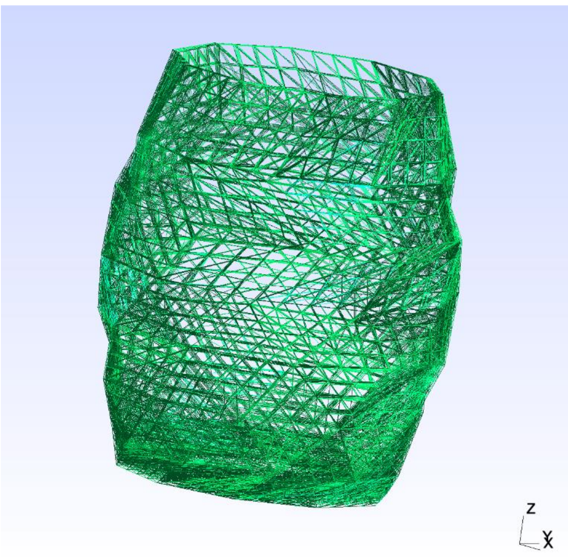
Simulating Deformation with a Parallel CUDA FEM Solver

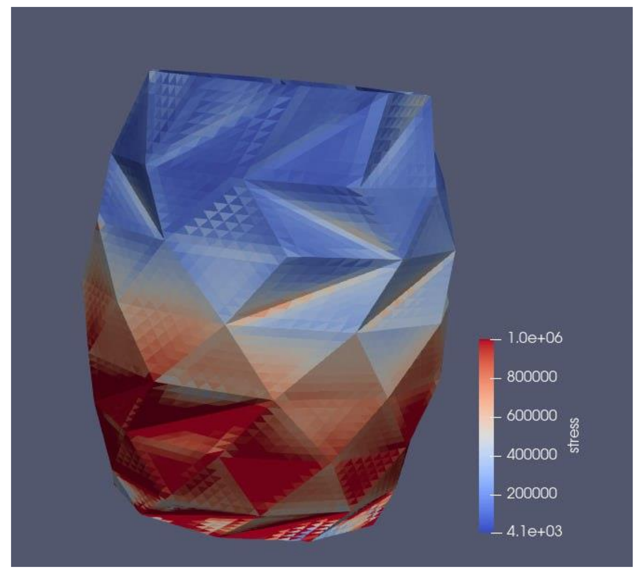
Carnegie Mellon University

Hong Lin (honglin) Ryan Lau (rwlau)

15-418 Final Project

I. DSM Background





The **Direct Stiffness Method (DSM)** is a variant of Finite Element Analysis that calculates displacements or internal stresses of an object under load forces.

Input: 3-D mesh of N nodes (each with 3 position coordinates) and T tetrahedral elements (with 4 node IDs), material constants (E, ρ, ν), boundary conditions (e.g. gravity, external forces).

Output: Von Mises stress for each element. Each stress is a scalar value.

II. DSM Stages

Wall-clock time for unit cube mesh: $N = 3419,\ T = 15923$

STAGE \ IMPL	CPU DENSE	GPU DENSE	GPU EBW
Stage 0 Allocating and initializing matrices	.189% .1945s (1x)	2.43% .0751s (2.59x)	40.0% .0685s (2.84x)
Stage 1 Computing local stiffness matrices and assembling the global stiffness matrix	.012% .0123s (1x)	.004% .0001s (9.49x)	.057% .0001s (12.6x)
Stage 2 Imposing boundary conditions (gravity and planar forces)	.001% .0013s (1x)	.032% .0010s (1.28x)	.687% .0011s (1.08x)
Stage 3 Solving for nodal displacements by conjugate gradient	99.8% 102.8s (1x)	97.5% 3.019s (34.1x)	59.2% .1015s (1013x)
Stage 4 Post-processing: computing of element von Mises stresses from the nodal displacements	.002% .0019s (1x)	.006% .0002s (10.5x)	.107% .0002s (10.4x)

III. DSM Overview

Matrix	Dim	Description
R	$(N, 3)$	Node coordinates
M	$(T, 4)$	Tetrahedron Node ID's
K	$(T, 12, 12)$	Local stiffness matrices
K_g	$(3N, 3N)$	Global stiffness matrix
A	$(N, *)$	Node adjacency sets
S	$(*,)$	List of surface faces in the mesh
F	$(3N,)$	External forces
u	$(3N,)$	Node displacements

1. Stiffness Matrices

For each tetrahedron M_i, calculate the local stiffness matrix K_i :

$$K_i = V_i B_i^T D B_i$$

$(K \in \mathbf{R}^{12\times12},\ V \in \mathbf{R},\ D \in \mathbf{R}^{6\times6},\ B_i \in \mathbf{R}^{6\times12})$

Assemble the global stiffness matrix K_g by summing over all K_i, relocating each degree of freedom (dof) according to its node ID.

