Optimization and parallelization of tensor and ODE/PDE computations on GPU

Anirudh Sundar Subramaniam

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Computer Science & Engineering

June 2018

Approval Sheet

This Thesis entitled Optimization and parallelization of tensor and ODE/PDE computations on GPU by Anirudh Sundar Subramaniam is approved for the degree of Master of Technology from IIT Hyderabad

[illegible]

(PROF. SPARSH MITTAL) Examiner
Dept. of Computer Science & Engineering
IITH

M.V. Panduranga Rao

(PROF. M.V.P. RAO) Examiner
Dept. of Computer Science & Engineering
IITH

[illegible]

(Dr. Ramakrishna Upadrasta) Advisor
Dept. of Computer Science & Engineering
IITH

[illegible]

(PROF. SUBRAHMANYAM KALYANASUNDARAM) Chairman
Dept. of Computer Science & Engineering
IITH

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)

(Anirudh Sundar Subramaniam)

CS16MTECH11002

(Roll No.)

Acknowledgements

I would first like to thank my Thesis Advisor, Prof. Ramakrishna Upadrasta, whose advice was invaluable not only in my Thesis, but to improve myself as a researcher and person in general. This work wouldn't have been achievable without his constant guidance and encouragement.

I would like to thank my parents who were with me every step of the way. They shared everyone of my difficulties throughout my life and have always helped me overcome them. I would not be here if not for my parents who supported every one of my decisions. I would also like to thank my other family members for their blessings and advises whenever I needed.

I am also extremely grateful to my colleagues at IIT Hyderabad, with whom I had a lot of helpful technical discussions, regarding any problems I was working on. I loved the discussions with all of them, and learned a lot from all of them.

Last but not the least, I feel blessed to have been a part of IIT Hyderabad, and it's amazing people, for the past two years. I learned a lot over here and I'd like to thank everyone who gave me any sort of help, advice and support during this time.

Dedication

To my family, friends and teachers.

Abstract

We propose a multi-level GPU-based parallelization algorithm to solve the multi-compartment Hodgkin Huxley (HH) model equation that requires solving the Hines matrix. We use a 'parallel-in-time' algorithm (like the Parareal strategy) for obtaining outer level parallelism, and an Exact Domain Decomposition (EDD) algorithm with fine-decomposition for inner-level parallelism. We show that our technique can also be applied to any differential equation like the heat equations which induce tridiagonal systems.

Typically, a solution to the HH equation runs for hundreds to tens of thousands of time-steps while solving a Hines matrix at each time step. Previous solutions by Michael Mascagni et al. (1991) and Hines et al. (2008) to this problem have tackled only solving the Hines matrix in parallel.

Our approach uses the dynamic parallelism of CUDA to achieve multi-level parallelism on GPUs. Our solution outperforms the sequential time method on standard neuron morphologies upto 2.5x. We also show that iterative part of parareal method converges in 5-7 iterations on average with an accuracy of 10^{-6}.

We also propose a GPU optimization for the Higher Order Tensor Renormalization Group problem, where the tensor contraction operations inside HOTRG is optimized by a multi-GPU implementation using cuBLAS xt API.

Contents

Chapter 1

Introduction

This report talks about work on three separate problems. The first and second problems are optimization problems where the goal is to make the computation faster, and this was done by implementing on GPU and taking advantage of its parallelism. The third problem was an implementation work that takes an outdated open source tool, and updates it to the latest version. A brief introduction to the three problems is given below.

1.1 Multi-level parallel Hines solver

The main aim of this work was to try to extract multi-level parallelism for solving the Hodgkin Huxley set of differential equations, which are used to model the action potential of a neuron. This work takes advantage of two separate parallelization algorithms and combines them to achieve multi-level parallelism to solve the set of equations and the results are obtained on a GPU

1.2 Optimizing Higher Order Tensor Renormalization Group

This work aims to optimize an algorithm from the physics domain, where the algorithm tries to approximate a single large-rank tensor with a network of smaller-rank tensors. The major computation being analyzed in this problem was tensor contractions and results are shown on how to optimize this on GPUs.

