GPU Implementation of Implicit Runge-Kutta Methods

Navchetan Awasthi, Abhijith J
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
navchetanawasthi@gmail.com, abhijith31792@gmail.com

***Abstract*—Runge-Kutta methods are an important family of implicit and explicit iterative methods used for the approximation of solutions of ordinary differential equations. Explicit Runge-Kutta methods are unsuitable for the solution of stiff equations as their region of stability is small. Stiff equation is a differential equation for which certain numerical methods for solving the equation are numerically unstable, unless the step size is very small. This project aims in parallelizing implicit Runge-Kutta methods which are stable even for the stiff problems . The major disadvantage is that Implicit Runge-Kutta methods are more computationally expensive. We will show how to accelerate the execution of implicit Runge-Kutta methods using GPUs.**

I. Introduction

Ordinary differential equations play a major role in the modeling of various physical, biological, ecological and other phenomena. Among the algorithms available , the family of Runge-Kutta methods are most widely used. Our aim is to implement implicit Runge-Kutta(RK) methods for these problems on GPUs. As with most numerical methods stability is a prime concern in RK methods also. Explicit RK methods are easier to implement but are known to be unstable for stiff system of ODEs. Implicit RK methods are stable even for stiff systems but are computationally much more expensive as they involve matrix operations.

The main emphasis of this project is the acceleration of implicit Runge-Kutta methods for heat equation. One of the interesting property of the heat equation is that the maximum principle which states that the maximum value of the temperature comes either from source or from earlier in time because heat permeates but is not created from nothing.Another property is that even if there is a discontinuity at an initial time $\mathbf{t_0}$ then the temperature becomes smooth as soon as $\mathbf{t} > \mathbf{t_0}$.This property makes for an easy check of the correctness ofnumerical simulations. It finds application in a lot of places like random walks and and Financial mathematics.

In section 2 related work done is explained including the explicit methods on GPU and their implementation. Our methodology is being explained in section 3. Subsection 3A explains the various mathematical equations, matrix used for our problem, initial conditions and the Butcher tableau with system of equations used for the calculation of slopes. Subsection 3B explains the parallel implementation of the problem in GPUs. The experimental setup and results are explained in section 4. Finally section 5 will give the main conculsions of the work.

II. Related Work

The parallel implementation of implicit RK methods on cluster of PC's has been extensively studied([1],[2]). The main caveats of implementing these methods lie in solving a linear system of equations that arise during intermediate steps. Good speed-up and scaling behavior was observed in these cases. [2] specifically looks at a class of methods known as Implicit-Explicit Range-Kutta(IMEXRK) methods which solves the system by splitting it into stiff and non stiff parts.

Murray[3] has studied the GPU acceleration of explicit RK integrators. Speed up of 115-fold over serial CPU implementations was reported using computationally low-storage explicit RK integrators. This paper also discusses a strategy to achieve load balancing between GPU threads in case of variable step size methods where the the number of steps that needs to be implemented by a thread is not known before hand.

III. Methodology

Explicit RK methods have been implemented in GPU's [3] while implicit RK methods have been implemented in clusters([1],[2]). A hybrid strategy involving both the host and device has been developed to implement the time stepping and linear algebra solver parts of the algorithm effectively.

Runge Kutta methods may be characterized by a set of constants that is represented by the Butcher tableau which represents the coefficients involved in the Runge Kutta method used.

$$\begin{array}{ccc|c} a_{11} & \dots & a_{1n} & \tau_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \dots & a_{nn} & \tau_n \\ \hline b_1 & \dots & b_n & \end{array}$$

A Runge-Kutta method is implicit if no permutation of rows can reduce $\mathbf{A}$ to strict lower-triangular form[1]. We will try to first decouple the computation associated with each stage when the implicit methods are implemented in GPU's. The main focus will be on parallelizing the time consuming matrix operations involved in the problem.

We have performed the iterations for multiple N (which will determine the stiffness associated with the problem) for a specified no of steps to take into account the effects of synchronization and communication costs.