Sparsity: All nonzero elements of K_g correspond to vertices and edges. Max-degree typically $O(1)$.

Parallelization: Computation of K_i can be data-parallelized over elements. For global summation, can statically determine nonzero entries and memory addresses of nonzero entries and synchronize using `atomicAdd`.

2. Boundary Conditions

Compute f, the external forces at every node.

Gravity: Compute each element's volume and distribute the weight to its nodes. Can be data-parallelized; synchronize with `atomicAdd`.

Planar forces: Identify the closest nodes and distribute the force over the surface faces. Some inter-thread dependencies to rank and filter nodes; synchronize using `atomic` operations and kernel lifecycles.

Dirichlet points: Must fix motion of ≥ 3 nodes to eliminate rigid body modes and make system solvable. Zero out corresponding entries in f and K_g (except diagonal entries, which are set to 1). Trivial to parallelize.

3. Solve for u

Solve $K_g u = F$. Since K_g is **positive semi-definite**, we can use the **conjugate gradient** method, which is iterative:

function CG(K_g, F, ε, u_0, `max_iter`)
 $r_0 \leftarrow F - K_g u_0$
 $p_0 \leftarrow r_0$
 for `i` in `1...max_iter` **do**
  $\alpha_i \leftarrow \frac{r_i^T r_i}{p_i^T K_g p_i}$
  $u_{i+1} \leftarrow u_i + \alpha_i p_i$
  $r_{i+1} \leftarrow r_i - \alpha_i K_g p_i$
  if $|r_{i+1}| \leq \varepsilon$ **then**
   return x_{i+1}
  $\beta_k \leftarrow \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
  $p_{i+1} \leftarrow r_{i+1} + \beta_i p_i$
 return x_{i+1}

Matrix-vector product: **97.5%** of all CUDA kernel runtime on gpu-dense, $N = 2199,\ T = 9636$.

Convergence: Iterations required scales with $O(N^{0.389})$ for meshes of our unit cube. ($R^2 =$ .999).

For $N = 51572,\ \varepsilon = 10^{-5}$, 1822 iterations were required.

Factor of ≥ 10 larger for vase mesh with same T.

4. Post-Processing

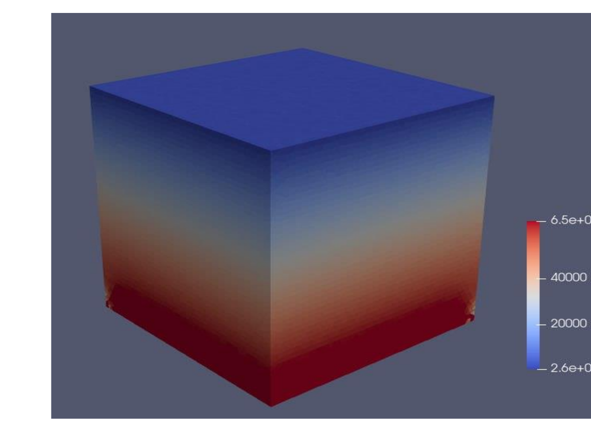
Compute element stress tensors:

$$\sigma_i = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yz}, \sigma_{zx}) = DB_i u_i$$

Convert to von Mises stress (scalar):

$$\sigma_v^2 = \frac{1}{2}\Big((\sigma_{xx} - \sigma_{yy})^2 + (\sigma_{yy} - \sigma_{zz})^2 + (\sigma_{zz} - \sigma_{xx})^2 + 6(\sigma_{xy}^2 + \sigma_{yz}^2 + \sigma_{zx}^2)\Big)$$

Output as a file for visualization in e.g. Paraview:



IV. Sparse Matrix Representations

1. Compressed Sparse Row (CSR)

2		4	
	6		
		10	
12			14

values

2	4	6	10	12	14

col_indices

0	2	1	2	0	3

row_ptr

0	2	3	4	6

2. Ellpack (ELL)

Improves **data locality** and access patterns by storing the same number of elements (with padding) for each row

Store `values` and `col_indices` in column-major order to improved **coalesced memory accesses**.

