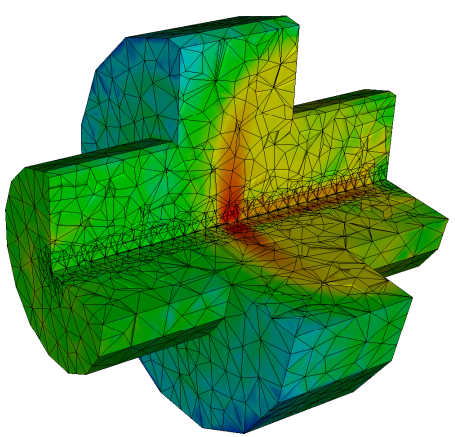
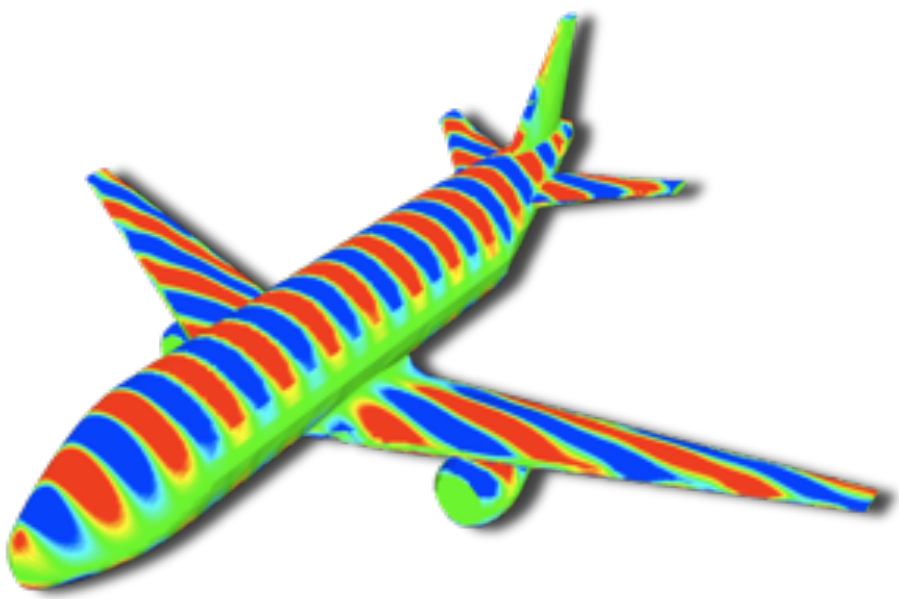
DG-FEM for PDE's
Lecture 8

Jan S Hesthaven
Brown University
Jan.Hesthaven@Brown.edu





A brief overview of what's to come

- Lecture 1: Introduction and DG in 1D
- Lecture 2: Implementation and numerical aspects
- Lecture 3: Insight through theory
- Lecture 4: Nonlinear problems
- Lecture 5/6: 2D DG-FEM, implementation and applications
- Lecture 7: Higher order/Global problems
- Lecture 8: 3D and advanced topics

Lecture 8

- ✓ Let's briefly recall what we know
- ✓ Part I: 3D problems and extensions
  - ✓ Formulations and examples
  - ✓ Adaptivity and curvilinear elements
- ✓ Part II: The need for speed
  - ✓ Parallel computing
  - ✓ GPU computing
  - ✓ Software beyond Matlab

Lets summarize

We are done with all the basics -- and we have started to see it work for us -- we know how to do

- ✓ 1D/2D problems
- ✓ Linear/nonlinear problems
- ✓ First and higher operators
- ✓ Complex geometries
- ✓ ... and we have insight into theory

All we need is 3D -- and with that comes the need for speed !

Extension to 3D ?

It is really simple at this stage !

Weak form:

$$\int_{\mathrm{D}^k} \left[\frac{\partial u_h^k}{\partial t} \ell_n^k(\boldsymbol{x}) - \boldsymbol{f}_h^k \cdot \nabla \ell_n^k(\boldsymbol{x}) \right] d\boldsymbol{x} = - \oint_{\partial \mathrm{D}^k} \hat{\boldsymbol{n}} \cdot \boldsymbol{f}^* \ell_n^k(\boldsymbol{x}) \, d\boldsymbol{x},$$

Strong form:

$$\int_{\mathrm{D}^k} \left[\frac{\partial u_h^k}{\partial t} + \nabla \cdot \boldsymbol{f}_h^k \right] \ell_n^k(\boldsymbol{x}) \, d\boldsymbol{x} = \oint_{\partial \mathrm{D}^k} \hat{\boldsymbol{n}} \cdot \left[\boldsymbol{f}_h^k - \boldsymbol{f}^* \right] \ell_n^k(\boldsymbol{x}) \, d\boldsymbol{x},$$

$$\boldsymbol{f}^* = \{\{\boldsymbol{f}_h(\boldsymbol{u}_h)\}\} + \frac{C}{2} [\![\boldsymbol{u}_h]\!]. \qquad C = \max_u \left| \lambda \left(\hat{\boldsymbol{n}} \cdot \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}} \right) \right|,$$

Nothing is essential new

Extension to 3D

Apart from the 'logistics' all we need to worry about is to choose our element and how to represent the solution

$$u(\boldsymbol{r}) \simeq u_h(\boldsymbol{r}) = \sum_{n=1}^{N_p} \hat{u}_n \psi_n(\boldsymbol{r}) = \sum_{i=1}^{N_p} u(\boldsymbol{r}_i)\ell_i(\boldsymbol{r}),$$

$$\boldsymbol{u} = \mathcal{V}\hat{\boldsymbol{u}}, \;\; \mathcal{V}^T\boldsymbol{\ell}(\boldsymbol{r}) = \boldsymbol{\psi}(\boldsymbol{r}), \;\; \mathcal{V}_{ij} = \psi_j(\boldsymbol{r}_i).$$

We need points

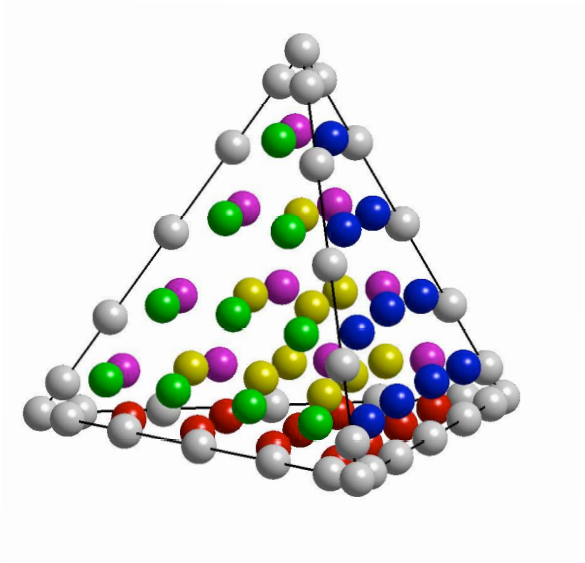
$$N_p = \frac{(N+1)(N+2)(N+3)}{6},$$

We need an orthonormal basis

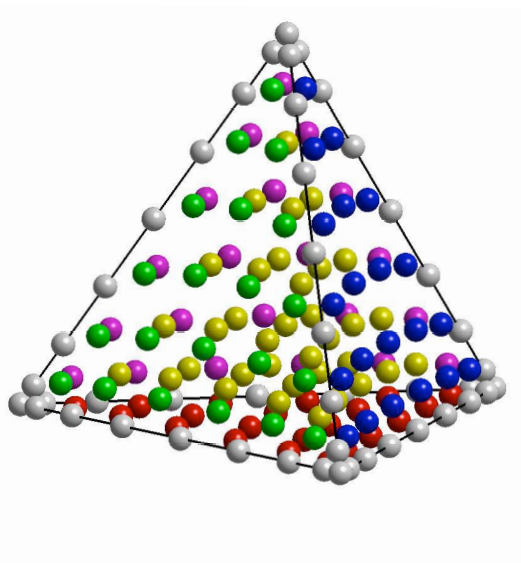
$$\psi_{ijk}(r,s,t) = 2\sqrt{2}P_i^{(0,0)}(a)P_j^{(2i+1,0)}(b)P_k^{(2i+2j+2,0)}(b)(1-b)^i(1-c)^{i+j},$$

Extension to 3D

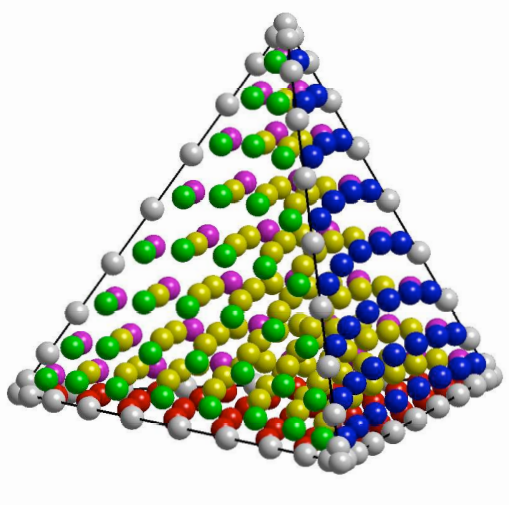
For other element types, one simply need to define nodes and modes for that elements

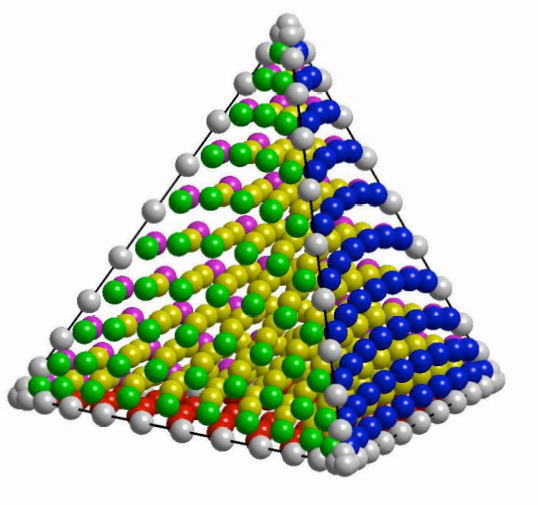


: 6



$N = 8$





Extension to 3D

Everything is identical in spirit

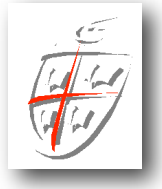
Mass matrix $$\mathcal{M}^k = J^k(\mathcal{V}\mathcal{V}^T)^{-1}.$$

Diff matrix $$\mathcal{D}_r\mathcal{V} = \mathcal{V}_r, \;\; \mathcal{D}_s\mathcal{V} = \mathcal{V}_s, \;\; \mathcal{D}_t\mathcal{V} = \mathcal{V}_t,$$

Derivative
$$\frac{\partial}{\partial x} = \frac{\partial r}{\partial x}\mathcal{D}_r + \frac{\partial s}{\partial x}\mathcal{D}_s + \frac{\partial t}{\partial x}\mathcal{D}_t,$$
$$\frac{\partial}{\partial y} = \frac{\partial r}{\partial y}\mathcal{D}_r + \frac{\partial s}{\partial y}\mathcal{D}_s + \frac{\partial t}{\partial y}\mathcal{D}_t,$$
$$\frac{\partial}{\partial z} = \frac{\partial r}{\partial z}\mathcal{D}_r + \frac{\partial s}{\partial z}\mathcal{D}_s + \frac{\partial t}{\partial z}\mathcal{D}_t,$$

Stiffness matrix $$\mathcal{S}_r = \mathcal{M}^{-1}\mathcal{D}_r, \;\; \mathcal{S}_s = \mathcal{M}^{-1}\mathcal{D}_s, \;\; \mathcal{S}_t = \mathcal{M}^{-1}\mathcal{D}_t.$$

Example - Maxwell's equations



Consider Maxwell's equations

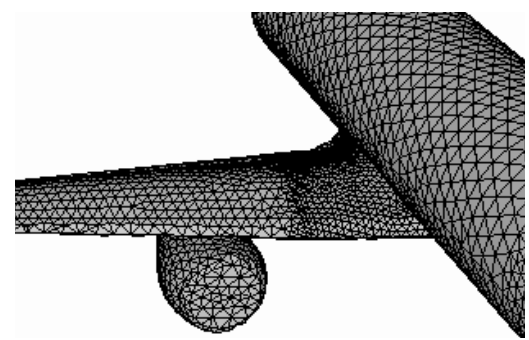
$$\varepsilon \partial_t E - \nabla \times H = -j, \qquad \mu \partial_t H + \nabla \times E = 0,$$

Write it on conservation form as

$$\frac{\partial q}{\partial t} + \nabla \cdot F = -J \qquad F = \begin{bmatrix} -\hat{e} \times H \\ \hat{e} \times E \end{bmatrix} \qquad q = \begin{bmatrix} E \\ H \end{bmatrix}$$

Represent the solution as

$$\Omega = \sum_k D^k \qquad q_N = \sum_{i=1}^{N} q(\mathbf{x}_i, t) L_i(\mathbf{x})$$



and assume

$$\int_D \left(\frac{\partial \boldsymbol{q}_N}{\partial t} + \nabla \cdot \boldsymbol{F}_N - \boldsymbol{J}_N \right) L_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \oint_{\partial D} L_i(\boldsymbol{x}) \hat{\boldsymbol{n}} \cdot [\boldsymbol{F}_N - \boldsymbol{F}^*] \, \mathrm{d}\boldsymbol{x}.$$

Example - Maxwell's equations

On each element we then define

$$\hat{M}_{ij} = \int_D L_i L_j \,\mathrm{d}\boldsymbol{x}, \quad \hat{S}_{ij} = \int_D \nabla L_j L_i \,\mathrm{d}\boldsymbol{x}, \quad \hat{F}_{ij} = \oint_{\partial D} L_i L_j \,\mathrm{d}\boldsymbol{x},$$

With the numerical flux given as

$$\hat{\boldsymbol{n}} \cdot [\boldsymbol{F} - \boldsymbol{F}^*] = \begin{cases} \boldsymbol{n} \times (\gamma \boldsymbol{n} \times [\boldsymbol{E}] - [\boldsymbol{B}]), \\ \boldsymbol{n} \times (\gamma \boldsymbol{n} \times [\boldsymbol{B}] + [\boldsymbol{E}]), \end{cases} \qquad [Q] = Q^- - Q^+$$

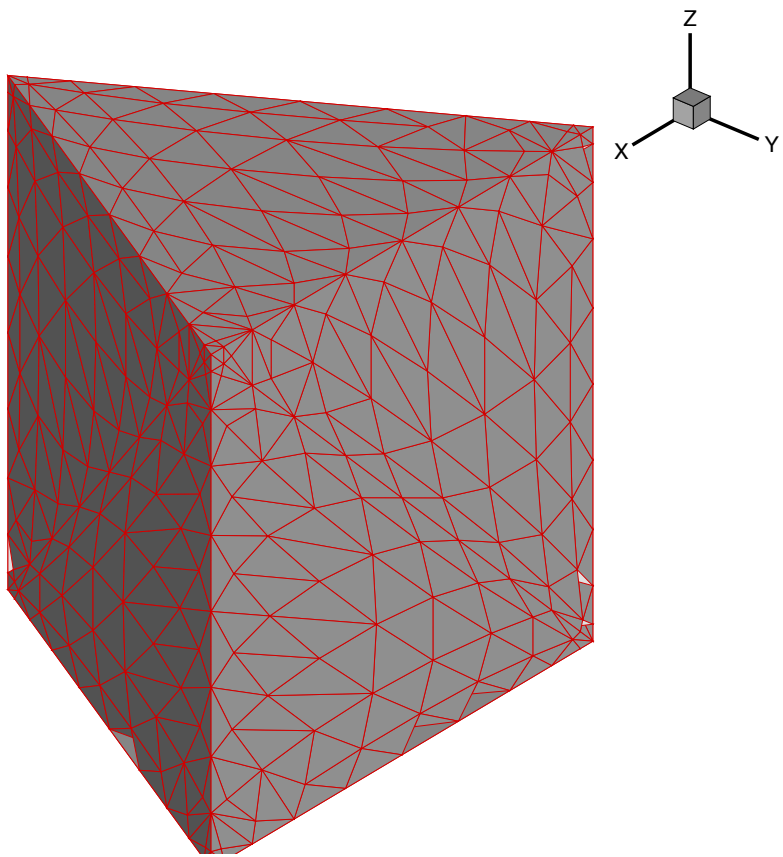
To obtain the local matrix based scheme

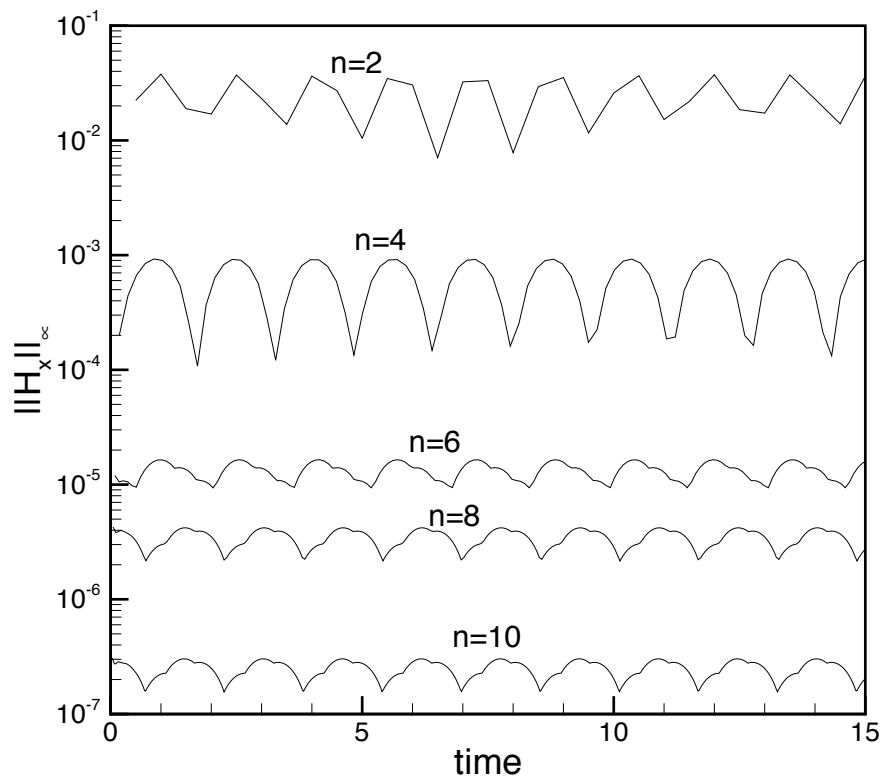
$$\hat{M} \frac{\mathrm{d}\hat{\boldsymbol{q}}}{\mathrm{d}t} + \hat{S} \cdot \hat{\boldsymbol{F}} - \hat{M}\hat{\boldsymbol{J}} = \hat{F}\hat{\boldsymbol{n}} \cdot [\hat{\boldsymbol{F}} - \hat{\boldsymbol{F}}^*],$$

One then typically uses an explicit Runge-Kutta to advance in time - just like 1D/2D.

