Large Scale FEA on the GPU

Krishnan Suresh

Associate Professor
Mechanical Engineering

High-Performance Trick

- Computations (i.e., 3.4*1.22): essentially free
- Memory access determines speed of code

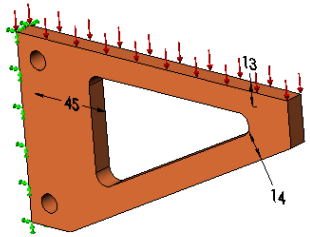
Pick memory-efficient algorithm!!

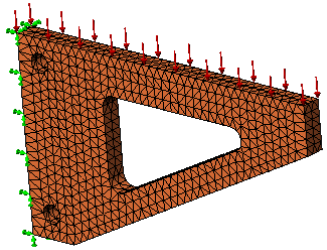
Finite Element Analysis

THE UNIVERSITY of WISCONSIN MADISON

- **Simulation of engineering phenomena**
  - **Structural, thermal, fluid, …**
  - **Static, modal, transient**
  - **Linear, non-linear**

Structural Static FEA





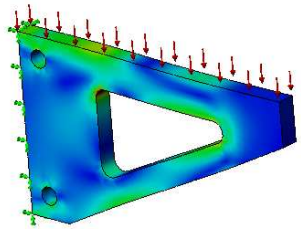
K_e

f_e

$$K = \sum K_e$$

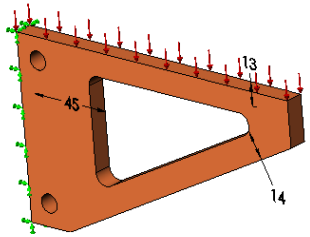
$$f = \sum f_e$$

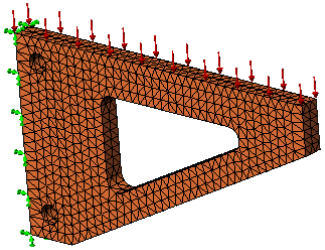
$$Ku = f$$



Model → Discretize → Element Stiffness → Assemble/ Solve → Post-process

Typical Bottleneck





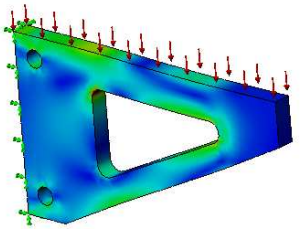
K_e

f_e

$$K = \sum K_e$$

$$f = \sum f_e$$

$$Ku = f$$



Model → Discretize → Element Stiffness → Assemble/ Solve → Post-process

… GPU emphasis

Linear Solvers

$$Ku = f$$

K is sparse & usually symmetric P.D

Direct

$$K = LDL^T$$

$$u = L^{-1}D^{-1}L^{-T}f$$

Direct Sparse on GPU

THE UNIVERSITY of WISCONSIN MADISON

$$Ku = f$$

Accelerating the ANSYS Direct Sparse Solver with GPUs

Géraud P. Krawezik
Acceleware Corp.
Calgary, AB, Canada

(2008)

Gene Poole
ANSYS Inc.
Canonsburg, PA, USA

Direct Sparse on GPU

$$Ku = f$$

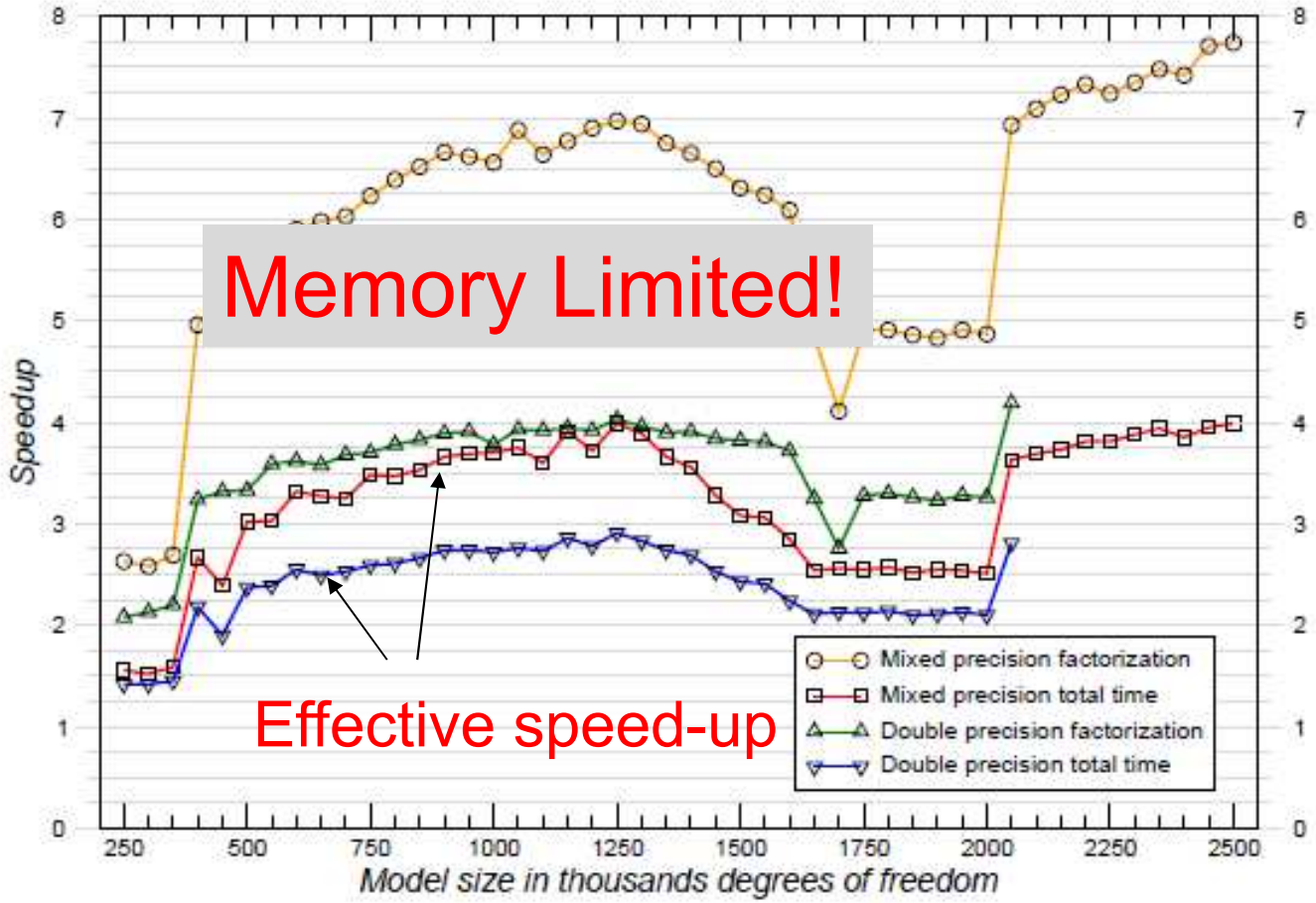


Fig. 6. Speedup versus two CPU cores

Linear Solvers

$$Ku = f$$

Direct

$$K = LDL^T$$

$$u = L^{-1}D^{-1}L^{-T}f$$

Iterative

$$u^{i+1} = u^i + B(f - Ku^i)$$

B : Preconditioner of K

- Repeated Matrix-Vector ops
- Ideally suited for GPU!

