# MULTIGRID PRECONDITIONED CONJUGATE GRADIENT SOLVER FOR POISSON EQUATION

**Sangeetha Grama Srinivasan**
Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI, 53726
sgsrinivasa2@wisc.edu

May 12, 2022

**ABSTRACT**

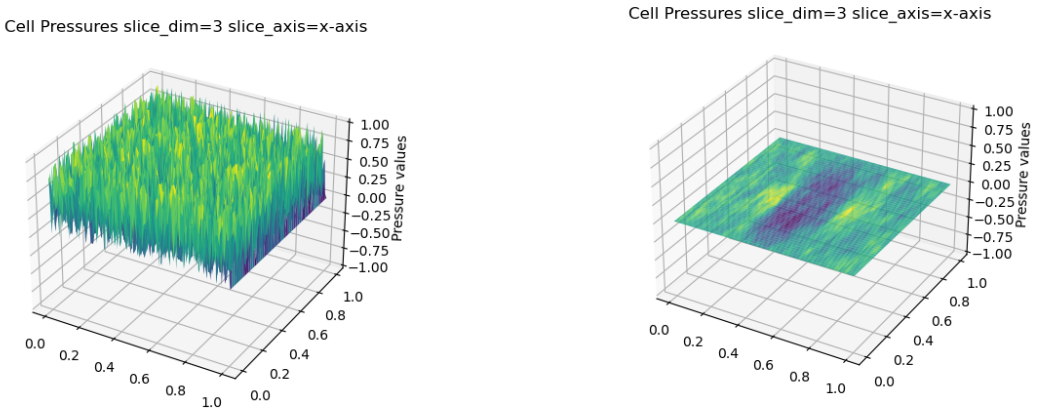
This course project aims to implement a Multigrid preconditioned Conjugate Gradients Solver for the poisson equation and use it for large scale fluid simulations. The design of a robust, parallel and scalable poisson solver for fluid simulation has been proposed in [1]. This work uses a geometric Multigrid V-cycle as a preconditioner to the Conjugate Gradients solver. The paper also discretizes pressure over the domain using Finite Differences. This Multigrid-preconditioned Conjugate Gradients (MGPCG) solver drastically improves convergence and robustness for a poisson solver. The authors of [1] design and implement the MGPCG solver on shared-memory multiprocessors. and focus on scaling up to high resolutions on the CPU. This project aims to implement this idea but using Finite Element instead of Finite Differences. It would be interesting to implement the MGPCG solver for a finite element discretization.

***K*eywords** Finite Element Method · Multigrid method · Conjugate Gradients · Preconditioning · Large scale fluid simulation

## 1 Introduction

Fluid simulations are the bread and butter of any Computer Graphics or Physics simulation pipeline. Many interactive simulations also require fast fluid simulations - like interactions of smoke with solids or fluid-solid interactions. Traditionally, these types of simulations are implemented using a cartesian lattice where the velocity of the simulation particles is stored in a staggered manner (MAC discretization) on cell faces and pressures are stored in cell centres. While this approach guarantees convergence, it is slow and does not scale to high resolutions at interactive speeds. The work in [1] addresses this issue by simplifying the fluid simulation problem to a poisson problem ( described in [2] as *voxelised poisson-equation*). While incomplete Cholesky preconditioned conjugate gradients (ICPCG) was shown to accelarate the solver for this equation [3], the work in [1] proposes using a V-cycle of Multigrid as the preconditioner for Conjugate gradients. This MGPCG is shown to be scalable to high resolution grids on shared-memory multiprocessors and perform better than the incomplete Cholesky preconditioner, while also accommodating irregular domains with both dirichlet and Neumann boundary conditions. The work in [1] however uses Finite Differences to discretize the scalar physical quantity on the domain. This project aims to implement the work in [1] with a slight modification - the scalar quantity is discretized and solved for using Finite Element Method instead of Finite Differences. This project also uses a deep-learning library known as pyTorch to implement this solver. The rationale behind using Torch to implement a numerical solver is to allow easy integration of numerical solvers with deep learning pipelines. (However, this project only aims to implement a correct solver and does not concern itself with the deep learning applications, for instance, training a neural network to solve poisson equation). This project has the following goals:

1. Construct a Finite Element discretization of a scalar quantity on a 3D domain with an aim to solve the poisson equation.




(a) Adding noise to the pressure of interior nodes in the domain.

(b) Using Conjugate Gradients to solve for pressures (with the right hand side of the voxelised poisson equation initialized to 0)

Figure 1: A slice of the 3D domain of dimensions 100x200x50 voxels after perturbing the interior node pressures (left) and after solving for pressures using conjugate gradients. (right).

2. Implement a Conjugate Gradients as a solver for this poisson system.
3. Implement and use a V-cycle of Multigrid as a preconditioner for the Conjugate Gradients Solver.
4. Analyse the performance gains or losses of using a Multigrid preconditioned Conjugate Gradients solver over a Conjugate Gradients solver.

The report is structured as follows: Section 1.1 outlines the related literature in this area, followed by the implementation details in Section 2. Section 3 highlights the results from the solver and analyses them. Section 4 highlights the shortcomings of the current design and proposed future directions of improvement. Section 5 visualizes snapshots of the pressures during the multigrid preconditioned solve.

