Chapter 5

Numerical Simulation of the Frank-Kamenetskii PDE: GPU vs. CPU Computing

Charis Harley

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/46463

1. Introduction

The efficient solution of the Frank-Kamenetskii partial differential equation through the implementation of parallelized numerical algorithms or GPUs (Graphics Processing Units) in MATLAB is a natural progression of the work which has been conducted in an area of practical import. There is an on-going interest in the mathematics describing thermal explosions due to the significance of the applications of such models - one example is the chemical processes which occur in grain silos. Solutions which pertain to the different geometries of such a physical process have different physical interpretations, however in this chapter we will consider the Frank-Kamenetskii partial differential equation within the context of the mathematical theory of combustion which according to Frank-Kamenetskii [16] deals with the combined systems of equations of chemical kinetics and of heat transfer and diffusion. A physical explanation of such a system is often a gas confined within a vessel which then reacts chemically, heating up until it either attains a steady state or explodes.

The focus of this chapter is to investigate the performance of the parallelization power of the GPU vs. the computing power of the CPU within the context of the solution of the Frank-Kamenetskii partial differential equation. GPU computing is the use of a GPU as a co-processor to accelerate CPUs (Central Processing Units) for general purpose scientific and engineering computing. The GPU accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU. The reason why the application is seen to run faster is because it is using the extreme parallel processing power of the GPU to boost performance. A CPU consists of 4 to 8 CPU cores while the GPU consists of 100s of smaller cores. Together they operate to crunch through the data in the application and as such it is this massive parallel architecture which gives the GPU its high compute performance.

The methods which will be investigated in this research are implicit methods, such as the Crank-Nicolson method (CN) and the Crank-Nicolson method incorporating the Newton method (CN_N) [26]. These algorithms pose a serious challenge to the implementation of

parallelized architecture as we shall later discuss. We will also consider Rosenbrock methods which are iterative in nature and as was indicated in Harley [22] showed drastically increased running times as the time period over which the problem was considered increased. Further pitfalls which arise when trying to obtain a solution for the partial differential equation in question when using numerical techniques is firstly the singularity which exists at $x = 0$. This complexity may be dealt with through the use of a Maclaurin expansion which splits the problem into two cases: $x = 0$ and $x \neq 0$.

The second hurdle is the nonlinear source term which may be dealt with using different techniques. In this chapter we will implement the Newton method which acts as an updating mechanism for the nonlinear source term and in so doing maintains the implicit nature of the scheme in a consistent fashion. While the incorporation of the Newton method leads to an increase in the computation time for the Crank-Nicolson difference scheme (see [22]) there is also an increase in the accuracy and stability of the solution. As such we find that the algorithms we are attempting to employ in the solution of this partial differential equation would benefit from the processing power of a GPU.

In this chapter we will focus on the implementation of the Crank-Nicolson implicit method, employed with and without the Newton method, and two Rosenbrock methods, namely ROS2 and ROWDA3. We consider the effectiveness of running the algorithms on the GPU rather than the CPU and discuss whether these algorithms can in fact be parallelized effectively.

2. Model

The steady state formulation of the equation to be considered in this chapter was described by Frank-Kamenetskii [16] who later also considered the time development of such a reaction. The reaction rate depends on the temperature in a nonlinear fashion, generally given by Arrhenius' law. This nonlinearity is an important characteristic of the combustion phenomena since without it the critical condition for inflammation would disappear causing the idea of combustion to lose its meaning [16]. Thus, in the case of a thermal explosion, the Arrhenius law is maintained by the introduction of the exponential term which acts as a source for the heat generated by the chemical reaction. As such we are able to write an equation modelling the dimensionless temperature distribution in a vessel as

$$\frac{\partial u}{\partial t} = \nabla^2 u + \delta e^{u/(1+\epsilon u)} \tag{1}$$

where u is a function of the spatial variable x and time t and the Frank-Kamenetskii parameter δ is given by

$$\delta = \frac{Q}{\lambda} \frac{E}{R{T_0}^2} r^2 \kappa \alpha e^{\left(-\frac{E}{RT_0}\right)}. \tag{2}$$

The value of the Frank-Kamanetskii parameter [16] δ is related to the critical temperature at which ignition of a thermal explosion takes place and is thus also referred to as the critical value. At values below its critical value δ_{cr} a steady state is reached for a given geometry and set of boundary conditions whereas an explosion ensues for values above it. The Laplacian operator takes the form

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{k}{x}\frac{\partial}{\partial x}, \qquad 0 < x < 1 \tag{3}$$

where k is indicative of the shape of the vessel within which the chemical reaction takes place: $k = 0, 1$ and 2 represent an infinite slab, infinite circular cylinder and sphere, respectively. The dimensionless parameter $\epsilon = \frac{RT_0}{E}$ is introduced as the critical activation parameter. To be able to speak of combustion the condition $\epsilon \ll 1$ must be satisfied due to the fact that the ambient temperature can normally be seen as much smaller in magnitude than the ignition temperature [16]. Equation (1) for $\epsilon = 0$ was derived by Frank-Kamenetskii [16]. Further work was done by Steggerda [31] on Frank-Kamenetskii's original criterion for a thermal explosion showing that a more detailed consideration of the situation is possible. For small x a solution was derived for the cylindrical system by Rice [27], Bodington and Gray [6] and Chambré [10]. While the steady state case - often termed the Lane-Emden equation of the second kind - has been considered extensively, the time dependent case is also of import and has been studied in [2], [32] and [33].

