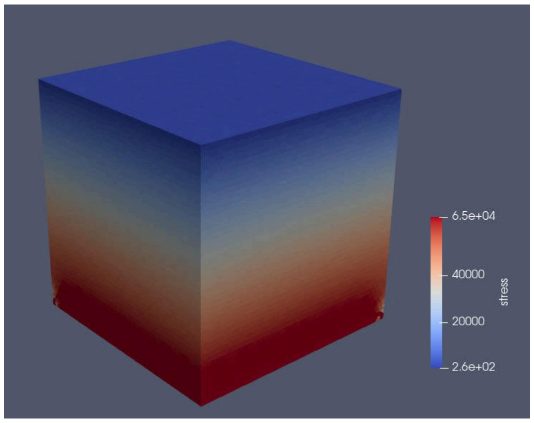
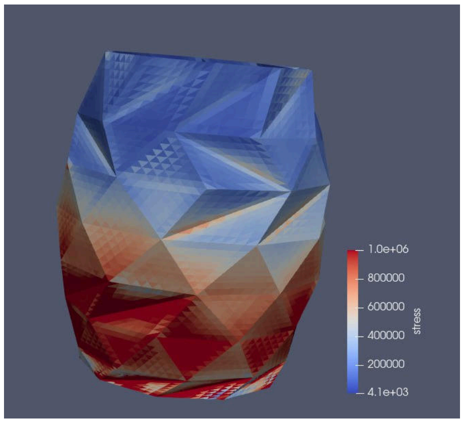
A Parallel CUDA FEM Solver

Hong Lin
honglin@andrew.cmu.edu

Ryan Lau
rwlau@andrew.cmu.edu





***Abstract*—The Finite Element Method (FEM) is a commonly used technique in engineering and physics to solve complex partial differential equations numerically by splitting the input domain into smaller, discrete elements. However, most modern FEM solvers like OpenEMS and Autodesk FEA do not support GPU acceleration. In this project, we aim to develop a parallel CUDA implementation of the Direct Stiffness Method (DSM), a variant of FEM used for calculating the internal stresses of an object under external load forces. In particular, we focused on exploring and optimizing various sparse matrix storage methods and sparse matrix-vector multiplication (SpMV) techniques while solving for the mesh node displacements via the conjugate gradient method.**

I. Background

The Finite Element Method (FEM) is a powerful numerical technique for approximating solutions to partial differential equations, with widespread applications in many fields like chemical engineering, geophysics and computer graphics. Most FEMs work by breaking down the domain into discrete elements. Local properties are calculated for each element before being assembled into a global system of equations.

The Direct Stiffness Method (DSM) is one variant of the FEM used in structural engineering that calculates the internal stresses within an object due to external load forces. In 3 dimensions, an object can be discretized as a mesh consisting of node points and tetrahedral elements each represented as a set of 4 node points.

Our implementation of DSM takes in a tetrahedral mesh representing a solid object, forces acting on the mesh, material properties of the object (Young's modulus E, Poisson ratio ν and material density ρ) and computes the von Mises stress output experienced by each element in the mesh. The mesh is specified by a `.msh` file (produced by a `gmsh` application from an `.stl` specification of the object), and the output is a comma-separated list of stresses that can, with simple postprocessing, be consumed by visualization software such as ParaView.

The key data structures in our program are various matrices:

TABLE I: Matrix and vector variables involved in the DSM algorithm.

Matrix	Dimension	Description
R	$(N, 3)$	Position of each node in the mesh
M	$(T, 4)$	IDs of the nodes of each tetrahedron
∇N	$(T, 4, 4)$	Gradients of each tetrahedron's shape functions
K	$(T, 12, 12)$	Local stiffness matrices. Each entry corresponds to a node and coordinate of the tetrahedron.
K_g	$(3N, 3N)$	Global stiffness matrix. Sum over all local stiffness matrices (at the corresponding nodes)
A	$(N, *)$	Adjacency sets of each node in the mesh
S	$(*,)$	List of surface faces in the mesh. A surface face is a face that belongs to exactly one element in the mesh.
F	$(3N,)$	External forces, such as gravity and contact forces.
u	$(3N,)$	Node displacements for each node and coordinate

Here, we shall present the calculations involved for each main step of the DSM algorithm and discuss their challenges and opportunities for parallelism.

