PHAS0102: Final Project

Srinath Kailasa*
Department of Physics,
University College London,
Gower Street, London, WC1E 6BT
(Dated: March 26, 2019)

Abstract: In this work we solve a matrix weighted Poisson equation with 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. We extend our analysis, by proceeding to solve a time-dependent version of the problem, amounting to a primitive diffusion model, benchmarking the performance of our GPU accelerated implementation, finding it to be I/O bound.

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)

Just as the Poisson equation appears in many physical contexts, so too does diffusion - from the conduction of heat or electricity in solids, to financial modelling - such as the famous Black-Scholes model for options pricing [2]. Generally the heat equation, which is a model of such processes, can be expressed as,

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0 \tag{4}$$

A similar Green's function based treatment may be used to analytically solve the heat equation too, however solving these integrals 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 compare and contrast the performance of different iterative methods, namely GMRES and BiCGSTAB, in terms of their convergence and mesh-size. We proceed to validate our work using the open-source Finite-Element modelling library FEniCS, as it takes a different approach to solving the same system. We extend this analysis by adding a time-discretisation to our problem, and solve a toy diffusion problem. We compare the diffusive behaviour resulting from different initial conditions, and benchmark the performance of our OpenCL implementation.

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,

$$-\nabla \cdot (\sigma(x, y)\nabla)\, u(x, y) = f(x, y) \tag{5}$$

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

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

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 [3].

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,

$$\approx \frac{\left(\sigma_{i+1/2,j}\frac{(u_{i+1,j}-u_{i,j})}{h}\right) - \left(\sigma_{i-1/2,j}\frac{(u_{i,j}-u_{i-1,j})}{h}\right)}{h} \quad (6)$$
$$+\frac{\left(\sigma_{i,j+1/2}\frac{(u_{i,j+1}-u_{i,j})}{h}\right) - \left(\sigma_{i,j-1/2}\frac{(u_{i,j}-u_{i,j-1})}{h}\right)}{h}$$

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

$$\sigma_{i+1/2,j} \approx \frac{1}{2}\left(\sigma_{i+1,j} + \sigma_{i,j}\right) \quad (7)$$

To simplify our problem, we will take $f(x, y) = 1$ in Equation (5), in which case the solution of our problem can be shown to be,

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

where,

$$\alpha = (\sigma_{i+1/2,j} + \sigma_{i-1/2,j} + \sigma_{i,j+1/2} + \sigma_{i,j-1/2}) \quad (9)$$

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,

$$Au = b \quad (10)$$

which we have now expressed in Equation (8). Without explicitly writing out the matrices of Equation (10), 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 (10), 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 [4]. The idea behind these methods is easily motivated. Consider a simple iterative method,

$$x_{k+1} = (I - A)x_k + b \quad (11)$$

After k iterations, we would have,

$$x_k = P_{k-1}(A)b \quad (12)$$

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

$$\text{span}\left\{b, Ab, A^2b, ..., A^{k-1}b\right\} \quad (13)$$

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 (13), and they have computational complexities of $\Theta(N^2)$, where N is the dimension of A [5]. 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 [6], 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 (10), then we can apply the same iterative methods as above to find a solution.

D. Extending the Finite Difference Method for Explicit Time Integration Schemes

For our toy model of diffusion, we'll consider the following parabolic PDE,

$$\frac{\partial u}{\partial t} = \nabla \cdot (\sigma(x,y)\nabla)\, u \tag{14}$$

where all symbols take their previous meanings. It is relatively simple to extend our analysis from section II A for time integration schemes, and in this work we approximate the time derivative operator using a forward difference scheme,

$$\frac{\partial u}{\partial t} \approx \frac{u(x,y,t+\Delta t) - u(x,y,t)}{\Delta t} \tag{15}$$

With the same spatial discretisation as above. Writing down the solution in terms of it's components we find,

$$\begin{aligned} u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{\Delta t}{(\Delta r)^2} \cdot (\\ (\sigma_{i+\frac{1}{2},j}(u_{i+1,j}^{n} - u_{i,j}^{n}) - \sigma_{i-\frac{1}{2},j}(u_{i,j}^{n} - u_{i-1,j}^{n})) \\ + (\sigma_{i,j+\frac{1}{2}}(u_{i,j+1}^{n} - u_{i,j}^{n}) - \sigma_{i,j-\frac{1}{2}}(u_{i,j}^{n} - u_{i,j-1}^{n})) \\ ) \end{aligned} \tag{16}$$

