A Technical Supplement for "An Implementation of the All-At-Once Method to EPDEs".

Anthony Goddard

I. Introduction

This technical report sets out to further explore some of the topics mentioned in the work that this is a supplement to (see [1]). We will mainly talk about Graphics Processing Unit (GPU) implementations instead of Central Processing Unit (CPU) implementations as CPU implementations were discussed in detail in [1]. We use the same notation from [1] and understand that n is the number of nodes in the spatial discretisation and ℓ is the number of nodes in the temporal discretisation. We will finish off with a discussion of the all-at-once method in the context of *non-linear partial differential equations*.

II. From CPU to GPU

In order to "port" the program from a CPU-core paradigm to a GPU-core paradigm we have to be mindful of a few points. The first, and most important point, is that memory is *severely* scarce. The second is that MPI largely abstracted us from the delegation of memory *location* in the CPU-bound case which we now have contend with in our GPU application. We are going to implement the GPU version of the all-at-once implementation using OpenCL. The OpenCL memory model is shown in Figure 1. Access to private memory is the fastest but is very limited in capacity and global memory is the slowest but is plentiful.

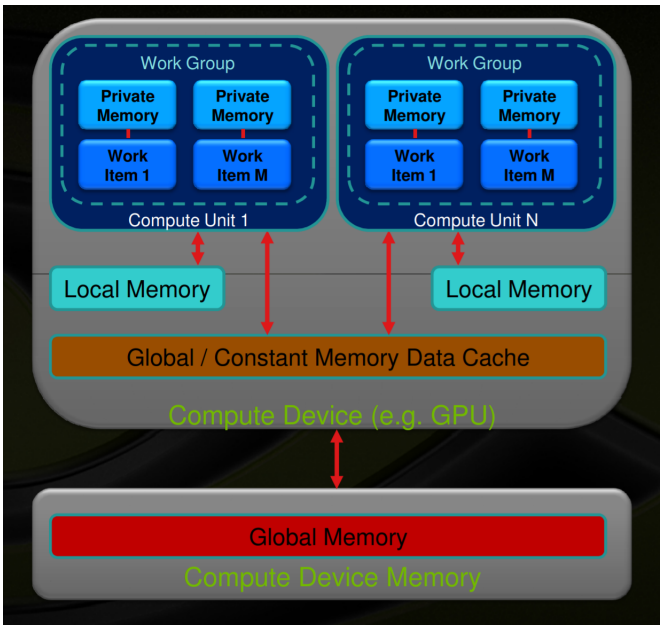

Fig. 1. OpenCL memory model. *Courtesy of nVidia.*

Advisor: Andrew Wathen, Mathematical Institute, University of Oxford.

III. Implementation

In [1] two implementations were discussed. One that used the Fast Fourier Transform (FFT) and another which did not. We found the implementation that utilised the FFT did not scale well and we believe this is a result of the amount of communication required in the execution of this implementation. When referring to the CPU implementation in this report we will be referring to the implementation that did not use the FFT.

There are two main components to the implementation discussed in [1]: matrix-vector multiplication and the solution of a tridiagonal system. Because the matrix $(U \otimes \mathbb{I}_n)$ has a very special structure we do not need to alter the matrix-vector multiplication algorithm, only the implementation of such algorithm.

Solution of Tridiagonal Systems

In [1] we used the Thomas algorithm to solve a block tridiagonal system. Since access to local memory is relatively cheap in the context of CPU programming we do not need to take this cost consideration. However, since accessing local memory on a GPU device is very expensive it is now of great concern.

In the CPU implementation we used the Thomas algorithm, which requires a few accesses to local memory. With the knowledge that local memory access is quite expensive we consider other algorithms as well. We consider Parallel Cyclic Reduction (PCR) and Thomas-PCR which are both described in [2]: The Thomas algorithm requires $\mathcal{O}(n)$ operations while PCR requires $\mathcal{O}(n \log n)$ operations, but the number of times PCR needs access to local memory is smaller. We also consider a PCR-Thomas algorithm.

PCR-Thomas, Thomas-PCR?

The difference between PCR-Thomas and Thomas-PCR is that PCR-Thomas uses PCR to split the system into smaller systems then uses Thomas on those smaller systems and Thomas-PCR uses Thomas to split the systems into smaller systems then uses PCR on those smaller systems.

IV. Results

Choice of Tridiagonal Solver

Due to the time constraints imposed we need to choose a tridiagonal solver and stick with it. Especially since solving

a tridiagonal system is a small part of the application of the preconditioner. In order to decide on which method we carry our some preliminary experiments. The results of these experiments are shown in Figure 2. We do not consider Thomas-PCR as it only performed slightly better than vanilla PCR in the case where $n = 256$. We believe that the reason behind the decline in efficiency for larger problem sizes is that this method puts pressure on register allocation, which is exactly what is said in [2]. We can see from Figure 2 there is a subsystem size "sweet spot". If we apply PCR until the subsystem size reduces to 4 then apply the Thomas algorithm to each of systems in PCR-Thomas we see quite significant reductions in the time taken to solve the tridiagonal system. Therefore, we will enlist this method to solve the tridiagonal system.