Linear Solvers

$$Ku = f$$

Direct *Iterative*

$$u^{i+1} = u^i + B(f - Ku^i)$$

$$B : \text{Preconditioner of K}$$

Two Goals:

1. Minimize #iterations … B dependent
2. Minimize cost of K*u (SpMv)

THE UNIVERSITY of WISCONSIN MADISON

Presentations at GTC 2012 Conference

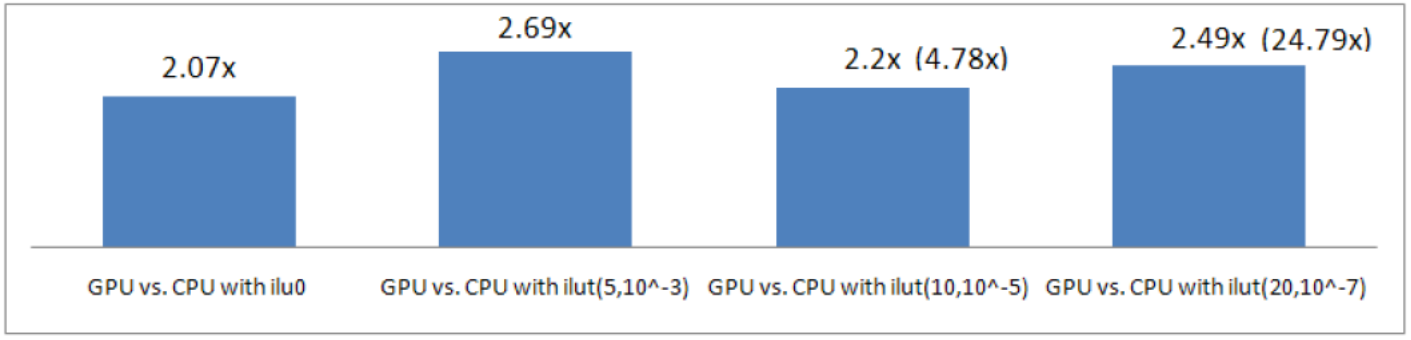
- Fine-grained Parallel Preconditioners
- CULA
- MAGMA
- Accelerating Iterative Linear Solvers
- Efficient AMG on Hybrid GPU Clusters
- Preconditioning for Large-Scale Linear Solvers
- …

THE UNIVERSITY of WISCONSIN MADISON

Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS

Maxim Naumov
NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050

June 21, 2011



CULA Sparse

THE UNIVERSITY of WISCONSIN MADISON

CULA sparse is a **GPU-accelerated library** for linear algebra that provides iterative solvers for sparse systems

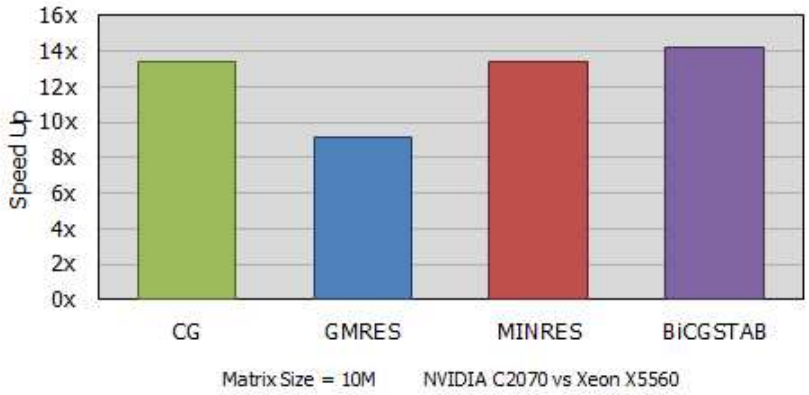
Offering a capability set that is unmatched in a ready-to-integrate library

With many solvers and preconditioners to choose from, you have the power to choose the solution that is best for you. Best of all, it is easy to use, with no complicated callback interfaces and a straightforward configuration.

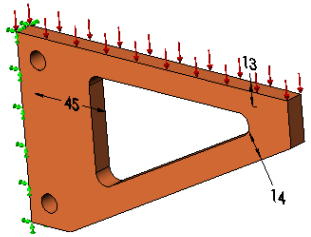
SOLVERS	PRECONDITIONERS	DATA FORMATS
Biconjugate Gradient (BICG)	Jacobi	Double-precision real and complex
Biconjugate Gradient Stabilized (BICGSTAB)	Block Jacobi	Compressed Sparse Row (CSR)
Conjugate Gradient (CG)	Incomplete LU (Ilu0)	Compressed Sparse Column (CSC)
Generalized Minimum Residual (GMRES)	Reordered (Ilu0)	Coordinate (COO)
Minimum Residual (MINRES)		

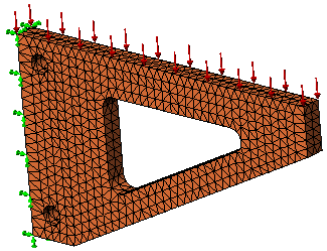
With performance that is 10x faster than competing solutions

We are experts at getting the very best performance available. We combine powerful GPU-acceleration with capable numerical algorithms to yield solutions that reach levels of performance not previously possible.



Ku = f in FEA context





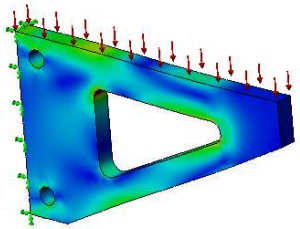
K_e

f_e

$$K = \sum K_e$$

$$f = \sum f_e$$

$$Ku = f$$



Model → Discretize → Element Stiffness → Assemble/ Solve → Post-process

Accelerate linear solve in FEA context!

