Leveraging DOLFINx data-oriented design for GPU implementation

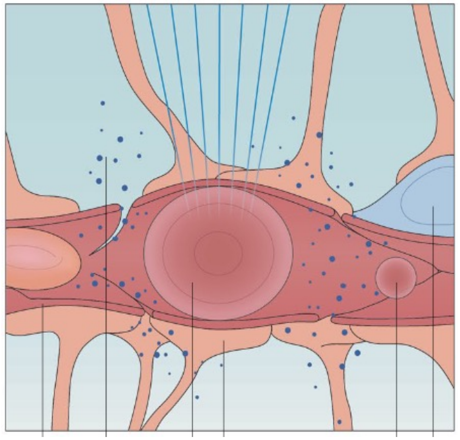
Adeeb Arif Kor[*†], Igor Baratta[^], Garth Wells[*]

[*]University of Cambridge

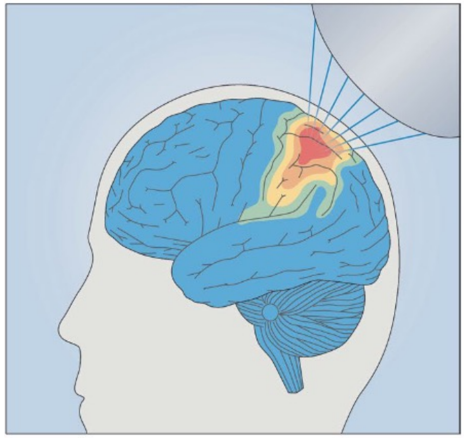
[†]University of Malaya

[^]NVIDIA

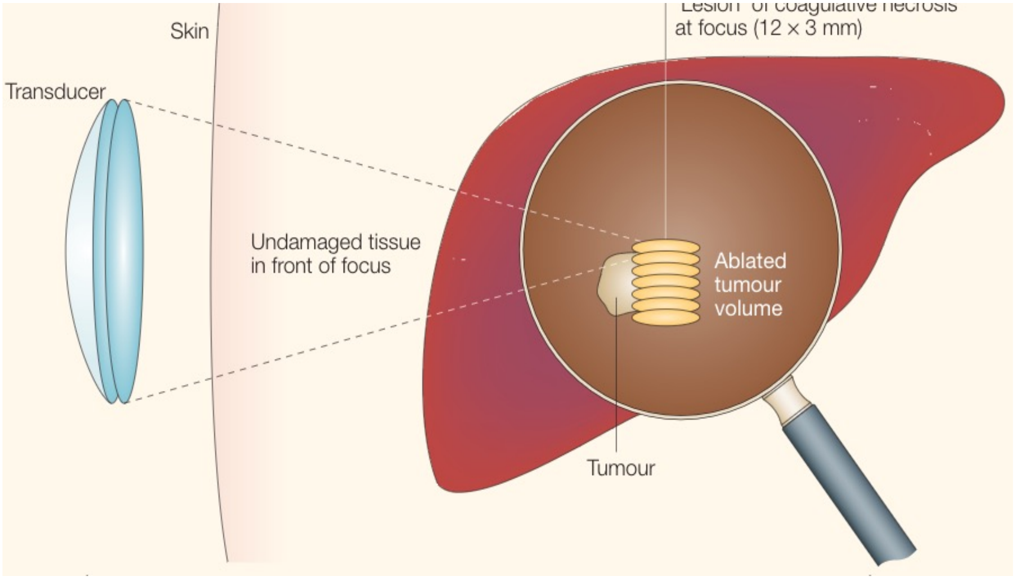
Motivation



†Drug delivery



†Neuromodulation



⁎Thermoablation

Generates and focuses acoustic waves on the targeted region

- †Meng et. al. (2020)
- ⁎Kennedy (2005)

Model equation

$$\frac{1}{\rho_0 c_0^2}\frac{\partial^2 p}{\partial t^2} - \frac{1}{\rho_0}\nabla^2 p = \frac{\delta}{\rho_0 c_0^2}\nabla^2\frac{\partial p}{\partial t} + \frac{\beta}{\rho_0^2 c_0^4}\frac{\partial^2 p^2}{\partial t^2} \quad \text{in } \Omega \times (0,T)$$

$$\nabla p \cdot \mathbf{n} + \frac{1}{c_0} p_t = g(t) \quad \text{on } \Gamma_s \times (0,T)$$

$$\nabla p \cdot \mathbf{n} + \frac{1}{c_0} p_t = 0 \quad \text{on } \Gamma \times (0,T)$$