A. Mathematical Formalism

The system of ODE's used for our experiment is obtained by semi-discretization of the 1-D heat equation. This leads to an equation of the form

$$\frac{d\mathbf{y}}{dt} = \mathbf{L}\mathbf{y} \tag{1}$$

where $\mathbf{y}$ is a vector of length N and $\mathbf{L}$ is an $N \times N$ tridiagonal matrix that is given below.

$$\mathbf{L} = -(N+1)^2 \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \tag{2}$$

The boundary conditions used for the problem is

$$y_0(t) = y_{N+1}(t) = 0.0, \tag{3}$$
$$y_i(0) = 100.0, \qquad i = 1, 2, \dots N$$

This is the simulation of a rod cooling down from while its ends are maintained at a lower temperature. The solution shows an exponential decay to zero. The solution is shown in $Fig \quad 1$

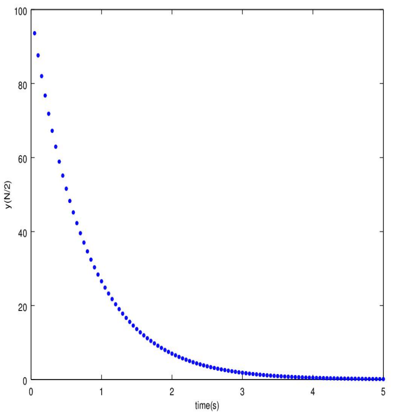


Fig. 1. Decay of the solution value at $y_{N/2}$

We have used a second order Runge-Kutta method with the following Butcher tableau,

$$\begin{array}{cc|c} 1 & 0 & 1 \\ -1 & 1 & 1 \\ \hline \frac{1}{2} & \frac{1}{2} & \end{array}$$

since the coefficient matrix is not strictly lower triangular the slopes at all intermediate steps are not explicitly calculable. The system of linear equations we need to solve to find the intermediate slopes $\mathbf{k_1}$ and $\mathbf{k_2}$.

$$\begin{pmatrix} \mathbf{I} - a_{11}\mathbf{L} & \mathbf{0} \\ -a_{21}\mathbf{L} & \mathbf{I} - a_{22}\mathbf{L} \end{pmatrix} \begin{pmatrix} \mathbf{k_1} \\ \mathbf{k_2} \end{pmatrix} = \begin{pmatrix} \mathbf{Ly} \\ \mathbf{Ly} \end{pmatrix} \tag{4}$$

This system of equations have a hex-diagonal nature shown in the bitmap image of the coefficient matrix of $Eq(4)$ below.

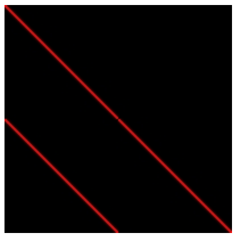


Fig. 2. Figure showing the structure of the coefficient matrix

In out implementation we have solved this system using Gauss-Seidel iterations.The solution is then updated using the equation

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h(b_1\mathbf{k_1} + b_2\mathbf{k_2}) \tag{5}$$

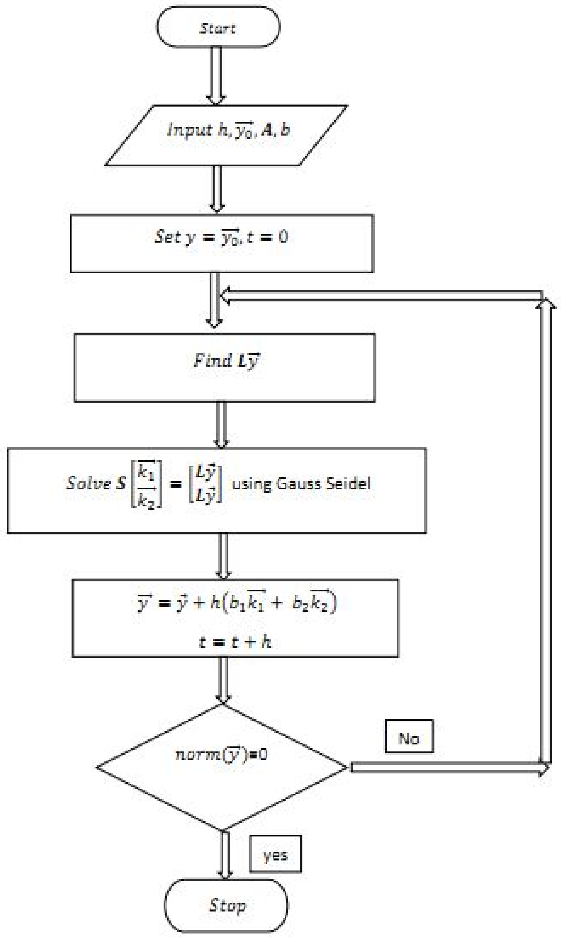
Here h is the time step. The time step must be chosen in accordance to the Courant-Friedrichs-Lewy(CFL) criteria *ie.* h must be small when N is large to ensure convergence. Another important metric that determines the accuracy of solution is the error bound set on the Gauss-Seidel iterations. The solution proceeds till all the points in the domain reach the value of zero.