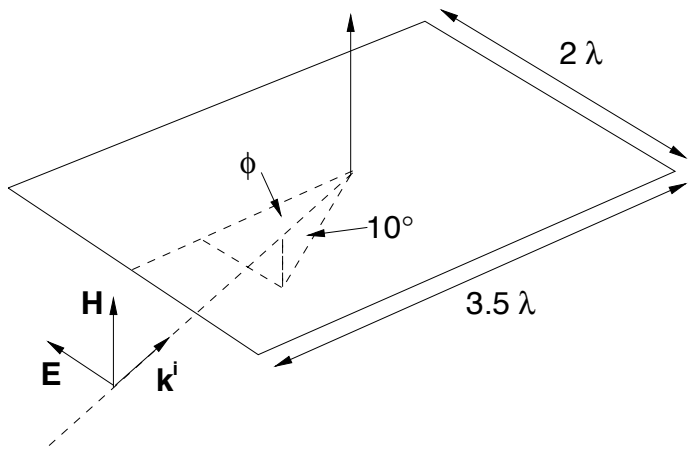
An example - Maxwell's equations

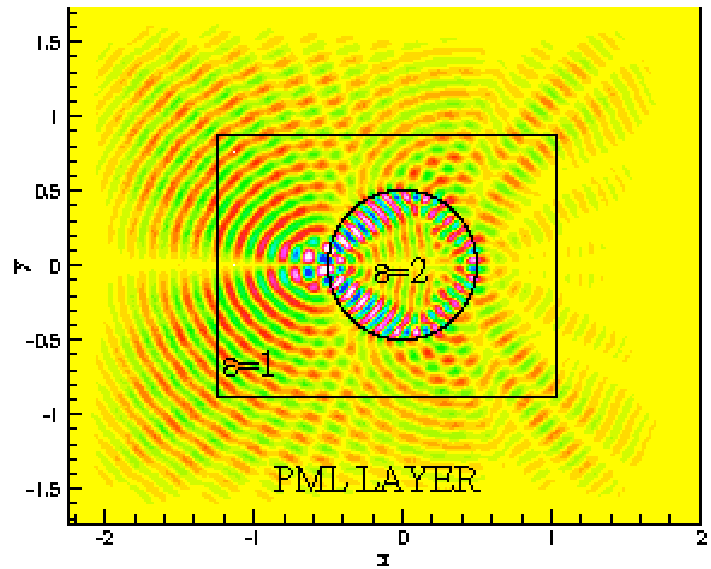
Simple wave propagation

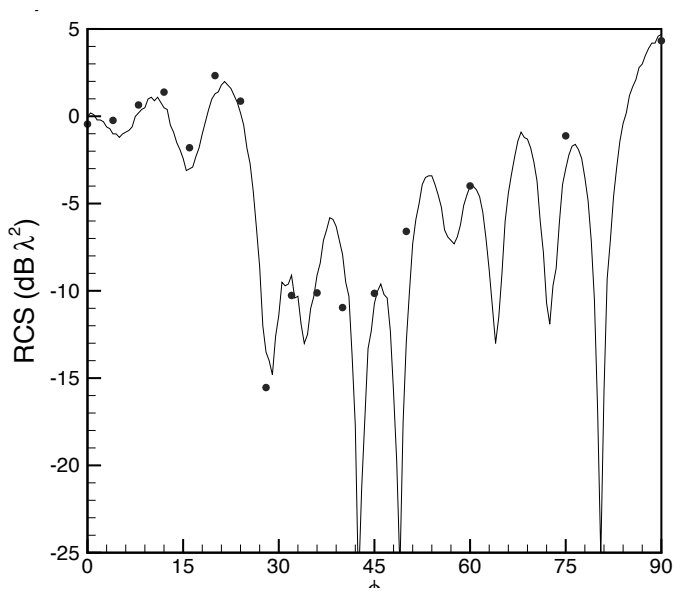


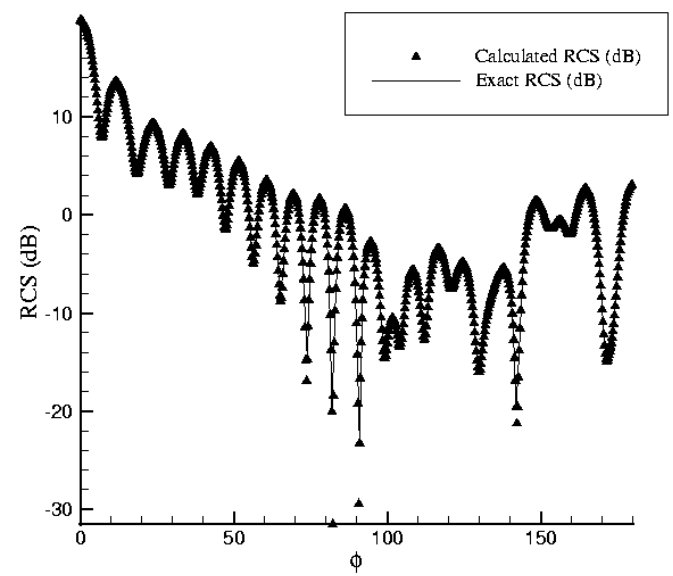


An example - Maxwell's equations

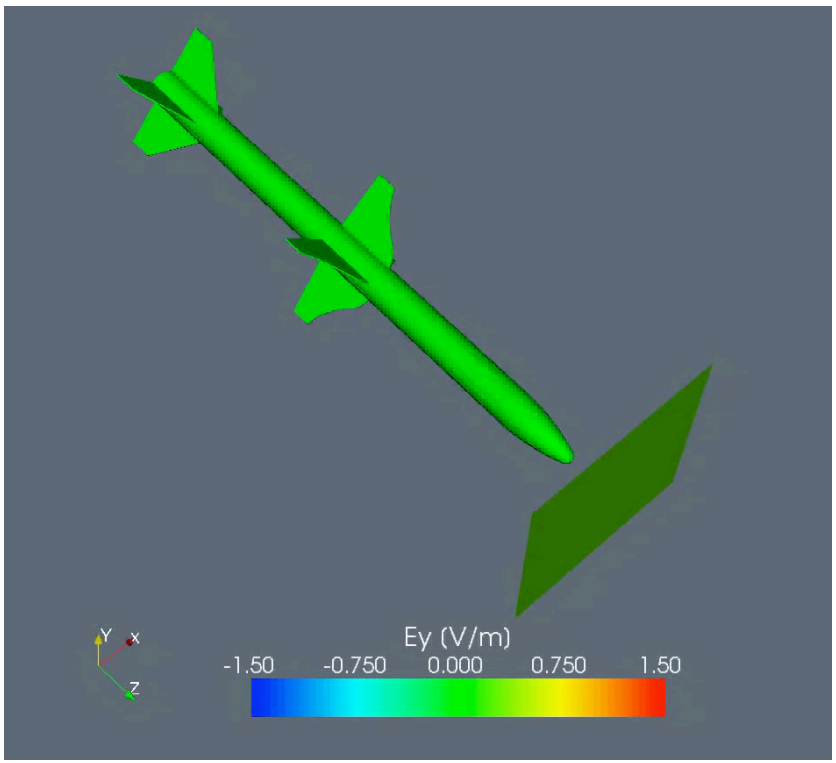


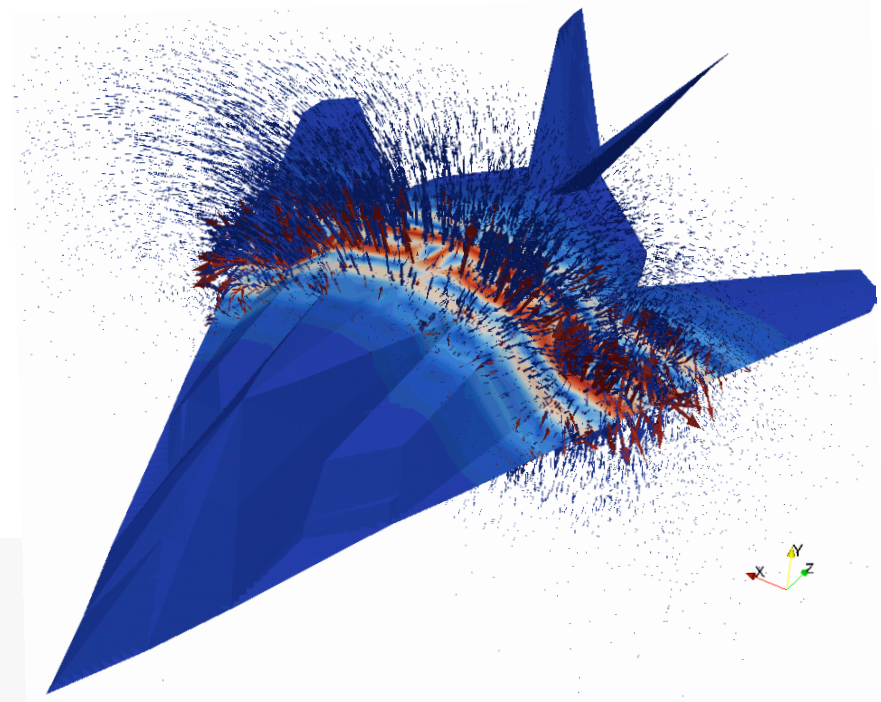


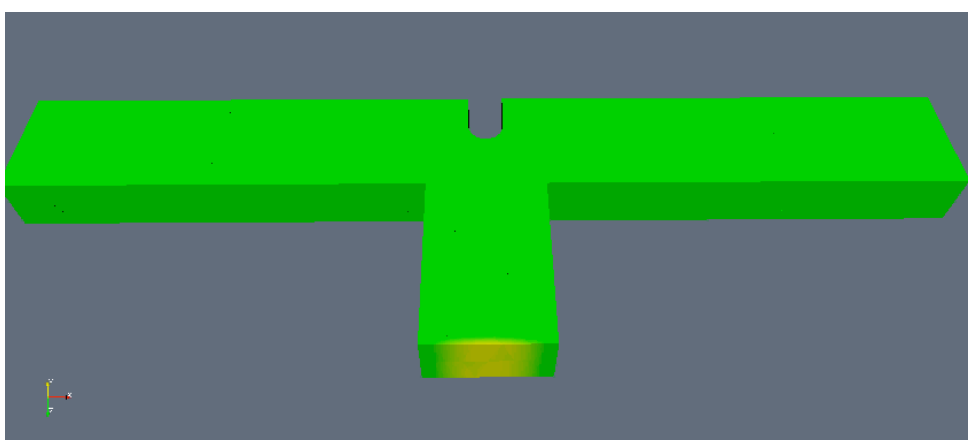


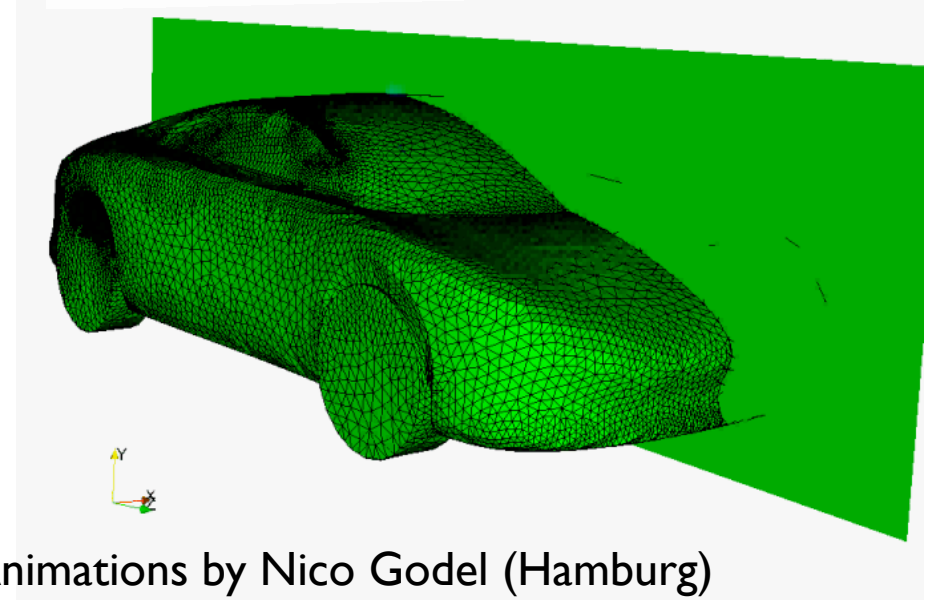


An example - Maxwell's equations







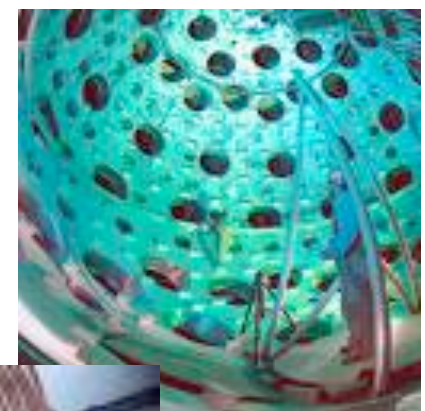


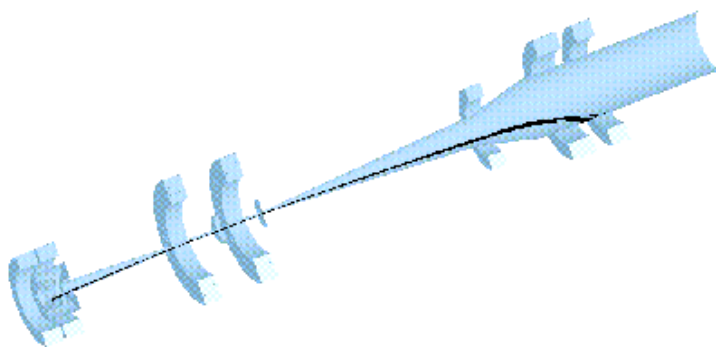
Animations by Nico Godel (Hamburg)

Kinetic Plasma Physics

Important applications

- ✓ High-power/High-frequency microwave generation
- ✓ Particle accelerators
- ✓ Laser-matter interaction
- ✓ Fusion applications, e.g., plasma edge
- ✓ etc





Kinetic Plasma Physics

In high-speed plasma problems dominated by kinetic effects, one needs to solve for f(x,p,t) - 6D+1

Vlasov/Boltzmann equation

$$\partial_t f + v \cdot \partial_x f + q(E + v \times B) \cdot \partial_p f = \langle \text{Sources} \rangle - \langle \text{Sinks} \rangle.$$

Maxwell's equations

$$\begin{aligned} \partial_t E - \frac{1}{\varepsilon} \nabla \times H &= -\frac{j}{\varepsilon}, \\ \partial_t H + \frac{1}{\mu} \nabla \times E &= 0, \\ \nabla \cdot H = 0, \qquad \nabla \cdot E &= \frac{\rho}{\varepsilon}. \end{aligned}$$

Coupled through $\rho := \int f \, dv, \qquad j := \int vf \, dv.$

Particle-in-Cell (PIC) Methods

This is an attempt to solve the Vlasov/Boltzmann equation by sampling with P particles

$$f(x,p,t) = \sum_{n=1}^{P} q_n S(x - x_n(t))\delta(p - p_n(t)),$$

$$\rho(x,t) = \sum_{n=1}^{P} q_n S(x - x_n(t)), \quad j(x,t) = \sum_{n=1}^{P} v_n q_n S(x - x_n(t))$$

Ideally we have

$$S(x) = \delta(x) \longleftarrow$$ a point particle

However, this is not practical, nor reasonable - so S(x) is a **shape-function**

Particle-in-Cell Methods

Maxwell's equations

$$\varepsilon \partial_t E - \nabla \times H = -j, \quad \mu \partial_t H + \nabla \times E = 0,$$
$$\nabla \cdot (\varepsilon E) = \rho, \quad \nabla \cdot (\mu H) = 0,$$

Particle/Phase dynamics

$$\frac{dx_n}{dt} = v_n(t) \qquad \frac{dmv_n}{dt} = q_n(E + v_n \times H) \qquad m = \frac{1}{\sqrt{1 - (v_n/c)^2}}$$

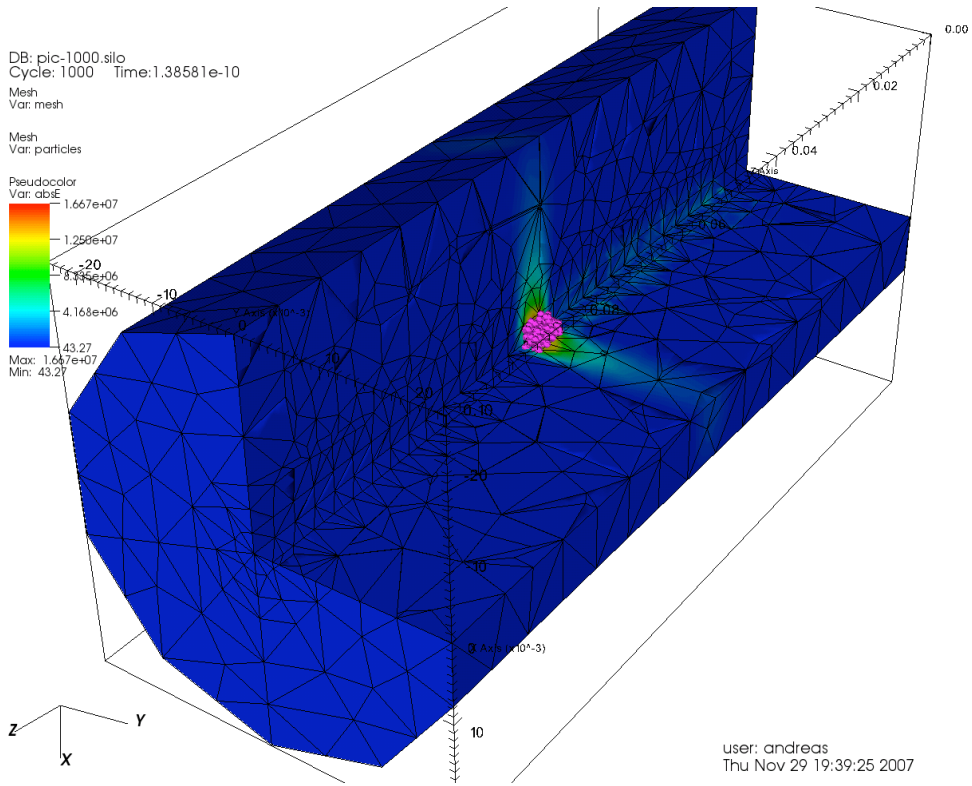
Particles-to-fields

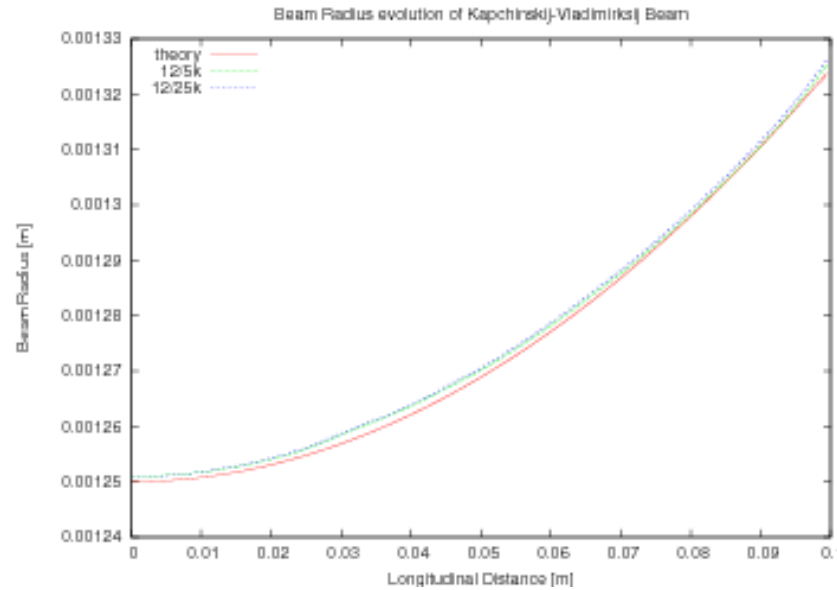
$$\rho(x,t) = \sum_{n=1}^{P} q_n S(x - x_n(t)), \quad j(x,t) = \sum_{n=1}^{P} v_n q_n S(x - x_n(t))$$