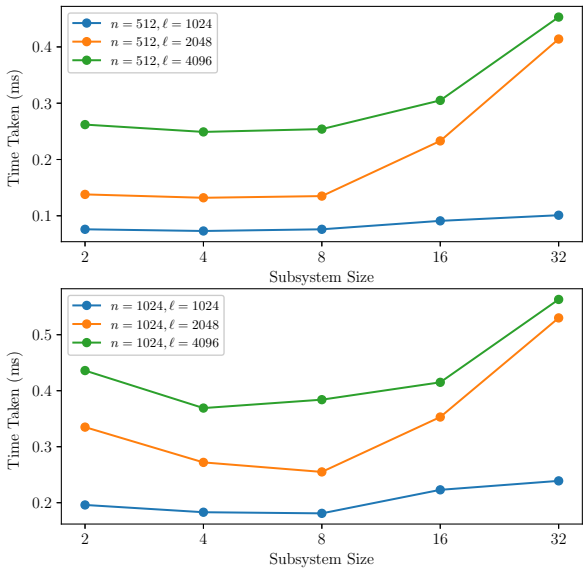

Fig. 2. The relationship between the point at which we switch to the Thomas algorithm during the hybrid PCR-Thomas algorithm and the time taken to solve a triangular system.

Solving the All-At-Once Problem

For $n = 512$ and $\ell = 1024$ the time taken to solve the all-at-once formulation of the heat equation was ~ 1.4 seconds. From referring to Table 4.1 of [1] we can see that this is a $9\times$ speed-up. Improvements of this magnitude, or thereabout, were observed for all values of n and ℓ tested. We believe we could achieve faster times by investigating alternative memory management schemes in the matrix-vector multiplication routine.

This highlights the bottleneck in GPU-core bound calculations. If the GPU has to request data from the global cache we see immense reductions in performance.

V. Non-Linear PDEs

We now turn our attention to the non-linear PDE

$$\frac{\partial u}{\partial t} = \nabla \cdot (u \nabla u). \tag{1}$$

In order to solve this system we use Newton's method. That is, we consider the residual operator

$$R(u) = u_t - \nabla \cdot (u \nabla u) \tag{2}$$

and the derivative of such residual operator

$$R'(u)(\delta u) = \delta u_t - \nabla \cdot (\delta u \nabla u + u \nabla \delta u) \tag{3}$$

in the equation

$$R'(u)\delta u = -R(u). \tag{4}$$

By discretising in space using linear finite elements, discretising in time using backwards Euler and making the substitution

$$\delta u^{k,\sigma} = \sum_j \phi_j \ \delta U_j^{k,\sigma}, \tag{5}$$

where ϕ_j is a finite element basis function, k is the time step index and σ is the Newton iterate index we obtain

$$\sum_j \big[(M_{ij} + \tau\langle \phi_i, \phi_j \nabla u^{k,\sigma}\rangle + \tau\langle \nabla\phi_i, u^{k,\sigma}\nabla\phi_j\rangle)\delta U_j^{k,\sigma} - M_{ij}\delta U_j^{k-1,\sigma}\big] = -\langle \phi_i, u^{k,\sigma}\rangle + \langle \phi_i, u^{k-1,\sigma}\rangle - \tau\langle \nabla\phi_i, u^{k,\sigma}\nabla u^{k,\sigma}\rangle.$$

From reading [1] we can see that this can be cast into a monolithic system on the form

$$\mathcal{A}(\mathbf{u}^\sigma)\delta\mathbf{U}^\sigma = \mathbf{b}^\sigma, \tag{6}$$

where

$$\mathcal{A}(\mathbf{u}^\sigma) = \begin{bmatrix} A_1^\sigma & & & \\ -M & A_2^\sigma & & \\ & \ddots & \ddots & \\ & & -M & A_\ell^\sigma \end{bmatrix}. \tag{7}$$

The important distinction between the linear and non-linear case is that

$$A_k^\sigma = M_{ij} + \tau\langle \phi_i, \phi_j \nabla u^{k,\sigma}\rangle + \tau\langle \nabla\phi_i, u^{k,\sigma}\nabla\phi_j\rangle. \tag{8}$$

That is, the diagonal entries depend on the current Newton iterate. These diagonal entries have to be reformulated every newton iterate.

VI. Conclusion

Throughout this report we have discussed implementations and numerical results of the extensions suggested in [1] using OpenCL. We would have used CUDA if we were to investigate this further. Unfortunately, the development of this code was done on an AMD GPU. We have also discussed an all-at-once formulation of a non-linear PDE, some further numerical experiments would be required to determine the validity of such a method.

References

[1] A. J. Goddard. An Implementation of the All-At-Once Method to Evolutionary PDEs. *M. Sc. Thesis. 2018.*

[2] E. László, M. Giles and J. Appleyard. Manycore Algorithms for Batch Scalar and Block Tridiagonal Solvers