TRA105 - GPU-Accelerated Computational Methods

GPU-Accelerated FEM Solver

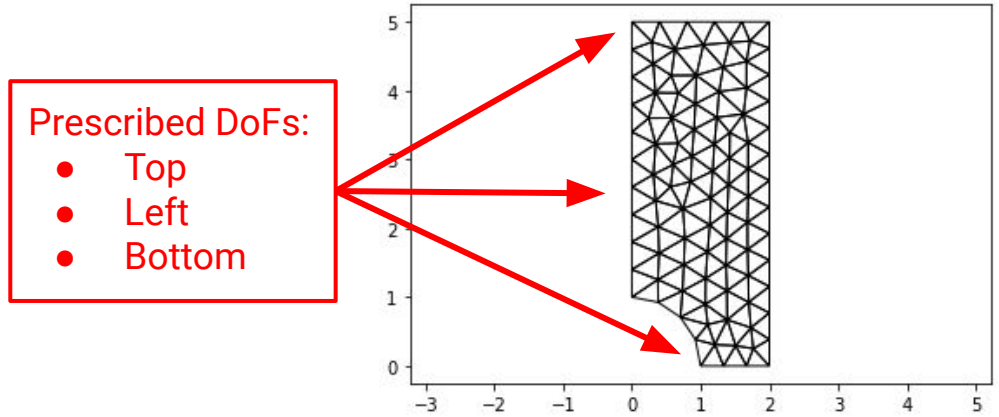
Stefano Ribes

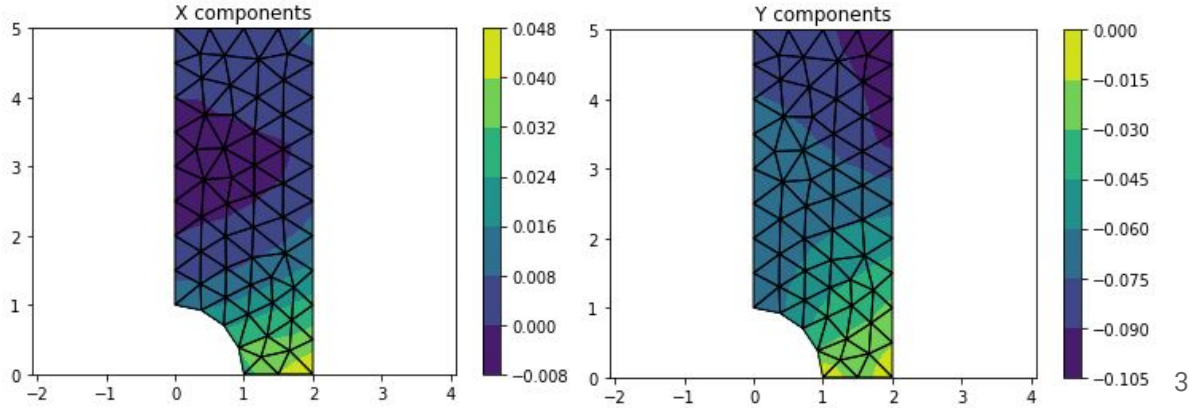
Overview

- FEM and Problem Formulation
- Problem Modeling on GPU
- K-Assembly Implementation
- Evaluation and Results
- Conclusions

Problem Description

- FEM Problem:
  - f = K @ a
- Where:
  - K: global stiffness matrix
  - a: nodal displacements
  - f: nodal forces
- Selected problem:
  - N. Dimensions = 2
  - DoFs per node = 2
  - Nodes per cell = 3
  - The mesh **characteristic length** can be adjusted





K-Assembly: Element Routine on GPU

- The element routine has been written in Numba as a CUDA device function
- The element routine is executed by each thread
- Each thread utilizes around 105 bytes
- The ke and re components are updated (and accumulated) on the host side