A. *Calculating the element local stiffness matrices*

The first step of the algorithm involves calculating the local stiffness matrix K for every element of the mesh. K is a matrix that describes how the displacement of each node's degree of freedom is affected by the forces applied to nodes in the element. Elements of K thus have units of force per unit length. Under the assumptions of small displacements and a homogeneous material, for some tetrahedral element T_i with nodes $[n_a, n_b, n_c, n_d]$, K_i can be approximated as

$$K_i = V_i B_i^T D B_i \tag{1}$$

where V_i is the volume of the element, B_i is the strain displacement matrix (6×12) and D is the material matrix (6×6). Both V_i and B_i are calculated or constructed using the coordinates of the nodes. D is a constant for all tetrahedral elements and is derived from E and ν. Full mathematical descriptions of V_i, B_i, D can be found in [1] and [2].

Since the matrices involved are of fixed and relatively small sizes, they can be stored and accessed simply as dense 2D arrays. Additionally, the local stiffness matrix calculations are independent from those of any other element, and can hence be data-parallelized.

B. *Assembling the global stiffness matrix*

The global stiffness matrix K_g describes how the displacement of all degrees of freedom relates to forces applied on all nodes. Each entry in K_g is the sum of elements of corresponding degrees of freedom in all local stiffness matrices. In other words, for each vertex and edge of each tetrahedron, there are 3×3 elements that have to be updated, which may be a source of locality.

We note that assembling this global stiffness matrix would require some form of synchronization. Since vertices and edges can be shared by several tetrahedra, parallelizing over tetrahedra (as a continuation of computing the element local stiffness matrix, to avoid materializing every single local stiffness matrix) may lead to multiple threads attempting to update the same element of the matrix.

Fortunately, as addition is commutative and associative, simply performing element-wise atomic additions is sufficient as a starting point for correctness (notwithstanding the properties of floating-point arithmetic).

The next problem becomes the representation of the global stiffness matrix. The size of K_g scales quadratically with the number of nodes but remains sparse since only elements whose rows and columns correspond to nodes that share an element will be non-zero. If we were to use a sparse representation of the matrix, then this introduces a nontrivial time to determine which entries are nonzero (i.e. computing the adjacency set of the mesh), which may have to be parallelized.

C. *Applying boundary conditions*

The final global system takes the form

$$K_g u = F \tag{2}$$

where u is the vector of unknown nodal displacements and F is a vector of the applied external forces.

Firstly, we note that the above linear equation is underdetermined. There are the rigid body modes of freedom (i.e. translations or rotations of the whole object) that do not result in any internal deformation. It thus becomes necessary to fix the motion of at least 3 nodes in the mesh in order for (2) to be solvable. These are the Dirichlet boundary conditions, achieved by by zeroing out the rows and columns of K_g (except the diagonals, which should be set to unity) and corresponding entries in F [3].

TABLE II: Code for setting Dirichlet boundary conditions

```
1: function DIRICHLET($K_g$, $F$, fixed_node_idxs)
2:     for node_idx in fixed_node_idxs do
3:         for dof in 0...2 do
4:             dof_index ← node_idx× 3 +dof
5:             $K$[dof_idx, :] ← 0
6:             $K$[:, dof_idx] ← 0
7:             $K$[dof_idx, dof_idx] ← 1
8:             $F$[dof_idx] ← 0
```

In addition, we want to study the deformation of the mesh under various types of forces:

- Gravity, which acts on every node in the mesh based on the mass of its adjoining elements; and
- Contact forces (e.g. the normal force of a cube resting on a surface), which act on the closest nodes with respect to a given direction, and is distributed based on the oriented area of the adjoining surface faces.

Since gravity has to be computed for every node, its workload is similar to computing the local stiffness matrix for each element.

The more challenging task is determining which nodes to compute the contact force. At the minimum, there are some dependencies between nodes when determining which nodes are closest to the direction the force is applied. Subsequently, we have to compute the faces to apply the force on and then distribute the force over every face. We can expect a large number of faces to not experience this force so there is some further sparsity in the problem.

D. *Solving the global system*

Since K_g is a symmetric positive semi-definite matrix, we can solve for u via the conjugate gradient (CG) method [4].

The CG method involves evaluating multiple iterations of the matrix vector product $K_g p_i$. For large meshes, the size and sparsity of K_g poses significant memory usage and computational challenges, making it critical to design efficient parallel matrix product algorithms that exploit the sparsity and locality of K_g. On the smallest scales, K_g is dense (as each edge/vertex contributes 3×3 nonzero elements), but on a slightly larger scale, K_g becomes sparse.

