# HPC Project Part 2 for SN: 15050727

## 1 Introduction

This report follows on from part 1. In part 2, instead of just solving a Poisson equation in space, a parabolic Poisson equation is solved, which is time dependent. Time dependent PDEs are very useful for modelling process in a variety of domains including the diffusion of heat as well as the dynamics of financial markets [1, p. 77][2].

This report seeks to explore how the level of time discretisation effects the solutions produced. In addition, methods for improving the performance of finite difference method software is explored. This is motivated by the fact that numerical methods software must be able to calculate solutions for highly discretised domains in a relatively short amount of time if such methods are to be used to calculate precise solutions to problems in science and engineering (e.g. Dupros et. al.[3]).

## 2 Method

The equation introduced in part 1 now varies in time as described by Equation (1).

$$\frac{\partial u}{\partial t} = \nabla \cdot (\sigma(x, y)\nabla)\, u \tag{1}$$

Where t is time, x and y are spatial coordinates and $\sigma$ is a random field. At time $t = 0$, $u(x, y) = g(x, y)$ where $g(x, y)$ is some arbitrary initial distribution. The boundaries are set equal to zero for all values of t. The RHS of Equation (1) can be discretised in space with the equivalent discretisation scheme used in part 1. Additionally, to discretise in the time dimension, the LHS side of Equation (1) can now also approximated using forward difference discretisation [1, p. 79].

$$\frac{u(x, y, t + \Delta t) - u(x, y, t)}{\Delta t} \approx \nabla \cdot (\sigma(x, y)\nabla)\, u \tag{2}$$

Where $\Delta t$ is a small time step that approximates an infinitesimal difference. Given an initial value of $u(x, y, t)$, one must calculate the value of $u(x, y, t)$ at a future time step $u(x, y, t + \Delta t)$. This is accomplished by rearranging Equation (2).

$$u(x, y, t + \Delta t) \approx \Delta t(\nabla \cdot (\sigma(x, y)\nabla)\, u) + u(x, y, t) \tag{3}$$

Given Equation (3) and the spatial discretisation introduced in part 1, the calculation implemented in the experiments presented to calculate future time

steps is defined by Equation (4).

$$
\begin{aligned}
& u(x_i, y_j, t + \Delta t) \\
& \approx \Delta t \Bigg( \frac{\left(\sigma(x_{i+1/2}, y_j) \frac{(u(x_{i+1}, y_j) - u(x_i, y_j))}{h}\right) - \left(\sigma(x_{i-1/2}, y_j) \frac{(u(x_i, y_j) - u(x_{i-1}, y_j))}{h}\right)}{h} \\
& \quad + \frac{\left(\sigma(x_i, y_{j+1/2}) \frac{(u(x_i, y_{j+1}) - u(x_i, y_j))}{h}\right) - \left(\sigma(x_i, y_{j-1/2}) \frac{(u(x_i, y_j) - u(x_i, y_{j-1}))}{h}\right)}{h} \Bigg) \\
& \quad + u(x_i, y_j, t)
\end{aligned}
\tag{4}
$$

The x and y coordinates can be calculated concurrently as was discussed in part 1. However, given that each calculated time step depends on the previous time step, time steps must be calculated in series. Furthermore, in part 2 the discretisation scheme described was implemented explicitly. As a result, the stability of the finite difference solver is dependent on the size of $\Delta t$ [1, p. 79]. Using a von Neumann stability analysis, the required values of t for a stable solver are given by Equation (5) [1, p. 79].

$$\Delta t \leq \frac{h^2}{2k} \tag{5}$$

Where k is a value that is dependent on the divergence of $\sigma(x, y)$. If $\Delta t$ is not below this threshold then the solver is prone to divergence. When implementing the described equations on multi-core CPUs or GPUs, constantly retrieving grid point values will be costly for the performance of the software implementations if the grid points described exist on off-chip memory (e.g. Holewinski et. al. [4]). A naive software implementation of the spatial discretisation described will have to retrieve most grid point values 5 times per time step. If the grid points loaded from off-chip memory can be temporarily cached on the private memory of the GPU (in OpenCL terms), then the number of computationally expensive global memory accesses can be reduced [4].

Furthermore, some OpenCL implementations can automatically attempt to vectorise OpenCL kernels [5, p. 134]. However, performance improvements can be devised that instead explicitly vectorise OpenCL calculations [5, p. 102]. This approach was taken in the results presented in an attempt to improve the performance of the finite difference software created.