### 1.1 Related Work

The MGPCG work proposed in [1] is shown to be faster than the ICPCG proposed in [3] and scalable to high resolutions on CPUs. MGPCG has been referenced in many subsequent research works, that either try to use Machine learning to learn properties of fluid simulations through data-driven approaches [4] or learning and predicting temporal evolution of fluid simulations as proposed in [5]. The work in [1] also inspired the need for more efficient sparse data structures, such as SPGrid [6] to handle irregular domains used for fluids (especially smoke simulations). Further, [7] proposes another approach to accelerate fluid simulations using Convolutional Neural Networks and claims that the MGPCG approach is hard to implement, difficult to parallelize on the GPU and will not work for domains with highly irregular boundaries. While this project does not address the highly irregular domain issue, it aims to examine the 2 things. First, using finite element method instead of finite differences and second, analyse the performance of a MGPCG solver that uses finite element discretization. Implementing MGPCG on the GPU using learning frameworks like Torch also allows for easy integration with data driven learning approaches for fluid simulations.

## 2 Implementation

In [1], the MGPCG solver is designed to solve the Poisson boundary value problem. The continuous equation that we wish to solve is given as:

$$\Delta p = f \; in \; \Omega \; \subset \; \mathbf{R}^3$$
$$p(x) \; = \; \alpha(x) \; on \; \Gamma_D$$

where the boundary of the domain $\partial\Omega = \Gamma_D$ is where Dirichlet conditions are imposed. [1] stores the pressure (scalar physical quantity) at cell centres of a uniform cartesian lattice. In this project, since we want to use the Finite Element Method, we store the pressures on nodes of each cell. So, for each node $N_{ijk}$ in the domain, there is a corresponding scalar physical quantity $p_{ijk}$. The current implementation only includes Dirichlet boundaries (discussion on including

Neumann boundary conditions is detailed in Section 4). The Dirichlet boundary conditions can be considered as being imposed on an air-water interface. The RHS $f$ can be set to any function. For the experiments in this project, we use $f = 0$, which simplifies the poisson equation to the Laplace equation.

### 2.1 Discretization

The domain is discretized into a uniform cartesian lattice and the pressures (unknown) are stored at nodes. The domain is voxelized into interior cells and dirichlet cells. For a cell with cartesian coordinates $ci, cj, ck$, the pressure stored at 8 of its nodes is denoted as $p_{ijk}$, where each of $i, j, k$ range between 0 and 1. This means that dirichlet cells can not only be imposed on the boundary but also on the interior parts of the domain. Domain flags (a UINT8 tensor which is the same size as the domain) are created to indicate the type of the voxel (interior/dirichlet).

#### 2.1.1 Variational Form

Using the basics of Finite Element Method, we first write the poisson equation in Variational form as shown below. In order to bring it into variational form, we first choose a Hilbert space $H^1(\Omega)$. Then we pick a test function $v \in H^1(\Omega)$, multiply the poisson equation with this test function and integrate by parts to get the following expression:

$$\int_\Omega \nabla p.\nabla v d\mathcal{S} = \int_\Omega f.v d\mathcal{S}$$

#### 2.1.2 Finite Element Method

We use uniform square elements in the form of a uniform cartesian lattice and using a piecewise trilinear approximation. We refer to each cell in the domain as an element. Each element has 8 nodes in 3D, and we pick a piecewise trilinear function, denoted by $\phi(x, y, z)_i$ to form an orthogonal basis for $v$ in this element. Since the lattice is uniform, the width of each element is $h$. Since there are 8 degrees of freedom for each element, it is clear that the basis $\phi_i$ has a dimension of 8. We use Mathematica to work out the details as shown below and obtain the following elemental stiffness matrix:

$$\boldsymbol{\phi[1] = (1 - x/h)(1 - y/h)(1 - z/h)}$$

$$\boldsymbol{\phi[2] = (1 - x/h)(y/h)(1 - z/h)}$$

$$\boldsymbol{\phi[3] = (x/h)(1 - y/h)(1 - z/h)}$$

$$\boldsymbol{\phi[4] = (x/h)(y/h)(1 - z/h)}$$

$$\boldsymbol{\phi[5] = (1 - x/h)(1 - y/h)(z/h)}$$

$$\boldsymbol{\phi[6] = (1 - x/h)(y/h)(z/h)}$$

$$\boldsymbol{\phi[7] = (x/h)(1 - y/h)(z/h)}$$

$$\boldsymbol{\phi[8] = (x/h)(y/h)(z/h)}$$

**mat = Table $\left[\text{Integrate}\left[\nabla_{\{x,y,z\}}\phi[i].\nabla_{\{x,y,z\}}\phi[j], \{x, 0, h\}, \{y, 0, h\}, \{z, 0, h\}\right], \{i, 1, 8\}, \{j, 1, 8\}\right]$;**

**mat//MatrixForm**

$$\begin{pmatrix}
\frac{h}{3} & 0 & 0 & -\frac{h}{12} & 0 & -\frac{h}{12} & -\frac{h}{12} & -\frac{h}{12} \\
0 & \frac{h}{3} & -\frac{h}{12} & 0 & -\frac{h}{12} & 0 & -\frac{h}{12} & -\frac{h}{12} \\
0 & -\frac{h}{12} & \frac{h}{3} & 0 & -\frac{h}{12} & -\frac{h}{12} & 0 & -\frac{h}{12} \\
-\frac{h}{12} & 0 & 0 & \frac{h}{3} & -\frac{h}{12} & -\frac{h}{12} & -\frac{h}{12} & 0 \\
0 & -\frac{h}{12} & -\frac{h}{12} & -\frac{h}{12} & \frac{h}{3} & 0 & 0 & -\frac{h}{12} \\
-\frac{h}{12} & 0 & -\frac{h}{12} & -\frac{h}{12} & 0 & \frac{h}{3} & -\frac{h}{12} & 0 \\
-\frac{h}{12} & -\frac{h}{12} & 0 & -\frac{h}{12} & 0 & -\frac{h}{12} & \frac{h}{3} & 0 \\
-\frac{h}{12} & -\frac{h}{12} & -\frac{h}{12} & 0 & -\frac{h}{12} & 0 & 0 & \frac{h}{3}
\end{pmatrix}$$

### 2.2 Conjugate Gradients

A Conjugate Gradients (CG) solver is implemented using the torch library. This implementation of Conjugate Gradients does not have any preconditioning. Unlike the matrix-free approach in [1], this implementation explicitly builds the 3D laplacian matrix. The solver requires domain flags that indicate which nodes are interior or dirichlet and solves the pressure equations for interior nodes. The result of perturbing a simple rectangular domain of dimension 100x200x50 with the cells on the left and right faces on the rectangle as dirichlet (and the rest of the cells as interior) and solving the system using the CG solver is shown in Figure 1. The plot visualizes a slice of the domain.

