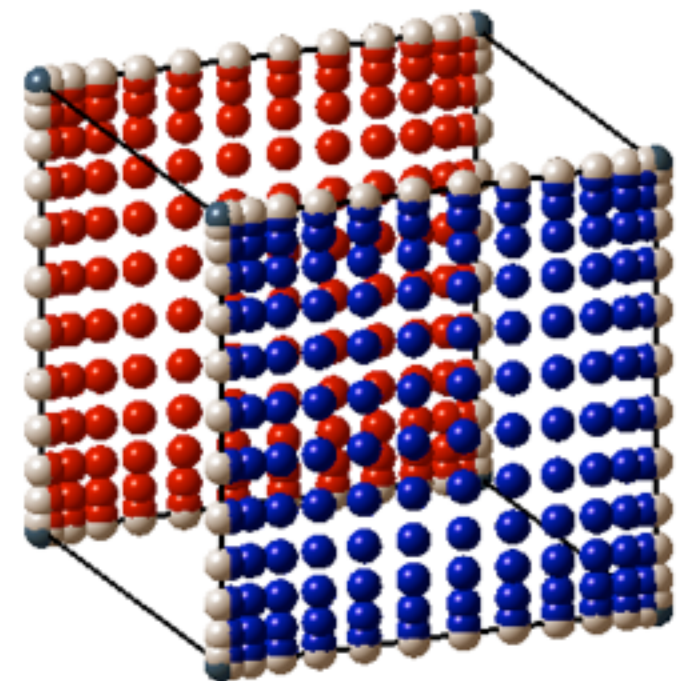


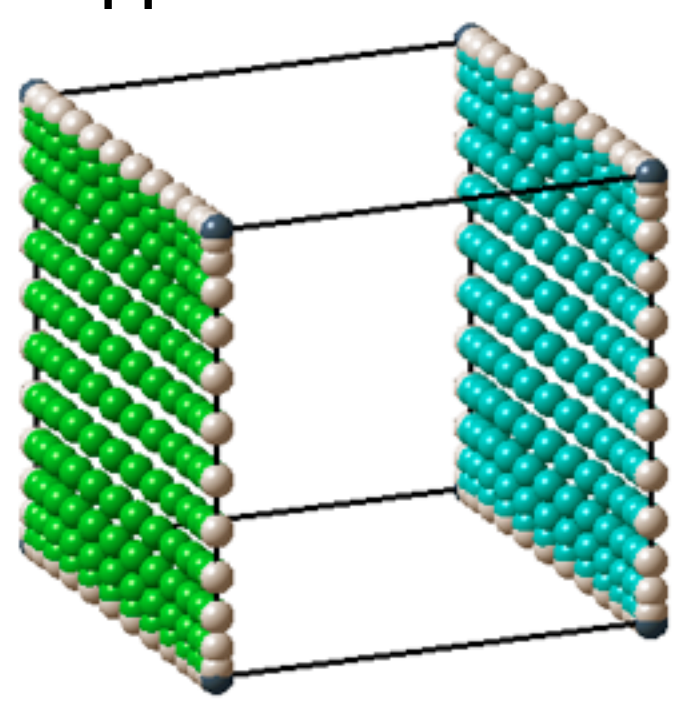
Fig. 5. Weak scaling for a dynamically adapted dG solution of the advection equation (1) from 12 up to 220,320 cores. The mesh is adapted and repartitioned, maintaining 3200 tricubic elements per core. The maximum number of elements is 7.0×10^8 on 220,320 cores, yielding a problem with 4.5×10^{10} unknowns. The top bar chart shows the overhead imposed by all AMR operations, which begins at 7% for 12 cores and grows to 27% for 220,320 cores. The bottom bar chart demonstrates an end-to-end parallel efficiency of 70% for an increase in problem size and number of cores by a factor of 18,360.

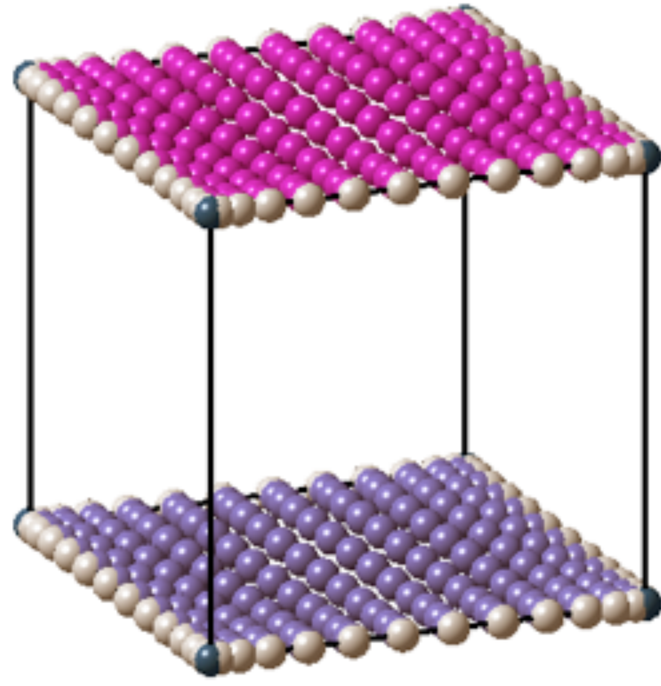
Hex Element DG Kernels



Derivative Kernel uses $(N+1)^3$ threads: each thread computes the 1D spatial derivatives at one node and applies the chain rule for physical derivatives.

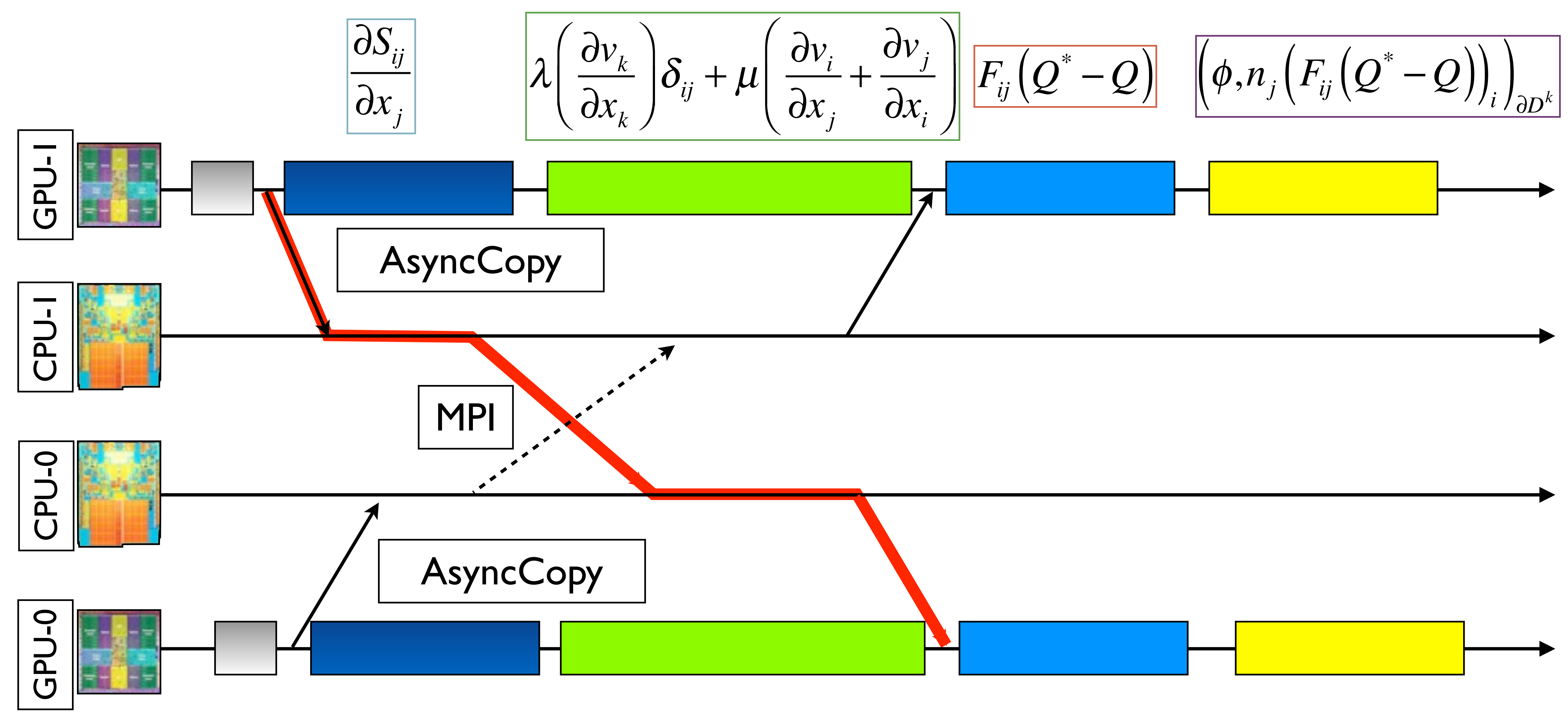






Surface Kernels use $2(N+1)^2$ threads: the thread-block modifies surface nodes on three pairs of faces in turn to avoid write conflicts.

Multi-GPU Flow

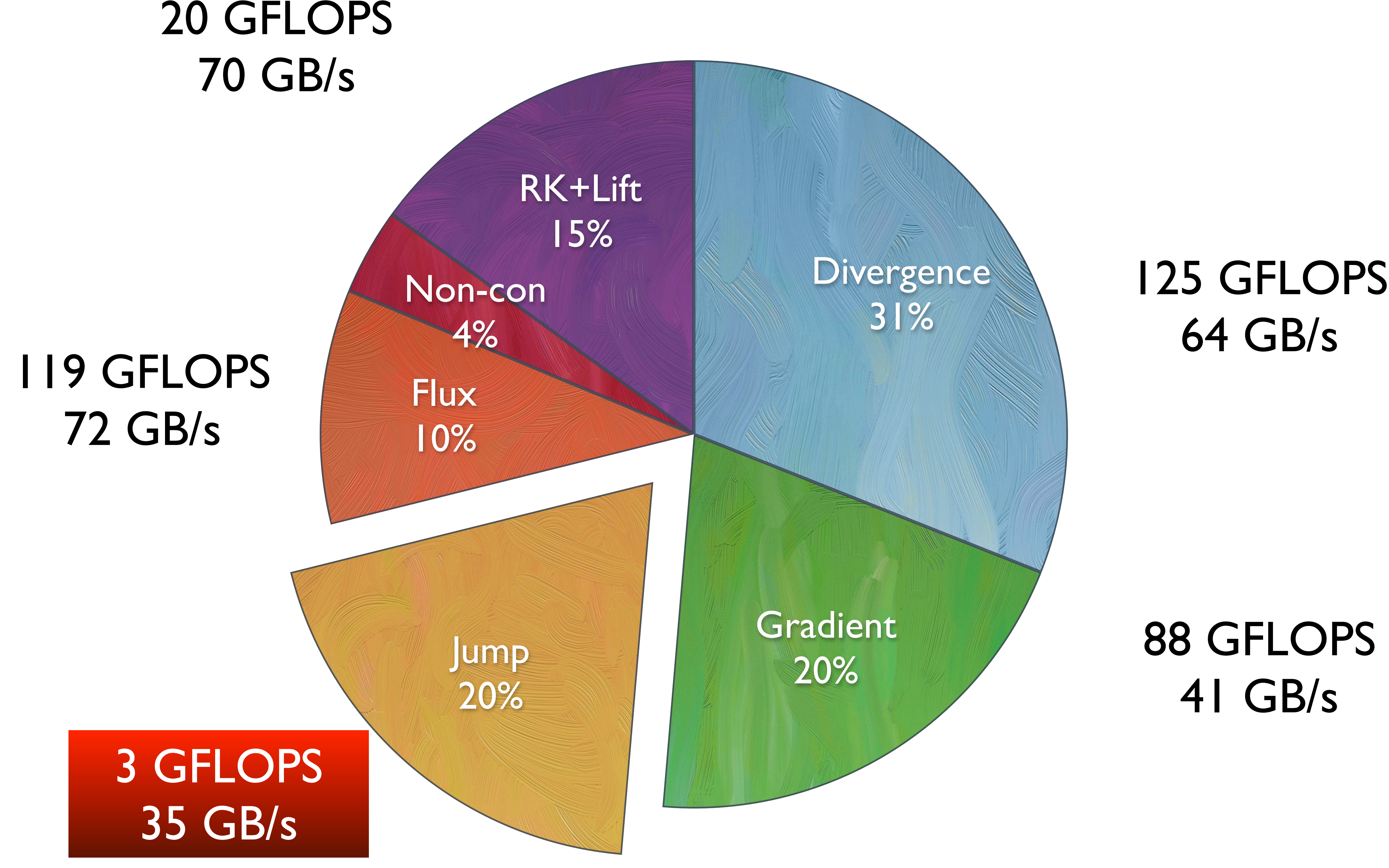


Standard copy and MPI latency hiding through
Async GPU<>CPU copies and non-blocking MPI communications.