1.3 Legup: High Level Synthesis with LLVM

This work was a purely implementation based work, where the goal was to port the LLVM version used in an open source high level synthesis tool called legup to the latest version. This work was needed as the Legup tool is not maintained and the last version of Legup used a very old version of LLVM. Porting LLVM in Legup provides a Verilog backend for the current version of LLVM.

Chapter 2

Multi-level parallel Hines solver

2.1 Introduction And Motivation

It is well known that Differential equations, both ordinary and partial, are one of the most significant class of computational problems. Many real-world simulation programs from physics, kinetics, and neuroscience are modeled as a set of differential equations whose solutions are intractable; they can only be solved numerically. Examples of such equations are black-scholes, Poisson's equation, Solow–Swan model etc. Further, these equations are solved using implicit numerical methods for their stability. Solving a differential equation using an implicit method requires solving a set of equations. Some of the implicit techniques are implicit Backward Euler, Crank-Nicholson, etc.

Solving a system of linear equations is well studied in Linear Algebra. On solving systems of Ordinary/Partial differential equations that are dependent on time using implicit methods, the given system of linear equations has to be solved once for every time step. At each time step, the system to be solved has a coefficient matrix, which has the same structure, but the values change, and they might depend on previous time step values.

A plethora of work has gone into optimizing/parallelizing the computations for solving a system of linear equations. There are both direct solver algorithms like Gaussian elimination, LU factorization, Cholesky factorization, etc. which give exact solutions and iterative solutions like Jacobi, Gauss-Seidel, Successive Over-relaxation, Krylov subspace methods like Conjugate Gradient, Minimal Residual method, etc. These techniques try to improve the execution time required to solve a system of linear equations by taking benefit of the properties of the matrix.

However, when solving systems of differential equations, these ideas only help with speeding up the time taken for a single time step. When solving for a large number of time steps, each time step requires the solutions to previous time step. Several ideas have been proposed to parallelize the code across time steps. [1], [2], [3]. These ideas can be broadly categorized into four categories. (i) Domain decomposition and waveform relaxation methods, (ii) Multigrid methods (iii) Multiple Shooting (iv) Direct methods.

This work is an attempt to achieve multi-level parallelization by combining time parallel techniques along with parallel techniques to solve systems of equations. We focus on solving the differential equation of Hodgkin-Huxley(HH) model from neuroscience [4], which models the action potential of a neuron membrane. Previous work to parallelize the solution to HH equation focused only on solving the linear system that arises at each time step [5] [6] [7] [8]. Not much work has been done on parallelizing the solution over multiple time steps.

2.2 Tridiagonal and Hines Matrix

2.2.1 Tridiagonal Matrix

There are different types of sparse matrices which appear when solving PDEs using implicit methods. One of the most well known among them is the Tridiagonal matrix. A tridiagonal matrix is a matrix which has non-zero elements only on its main diagonal, sub-diagonal and super-diagonal of the main diagonal in each row. Such matrices belong to the specific type of more general class of matrices known as the banded matrices which contain a diagonal band of elements on both sides of its main diagonal. Example of a matrix is given in figure 2.1:

$$\begin{bmatrix} 2 & 1 & & & & \\ -1 & 2 & 1 & & & \\ & -1 & 2 & 1 & & \\ & & -1 & 2 & 1 & \\ & & & -1 & 2 & 1 \\ & & & & -1 & 2 \end{bmatrix}$$

Figure 2.1: Example of a famous tridiagonal matrix called Toeplitz matrix that comes up in many places including when solving the heat equation or the linear dendritic cable equation.