TABLE III: Code for solving u iteratively using the Conjugate Gradient method

```
1:  function CG($K_g$, $F$, $\varepsilon$, $u_0$, max_iter)
2:      $r_0 \leftarrow F - K_g u_0$
3:      $p_0 \leftarrow r_0$
4:      for i in 1...max_iter do
5:          $\alpha_i \leftarrow \frac{r_i^T r_i}{p_i^T K_g p_i}$
6:          $u_{i+1} \leftarrow u_i + \alpha_i p_i$
7:          $r_{i+1} \leftarrow r_i - \alpha_i K_g p_i$
8:          if $|r_{i+1}| \leq \varepsilon$ then
9:              return $x_{i+1}$
10:         $\beta_k \leftarrow \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
11:         $p_{i+1} \leftarrow r_{i+1} + \beta_i p_i$
12:     return $x_{i+1}$
```

In the largest meshes we tested on, we observed that several thousand iterations were needed for CG to converge. This means that CG should likely occupy the bulk of the runtime and be the priority to optimize.

E. Postprocessing

After solving for u, the final step is to compute the internal stresses within each element. This postprocessing phase allows us to visualize the regions of high stress and potential failure points of the object. For linear elastic materials, the internal stress tensor is given by

$$\sigma_i = \begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{pmatrix} = DB_i u_i \tag{3}$$

where u_i is the vector of nodal displacements for the element. To evaluate the material yield critieria, we convert the stress tensor into a scalar using the von Mises stress σ_v

$$\sigma_v^2 = \frac{1}{2}\Big(\left(\sigma_{xx} - \sigma_{yy}\right)^2 + \left(\sigma_{yy} - \sigma_{zz}\right)^2 + \left(\sigma_{zz} - \sigma_{xx}\right)^2 + 6\left(\sigma_{xy}^2 + \sigma_{yz}^2 + \sigma_{zx}^2\right)\Big) \tag{4}$$

We then save σ_v alongside each element in a `.vtu` file for stress visualization in Paraview. Similar to the local stiffness matrix step, each element's σ_v is independent and can be data-parallelized.

II. Approach

Due to the large number of similar, simple arithmetic operations that can be performed, we decided to target CUDA as our platform of parallelism. We used the same CUDA API as Assignment 2, which extends a subset of C++. We targeted the GHC clusters, which are equipped with NVIDIA RTX 2080 GPUs.

We started by creating a script to generate tetrahedral meshes using `GMSH`. The script converts `.stl` files to `.msh` files that contain a list of coordinates of each node and a mapping of nodes to tetrahedrons. For the purposes of benchmarking our serial and parallel algorithms, we generated meshes of a unit cube at varying levels of mesh sizes.

To consume our output, we wrote a script to convert our output (comma-separated values of the von Mises stress) into `.vtu` files which can be consumed by ParaView, a visualization tool.

Our baseline serial algorithm parses the mesh and stores a list of nodes as triplets of 3D coordinates and a list of tetrahedrons as 4-tuples of node indices. Local stiffness matrices for each tetrahedral element are calculated sequentially and added to a dense global stiffness matrix represented as a 2D array. The conjugate gradient method uses a simple iterative matrix-vector multiplication algorithm, looping through the rows and columns of K_g to calculate the dot products of each row with p_i. However, we quickly ran into runtimes in the order of minutes with a mesh size of $N \geq 3419$.

A. Dense matrix

A simple initial improvement over the serial implementation would be to parallelize the dense matrix vector multiplication. Since multiple threads may update the same element of the global stiffness matrix during combination of the local stiffness matrices, we use atomic add operations to prevent possible race conditions.

The dense matrix vector multiplication operation may be parallelized over rows, where each thread calculates the dot product of a row of K_g with F. To reduce memory loads, threads in a thread block load in elements from F `blockSize` at a time into shared memory before accumulating their dot products.

However, this can lead to uncoalesced memory access patterns. To avoid this, for larger K_g, we instead have each thread block calculate the dot product over a single row. Each thread starts at the `threadIdx`th element of the row and calculates the cumulative partial product with every other `blockSize` element. The partial dot products are then combined using parallel range sums.

Unfortunately, storing all elements of K_g quickly leads to memory issues for meshes larger than a few thousand elements and multiplying across the full matrix is incredibly computationally inefficient.