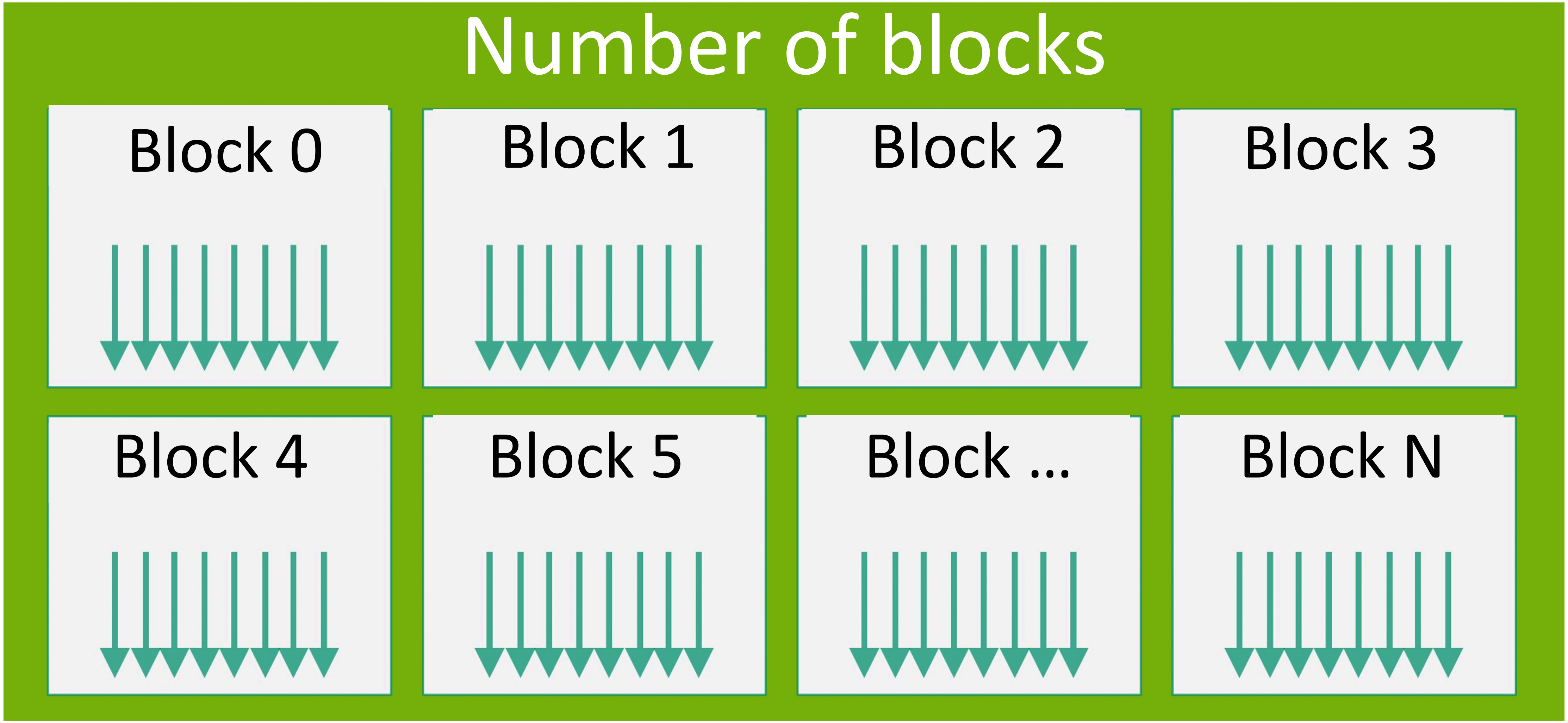
$$p(\boldsymbol{x}, 0) = 0 \quad \text{for } \boldsymbol{x} \in \Omega$$