One example of a PDE where tridiagonal matrices arises is the cable equation for a linear dendritic cable given by equation 2.1:

$$C\frac{\partial V}{\partial t} = \frac{a}{2R}\frac{\partial^2 V}{\partial V^2} - gV \tag{2.1}$$

Implicit methods are preferred over explicit methods for these equations due to better numerical stability. When implicit finite difference methods are applied to the PDEs arising from such models, the resulting equation requires solving a system of linear equations where the coefficient matrix is of some sparse structure. For example, applying backward Euler method to the above-mentioned cable equation 2.1 leads to the equation :

$$C\frac{V_i^k + 1 - V_i^k}{\Delta t} = \frac{a}{2R}\frac{V_{i+1}^{k+1} - 2V_i^{k+1} + V_{i-1}^{k+1}}{(\Delta x)^2} - gV_i^{k+1} \tag{2.2}$$

Rearranging this equation provides us a system of linear equations where the coefficient matrix is formed by the tridiagonal matrix V_{i+1}, V_i, and V_{i-1}, where V_i represents the voltage of the i^{th} compartment of the neuron. The accuracy of multi-compartment models depends on many things including the number of compartments and the length of each compartment. Thus there might be models where the number of compartments can range from a few hundred for simpler experiments to tens or hundreds of thousands for very complex experiments. Such experiments require solving Linear system involving matrices of size hundreds of thousands. Clearly, this states the requirement of efficient algorithms.

With the cable equation for a linear dendritic model, as shown in equation 2.1, we obtain a tridiagonal matrix which has to be solved every time step. Solving tridiagonal matrices and in general banded matrices is a well-studied problem [9]. One of the famous algorithms to solve a tridiagonal system of equations is the TDMA(Tridiagonal Matrix Algorithm) also known as the Thomas' Algorithm by Thomas et al. [10] This algorithm is sequential. There are many more parallel algorithms that have been devised to solve Tridiagonal systems of equations including the Cyclic Reduction and parallel cyclic reduction [11] [12] proposed by Stone et. al., or some recent ones like the SPIKE Algorithm proposed by Sameh et.al. [13] [14] [15] for the general banded matrices case.

2.2.2 Hines Matrix

In the cable equation 2.1, when a branched dendrite is modeled instead of a linear dendritic model, at the junction of a branch, we will have equations where the diagonal element