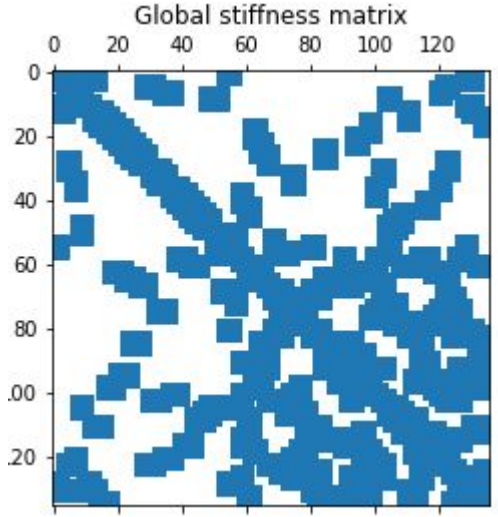
```
@cuda.jit(device=True, inline=True)
def element_routine(thickness, xe, ue, weights, stiffness, re, ke):
    # Thread local arrays. NOTE: re and ke are on the host side
    B = nb.cuda.local.array((n_nodes_per_cell, n_dofs_per_cell), float32)
    epsilon = nb.cuda.local.array((n_nodes_per_cell), float32)
    sigma = nb.cuda.local.array((n_nodes_per_cell), float32)
    BTsigma = nb.cuda.local.array(n_dofs_per_cell, float32)
    BTdsde = nb.cuda.local.array((n_dofs_per_cell, n_dofs_per_cell), float32)
    BTdsdeB = nb.cuda.local.array((n_dofs_per_cell, n_dofs_per_cell), float32)
    # Thread Computation
    detJ = jacobi_det(xe)
    B_operator(xe, B)
    nqp = weights.shape[0]
    for qp in range(nqp):
        w = weights[qp]
        matvmul(B, ue, epsilon)
        matvmul(stiffness, epsilon, sigma)
        matvmul(B.T, sigma, BTsigma)
        matmul(B.T, stiffness, BTdsde)
        matmul(BTdsde, B, BTdsdeB)
        detJw = detJ * w
        macv(BTsigma, detJw, re)
        macv(BTdsdeB, detJw, ke)
    mulv(re, thickness)
    mulv(ke, thickness)
```

K-Assembly: CUDA Kernel

```
@cuda.jit
def elem_routine_kernel(thickness, stiffness_host, weights_host, xe_host, ue_host, re_host, ke_host):
    # Thread ID in a 1D block
    tx = cuda.threadIdx.x
    # Block ID in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    # Get grid size and check if within grid
    n_cells = xe_host.shape[0]
    if pos < n_cells:
        element_routine(thickness,xe_host[pos], ue_host[pos], weights_host, stiffness_host, re_host[pos],
                        ke_host[pos])
```

Efficient Representation of the K Matrix

- For the chosen problem, K is **always sparse and symmetric**
- Two efficient formats for building it:
  - DoK (Dictionary of Keys)
    - A Python dict with nnz indexes as keys
    - O(1) access to elements
    - **Duplicate (i,j) entries are replaced if not handled properly**
    - Not ideal for slicing
  - COO (COOrdinate format)
    - Keep three lists/arrays: data, x-indexes, y-indexes
    - “Conflicting” duplicate (i,j) entries are allowed
    - **When the matrix is needed, duplicate (i,j) entries are summed together**
    - Fast conversion to CSR and CSS formats



Each Kernel call will generate a set of ke and re components, then the COO format in CuPy will accumulate the duplicate entries

K-Assembly: Calling the Kernel

1. Define a batch size
2. Define GPU buffers
3. Define K matrix as COO format
4. Foreach batch in n_cells/n_batches:
    a. Fill GPU input buffers
    b. Run element routine kernel
    c. Transfer ke and re from GPU to host
    d. Update COO components
5. Convert COO components to the final K matrix