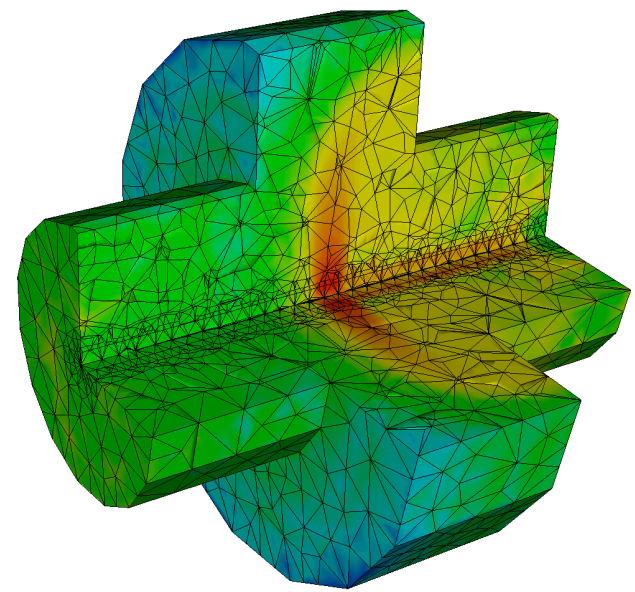
Fields-to-particles

$$E(x_n), H(x_n)$$

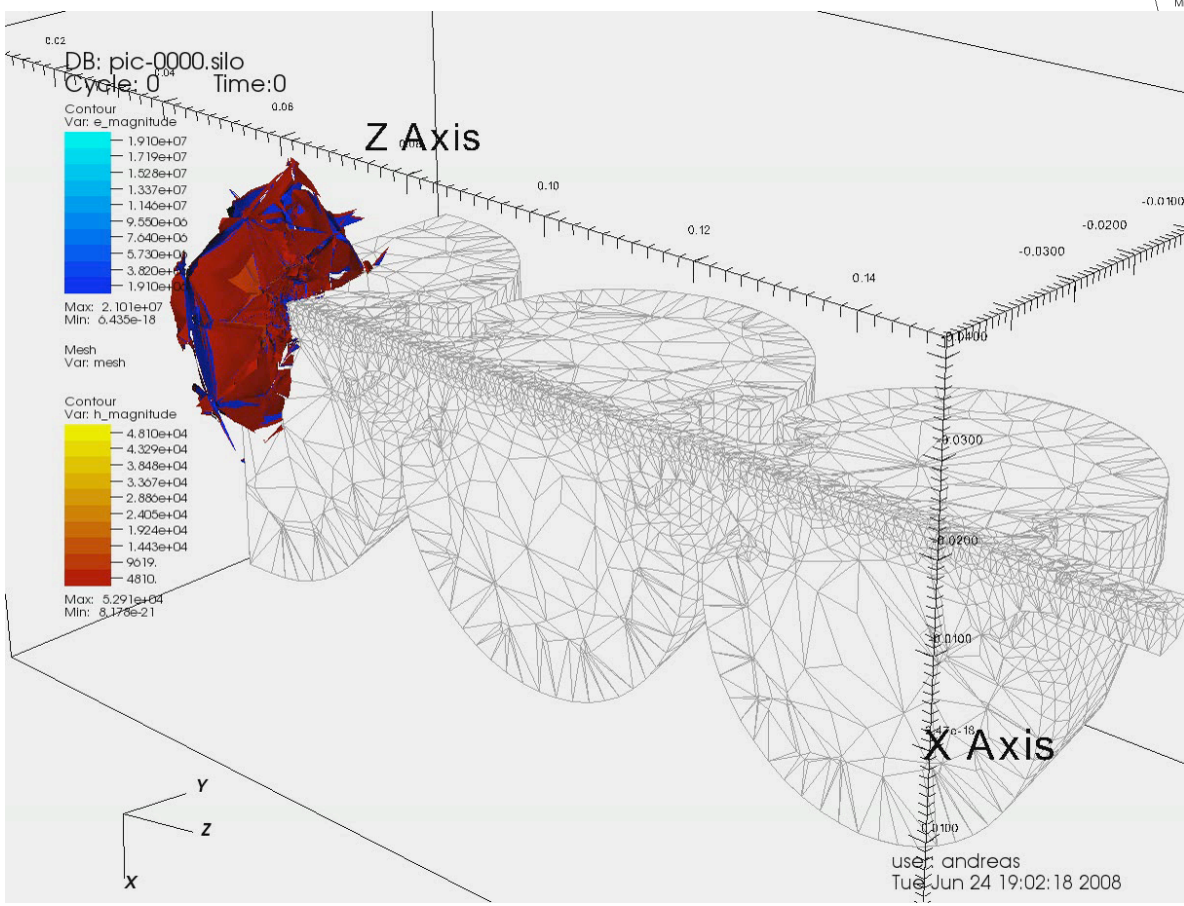
Some 3D results

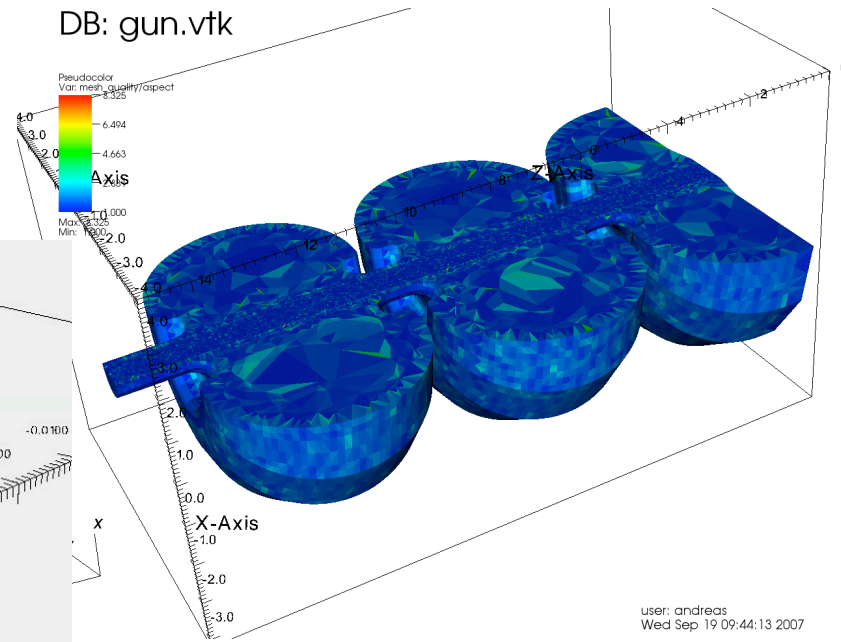






Particle gun





Summary of Part I

We have generalized everything to 3D

✓ Linear/nonlinear problems
✓ First order/higher order operators
✓ Complex geometries

There is only one significant obstacle to solving large problems

SPEED !

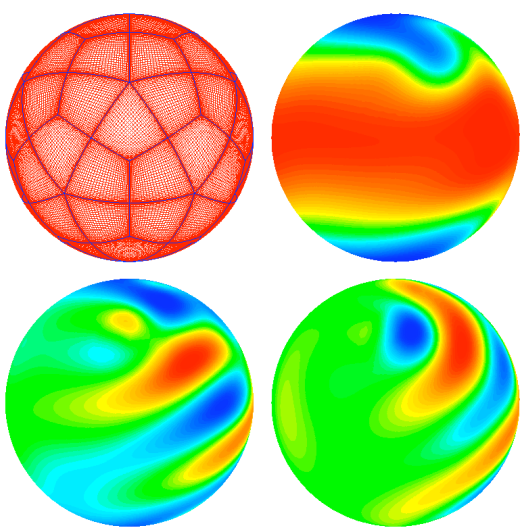
Lecture 8

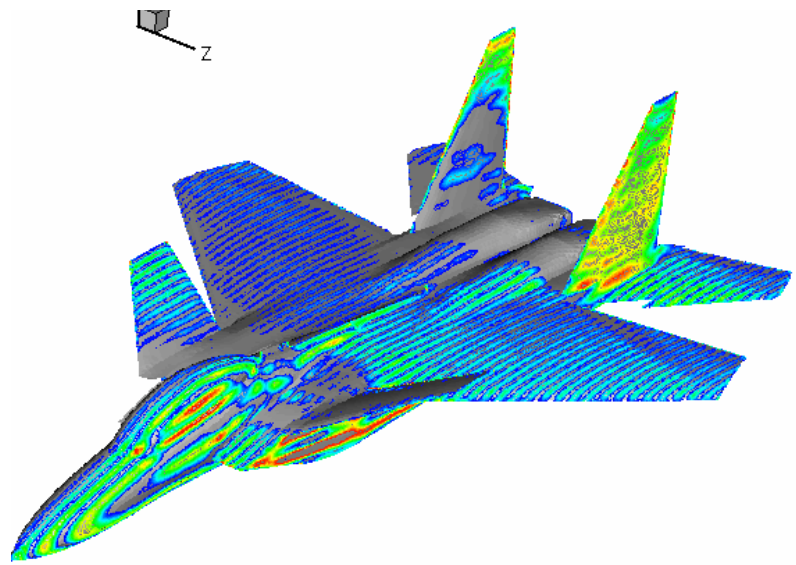
- ✓ Let's briefly recall what we know
- ✓ Part I: 3D problems and extensions
  - ✓ Formulations and examples
  - ✓ Adaptivity and curvilinear elements
- ✓ Part II: The need for speed
  - ✓ Parallel computing
  - ✓ GPU computing
  - ✓ Software beyond Matlab

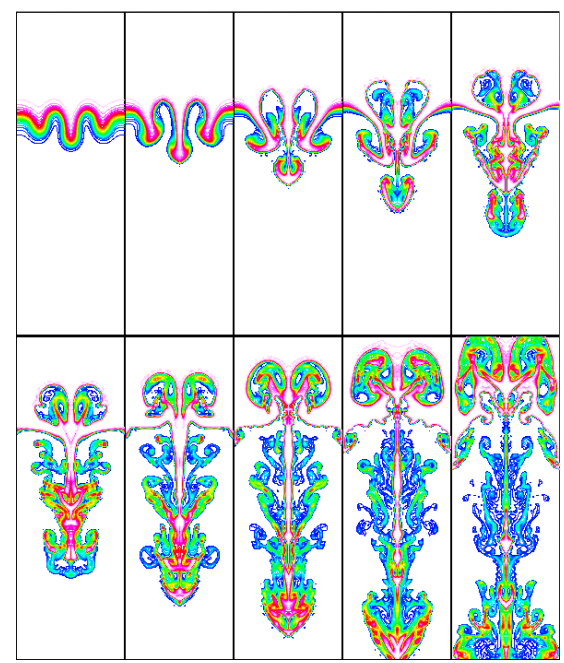
The need for speed !

So far, we have focused on ‘simple’ serial computing using Matlab based model.

However, this will not suffice for many applications

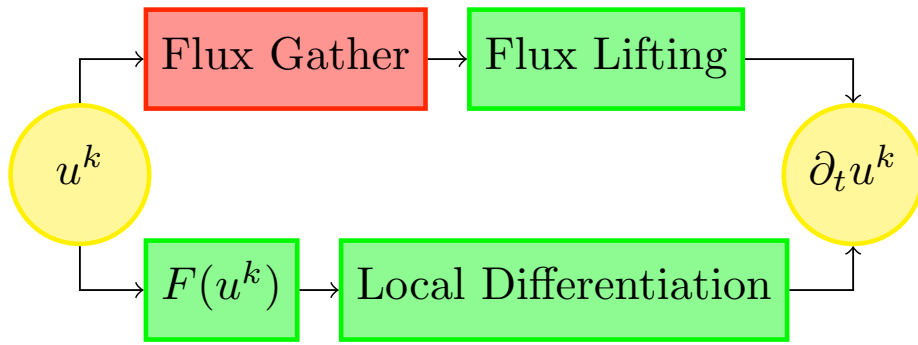


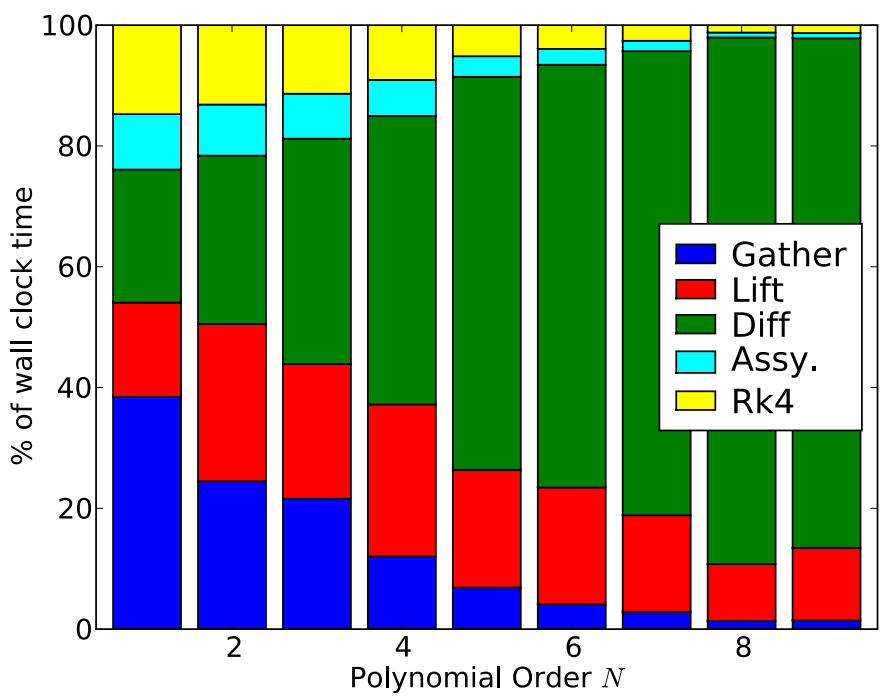




The need for speed

Let us first understand where we spend the time





Test case is
3D Maxwells

The majority of
work is local

The need for speed

The locality suggest that parallel computing will be beneficial

✓ Using OpenMP, the local work can be distributed over elements through loops.

✓ Using MPI the locality ensures a surface communication model.

✓ Mixed OpenMP/MPI models also possible

✓ A similar line of arguments can be used for iterative solvers.

Parallel performance

# Processors	64	128	256	512
Scaled RK time	1.00	0.48	0.24	0.14
Ideal time	1.00	0.50	0.25	0.13

High performance is achieved through -

- ✓ Local nature of scheme
- ✓ Pure matrix-matrix operations
- ✓ Local bandwidth minimization
- ✓ Very efficient on-chip performance (~75%)

Challenges -

- ✓ Efficient parallel preconditioning

Parallel computing

DG-FEM maps very well to classic multi-processor computing clusters and result in excellent speed-up.

... but such machines are expensive to buy and run.

Ex: To get on the Top500 list, requires about $3m to purchase a cluster with 16-18Tflop/s performance.

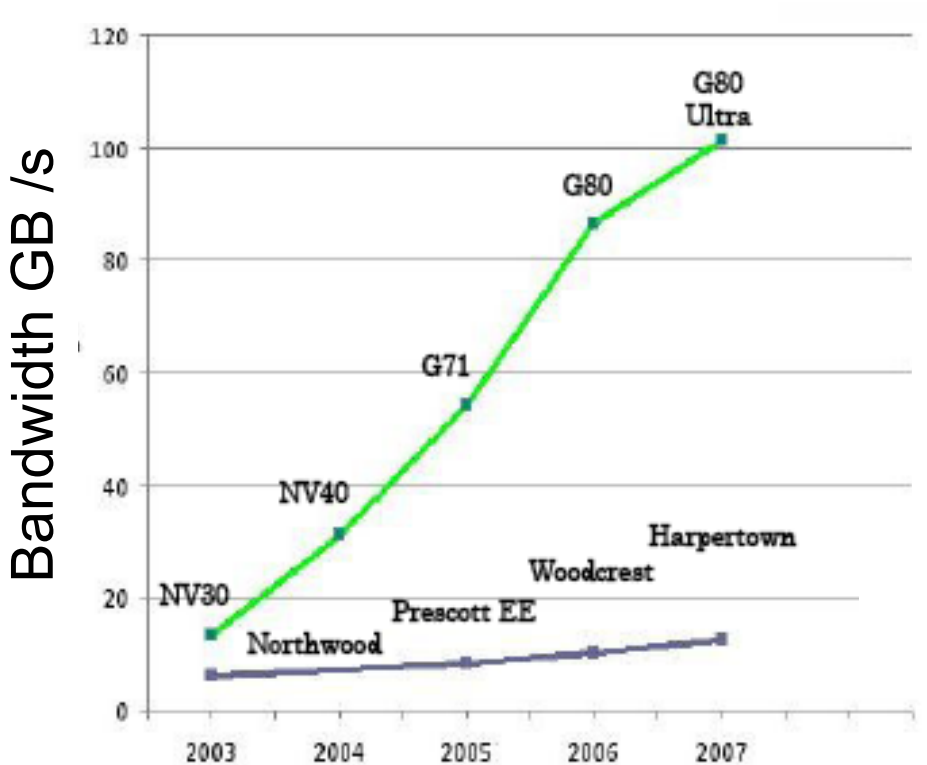
What we need is supercomputing on the desktop

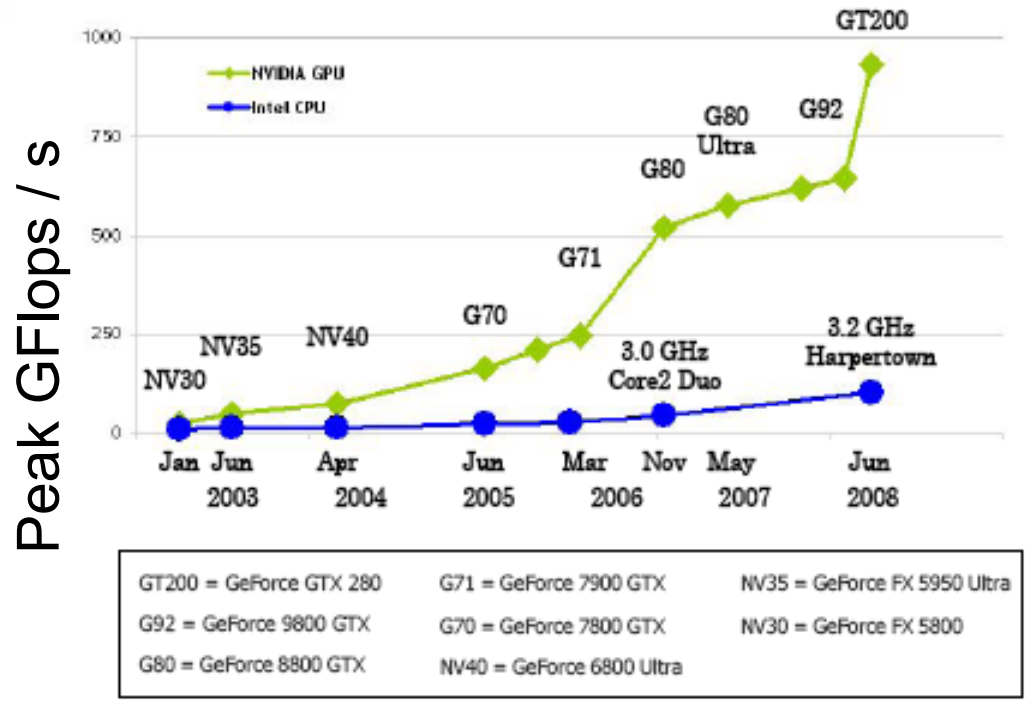
For FREE !