2		1	3
	6		
		10	
12			14

values

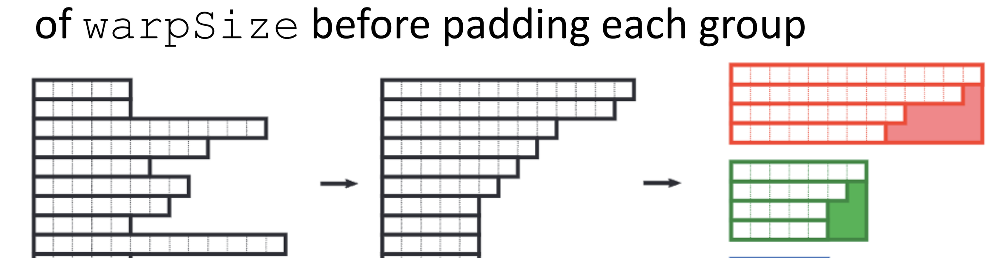
2	1	3
6	-	-
10	-	-
12	14	-

col_indices

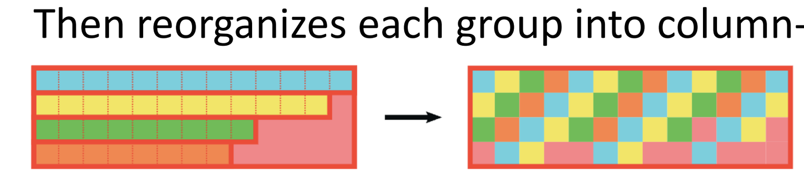
0	2	3
1	-	-
2	-	-
3	1	-

3. ELL-WARP [Wong, Kuhl, Darve (2015)]

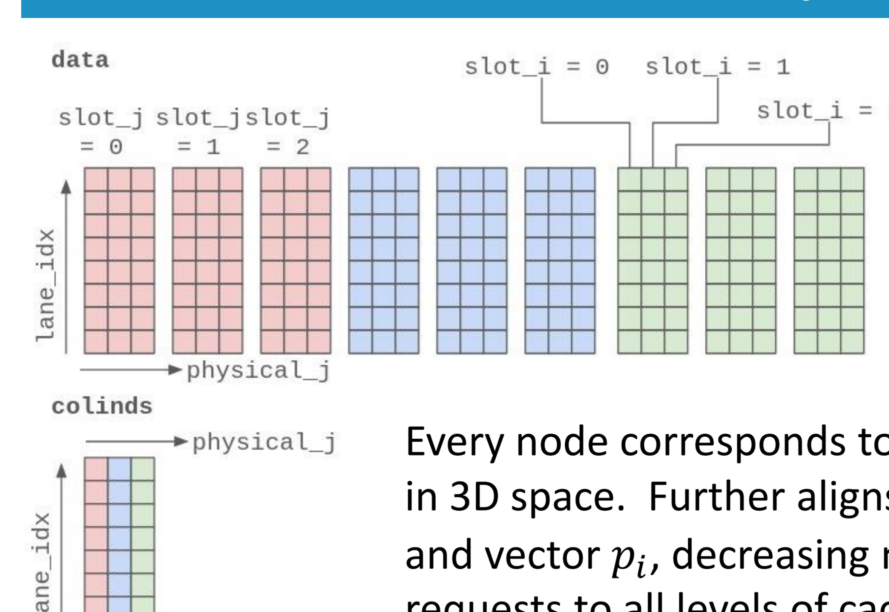
Improves thread **work distribution** and **reduces memory usage** by sorting matrix columns by length and organizing into groups of `warpSize` before padding each group



Then reorganizes each group into column-major order



3. ELL-Block-WARP (EBW)



Each thread gets 3×3 block of elements. Processes 3 rows at a time.

Every node corresponds to 3 rows and cols in 3D space. Further aligns accesses to K_g and vector p_i, decreasing number of requests to all levels of caches.

V. Results

- **Dense methods OOM** at roughly $N > 10000$. Sparse methods support mesh sizes of over $N = 1{,}000{,}000$.
- **Main improvements** across sparse matrix implementations are due to **reduced DRAM loads**
- ELL implementations have $5 - 10\times$ speedup over CSR. EBW shows $10 - 20\%$ improvement over other ELL variants at large mesh sizes.
- SpMV (Sparse Matrix-Vector multiplication) implementations are **bottlenecked by warp stalls** due to irregular accesses to the dense vector. Setup times also become significant portion of the runtime and could be further optimized.