The index n indicates the time step, Δt is the magnitude of the time step, and Δr is the spatial step (we take this to be the same for both dimensions in the domain). As with (8) it is clear that the solution at the next time step is in the form of a matrix-vector product (10), and for a specific grid-point independent of the solution at other grid points, making the problem trivially parallelisable. Furthermore, from expression (16) it's clear that there are bounds to the relationship between Δt and Δr in order to ensure that (16) converges. Specifically,

$$\Delta t \leq (\Delta r)^2 \tag{17}$$

Assuming the quantities from the matrix vector product are greater than or equal to zero. We note that this is a CFL-type condition [7]. Below we examine this relationship to see determine experimentally what this bound is for our system.

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 [8]. 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 (8), accessing the appropriate components from memory, and writing the results to a buffer. This allows us to parallelise the computation of the left-hand side of equation (10).

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 (10), we are able to effectively accelerate our implementation of FDM using OpenCL. This logic is easily extended to accelerate the solution of (16) for each time step.

IV. EXPERIMENTS

A. Poisson

Figure (1) demonstrates the OpenCL implementation of the matrix vector product in equation (10), 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.

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 polynomials to these two relationships we find that for BiCGSTAB, there is an approximate linear relationship

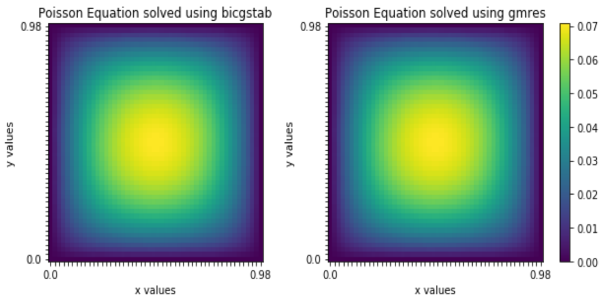

FIG. 1. Solved with square matrices of dimension $N = 50$ using OpenCL.

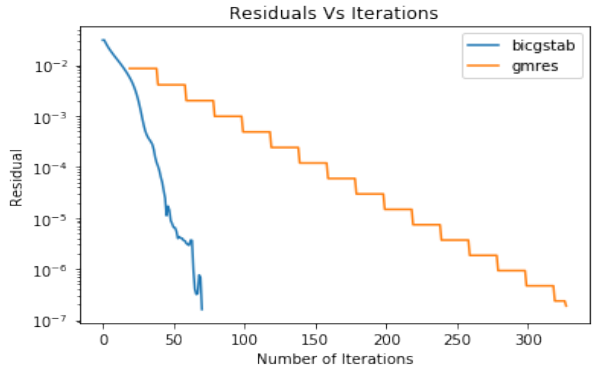

FIG. 2. Evolution of the residual error per iteration, until convergence. Solved with square matrices of dimension $N = 50$.

between iterations and mesh size, as expected, with a gradient of 1.28 (3sf). For GMRES, we note that for relationships of the form,

$$y = Cx^p \tag{18}$$

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

$$\log(y) = p\log(x) + \log(C) \tag{19}$$

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.

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 (5). 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

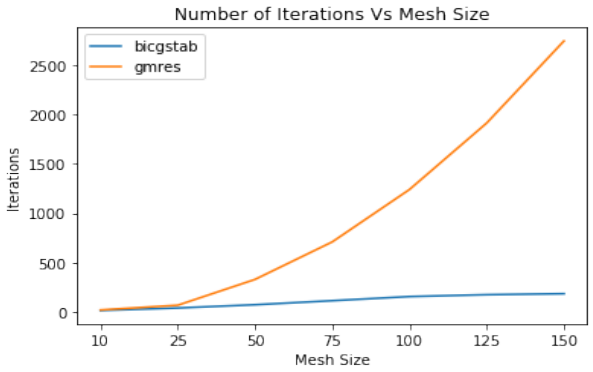

FIG. 3. 'Mesh Size' is the equivalent to the dimension N of the square matrix being considered, therefore larger mesh size, means finer meshes.