B. Compressed Sparse Row (CSR) matrix

In most mesh based methods, the global matrices are typically large and sparse. For instance, our most complicated mesh, the car mesh, has $N = 12,567$ and $T = 68,388$. However, the maximum degree over all nodes is $J = 82$, as each node only shares tetrahedrons with a small number of other nodes. As nonzero entries only occur at edges in the mesh, this upper-bounds the number of nonzero entries by $3N \times 3J$.

One of the most common sparse matrix formats is the Compressed Sparse Row (CSR). Elements are stored as three 1D arrays:

- `values`: a list of non-zero entries in row-major order
- `col_indices`: the column index of each element in the `values` array
- `row_ptr`: index in the `values` array where each row starts

We note that with a dense matrix, we run out of memory on the RTX 2080 at $N \leq 12,342$, but with CSR, we are still able to process matrices as large as $N \geq 229,515$.

In the sparse-matrix vector (SpMV) kernel for CSR, we allocated one CUDA thread to a single row of the input matrix, producing one entry in the result vector. Hence, there is no need for synchronization (other than waiting for all blocks to complete, which is handled by the CUDA runtime).

C. Ellpack matrix

Ellpack is another sparse matrix format that improves on the data locality and regularity of access patterns of CSR by storing the same number of elements `ellpackCols` for each row. Ellpack organizes the matrix into two dense arrays:

- `values`: A 2D array where each row stores the nonzero entries of the corresponding matrix row. The value on the diagonal is always stored as the first element in its row and the rest of the non-zero entries are stored in order of column index.
  If a row has fewer than `ellpackCols` non-zero entires, the remaining entires are padded with a dummy value.
- `col_indices`: A 2D array storing the column index for each entry in `values`.

Since the connectivity of the nodes can be deduced from the mesh input beforehand, during the setup phase of the solver, we compute the adjacency sets for each node, take `ellpackCols` as the maximum size of the adjacency sets times 3, and populate `col_indices` according to the adjacency sets. Threads can then use atomic add operations to update the matrix when assembling the global stiffness matrix.

In the SpMV kernel for Ellpack, we allocated one CUDA thread to a single row of the input matrix as well, producing one entry in the result vector. Storing `values` and `col_indices` in column-major order optimizes global memory access patterns (unit stride over `values` and `col_indices`), improving throughput (as measured by iterations of conjugate gradient per second) by about 33% (as compared to row-major), i.e. the stride of the vertical-axis is 1, and the stride of the horizontal-axis is the number of rows $3N$.

D. ELL-WARP

The number of neighboring nodes may differ significantly across the mesh. For example, nodes at the faces and corners will naturally have much fewer neighboring nodes that share tetrahedral elements. As such, some threads in a warp may end up processing rows in K_g with more padding elements than others, resulting in wasted work.

An approach by Wong, Kuhl and Darve in 2015 [5] known as ELL-WARP attempts to mitigate this. They first sort the nodes according to their connectivity degree before constructing K_g. Then, by grouping contiguous groups of `warpSize` rows together, rows only have to be padded up to the maximum length of rows within the group. This ensures that each warp gets a group of rows that have similar amounts of padding and minimizes the total number of padding columns that have to be stored. Finally, we store the permutation vector (and inverse permutation vector) mapping the logical i-indices (i_L) and the physical i-indices (i_P), so that in SpMV, we will know which element of the vector to read from.

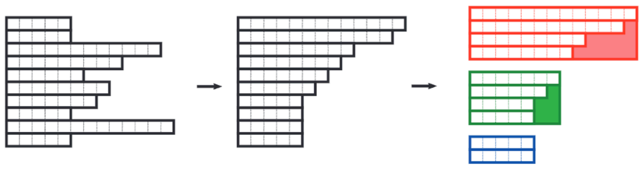


Fig. 2: With a warp size of 4 for this example, Ellpack rows are sorted by length, split into groups of warp size, and padded to the maximum length of the group.

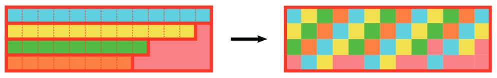


Fig. 3: Rows within a group are reordered into column major order

Similarly, we optimize for coalesced memory access patterns by storing each group of rows in column-major order. Specifically, from the slowest-varying to the fastest-varying, the memory is laid out as follows:

1) `warp_idx` $= \lfloor i_P / \text{ warp\_size} \rfloor$
2) $j_P =$ 'Physical' j-index in Ellpack (from 0 to `warp_lengths[warp_idx]`)
3) Lane index $= i_P \bmod$ `warp_size`

The SpMV kernel for ELL-WARP is very similar to Ellpack, with one CUDA thread responsible for one row of the input matrix, but now the iteration length for each warp is dependent on `warp_lengths[warp_idx]`.