$$p_t(\boldsymbol{x}, 0) = 0 \quad \text{for } \boldsymbol{x} \in \Omega$$

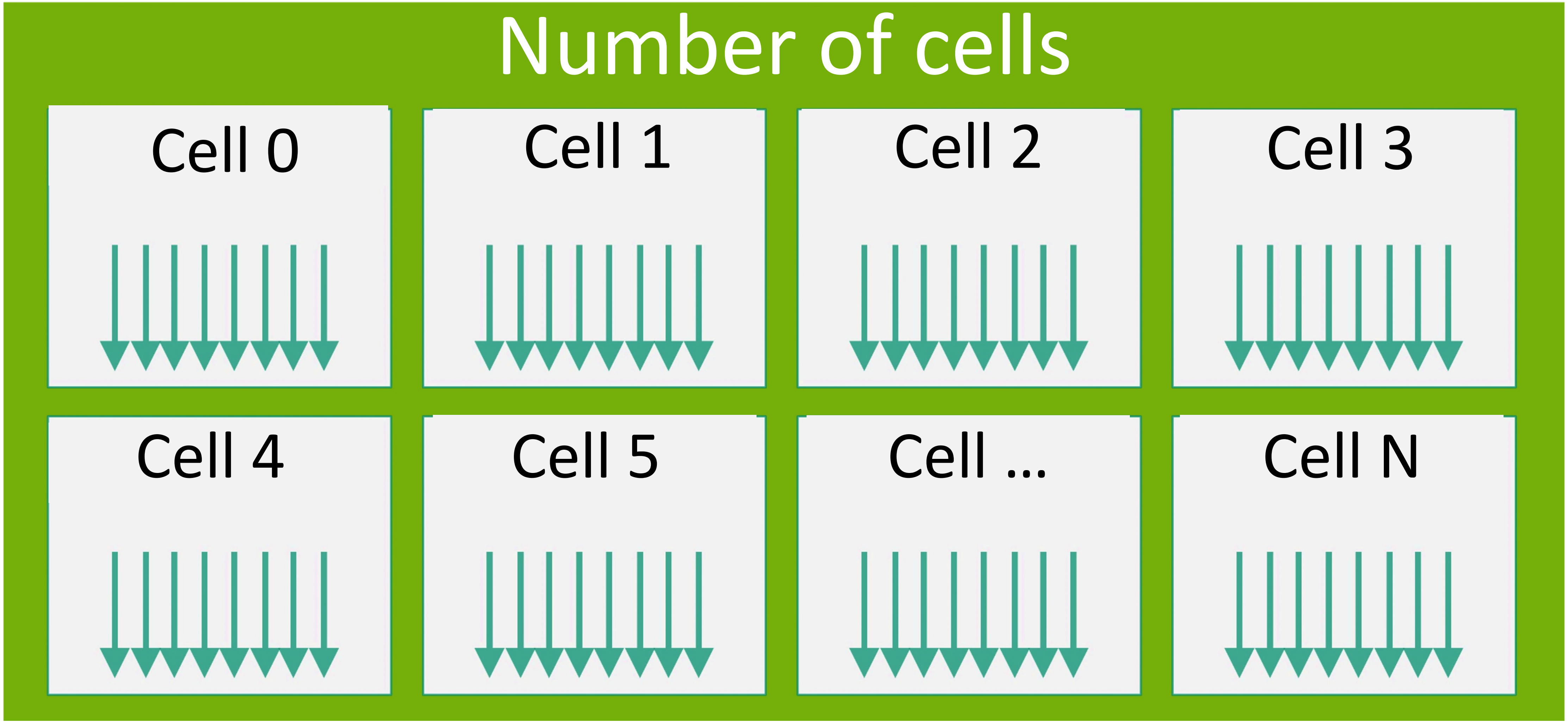
Solver design

- Fully hexahedral mesh
- Arbitrary high-order GLL-based Lagrange finite element basis function
- Numerical quadrature is performed using GLL quadrature
- Mass lumped scheme
- 4^{th} order explicit Runge-Kutta scheme

Idea


Number of blocks
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block ...
Block N