... or at least at a fraction of the price

CPUs vs GPUs

Notice the following





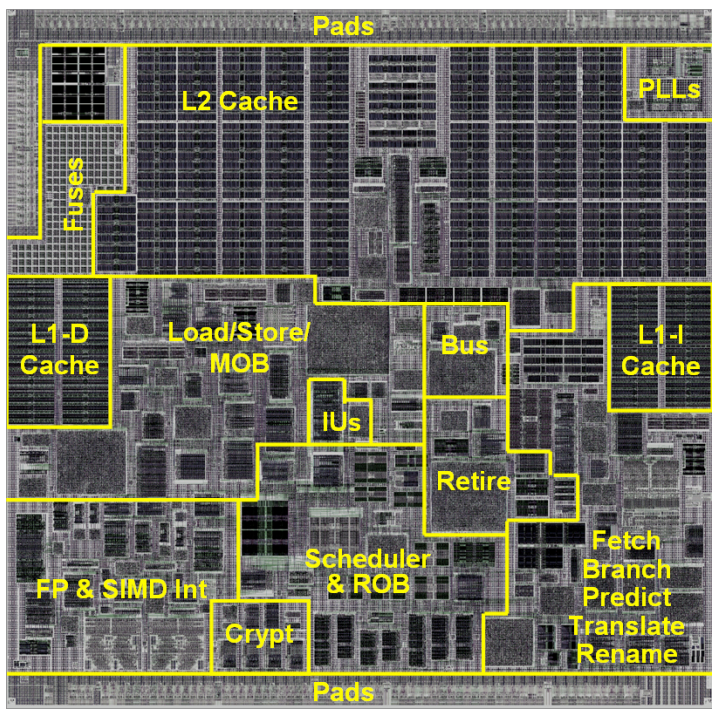
The memory bandwidth and the peak performance on Graphics cards (GPU's) is developing MUCH faster than on CPU's

At the same time, the mass-marked for gaming drives the prices down -- we have to find a way to exploit this !

But why is this ?

Target for CPU:

- ✓ Single thread very fast
- ✓ Large caches to hide latency
- ✓ Predict, speculate etc

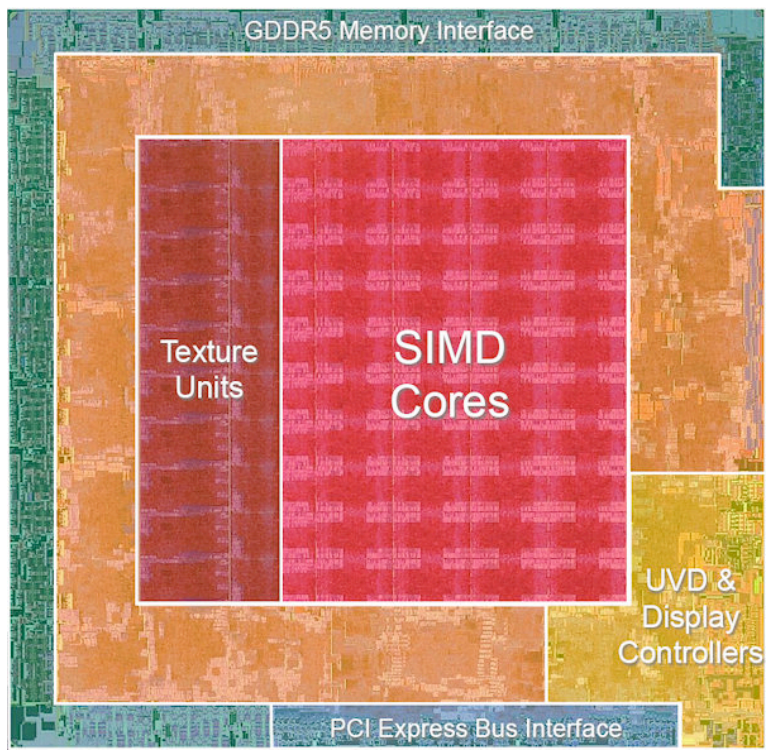


Lots of very complex logic to predict behavior

But why is this ?

For streaming/graphics cards it is different

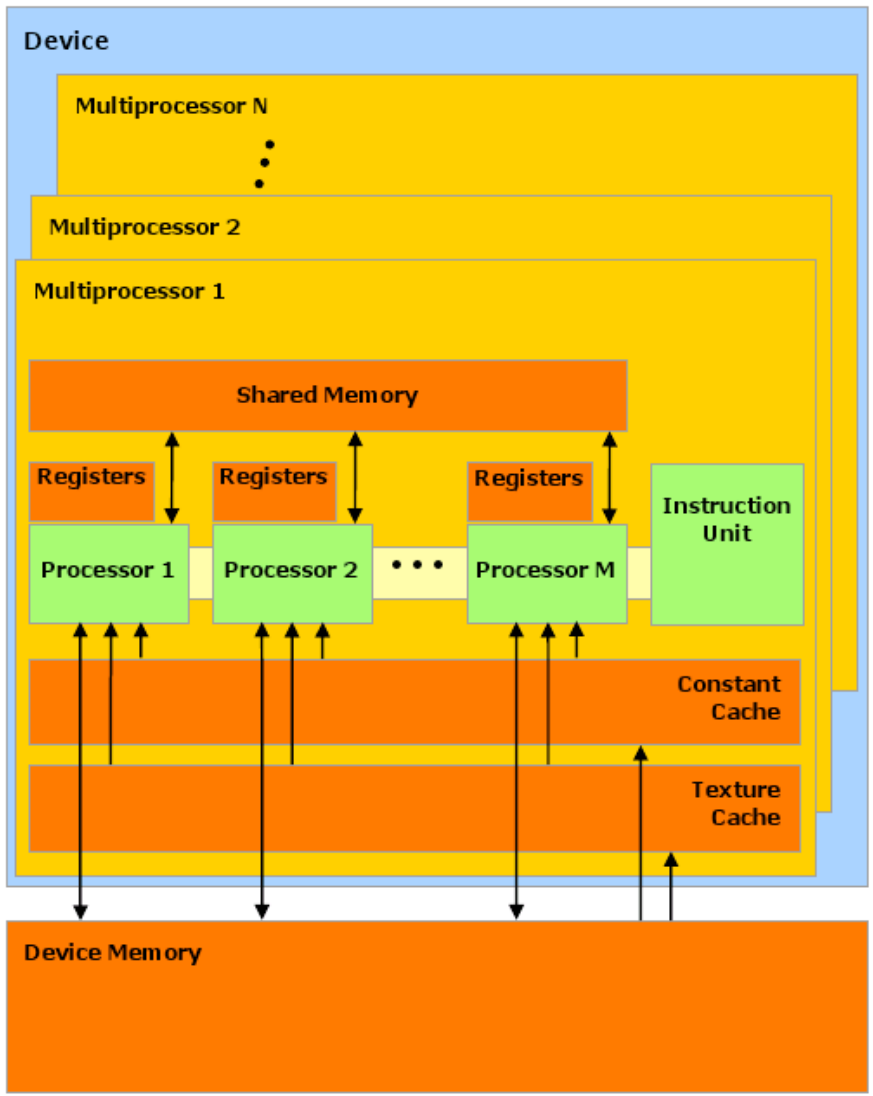
- ✓ Throughput is what matters
- ✓ Hide latency through parallelism
- ✓ Push hierarchy onto programmer



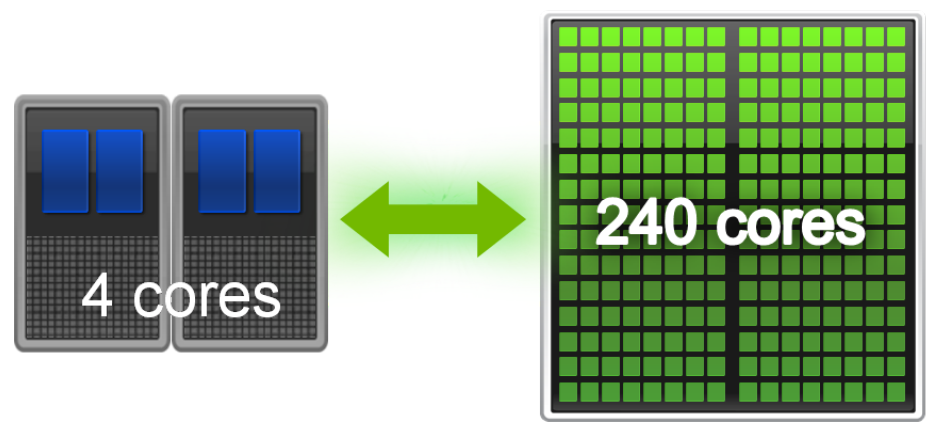
Much simpler logic with a focus on performance

GPUs 101

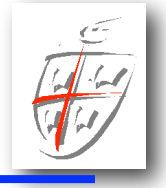
GPU layout



- ✓1 GPU = 30 MPs
- ✓1 MP has 1 IU, 8 SP, 1 DP
- ✓1 MP has 16KiB shared and 32 KiB Register memory
- ✓240 (512) threads
- ✓Dedicated RAM at 140GB/s
- ✓Limited caches



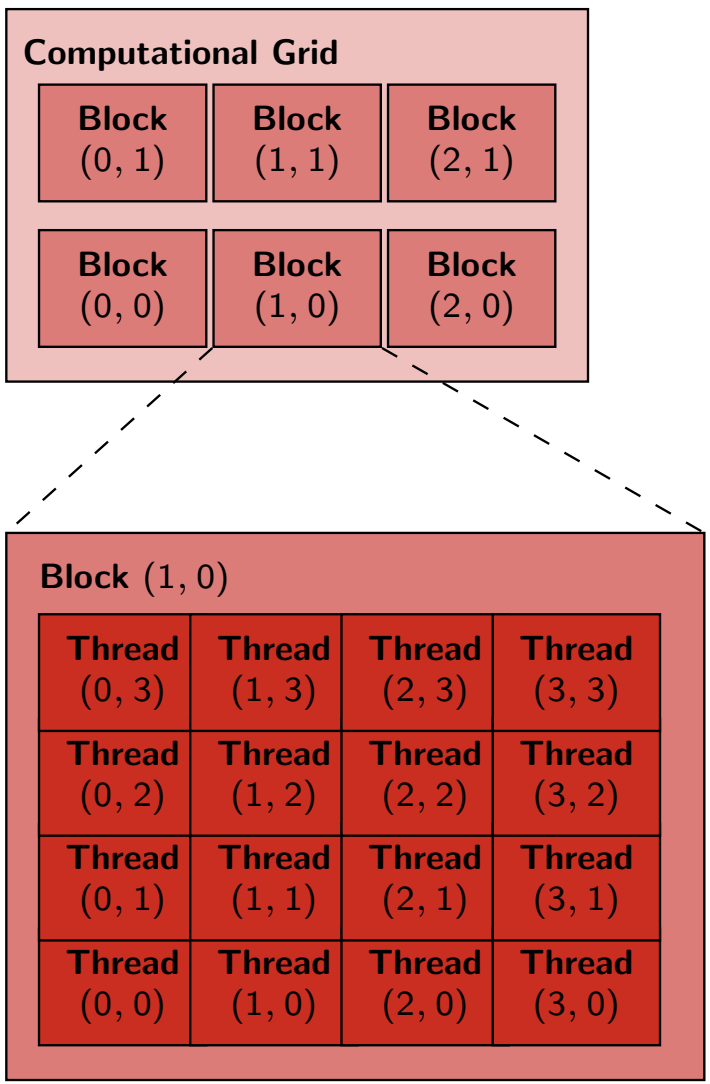
GPUs 101



Gains	Losses
⊕ Memory Bandwidth (140 GB/s vs. 12 GB/s)	⊖ Recursion
⊕ Compute Bandwidth (Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)	⊖ Function pointers
	⊖ Exceptions
	⊖ IEEE 754 FP compliance
	⊖ Cheap branches (i.e. `ifs`)

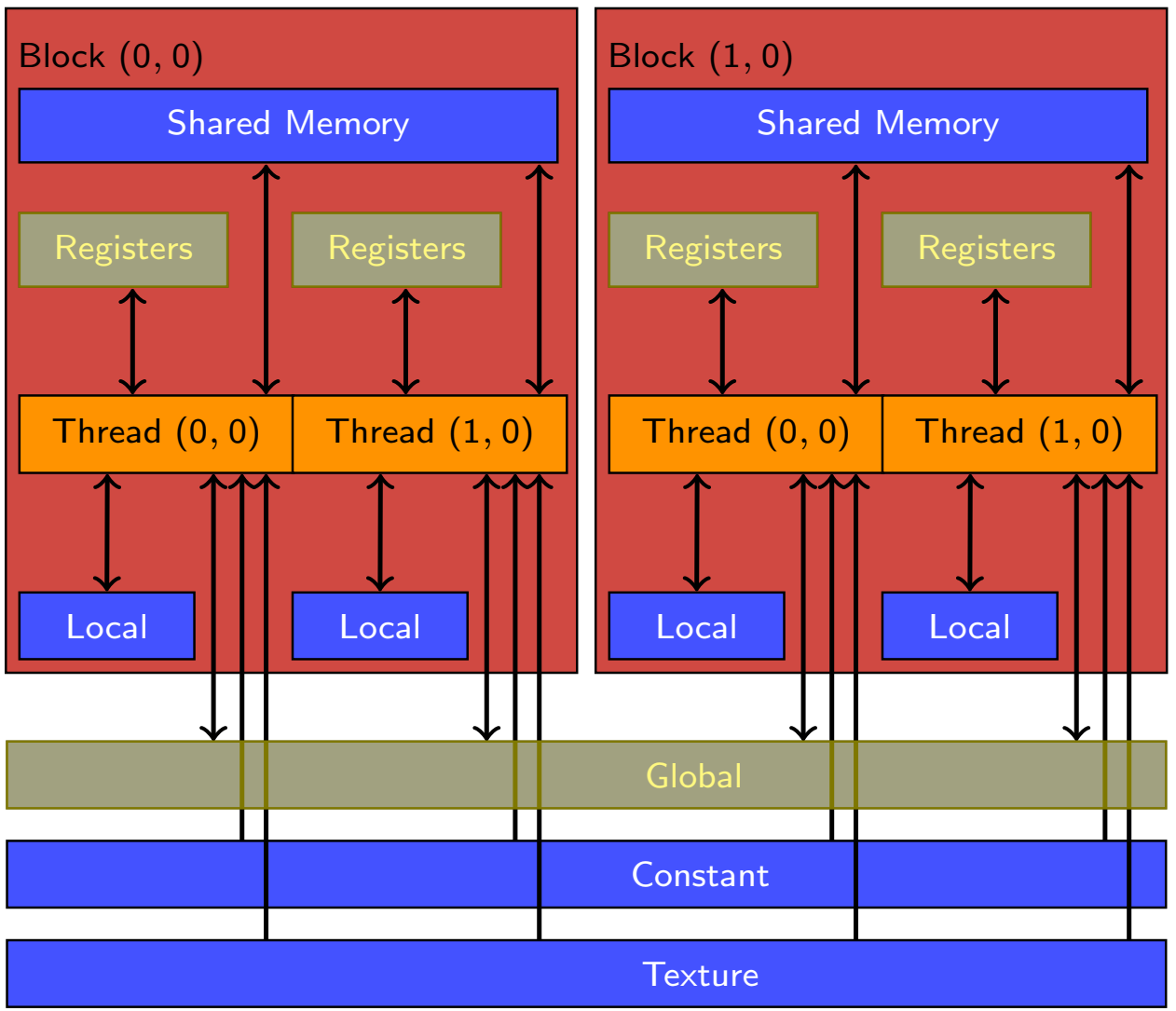
Already here it is clear that programming models/codes may have to undergo substantial changes -- and that not all will work well

GPUs 101



- ✓ Genuine multi-tiered parallelism
  - ✓ Grids
  - ✓ blocks
  - ✓ threads
- ✓ Only threads within a block can talk
  - ✓ Blocks must be executed in order
- ✓ Grids/blocks/threads replace loops
- ✓ Until recently, only single precision
- ✓ Code-able with CUDA (C-extension)

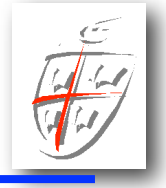
GPUs 101



Memory model:

- ✓ Registers
- ✓ Local shared
- ✓ Global

GPUs 101

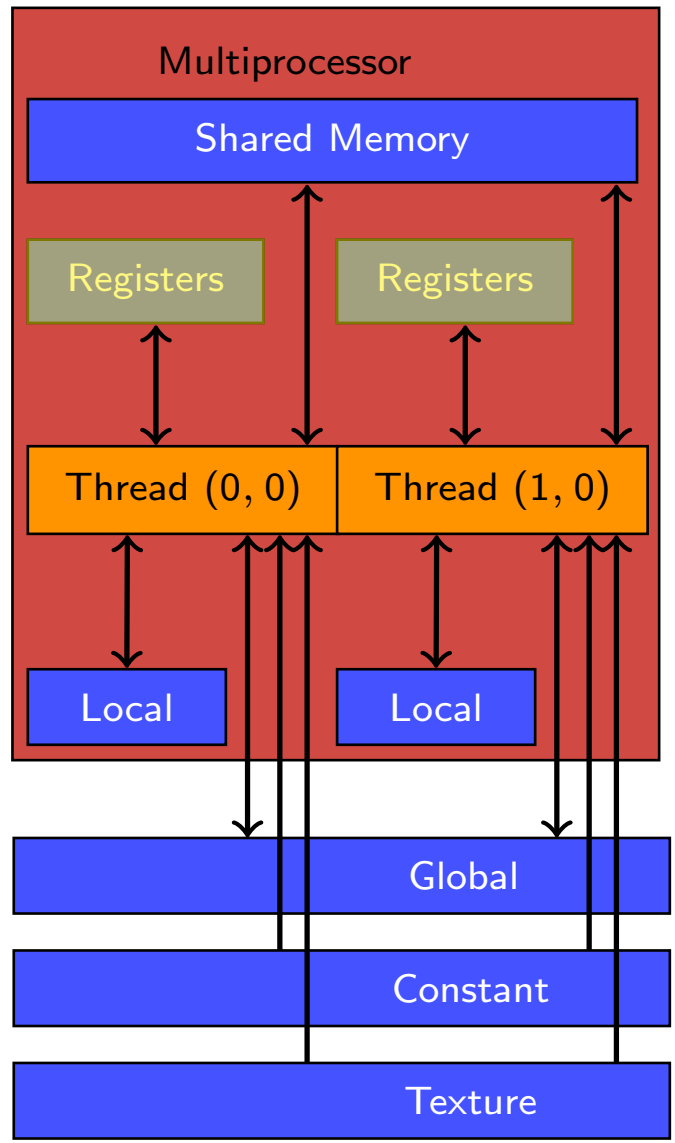


✓ Lots of multi-processors (about 30)

... communicate through global mem

✓ Registers, shared memory, and threads communicate with low latency

... but memory is limited (16-32 KiB)