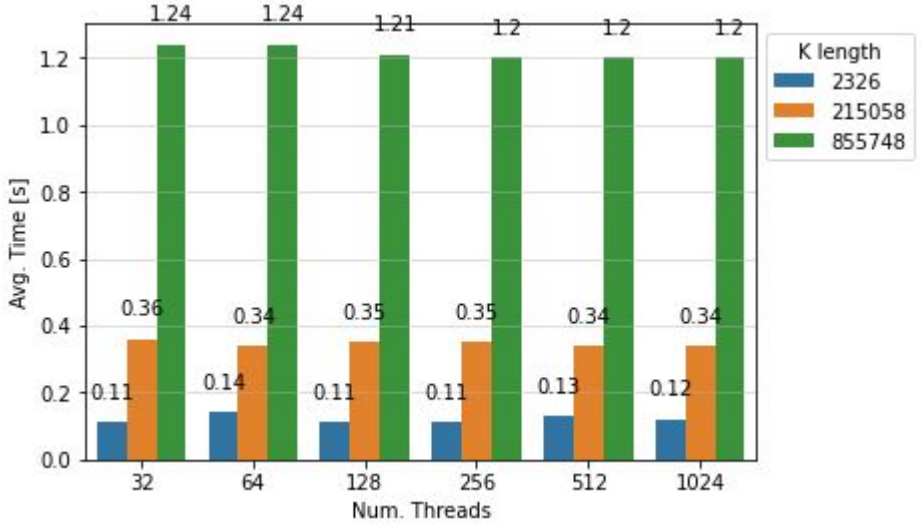
```
def naive_k_assembly_gpu(a, weak_form, dh, batch_sz=-1, threadsperblock=64):
    n_cells = dh.grid.get_num_cells()
    n_dims = dh.grid.n_dim
    n_dofs_cell = dh.ndofs_per_cell(dh.grid)
    n_nodes_cell = dh.grid.nnodes_per_cell()
    # Init local variables for element loop (re-use them across iterations)
    dofs = np.empty((batch_sz, n_dofs_cell), dtype=np.int32)
    xe = np.empty((batch_sz, n_nodes_cell, n_dims), dtype=np.float32)
    ue = np.empty((batch_sz, n_dofs_cell), dtype=np.float32)
    ke = np.zeros((batch_sz, n_dofs_cell, n_dofs_cell), dtype=np.float32)
    re = np.zeros((batch_sz, n_dofs_cell), dtype=np.float32)
    # Specify CUDA kernel dimensions
    blockspergrid = (batch_sz + (threadsperblock - 1)) // threadsperblock
    # Move "constant" data to device (i.e. not depending on cell ID)
    stiffness_d = cuda.to_device(weak_form.material.stiffness)
    weights_d = cuda.to_device(weak_form.element.weights)
    # Run over batches
    # NOTE: Each "batch" will become a "grid" from the kernel perspective.
    for batchptr in range(0, n_cells, batch_sz):
        actual_batch_sz = min(batch_sz, n_cells - batchptr)
        # Setup kernel inputs
        dofs.fill(0)
        xe.fill(0)
        ue.fill(0)
        ke.fill(0)
        re.fill(0)
        xe = dh.grid.nodes[dh.grid.cells[batchptr:batchptr+actual_batch_sz]]
        get_dofs(dh.grid.cells[batchptr:batchptr+actual_batch_sz], a, dofs, ue)
        xe_d = cuda.to_device(xe)
        ue_d = cuda.to_device(ue)
        re_d = cuda.to_device(re)
        ke_d = cuda.to_device(ke)
        # Run kernel
        elem_routine_kernel[blockspergrid, threadsperblock](weak_form.thickness,
            stiffness_d, weights_d, xe_d, ue_d, re_d, ke_d)
        K_idx, f_idx = get_coo_indices(dofs)
        if batchptr == 0:
            K_data = ke_d.copy_to_host().flatten()
            f_data = ke_d.copy_to_host().flatten()
            K_rows, K_cols = K_idx
            f_rows, f_cols = f_idx
        else:
            K_rows = np.concatenate((K_rows, K_idx[ 0])).flatten()
            K_cols = np.concatenate((K_cols, K_idx[ 1])).flatten()
            f_rows = np.concatenate((f_rows, f_idx[ 0])).flatten()
            f_cols = np.concatenate((f_cols, f_idx[ 1])).flatten()
            # Retrieve data from GPU and update K and f COO components
            K_data = np.concatenate((K_data, ke_d.copy_to_host().flatten()))
            f_data = np.concatenate((f_data, ke_d.copy_to_host().flatten()))
    # Convert lists to COO matrices
    K_data = np.array(K_data, dtype=np.float32).flatten()
    K_rows = np.array(K_rows, dtype=np.int32).flatten()
    K_cols = np.array(K_cols, dtype=np.int32).flatten()
    K = coo_matrix((K_data, (K_rows, K_cols)), dtype=np.float32)
    f_data = np.array(f_data, dtype=np.float32).flatten()
    f_rows = np.array(f_rows, dtype=np.int32).flatten()
    f_cols = np.array(f_cols, dtype=np.int32).flatten()
    f = coo_matrix((f_data, (f_rows, f_cols)), dtype=np.float32).toarray()
    return K, f
```