We had to pay attention to pinning conflicts between CUDA & MPI.

Hex DGTD: Single C1060

Performance of each kernel (N=7)

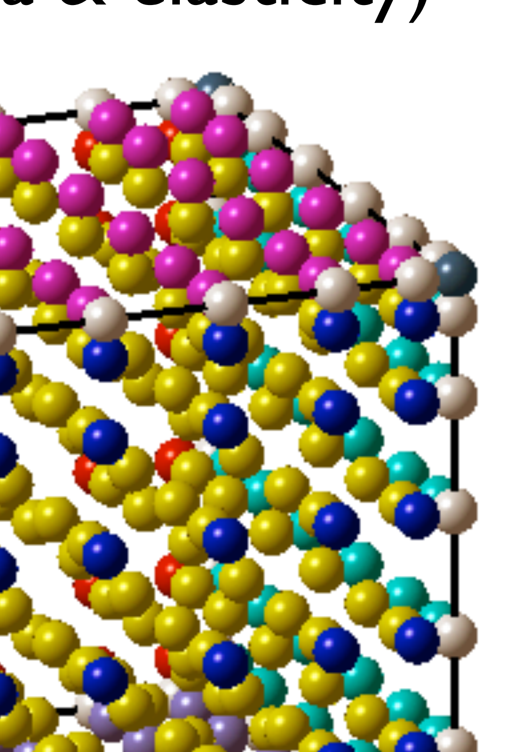


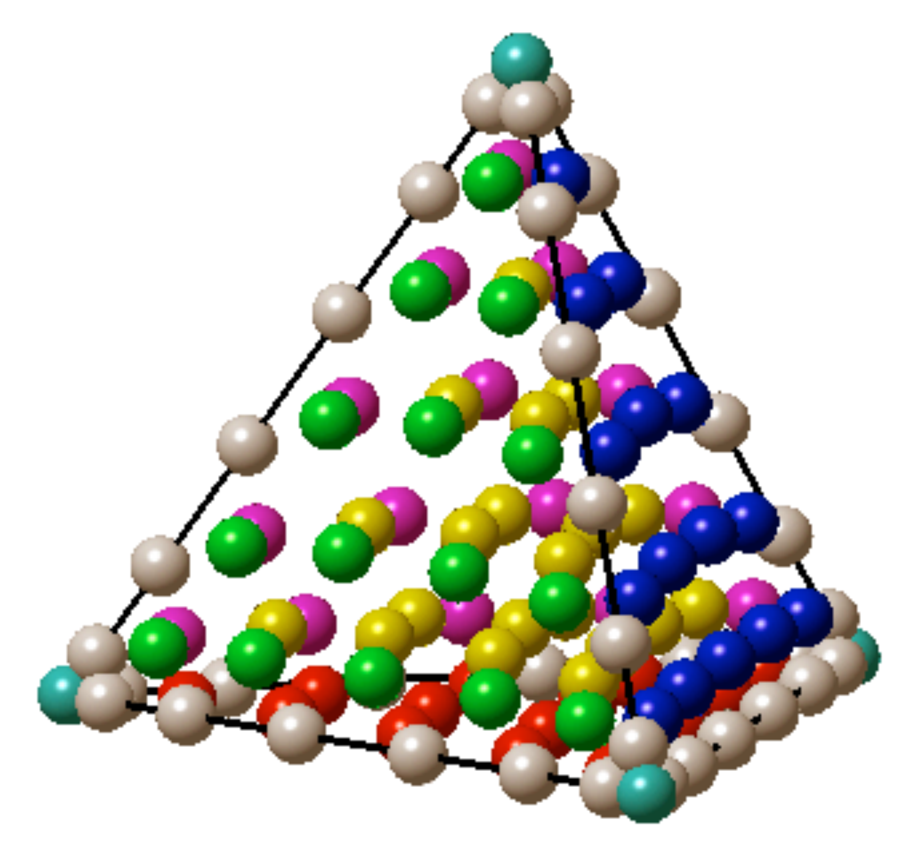
Overall 72 GFLOPS, 53 GB/s.

The tensor product structure of the hex causes some performance issues.

Hex v. Tet on a Fermi GPU

(N=7 hexahedra & elasticity) v. (N=7 tetrahedra & Maxwell's)





Hex Aggregate:
132 GFLOPS

Tet Aggregate:
460 GFLOPS

Tetrahedra consume more FLOPS but at a higher rate than hexahedra.
The elements achieve similar throughput determined by bandwidth not operation count.

The Longhorn GPU Cluster

Longhorn GPU Cluster

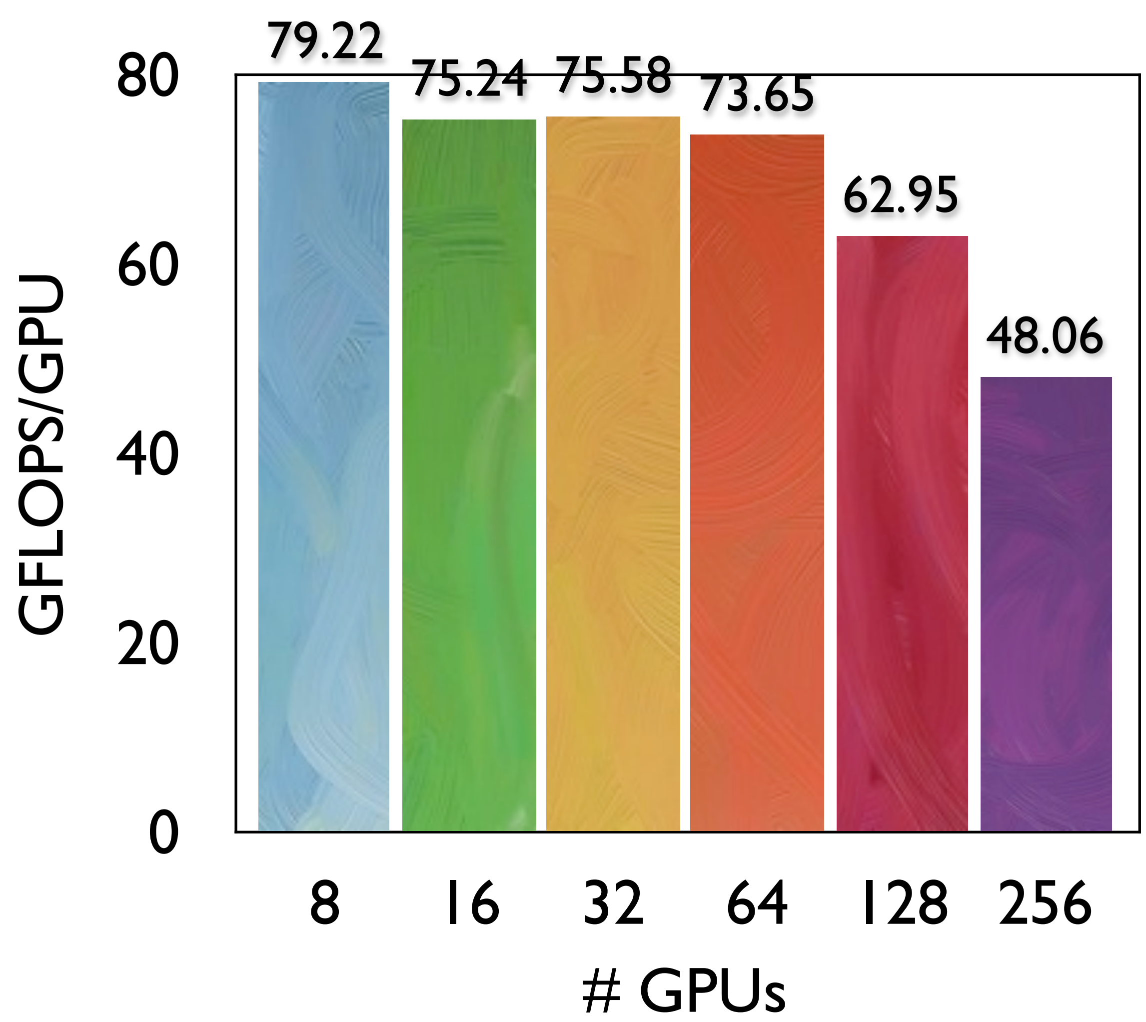
Longhorn is The Texas Advanced Computing Center's
Dell XD Visualization Cluster, consisting of:

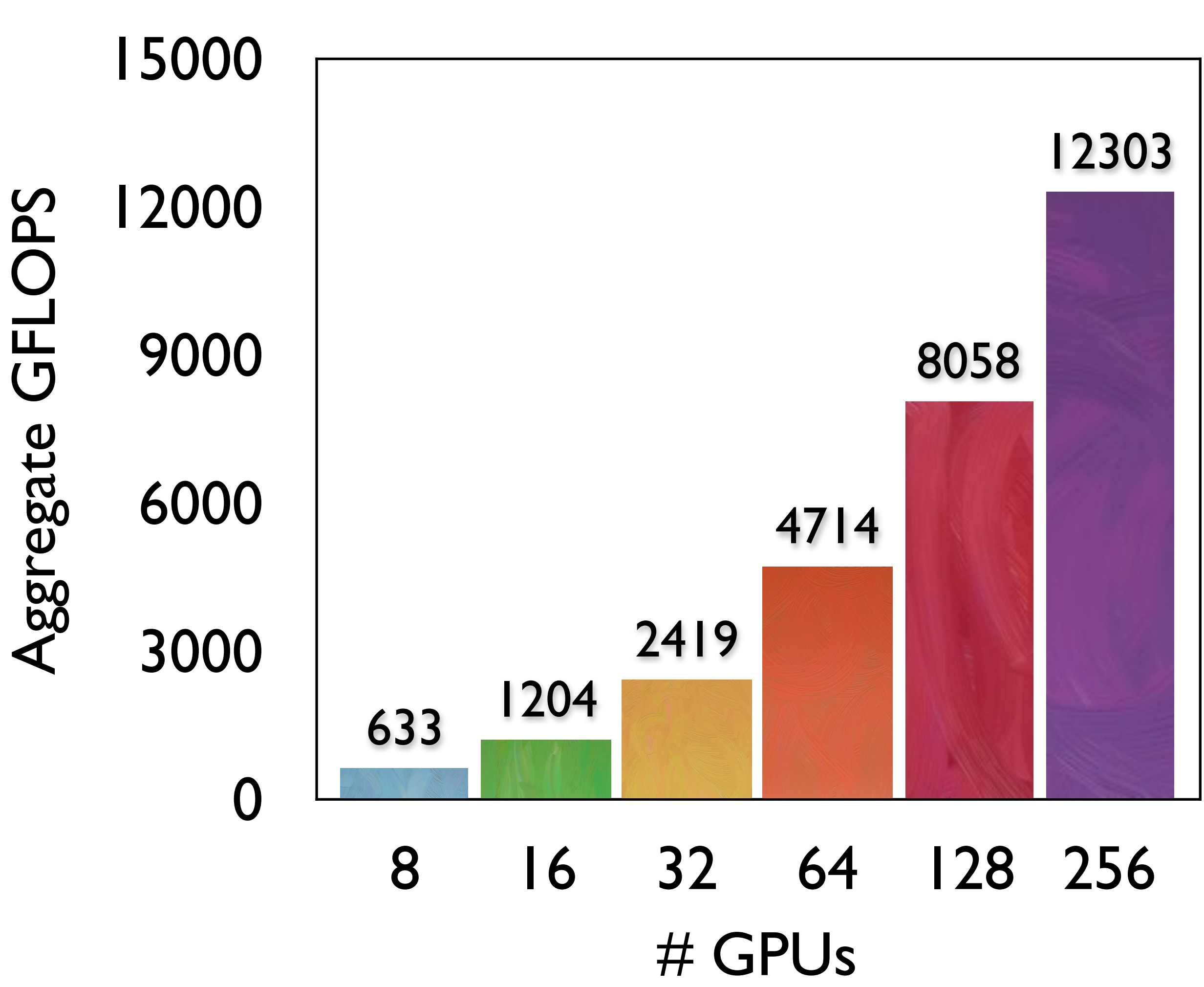
2048 compute cores
14.5 TB aggregate memory
QDR InfiniBand interconnect
Lustre parallel file system
256 nodes + 2 login nodes
2 NVIDIA Quadro FX 5800 (gt200) GPUs per node

Each pair of GPUs shares a PCI Express bus (i.e. half bandwidth)

Strong Scaling: FX5800

Strong scaling: fixed size problem with K=224,048 elements of degree N=7.