GPUs 101

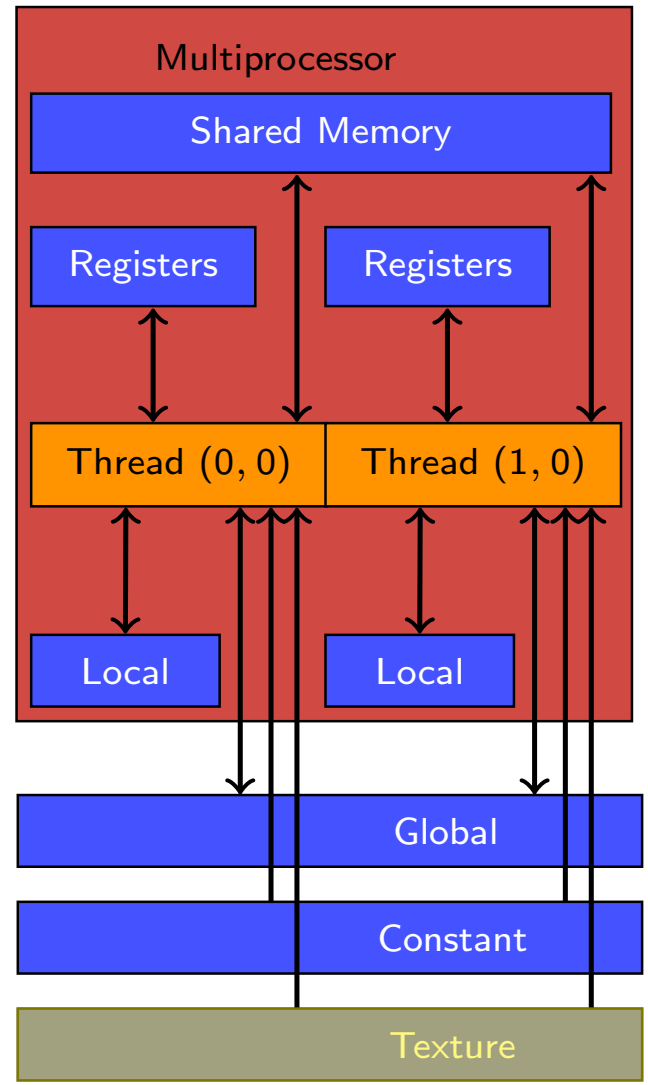
✓ Global memory (4GiB/PU) is plentiful

... but latency is high (512 bit bus)
... and stride one is preferred

✓ Texture is similar to global memory

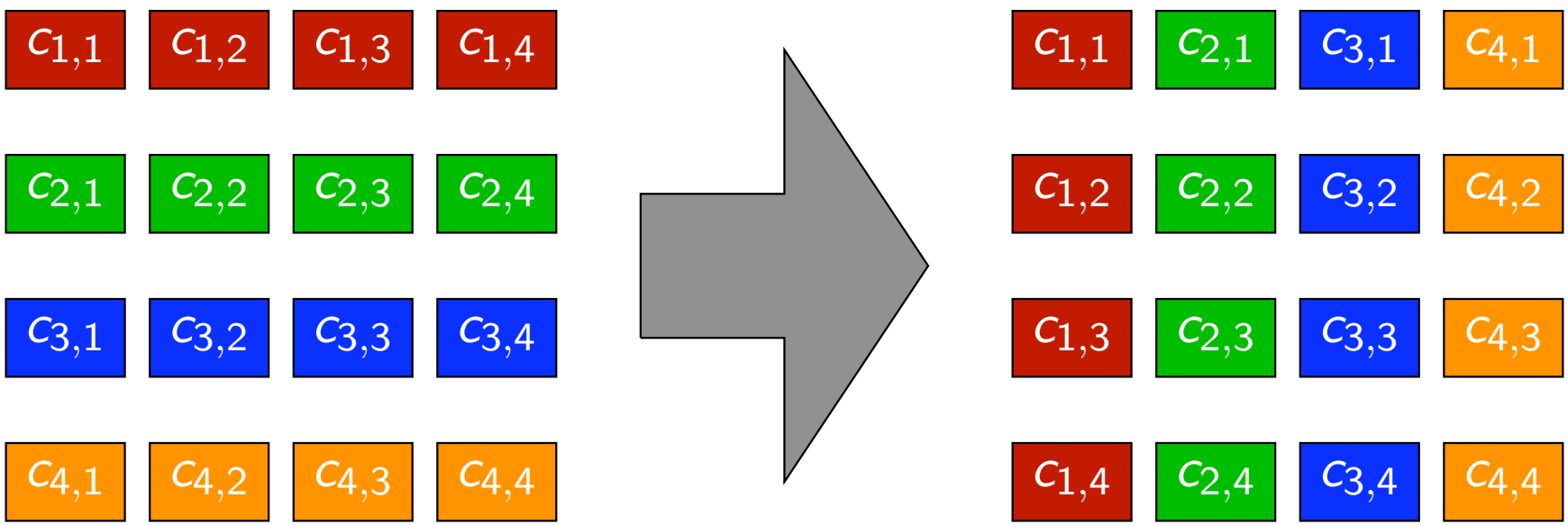
... allows more general access patterns
... but it is read only

Type	Per	Access	Latency
Registers	thread	R/W	1
Local	thread	R/W	1000
Shared	block	R/W	2
Global	grid	R/W	1000
Constant	grid	R/O	1-1000
Texture	grid	R/O	1000



Let's consider an example

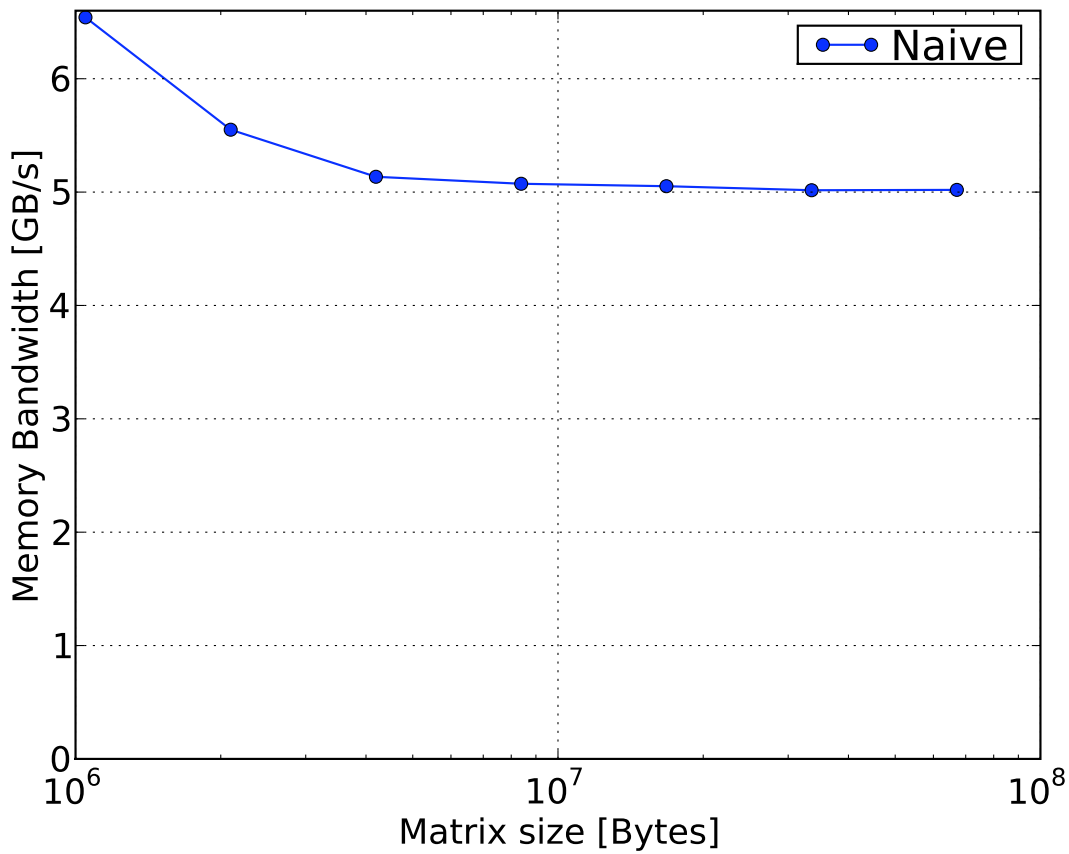
Matrix transpose



Memory bandwidth will be a limit here

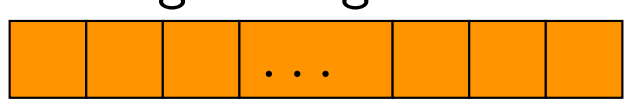
Let's consider an example

Using just global memory



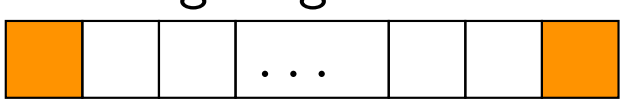
As CPU

Reading from global mem:



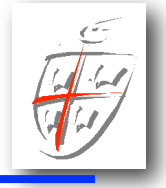
stride: 1 → one mem.trans.

Writing to global mem:

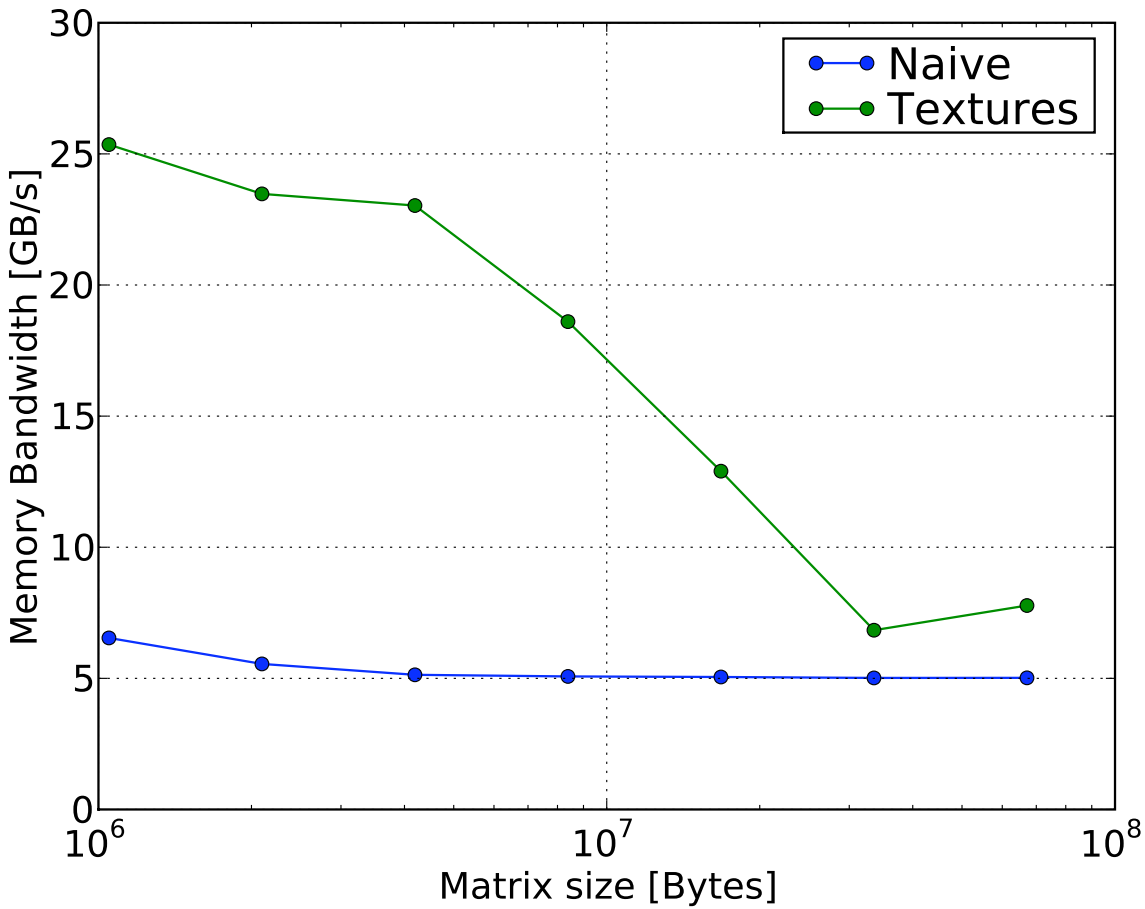


stride: 16 → **16 mem.trans.!**

Let's consider an example



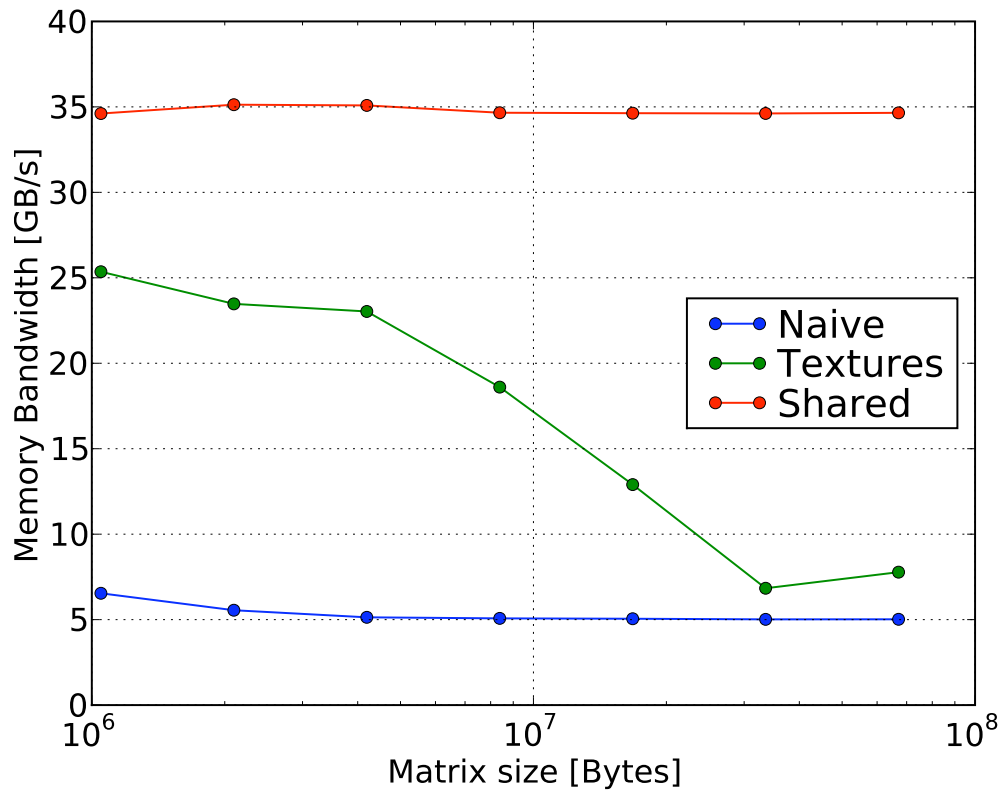
Using just texture(read)+global(write) memory



Getting better

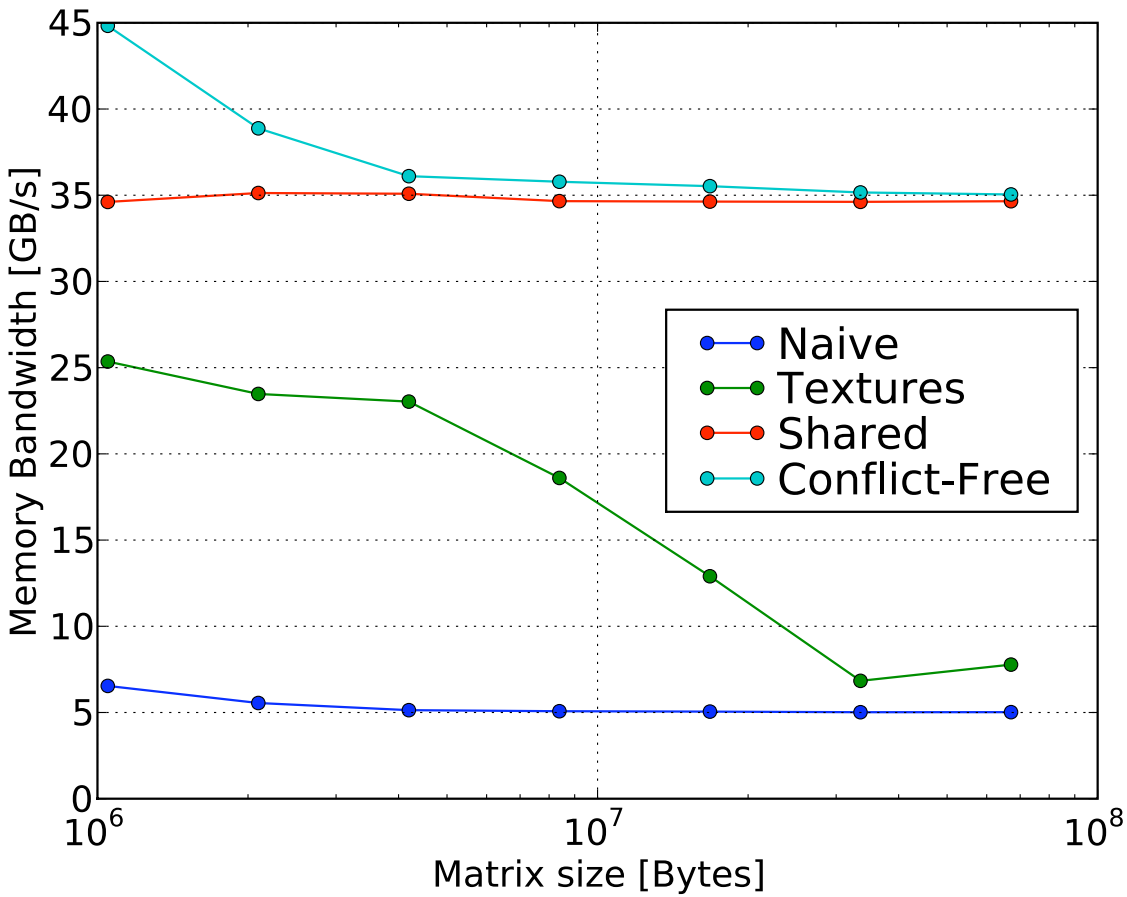
Let's consider an example

Transpose block-by-block in shared memory - this does not care about strides



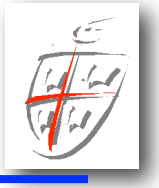
Let's consider an example

Additional improvements are possible for small matrices - bank conflicts in shared memory



A factor of 7-8 over CPU

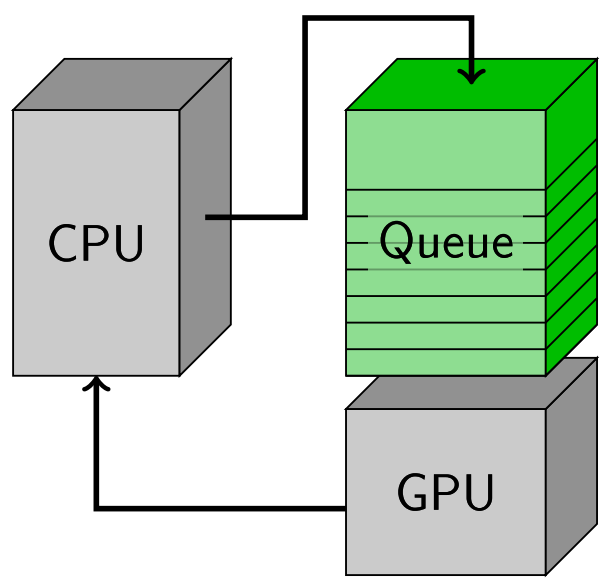
CPUs vs GPUs



As T. Warburton said in the his talk:

The CPU is mainly the traffic controller
... although it need not be

- ✓ The CPU and GPU runs asynchronously
- ✓ CPU submits to GPU queue
- ✓ CPU synchronizes GPUs
- ✓ Explicitly controlled concurrency is possible

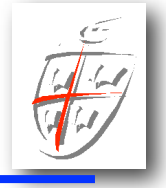


GPUs overview

- ✓ GPUs exploit multi-layer concurrency
- ✓ The memory hierarchy is deep
- ✓ Memory padding is often needed to get optimal performance
- ✓ Several types of memory must be used for performance

- ✓ First factor of 5 is not too hard to get
- ✓ Next factor of 5 requires quite some work
- ✓ Additional factor of 2-3 requires serious work

Nodal DG on GPU's

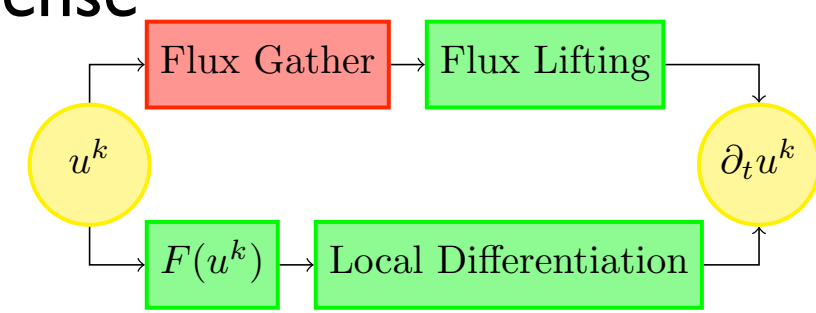


So what does all this mean ?

