PHAS0102: Final Project

Srinath Kailasa*
Department of Physics,
University College London,
Gower Street, London, WC1E 6BT
(Dated: March 8, 2019)

Abstract: In this work we solve a matrix weighted Poisson equation using OpenCL, using a GPU accelerated finite difference method in order to compare multiple iterative solvers. We validate our approach using, FEniCS, an open-source Finite-Element modelling library, and compare and contrast the performance of different iterative solvers, namely GMRES and BiCGSTAB. We find BiCGSTAB to offer faster convergence for our problem.

I. MOTIVATION

The Poisson equation is common in many physical contexts, from electrostatics and gravitation in physics, to expressing the pressure field in an in-compressible fluid in fluid dynamics [1]. Generally it can be expressed as,

$$\nabla^2 u = v \tag{1}$$

where $u(\mathbf{r})$ is some scalar potential which is to be determined, and $v(\mathbf{r})$ is a known "source function." The most common boundary condition applied to this equation is that the potential u is zero at infinity. The solutions to the Poisson equation are superposable, meaning that a general solution can be built up of solutions for point sources. For a point source, the solution is of the form,

$$u(\mathbf{r}) = \frac{1}{|\mathbf{r} - \mathbf{r}'|} \tag{2}$$

where the general solution is given by the integral over the problem domain,

$$u(\mathbf{r}) = \int G(\mathbf{r}, \mathbf{r}') v(\mathbf{r}') d\mathbf{r}'. \tag{3}$$

$G(\mathbf{r}, \mathbf{r}')$ being given by (2)

Analytically solving this integral can be tricky in realistic systems, with complex boundary or initial conditions, and we must appeal to some form of numerical approximation. Unfortunately, for the large and complex systems we often want to solve in science and engineering, numerical algorithms can bear a significant time overhead. However, we can exploit the parallelism of modern computer architectures, specifically GPUs, to accelerate these algorithms. In this report and adjoining codes, we examine a toy Poisson problem, and provide accelerated implementations of various iterative solvers using the OpenCL framework, SciPy (Python's scientific computing library) and a finite difference approach. We proceed to compare and contrast the performance of different iterative methods, namely GMRES and BiCGSTAB, in terms of their convergence and mesh-size. Finally, we validate our work using the open-source Finite-Element modelling library FEniCS, as it takes a different approach to solving the same system.

II. MATHEMATICAL BACKGROUND

A. Discretisation of the Poisson Equation using Finite Differences

Consider the following 2D matrix-weighted Poisson equation which will serve as our toy problem,

$$-\nabla \cdot (\sigma(x, y)\nabla)\, u(x, y) = f(x, y) \tag{4}$$

for $(x, y) \in [0, 1] \times [0, 1]$ with boundary conditions $u(x, y) = 0$, where we take $\sigma(x, y)$ to be some non-linear function over the domain. We will solve this equation using a finite-difference method (FDM), which at its core, is nothing more than a direct conversion of the Poisson equation from continuous functions and operators into their discretely-sampled counterparts. This converts the problem into a system of linear equations that may be readily solved via matrix inversion. The accuracy of such a method is therefore directly tied to the ability of a finite grid to approximate a continuous system, and errors may be arbitrarily reduced by simply increasing the number of samples [2].

Using a standard centered finite difference scheme, let $u_{i,j} := u(x_i, y_j) = u_{i,j}$ be the solution on given grid point. We can approximate the application of the left-hand side operator $\nabla \cdot (\sigma(x, y)\nabla)$ through,

$$\begin{aligned} &\approx \frac{\left(\sigma_{i+1/2,j} \frac{(u_{i+1,j} - u_{i,j})}{h}\right) - \left(\sigma_{i-1/2,j} \frac{(u_{i,j} - u_{i-1,j})}{h}\right)}{h} \\ &+ \frac{\left(\sigma_{i,j+1/2} \frac{(u_{i,j+1} - u_{i,j})}{h}\right) - \left(\sigma_{i,j-1/2} \frac{(u_{i,j} - u_{i,j-1})}{h}\right)}{h} \end{aligned} \tag{5}$$

and the fractionally indexed $\sigma(x, y)$ using means of the form,

$$\sigma_{i+1/2,j} \approx \frac{1}{2}\left(\sigma_{i+1,j} + \sigma_{i,j}\right) \tag{6}$$