Linear Solvers

$$Ku = f$$

Direct

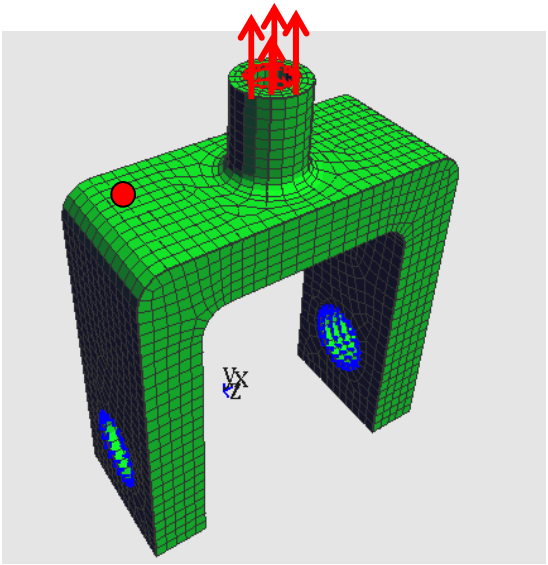
Iterative

$$u^{i+1} = u^i + P(f - Ku^i)$$

P : Preconditioner of K

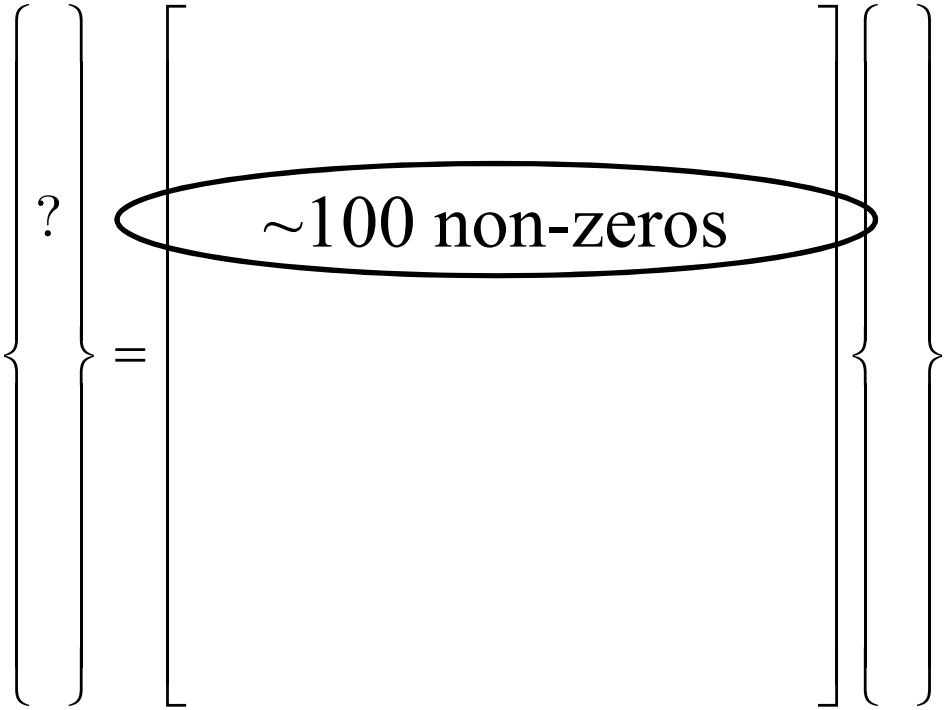
Assembled

*Assembled K*u*



GPU thread per node

1. Assemble K (CPU/GPU)
2. Push to GPU
3. Execute K*u in parallel



Coalesced memory access limits performance!

Linear Solvers

$$Ku = f$$

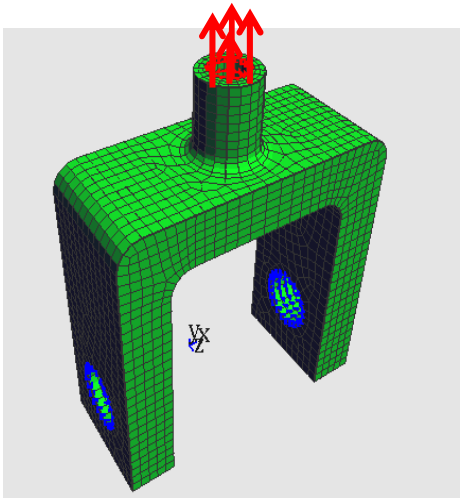
Direct — *Iterative*

$$u^{i+1} = u^i + P(f - Ku^i)$$

P : Preconditioner of K

Assembled — *Assembly-free*

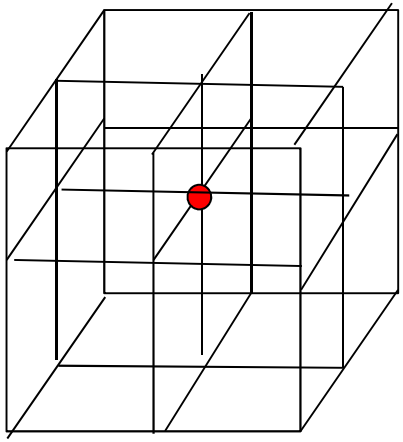
Matrix-Free



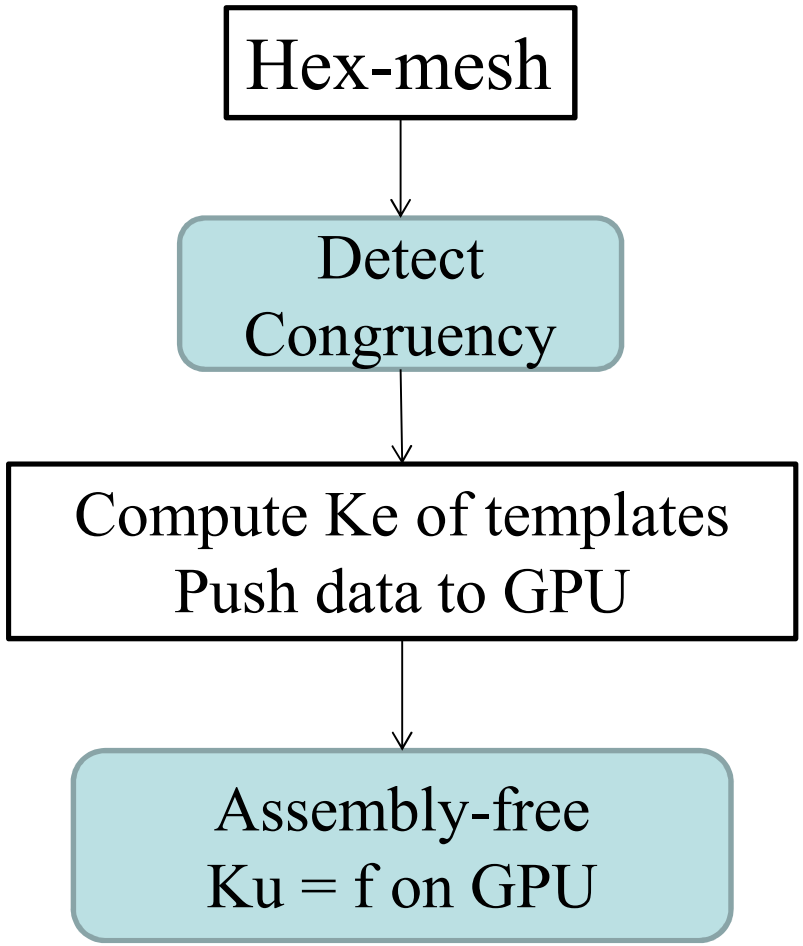
$$K = \left(\sum_e K_e \right)$$

$$Ku = \left(\sum_e K_e \right) u = \sum_e \left(K_e u_e \right)$$

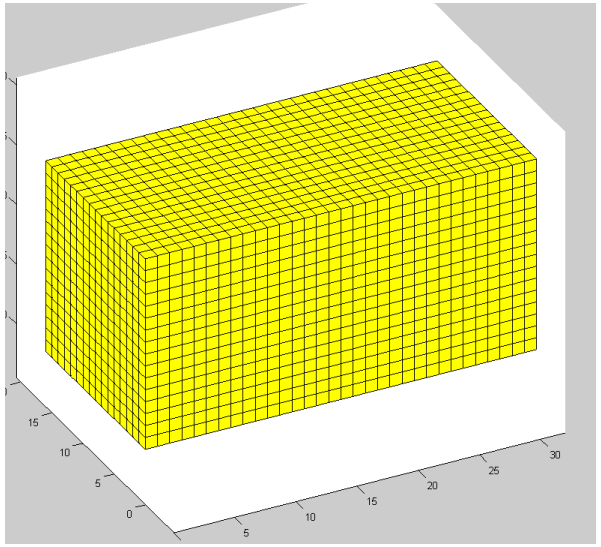
$$? = \left[\quad \right] \left\{ \quad \right\} + \left[\quad \right] \left\{ \quad \right\} + \ldots \left[\quad \right] \left\{ \quad \right\}$$