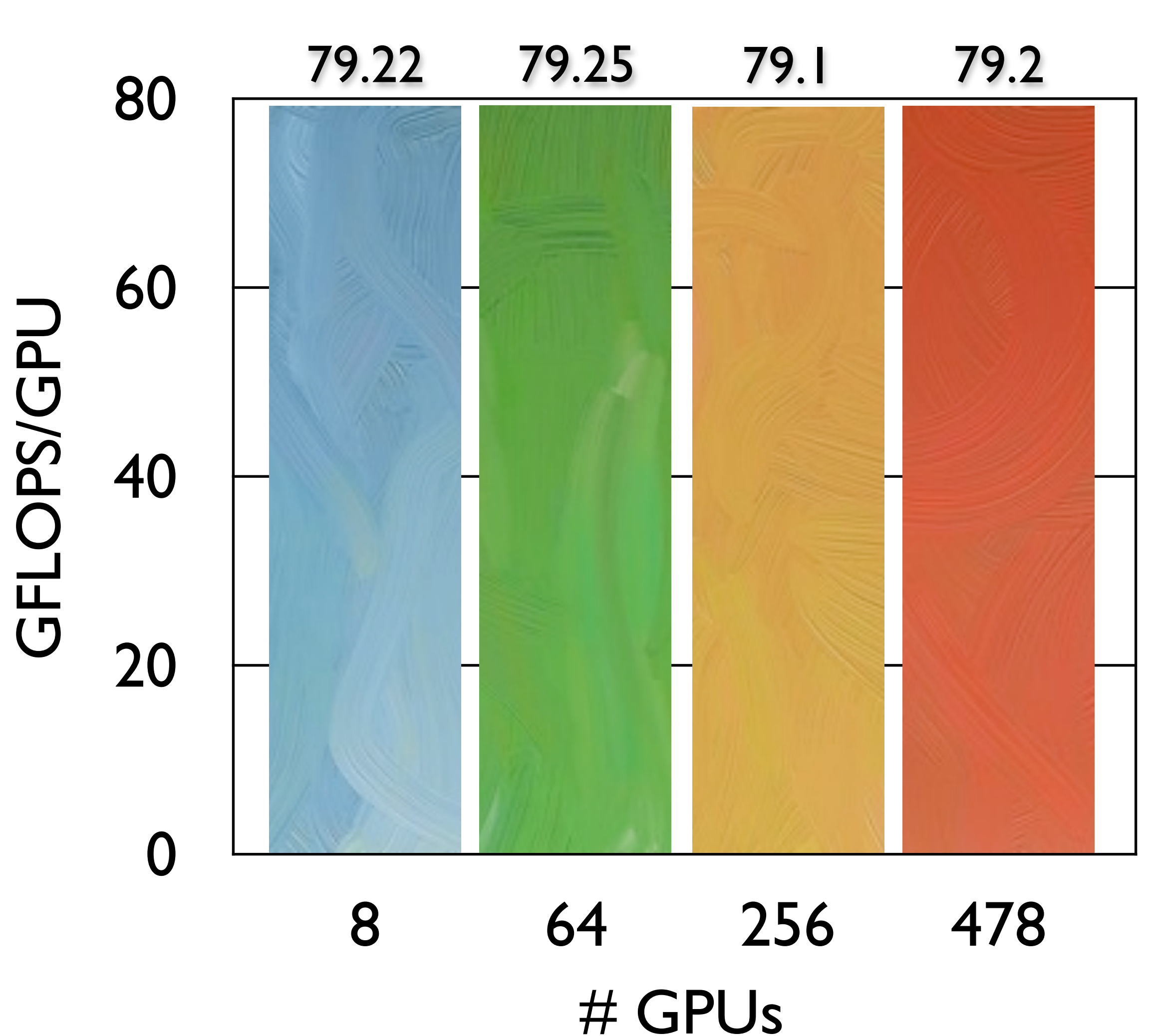
Performance drops with increasing #GPUs due to:

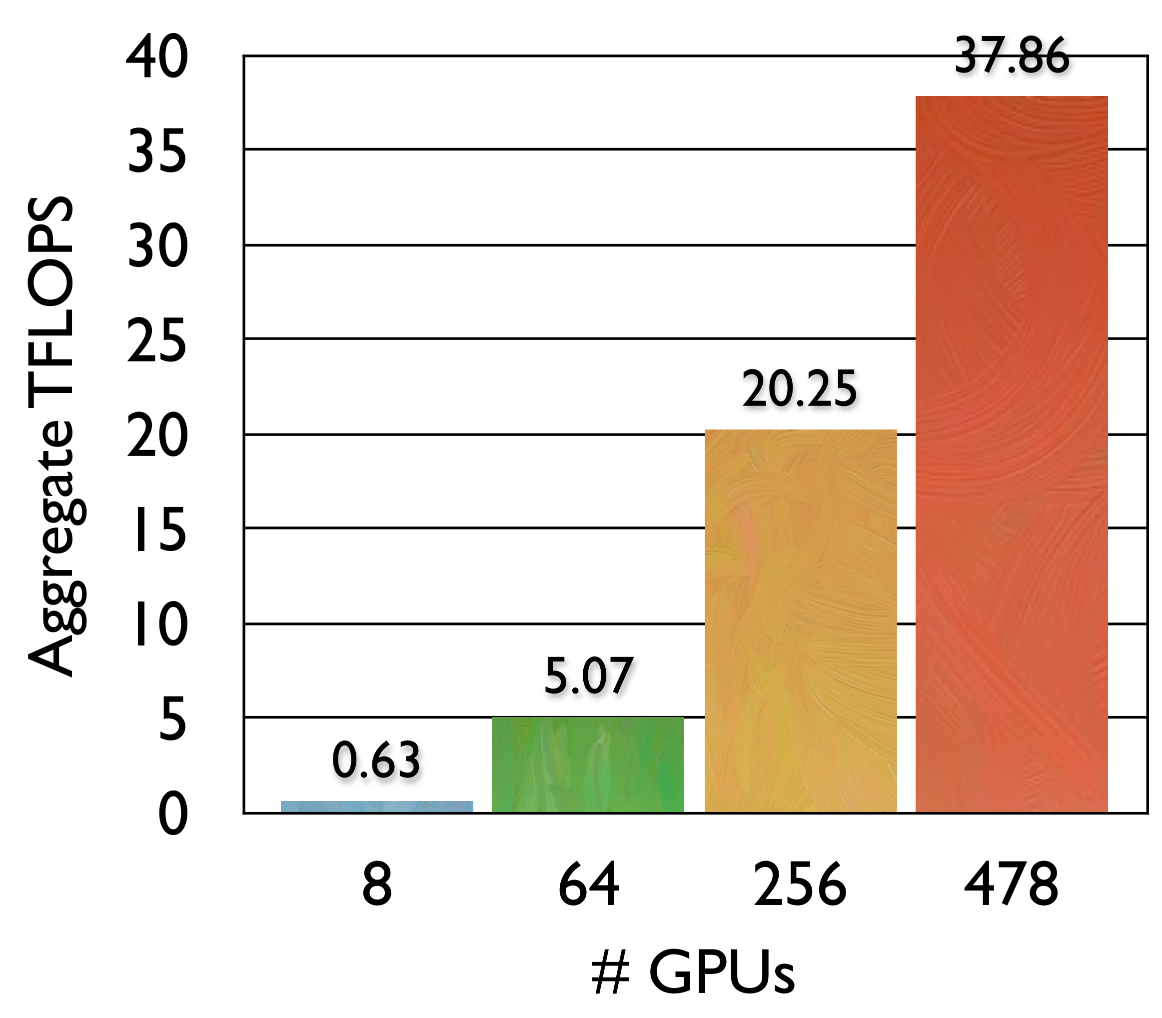
1. Less compute time to hide communication latency.
2. Shared PCI Express bus.
3. Fixed kernel invocation time starts to matter.

Weak Scaling: FX 5800

Weak scaling: #elements in the range [22,000 , 26,000] per GPU with N=7

Hexes:	~12 million
Nodes:	~6 billion
DOFS:	~56 billion





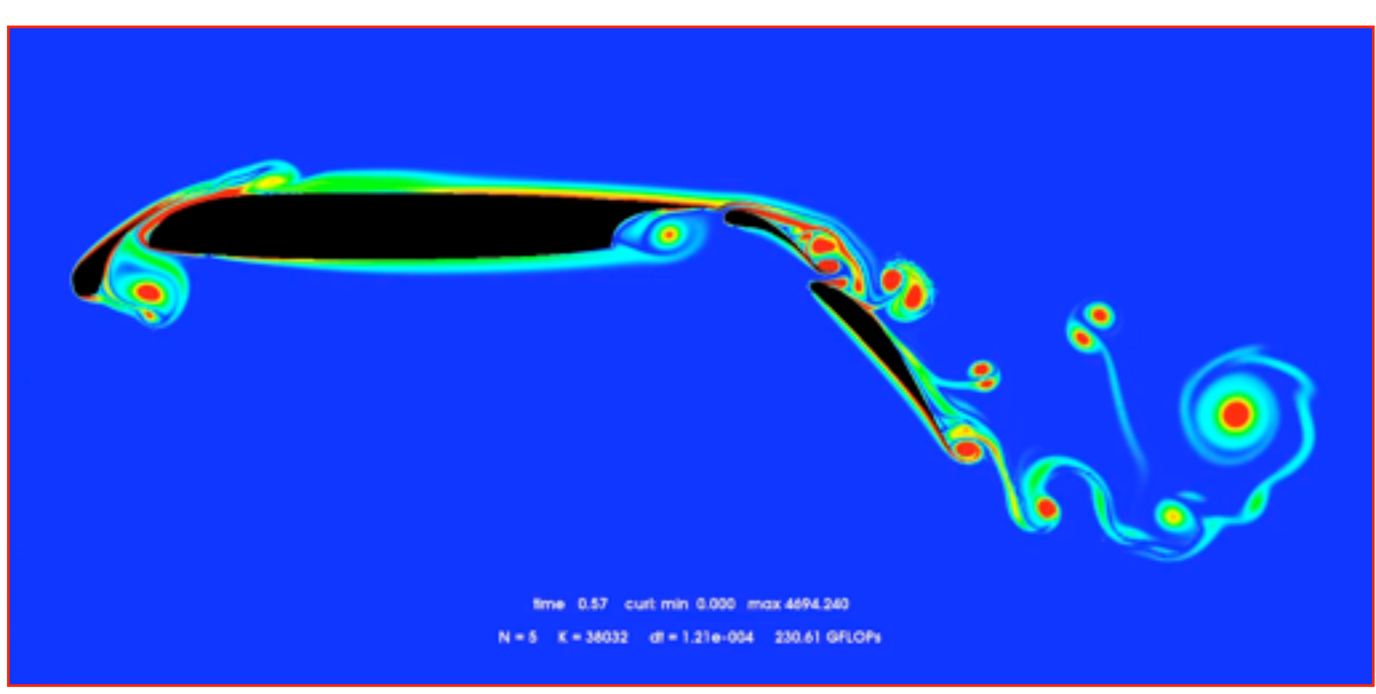
Average bandwidth on each GPU is ~62 GB/s out of ~102GB/s peak.
i.e. we need to restructure the algorithm to substantially improve performance.