* srinath.kailasa.18@ucl.ac.uk

To simplify our problem, we will take $f(x, y) = 1$ in Equation (4), in which case the solution of our problem can be shown to be,

$$u_{i,j} = \frac{1}{\alpha}(h^2 + \sigma_{i+1/2,j}u_{i+1,j} + \sigma_{i-1/2,j}u_{i-1,j} + \sigma_{i,j+1/2}u_{i,j+1} + \sigma_{i,j-1/2}u_{i,j-1}) \quad (7)$$

where,

$$\alpha = (\sigma_{i+1/2,j} + \sigma_{i-1/2,j} + \sigma_{i,j+1/2} + \sigma_{i,j-1/2}) \quad (8)$$

for interior grid points, where the boundary conditions do not apply. In order to make use of standard iterative solvers, we needed to reformulate our problem into a linear system of equations, of the form,

$$Au = b \quad (9)$$

which we have now expressed in Equation (7). Without explicitly writing out the matrices of Equation (9), it is easy to see that A can be constructed from the linear relations between the components of u and σ for interior points (for boundary points the effect of applying A will just be an identity operation), and that b encapsulates encapsulates the initial conditions, with 0 components for boundary terms, and mesh parameters, with $-h^2$ for interior terms. Additionally it becomes clear that computation of the solution at a grid point $u_{i,j}$ is independent of the computation of a solution at any other grid point, exposing the inherent parallelism in solving this system, and therefore that the left-hand side Au, is trivially parallelisable.

B. Iterative Methods

For very large linear systems (> 1000 grid points) of the form (9), direct methods such as LU decomposition or Gaussian Elimination (which have computational complexities of $O(N^3)$ where N is the dimension of the square matrix in question) become too expensive, so we must appeal to iterative methods which instead approximate solutions. For such systems, the classical approaches for solving a linear system using iterative methods (Jacobi/Gauss-Seidel) are not appropriate, therefore we use Krylov subspace methods [3]. The idea behind these methods is easily motivated. Consider a simple iterative method,

$$x_{k+1} = (I - A)x_k + b \quad (10)$$

After k iterations, we would have,

$$x_k = P_{k-1}(A)b \quad (11)$$

where $P_{k-1}(A)$ is some polynomial in A, where with appropriately chosen coefficients would approximate $A^{-1}b$. Defining Krylov subspaces at iteration k as

$$\text{span}\left\{b, Ab, A^2b, ..., A^{k-1}b\right\} \quad (12)$$

for the iterative methods we consider, GMRES and BiCGSTAB, we choose x_k from the k^{th} Krylov subspace by in some way minimising the norm of the residual $r_k = Ax_k - b$. This amounts to narrowing the domain of the problem to the 2D vector subspace defined by (12), and they have computational complexities of $\Theta(N^2)$, where N is the dimension of A [4]. This represents an improvement in complexity, in comparison to direct methods. and offers a way to compute solutions to large linear systems. Summarising the differences between these methods, GMRES and BiCGSTAB are both used for solving non-symmetric systems, however if memory and or large numbers of iterations are possible constraints BiCGSTAB is more commonly used.

C. Finite Element Methods

We validate our approach using FenICS which solves PDEs using a finite element modelling (FEM) approach. We defer to the literature for a full description of FEM [5], for our use case it is sufficient to understand that it amounts to a more complex version of finite difference modelling. Instead of discretising the problem domain into a set of grid points, we instead form a mesh made of discrete elements, FEM then interpolates the solution between node points on this mesh. If we can find a discretisation operator such that our problem is of the form given by equation (9), then we can apply the same iterative methods as above to find a solution.

III. COMPUTATIONAL BACKGROUND

The ubiquity of parallel computing has now lead to many new applications in computational science. Specifically, convenient frameworks have emerged that make it easy to map parallel problems to modern computer architectures, both CPU and GPU. Here we use OpenCL which has emerged as a new standard API for parallel computation across heterogeneous platforms (CPU and GPU), and we use a simple Intel HD Graphics 5000 card, shipped with the 2013 MacBook Air, for computation.

The basic idea behind parallel computing is that if processes are independent of each other for their computation, they can be done at the same time. The fundamental speed of our implementation underscored by the fraction of computation that is parallelisable (Amdahl's law).

CPUs are designed for general purpose computing, and therefore are able to handle complex data structures, and

arbitrary instructions, however GPUs are somewhat simpler in their architecture - being better suited to 'Single Instruction Multiple Data' (SIMD) type problems, an illustrative example being a matrix vector product.

The idea behind OpenCL and other frameworks (CUDA) is to abstract away from the physical architecture, and allow a developer to consider instead an abstract hierarchy of memory access, and groups of computations that are to be executed in parallel [6]. The implementation consists of a 'kernel' which defines an instruction to be executed across multiple data (SIMD). Therefore, in the example of matrix vector products, our kernel will essentially apply equation (7), accessing the appropriate components from memory, and writing the results to a buffer. This allows us the parallelise the computation of the left-hand side of equation (9).