The OpenCL kernel created used the double8 vector data type to perform the computations in Equation (3). This was accomplished by loading eight grid points in j direction for each variable in Equation (3). Given that the grid points in the j direction are right next to each other, the double8 vectors significantly overlap in the j direction. This means that once values are loaded into a double8 vector from global memory, some adjacent vectors can access this data from the vector next to it that is stored in private memory. Additionally,

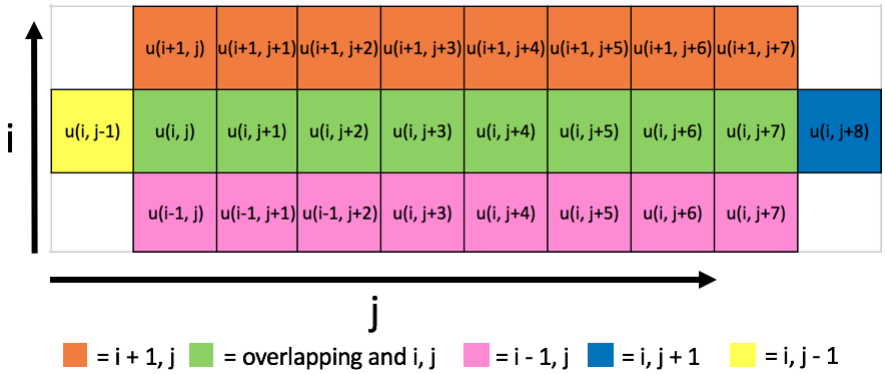


*Figure 1: Representation of the 5 vectors used for the optimised implementation. The blue and yellow vectors significantly overlap (represented by the green overlapping colour), meaning values in this vector can be recycled in the overlapping vectors to reduce global memory access. An additional set of vectors in an i+1 position could be added on top for further overlapping an recycling of data.*

computations could be performed two at a time in the i direction by the kernel to create an additional overlap of data between vectors, allowing the number of global memory accesses to be further reduced. However, this was not implemented in the results presented. A schematic diagram describing the caching scheme can be seen in Figure 1.

# 3 Results

Heterogeneous OpenCL code was written to apply the spatial discretisation in space, which was then applied in series for each time step. The result of this function being applied to a central block of ones over 1000 time steps can be seen in Figure 2 for a 64×64 grid. Furthermore, the action of the solver produced on a random field $g(x, y)$ can be seen in Figure 3. A physically plausible diffusion pattern can be seen as the number of time steps increases as expected. Given the stochastic nature of the $\sigma$ value in Equation (5), there is some variability in the lowest potential value of $\Delta t$ for a stable finite difference solver.

It was estimated that for a sigma function with a mean of 1 and standard deviation of 0.2, that an estimate of the highest possible value of $\Delta t$ for a $64 \times 64$ grid would be equal to approximately $2 \times 10^{-5}$. This was confirmed by performing experiments with different values of $t$ and observing the resulting solutions. A $\Delta t$ value in the region of $2 \times 10^{-5}$ or larger would cause the solver to diverge, while a value of $1 \times 10^{-5}$ seemed to generally produce a stable solver.

A number of benchmarking experiments were performed to observe how the vectorised implementation improves on the results produced by the naive im-

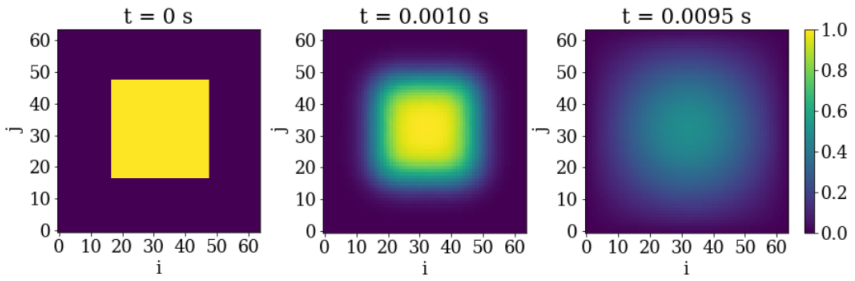


*Figure 2: Action of the finite difference method on a central concentration of ones in the input g(x, y) matrix. Axes labels refer to the indices of the grid points. A 64×64 grid was used for this particular experiment. The domain used was* $(x, y) \in [0, 1] \times [0, 1]$. $\Delta t$ *was set to equal 0.00001. Boundaries are not shown (they are equal to zero).*

plementation. Such experiments were completed by varying the number of grid points and the value of $\Delta t$, while measuring the time taken for the two implementations to apply the finite difference scheme. Experiments were performed in sets of 100, with the average value of the time elapsed in the experiments being presented in Table 1. Further, 1000 forward time steps were implemented in each case. It can be seen that as the size of the discretisation grid grows, the difference in performance seems to increase between the two methods. There are some drawbacks to the current optimised implementation, namely that it only supports grid sizes that are a factor of 8 in both dimensions. Furthermore, the vectorised code is significantly less readable.