Idea


Number of cells
Cell 0
Cell 1
Cell 2
Cell 3
Cell 4
Cell 5
Cell ...
Cell N

CPU Implementation

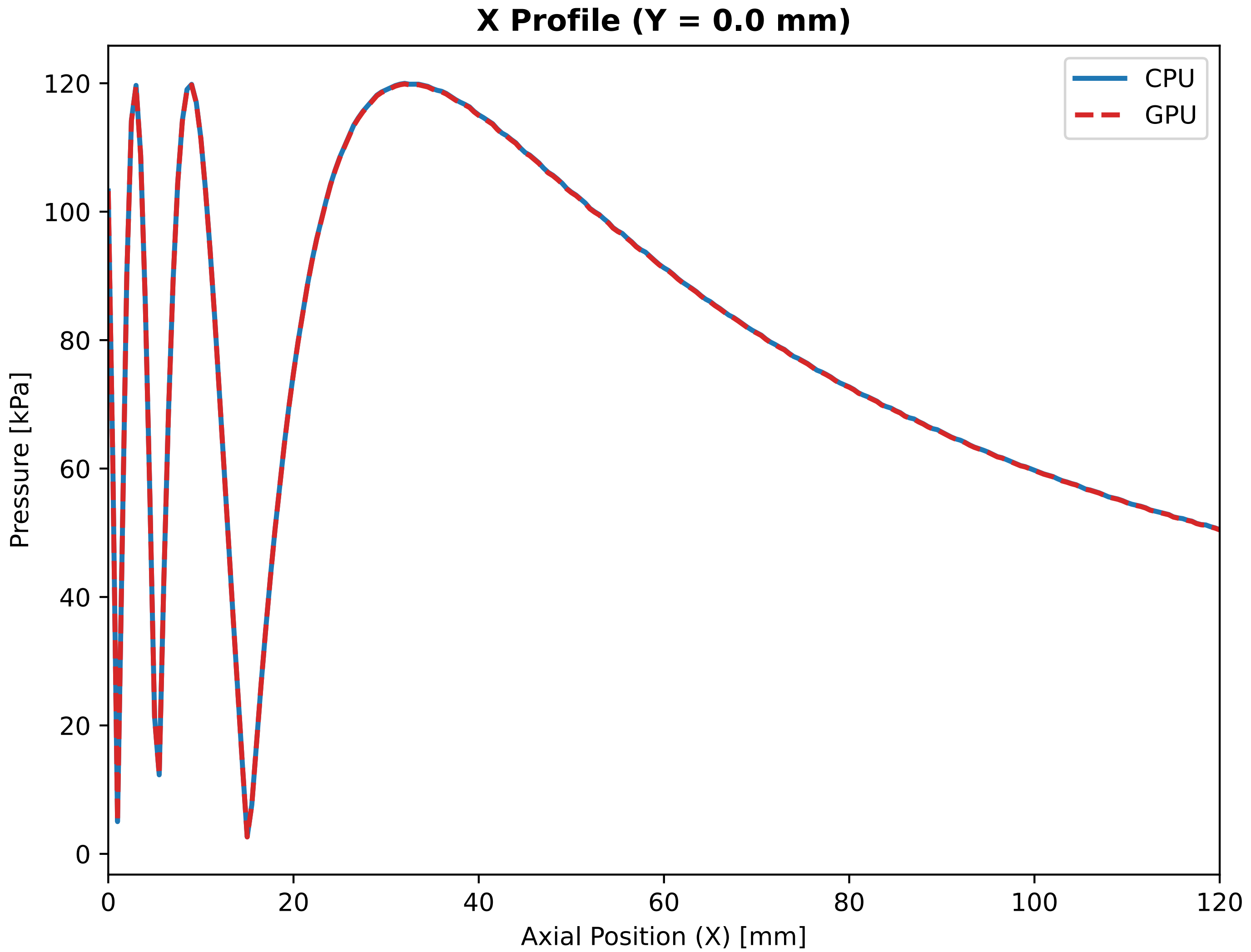
```
num_entities = entity_constants.size

# Initialise temporaries
x_ = np.zeros(N, float_type)

for entity in range(num_entities):
    # Pack coefficients
    for i in range(N):
        x_[i] = x[entity_dofmap[entity][i]]

    # Apply transform
    for i in range(N):
    x_[i] *= entity_detJ[entity][i] * entity_constants[entity]

    # Add contributions
    for i in range(N):
        y[entity_dofmap[entity][i]] += x_[i]
```

GPU Implementation

```
thread_id = cuda.threadIdx.x # Local thread ID (max: 1024)
block_id = cuda.blockIdx.x # Block ID (max: 2147483647)
idx = thread_id + block_id * cuda.blockDim.x # Global thread ID

entity = idx // entity_dofmap.shape[1]
local_dof = idx % entity_dofmap.shape[1]

if idx < entity_dofmap.size:
    # Compute the global DOF index
    dof = entity_dofmap[entity, local_dof]

    # Compute the contribution of the current DOF to the mass operator
    value = x[dof] * detJ_entity[entity, local_dof] * entity_constants[entity]

    # Atomically add the computed value to the output array `y`
    cuda.atomic.add(y, dof, value)
```