### 2.3 Geometric Multigrid

We implement a V-cycle of the geometric multigrid scheme. The number of levels of multigrid is specified by the user. Since torch is a library primarily used for building neural network architecture, it has built in implementations for MAX-POOL, TRILINEAR INTERPOLATION. We leverage these implementations in our geometric multigrid implementation. At each level, the domain flags are coarsened using maxpooling and the coarsened pressures for the coarse interior nodes are initialized. The restriction and prolongation operators are constructed using the convolution and interpolation methods available in torch library. In one V-cycle of the geometric multigrid the following computations are needed:

1. Smoother: A smoother is implemented to smooth the error at all finer levels. This multigrid solver uses a Damped-Jacobi smoother (with a damping factor of 2/3).
2. Restriction Operator: The restriction operator transfers the low frequency error on the fine grid to the coarse grid using the convolution operator in Torch.
3. Solve at the coarsest level: The equation is solved completely at the coarsest level. The Conjugate Gradients solver is used to compute the solution.
4. Prolongation Operators: The prolongation operator, used to transfer the corrections from the coarser grids to the finer grids is implemented using the interpolation operation in torch.

#### 2.3.1 Coarsening the domain

Each coarse domain has a grid spacing of twice that of the fine domain. We coarsen the domain such that the corners of the fine and coarse grids are aligned. The rule used to label the coarse cells is as follows: If any of the fine cells composing the corresponding coarse cell is dirichlet, the coarse cell is marked dirichlet. If none of the fine cells composing a coarse cell are dirichlet, but atleast one of them is interior, the coarse cell is marked interior. Further, all nodes are initially marked interior and then, the nodes of a dirichlet cell are relabelled as dirichlet nodes.

#### 2.3.2 Restriction and Prolongation

The restriction operator is used to transfer quantities from the coarse grid to the fine grid. The restriction operator $\mathcal{R}$ for the 3D grid is computed as follows:

$$\mathcal{R} = \mathcal{B} \otimes \mathcal{B} \otimes \mathcal{B}$$

where

$$(\mathcal{B}u^h)(x) = \frac{1}{2}(u^h)(x-h) + (u^h)(x) + \frac{1}{2}(u^h)(x+h) = u^{2h}(x)$$

Since we align the corners of the fine grid and the coarse grid, the pressure quantities, stored at the nodes of both these grids will be aligned. To compute the value at one coarse node, this restriction operator is applied on the corresponding 3x3x3 fine node block. In this solver, we leverage the torch convolution operator to implement restriction. The prolongation operator $\mathcal{P}$ is the transpose of the restriction operator $\mathcal{P} = \mathcal{R}^T$. The prolongation operator is equivalent to trilinear interpolation.

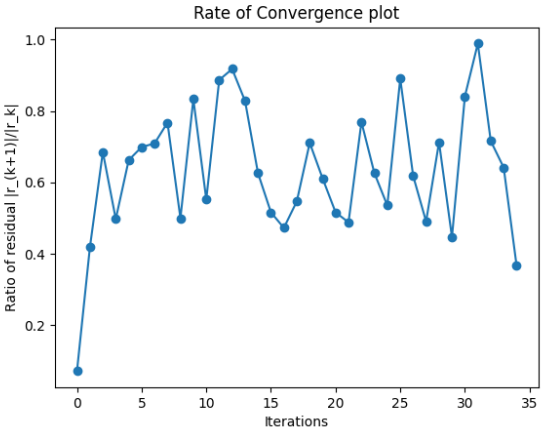



Figure 2: Convergence rate of the MGPCG solver. Y-axis is the ratio of the norm of the residual in the (k+1)th iteration to the norm of the residual in the (k)th iteration.

#### 2.3.3 Damped-Jacobi Smoother

We implement a Damped-Jacobi smoother with damping factor $\frac{2}{3}$. The pressures on interior nodes are smoothed using the FEM stiffness matrix derived in Section 2.1.2. Since the domain is regular with only dirichlet conditions, no boundary smoothing is done explicitly.

### 2.4 Multigrid-preconditioned Conjugate Gradients

The multigrid-preconditioned Conjugate Gradients, as given by [1] is implemented. Since we do not implement examples with all Neumann conditions, we do not include the corrections needed to remove the nullspace. In this solver, we use 4 levels of discretization for a v-cycle of geometric multigrid. A CG solve is done at the coarsest level. The prolongation and restriction operators are implemented as outlined in Section 2.3.2. In each call to the preconditioner, we perform 1 V-cycle smoothing. The number of smoothing iterations at the finer levels is set to 2. The convergence rate of multigrid for a domain size of 64x64x64 is shown in Figure 2.

### 2.5 Multiplication with the system matrix

Since multiplication of a vector with the system matrix is invoked multiple times in all stages of the solve, it is extremely essential to make this stage computationally efficient. We use 2 types of implementations, first of which is the convolution operation. The 3D FEM laplacian stencil, with 27 spokes is implemented as a convolution kernel and the entire domain can be convolved with this kernel. The second approach is through indices selection (gather). Only the interior cells are selected and the FEM laplacian stencil is applied, without the use of loops. From experiments, we observe that using the convolution operation is faster than the gather operation on the GPU but slower than gather on the CPU. This observation might be specific to the implementation specifics in the Torch library.