Full Algorithm

1. Define a batch size
2. Define GPU buffers
3. Define K matrix as COO format
4. Foreach batch in n_cells/n_batches:
    a. Fill GPU input buffers
    b. Run element routine kernel
    c. Transfer ke and re from GPU to host
    d. Update COO components
5. Convert COO components to the final K matrix in CuPy (interoperability b/w Numba and CuPy)
6. Slice K according to the prescribed and free DoFs (convert COO to to CSS for column-slicing and then to CSR for row-slicing)
7. Use CuPy solvers to find nodal forces

Evaluation

- Nvidia Tesla T4 GPU
- GPU Memory: 15 GB
- MaxGridDimX: 2,147,483,647
- MaxBlockDimX: 1024
- MaxSharedMemoryPerBlock: 49,152 B
- MaxRegistersPerBlock: 65,536 B
- WarpSize: 32
- MaxThreadsPerBlock: 1024



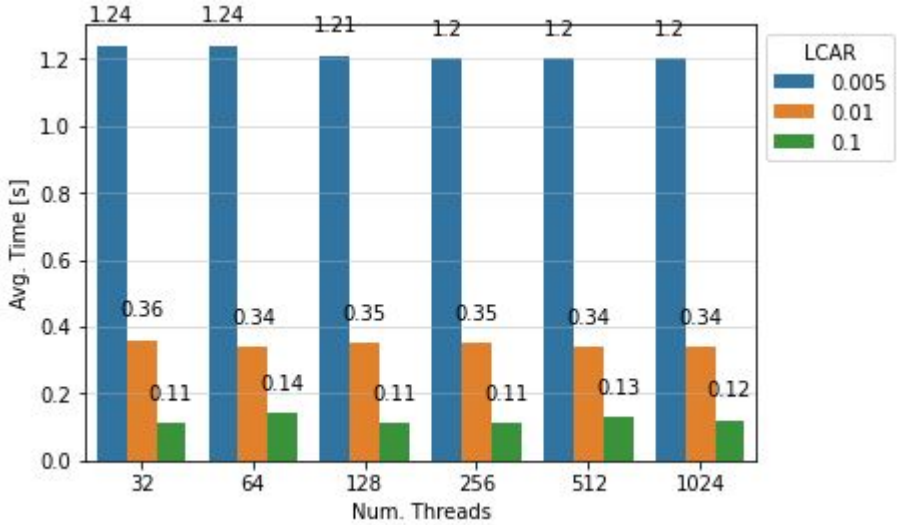
Conclusions

- Numba and Cupy combined use is very efficient
- Cell coloring strategies to be evaluated
- More in-depth design space exploration to be performed (will be in the report)

Backups

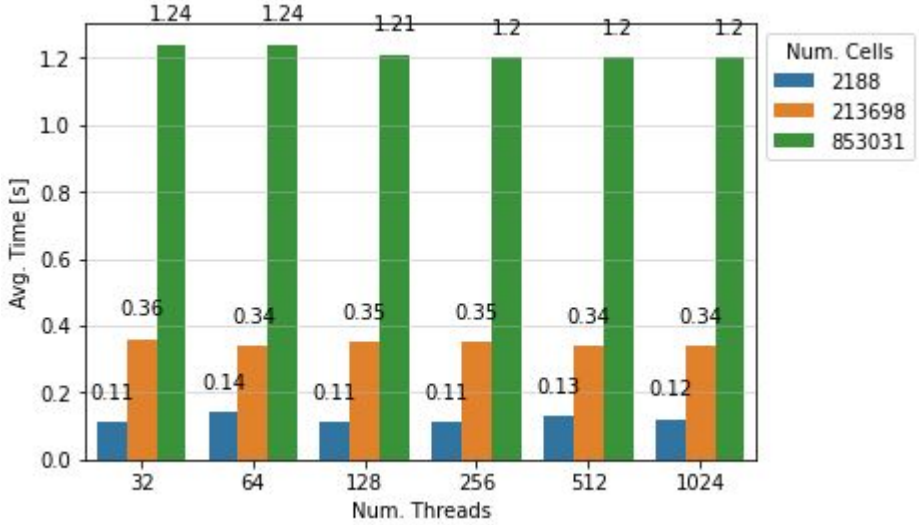
Evaluation: Avg. Time vs. LCAR

- Nvidia Tesla T4 GPU
- GPU Memory: 15 GB
- MaxGridDimX: 2,147,483,647
- MaxBlockDimX: 1024
- MaxSharedMemoryPerBlock: 49,152 B
- MaxRegistersPerBlock: 65,536 B
- WarpSize: 32
- MaxThreadsPerBlock: 1024