In this chapter we consider numerical solutions for equation (1) modelling a thermal explosion within a cylindrical vessel, i.e. $k = 1$. A thermal explosion occurs when the heat generated by a chemical reaction is far greater than the heat lost to the walls of the vessel in which the reaction is taking place. As such this equation is subject to certain boundary conditions given at the walls of the vessel. The appropriate boundary conditions for this problem are

$$u(x, 0) = 0, \tag{4}$$

$$\frac{\partial u}{\partial x}(0, t) = 0, \qquad (a) \qquad\qquad u(R, t) = 0 \qquad (b) \tag{5}$$

where $R = 1$ is the radius of the cylinder. The boundary conditions (4) and (5) imply that the temperature at the vessel walls is kept fixed and the solution is symmetric about the origin.

Frank-Kamenetskii [16] obtained a steady state solution to this problem with $\epsilon = 0$. Zeldovich et al. [34] considered similarity solutions admitted by (1) for $k = 1$ that exhibit blow-up in finite time. These kinds of solutions, while noteworthy, have limited significance due to the restricted form of the initial profiles compatible with the similarity solutions. These solutions correspond to very special initial conditions for the temperature evolution profile, limiting the degree to which results obtained in this manner are applicable. This disadvantage has been noted by Anderson et al. [3] while analytically investigating the time evolution of the one-dimensional temperature profile in a fusion reactor plasma. A solution which also models blow-up in finite time has been obtained by Harley and Momoniat [18] via nonlocal symmetries of the steady-state equation.

In Harley [21] a Crank-Nicolson- and hopscotch scheme were implemented for equation (1) subject to (4) and (5) where $\delta = 1$ and $\epsilon = 0$. The nonlinear source term was kept explicit when the Crank-Nicolson method was employed, as commented on by Britz et al. [9] in whose work the nonlinear term was incorporated in an implicit manner in a style more consistent with the Crank-Nicolson method. Britz et al. [9] implemented the Crank-Nicolson scheme with the Newton iteration and showed that it outperformed the explicit implementation of the nonlinearity as in [21] in terms of accuracy. However it does require more computer time as would be expected.

In recent work (see [22]) the Crank-Nicolson method was implemented with the Newton iteration as done by Britz et al. [9] by computing a correction set in each iteration to obtain

approximate values of the dependent variable at the next time step. The efficiency of the Crank-Nicolson scheme, hopscotch scheme (both of these methods were implemented with an explicit and then an implicit discretisation of the source term) and two versions of the Rosenbrock method were compared [22]. Using the `pdepe` function in MATLAB and the steady state solution obtained by Frank-Kamenetskii [16] as a means of comparison, it was found that the incorporation of the Newton method for the Crank-Nicolson- and hopscotch scheme led to increased running times as T, where $0 \leq t \leq T$, increased.

Furthermore, it was shown that while the Crank-Nicolson- and hopscotch method (with or without the implementation of the Newton method) performed well in terms of accuracy for $T = 0.3$ and 0.5, they were in fact able to outperform `pdepe` at $T = 4$. The Rosenbrock methods employed (ROS2 and ROWDA3) performed similarly with regards to accuracy, however showed almost an exponential increase in their running times as T increased, indicating that using the Crank-Nicolson- or hopscotch scheme may be more efficient. Thus, given that the Rosenbrock methods performed even poorer with regards to running time, it seems reasonable to suggest that implementing the Crank-Nicolson- or hopscotch scheme with a Newton iteration is most ideal. The Crank-Nicolson method using the Newton method as a means of maintaining the implicit nature of the source term in the difference scheme has been used by Anderson and Zienkiewicz [2]. In Harley [21] and Abd El-Salam and Shehata [1] the discretisation of the exponential terms were kept explicit, thereby removing the nonlinearity.

As a consequence of these findings and due to the complexity created by the nonlinear source term which serves a critical function in the model, further work regarding faster algorithms for the solution of such an equation are of interest. This chapter will not consider the hopscotch scheme directly as an appropriate method for the solution of the Frank-Kamenetskii partial differential equation due to work done by Feldberg [15] which indicated that for large values of $\beta = \frac{\triangle t}{\triangle x^2}$ the algorithm produces the problem of propagational inadequacy which leads to inaccuracies - similar results were obtained in [22]. Given the improved accuracy of the Crank-Nicolson method incorporating the Newton method [22] - the order of the error for this method is $O(\triangle t^2)$ which is only approximately the case for the Crank-Nicolson method without the Newton iteration incorporated [9] - it seems more fitting to consider an improvement in the computing time of this method. Hence a consideration of such an improvement on the algorithm's current running time will be the focus of this chapter. The means by which we wish to accomplish this is through the use of the the Parallel Computing Toolbox in MATLAB. It is hoped that this is the next step towards creating fast and effective numerical algorithms for the solution of a partial differential equation such as the one originating from the work of Frank-Kamenetskii [16].

3. Executing MATLAB on a GPU

The advantage of using the Parallel Computing Toolbox in MATLAB is the fact that it allows one to solve computationally and data-intensive problems using multicore processors, GPUs, and computer clusters. In this manner one can parallelize numerical algorithms, and in so doing MATLAB applications, without CUDA or MPI programming. Parallelized algorithms such as `parfor`, used within the context of what is usually a `for` loop, allows you to offload

work from one MATLAB session (the client) to other MATLAB sessions, called workers. You can use multiple workers to take advantage of parallel processing and in this way improve the performance of such loop execution by allowing several MATLAB workers to execute individual loop iterations simultaneously. In this context however we are not able to implement in-built MATLAB functions such as `parfor` due to the numerical algorithms which we have chosen to consider. The CN- and CN_N method, both implicit, loop through the index m until $t_0 + m\triangle t = T$. These iterative steps are not independent of each other, i.e to obtain data at the $m + 1^{th}$ step the data at the m^{th} step is required. In a similar fashion the ROS2 and ROWDA3 methods also iterate through dependent loops to obtain a solution. As such we attempt to run the code directly on the GPU instead of the CPU in order to decrease the running time of the algorithms.