## 3 Results

We use the multigrid-preconditioned conjugate gradients solver on various example domains. In this section, we first outline the experiments and visualize the results. Then we profile the experiments and analyse the time taken on both CPU and GPU.

We test the solver on domains with Dirichlet conditions. An example domain is shown in Figure 3. We perturb the pressures of interior nodes by adding a noise. The pressures for each interior node after adding some noise is shown in Figure 4. The solved state of this domain after the multigrid-preconditioned solve is shown in Figure 5. A number of experiments with different domain sizes and similar boundary conditions are carried out on both the CPU and the GPU. The CPU is an Intel(R) Xeon(R) Gold 6312U CPU @ 2.40GHz with 48 cores and the GPU is a NVIDIA RTX A6000 with 48GB memory. The time required for initialization and solve is recorded for all the experiments and given in Table 1.

| Multigrid PCG (on CPU) | | | | | | | |
|---|---|---|---|---|---|---|---|
| Resolution | 64x3 | 96x3 | 128^3 | 192^3 | 256^3 | 384^3 | 512^3 |
| Initialization Time (s) | 0.0166 | 0.0293 | 0.0359 | 0.0837 | 0.1616 | 0.5468 | 1.1513 |
| Iterations to r = 1e-4 | 19 | 23 | 26 | 26 | 27 | 28 | 28 |
| Iterations to r = 1e-8 | 34 | 40 | 46 | 51 | 55 | 56 | 56 |
| Time to r = 1e-4 (s) | 0.632 | 3.9122 | 5.6611 | 14.4651 | 39.1442 | 126.2304 | 304.3648 |
| Time to r = 1e-8 (s) | 1.054 | 5.7998 | 9.7277 | 26.6973 | 77.7603 | 232.5261 | 609.1975 |
| Avg time per iteration | 0.0643 | 0.3151 | 0.4292 | 1.0798 | 2.8636 | 8.6605 | 21.7487 |
| | | | | | | | |
| Multigrid PCG (on GPU) | | | | | | | |
| Resolution | 64x3 | 96x3 | 128^3 | 192^3 | 256^3 | 384^3 | 512^3 |
| Initialization Time (s) | 2.9058 | 3.0854 | 2.8844 | 2.7641 | 2.8915 | 2.906 | 3.266 |
| Iterations to r = 1e-4 | 19 | 23 | 26 | 26 | 27 | 28 | 28 |
| Iterations to r = 1e-8 | 34 | 40 | 46 | 52 | 54 | 56 | 56 |
| Time to r = 1e-4 (s) | 2.0791 | 2.0816 | 2.3263 | 2.9375 | 4.2127 | 8.8284 | 18.579 |
| Time to r = 1e-8 (s) | 2.0887 | 2.263 | 2.6761 | 3.8931 | 6.2674 | 15.376 | 33.7145 |
| Avg time per iteration | 0.1709 | 0.1471 | 0.1476 | 0.1878 | 0.2721 | 0.5899 | 1.2656 |
| | | | | | | | |
| Conjugate Gradients (on CPU) | | | | | | | |
| Resolution | 64x3 | 96x3 | 128^3 | 192^3 | 256^3 | 384^3 | 512^3 |
| Initialization Time (s) | 0.0127 | 0.0037 | 0.0035 | 0.0151 | 0.0502 | 0.1626 | 0.4043 |
| Iterations to r = 1e-4 | 109 | 155 | 200 | 284 | 366 | 510 | 648 |
| Iterations to r = 1e-8 | 229 | 314 | 405 | 566 | 728 | 972 | 1299 |
| Time to r = 1e-4 (s) | 0.2577 | 1.5436 | 4.0436 | 12.5108 | 53.0532 | 238.011 | 734.2831 |
| Time to r = 1e-8 (s) | 0.6356 | 2.8888 | 8.3283 | 25.5901 | 105.3293 | 453.8295 | 1514.8306 |
| Avg time per iteration | 0.0051 | 0.0192 | 0.0408 | 0.0893 | 0.2896 | 0.9336 | 2.2993 |
| | | | | | | | |
| Conjugate Gradients (on GPU) | | | | | | | |
| Resolution | 64x3 | 96x3 | 128^3 | 192^3 | 256^3 | 384^3 | 512^3 |
| Initialization Time (s) | 2.9247 | 2.9038 | 2.8842 | 2.8907 | 2.901 | 2.8836 | 3.1554 |
| Iterations to r = 1e-4 | 110 | 154 | 199 | 286 | 365 | 507 | 645 |
| Iterations to r = 1e-8 | 227 | 316 | 404 | 591 | 729 | 1063 | 1280 |
| Time to r = 1e-4 (s) | 1.7215 | 1.6113 | 1.7247 | 2.4354 | 4.2282 | 14.1253 | 40.1211 |
| Time to r = 1e-8 (s) | 1.6297 | 1.6894 | 1.9455 | 3.4258 | 6.9479 | 28.1942 | 77.5975 |
| Avg time per iteration | 0.0228 | 0.0158 | 0.0135 | 0.0143 | 0.0211 | 0.0544 | 0.1228 |

Table 1: Performance (time taken in seconds) of the Conjugate Gradients Solver and the multigrid-preconditioned Conjugate Gradients Solver on different domain sizes with similar boundary conditions on both the CPU and GPU.