✓ GPU's has deep memory hierarchies so local is good
- ➡ The majority of DG operations are local

✓ Compute bandwidth >> memory bandwidth
- ➡ High-order DG is arithmetically intense

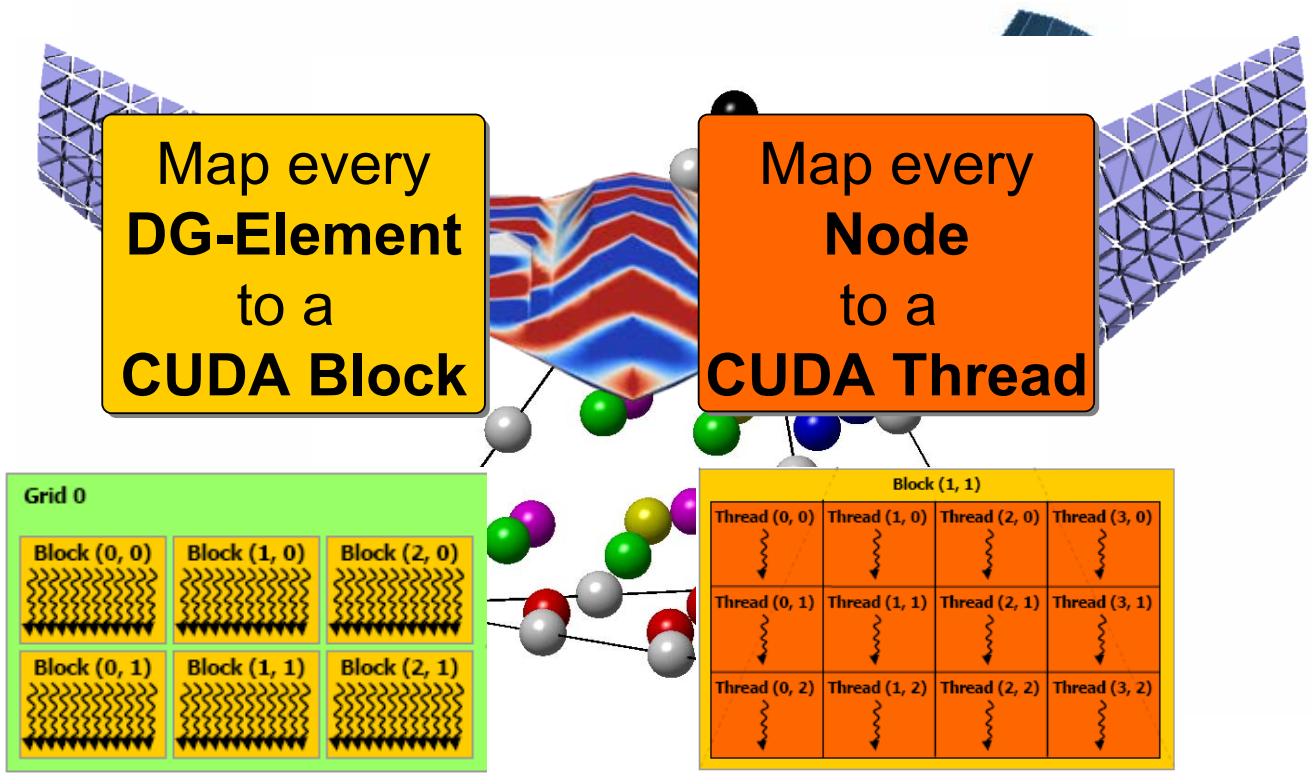
✓ GPU global memory favors dense data
- ➡ Local DG operators are all dense



With proper care we should be able to obtain excellent performance for DG-FEM on GPU's

Nodal DG on GPU's

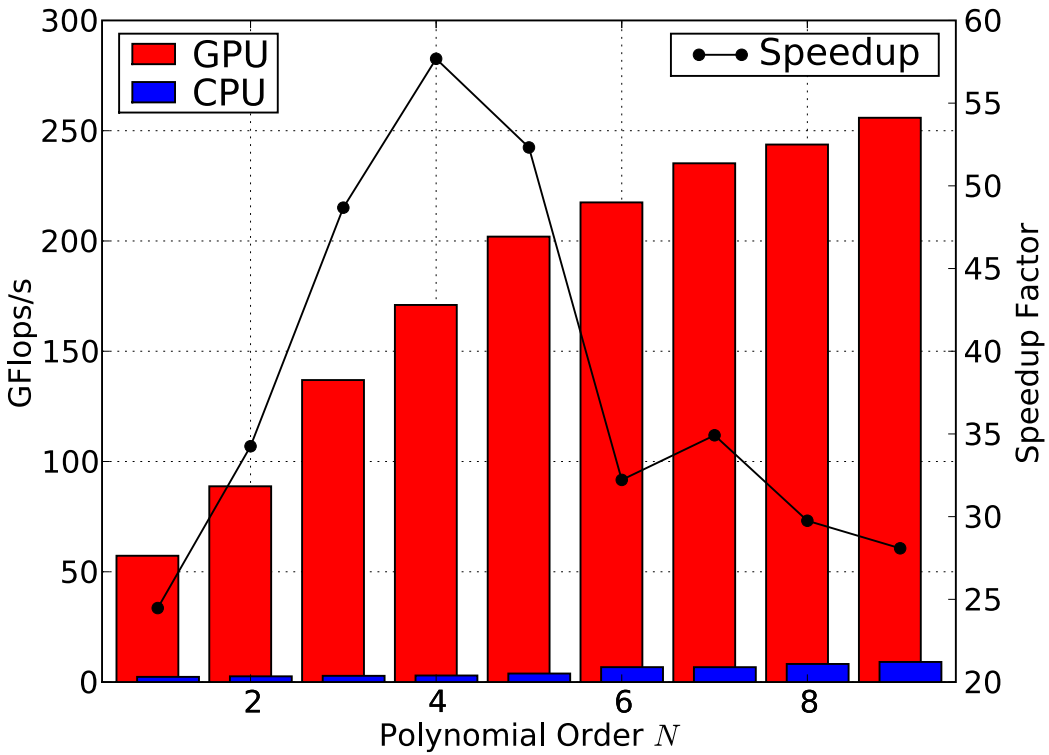
Nodes in threads, elements for blocks



Other choices:

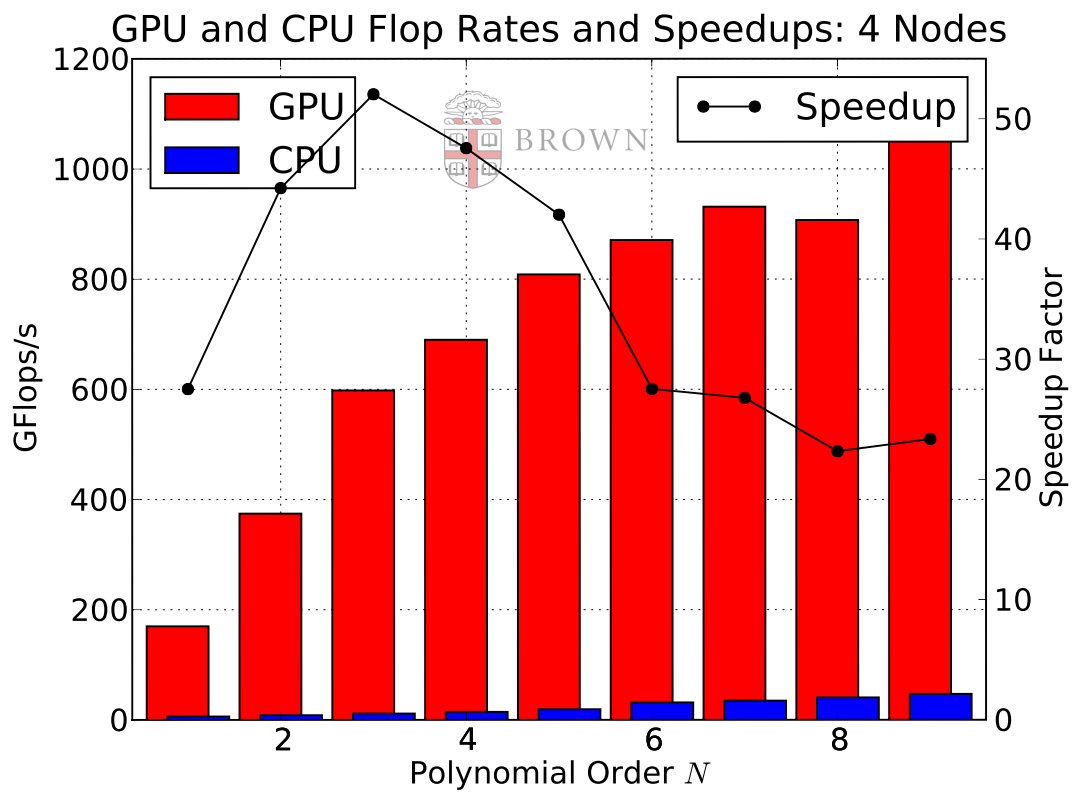
- ✓D-matrix in shared, data in global (small N)
- ✓Data in shared, D-matrix is global (large N)