GPU Accelerated Discontinuous Galerkin Methods for Compressible Flows

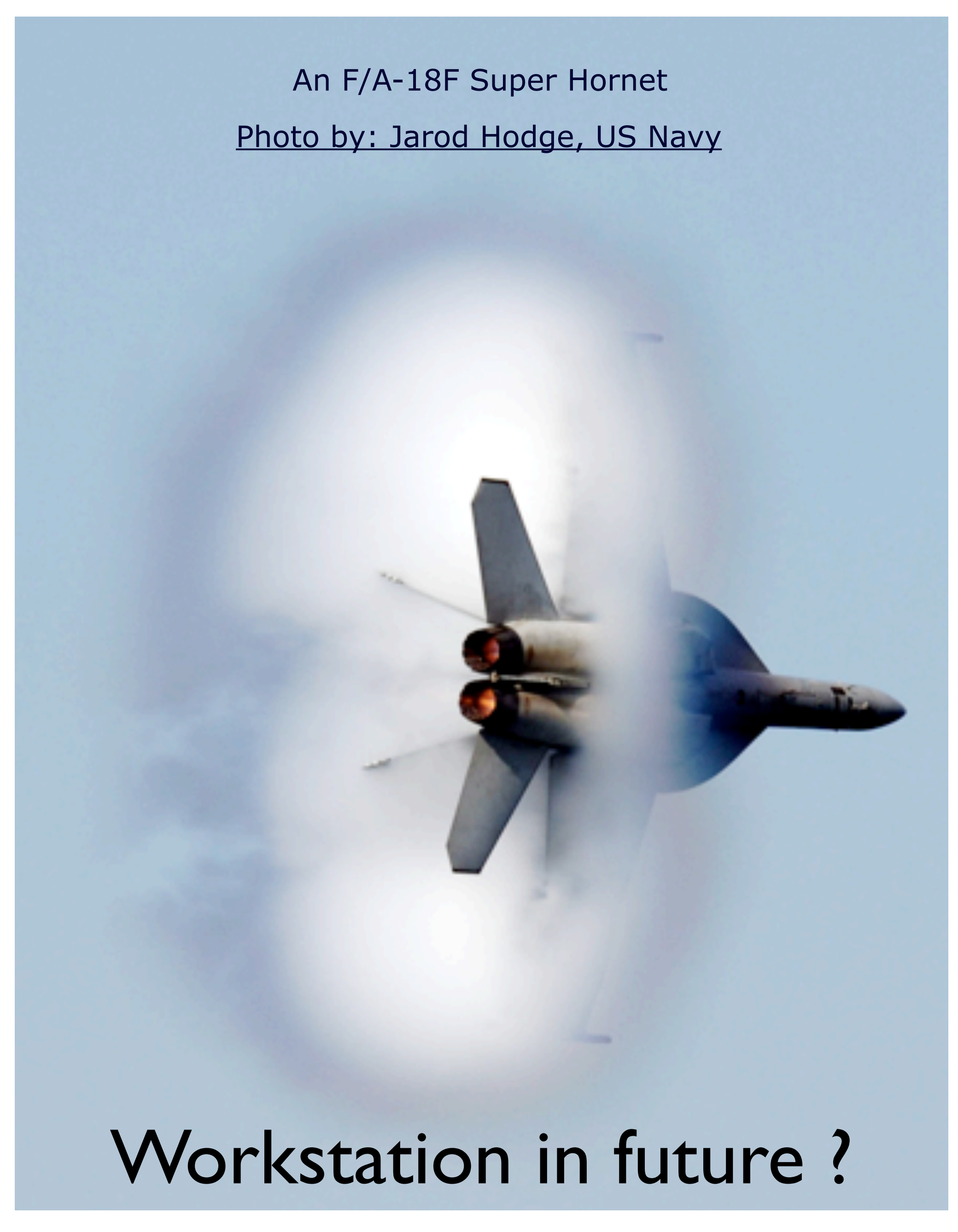
Similar machinery as we have just seen for electromagnetics and elasticity.
Major exceptions: time-stepping for mildly stiff parabolic terms & handling shocks.

Andreas Klöckner, Tim Warburton, Jan S. Hesthaven, *Viscous Shock Capturing with an Explicitly Time-Stepped Discontinuous Galerkin Method*, Mathematical Modelling of Natural Phenomena, 2011.

Compressible Navier-Stokes ...



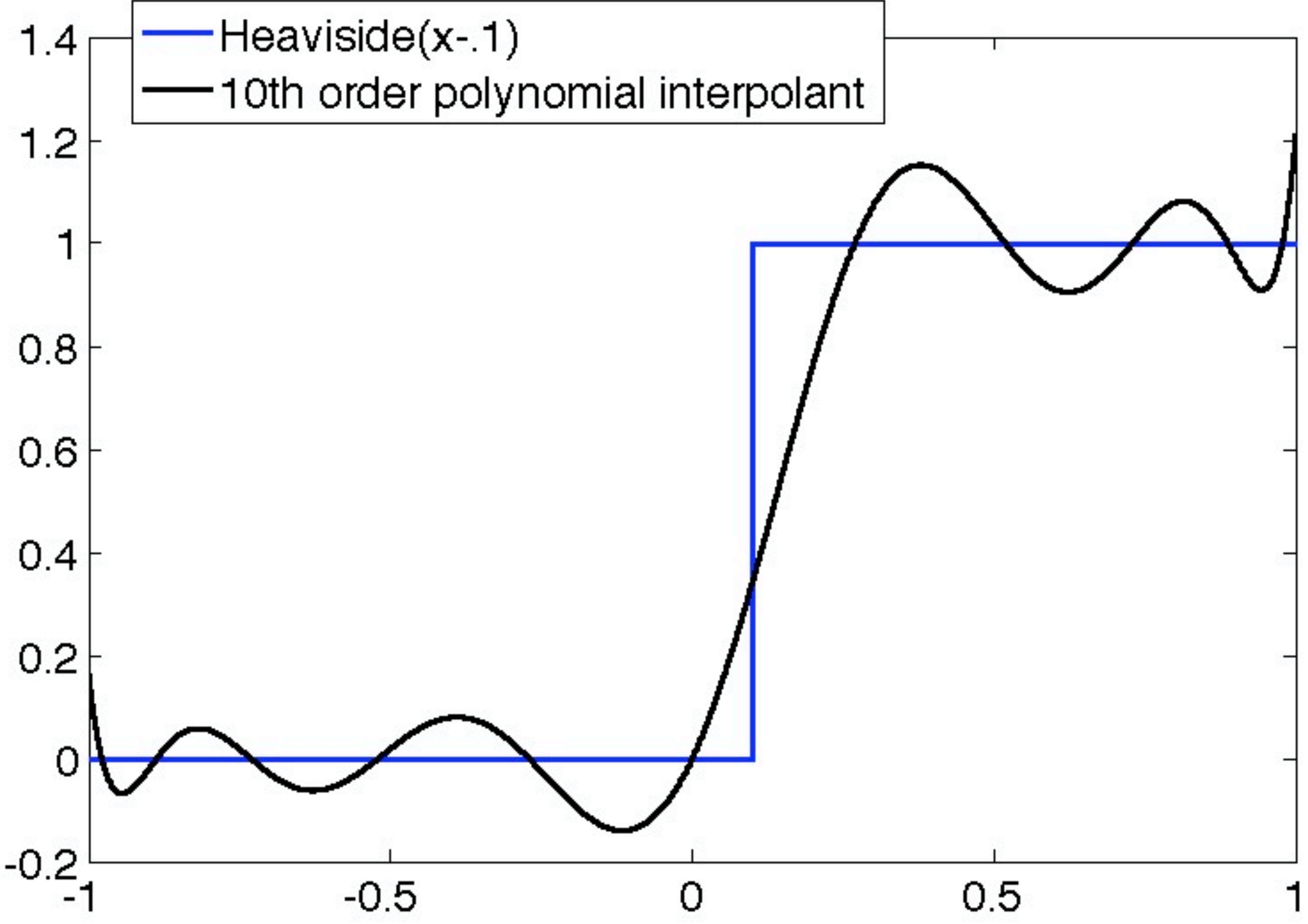
GPU Workstation <1hr



We consider the compressible Navier-Stokes equations in subsonic, transitional and supersonic regimes.

Gibb's Phenomenon

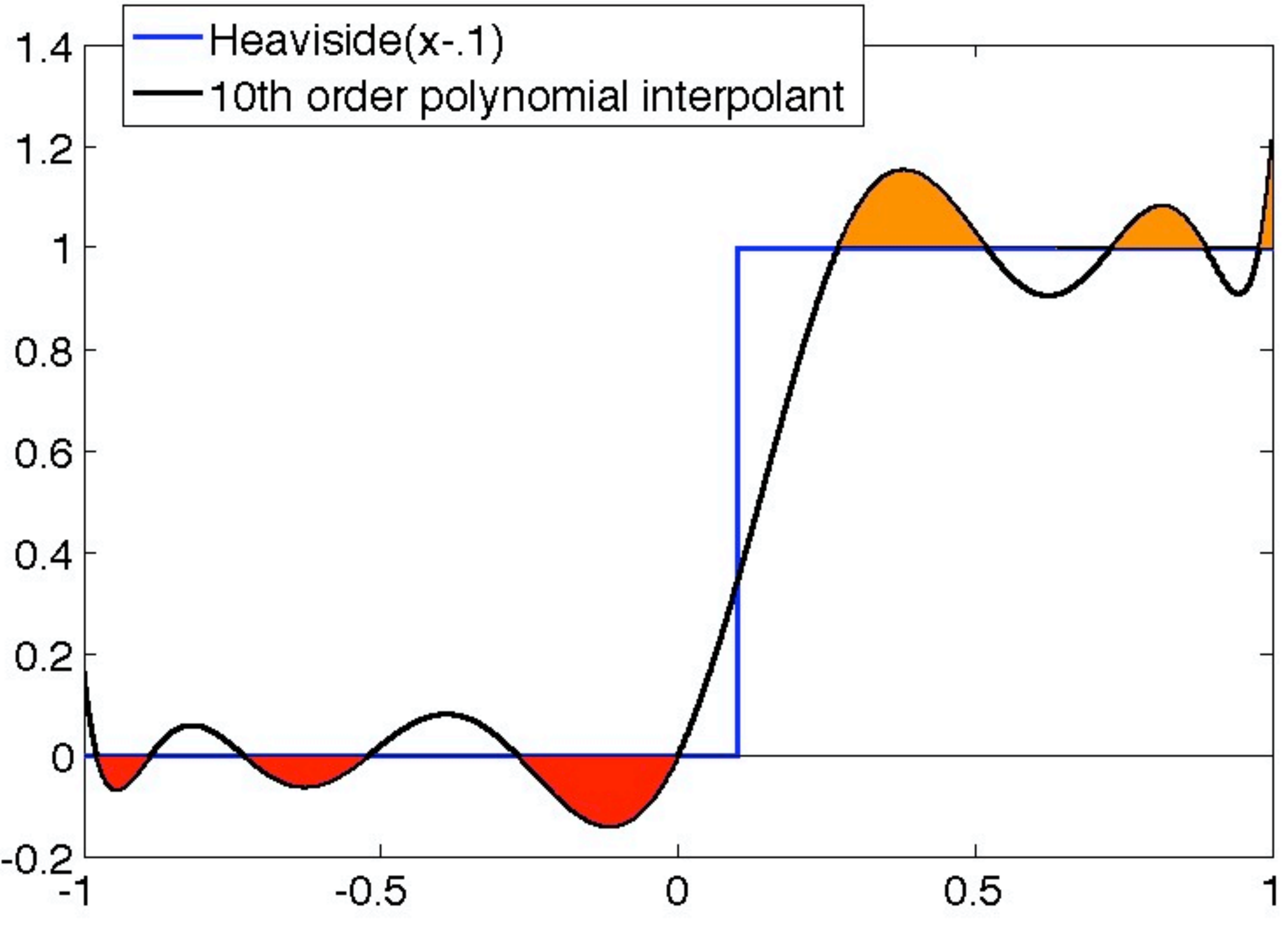
High order polynomial elements yields better utilization, but...



... can suffer critical oscillations when approximating shocks: Gibbs.

Shocking Death for High-Order

The undershoots can cause bad physics...



... the undershoots can stop computations.

Artificial Viscosity

A popular strategy developed by von Neumann and Richtmyer is to smooth the shocks "slightly"

$$\frac{\partial \rho}{\partial t} = -\frac{\partial \rho u_i}{\partial x_i} + \frac{\partial}{\partial x_i}\left(\tilde{\mu}\frac{\partial \rho}{\partial x_i}\right)$$

$$\frac{\partial \rho u_i}{\partial t} = \frac{\partial}{\partial x_j}\left(\tilde{\sigma}_{ij} - \rho u_i u_j - p\delta_{ij}\right)$$

$$\tilde{\sigma}_{ij} = \left(\mu + \tilde{\mu}(\rho)\right)\left\{\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}\right\}$$

$$p = \rho RT$$

We devised a new shock detector and use it to turn on the extra diffusion at shocks and for under resolved features in the flow.

Compressible Navier-Stokes

- First order terms:
  - Roe averaging & Riemann solvers for flux terms.