### 3.1 Analysis

The behavior for different boundary conditions across different domain sizes have been consistent for both the CG and the MGPCG solvers. Figure 6 compares the time taken for the MGPCG and the CG implementations on the GPU and Figure 7 compares the time taken for the MGPCG and CG implementations on the CPU. The trend in both these graphs suggest the following:

- The performance of the MGPCG solver is better than the CG solver for domain sizes larger than 192x192x192. As the resolution of the domain increases, the MGPCG solver gives higher speed-up over the CG solver. For a domain size of 256x256x256, the speedup given by the MGPCG solver is 1.1x whereas for the 512x512x512 domain, the speedup is 2.5x. The speed-up will only increase when the resolution is higher than 512x512x512.
- For resolutions of 128x128x128 or lesser, the CG solver is slightly faster than the MGPCG solver. This is due to the fact that the added V-cycle compute for every CG iteration in the MGPCG solver increases the time for each iteration, and for smaller resolutions, this additional cost is higher than the no-conditioning CG solve.
- The time taken for each iteration of the CG solver is less than that of the MGPCG solver due to the preconditioning using a multigrid V-cycle. On the GPU, while the average time taken per MGPCG solver iteration is 1.2s, the average time taken for one iteration of the CG solver is 0.12s, the latter is 10x faster. This indicates that the V-cycle preconditioning in each iteration of the MGPCG solver adds a lot of overhead.

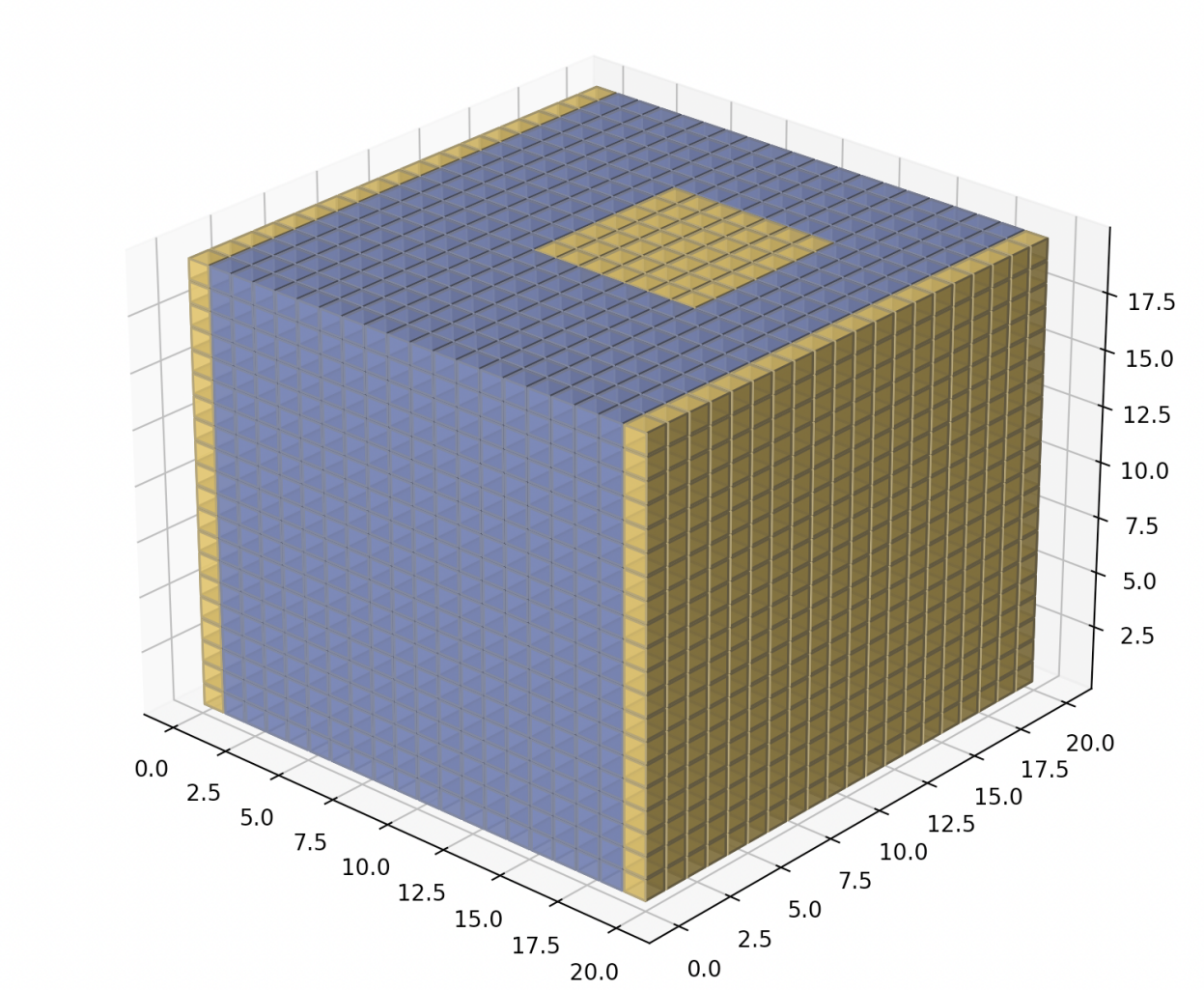



Figure 3: An example of a domain used with the solver. The yellow cells indicate dirichlet boundary conditions and the blue cells are interior cells.

- The similar behaviour across CPU and GPU suggests that the multigrid-preconditioned CG algorithm in [1] can be extended onto the GPU and similar benefits can be observed even on the GPU. However, using libraries like torch, which are ideal for deep learning, might not be very efficient for fine-grained memory access (such as boundary smoothing mentioned in [1]). For instance, traversing the entire domain to find boundary cells (cells within some radius of either Dirichlet or Neumann cells) will itself be expensive when implemented using libraries like torch. To the best of our knowledge, torch does not allow for specifying explicit parallelism on the CPU.