The Parallel Computing Toolbox in MATLAB allows one to create data on and transfer it to the GPU so that the resulting GPU array can then be used as an input to enhance built-in functions that support them. The first thing to consider when implementing computations on the GPU is *keeping* the data on the GPU so that we do not have to transfer it back and forth for each operation - this can be done through the use of the `gpuArray` command. In this manner computations with such input arguments run on the GPU because the input arguments are already in the GPU memory. One then retrieves the results from the GPU to the MATLAB workspace via the `gather` command. Having to recall the results from the GPU is costly in terms of computing time and can in certain instances make the implementation of an algorithm on the GPU less efficient than one would expect. Furthermore, the manner in which one codes algorithms for GPUs is of vital importance given certain limitations to the manner in which functions of the Toolbox may be implemented (see [25]). More importantly however, is whether the method employed can allow for the necessary adjustments in order to improve its performance. In this chapter we will see that there are some problems with implementing the kind of algorithms considered here on the GPU.

In this chapter we are employing MATLAB under Windows 7 (64 bits) on a PC equipped with an i7 2.2 GHz processor with 32 GB of RAM.

3.1. Crank-Nicolson implicit scheme

We will implement the Crank-Nicolson method while maintaing the explicit nature of the nonlinear source term and also apply the method by computing a correction set in each iteration to obtain approximate values of the dependent variable at the next time step through the use of the Newton method [26]. The methodology will be explained briefly here; the reader is referred to [7–9] for clarification.

When implementing the Crank-Nicolson method we employ the following central-difference approximations for the second-and first-order spatial derivatives respectively

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{\triangle x^2}, \tag{6}$$

$$\frac{\partial u}{\partial x} \approx \frac{u_{n+1}^m - u_{n-1}^m}{2\triangle x} \tag{7}$$

while a forward-difference approximation

$$\frac{\partial u}{\partial t} \approx \frac{u_n^{m+1} - u_n^m}{\triangle t} \tag{8}$$

is used for the time derivative. We implement a Crank-Nicolson scheme by approximating the second-derivative on the right-hand side of (1) by the implicit Crank-Nicolson [12] approximation

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{2\triangle x^2} + \frac{u_{n+1}^m - 2u_n^m + u_{n-1}^m}{2\triangle x^2}. \tag{9}$$

In a similar fashion the first-derivative on the right-hand side becomes

$$\frac{\partial u}{\partial x} \approx \frac{u_{n+1}^{m+1} - u_{n-1}^{m+1}}{4\triangle x} + \frac{u_{n+1}^m - u_{n-1}^m}{4\triangle x}. \tag{10}$$

To impose zero-shear boundary conditions at the edges we approximate the spatial first-derivative by the central-difference approximation (7) which leads to the following condition

$$u_{-1}^m = u_1^m. \tag{11}$$

As mentioned before the boundary condition (5a) at $x_0 = 0$ can pose a problem for the solution of equation (1). One could discretise it directly as a forward difference formula, such as the three-point approximation $-3u_0^m + 4u_1^m - u_2^m = 0$, and add this to the set of equations to solve when using the Crank-Nicolson scheme. Alternatively one could use the more accurate symmetric approximation, $u_{-1}^m = u_1^m$, which introduces a 'fictitious point' at $x = -\triangle x$. This however, would lead to another problem due to the singularity in the differential equation at $x_0 = 0$. Instead we choose to overcome this difficulty by using the Maclaurin expansion

$$\lim_{x \to 0} \frac{1}{x} \frac{\partial u}{\partial x} = \left. \frac{\partial^2 u}{\partial x^2} \right|_{x=0} \tag{12}$$

which simplifies the equation for the case $x_0 = 0$. It has been noted by Britz et al. [9] that using (12) turns out to be more convenient and accurate. Due to the fact that the point $x_0 = 0$ would lead to a singularity in equation (1) we structure the code to account for two instances: $x = 0$ and $x \neq 0$. Using (12) for equation (1) we attain the following approximation

$$\frac{\partial u}{\partial t} = 2\frac{\partial^2 u}{\partial x^2} + e^u \tag{13}$$

to equation (1) at $x_0 = 0$. This approximation has been taken into account in the system given by (16) below. Such an approximation has been used in many numerical algorithms. In Crank and Furzeland [13], for instance, they presented a modified finite-difference method which eliminates inaccuracies that occur in the standard numerical solution near singularities. The approximation has also been used by Harley and Momoniat [19] to generate a consistency criteria for initial values at $x_0 = 0$ for a Lane-Emden equation of the second-kind. From the equation under consideration (1) an initial condition for $u(x,t)$ is obtained at $x_0 = 0$ giving

the following

$$(1+2\beta)\,u_0^{m+1} - 2\beta u_1^{m+1} - \triangle t\delta e^{u_0^{m+1}} = (1+2\beta)\,u_0^m + 2\beta u_1^m + \triangle t\delta e^{u_0^m} \tag{14}$$

as the initial difference scheme with $\beta = \frac{\triangle t}{\triangle x^2}$. Implementing the difference approximations discussed above we obtain the general numerical scheme

$$-\frac{\lambda_n}{2}u_{n-1}^{m+1} + (1+\beta)\,u_n^{m+1} - \frac{\gamma_n}{2}u_{n+1}^{m+1} - \triangle t\delta e^{u_n^{m+1}} = \frac{\lambda_n}{2}u_{n-1}^m + (1-\beta)\,u_n^m + \frac{\gamma_n}{2}u_{n+1}^m + \triangle t\delta e^{u_n^m} \tag{15}$$

where $x_n = n\triangle x$ and $\beta = \frac{\triangle t}{\triangle x^2}$ such that $\gamma_n = \beta\left(1 - \frac{1}{2n}\right)$ and $\lambda_n = \beta\left(1 + \frac{1}{2n}\right)$. This difference scheme (15), including the initial difference condition (14), form a system of equations which are to be solved iteratively.