E. EBW

The final optimization we made came from the observation that all entries in the matrix occur in 3×3 blocks, due to the 3-D space that we operate in. In ELL-WARP, while there is some theoretical locality in what each group of 3 adjacent threads may access (since they will access the same entries in the input vector), there are too many such different, randomly-distributed groups in a matrix for any coalescing of memory accesses to occur.

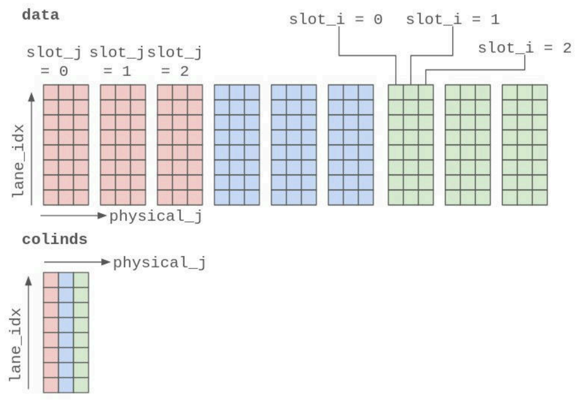


Fig. 4: EBW Memory Layout for a single warp. Each color corresponds to a column of a warp in the `colinds` matrix. (Here, `warp_size = 8` for illustration purposes.) Each element in the `colinds` matrix maps to 3×3 elements in the `data` matrix. Both matrices are stored in column-major order according to the above illustration; the `slot_i` and `slot_j` values are obtained by taking the modulus of the logical i and j indices with respect to the block size (3×3), but have a higher stride than `lane_idx` to facilitate memory access coalescing.

In EBW (ELL-Block-Warp), we divide each row of the matrix into 3×3 blocks (implemented as template parameters). In the SpMV kernel, each thread is responsible for one row of blocks (i.e. a row of 3 elements). The memory layout, from slowest-varying to fastest-varying is now given by:

1) `warp_idx` $= \lceil i_P / \text{warp\_size} \rceil$
2) 'Physical' j-index in Ellpack (from 0 to `warp_lengths[warp_idx]`)
3) $j_P \bmod 3$
4) $i_P \bmod 3$
5) Lane index $= i_P \bmod$ `warp_size`

Each thread therefore computes 3 independent entries in the result vector. Similarly, there is no need for synchronization (besides waiting for all blocks to complete).

III. Results

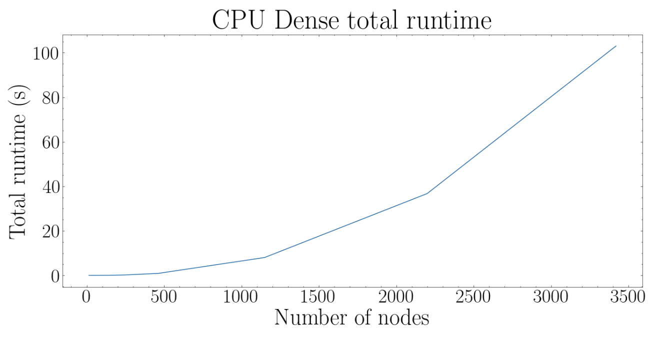


Fig. 5: CPU serial implementation run time over different unit cube mesh sizes

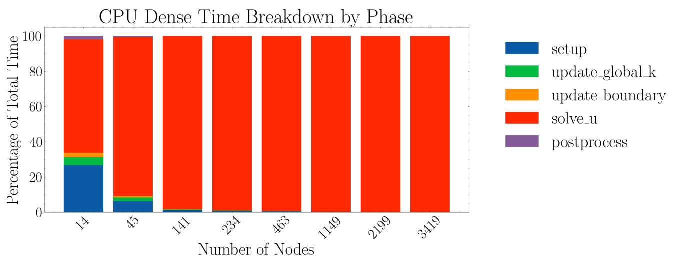


Fig. 6: Percentage of CPU runtime by phase. `setup` involves the parsing of the mesh file and the initialization of K_g. `update_global_k` calculates the local stiffness matrices for each tetrahedron and addes them to K_g. `update_boundary` generates the force vector F and applies the Dirichlet boundary conditions. `solve_u` runs CG to find the displacements `u`. `postprocessing` calculates the element von Mises stresses and outputs them as a `.csv`.