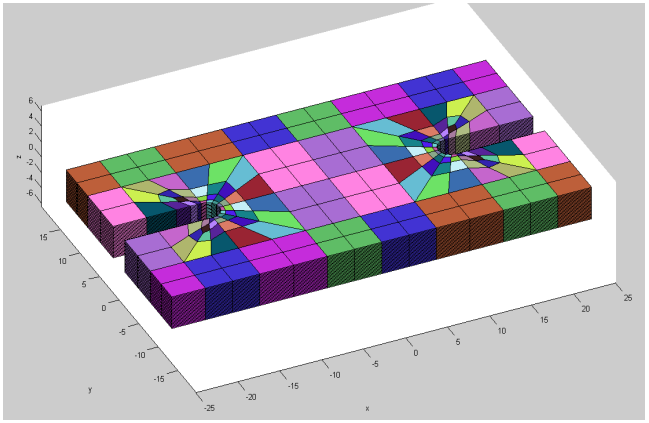
Overview


Hex-mesh
Detect
Congruency
Compute Ke of templates
Push data to GPU
Assembly-free
Ku = f on GPU

Congruence Examples

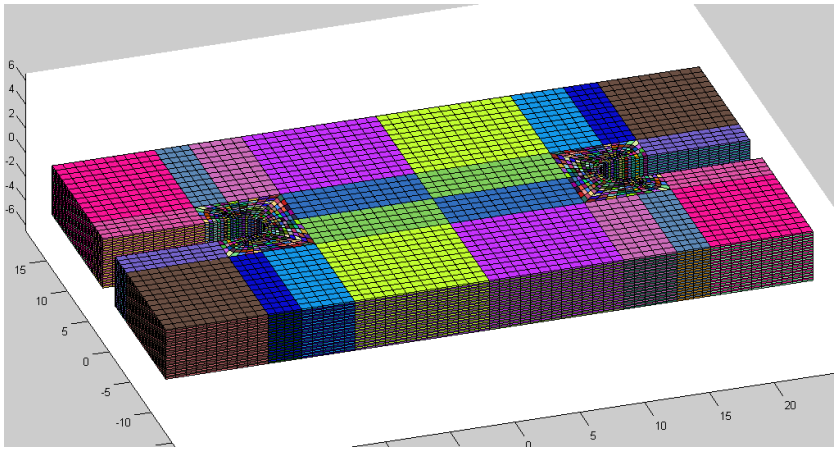


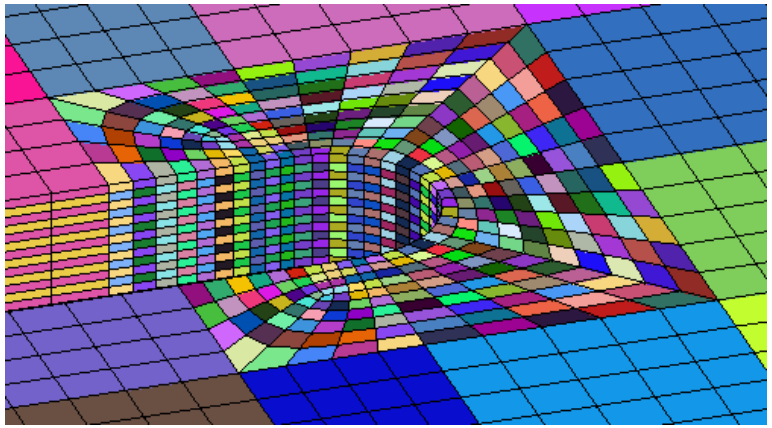
- Structured grid, isotropic material
  - 8192 elements
  - One template element
  - 1 Ke matrix (24*24); 4.6 KB



- Unstructured grid, isotropic material
  - 6144 elements
  - 34 template elements (~0.5%)
  - 34 Ke matrices; 156 KB (vs 28 MB)

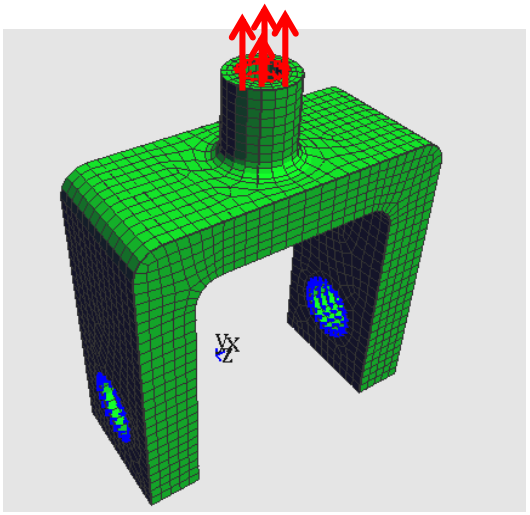
Results



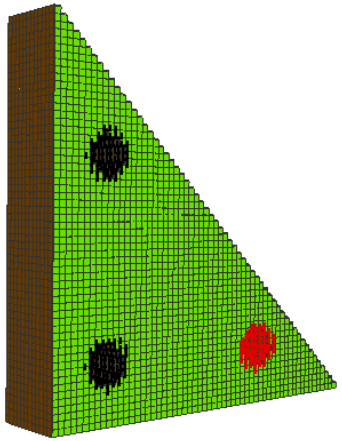


- Unstructured grid, composite layers
  - 83584 elements
  - Material congruency check
  - 322 templates (~0.4%)
  - 1.48 MB storage (vs 384 MB)