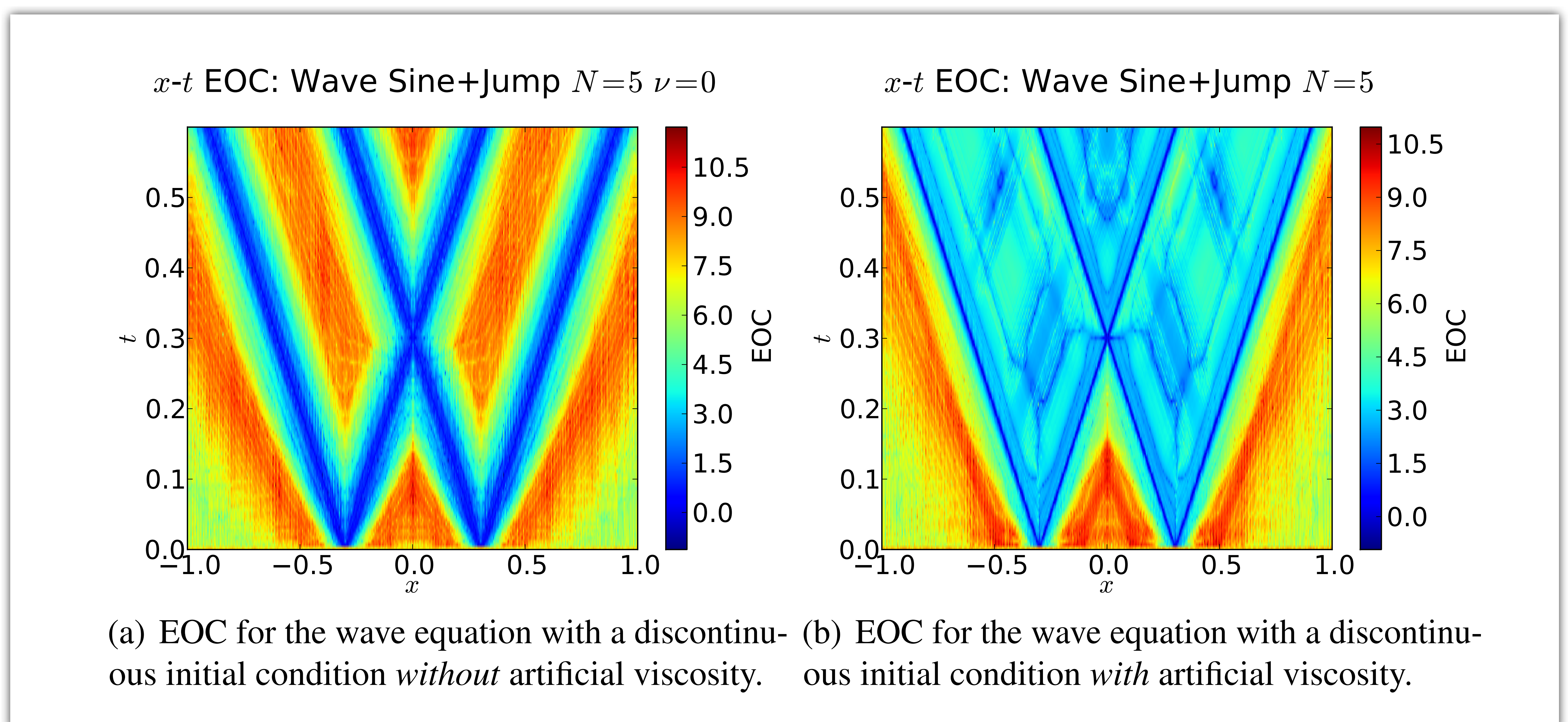
- *Second order terms:*
  - Brezzi et al '97 mixed form DG.

- *Artificial viscosity:*
  - Persson & Peraire, Barter & Darmofal.

- Departures:
  i. Low storage curvilinear DG on unstructured meshes.
  ii. Continuous piecewise linear artificial viscosity reconstruction.
  iii. Shock sensor based on modified regularity estimator of Houston & Süli.

Artificial Viscosity: global pollution

Cockburn & Guzman*: DG applied to linear transport localizes pollution.



(a) EOC for the wave equation with a discontinuous initial condition *without* artificial viscosity.

(b) EOC for the wave equation with a discontinuous initial condition *with* artificial viscosity.

For 1D acoustics a scalar artificial viscosity, based on density alone, generates global pollution by cross coupling characteristics

* B.Cockburn & J.Guzman, *Error estimates for the Runge-Kutta discontinuous Galerkin method for the transport equation with discontinuous initial data*. SINUM 2008.

Artificial Viscosity: global pollution

Order reduction appears in the cone of influence of shocks as expected.

Sod's Problem with $N=5$ and $K=80$

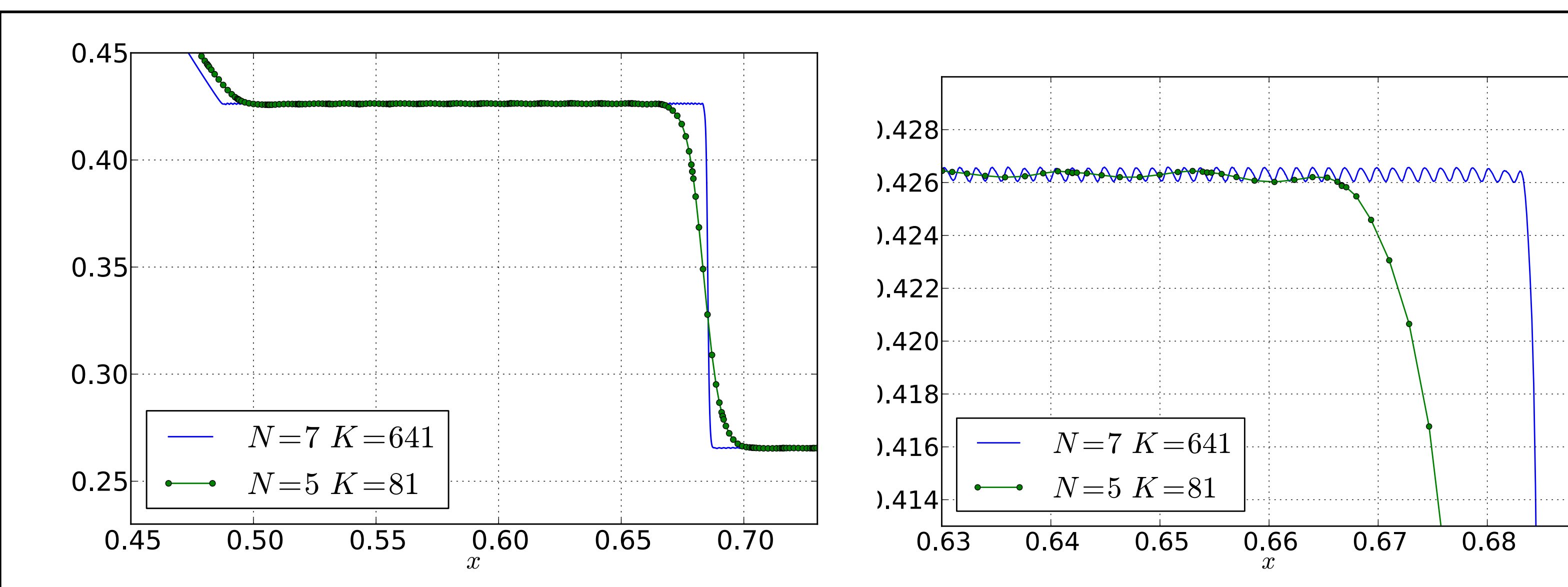
(a) L^2-projected exact and approximate numerical solutions of Sod's problem for polynomial degree $N = 5$ in $K = 80$ elements.

x-t EOC: Euler Sod $N=5$

(b) Space-time diagram of the empirical order of convergence for Sod's problem, computed with artificial viscosity.

Mesh adaptivity and/or post processing is indicated.

Artificial Viscosity: zooming in



(a) Close-up view of the contact discontinuity in Figure 9(a) at low and high numerical resolutions. Interpolation nodes for the low-resolution case are shown as dots.

(b) Extreme close-up view of the tip of the contact discontinuity in Figure 10(a), at low and high numerical resolutions.

Small "coherent" sub element scale oscillations emanate from the shock.

Known problem:
see e.g. Arora & Roe,
Efraimsson & Kreiss

Experimental L_1 errors for the Sod shock problem scale approximately as:

$$\varepsilon(h,N) \sim \frac{h}{N^{2/3}}$$

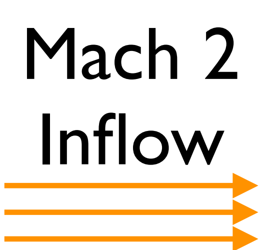
	$N=4$	$N=5$	$N=7$	$N=9$	EOC
$h/1$	$9.982 \cdot 10^{-3}$	$7.934 \cdot 10^{-3}$	$6.522 \cdot 10^{-3}$	$5.567 \cdot 10^{-3}$	0.70
$h/2$	$5.442 \cdot 10^{-3}$	$4.231 \cdot 10^{-3}$	$3.395 \cdot 10^{-3}$	$2.921 \cdot 10^{-3}$	0.75
$h/4$	$2.945 \cdot 10^{-3}$	$2.219 \cdot 10^{-3}$	$1.778 \cdot 10^{-3}$	$1.568 \cdot 10^{-3}$	0.76
$h/8$	$1.548 \cdot 10^{-3}$	$1.166 \cdot 10^{-3}$	$9.488 \cdot 10^{-4}$	$8.329 \cdot 10^{-4}$	0.74
$h/16$	$8.087 \cdot 10^{-4}$	$6.006 \cdot 10^{-4}$	$5.121 \cdot 10^{-4}$	$4.598 \cdot 10^{-4}$	0.66
$h/32$	$4.207 \cdot 10^{-4}$	$3.111 \cdot 10^{-4}$	$2.806 \cdot 10^{-4}$	—	0.69
EOC	0.93	0.95	0.92	0.92	

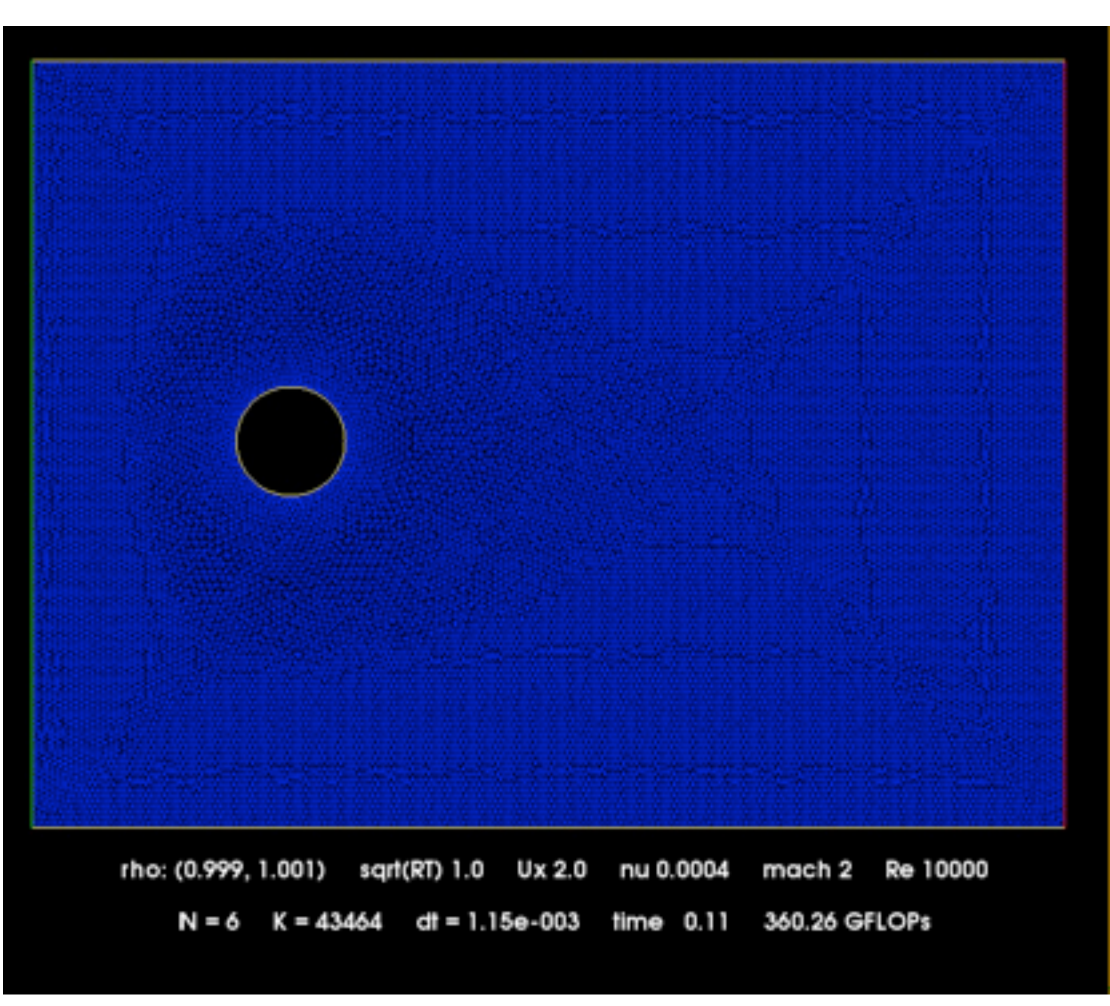
Artificial Viscosity

Good Match for GPGPU:

i. Shock detector is element local => natural thread blocking

ii. Extra damping terms only require small code modifications.