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.

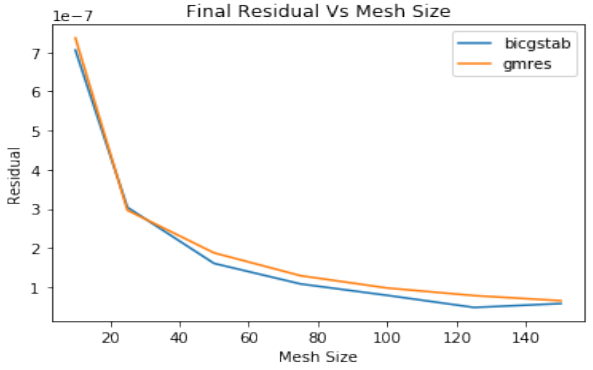

FIG. 4. Final residual error for increasingly fine meshes. Again, 'Mesh Size' is the equivalent to the dimension N of the square matrix being considered see fig. (3)

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

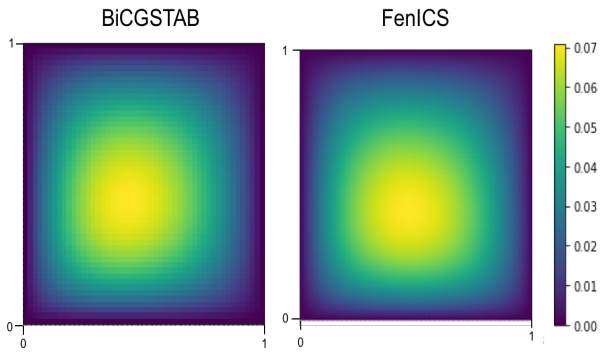

FIG. 5. Computed on a square matrix of dimension $N = 50$

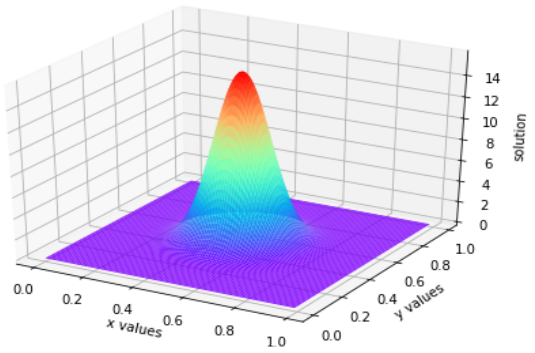

FIG. 6. Computed on a square matrix of dimension $N = 100$

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 (5) 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 $\sigma(x, y)$.

B. Diffusion

As noted in section II D, we expect there to be a bound defining the relationship between temporal and spatial step size (17). We can model this as a simple power relationship, in order to verify our conjecture,

$$\Delta t = \kappa \cdot (\Delta r)^2 \tag{20}$$

where κ is some coefficient we are aiming to find. As we have not solved (14) analytically, it's difficult to find a rigorous error estimate, however by calculating the difference in magnitude between the solutions at the final 2 time steps of a simulation, we can get an indication of divergence. For our initial conditions, we will use a 2D Gaussian-like distribution with a centre defined over the middle of the domain and variance of 0.1 in each dimension, however for convenience we will force boundary conditions of 0 (figure (6)),

As we shall see below, this proves to be an astute choice. We then proceed to evolve the system over 10 time steps in order to calculate the difference described above with κ values between 0.1 and 5.

From figure (7) we can see that our solution begins to diverge with increasing κ, and does so more rapidly with coefficients approximately ≥ 0.3. We can run simulations with this in mind in order to verify this behaviour.

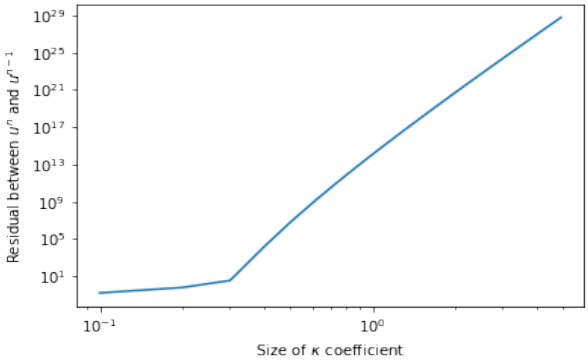

FIG. 7. We can see that solutions begin to diverge more rapidly after a critical point.