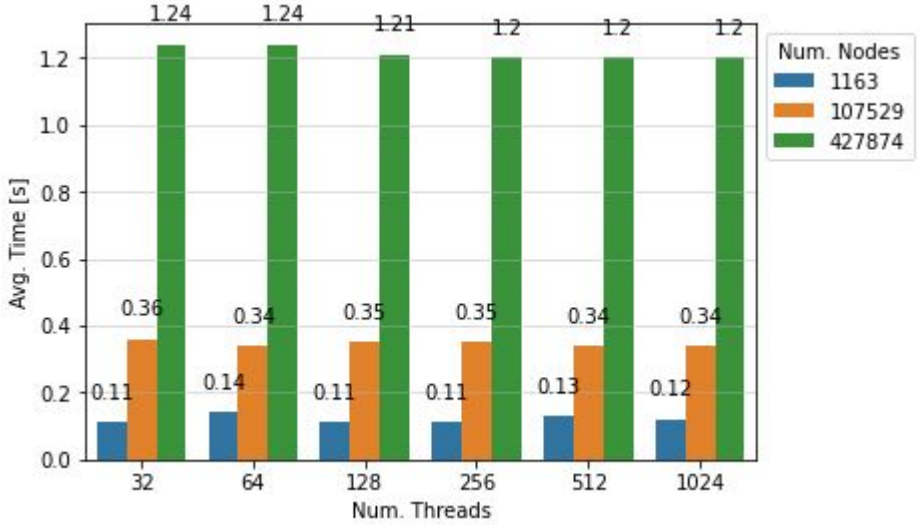
Evaluation: Avg. Time vs. Num. Cells

- Nvidia Tesla T4 GPU
- GPU Memory: 15 GB
- MaxGridDimX: 2,147,483,647
- MaxBlockDimX: 1024
- MaxSharedMemoryPerBlock: 49,152 B
- MaxRegistersPerBlock: 65,536 B
- WarpSize: 32
- MaxThreadsPerBlock: 1024



Evaluation: Avg. Time vs. Num Nodes

- Nvidia Tesla T4 GPU
- GPU Memory: 15 GB
- MaxGridDimX: 2,147,483,647
- MaxBlockDimX: 1024
- MaxSharedMemoryPerBlock: 49,152 B
- MaxRegistersPerBlock: 65,536 B
- WarpSize: 32
- MaxThreadsPerBlock: 1024