Nodal DG on GPU's



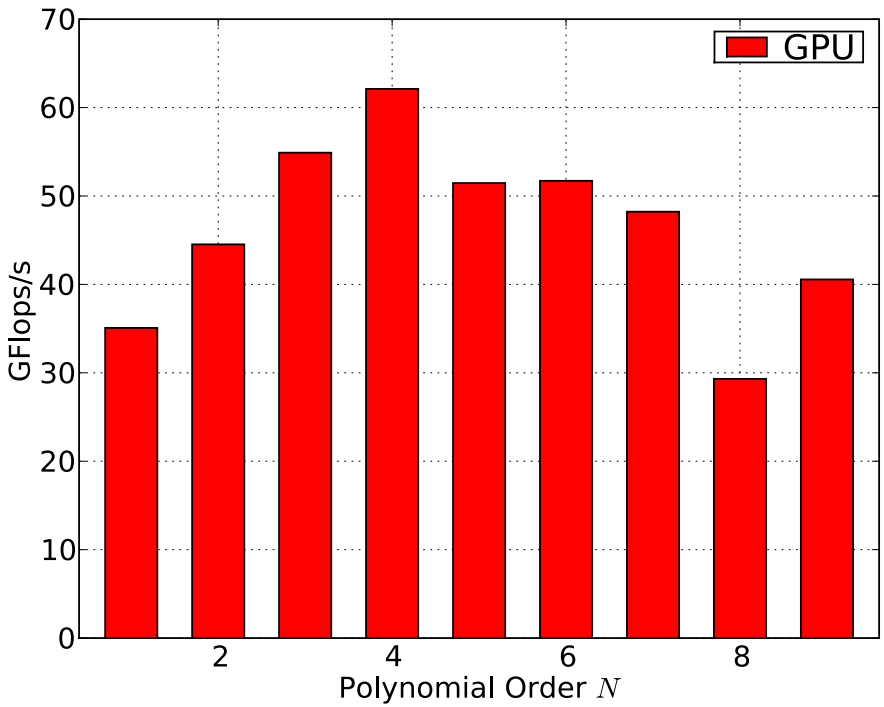
DG-FEM on one GPU

DG-FEM on four GPU one card

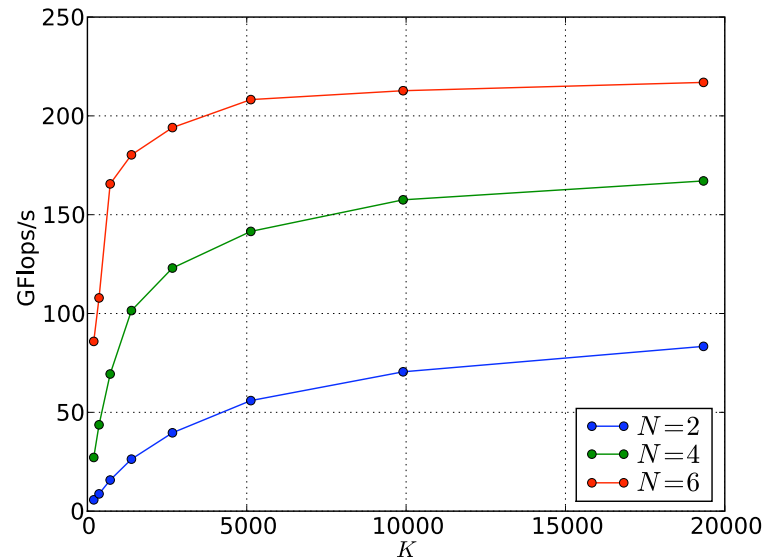


Nodal DG on GPU's

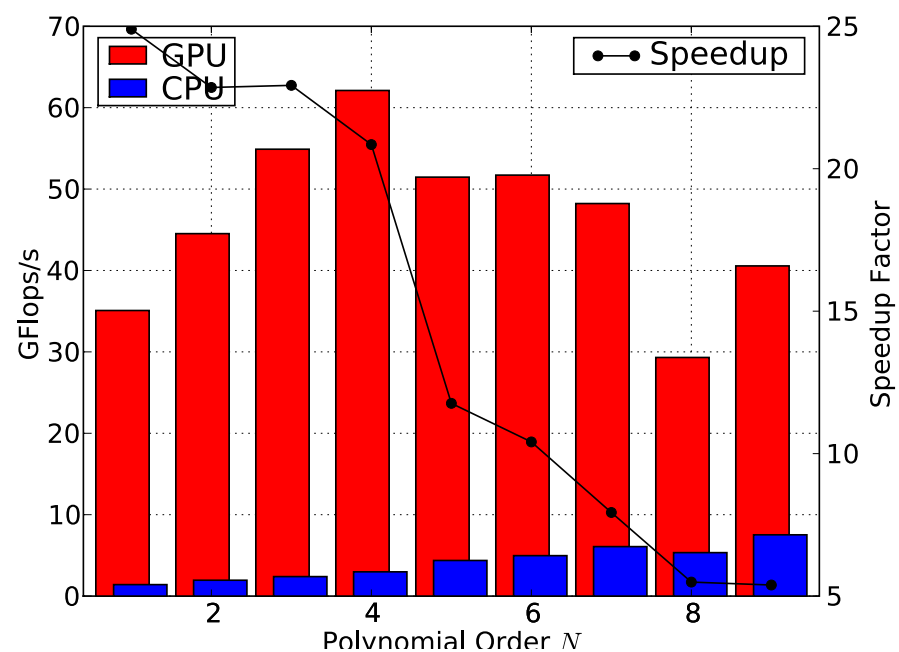
Where you need it most



... and for larger and larger grids

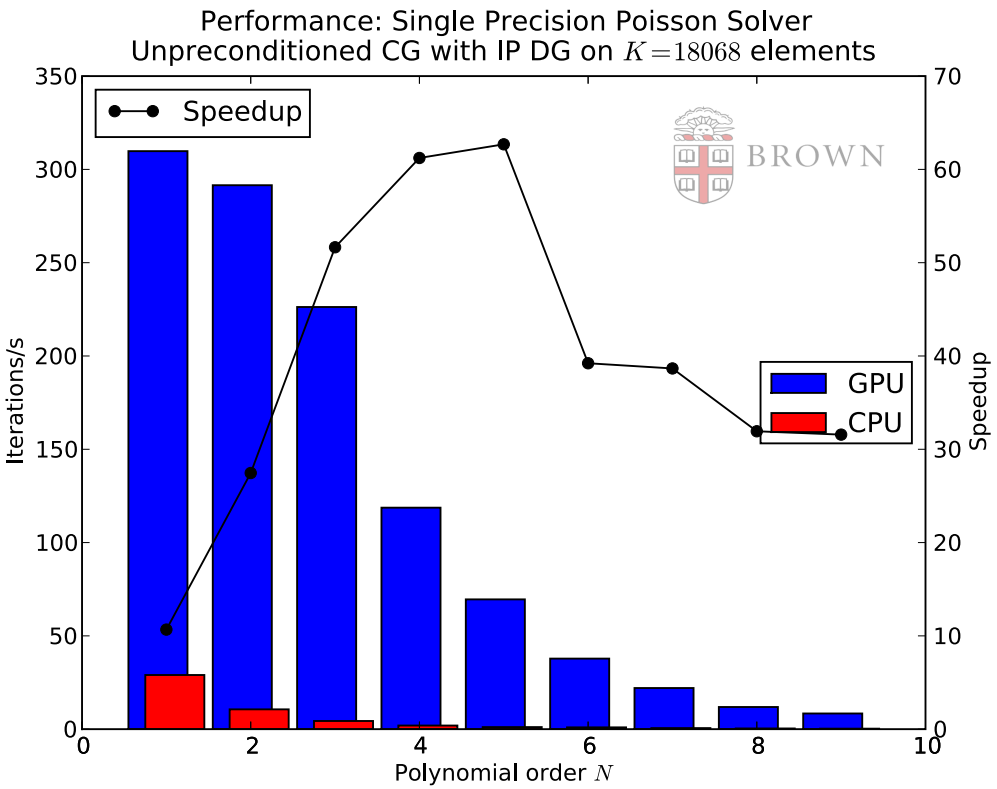


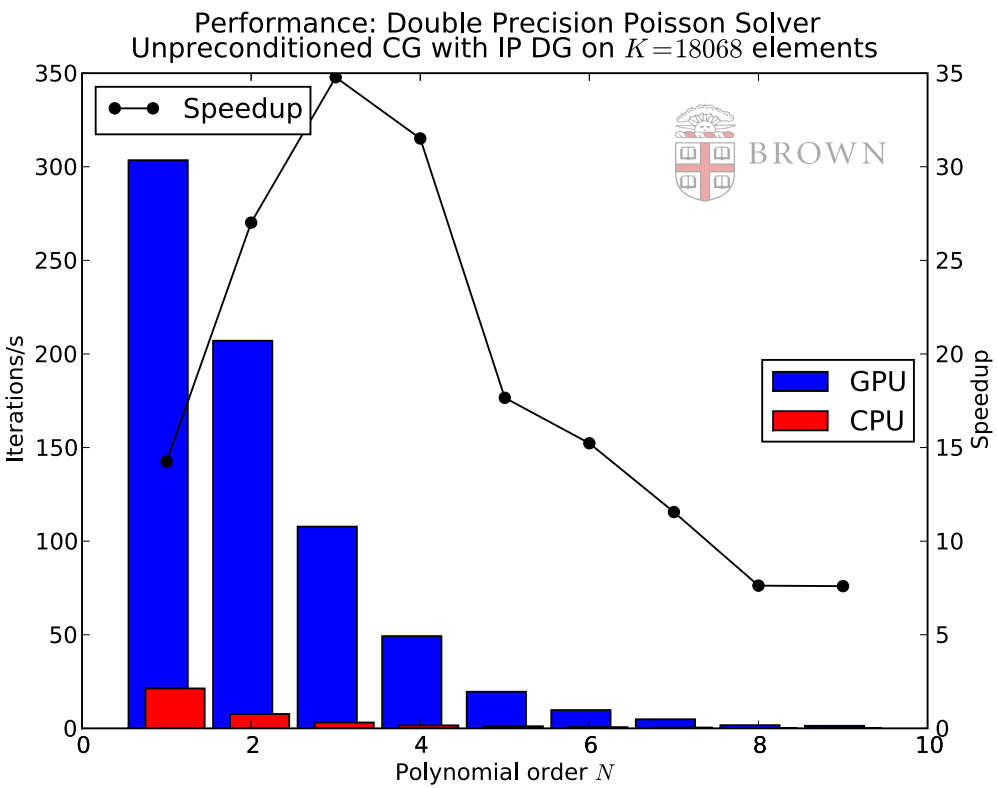
Also in double precision



Nodal DG on GPU's

Similar results for DG-FEM Poisson solver with CG

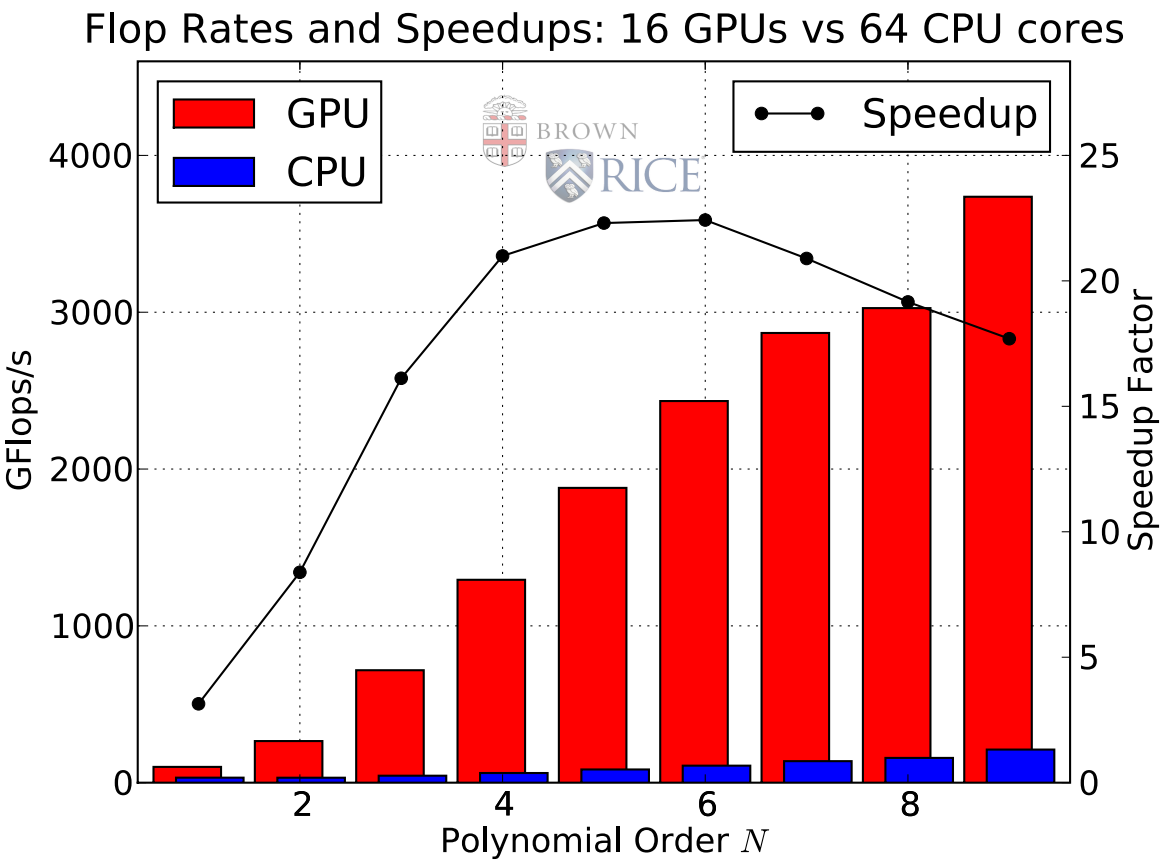


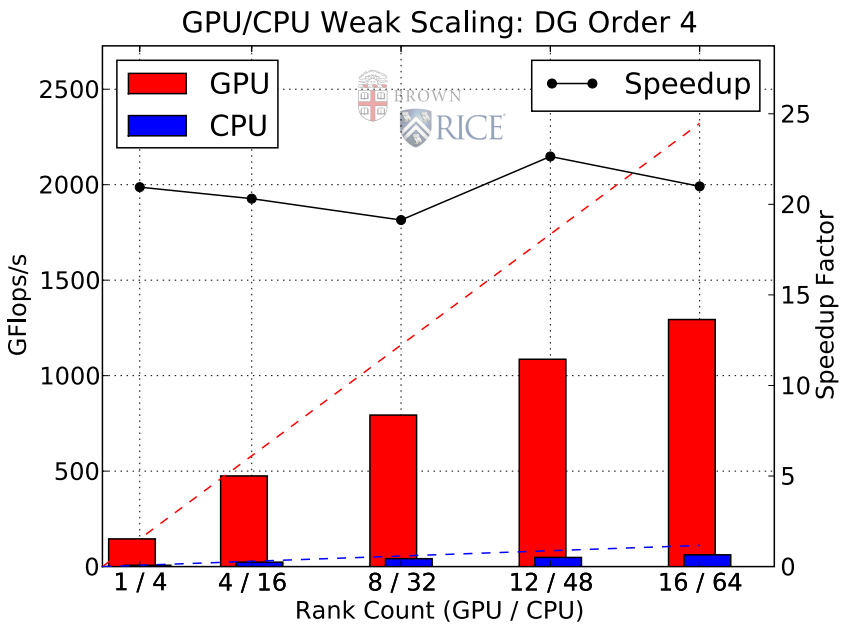


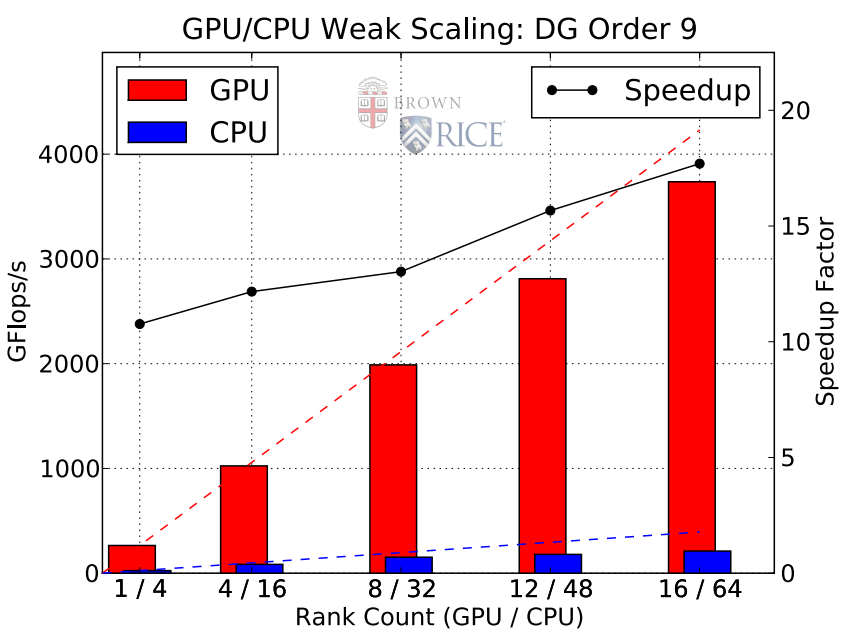
Note: No preconditioning

Combined GPU/MPI solution

MPI across network

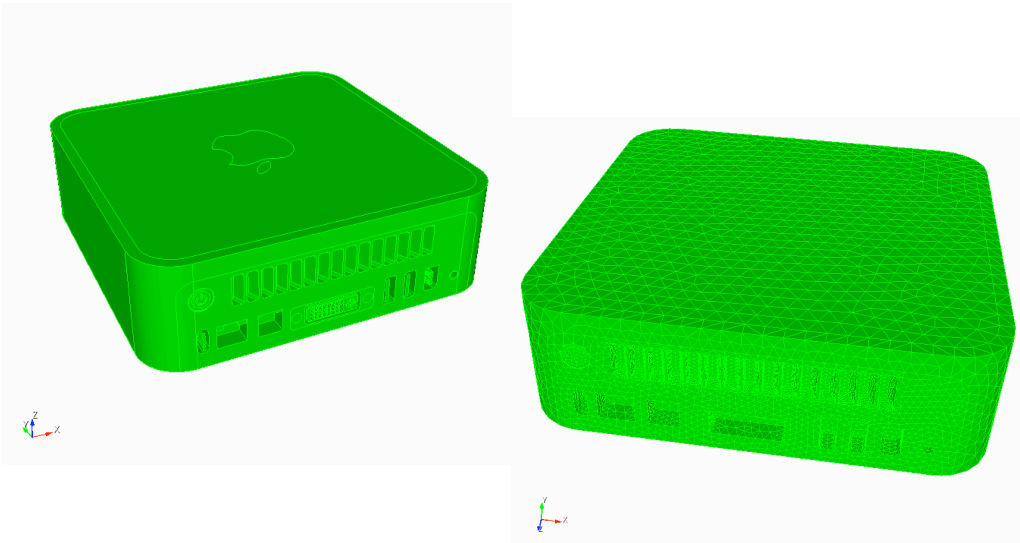


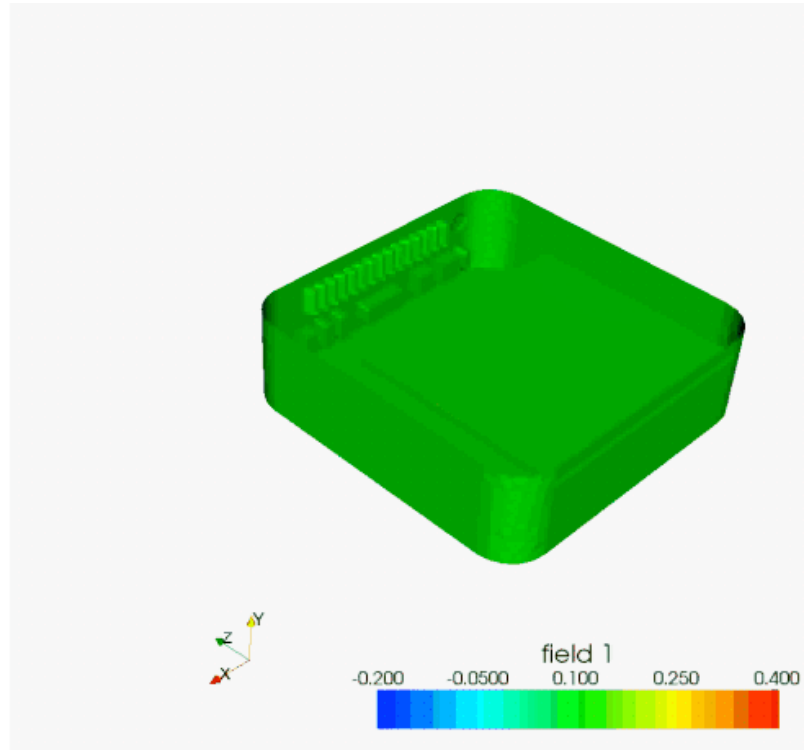


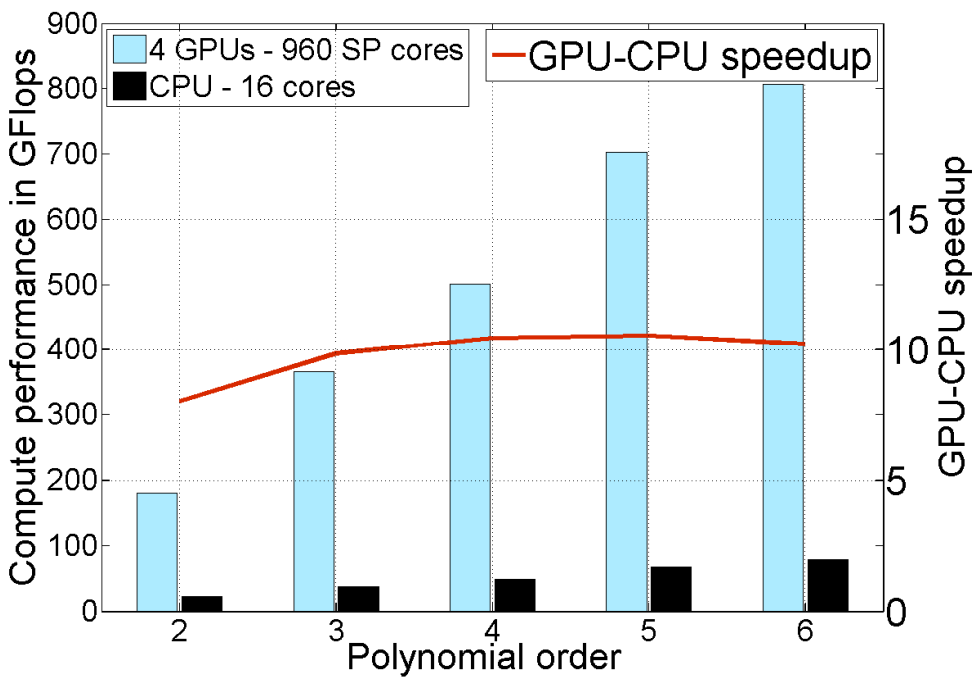


Good scaling when problem is large

Example - a Mac Mini



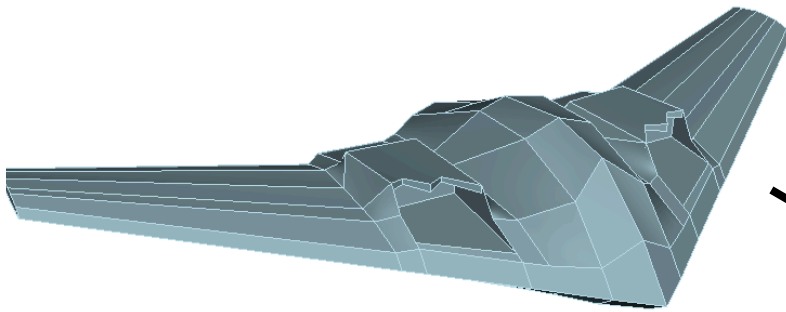


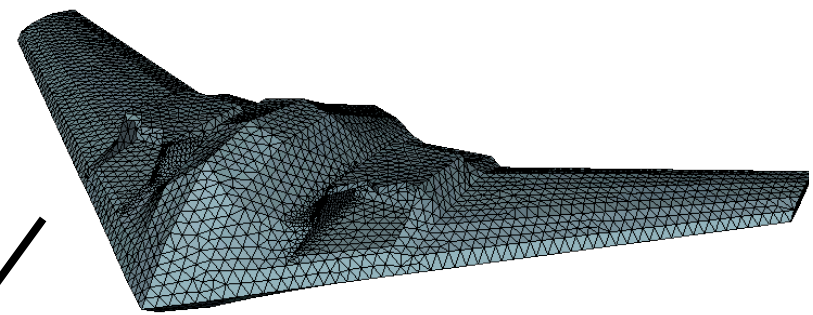


K=201765 elements
3rd order elements

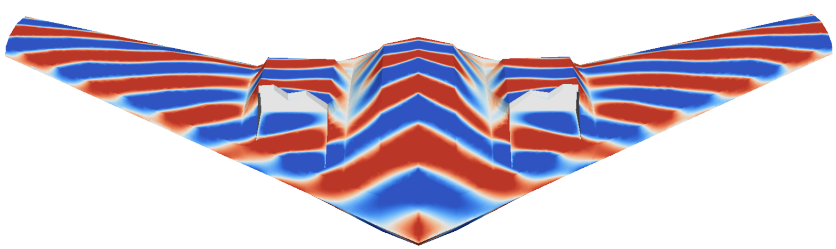
Computation by N. Godel

Example: Military aircraft

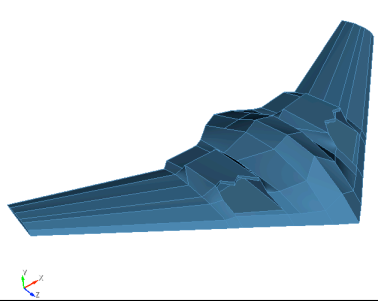




K=130413 elements
3rd order elements
15.6E6 DoF



Computation by N. Godel

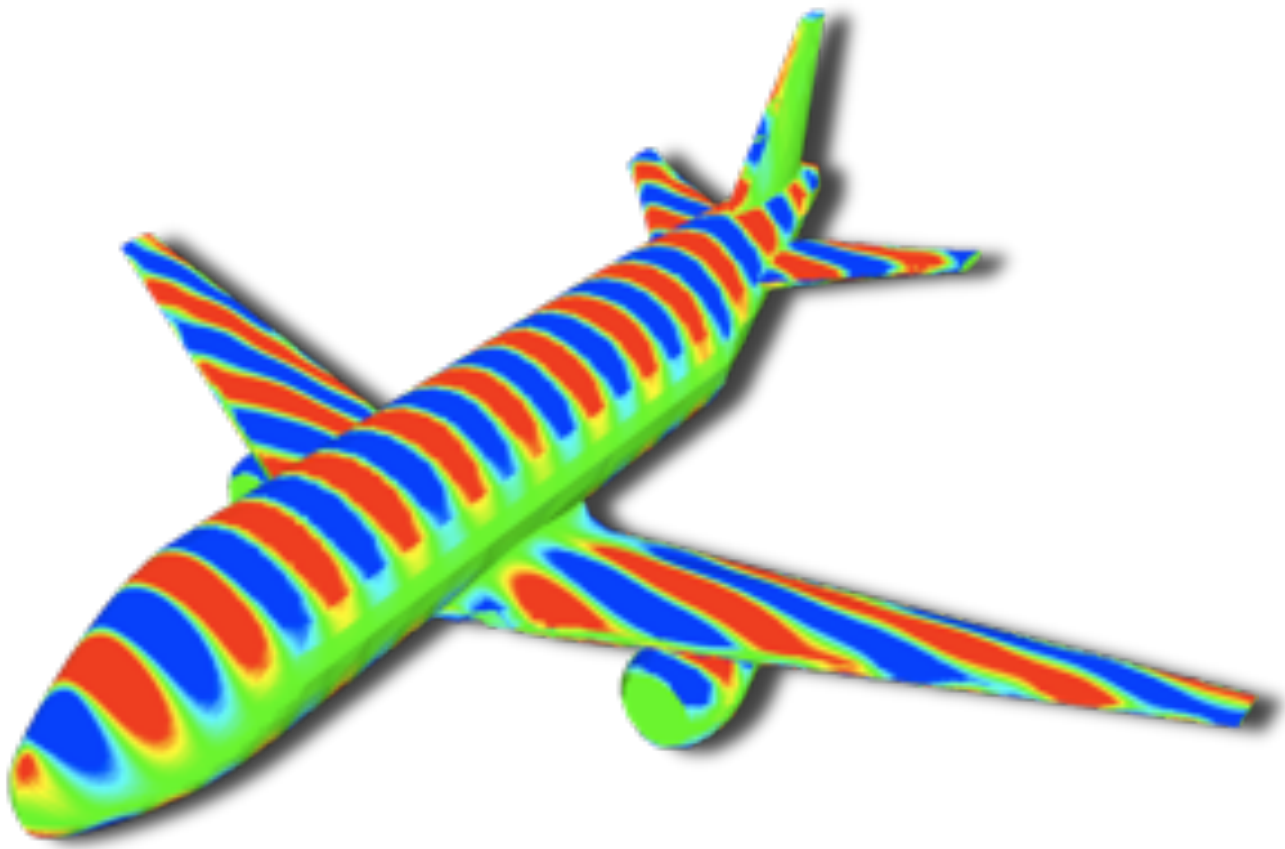
	CPU global	29 h 6 min 46 s	1.0
	GPU global	39 min 1 s	44.8
	GPU multirate	11 min 50 s	147.6