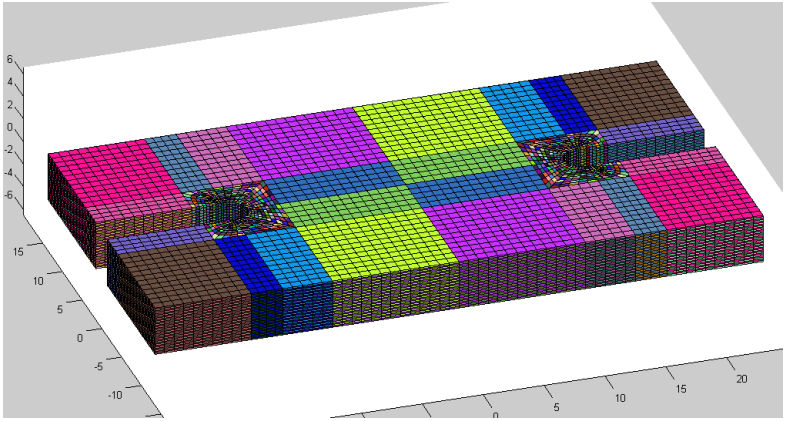
Exploit Element Congruency



120 distinct elements



1 distinct element



20 distinct elements

1.48 MB storage (vs. 384 MB)

Philosophy

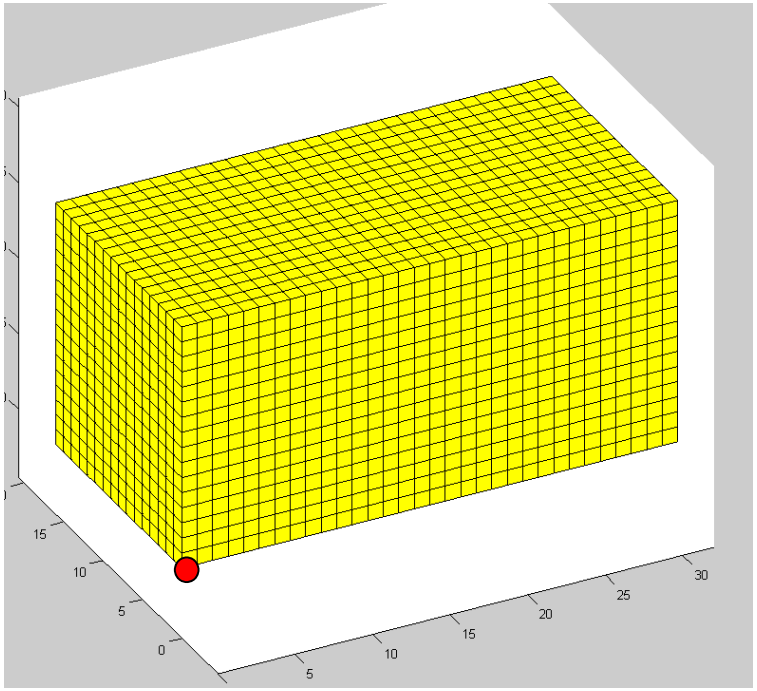
THE UNIVERSITY of WISCONSIN MADISON

Less memory you use

$\Rightarrow$ Less memory you will fetch

$\Rightarrow$ Faster will be your code!!

Assembly-free y = Kx in GPU



Algorithm (per thread/node):

For each neighboring elem
    For each of 8 nodes of elem
        Get (u,v,w) of node
        Apply Ke of elem, node
        Accumulate into uResult, …
Push uResult to Memory

GPU Data

THE UNIVERSITY of WISCONSIN MADISON

```
struct GPUData {
    // GPU Variables; see HexMesh.h for definitions
    int nNodes;// total number of nodes
    int nElems; // total number of elements
    int nTemplateElements;// how many distinct elements, 1 for a uniform hex grid
    int maxNeighboringElements;// for each node, 8 for a uniform hex grid

    int *d_elems; // 8*nElems, store the first nodes of all elements, then second nodes, etc
    int *d_neighborElems; // maxNeighboringElements*nNodes, the element neighbors of a node; the 0th nei
    int *d_nNeighboringElems; // for each node, how many valid neighboring elements for a given node,  8
    int *d_templateID; // for each element, the corresponding template id for looking up Ke stiffness ma
    int *d_freeu, *d_freev, *d_freew;// for each node, 0 implies Dirichlet, 1 implies free
    REAL *d_loadu, *d_loadv, *d_loadw; //  force on each node
    REAL *d_KETemplates;//  24*24*nTemplateElements;
    REAL *d_u0, *d_v0, *d_w0;// initial displacement guess at each node
    REAL *d_u, *d_v, *d_w;// final displacement solution at each node

    REAL *d_tempu[3],*d_tempv[3],*d_tempw[3];// temporarily used by various GPU algorithms

} ;
```

- Separate storage for (u,v,w) for coalesced memory access.
- Can reduce foot-print further

K*u on GPU

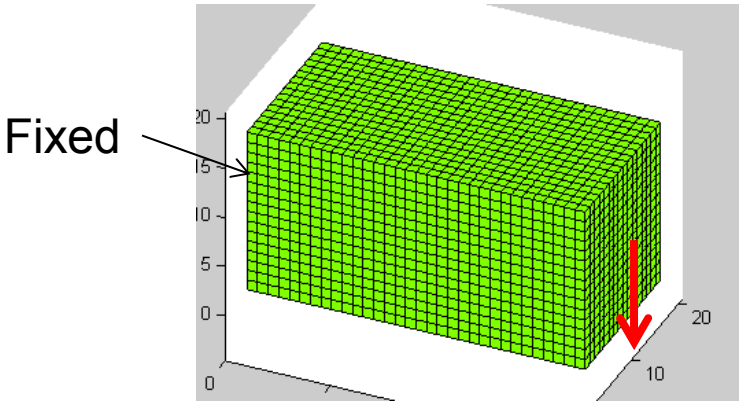
```
node = blockIdx.x*blockDim.x + threadIdx.x;

for (j =0; j < maxElem; j++) {//
    elem = d_neighborElems[(nNodes+1)*j + node];// process elem-j neighbor
    elemID = d_templateID[elem];

    rho = d_pseudodensity[elem];
    if (rho == 0)// element exists but is void
        continue;

    for (i = 0; i < 8; i++) {// now accumulate for each of the 8 nodes
        node2 = d_elems[(nElems+1)*i + elem];
        uVal = d_u[node2];
        vVal = d_v[node2];
        wVal = d_w[node2];

        KEindex = (24*3*nodeIndex + 3*i)*(nTemplateElements) + elemID;
        KE = d_KETemplates[KEindex];
        uResult += KE*uVal;


// write back the result
d_u_out[node] = ufree*uResult;// Dirichlet condition
d_v_out[node] = vfree*vResult;
d_w_out[node] = wfree*wResult;
```

Platform

THE UNIVERSITY of WISCONSIN MADISON

- 64 bit Windows Office Professional
- CPU:
  - i7, 3.2 GHz, 6 GB
  - C Code single core
- GPU:
  - GTX 480 (400 cores)
  - 1.5 GB

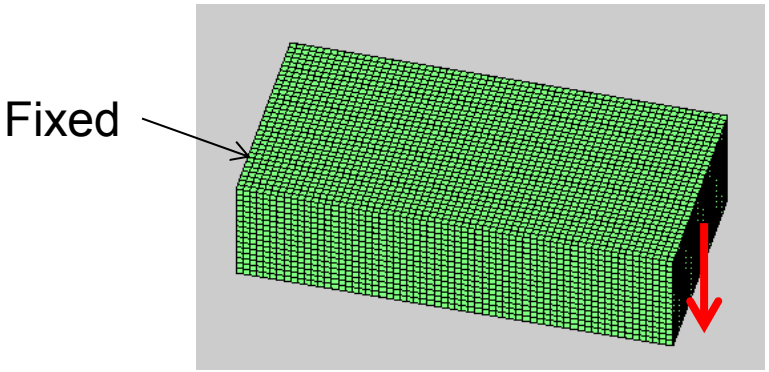
Timing: Problem 1



- Isotropic structured hex mesh
- 32 x 16 x 16; 1 template element
- 28,000 dof

	Overhead cost (s)	Solution time (s)
Matlab – Assembled; direct solve	1.8 (K assembly)	7.0
Matlab – Assembled; CG (1e-10)	1.8 (K assembly)	8.7
C– Assembly-free; CG (1e-10)	-	8.0
CUDA – Assembly-free; CG (1e-10)	0.13 (GPU transfer)	0.20