Figures (8) and (9) evolve the same system over 50 time steps, clearly demonstrating the dramatic effect that a poor choice of κ could have.

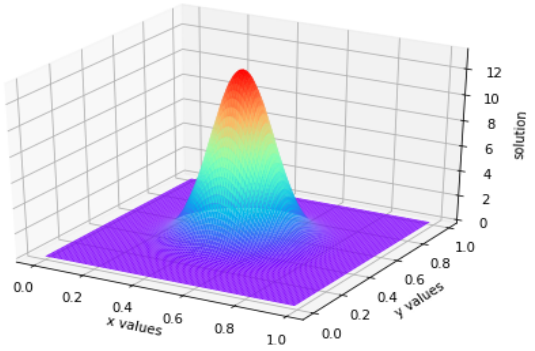

FIG. 8. Computed on a square matrix of dimension $N = 100$ with $\kappa = 0.2$, we can see that the amplitude of the initial Gaussian has slightly decayed in comparison to fig. (6)

With an appropriately chosen κ, we find that the amplitude of the initial Gaussian slowly decays (fig. (10)) as the system evolves over time.

We can also experiment with different initial condi-

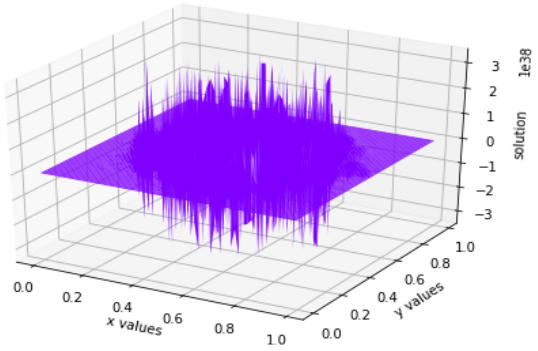

FIG. 9. Computed on a square matrix of dimension $N = 100$, with $\kappa = 1$. The solution has rapidly diverged.

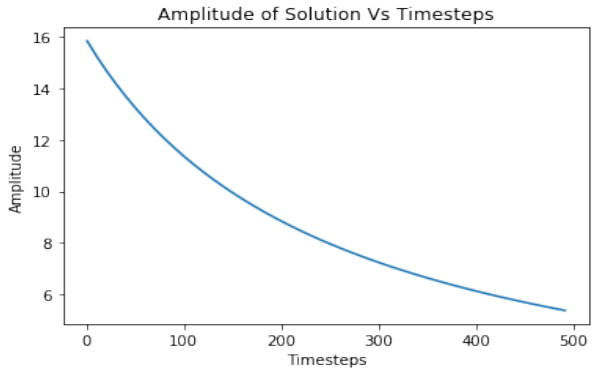

FIG. 10. Computed on a square matrix of dimension $N = 100$, with $\kappa = 0.2$. The amplitude of the solution slowly decays as the system evolves over time

tions. Instead of an initial Gaussian, we can repeat the above experiment with a square 'block' defined in the domain,

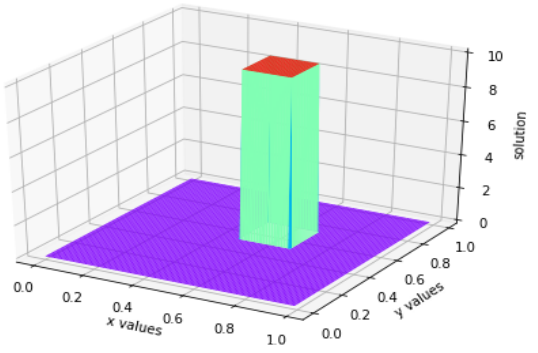

FIG. 11. Square block, with an initial amplitude of 10. Computed on a square matrix of dimension $N = 100$.

Evolving this system over time we see that it converges to something that looks a great deal like a Gaussian (figure (12)).