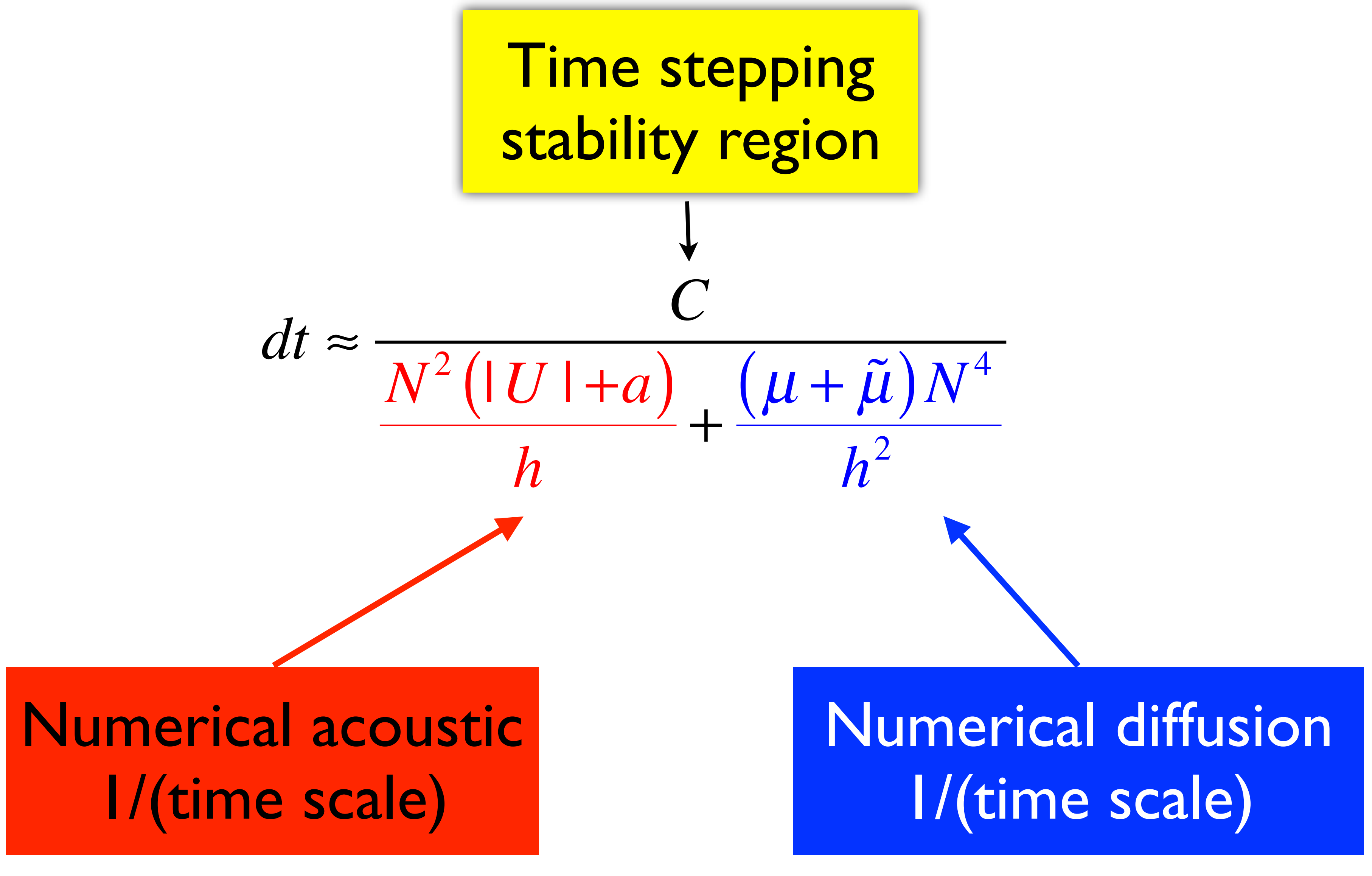
GPGPU Bottlenecks:

i. The shock sensor involves a local reduction in the thread block.

ii. Viscosity reconstruction step uses global gather and scatter at element vertices.

Explicit Time Stepping Restriction

The time step restriction is driven by three factors



Initially we have used an adaptive Runge-Kutta-Chebyshev time stepping method designed by Medovikov.

** A. Medovikov. High order explicit methods for parabolic equations. BIT Numerical Mathematics, 38:372-390, 1998*

Time Stepping Considerations on the GPU

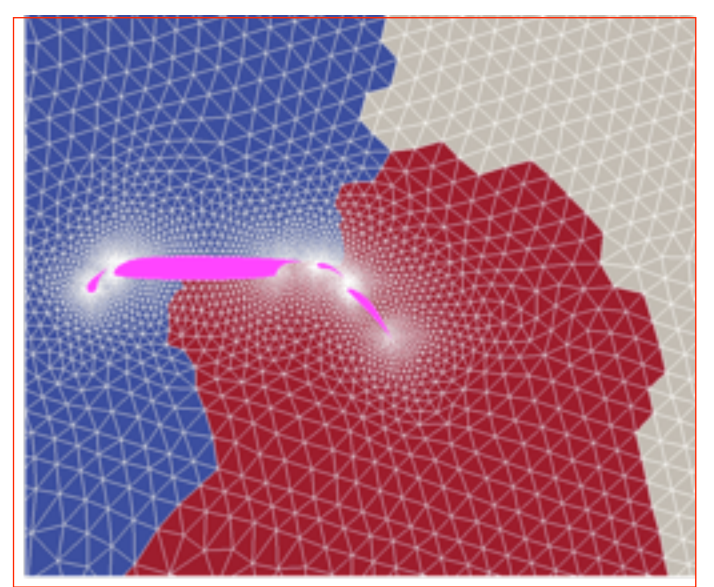
The issues we ignore on a modest CPU cluster matter even on a single GPU chip.

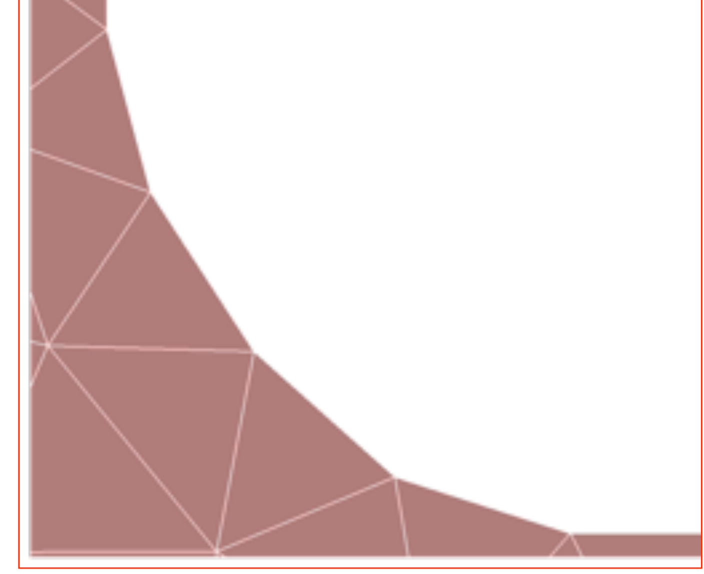
	Explicit	Semi-implicit	All Implicit
Good	Few reduction operations Very low storage	Block local preconditioners	
Bad	Reputation	Global preconditioners: CPU worthy ? (multigrid, domain decomposition) Inner-products (e.g. for computing Krylov updates)	
Ugly *(main dt restriction)*	$dt \sim \dfrac{C}{\dfrac{N^2(\lvert u\rvert + a)}{h} + \dfrac{N^4 \mu}{h^2}}$	$dt \sim \dfrac{Ch}{N^2(\lvert u\rvert + a)}$	$dt^M \sim h^{N+1}$

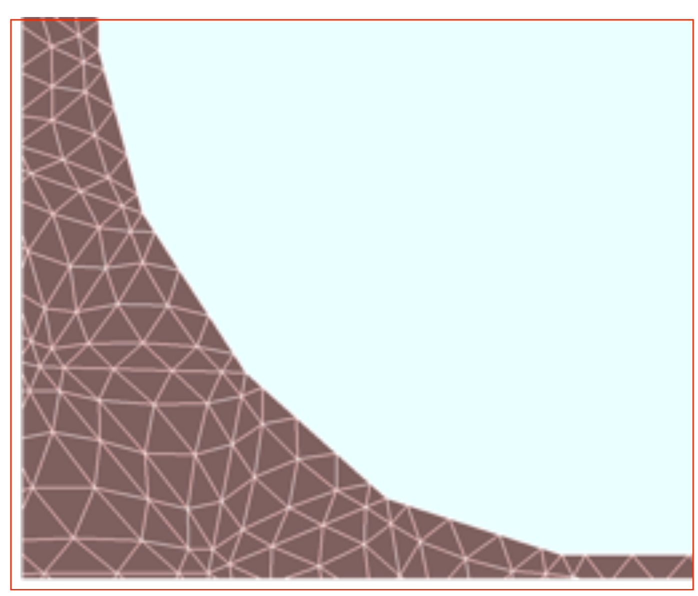
<u>Need</u>: efficient hp-Multigrid, AMG for block sparse systems on the GPU.

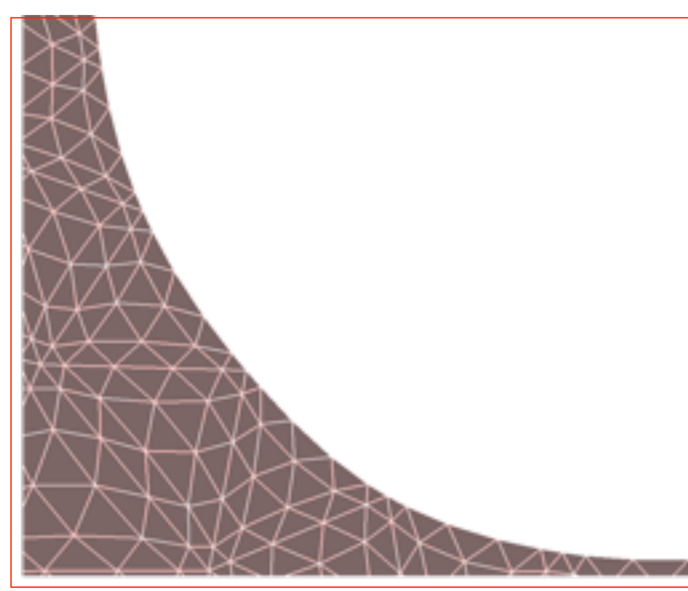
Curvilinear Flow Geometry

We use isoparametric fits to the exact geometry if available
or spline fits to the geometry otherwise