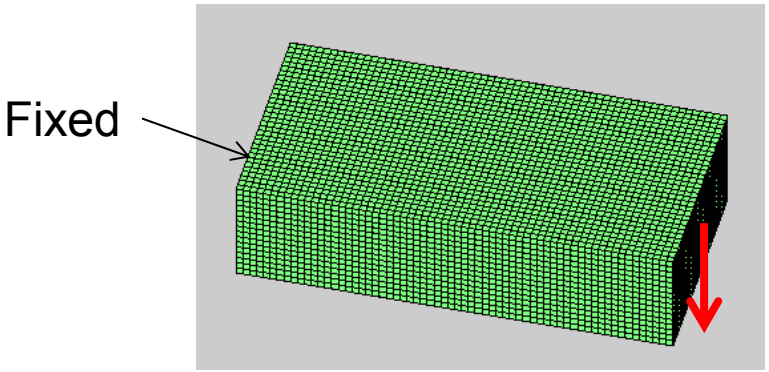
Timing: Problem 2



- Isotropic structured hex mesh
- 32 x 32 x 16; 1 template element
- 55,539 dof

	Overhead cost (s)	Solution time (s)
Matlab – Assembled; direct solve	3.6 (K assembly)	33.2
Matlab – Assembled; CG (1e-10)	3.6 (K assembly)	2.73
C– Assembly-free; CG (1e-10)	-	19.8
CUDA – Assembly-free; CG (1e-10)	0.16 (GPU transfer)	0.29

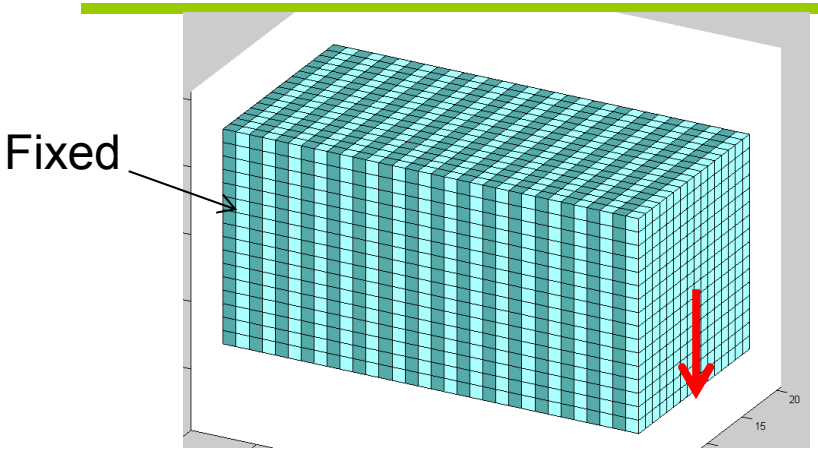
Timing: Problem 3



- Isotropic structured hex mesh
- 64x 32 x 16; 1 template element
- 109,385 dof

	Overhead cost (s)	Solution time (s)
Matlab – Assembled; direct solve	7.1 (K assembly)	> 8 mins (stalled)
Matlab – Assembled; CG (1e-10)	7.1 (K assembly)	8.8
C– Assembly-free; CG (1e-10)	0.0	60.8
CUDA – Assembly-free; CG (1e-10)	0.19 s (GPU transfer)	0.66

TIming: Problem 4



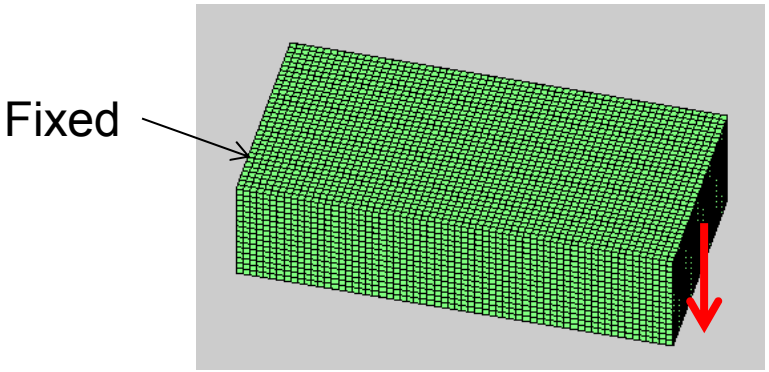
- Isotropic structured hex mesh
- 64 x 32x 16; 2 template element
- 109,000 dof

	Overhead cost (s)		Solution time (s)	
	Isotropic	Composite	Isotropic	Composite
Matlab: Direct	--	--	--	--
Matlab: CG	7.1	7.1	8.8	22.0
C: CG, AF	0.0	0.0	60.8	154
CUDA: CG, AF	0.16	0.17	0.66	1.6

Same K, but 2.5 x CG iterations.

No penalty in GPU due to non isotropy

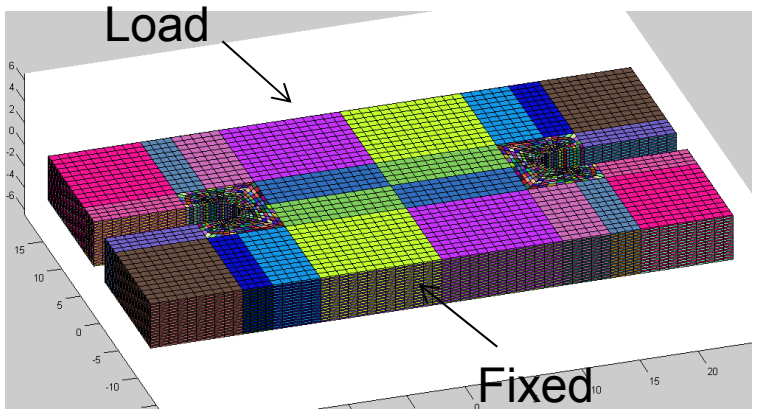
Timing: Problem 5



- Isotropic structured hex mesh
- 256 x 128 x 64; 1 template element
- 6.5 million dof
- 468 MB GPU memory

	Overhead cost (s)	Solution time (s)
Matlab – Assembled; direct solve	--	Stalls!
Matlab – Assembled; CG (1e-10)	--	Stalls!
C– Assembly-free; CG (1e-10)	--	Stalls!
CUDA – Assembly-free; CG (1e-10)	0.41 s (GPU transfer)	87.2

Timing: Problem 6

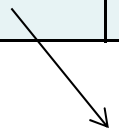


- Unstructured hex mesh
- Isotropic and Composite (alternate layers +/- 60 deg fibers)
- 274,000 dof
- 322 template elements

	Overhead cost (s)		Solution time (s)	
	Isotropic	Composite	Isotropic	Composite
Matlab: Direct	--	--	--	--
Matlab: CG	18.3	18.3	214.2	216.3
C: CG, AF	0.0	0.0	720.3	770.6
CUDA: CG, AF	0.18	0.18	29.3	35.2

Typical FEA Computing Time

	DOF	PareTO	
		CPU	GPU
Cantilever Beam; Edge	110K	8.3 secs	1.9 secs
Stool	2.7M	186 secs	32 secs
Point Load Cantilever	15M	51 mins	5.73 mins
	92M	12 hr	-