As indicated by the boundary conditions (5a) and (5b) we consider the problem for $x \in [0,1]$ and $t \in [0,T]$. The domain $[0,1]$ is sub-divided into N equidistant intervals termed $\triangle x$, i.e. $0 = x_0 < x_1 < x_2 < \cdots < x_{N-1} < x_N$ where $x_{n+1} = x_n + \triangle x$. In a similar fashion the domain $[0,T]$ is sub-divided into M intervals of equal length, $\triangle t$, through which the scheme iterates. The system will iterate until $t_m + \triangle t = T$, i.e. for $M = T/\triangle t$ steps. The system generated by (15) can be written in compact form as

$$A\mathbf{u}^{m+1} = B\mathbf{u}^m + \triangle t\delta e^{\mathbf{u}^m} + \triangle t\delta e^{\mathbf{u}^{m+1}} \tag{16}$$

and is solved as follows

$$\mathbf{u}^{m+1} = (A)^{-1}\left(B\mathbf{u}^m + \triangle t\delta e^{\mathbf{u}^m} + \triangle t\delta e^{\mathbf{u}^{m+1}}\right). \tag{17}$$

The inverse of A is calculated using the $\backslash$ operator in MATLAB which is more efficient than the `inv` function. The nonlinear term on the $m+1^{th}$ level is dealt with through an implementation of the Newton method [26] in an iterative fashion as done by Britz et al. [9] and discussed in [8]. The system $\mathbf{J}\delta\mathbf{u} = -\mathbf{F}(\mathbf{u})$ is solved where $\mathbf{F}$ is the set of difference equations created as per (16) such that $\mathbf{F}(\mathbf{u}) = 0$. The starting vector at $t = 0$ is chosen as per the initial condition (4) such that $\mathbf{u} = 0$. The Newton iteration converges within 2-3 steps given that changes are usually relatively small.

3.2. Rosenbrock method

We now consider two particular Rosenbrock methods, ROS2 and ROWDA3, as a means of comparison for the effectiveness of the methods discussed in the previous section. The Rosenbrock methods belong to the class of linearly implicit Runge - Kutta methods [11, 17]. They were used successfully for the numerical solution of non-electrochemical stiff partial differential equations, including equations of interest to electrochemistry. For further information regarding the particulars of such methods interested readers are referred to the numerical literature of [17, 28–30].

The reason for the use of the Rosenbrock methods in this paper is the ease with which they are able to deal with the nonlinear source term and the fact that no Newton iterations are

necessary. The advantages of these methods are great efficiency, stability and a smooth error response if ROS2 or ROWDA3 are used (see [4] for instance) and the ease with which they are able to handle time-dependent and/or nonlinear systems.

We consider equation (1) as the following system

$$\frac{du_n}{dt} == \begin{cases} \frac{(1+k)}{\triangle x^2}(2u_1 - 2u_0) + \delta e^{u_0} \text{ if } n = 0 \\ \frac{\gamma_n}{\triangle t}u_{n-1} - \frac{2}{\triangle x^2}u_n + \frac{\lambda_n}{\triangle t}u_{n+1} + \delta e^{u_n} \text{ if } n = 1, 2, ..., n-2 \\ -\frac{2}{\triangle x^2}u_{N-1} + \frac{\lambda_{N-1}}{\triangle t}u_{N-2} + \delta e^{u_{N-1}} \text{ if } n = N-1 \end{cases} \tag{18}$$

which along with $0 = u_N$ can be written in the compact form

$$\mathbf{S}\frac{d\mathbf{u}}{dt} = \mathbf{F}(t, \mathbf{u}) \tag{19}$$

where $\mathbf{S} = \text{diag}(1, 1, 1, ..., 1, 0)$ is the selection matrix containing zeros in those positions where the set of differential algebraic equations has an algebraic equation (i.e. zero on the left-hand side of (18)) and unity in those positions corresponding to the ordinary differential equations. The function $\mathbf{F}(t, \mathbf{u})$ can be written as: $\mathbf{F}(t, \mathbf{u}) = \mathbf{J}\mathbf{u} + \mathbf{s}$ where the matrix $\mathbf{J}$ is the Jacobian and the vector $\mathbf{s}$ arises from the constant terms of the set of differential algebraic equations. We can thus write $\mathbf{F}(t, \mathbf{u}) = \mathbf{J}\mathbf{u} + \mathbf{s}$ as

$$= \frac{1}{\triangle t}\begin{bmatrix} -2(1+k)\beta & 2(1+k)\beta & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & \gamma_1 & -2\beta & \lambda_1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & \gamma_2 & -2\beta & \lambda_2 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \dots & \dots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & \gamma_{N-2} & -2\beta & \lambda_{N-2} & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & \gamma_{N-1} & -2\beta & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{bmatrix}$$

$$+ \begin{bmatrix} \delta e^{u_0} \\ \delta e^{u_1} \\ \delta e^{u_2} \\ \vdots \\ \delta e^{u_{N-2}} \\ \delta e^{u_{N-1}} \\ 0 \end{bmatrix} \tag{20}$$

such that

$$\mathbf{F}_u = \frac{1}{\triangle t}\begin{bmatrix} (1+k)\beta(2u_1 - 2u_0) + \delta e^{u_0} \\ \gamma_1 u_0 - 2\beta u_1 + \lambda_1 u_2 + \delta e^{u_1} \\ \gamma_2 u_1 - 2\beta u_2 + \lambda_2 u_3 + \delta e^{u_2} \\ \vdots \\ \gamma_{N-2}u_{N-3} - 2\beta u_{N-2} + \lambda_{N-2}u_{N-1} + \delta e^{u_{N-2}} \\ -2\beta u_{N-1} + \lambda_{N-1}u_N - 2 + \delta e^{u_{N-1}} \\ u_N \end{bmatrix}. \tag{21}$$