Furthermore, as Python offers many common iterative solvers through the SciPy library, which take as an input the left-hand side and right-hand side of equation (9), we are able to effectively accelerate our implementation of FDM using OpenCL.

IV. EXPERIMENTS

Figure (1) demonstrates the OpenCL implementation of the matrix vector product in equation (9), in combination with SciPy's sparse-linear algebra solvers, specifically we show the results of using GMRES and BiCGSTAB. We've taken $\sigma(x, y)$ to be a matrix containing normally distributed random values (with parameters $\mu = 0$ and σ=0.1). The similarity of the both solutions demonstrates their convergence for our toy system.

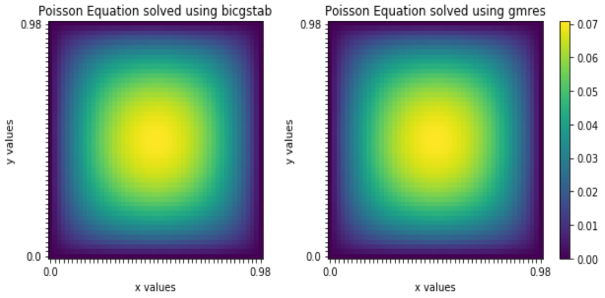

FIG. 1. Solved with square matrices of dimension $N = 50$ using OpenCL.

However, the differences in the implementation of both algorithms makes it worthwhile to examine the nature of their convergence. In figure (2) we see that BiCGSTAB converges considerably faster in terms of time steps than GMRES for our system.

We can also test how the methods scale in terms of number of iterations to convergence, with increasingly fine meshes. Figure (3) demonstrates that though BiCGSTAB scales approximately linearly with increasing system size, GMRES appears quadratic. Fitting

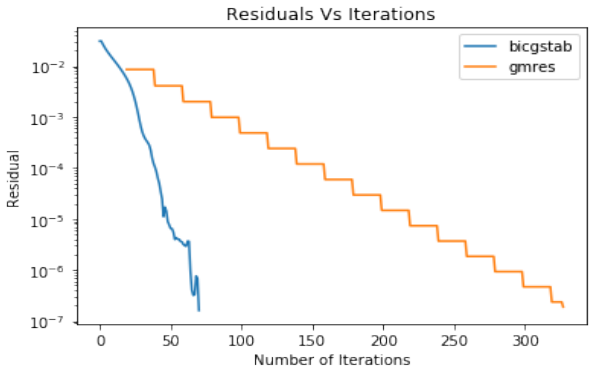

FIG. 2. Evolution of the residual error per iteration, until convergence. Solved with square matrices of dimension $N = 50$.

polynomials to these two relationships we find that for BiCGSTAB, there is an approximate linear relationship between iterations and mesh size, as expected, with a gradient of 1.28 (3sf). For GMRES, we note that for relationships of the form,

$$y = Cx^p \tag{13}$$

We can calculate the power p by taking the log of both sides, and fitting a straight line,

$$\log(y) = p\log(x) + \log(C) \tag{14}$$

Where the fitted value of p is the gradient. Performing this we find this value to be 1.90 (3sf) for GMRES, broadly consistent with our conjecture of quadratic behaviour.

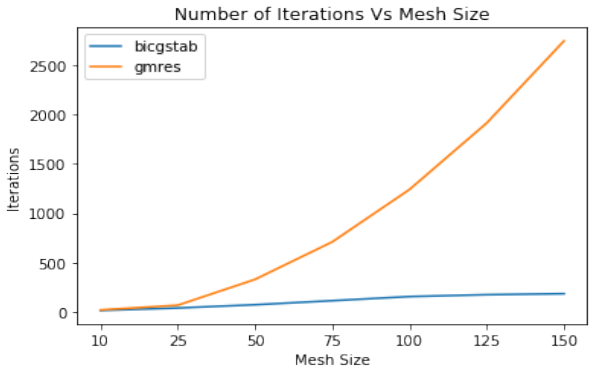

FIG. 3. 'Mesh Size' is the equivalent to the dimension N of the square matrix being considered, therefore larger mesh size, means finer meshes.

The performance difference between the two methods, both in terms of scaling, as well as convergence, is likely due to the underlying nature of the left-hand side of equation (4). Due to the highly non-symmetric random weighting of $\sigma(x, y)$, GMRES is clearly having to

generate very large Krylov subspaces before converging. Although not examined here, this will also result in an additional space-overhead for this iterative method.

We know both methods are bounded by the same computational complexity, and therefore we would expect that since GMRES computes the solution in more time steps, each time step should be less costly than a time step of BiCGSTAB. Performing this experiment (with a square matrix of dimension $N = 50$) we find that given time step of GMRES costs approximately 0.0017 seconds (2sf) on our architecture, compared to 0.0034 seconds (2sf) for BiCGSTAB which as expected is lower. However from figure (2), we can see that GMRES takes around 4.5 times as many iterations to converge compared to BiCGSTAB, implying that BiCGSTAB has a slightly more favourable runtime complexity for this problem.