The algorithm is given in $Fig \quad 3$

B. Parallelization of solution in GPU

We have achieved parallel implementation of the system described above by using a two pronged approach. First, do all intensive matrix-vector operations and linear equation solving in parallel in the GPU. Second, use the CPU for simple vector- vector operations (like the update step) to avoid too many kernel launches that may slow down the execution. Also it has been found helpful to overlap CPU and GPU computations where ever possible. All matrix operations are reduced to tridiagonal operations and specialized routines were made that exploited the tridiagonal property. $\mathbf{L}$ is already a tridiagonal matrix so the multiplication $\mathbf{Ly}$ can be performed on the GPU by assigning each row of $\mathbf{L}$ to a single Cuda thread. Each thread would then have to do only three multiplications to come up with the result.

a) Solving the hex-diagonal system: The system of linear equations given by $Eq(4)$ is solved by noting that the block matrix form already has a triangular structure(this is not true



Start
Input $h, \overline{y_0}, A, b$
Set $y = \overline{y_0}, t = 0$
Find $L\overline{y}$
Solve $S\begin{bmatrix}\overline{k_1}\\ \overline{k_2}\end{bmatrix} = \begin{bmatrix}L\overline{y}\\ L\overline{y}\end{bmatrix}$ using Gauss Seidel
$\overline{y} = \bar{y} + h(b_1\overline{k_1} + b_2\overline{k_2})$
$t = t + h$
$norm(\overline{y})=0$
No
yes
Stop

Fig. 3. Flowchart of RK2 Method to solve semi-discretized heat equation

for the actual matrix, so a direct triangular solve cannot be applied). So this $2N \times 2N$ system can be decomposed trivially to two $N \times N$ systems given below. The first one can be solved to find $\mathbf{k_1}$ which can be then substituted into the second system to find $\mathbf{k_2}$.

$$(\mathbf{I} - a_{11}\mathbf{L})\mathbf{k_1} = \mathbf{Ly} \tag{6}$$

$$(\mathbf{I} - a_{22}\mathbf{L})\mathbf{k_2} = \mathbf{Ly} + a_{21}\mathbf{Lk_1} \tag{7}$$

These are tridiagonal systems that can be solves using Gauss-Seidel method using Red-Black ordering scheme.

b) Synchronization strategies for red black ordering: The red-black ordering is a fairly straight forward strategy that ensures that the faster convergence property of Gauss Seidel iterations are preserved even when implemented in parallel. In Cuda since threads can be directly indexed, the Red black ordering involves computing first with threads with even index, updating the values(a barrier must be present at this stage), and then computing the threads with odd index. This is easily implementable in a single block as there exists straight forward functions in Cuda that help in synchronization of threads within a block. But this limits the problem size to 1024 spatial points as it is the maximum number of threads that can be held by a block. So a more complicated scheme has to be developed that can synchronize the threads in all blocks and also calculate the global error value by reduction across all blocks(this error value is needed to terminate the Gauss-Seidel iterations). This is done by multiple kernel launches. In the single block scheme the entire iteration is done on the GPU. But in the multi-block scheme a single kernel launch is done for a single iteration and the error is then written into a global variable. The error vector is then reduced using another kernel launch and the final error comparison with the threshold(e_{max}) is then done on the CPU, which then decides whether to terminate the iteration or proceed.