## 4 Future Work

- *Including Neumann Boundary Conditions*: While it is straightforward to make changes to our implementation to include Neumann boundary conditions, we do not include them for 2 main reasons:
  - The Neumann boundary conditions implementation requires traversing through the domain and involves making a non-trivial change to the FEM system matrix. This domain traversal becomes expensive while using a library like torch. A departure from using torch and adopting multithreading libraries can alleviate the cost for this step.

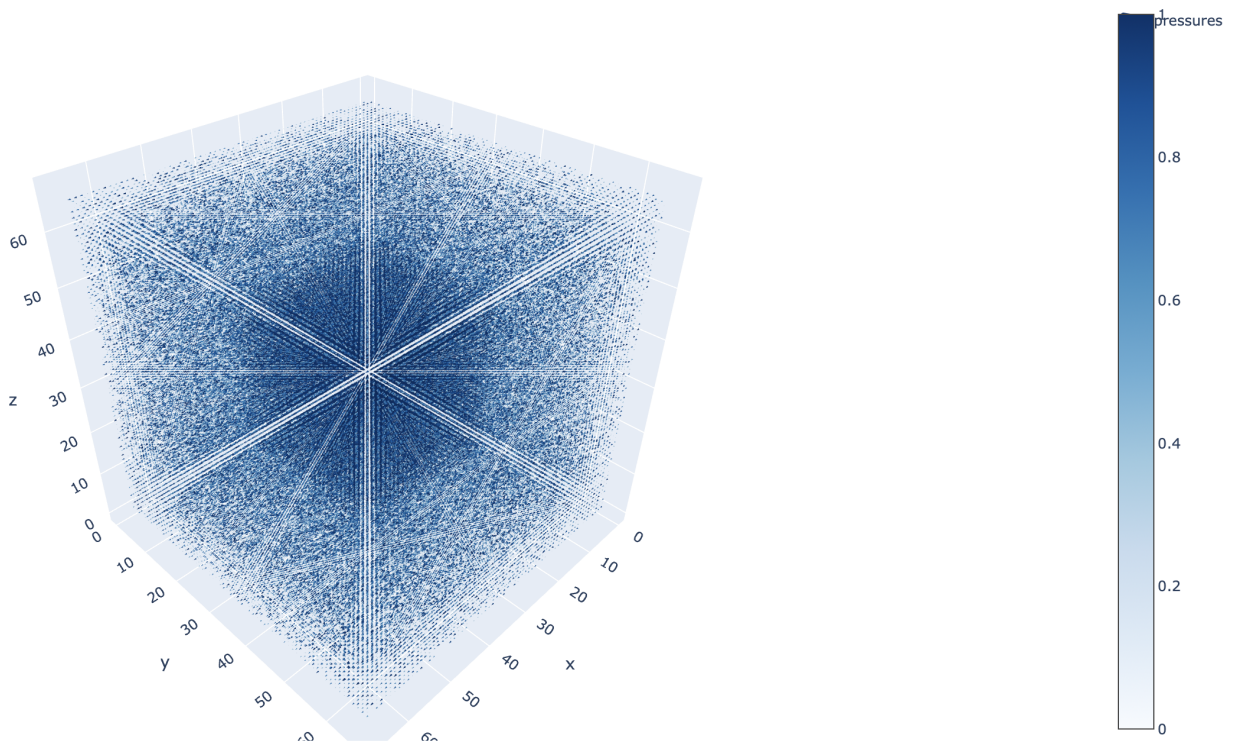



Figure 4: Pressure visualized for all interior nodes after adding noise. The domain is sized 64x64x64 and the cell block from (20:50)x(20:50)x(20:50) is initialized to dirichlet (and all nodes in this block is set to a value of 1).

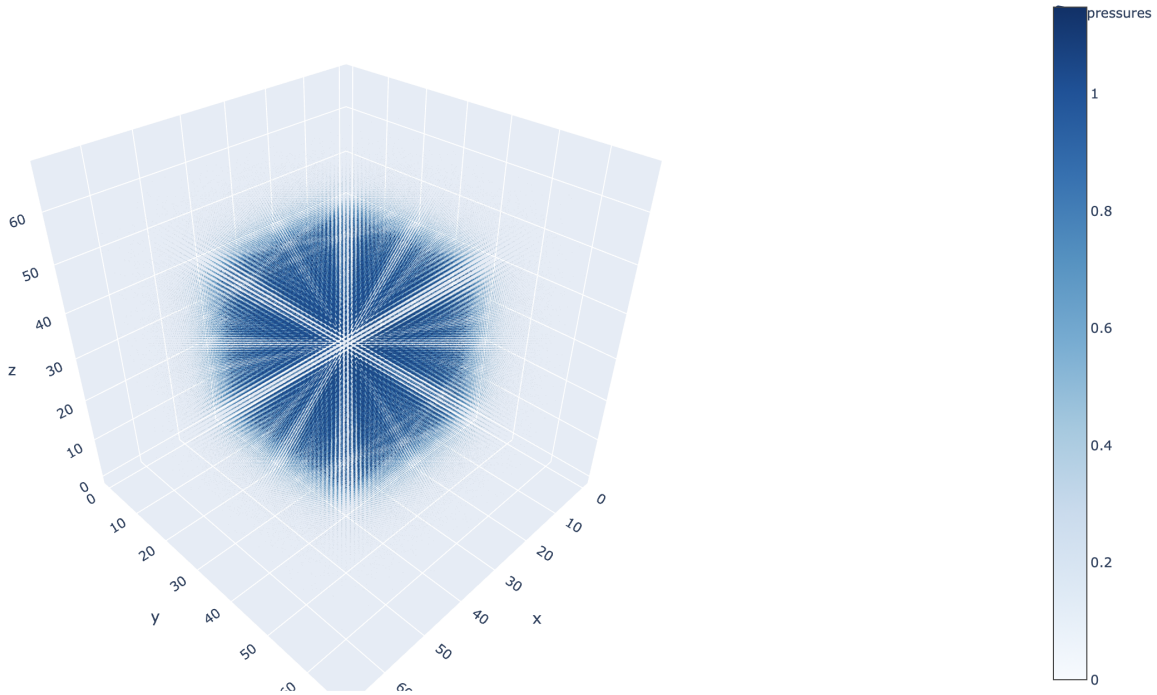



Figure 5: Pressure visualized for all interior nodes after the multigrid preconditioned solve. The domain is sized 64x64x64 and the cell block from (20:50)x(20:50)x(20:50) is initialized to dirichlet (and all nodes in this block is set to a value of 1).