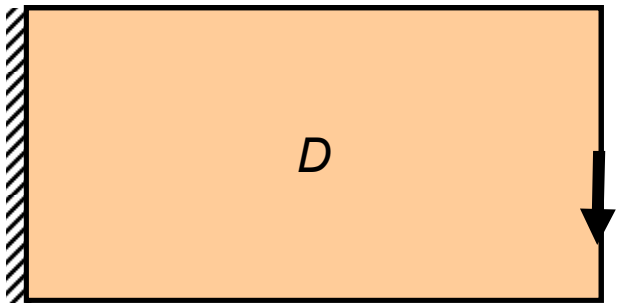


1.5 hr on 44 core (ONR)

Topology Optimization

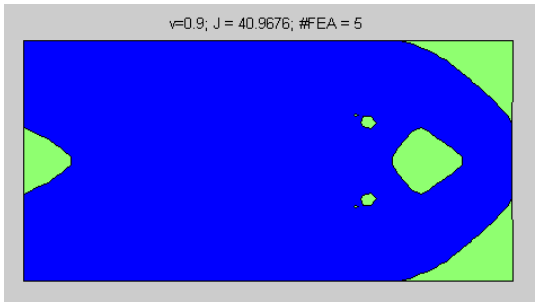
THE UNIVERSITY of WISCONSIN MADISON

Structure problem

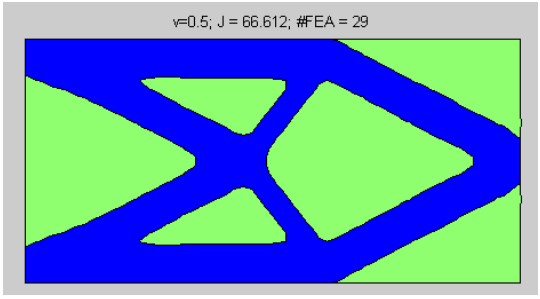


Reduce weight, but keep it stiff

Optimal topologies



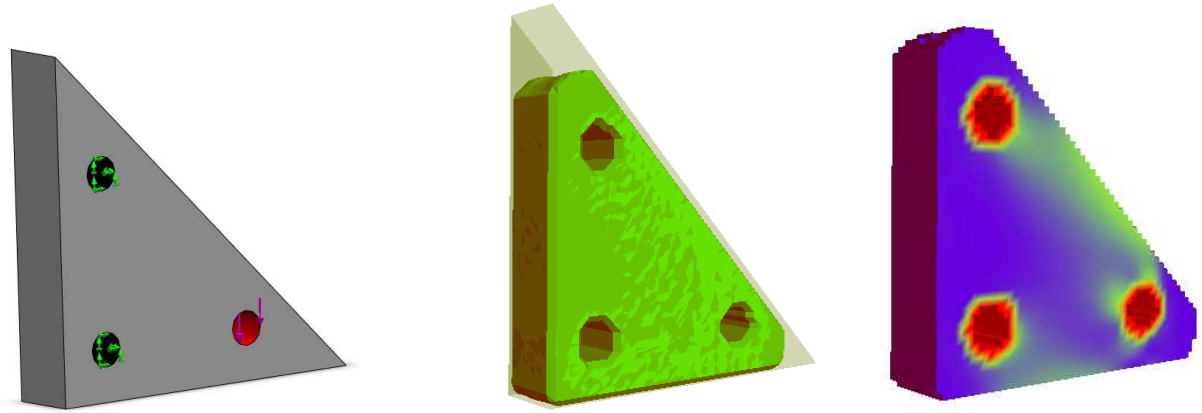
10% reduction in weight
2% loss in stiffness



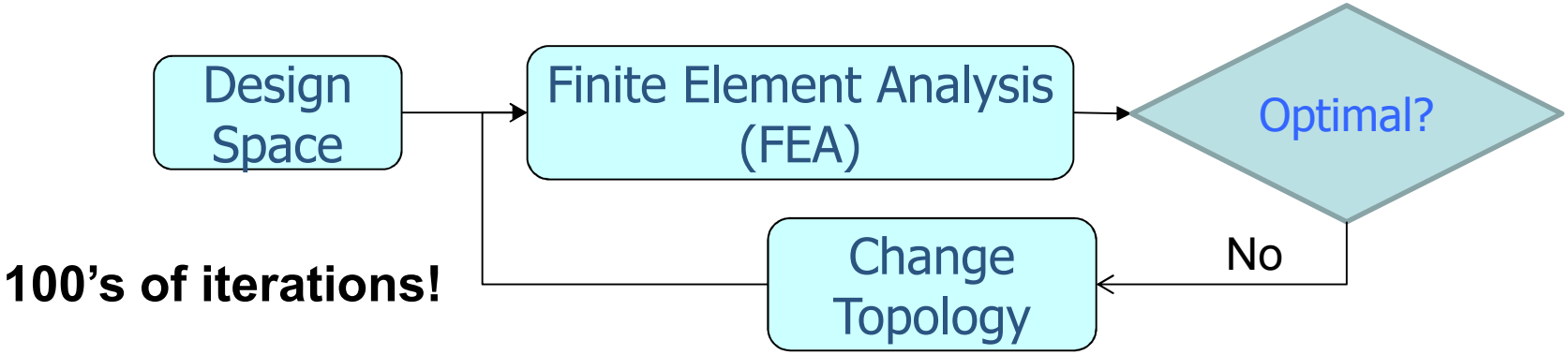
50% reduction in weight
12% loss in stiffness

Topology optimization is the systematic generation of such structures

Topology Optimization



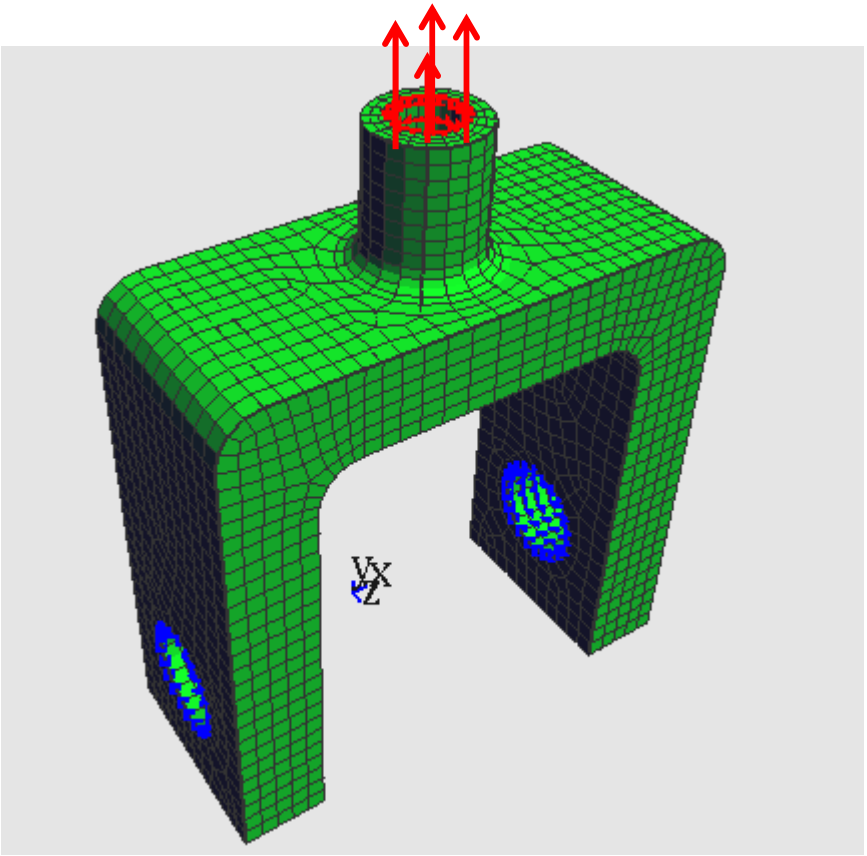
10^5 ~ 10^7 dof
Solve Ku = f
K: Sparse SPD