IV. Experiments and Results

The correctness of the simulations were verified by running the simulations till convergence *ie.* when all points reach a value of zero (coressponding to equilibrium temperature). Three different codes were made and compared.

1 Serial
2 Parallel with all threads in a single block
3 Parallel with multiple blocks (32 unless otherwise specified) with synchronization achieved using multiple kernel launches

A. Experiment Setup

All codes were ran on the Fermi cluster at SERC. The CPU and GPU specifications are given below

- CPU : Intel(R) Xeon(R) CPU W3550 processor operating at 3.07 Ghz with 16Gb RAM
- GPU : Nvidia C2070 GPGPU

The experiments were done to study the variation of execution time with the size of the problem(N), the maximum error threshold(e_{max}), and the number of blocks

B. Results

The execution time comparisons of the three schemes for varying problem size(N) mentioned above show a familiar trend. Parallelization of implicit RK method for small problem sizes is over kill as synchronization costs and a large number of kernel launches slow down the execution. The point to note is that the number of kernel launches remain constant even when the problem size increases. So while the timings for the serial execution scale as $O(N^3)$ the timings of the parallel executions scale much slower. The single block case can be run only for up to 1024 threads and was found to be superior to the multi-block case for smaller sizes. This is expected because the multi block case was found to have at-least three orders of magnitude more number of kernel launches when compared to the single block case. But this effect is overshadowed while the problem size grows up. The multi block execution comes out as the clear winner when it comes to handling large problem sizes.

The speed up variation with problem size is shown below. We can see that speed-up more than doubles when N (the number of threads) goes up. This indicates an increase in

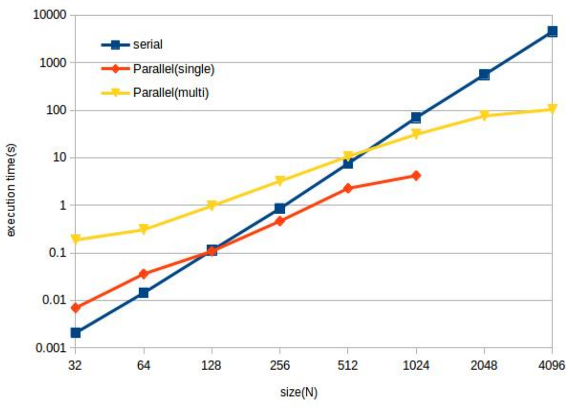


Fig. 4. log-log plot showing the variation of execution time with problem size for the three different schemes.

efficiency of the parallel implementation, again pointing to the fact that parallelization is an effective strategy for large problem sizes.

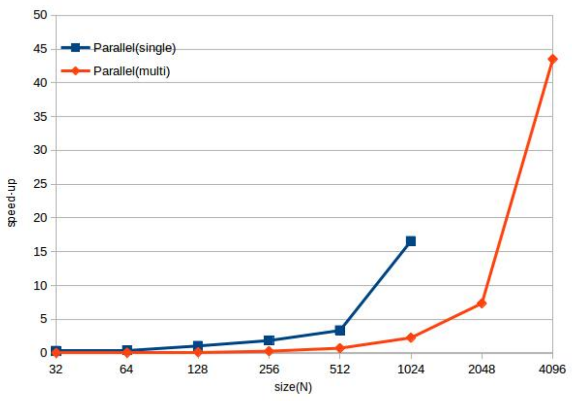


Fig. 5. Speed up of the two parallel schemes.

The variation in the execution time with e_{max} and number of blocks are shown in $Fig\quad 6$ and $Fig\quad 7$. It is seen that decreasing the error threshold below a certain limit doesn't affect the timings much. The execution times don't show much variation when the number of blocks are changed. This is because the number of kernel launches are independent of the number of blocks alloted. Even though the error reduction has to happen across the blocks (and hence depends on the number of blocks) its effect on execution time is not very high as the number of blocks is very less compared to the number of threads. Both the graphs shown below show the behavior of $N = 512$ case.