In order to implement the Rosenbrock methods a number s of $\mathbf{k}_i$ vectors are computed with s the order chosen. The general equation given by (22) is solved iteratively to obtain each vector $\mathbf{k}_i$ for all i specified which will then be used to update the vector $\mathbf{u}$ for the next time step. We use the notation employed in [7] for the general equation to be used to obtain the values for $\mathbf{k}_i$

$$-\frac{\mathbf{M}}{\beta}\mathbf{k}_i = -\frac{\triangle t}{\beta}\mathbf{F}\left(t+\varphi_i\triangle t, \mathbf{u}+\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j\right) - \frac{\mathbf{S}}{\beta}\sum_{j=1}^{i-1} c_{ij}\mathbf{k}_j - \frac{\kappa_i}{\beta}\triangle t^2\mathbf{F}_t(t,\mathbf{u}) \tag{22}$$

where we define $M = \frac{\mathbf{S}}{\kappa} - \triangle t\mathbf{F}_u$ were the function $\mathbf{F}$ is applied at partly augmented t and $\mathbf{u}$ values and the time derivative $\mathbf{F}_t$ is zero in this case since the system does not include functions of time. Having calculated the s $\mathbf{k}_i$ vectors the solution is obtained from

$$\mathbf{u}_{m+1} = \mathbf{u}_m + \sum_{i=1}^{s} m_i\mathbf{k}_i \tag{23}$$

where the m_i are weighting factors included in the tables of constants specified for each method (see [4] and [7]).

In this chapter we implement the ROS2 and ROWDA3 methods though there are other variants of the Rosenbrock methods. Lang [24] described a L-stable second-order formula called ROS2. A third-order variant thereof is called ROWDA3 and described by Roche [28] and later made more efficient by Lang [23]. The latter is a method favoured by Bieniasz who introduced Rosenbrock methods to electrochemical digital simulation [4, 5]. For a more detailed explanation and discussion regarding the method and its advantages refer to [7]. The focus of the work done here is with regards to whether the Rosenbrock algorithms lend themselves toward parallelized implementation. It has already been noted that functions such as the `parfor` command cannot be used in this instance. It now remains to consider the method's performance when run on a GPU via the MATLAB Parallel Computing Toolbox.

4. Discussion of numerical results

The results noticed, as per Table 1, indicate the extent to which implementing the code on the GPU slows down overall performance of the CN, CN_N, ROS2 and ROWDA3 methods. The question is why this would be the case. In Table 1 the results for the different methods run on a CPU were obtained by running the code on one CPU only instead of all of those available to MATLAB on the computer employed. This was done to get a better understanding of the one-on-one performance between the processing units, and yet implementing the code on the GPU still led to poor performance.

To gain a better understanding of these results we consider the baseline structure for our CN code:

```
A = gpuArray(A);
B = gpuArray(B);
```

```
u0 = gpuArray(u0);
for m = 1:T
    b = delta.*dt.*exp((1+eps.*u0).\(u0));
    u0 = mldivide(A,(B*u0+b));
end
```

In doing so, we realise that the main components thereof are matrix and elementwise vector operations. In order to understand why we are achieving the results we do (see Table 1) we run a few simple 'test'-codes to consider the speed taken by the CPU vs. the GPU to perform such elementary operations as `C\d` and `d.*f` where C is a matrix and d and f are vectors. In Figure 1 we see the speed of the CPU over the GPU computed as $\frac{CPU\ \ running\ \ time}{GPU\ \ running\ \ time}$. You will notice that as the size of the matrix and corresponding vector increases so too does the speed at which the GPU is able to compute `C\d` allowing it to overtake the CPU. This is what one would expect given that the GPU will only 'kick in' once the CPU is overloaded with data structures too large for it to compute effectively. Thus the efficiency in terms of running time of the code provided above is heavily dependent upon the size of the matrices A and B. At this juncture it is important to remember that we are considering the range $x \in [0,1]$ with $\triangle x = 0.1$ which means that our A matrix is a 10×10 matrix and as such not large enough to give the GPU the chance to expose its ability to improve the performance of the algorithm. The reason for the choice in the size of the matrix for the problem considered is twofold: (1) it is due to the influence of the ratio $\beta = \triangle t/\triangle x^2$ which one usually tries to keep close to 1 for reasons of stability, and (2) the limitations of memory of the PC being used.

The next step in this evaluative process is to now consider the speed at which vector operations are performed. This was done in a similar fashion to the previous case by considering the speed taken by the CPU and GPU to perform the elementwise operation `d.*f` where d and f are vectors. The ratios of the speeds $\frac{CPU\ \ running\ \ time}{GPU\ \ running\ \ time}$ were also considered for the in-built function `arrayfun` which performs elementwise operations on all its inputs. It can clearly be seen in Figure 2 that the in-built function outperforms the normal $.*$ operation. What is interesting in this case is that the size of the vector required for the GPU to outperform the CPU is very large - we considered vectors of sizes between 200 000 and 201 000 as indicated. For smaller vector lengths the GPU is completely outperformed by the speed at which calculations are done on the CPU. As such, to improve the speed at which these vector calculations are performed we would either (1) have to diminish $\triangle x$ to the degree needed to obtain vectors of the required length (2) or be required to move the vectors from the GPU memory to the CPU memory every time calculations need to be made. The first approach would require a memory capacity beyond that of the computer used here and the second would greatly increase the running time of the algorithm and as such is not worth implementing.