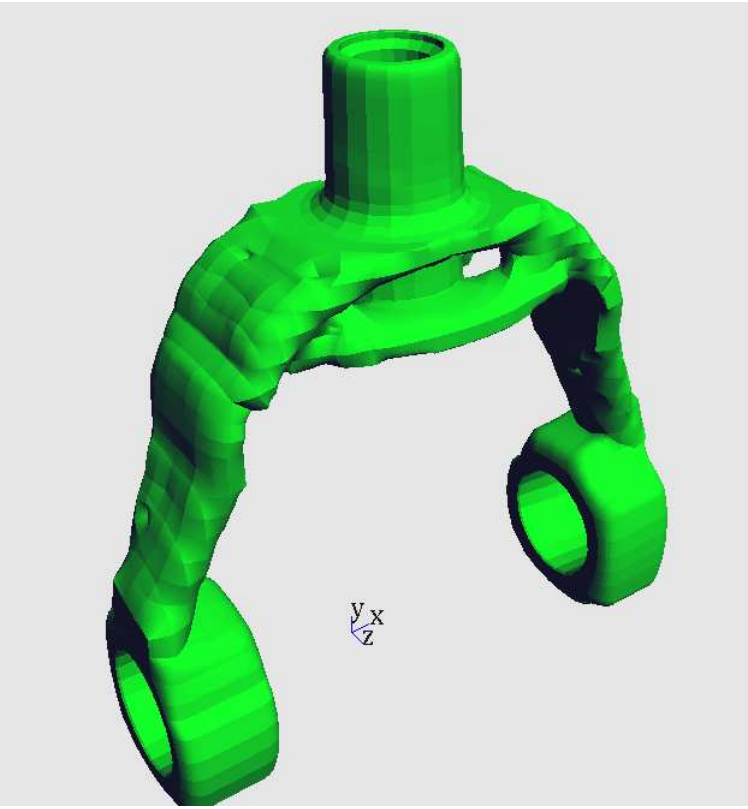


100's of iterations!

- Multi-load
- Nonlinear FEA
- Constraints
- …

Knuckle Optimization



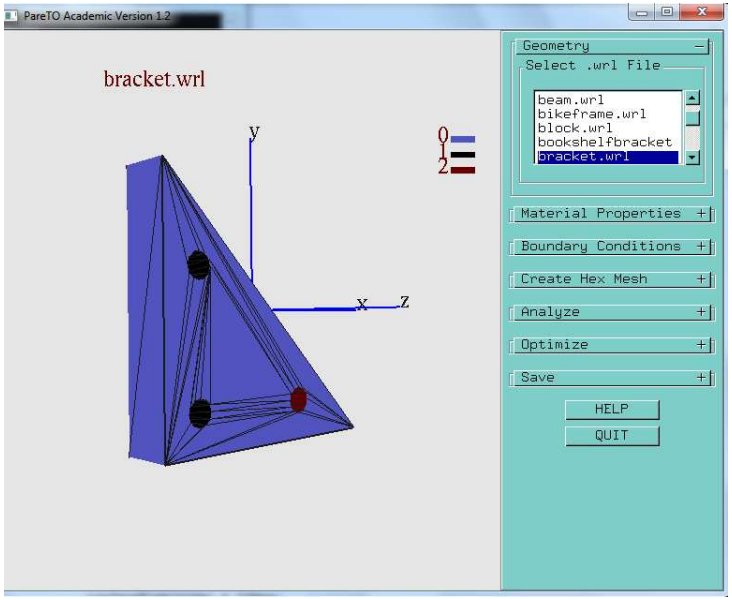


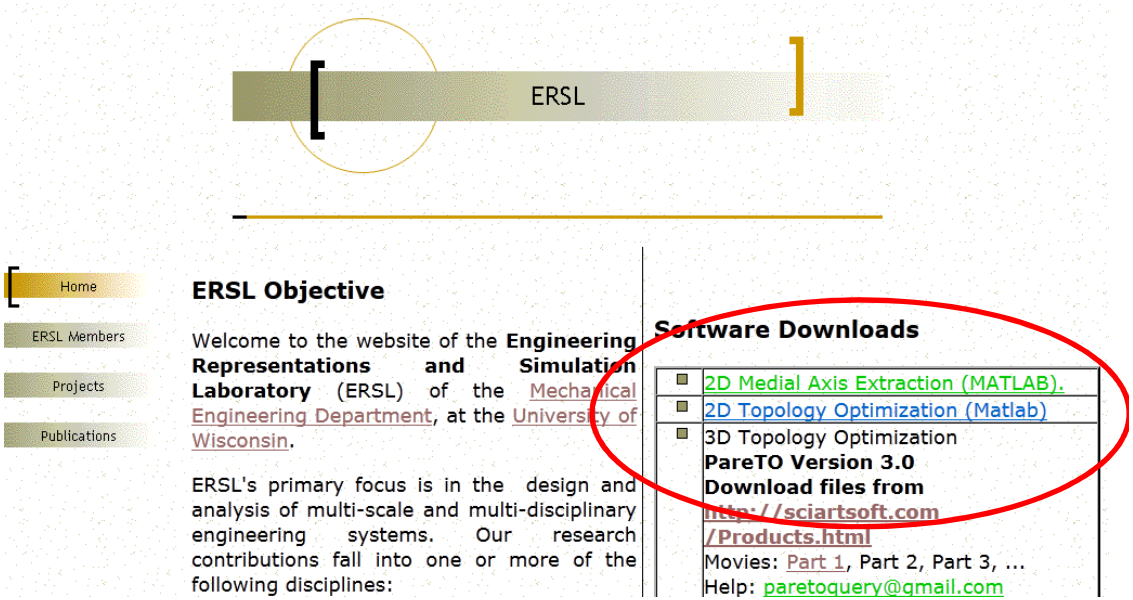
TopOpt Computing Time

THE UNIVERSITY of WISCONSIN MADISON

Name of part & volume fraction	**DOF**	**Pub. Data**	**PareTO**	
			CPU	GPU
Cantilever Beam; Edge (0.50)	110K	2.4 hr	200 secs	45 secs
Knuckle (0.55)	20K	--	111 secs	44 secs
Bridge (0.35)	113K	--	2 mins	36.2 secs
Stool; N=96 (0.20)	2.7M	21.8 hr	1 hr, 24 mins	14 mins
Point Load Cantilever (0.50)	783K	3.9 hr	16 mins	125 s
	15M	-	19hr, 28 mins	2hr, 12 mins
	92M	-	12 days, 2hr	-

Download: www.ersl.wisc.edu





Email: suresh@engr.wisc.edu

Acknowledgements

- Graduate Students
- NSF
- UW-Madison
- Kulicke and Soffa
- Luvata
- Trek Bicycles

Publications available at
www.ersl.wisc.edu

Email
suresh@engr.wisc.edu