To motivate our parallel algorithm, we started by benchmarking the wall clock times of each phase of the CPU dense algorithm across various mesh sizes of a unit cube. From Fig. 5, we can see that the total runtime increases rapidly with mesh size and past ~ 4000 nodes, K_g was no longer able to be fully stored in memory.

Further breaking down the runtime by phase in Fig. 6, the algorithm becomes dominated by the conjugate gradient step (`solve_u`). With reference to dense matrix representations on GPU, the matrix multiplication kernel took as much as 97.5% of the runtime on the largest possible workload.

TABLE IV: CUDA Kernel Statistics for $N = 2199$.

Time (%)	Total Time (ms)	Kernel Description
97.5	1289.50	Dense matrix-dense vector product
1.4	18.67	General dense-vector addition: $ax + by$
1.0	13.77	Dense-vector dot product
0.0	.40	Compute local stiffness and assemble global stiffness
0.0	.11	Apply contact forces
0.0	.03	Apply gravity

This supports the hypothesis that focusing on matrix storage and multiplication optimizations involved in this step would result in the most significant improvements to performance. We therefore use the throughput (number of CG iterations per second) as a proxy metric for benchmarking SpMV performance.

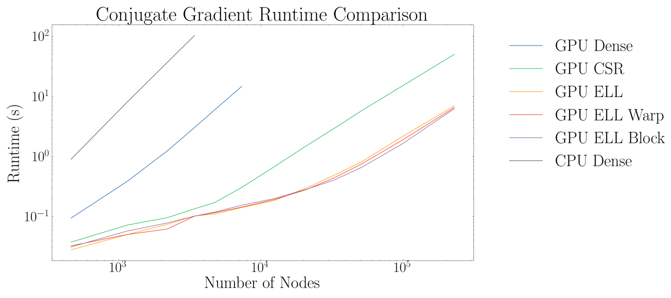


Fig. 7: Log-log plot of runtime against number of nodes for each matrix representation

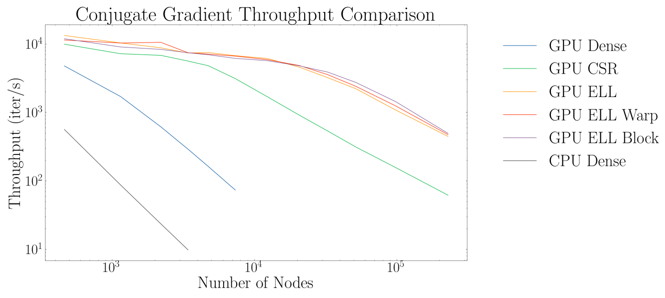


Fig. 8: Log-log plot of throughput (CG iters/sec) against number of nodes for each matrix representation

We then compared the CG runtimes and throughputs of the serial algorithm with each GPU matrix storage mode. Each GPU SpMV kernel was run with a block size of 1024. From Fig. 7, the GPU dense matrix experienced a 10 to 34 times throughput improvement over the serial implementation from a node size of 463 to 3419.

CSR has a 2 to 42 times throughput improvement over GPU dense as the number of nodes increases from 463 to 7348. Incidentally, the amount of speedup is roughly proportional to the mesh size, which is consistent with the improvement from a $O(n^2)$ dense matrix vector multiplication algorithm to a $O(n)$ sparse one.

The throughput of the CSR and ELL implementations remains roughly comparable until a node size of 7348, after which ELL begins to significantly outperform CSR. In the log-log plot of Fig. 8, this is marked by a 'kink' or drop in the gradient of throughput against number of nodes for CSR. A similar falloff in efficiency occurs later at a mesh size of $N = 51572$ for the ELL matrices. We observe that the number of CSR L1 misses rises markedly when moving from the $N = 4759$ to $N = 7348$ meshes, suggesting that at this threshold, K_g and p_i may no longer fit within the L1 cache, resulting in cache thrashing as multiple threads access different locations of K_g and p_i. On the other hand, the ELL matrices have more regular alignments of data. By storing their elements in column major order, there is a higher proportion of coalesced memory accesses and greater spatial locality of between threads that reduces the chances of cache pollution. As such, the limitations of cache thrashing only become apparent at much larger mesh sizes. A more detailed analysis of the memory access patterns will be conducted in the next section.

