CUDA Differential Equation Solver

Sakthi Kumar Sampath Kumar[ssamp016]

***Abstract*— In this project, we have the CUDA implementation of a differential equation solver using the Runge-Kutta method. The solver is designed to run on a GPU for efficient parallel computation. And have Implemented certain analyses after solving (such as finding peaks for a large system of particles) can be accelerated using GPUs.**

I. Overview

In this project, we have developed a CUDA implementation of a differential equation solver using the renowned Runge-Kutta method. By harnessing the power of GPU computing, our solver is designed to efficiently handle complex computations in parallel, resulting in significant speed and performance improvements.

Furthermore, our implementation goes beyond just solving the equations. We have also incorporated additional analyses that can be performed on the solved equations. These analyses provide valuable insights into the behavior and characteristics of the system being modeled, allowing for a deeper understanding and interpretation of the results.The method given Below is the RKF45 method (with optimal weights) for numerical solution of a system.

$$k_1 = f(x, y, p, h)$$

$$k_2 = f(x + \frac{h}{4}, y + \frac{h}{4}k_1, p, h)$$

$$k_3 = f(x + \frac{3h}{8}, y + \frac{3h}{32}k_1 + \frac{9h}{32}k_2, p, h)$$

$$k_4 = f(x+\frac{12h}{13}, y+\frac{1932}{2197}hk_1-\frac{7200}{2197}hk_2+\frac{7296}{2197}hk_3, p, h)$$

$$k_5 = f(x+h, y+\frac{439}{216}hk_1-8hk_2+\frac{3680}{513}hk_3-\frac{845}{4104}hk_4, p, h)$$

$$k_6 = f(x+\frac{h}{2}, y-\frac{8}{27}hk_1+2hk_2-\frac{3544}{2565}hk_3+\frac{1859}{4104}hk_4-\frac{11}{40}hk_5)$$

$$y_1 = y + h\left(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\right)$$

$$z_1 = y+h\left(\frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6\right)$$

$$s = 0.84\left(\frac{h \cdot 0.0001}{\max(z_1 - y_1)}\right)^{1/4}$$

$$ds\_step[i] = s \cdot h$$

$$x[t, i, 0] = x$$

$$x[t, i, 1] = y$$

$$x[t, i, 2] = z_1$$

$$x_0[i, 0] += s \cdot h$$

$$x_0[i, 1] = y_1$$

II. INTRODUCTION

A. How is the GPU used to accelerate the application? Examples of this includes

In this project we try to optimize solving of those problems where a numerous nodes are present in a network communicating with each other sharing energy between edges or when we try to stimulate n particles interacting with other n-1 particles. Examples of such systems include neuron network model, Gene Regulatory networks and protein-protein interaction(PPI) networks. Initially, in a sequential program, each particle is

processed once before going to the next step. This becomes more and more challenging and time-consuming as we increase the number of particles or when we want to test for multiple parameters or initial conditions.

In this implementation, we have demonstrated N systems with different initial conditions. And each system is given a GPU block, and each block has n threads, where each thread evolves a particle or node.

We have utilized shared memory for each block storing the current step X_i ,which is required by all the n threads for calculating next step and optimal step size calculated through RKF45.

For the peaks finding kernel we have implemented a parallel algorithm performs the operation like assignment 4.

B. Libraries Used

- **networkx:** we use networkx for generating the adjacency matrix of a Watts–Strogatz model.

C. Limitations

1) The solver does not stop after encountering NaN.
2) Cannot pass function as a parameter, we have to modify the hard-coded ode function in kernel.py.
3) No dynamic shared memory(fixed size, hard-coded).

D. Running the Program

E. Solver:

Input:

1) Modify Discrete Differential System or ODE (function lo) in $kernel.py$.
2) Modify N (no of systems)
3) n (nodes/ particles in a system)
4) $Tspan$ (short for time span)(no of steps to evolve)
5) p (parameter list)
6) x_0 = random initial values for n nodes of N systems.

Output:

1) Time-series for N systems, n nodes and d dimensions for time steps(tspan) in the form of a 4d array of dimensions N*tspan*n*dimensions
2) In our example we are using a system of dimension 2. The below is a heat-map of the system for a given initial value.

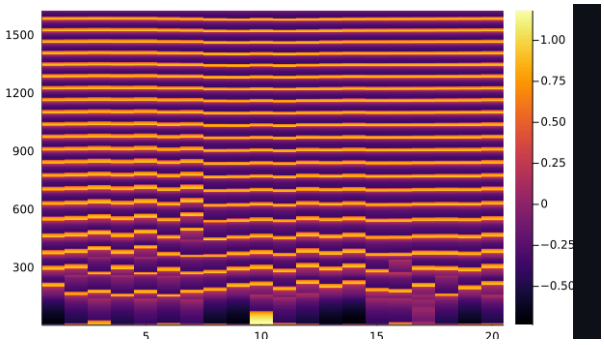

Fig. 1. example of the evolution of FHN (FitzHugh–Nagumo) system (n=20).

F. Finding peaks

Input:

1) $data$(n dimensional array)

Output:

1) N dimensional Boolean array(if $X_{ij} = True$ then its is a peak)

III. Evaluation/Results

- When running the code against naive single threaded program, the performance boost is nearly 505x for tspan=10000.
- When running the code against multi threaded optimized CPU workload using numba-jit version the following fig is obtained.

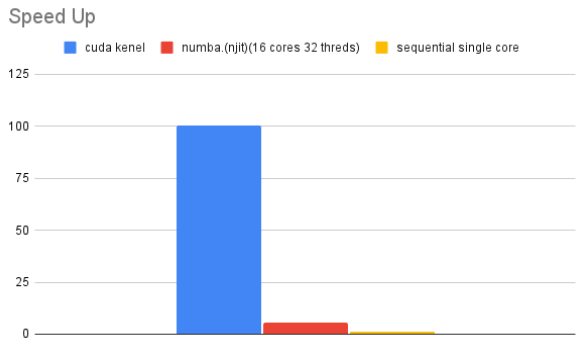

Fig. 2. performance comparison for large tspan

IV. Status of your project

- The implementation for the Discrete Differential Equations works for multiple dimensions.
- The implementation for the Continuous / Ordinary Differential Equations is limited to 2 dimensions.

V. Future Improvements

A. Solver

- To implement Dynamic shared memory.
- Implement async method to copy to global memory.
- Add the ability to use different types of numerical solving methods.
- Include support for multidimensional systems.

B. Finding peaks

- Include support for multidimensional systems.

VI. CONCLUSION

A. Solver

Large Performance improvement is achieved by the CUDA kernel and the use of shared memory; the performance improvement is almost 500x when the number of systems (N) is large and 100x for large(n) and small (N). The difference between a sequential algorithm and a parallel kernel will only get larger as n increases or dimensions increase.

B. utility Function

Implementation for the utility kernels in GPU also have a large performance improvement and throughput. In our kernel to identify peaks, it is obseved that the implementation of a parallel version have vast improvement over others.