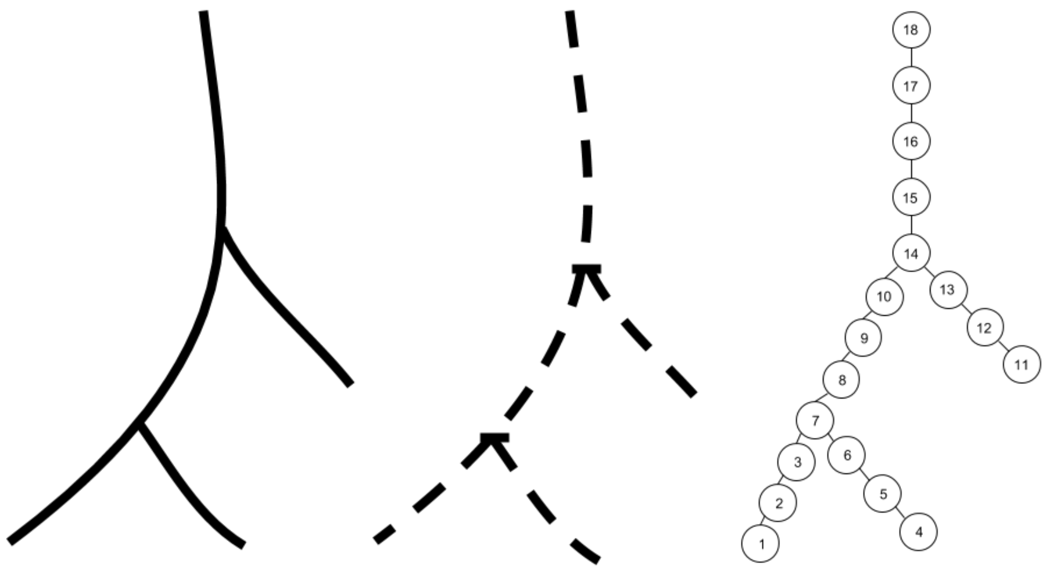
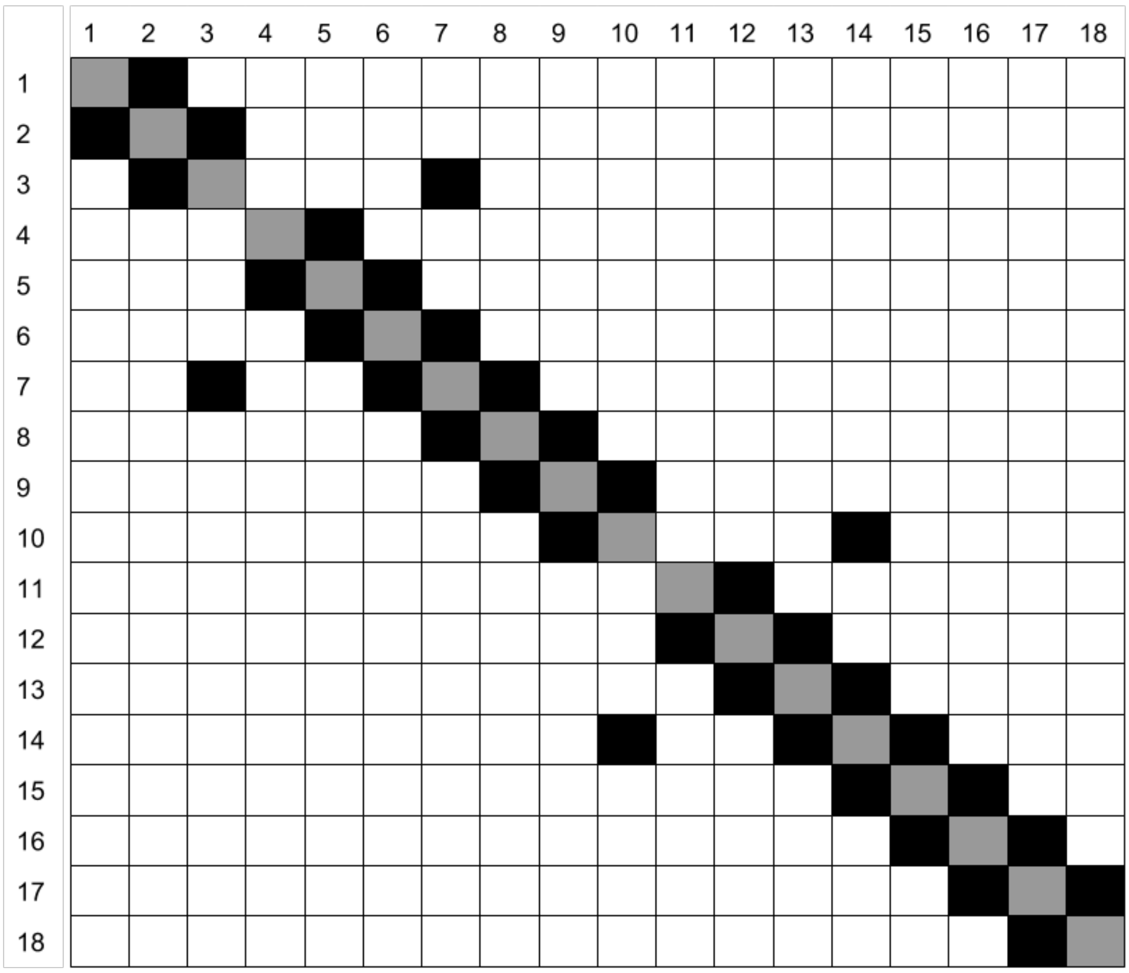