Between each ELL implementation, ELL-WARP offers a $5-10\%$ speedup over base ELL, and EBW offers a further $5-15\%$ speedup over ELL-WARP at the largest mesh sizes.

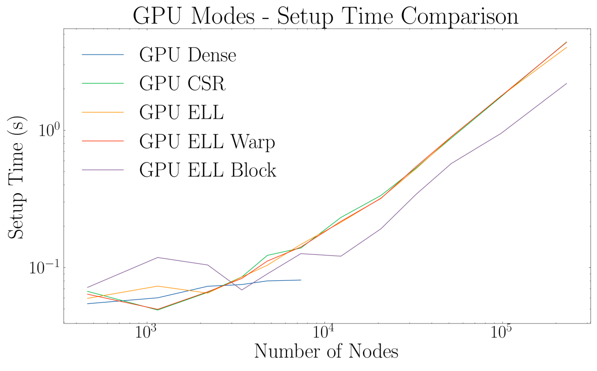


Fig. 9: Setup times of each GPU matrix mode

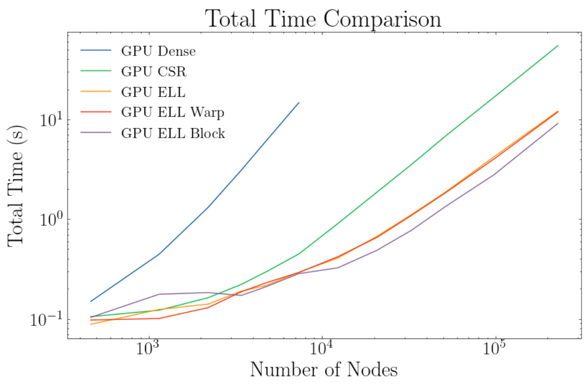


Fig. 10: Total algorithm runtimes of each GPU matrix modes

The EBW groupings of matrix elements into 3×3 blocks of node coordinates also reduces the size of the logical index mappings by 9 times. This is reflected in a noticeable reduction in setup times over the other ELL implementations, further improving runtimes for the overall algorithm.

A. *Analysis of Bottlenecks*

We used the `ncu` tool to profile the bottlenecks of the program.

TABLE V: Throughput of various components

Matrix	N	Compute	L1/TEX	L2	DRAM
GPU-Dense	7.3K	31.37	26.42	43.54	80.00
CSR	7.3K	3.44	32.63	21.33	73.21
ELLPACK	7.3K	11.24	28.11	16.81	43.89
ELLWARP	7.3K	9.63	31.12	16.99	39.89
EBW	7.3K	7.62	71.57	12.70	21.50
CSR	229K	1.78	9.55	20.47	69.72
ELLPACK	229K	15.21	23.07	31.00	74.90
ELLWARP	229K	13.06	22.31	31.50	70.70
EBW	229K	7.31	39.83	32.46	83.57

From Table V we can see that consistently, DRAM throughput is the highest and is hence the bottleneck of the SpMV kernel.

Let us take a closer look at the exact memory access characteristics to understand how the throughput of the kernel is affected. From Table VI we can see that the number of DRAM sectors loaded drastically decreases across our 5 implementations (Dense, CSR, ELLPACK, ELLWARP, EBW), with EBW performing the least loads and stores of them all.

TABLE VI: DRAM access characteristics

Matrix	**DRAM Sectors Loaded ($N = 7.3k$)**	**DRAM Sectors Stored ($N = 7.3k$)**	**DRAM Sectors Stored ($N = 229k$)**	**DRAM Sectors Stored ($N = 229k$)**
GPU-Dense	121.74M	22,539.5K	N/A	N/A
CSR	2.36M	25.9K	145.34M	860K
ELLPACK	.40M	23.6K	17.02M	854K
ELLWARP	.36M	22.5K	16.02M	980K
EBW	.26M	7.6K	14.13M	529K

From Table VII, at large node counts, we can see that CSR has the worst performance of all implementations because it does not exploit any locality or coalesce any memory accesses. It requests an average of 7.0 bytes per thread per memory request, but 20.6 sectors (i.e. 660.7 bytes) is served per request, i.e. over 99% of bytes served by the cache are wasted and unused. Note that it is the only implementation of the four which has a 1-D style memory layout; the SpMV kernel indexes into each array randomly. On the other hand, all the ELL variations store data in at least 2 dimensions and is column-major to facilitate unit-strided access into the data and index matrices.