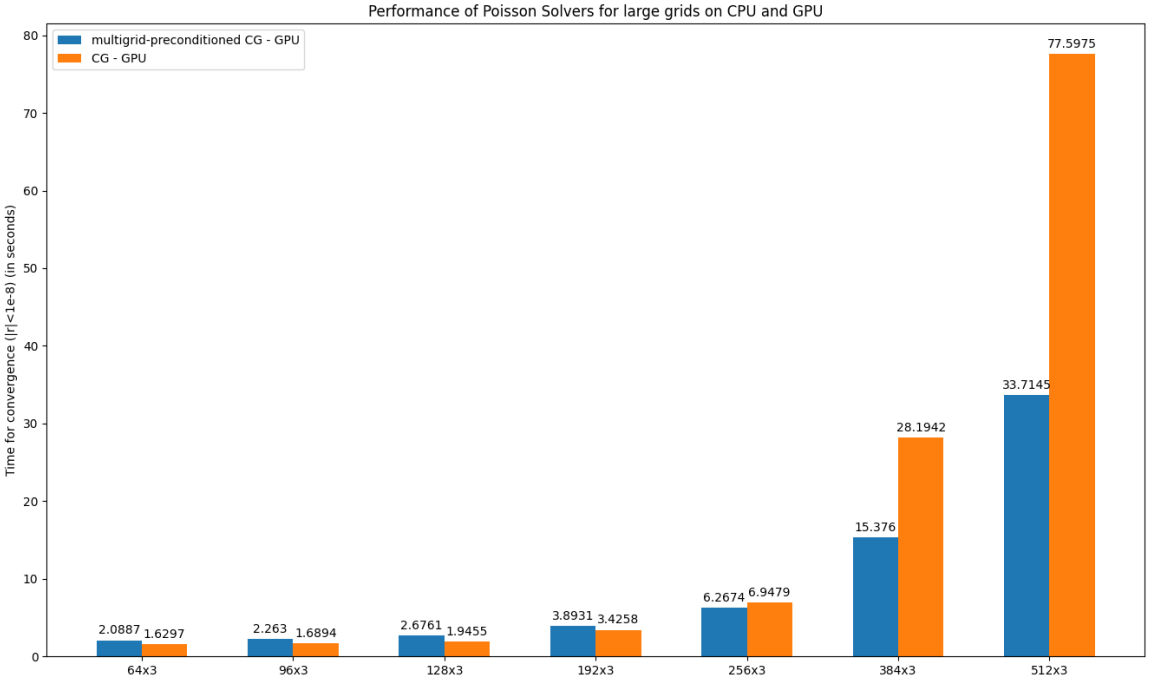



Figure 6: A plot of time taken for the MGPCG solver vs the CG solver for different domain sizes on the GPU.

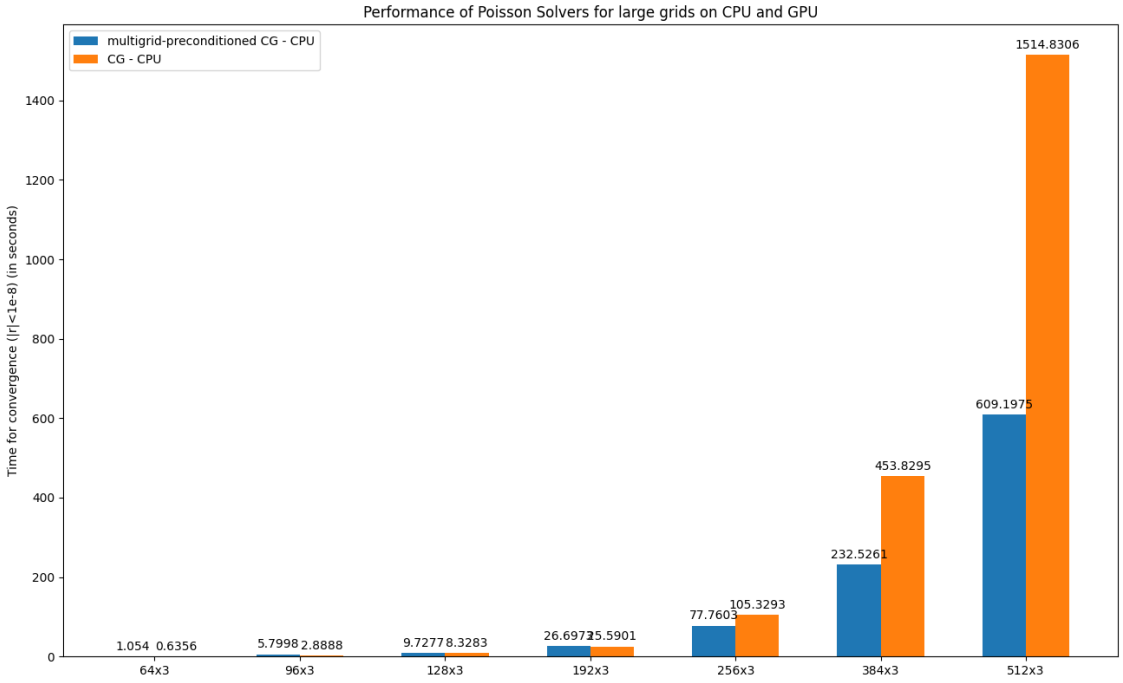



Figure 7: A plot of time taken for the MGPCG solver vs the CG solver for different domain sizes on the CPU.