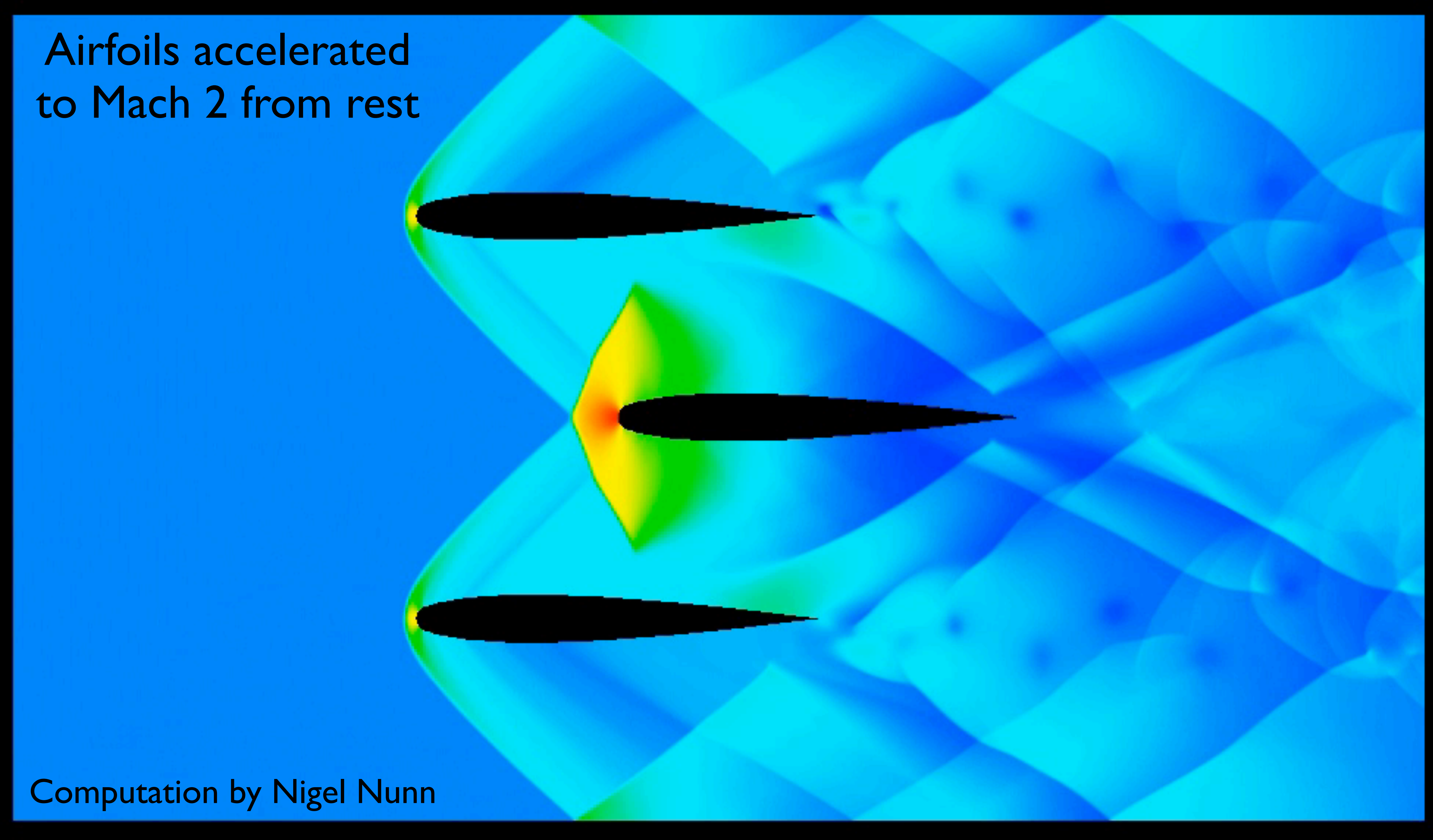








Mach 2 Flow in a Channel

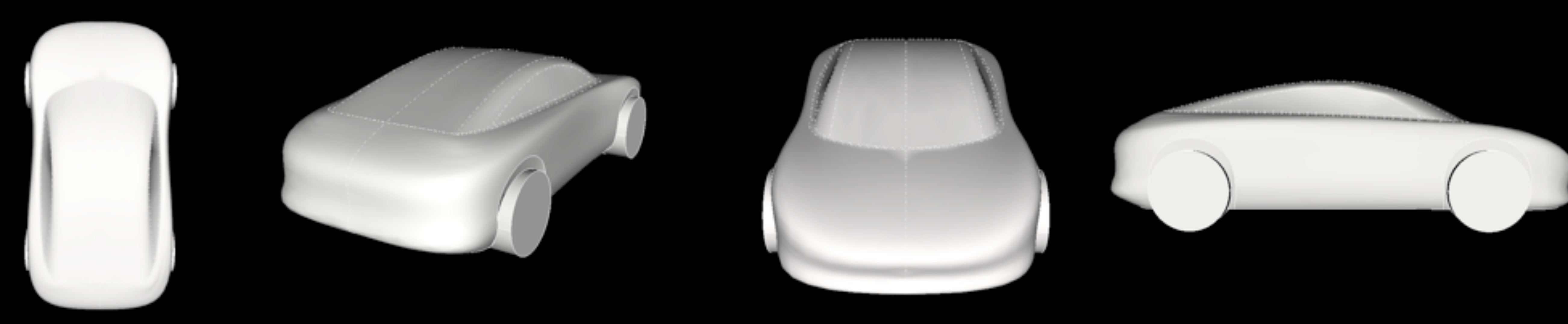


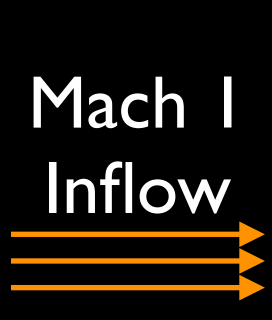
rho: (0.600, 5.543)   sqrt(RT) 1.0   Ux 2.0   nu 0.0001   mach 2   Re 20000

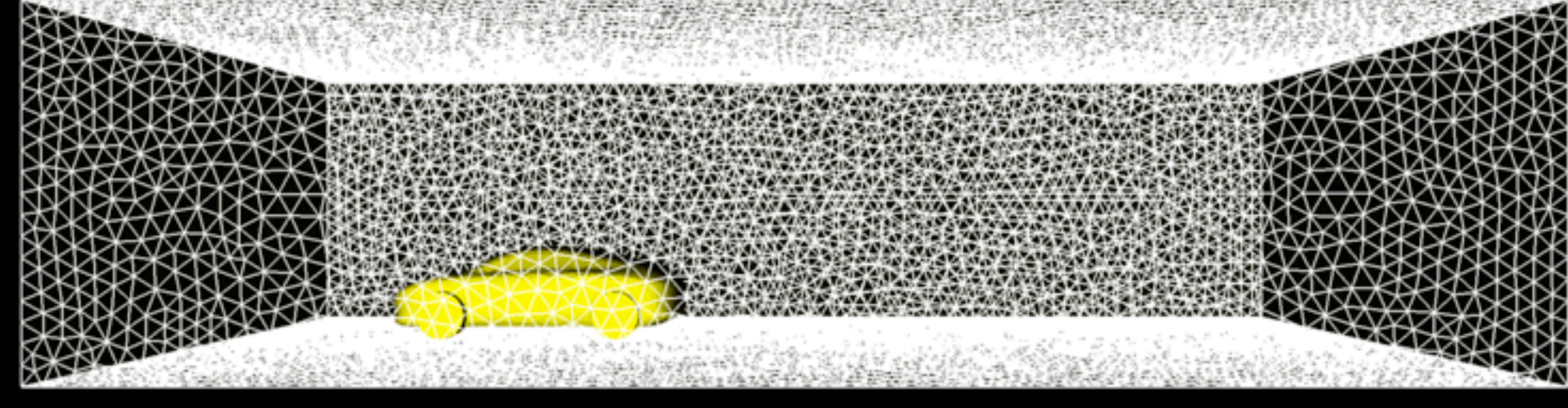
N = 6   K = 20561   dt = 1.09e-004   time 14.17   504.17 GFLOPs

3D Simulation for Fun

For fun we accelerated a model car to Mach one in our GPU powered virtual wind tunnel

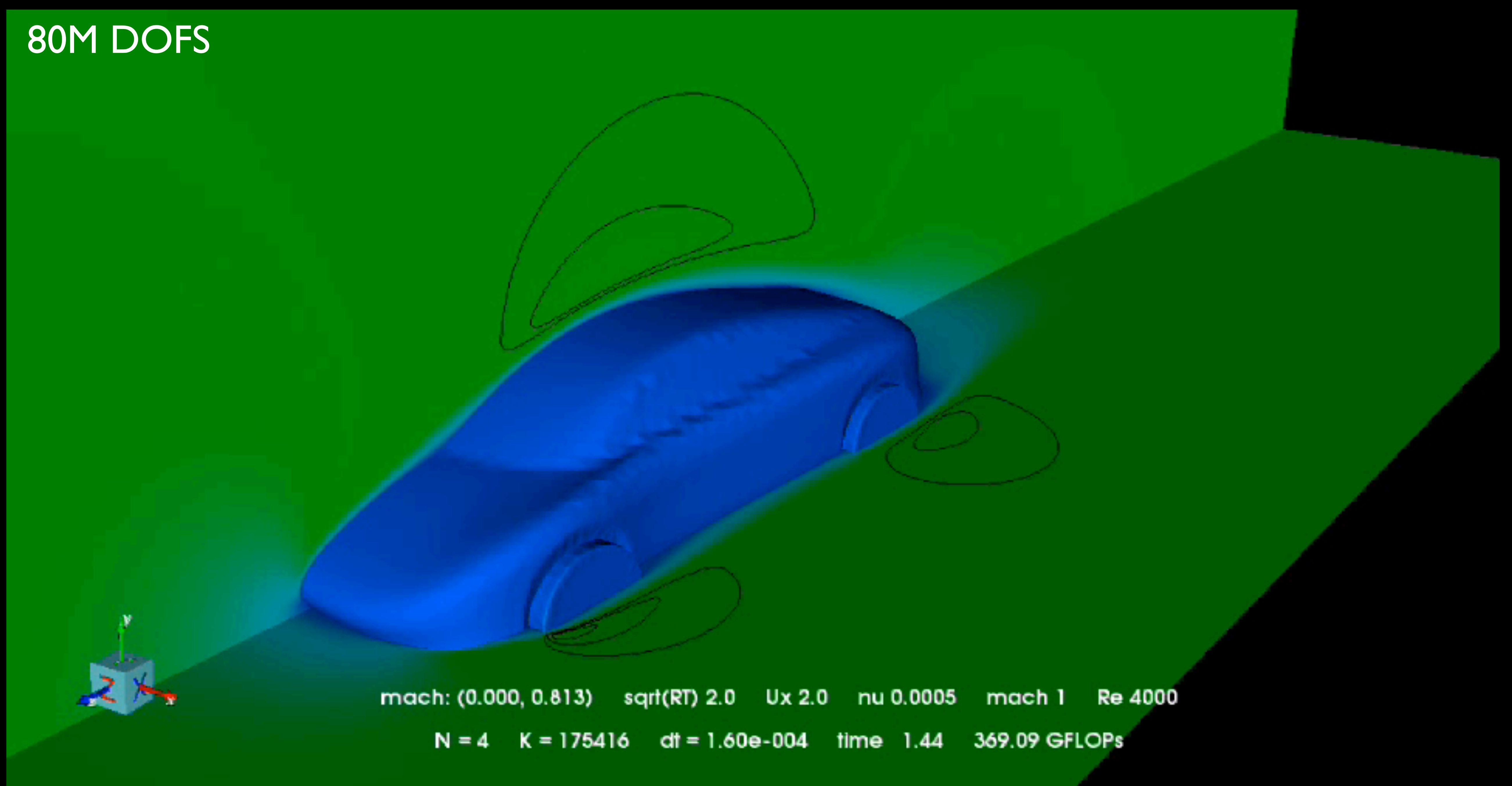






Supersonic Sedan

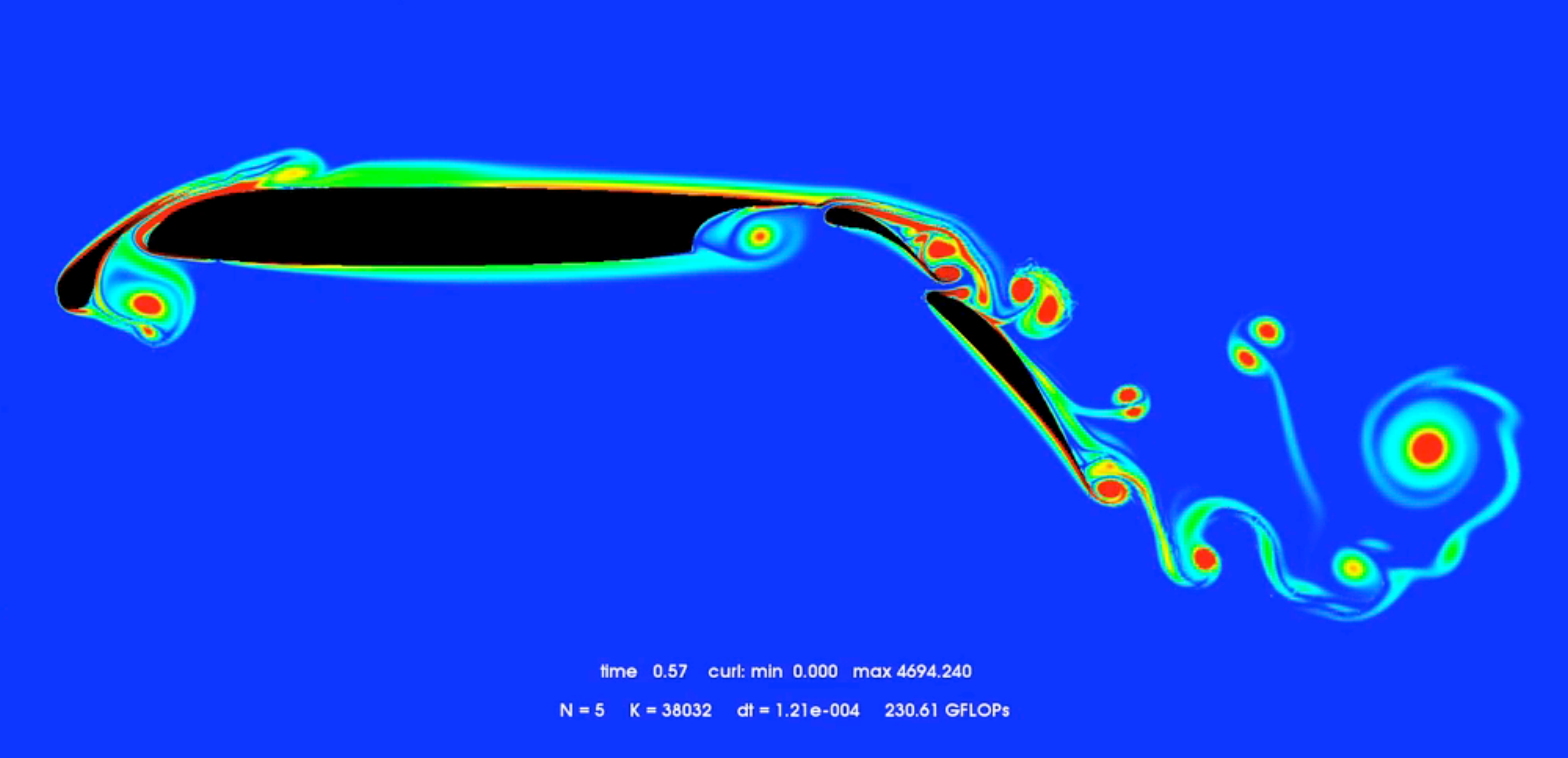
For fun we accelerated a model car to Mach one in our GPU powered virtual wind tunnel



Several unique kernels, up to 170K thread blocks, 64 threads per block, called >1e6 times.