Figure 2.2: Example of compartmentalizing the neuron and visualizing it as a tree. The tree has been numbered in depth first order. The adjacency matrix of this tree will be the Hines Matrix.

depends on some (possibly multiple) off-diagonal elements. This can be modeled as a tree. In this case, the regular tridiagonal matrix algorithms don't directly work. In 1984, Michael Hines et. al. [16] provides a way of using a modified form of TDMA algorithm to solve this system of equations. Hines' idea was to number the nodes of the tree in a depth-first manner so that the child is always numbered less than the parent. With this ordering, we can construct the adjacency matrix of the tree where each row depends only on its lower rows. This matrix is known as Hines matrix and is of interest in modeling neurons as most dendritic structures are branched.

An example of how the Hines matrix is constructed from the compartmentalized neuron model is given in Figure 2.2.

The adjacency matrix of tree constructed in Figure 1.2 gives the Hines matrix, shown in Figure 2.3.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	▨	■																
2	■	▨	■															
3		■	▨				■											
4				▨	■													
5				■	▨	■												
6					■	▨	■											
7			■			■	▨	■										
8							■	▨	■									
9								■	▨	■								
10									■	▨				■				
11											▨	■						
12											■	▨	■					
13												■	▨	■				
14										■			■	▨	■			
15														■	▨	■		
16															■	▨	■	
17																■	▨	■
18																	■	▨

Figure 2.3: Hines matrix corresponding to the tree in Figure 1.2

2.3 Related Work

2.3.1 Background on Neuroscience

The significant parts of studies done in computational neuroscience are Neural Encoding and Decoding, Information Theory, Modeling electrical properties of a neuron and Modeling neuron networks to understand learning. Among this, modeling a neuron and its morphology is done to know its electrical properties. A. L. Hodgkin and A. F. Huxley introduced one such famous model in their seminal paper in 1952 [4], which describes the electrical characteristics responsible for the generation (Single compartment model) and propagation (Multi-compartment model) of action potentials in a neuron. The Hodgkin Huxley model attempts to model the action potential of a neuron using the below differential equations.

$$\begin{aligned}
C_m \frac{dV}{dt} &= -g_L(V - E_{\mathrm{L}}) - g_{Na} m^3 h (V - E_{Na}) - g_K n^4 (V - E_{\mathrm{K}}) + I \\
\frac{dm}{dt} &= \alpha_m(V)(1-m) - \beta_m(V)m \\
\frac{dh}{dt} &= \alpha_h(V)(1-h) - \beta_h(V)h \\
\frac{dn}{dt} &= \alpha_n(V)(1-n) - \beta_n(V)n
\end{aligned}$$

The compartmentalized neuron can be viewed as a tree where each node is a compartment and edge between the nodes representing the transfer of membrane potential along that edge. The multi-compartment model tries to approximate the continuous membrane potential of a neuron by splitting the neuron into regions or compartments and computing the discrete set of values for these compartments [17].

2.3.2 Prior work on solving Hines matrix

Modeling a neuron as a tree leads to a system where the coefficient matrix is the Hines matrix. Hines et al. (1984) [16] proposed a modified Gauss Elimination Algorithm with a complexity of O(N), in which N refers to the number of rows in the given Hines matrix. Hines also proposed parallel algorithms to solve the system [5] [6]. Mascagni et al. (1990) [7] and Larriba Pey et al. (1994) [18] came up with Exact Domain Decomposition(EDD), which can be applied to Hines matrix.

EDD algorithm decomposes the matrix into smaller matrices which are tridiagonal and solves them. It then computes the final solution using solutions to these smaller matrices and solutions to a **"domain"** matrix. This algorithm is explained in detail in section Hines matrix using EDD

Roy Ben-Shalom et al. (2013) [19] proposed a way of solving the system using GPUs to accelerate compartmental modeling.

2.3.3 Prior work on Temporal Discretization

Generally, the parallel space-time algorithms can be classified into four categories based on how the parallel solver strategy is applied. They are methods based on multigrid, direct time-parallel methods, multiple shooting and methods based on domain decomposition and waveform relaxation.