As a means of further investigation we consider the CN code as a test case for the use of the `arrayfun` function. Obviously implementing this in-built function as follows

```
A = gpuArray(A);
B = gpuArray(B);
u0 = gpuArray(u0);
u0 = arrayfun(@myCrank,u0,A,B)
```

$\beta = 0.01$ and $T = 0.3$				
	CN	CN_N	$ROS2$	$ROWDA3$
CPU	7.8449e-05	2.7737e-04	2.3002e-05	9.1288e-05
GPU	0.0014	0.0960	0.0033	0.0063
$\beta = 0.01$ and $T = 5$				
	CN	CN_N	$ROS2$	$ROWDA3$
CPU	1.4783e-05	2.0018e-04	1.5354e-05	6.3574e-05
GPU	8.0940e-04	0.0047	0.0028	0.0047
$\beta = 2$ and $T = 5$				
	CN	CN_N	$ROS2$	$ROWDA3$
CPU	2.4573e-05	0.0033	1.7146e-05	6.8119e-05
GPU	0.0048	0.4731	0.0042	0.0073

Table 1. Running times per iteration of $\triangle t$ for the relevant methods implemented for $\triangle t = 0.0001$, $\triangle x = 0.1$, $\delta = 1$ and $\epsilon = 0$.

$\epsilon = 0.01$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0137	0.0025	0.0059	0.0138
$\epsilon = 0.05$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0133	0.0024	0.0067	0.0149
$\epsilon = 0.1$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0146	0.0025	0.0057	0.0128
$\epsilon = 0.25$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0145	0.0024	0.0059	0.0133

Table 2. The ratio $\frac{CPU \quad running \quad time}{GPU \quad running \quad time}$ for the relevant methods implemented for $\triangle t = 0.0001$, $\triangle x = 0.1$, $\delta = 1$ and $T = 0.3$.

$\delta = 0.5$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0146	0.0012	0.0064	0.0150
$\delta = 1$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0275	0.0026	0.0061	0.0160
$\delta = 2$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0151	0.0030	0.0062	0.0151
$\delta = 3$			
CN	CN_N	$ROS2$	$ROWDA3$
0.0160	0.0042	0.0063	0.0140

Table 3. The ratio $\frac{CPU \quad running \quad time}{GPU \quad running \quad time}$ for the relevant methods implemented for $\triangle t = 0.0001$, $\triangle x = 0.1$, $\epsilon = 0$ and $T = 0.3$.

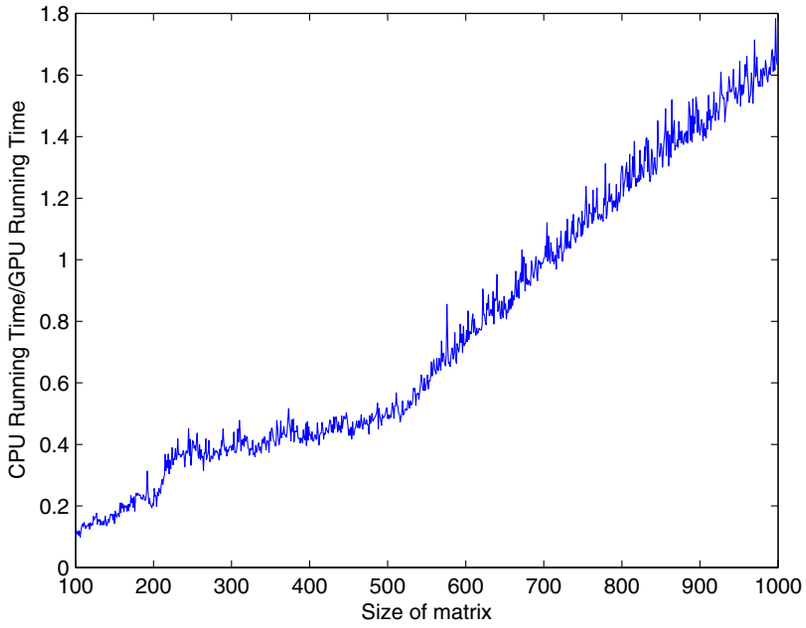

Figure 1. Plot showing the CPU Running Time/GPU Running Time for matrices and corresponding vectors of sizes 100 to 1000.

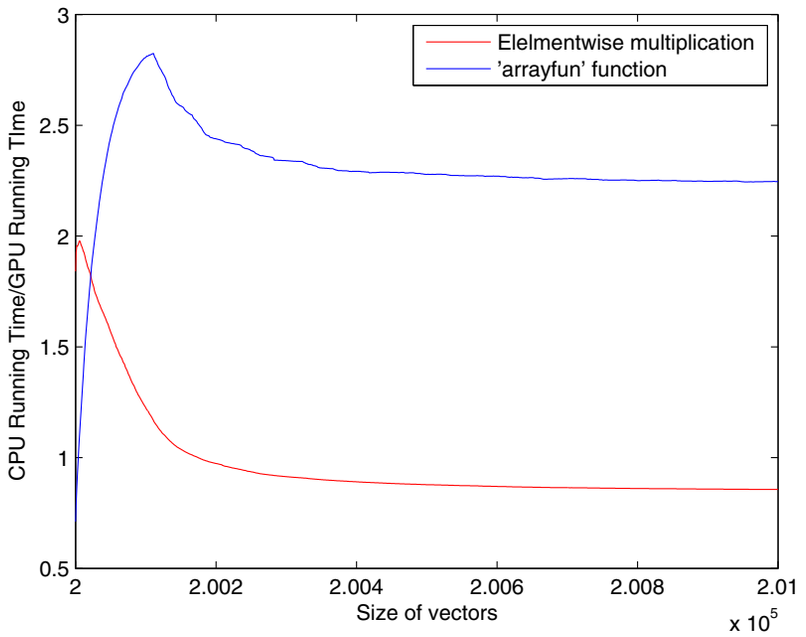

Figure 2. Plot showing the CPU Running Time/GPU Running Time for vectors of sizes 200 000 to 201 000.