As demonstrated by figure (4), both methods result in solutions of around the same precision. Additionally we observe that we can arbitrarily increase the precision by using finer meshes, as expected. Strangely, the finest mesh results in a decrease of precision for BiCGSTAB, however this may well be due to rounding errors.

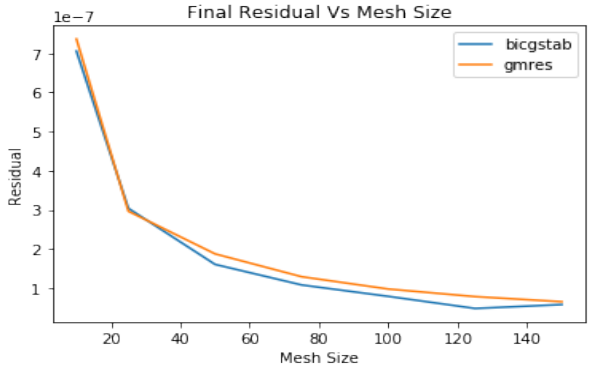

FIG. 4. Final residual error for increasingly fine meshes. Again, 'Mesh Size' is the equivalent to the dimension N of the square matrix being considered see fig. (3)

Finally, we can validate our approach using FEniCS. The idea here being that as it's an independent implementation, using a different mathematical approach, a similar result would offer us validation for our results.

Instead of using a normally distributed random matrix for $\sigma(x, y)$, we now use a known non-linear function of the grid points $\sigma(x, y) = 1 + x^2 + y^2$. This is to ensure that each element on the left hand side of equation (4) is being multiplied by a unique number, ensuring that the FenICS and OpenCL results are comparable.

It is difficult to compare the results in a more quantitative way, due to the differences in the underlying data structures used by both methods (grid points vs triangular meshes), however figure (5) demonstrates the qualitative similarity between solutions. The effect of the the different non-linear function appearing to slightly distort in comparison to the solution using a normally distributed random $\sigma(x, y)$.

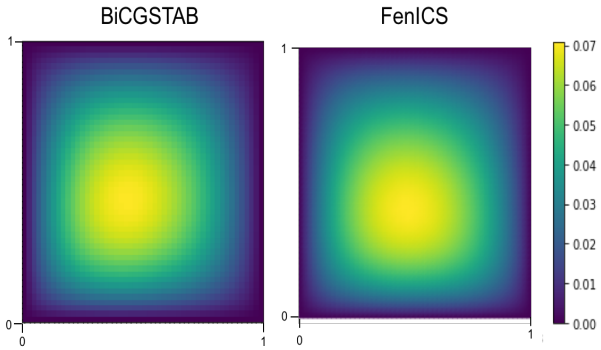

FIG. 5. Computed on a square matrix of dimension $N = 50$

V. CONCLUSIONS

In this report we've examined some common iterative methods of the Krylov subspace variety in order to solve a toy-PDE. We accellerated our implementation using the OpenCL framework, finding BiCGSTAB outperformed GMRES for this problem. Finally we validated our result by considering a different implementation of the same problem via FenICS. Future work could focus on preconditioning the problem in order to improve the Krylov methods examined.

[1] Y. Young, *University Physics with Modern Physics* (Pearson Education, 2008).

[2] Solving the generalized poisson equation using the finite-difference method (fdm), `http://www.ece.utah.edu/~ece6340/LECTURES/Feb1/Nagel\%202012\%20-\%20Solving\%20the\%20Generalized\%20Poisson\%20Equation\%20using\%20FDM.pdf`, accessed: 2019-03-07.

[3] A brief introduction to krylov space methods for solving linear systems, `http://www.sam.math.ethz.ch/~mhg/pub/biksm.pdf`, accessed: 2019-03-07.

[4] B. Oancea, Improving the Performance of the Linear Systems Solvers Using CUDA (2015), arXiv:1511.07207v1.

[5] E. Sliand D. F. Mayers, *An Introduction to Numerical Analysis* (Cambridge University Press, 2003).

[6] A gentle introduction to opencl, `http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854`, accessed: 2019-03-07.

[7] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003).

[8] Understanding the bi-conjugate gradient stabilized method (bi-cgstab), `https://static1.`

squarespace.com/static/55ade5ebe4b0d3eba632821b/t/576ebb166a4963e8802f5d67/1466874648553/Yuvashankar_Nejad_Liu.pdf, accessed: 2019-03-07.