| $\Delta t$ (s) | N | Time Elapsed for Optimised (s) | Time Elapsed for Naive (s) |
|---|---|---|---|
| $10^{-4}$ | 16 | 0.016 | 0.016 |
| $10^{-5}$ | 32 | 0.023 | 0.032 |
| $10^{-5}$ | 64 | 0.043 | 0.095 |
| $10^{-6}$ | 128 | 0.12 | 0.35 |
| $10^{-6}$ | 256 | 0.53 | 1.4 |

*Table 1: N refers to the dimensions of the square grid used, that is an N×N would be used for a given value of N. 1000 forward time steps were implemented in each case.*

During development, it was empirically observed that vectorisation of the solver caused a 2× to 3× improvement in the speed of solver. However, attempts to cache grid points in private memory did not have much of an effect of the performance of the optimised implementation. There are two hypotheses for why this caching method did not improve performance. First, modern CPUs automatically make caches when performing calculations, so perhaps the method used doesn't make an improvement on such automatic caching mechanisms [6]. Alternatively, perhaps the software created is not actually accessing values from private memory as it is designed to.

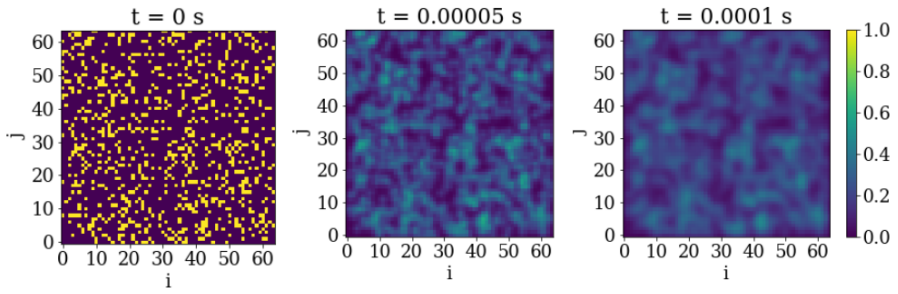


*Figure 3: Action of the finite difference method on a random field input for the g(x, y) matrix with 20% of the values being set to one. Axes labels again refer to the indices of the grid points. The same domain was used as in the central slab experiment. $\Delta t$ was again set equal to 0.00001.*

# 4 Conclusion

An explicit finite difference method was implemented with OpenCL to solve a parabolic time-dependent Poisson equation. The solver crucially depended on using small values of $\Delta t$ that depend on the corresponding discretisation in space. When running the software created on multicore CPUs, the best performance was obtained by using explicit vectorisation. It was also observed that this improvement became more dramatic as the number of grid points used increased.

In the future, different methods of caching data could be investigated to further improve performance. Implementations that target GPU cards could present the most effective method for implementing the finite difference scheme presented efficiently [3]. In order to get around the dependence on a small value $\Delta t$ for a stable solver (and the corresponding performance drawbacks), the implicit Euler method could be used, which would also improve the space complexity of the implementation presented [7, p. 363].

# References

[1] DM Causon and CG Mingham. *Introductory finite difference methods for PDEs*. Bookboon, 2010.

[2] Guy Barles and Halil Mete Soner. Option pricing with transaction costs and a nonlinear black-scholes equation. *Finance and Stochastics*, 2(4):369–397, 1998.

[3] Fabrice Dupros, Florent De Martin, Evelyne Foerster, Dimitri Komatitsch, and Jean Roman. High-performance finite-element simulations of seismic

wave propagation in three-dimensional nonlinear inelastic geological media. *Parallel Computing*, 36(5-6):308–325, 2010.

[4] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[5] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[6] Cedric Nugteren. Tutorial: Opencl sgemm tuning for kepler. https://cnugteren.github.io/tutorial/pages/page4.html. Accessed: 2019-03-28.

[7] J.D. Hoffman and S. Frankel. *Numerical Methods for Engineers and Scientists, Second Edition,*. Taylor & Francis, 2001.