where the `@myCrank` function performs the loop through m, instead of the code presented previously produces incorrect results. The results obtained do however support our findings that the `arrayfun` function is able to increase the speed with which elementwise operations are performed. In this instance `arrayfun` is computing on the GPU since the inputs are

all GPU array objects. We found for $T = 10$ with $\triangle t = 0.0001$ as we decreased $\triangle x$ that computing on the GPU was faster than doing so on the CPU: for $\triangle x = 0.1$ and 0.01 the ratios were $\frac{CPU\ \ running\ \ time}{GPU\ \ running\ \ time} = 1.5709$ and $\frac{CPU\ \ running\ \ time}{GPU\ \ running\ \ time} = 6.5906$ respectively. This makes sense given that smaller values of $\triangle x$ would increase the sizes of the matrices A and B and the vectors b and $u0$. As such, it seems likely that using a PC with a greater memory capacity would lead to the GPU outperforming the CPU by a large margin as $\triangle x$ decreases.

4.1. Influence of changing parameter values on the running time of the algorithms

Just a few brief comments upon the results obtained for CN, CN_N, ROS2 and ROWDA3 for varying values of ϵ and δ will be made in this section. Firstly we considered the schemes for $\delta = 1$ and $\epsilon = 0.01, 0.05, 0.1$ and 0.25 and then we also considered the case for $\epsilon = 0$ with $\delta = 0.1, 1, 2$ and 3. The reader will notice considering Tables 2 and 3 that there does not seem to be any noticeable trend to the results obtained. As such the values of ϵ and δ do not seem to have a meaningful impact on the speed at which the algorithms compute.

5. Concluding remarks

The implementation of numerical algorithms such as those considered in this chapter are widely used for the solution of many differential equations which model physical processes and applications. As such it is of vital importance that we be able to perform such calculations at high speed given the requirement of fine grids to improve accuracy. It is in this context that the use of GPUs becomes of prime importance. However it is not simply a matter of running the algorithm on the GPU - the method employed needs to lend itself to being adjusted in the required manner so that the parallel processing power of the GPU may be taken advantage of. Though we found that the numerical methods considered here were not entirely suited to being implemented on the GPU as we would have hoped we were able to explain why this was the case.

This work has investigated the effectiveness of matrix and elementwise operations when run on a GPU vs. a CPU and found that the speed taken to do such operations heavily relies on the choice of $\triangle x$. It was discovered that the introduction of the nonlinear source term is problematic due to the length of time taken to do elementwise calculations on the GPU. While matrix operations were also shown to be slow it was more specifically this aspect of the code which increased the running time.

We also discovered the power of the in-built function `arrayfun` which was able to improve upon the performance of the GPU with regards to computing time to the degree that it outperformed the CPU even for a grid with 'large' $\triangle x$, i.e. small matrices and vectors within the computations. As the grid became finer the performance of the GPU over the CPU improved, indicating the impact of the size of the matrices upon which computations are being performed and the degree to which `arrayfun` is able to improve computations occurring on the GPU. Thus, the manner in which `arrayfun` computes elementwise is extremely efficient and if such a structure could be developed for matrix operations then that would truly allow the performance of the GPU to overtake that of CPU computing.

What the work in this chapter has shown is that the structures of the GPU and the Parallel Computing Toolbox in MATLAB are such that while certain algorithms have the ability to be adjusted for improved performance not all methods do. In particular it seems clear that implicit methods with matrix and vector operations will in fact run much slower on the GPU than the CPU. Thus whether GPU computing is able to improve the performance of a numerical scheme is very much dependent upon the type of computations which need to be done. In our case we discovered that the implicit and nonlinear nature of our numerical schemes do not lend themselves towards improved performance via the implementation of the parallel processing power of a GPU.

Acknowledgements

I would like to thank Mr. Dario Fanucchi for invaluable discussions.

Author details

Charis Harley
Faculty of Science, University of the Witwatersrand, School of Computational and Applied Mathematics, Centre for Differential Equations, Continuum Mechanics and Applications, South Africa

6. References

[1] Abd El-Salam, M. R. & Shehata, M. H. (2005). The numerical solution for reaction diffusion combustion with fuel consumption, *A*ppl. Math. Comp., 160:423Ű-435.
[2] Anderson, C. A.; Zienkiewicz, O. C. (1974). Spontaneous ignition: finite element solutions for steady and transient conditions, *J*. Heat Transfer, 96(3):398–404
[3] Anderson, D.; Hamn*é*n, H.; Lisak, M.; Elevant T. & Persson, H (1991). Transition to thermonuclear burn in fusion plasmas, *P*lasma Physics and Controlled Fusion, 33(10):1145–1159
[4] Bieniasz, L. K. (1999). Finite-difference electrochemical kinetic simulations using the Rosenbrock time integration scheme, *J*ournal of Electroanalytical Chemistry, 469:97–115
[5] Bieniasz, L. K. & Britz, D. (2001). Chronopotentiometry at a Microband Electrode: Simulation Study Using a Rosenbrock Time Integration Scheme for Differential-Algebraic Equations, and a Direct Sparse Solver, *J*ournal of Electroanalytical Chemistry, 503:141–152
[6] Boddington, T. & Gray, P. (1970). Temperature profiles in endothermic and exothermic reactions and interpretation of experimental rate data, *P*roc. Roy. Soc. Lond Ser A - Mat. Phys. Sci., 320(1540):71–100
[7] Britz, D. (2005). *D*igital Simulation in Electrochemistry, 3rd Edition, Lecture Notes in Physics, Springer, 3 − 540 − 23979 − 0, Berlin Heidelberg
[8] Britz, D.; Baronas, R.; Gaidamauskaitė, E. & Ivanauskas, F. (2009). Further Comparisons of Finite Difference Schemes for Computational Modelling of Biosensors, *N*onlinear Analysis: Modelling and Control, 14(4):419–433
[9] Britz, D.; Strutwolf J. & Østerby, O. (2011). Digital simulation of thermal reactions, *A*ppl. Math. and Comp., 218(4), 15:1280–1290