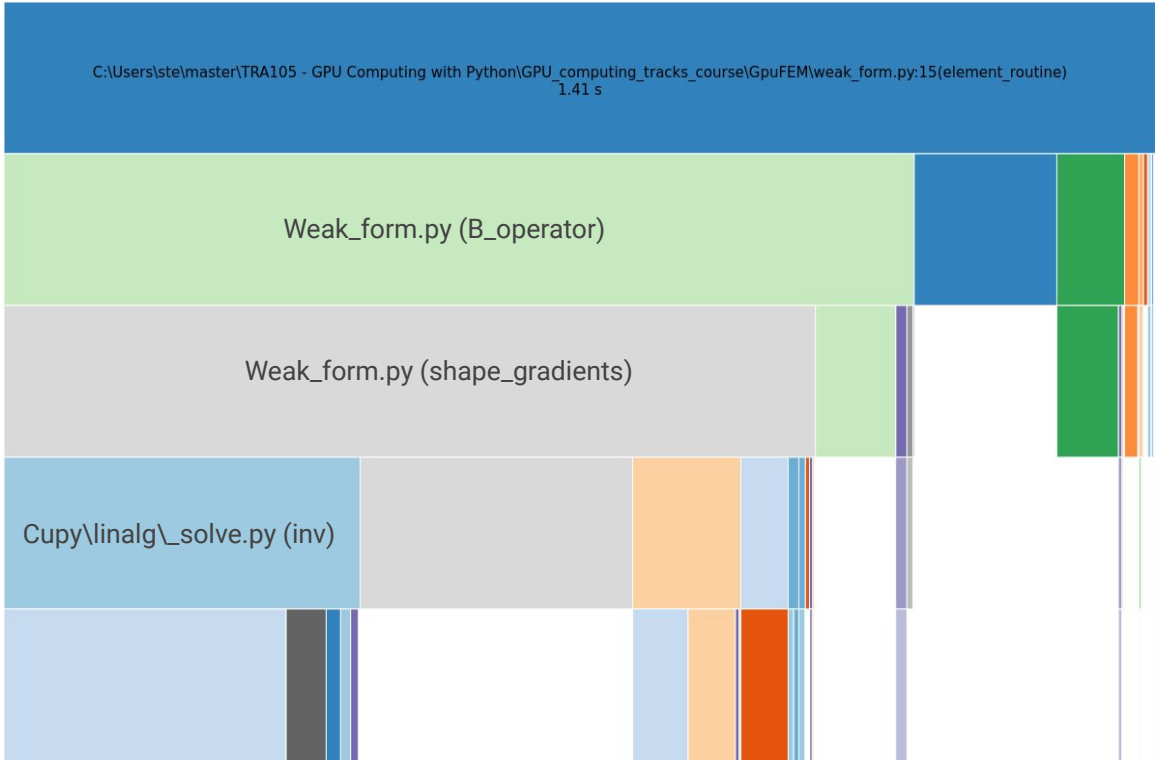
K-Assembly Profiling

- Profiling
- Possible GPU implementations

K-Assembly Profiling on GPU (CuPy)

- The indexing of the sparse matrix K is dominating the computation
- Because of that, we profiled on a small densely-allocated K matrix
- Inverse operation alone occupies around 30% of the cumulative execution time:

Cumtime	%	Filename(Function)
1.415	100%	Weak_form.py (element_routine)
1.118	79%	Weak_form.py (B_operator)
0.9967	70%	Weak_form.py (shape_gradients)
0.4376	31%	Cupy\linalg_solve.py (inv)



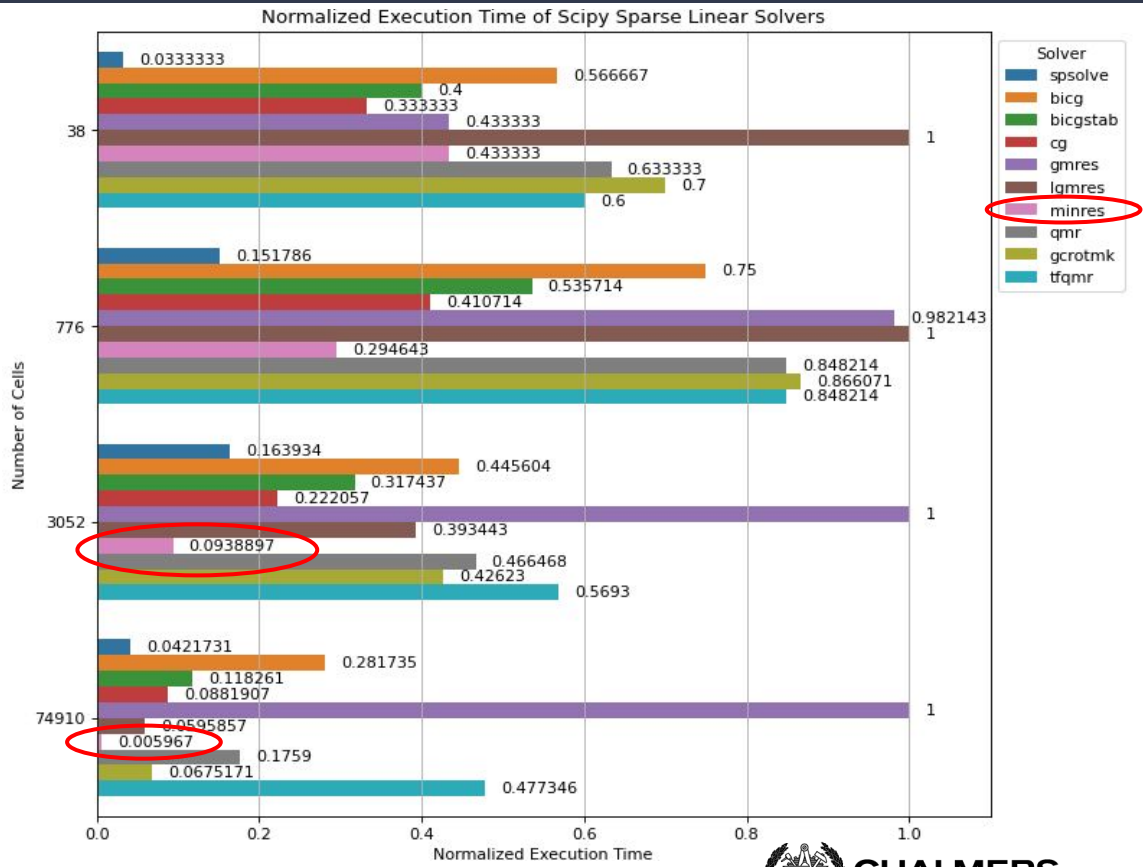
Linear Solvers Profiling

- CPU Profiling
- GPU Profiling

CPU Sparse Linear Solvers Profiling (SciPy)