The first work in the parallel algorithm concerning time domain was by Nievergelt et al. in 1964 [1]. He proposed a parallel time-integration algorithm which belongs to direct time parallel method category. The main idea was to spit the entire time interval into smaller sub intervals. Each sub interval is solved concurrently. Achieving parallelism at the cost of repetition of computation. Then later, Miranker in 1967 [2] came up with predictor-corrector methods. Here also the idea was to evaluate multiple time step values simultaneously.

In 1999 Gander et al. came up method using domain decomposition and waveform relaxation [3]. The first method of the multigrid category was proposed by Hackbusch et al. in 1984 [20]. For initial guesses, rough values are used. These methods are not parallel by default, but their parts can be executed in parallel space-time domain as simultaneous work is done on entire space-time domain. The recent development in this domain is by Christlieb et al. [21] in 2010 with a goal of creating parallel time integration algorithms suitable for multicore architectures and resulted in small-scale parallelism. Later Guttel et al. in 2012 proposed a direct time parallel method which relies on overlapping time decomposition method [22].

2.3.4 Contributions

The major contribution of this work is to apply parallelism to solve the Hodgkin-Huxley equation. We take advantage two algorithms that provide solutions for parallel execution in two separate problems. We combine the solutions to the two problems and extract multi-level parallelism. We use Parareal algorithm by Lions and Maday [23] to extract parallelism in time domain. We use the Exact Domain Decomposition based algorithm by Mascagni [7] to exploit parallelism at the inner level, where solution to the linear system with Hines matrix is needed. Further, both of these solutions are implemented on GPU with the help of CUDA's dynamic parallelism

2.4 Parareal Algorithm

Parareal is a parallel in time algorithm, first proposed by Lions and Maday in 2001 [23]. Parareal could be categorized as either multigrid method in time or as multiple shooting methods among the temporal discretization methods.

Parareal algorithm solves an implicit finite-difference equation in parallel across time. It decomposes the time dimension into N smaller time parts, each of which will be solved in parallel. This is achieved by using two finite-difference methods applied iteratively. The first one is a coarse iteration, denoted as $\mathscr{G}$ and a fine iteration method, denoted as $\mathscr{F}$.

Normally a type of Runga-Kutta method is used for both the coarse and fine iterations, where the coarse iteration has lower accuracy by taking large steps, and the fine iteration uses small steps. The Coarse iteration if first done sequentially, and then the solution at each step of coarse iteration are used as initial values to compute fine steps in parallel. This process is repeated iteratively until an error convergence is achieved. The computation of solutions from coarse and fine iteration solutions is given below:

$$y_{i+1}^{k+1} = \mathscr{G}(y_i^{k+1}, t_i, t_{i+1}) + \mathscr{F}(y_i^k, t_i, t_{i+1}) - \mathscr{G}(y_i^k, t_i, t_{i+1}) \tag{2.3}$$

An illustration of the parareal algorithm working with the coarse and fine grain iterations is shown in figure 2.4.

The authors of Parareal also gave a truncation error analysis, which says that the error is of order k. There was also a convergence analysis of the solution analyzed by Gander in 2008 [24].