C. *Comparison with related work*

The maximum speed-up we obtained during our experiments was 43.53 using the multiblock scheme for $N =$ 2^{12}(which is also the number of threads). Although the same combination (implicit/GPU) is not found in literature it would still be insightful to compare our results with some other papers.

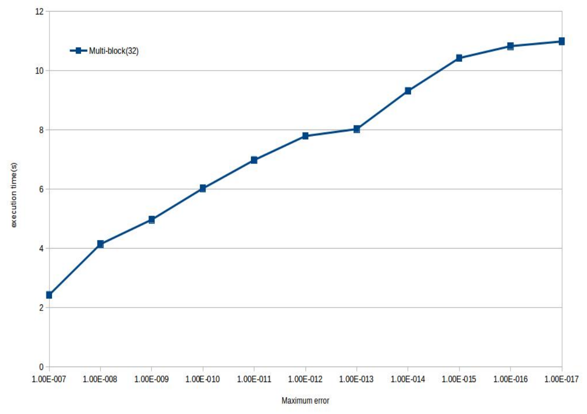


Fig. 6. Variation in execution time with e_{max}.

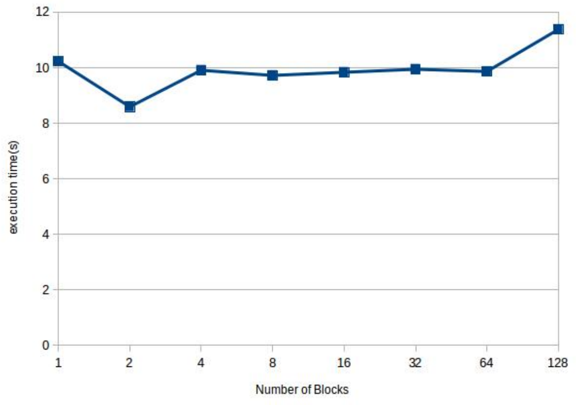


Fig. 7. Variation in execution time with number of blocks.

Mantas et al. [2] (implicit/cluster) has obtained a maximum speed-up of 13.47 for $N = 4000$ by running on 15 processors using IMEXRK methods for simulating Boltzmann equations for rarefied gas.

Murray[3](explicit/GPU) reports a maximum speed up of 115.4 for 2^{19} threads using DOPRI5 (which is an explicit method) for single precision arithmatic for simulating the Loretnz 96 system of equations on Nvidia Tesla C1060 GPU.

V. Conclusions

It is seen from the above results that the implicit Runge-Kutta methods are ripe for parallelization using GPGPU's and there is much improvement in terms of execution time to be gained in comparison to the serial execution of the same. It was seen that the parallel strategies consistently do better when dealing with large problem sizes and show excellent scaling behavior in that respect($Fig\quad 4, 5$).

This topic may be further investigated in the following lines. Direct methods can be used instead of the Gauss-Seidel iterations and their behavior once parallelized can be studied. Also Runge-Kutta methods with different Butcher tableaus and orders can also be parallelized and their behaviors studied so as to reach a consolidated idea of how implicit Runge-Kutta methods behave in general when subjected to fine-grain parallelism using GPGPUs.

VI. References

- [1] Karakashian, O. A., & Rust, W. (1988). On the parallel implementation of implicit Runge-Kutta methods. SIAM journal on scientific and statistical computing, 9(6), 1085-1090.
- [2] Mantas, J. M., Gonzlez, P., & Carrillo, J. A. (2005). Parallelization of implicit-explicit runge-kutta methods for cluster of PCs. In Euro-Par 2005 Parallel Processing (pp. 815-825). Springer Berlin Heidelberg.
- [3]Murray, L. (2012). GPU acceleration of Runge-Kutta integrators. Parallel and Distributed Systems, IEEE Transactions on, 23(1), 94-101.
- [4]Adams, Loyce, and J. Ortega. ”A multi-color SOR method for parallel computation.” ICPP. 1982.
- [5]Nvidia, C. U. D. A. ”Programming guide.” (2008).
- [6]Butcher, John. ”Runge-Kutta methods.” Scholarpedia 2.9 (2007): 3147.