[10] Chambré, P. L. (1952). On the solution of the Poisson-Boltzmann equation with application to the theory of thermal explosions, *J*. Chem. Phys., 20:1795–1797

[11] Chan, Y. N. I.; Birnbaum, I. & Lapidus, L. (1978). Solution of Stiff Differential Equations and the Use of Imbedding Techniques, *I*nd. Eng. Chem. Fundam., 17(3):133–148

[12] Crank J. & Nicolson, E. (1947). A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type, *P*roc. Camb. Phil. Soc., 43:50–67

[13] Crank J. & Furzeland, R. M. (1977). The treatment of boundary singularities in axially symmetric problems containing discs, *J*. Inst. Math. Appl., 20(3):355–370

[14] Evans. D. J. & Danaee, A. (1982). A new group Hopscotch method for the numerical solution of partial differential equations, *S*IAM J. Numer. Anal., 19(3):588–598

[15] Feldberg, S. W. (1987). Propagational inadequacy of the hopscotch finite difference algorithm: the enhancement of performance when used with an exponentially expanding grid for simulation of electrochemical diffusion problems, *J*. Electroanal. Chem., 222:101–106

[16] Frank-Kamenetskii, D. A. (1969). *D*iffusion and Heat Transfer in Chemical Kinetics, Plenum Press, New York

[17] Hairer E. & Wanner, G. (1991). Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems, Springer-Verlag, $3-540-60452-9$, Berlin

[18] Harley, C. & Momoniat, E. (2007). Steady state solutions for a thermal explosion in a cylindrical vessel, *M*odern Physics Letters B (MPLB), 21(14):831–841.

[19] Harley, C. & Momoniat, E. (2008). Instability of invariant boundary conditions of a generalized Lane-Emden equation of the second-kind, *A*pplied Mathematics and Computation, 198:621–633

[20] Harley, C. & Momoniat, E. (2008). Alternate derivation of the critical value of the Frank-Kamenetskii parameter in the cylindrical geometry, *J*ournal of Nonlinear Mathematical Physics, 15(1):69–76

[21] Harley, C. (2010). Explicit-implicit Hopscotch method: The numerical solution of the Frank-Kamenetskii partial differential equation, *J*ournal of Applied Mathematics and Computation, 217(8):4065–4075

[22] Harley, C. (2011). Crank-Nicolson and Hopscotch method: An emphasis on maintaining the implicit discretisation of the source term as a means of investigating critical parameters. Special Issue on 'Nonlinear Problems: Analytical and Computational Approach with Applications', *Abstract and Applied Analysis*, Submitted.

[23] Lang, J. (1996). High-resolution self-adaptive computations on chemical reaction-diffusion problems with internal boundaries, *C*hemical Engineering Science, 51(7):1055–1070

[24] Lang, J. (2001). *A*daptive Multilevel Solution of Nonlinear Parabolic PDE Systems, Springer, 9783540679004, Berlin

[25] The MathWorks, Inc. ©1994-2012. Parallel Computing Toolbox Perform parallel computations on multicore computers, GPUs, and computer clusters, *http* : *//www.mathworks.com/products/parallel − computing/*.

[26] Press, W. H.; Teukolsky, S. A.; Vetterling, W. T. & Flannery, B. P. (1986). *N*umerical Recipes in Fortran, 2nd Edition, Cambridge University Press, $0-521-43064-X$, Cambridge

[27] Rice, O. K. (1940). The role of heat conduction in thermal gaseous explosions, *J*. Chem. Phys., 8(9):727–733
[28] Roche, M. (1988). Rosenbrock methods for differential algebraic equations, *N*umerische Mathematik, 52:45–63
[29] Rosenbrock, H. H. (1963). Some general implicit processes for the numerical solution of differential equations, *T*he Computer Journal, 5(4):329–330
[30] Sandu, A.; Verwer, J. G.; Blom, J.G.; Spee, E. J. & Carmichael, G. R. (1997). Benchmarking Stiff ODE Solvers for Atmospheric Chemistry Problems II: Rosenbrock Solvers, *A*tmospheric Environment, 31:3459–3472
[31] Steggerda, J. J. (1965). Thermal stability: an extension of Frank-Kamenetskii's theory, *J*. Chem. Phys., 43:4446–4448
[32] Zhang, G.; Merkin J. H. & Scott, S. K. (1991). Reaction-diffusion model for combustion with fuel consumption: Ii. Robin boundary conditions, *I*MA J. Appl. Math., 51:69–93
[33] Zhang, G.; Merkin J. H. & Scott, S. K. (1991). Reaction-diffusion model for combustion with fuel consumption: I. Dirichlet boundary conditions, *I*MA J. Appl. Math., 47:33–60
[34] Zeldovich, Y. B.; Barenblatt, G. I.; Librovich, V. B. & Makhviladze, G. M. (1985). The Mathematical Theory of Combustion and Explosions, *C*onsultants Bureau, New York