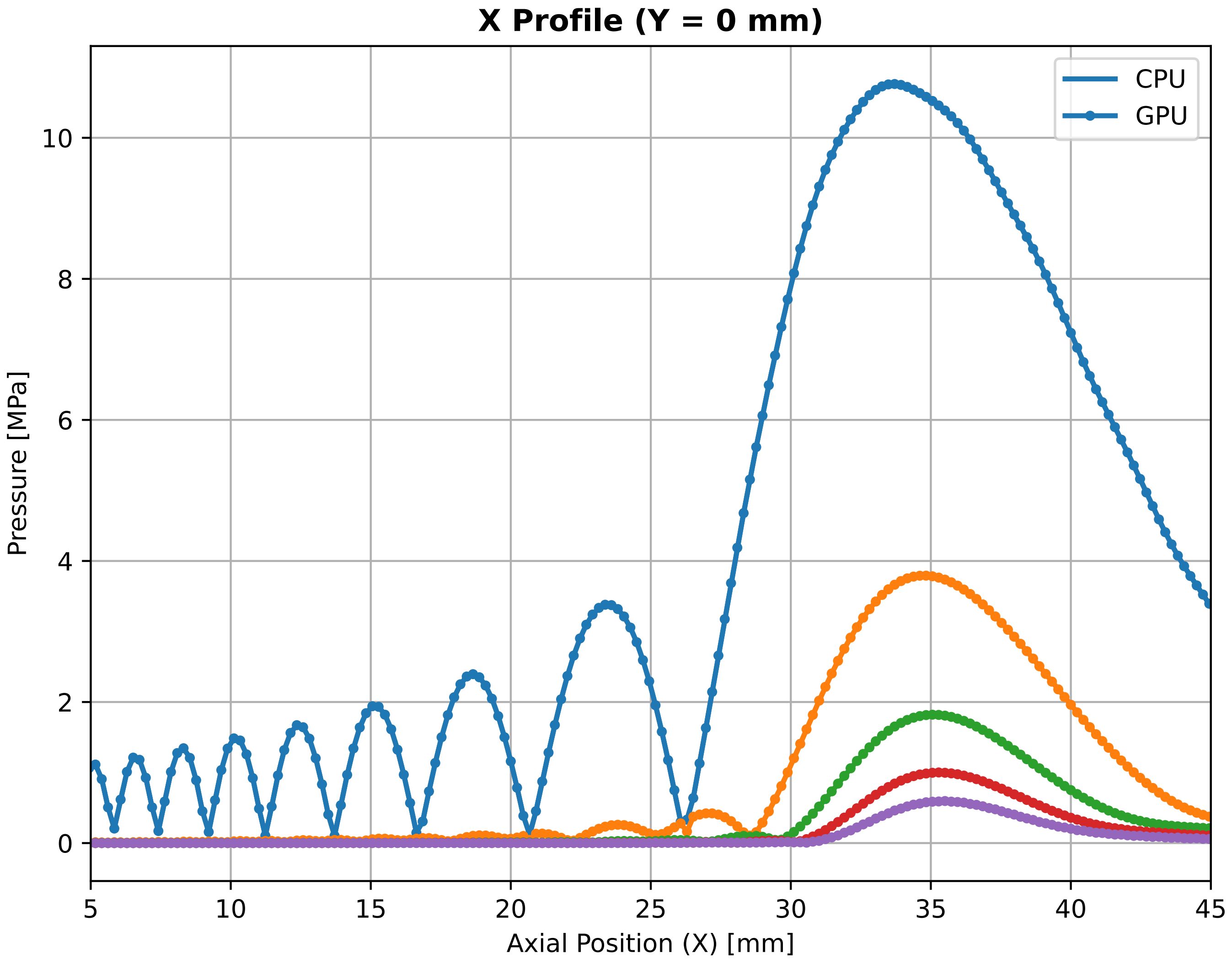
Performance

Validation – Linear



- Transcranial focused ultrasound problem.
- Homogeneous medium – water
- 31 x 10^6 degrees-of-freedom
- 6 minutes using NVIDIA RTX A1000 with double precision
- 7 minutes using 62 Intel Icelake CPUs with double precision

Validation – Nonlinear



- High-intensity focused ultrasound
- H131 focused bowl transducer
- 100W
- Homogenous medium – Water
- 430 x 10^6 degrees-of-freedom
- 5 hours using 2 NVIDIA A100 with double precision (single DGX A100)
- 3 hours using 448 Intel Skylake CPUs with single precision (14 nodes)

Acknowledgement

The travel award for this presentation was provided by NumFOCUS.