We note that the L1 hit rate actually sharply decreases from ELLWARP to EBW. This might be explained by the fact that there is a larger working set since each thread has to compute 3 different sums as it is responsible for 3 rows. This working set is still able to fit in the L2 cache, however. The reason for the slight decrease from ELLPACK to ELLWARP might be quite similar as well. Nevertheless, the much lower number of L1 requests demonstrates a much better use of locality, and is able to compensate for this decrease in hit rate.

TABLE VII: L1 access characteristics for $N = 229k$

Matrix	**L1 Sectors Loaded**	**L1 Load Hit Rate**	**L2 Sectors Loaded**	**L2 Load Hit Rate**
CSR	72.97M	1.67%	80.08M	2.09%
ELLPACK	23.21M	9.80%	24.23M	35.12%
ELLWARP	22.90M	8.87%	24.60M	39.55%
EBW	18.64M	1.92%	21.52M	40.39%

Are we able to further optimize our implementation? Examining the source counts for our implementation of EBW, we observe that 27% of warp stalls occur at reading from the vector during SpMV. About 44% of sectors are excessively accessed due to uncoalesced global accesses. Unfortunately, due to the sparsity of the global stiffness matrix and the irregularity of the elements in each row, we doubt that we will be able to further improve upon this performance. Since each row of the matrix has a different, randomly distributed positioning of the nonzero elements, it is dubious whether sorting the array would allow us to alleviate this as a permutation which may work well for one row could be extremely incompatible with many other rows.

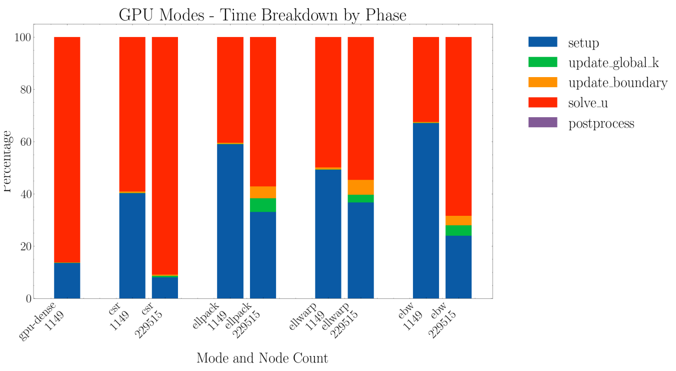


Fig. 11: Runtime breakdown of each GPU matrix mode over small ($N = 1149$) and large ($N = 229515$) mesh sizes.

From Fig. 11, the set-up time also becomes a significant proportion of the runtime even as the number of nodes increase. We did not consider this as a priority to parallelize on CUDA and this could also be explored as another avenue for performance gains.

We believe that our choice of machine target (GPU) was sound as the workload was ultimately memory-bound. GPU memory bandwidths tend to be higher and are more optimized for parallel operations on large datasets, which our graphs might be suited for in general.

IV. Work Distribution

The credit is equally distributed.

Hong Lin
- Final report
- Mesh parsing
- ELL implementations
- Application of boundary conditions
- Benchmarking

Ryan Lau
- Final report
- Physical modelling
- Dense, CSR implementations
- Mesh generation
- Visualization, postprocessing

References

[1] "CIVL 8/7117 Finite Elements in Structural Mechanics Chapter 11 Notes." Accessed: Apr. 29, 2025. [Online]. Available: https://www.ce.memphis.edu/7117/notes/presentations/chapter_11.pdf

[2] P. I. Kattan and P. I. Kattan, "The linear tetrahedral (solid) element," *MATLAB Guide to Finite Elements: An Interactive Approach*, pp. 329–357, 2003.

[3] "Boundary Conditions — FEM Tutorial 0.1.0 documentation." Accessed: Apr. 29, 2025. [Online]. Available: https://www8central.ww.uni-erlangen.de/courses/fem101/fem/boundary_conditions.html#dirichlet-boundary-conditions

[4] M. Okereke, S. Keates, M. Okereke, and S. Keates, "Direct stiffness method," *Finite Element Applications: A Practical Guide to the FEM Process*, pp. 47–106, 2018.

[5] J. Wong, E. Kuhl, and E. Darve, "A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems," *International Journal for Numerical Methods in Engineering*, vol. 102, no. 12, pp. 1784–1814, 2015.