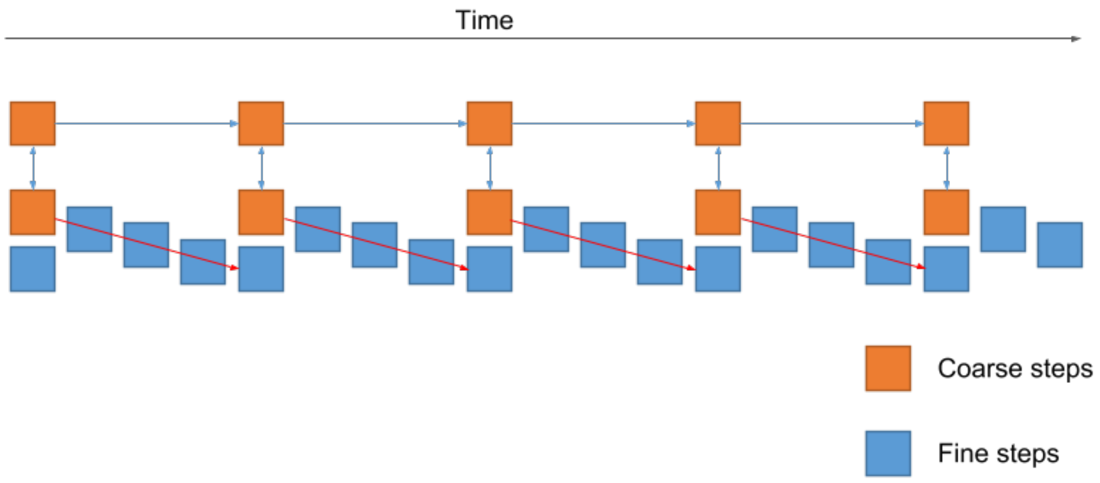


Figure 2.4: Illustration of parareal algorithm

2.5 Hines Matrix Solver using EDD

The main idea behind EDD algorithm is that all branches are connected to one or two junction nodes. Thus the solutions to the branch can be derived as the solutions to two/three solutions. One solution is the solutions to the branch equations with junction node value assumed zero. Second and third as the branch equations with "left" or "right" junction node value assumed one and the right-hand side assumed zero.

The solution to each node in a branch can be written as a linear combination of solutions to sub-domain matrices

$$x_i = x_{0i} + x_{1i} * x_{j1} + x_{2i} * x_{j2} \tag{2.4}$$

Where x_i refers to the solution to the i^{th} node and x_{0_i} refers to the solution of the node when the junction node was assumed zero. x_{1_i} and x_{2_i} refer to the solution of the node when the first and second junction node connected to the branch is assumed one respectively. x_{j_1} and x_{j_2} are the correct solutions to the junction nodes connected to the branch.

x_{0_i}, x_{1_i}, and x_{2_i} for the different branches can be computed in parallel, and then substituted in the junction node equation to find the solutions to the junction nodes, which then gives solutions to the branches. For more details see Mascagni. (1991) [7]

Each tridiagonal Tr_i^r has to be solved with the value of junction nodes assumed as zero and the original RHS. This gives solutions for the x_i when junction node is 0.This is denoted as