  - The appplication of the finite element stencil on the grid is implemented using the convolution operator in torch. However, the library does not extend support to convolving with different custom kernels at each grid cell. The 2 options to circumvent this would be to either pay the cost of convolving the entire grid with multiple kernels with different weights, each time keeping the values of the desired nodes, or using explicit multithreading on a matrix-vector multiplication. With the second option, the explicit finite element matrix needs to be stored and we might lose the advantage of using the gradients computed by torch.

  From the observations made during the implementation of this solver using the torch library, a complex domain with a mixture of Dirichlet and Neumann boundary conditions will not be as efficient as a carefully optimized C++ implementation.

- *Support for better visualization*: Once solved, the pressures are then used to compute velocity of particles using an advection equation. These particle velocities are then used for visualization of the fluid particles. This work can be further extended to solve for velocities as described in [3]. Note that since we store pressures at nodes instead of cell centres, the pressure at a cell centre can be the average of pressures of all the nodes.

## References


[1] A. McAdams, E. Sifakis, and J. Teran. A parallel multigrid poisson solver for fluids simulation on large grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, page 65–74, Goslar, DEU, 2010. Eurographics Association.

[2] Christopher Batty, Florence Bertails, and Robert Bridson. A fast variational framework for accurate solid-fluid coupling. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, page 100–es, New York, NY, USA, 2007. Association for Computing Machinery.

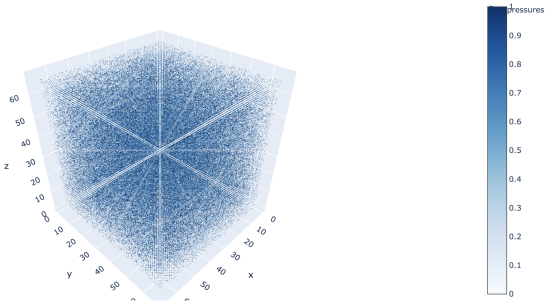
[3] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 23–30, New York, NY, USA, 2001. Association for Computing Machinery.

[4] Cheng Yang, Xubo Yang, and Xiangyun Xiao. Data-driven projection method in fluid simulation. *Computer Animation and Virtual Worlds*, 27(3-4):415–424, 2016.

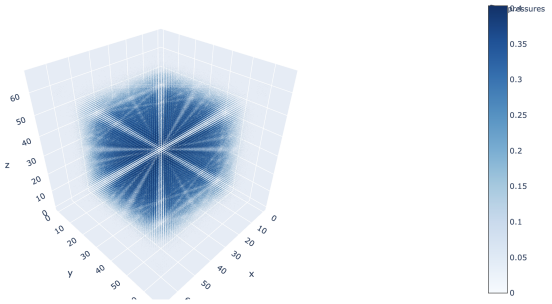
[5] S. Wiewel, M. Becher, and N. Thuerey. Latent space physics: Towards learning the temporal evolution of fluid flow. *Computer Graphics Forum*, 38(2):71–82, 2019.

[6] Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph.*, 33(6):205:1–205:12, November 2014.

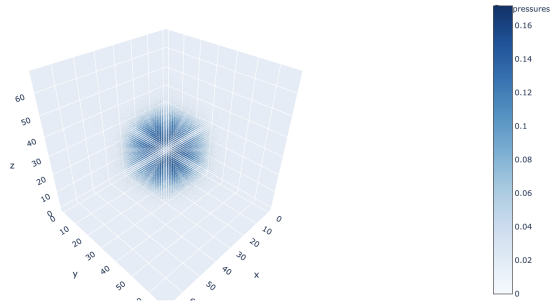
[7] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating Eulerian fluid simulation with convolutional networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3424–3433. PMLR, 06–11 Aug 2017.



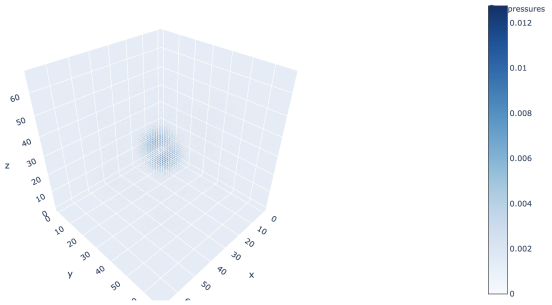

(a) Snapshot of the domain at 0 MGPCG iterations.



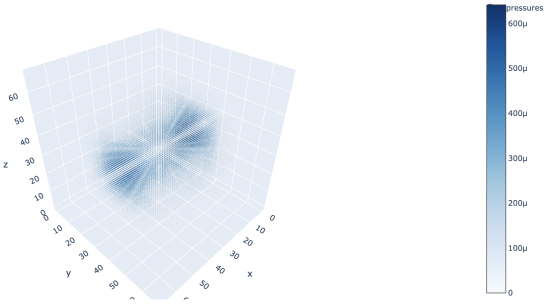

(b) Snapshot of the domain at 5 MGPCG iterations.




(c) Snapshot of the domain at 10 MGPCG iterations.




(d) Snapshot of the domain at 15 MGPCG iterations.




(e) Snapshot of the domain at 20 MGPCG iterations.

Figure 8: Snapshot of the domain at every 5 iterations during the MGPCG solve. Dirichlet boundaries specified at the faces of the domain.

## 5 Appendix

We show the progress of the MGPCG solve after every 5 iterations in a 20 iterations solve in Figure 8.