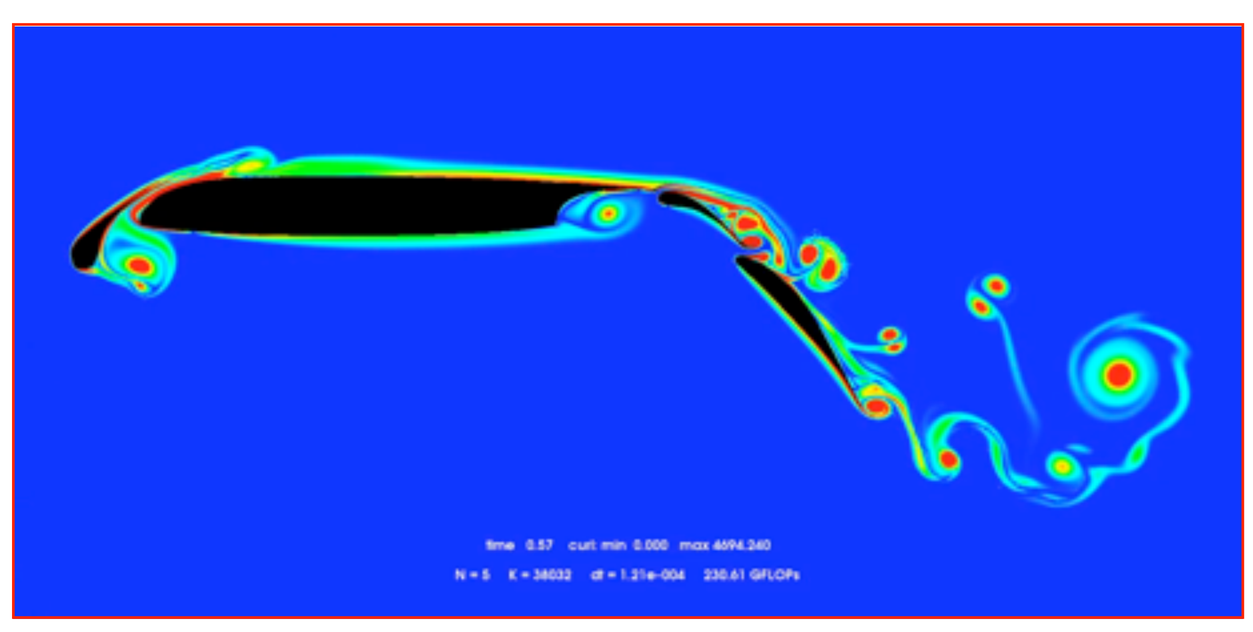
Multiple Element Airfoil Flow

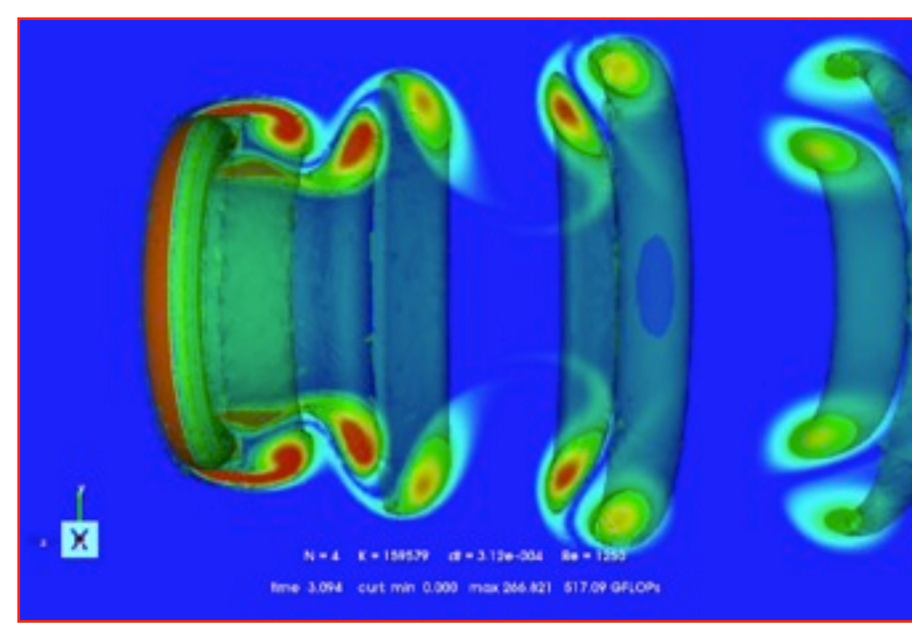
$K = 38032,\ N = 5,\ \alpha = 10^\circ$

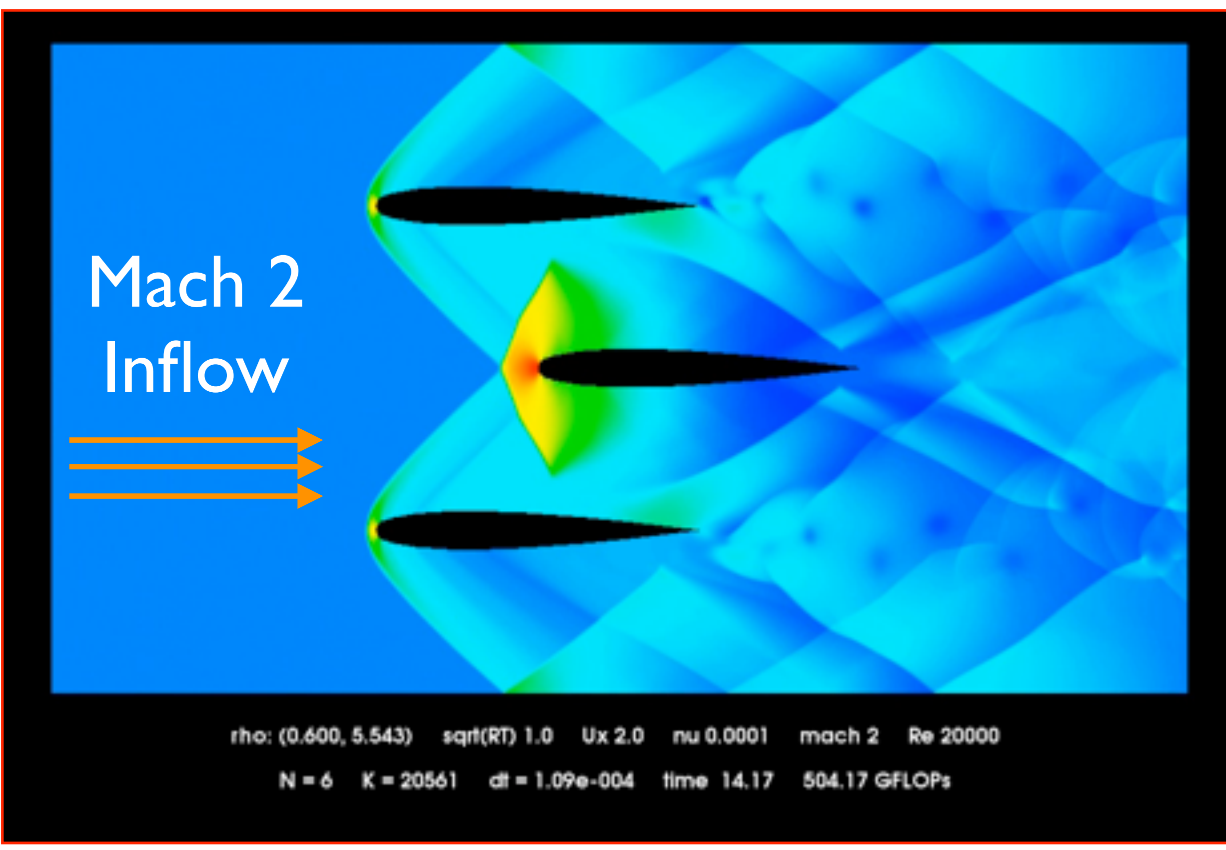


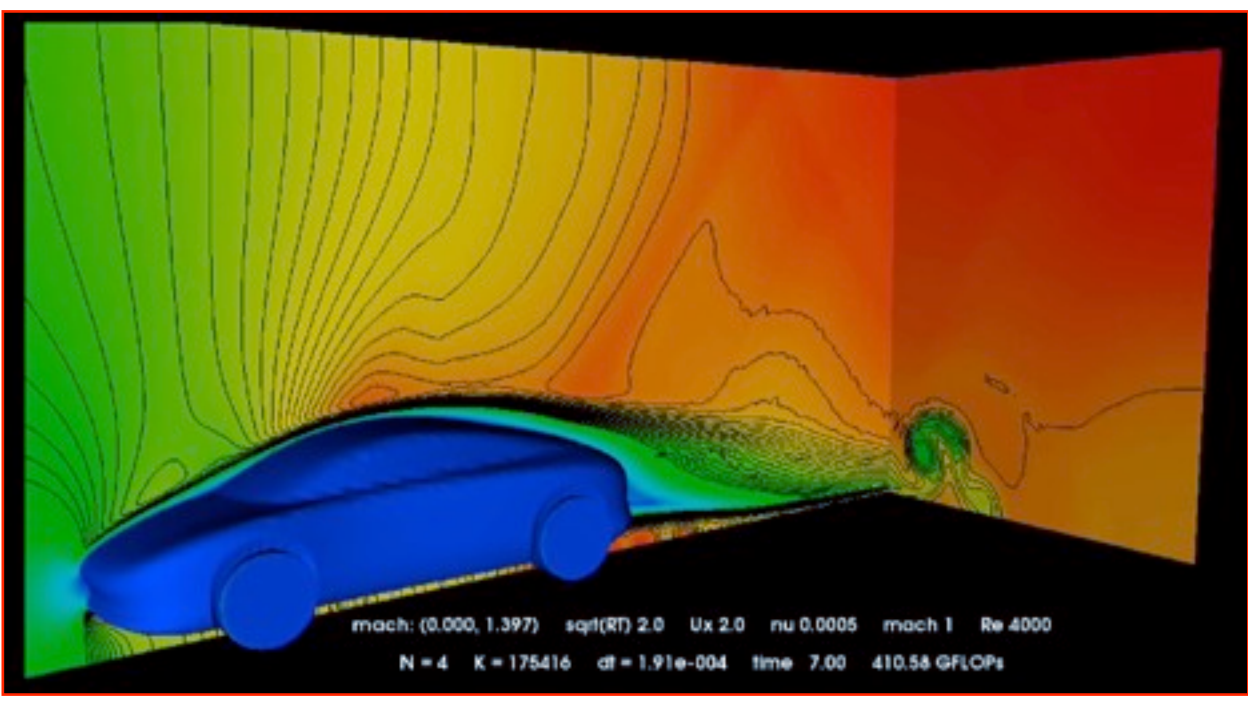
One hour on 3 Tesla class GPUs (+1 GPU rendering)

GPGPU Compressible Flow Simulations









Manual GPU kernel tuning and work partitioning becomes challenging.

Summary

- GPUs enable some high-order 3D simulations on a workstation:
  - DGTD scales to micro-threading on GPUs.
  - Local time stepping gives extra speed boost.
  - Teraflop scale work station performance.
  - A GPU customized low storage curvilinear DG.
- Main messages:
  - Many-core NUMA favors algebraically intense formulations.
  - Comparing algorithms by "operation count" alone is archaic.
- Current/future:
  - Hybrid implicit-explicit solvers on the CPU/GPU ????
  - 3D compressible Navier-Stokes with turbulence model.
  - Improved artificial viscosity based shock capturing.
  - Adaptivity in space.

INRIA-CEA-EDF: Toward petaflop numerical simulation on parallel hybrid architectures

Tim Warburton, Rice University.