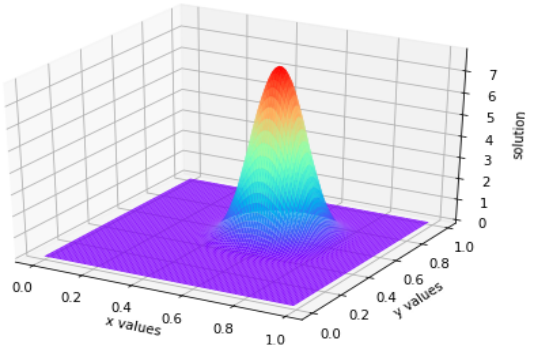

FIG. 12. Computed on a square matrix of dimension $N = 100$, with $\kappa = 0.2$, over 100 time steps. The amplitude of the solution decays over time, and it evolves into a Gaussian-style distribution.

We can understand this by extending some of the analysis of section II. Our system (14) is of the form,

$$\frac{du}{dt} = Au \tag{21}$$

Where $u(t)$ is a vector in $\mathbb{R}^{dim^2}$ and A is a constant real matrix of size $(dim^2) \times (dim^2)$. Where dim refers to the number of discrete points in one axis of the spatial domain.

We then note that we can take,

$$u(t) = \exp(\lambda t)v \tag{22}$$

to be a solution of our system if v is an eigenvector of A with an eigenvalue of λ, this works because $\exp(\lambda t)$ is an eigenfunction of the time derivative operator $\frac{d}{dt}$ with an eigenvalue of λ. If A is diagonilisable, then the general solution at a given timestep is given by a linear combination of exponential terms like those above, over the eigenvectors.

$$u(t) = \Sigma_{k=0}^{n} C_k \exp(\lambda_k t)v_k \tag{23}$$

The fact that the solutions of our system are written in terms of exponentials, helps to explain why both initial conditions examined converged to similar shapes during the simulation.

Finally, we can benchmark our solution to see how increasingly fine mesh sizes effect computation time demonstrated in figure (13). In order to compute this graph, we run the first iteration of the simulation 10 times for different mesh sizes, calculating an average and standard deviation for the the computation time. The extremely large error bars and relatively flat relationship indicate that the OpenCL kernel we are using is I/O bound in terms of performance, at least for the limited number

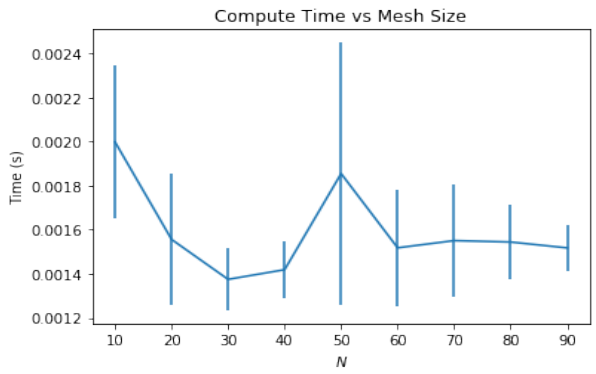

FIG. 13. Computed on square matrices of dimension N.

of mesh sizes we are able to examine with our computational resources. This is likely due to it's reliance on repeated global memory accesses in our kernel (c.f. adjoining codes).

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 accelerated our implementation using the OpenCL framework, finding BiCGSTAB outperformed GMRES for this problem. We validated our result by considering a different implementation of the same problem via FenICS. We proceeded to extend our analysis, considering a simple model of diffusion which made use of the same spatial discretisation. We found there to be a bound in the relationship between temporal and spatial step size in order to ensure convergence, which we empirically estimated. Furthermore, we found that solutions converged over time to Guassian-like shapes, likely due to the fact that solutions to the system we examined can be written in terms of exponentials. Extensions to this work could focus on preconditioning the problem in order to improve the Krylov methods examined for the Poisson equation. Additionally, the OpenCL kernel was demonstrated to be I/O bound in terms of it's performance, future work could take advantage of local memory caching in order to improve performance.

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

[2] U. Cetin, Diffusion transformations, Black-Scholes equation and optimal stopping (2018), arXiv:1701.01085.

[3] 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.

[4] 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.

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

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

[7] D. Dutykh, How to overcome the CourantFriedrichsLewy condition of explicit discretizations? (2016), arXiv:1611.09646.

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

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

[10] 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.