Nodal DG on GPU's

Not just for toy problems

228K elements
5th order elements
78m DOF
68k time-steps

Time ~ 6 hours

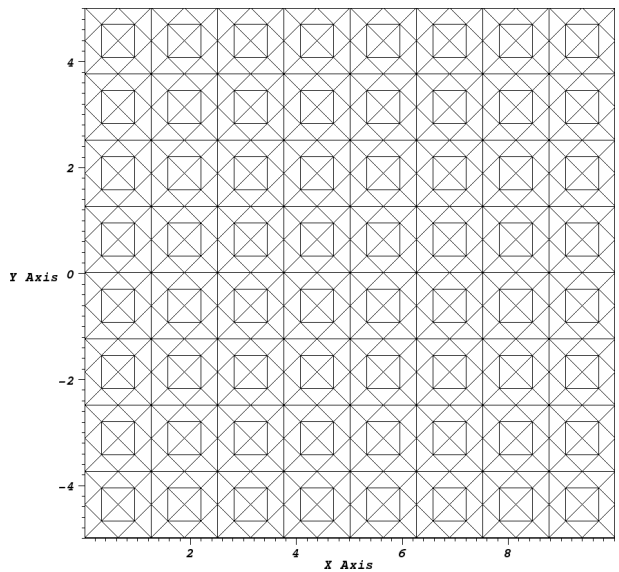


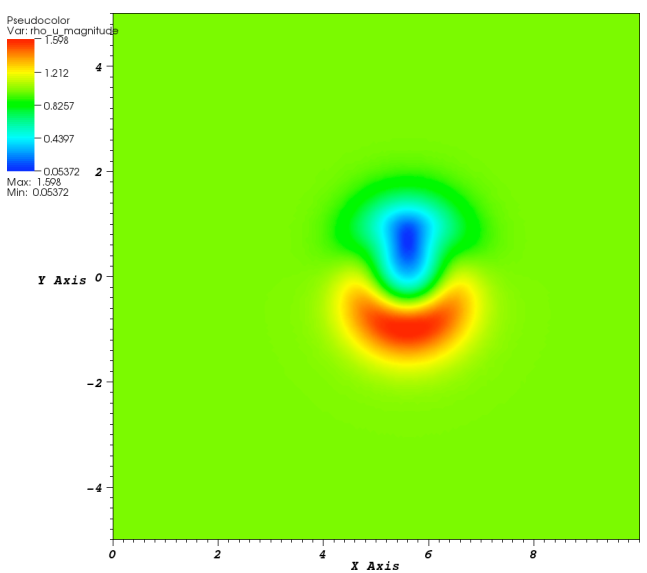
711.9 GFlop/s on one card

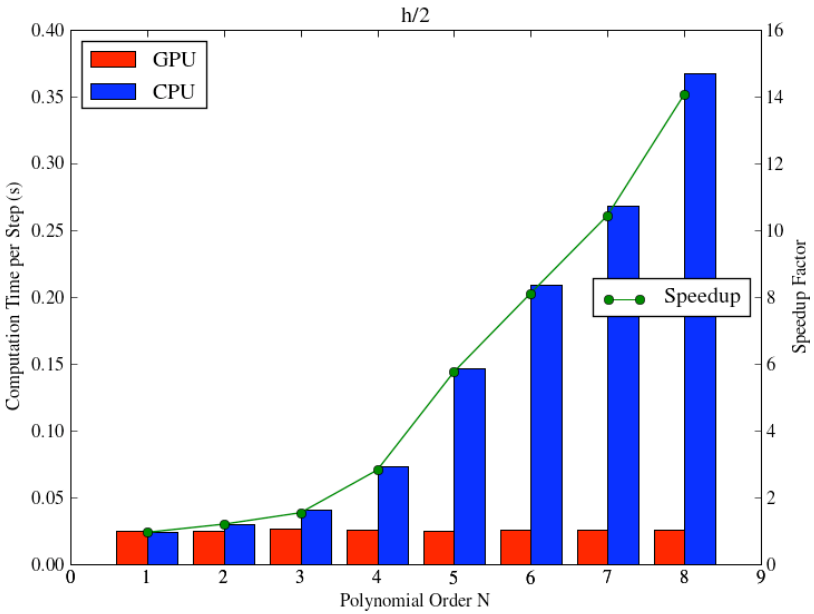
Computation by N. Godel

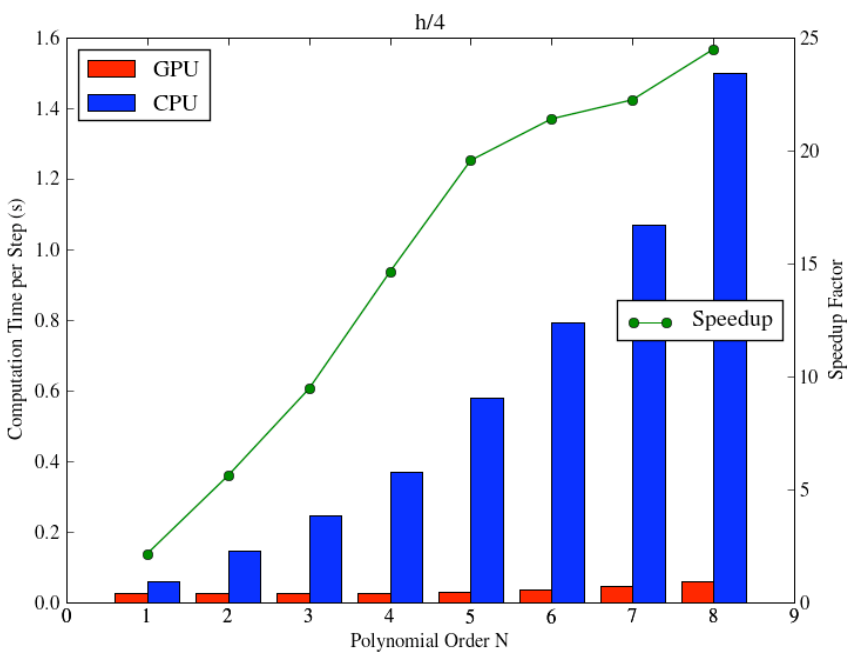
Beyond Maxwell's equations

2D Euler test case



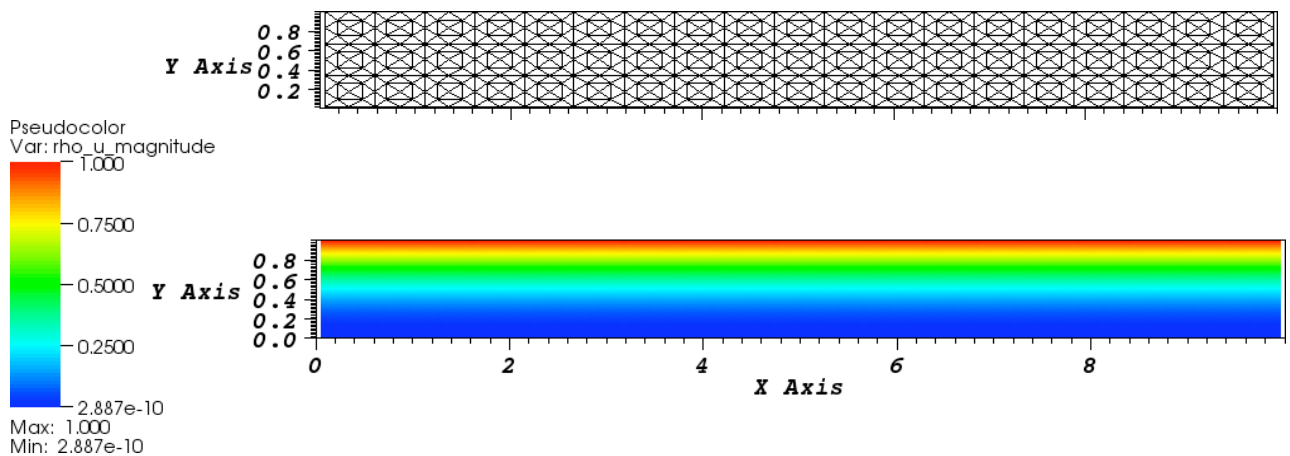


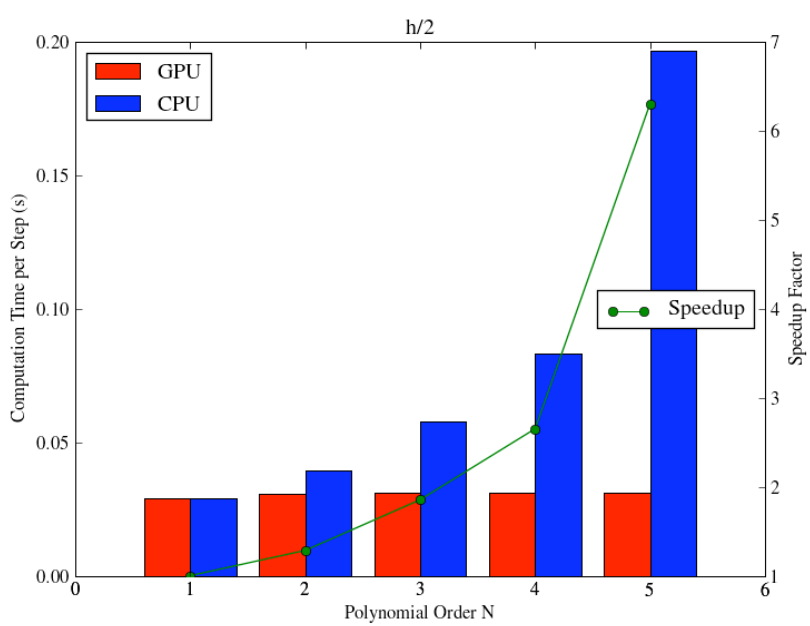


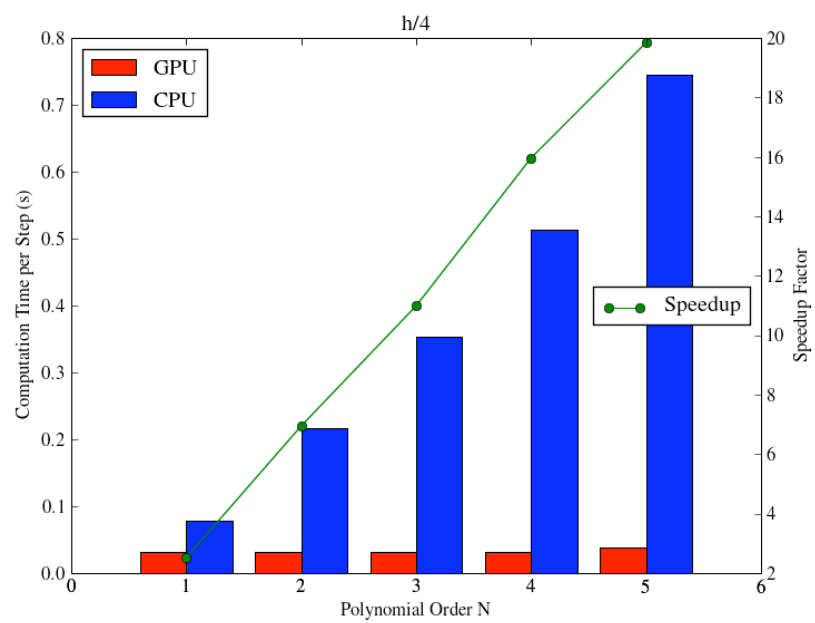


Beyond Maxwell's equations

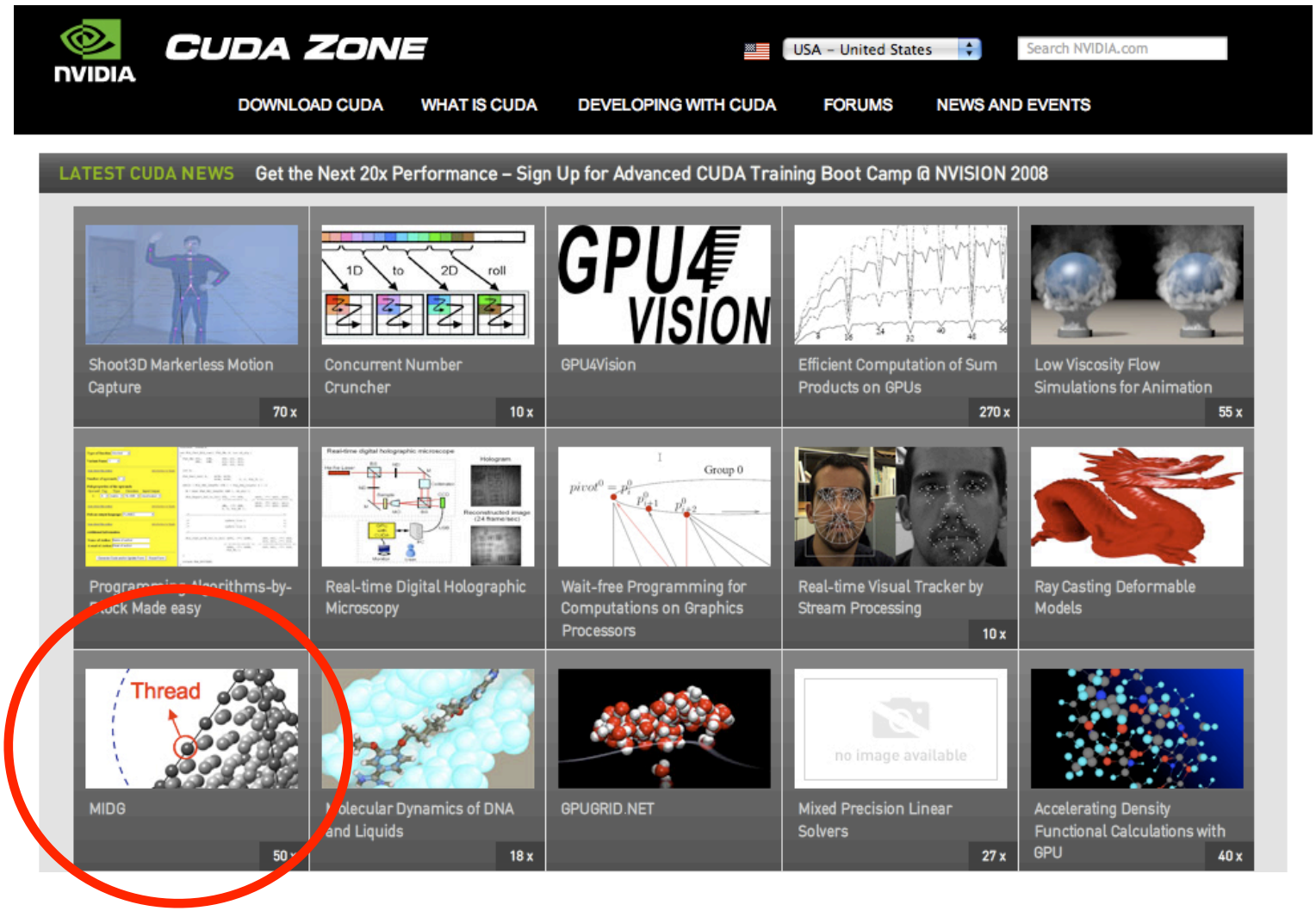
2D Navier-Stokes test case







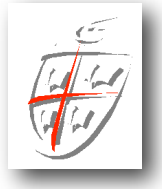
Want to play yourself ?



Code MIDG available at http://nvidia.com/cuda

Note: Single precision - double to come

Nodal DG on GPU's



Several GPU cards can be coupled over MPI at minimal overhead (demonstrated). Lets do the numbers

One 700GFlop/s/4GB mem card costs ~$8k

So $250k will buy you 16-18TFlop/s sustained

This is the entry into Top500 Supercomputer list !

... at 5%-10% of a CPU based machine

This is **a game changer** -- and the local nature of DG-FEM makes it very well suited to take advantage of this

Why are these things important

The obvious reason is that we get speed for very little cost.

However, it is also important because the next big machine (Exa-scale) is likely a

1 GHz, Kilo-Core, Mega-Node machine

.. so you may as well understand now how to do this and which methods map well to many-core machines

Do we have to write it all ?

No :-)

✓ Book related codes - all at www.nudg.org

✓ Matlab codes

✓ NUDG++ - a C++ version of 2D/3D codes (serial)

✓ hedge - a Python based meta-programming code. Support for serial/parallel/GPU

✓ MIDG - a bare bones parallel/GPU code for Maxwell's equations

Do we have to write it all ?

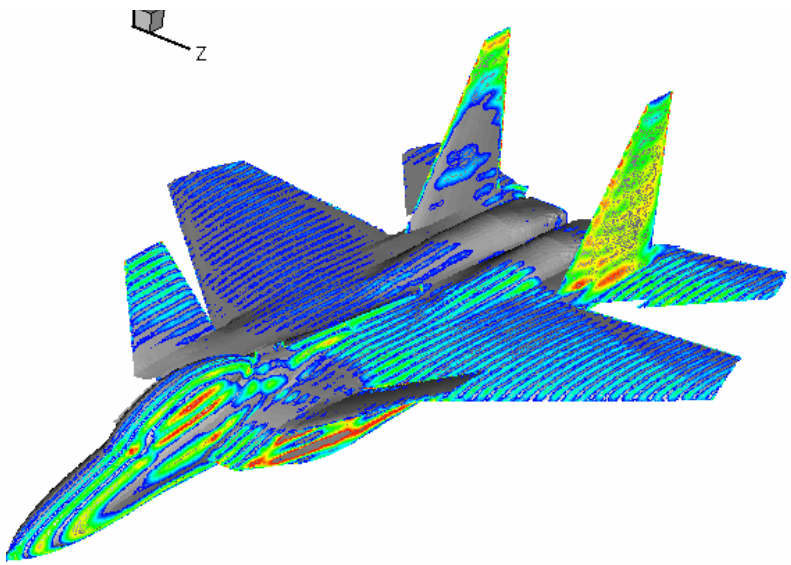
Other codes

✓**Slegde++** - C++ operator code. Interfaced with parallel solvers (Trilinos and Mumps) and support for adaptivity and non-conformity. Contact Lucas Wilcox (UT Austin/ICES)

✓**deal.II** - a large code with support for fully non-conforming DG with adaptivity etc. Only for squares/cubes. www.dealii.org

✓**Nektar++** - a C++ code for both spectral elements/hp and DG. Mainly for CFD. Contact Prof Spencer Sherwin (Imperial College, London)

Progress ?



Year 2001

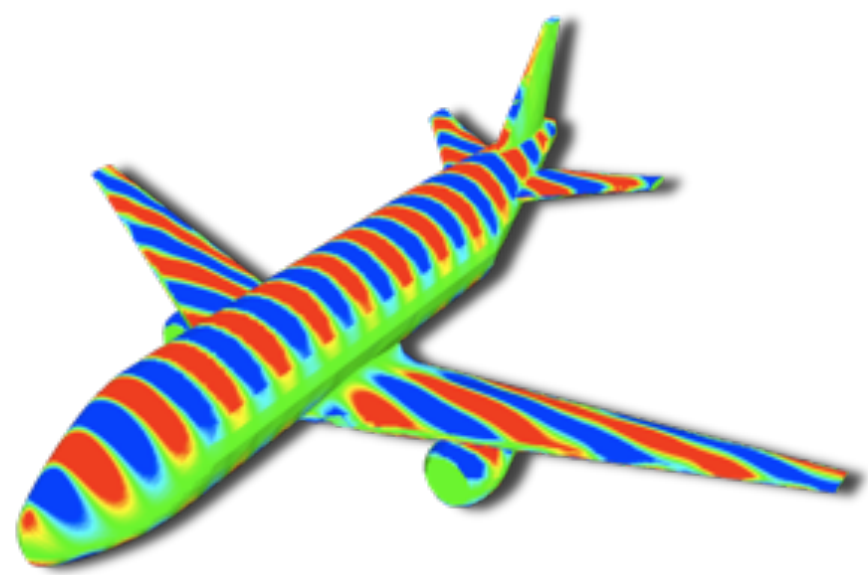
250k tets, 4th order
50m dof, 100k timesteps

24 hours on 512 procs

Year 2008

82k tets, 4th order
17m dof, 60k timesteps

Few hours on GPU



Thanks !

Many people have contributed to this with material, figures, examples etc

- ✓ Tim Warburton (Rice University)
- ✓ Lucas Wilcox (UT Austin)
- ✓ Andreas Kloeckner (Brown)
- ✓ Nico Goedel (Hamburg)
- ✓ Jeronimo Rodriguez (USC)
- ✓ Hendrick Riedmann (Stuttgart)
- ✓ Francis Giraldo (NRL, Monterrey)
- ✓ HyperComp Inc (LA, CA)
- ✓ ... and many more

Thanks !

Thank you

to you for hanging in there.

Happy computing !