- Profiled on an Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
- 8 cores without hyperthreading
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 12288K

- Varying Grid mesh granularity (resulting different number of cells)
- Exec. time averaged over 20 runs
- The lower the better
- spsolve and minres achieved best performance

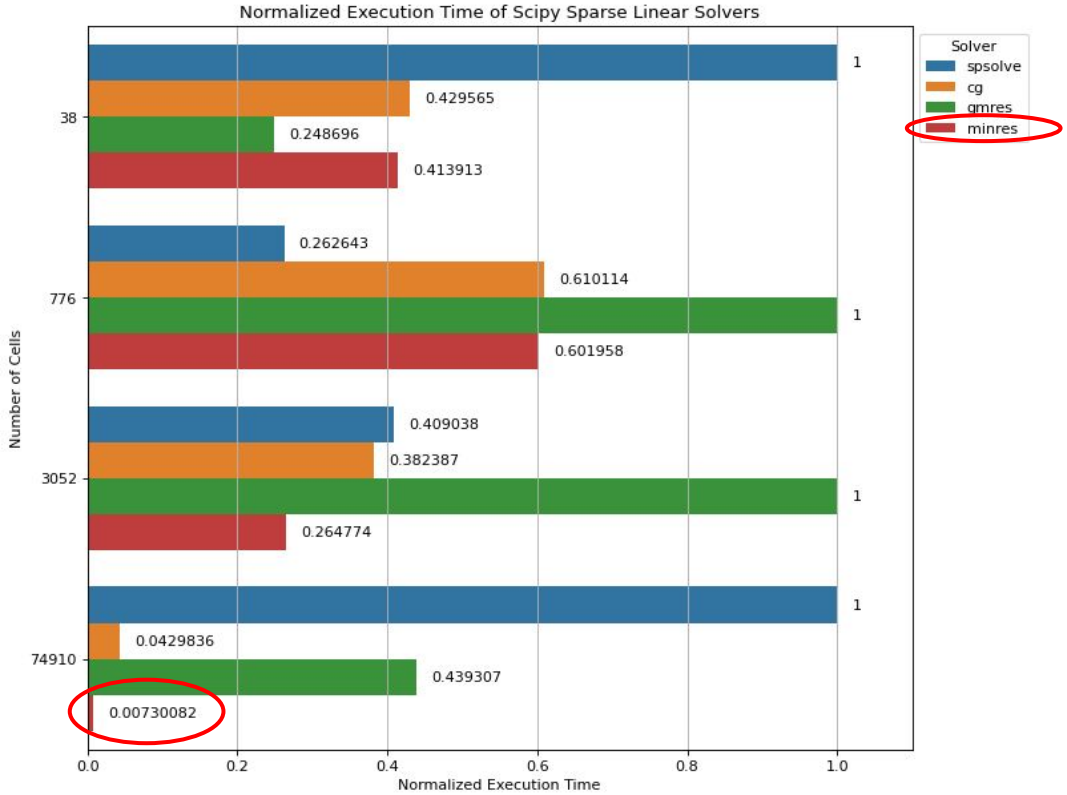


GPU Sparse Linear Solvers Profiling (CuPy)

- NVIDIA GeForce RTX 3060
- 12GB of GDDR6 memory
- 3,584 CUDA cores
- CUDA Version 11.2

- Varying Grid mesh granularity (resulting different number of cells)
- Exec. time averaged over 20 runs
- The lower the better
- spsolve and minres achieved best performance

Fine tuning the linear solver parameters??



Planned Next Steps

- Efficient Store and Indexing of K Matrix
- CuPy Implementation
- Custom GPU Kernels

Planned Next Steps

- Efficient Store and Indexing of the K Matrix
  - Linked-list Format (ll_mat)
  - Compressed Sparse Row Format (csr)
  - Sparse Skyline Format
- Batched CuPy Implementation
- Custom GPU Kernels
  - Writing custom fine-grain kernels to handle independent operations w/in the K-assembly part of the algorithm