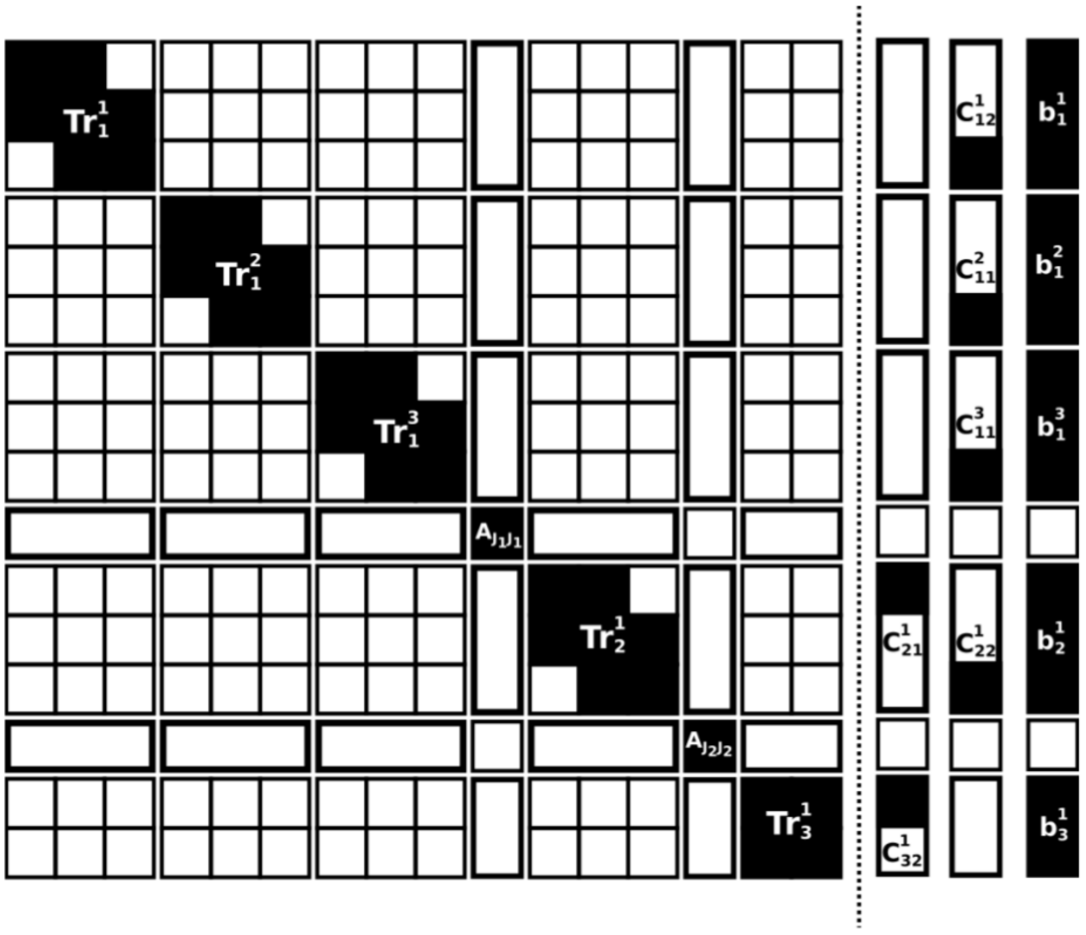


Figure 2.5: Example of EDD split on hines matrix and the corresponding 3 RHS values

x_{0_i}'s. Then Each Tr_i^r is solved with junction node value assumed as 1 and RHS as 0. This gives solutions for x_i's when junction node is 1. This is denoted as x_{1_i}'s. Then the value of each x_i from $Ax = b$ can be written i = x_{0_i} + $(x_{1_i} * x_{junction\_node})$. Thus, if we can find $x_{junction\_node}$ for each junction, we can substitute that to find x_i's. Junction node values are found by substituting x_{0i}'s and x_{1_i}'s in the equation containing junction node, which gives a system with just junction node values as unknown. This system is the domain matrix, and solving this gives all the solutions.

Note: EDD algorithm can also be applied directly to the adjacency matrix of any undirected graph including tridiagonal matrices, which are a result of a linear chain.

2.6 Combining Parareal and EDD algorithm

The parareal algorithm explained in 2.4 provided a way of computing the solutions to the equations in parallel across time. This algorithm was introduced for general differential equations. When using a time-implicit finite-difference approximation for ordinary/partial differential equations, we would need to solve a system of linear/non-linear equations at every time step.

In case of Hodgkin Huxley equation, this system turns out to be the hines matrix, which can be solved in parallel using the Exact Domain Decomposition (EDD) algorithm.

Thus, combining the two parallel algorithms, we can compute the solutions to the original Hodgkin-Huxley system of equations for the multi-compartment model. This combination of algorithms provides a multi-level parallel solution. We implement this combination of algorithms in CUDA to take advantage of the huge parallelism capability of modern GPUs.

2.7 Experimental setup and implementation details

The results were tested on NVIDIA P100 GPU with 12GB memory. The implementation of the combinations of both Parareal and EDD algorithms to find multi-level parallelism was done in NVIDIA CUDA C++ language. Multi-level parallelism requires calling a GPU kernel from within another GPU kernel. This was done by taking advantage of CUDA's dynamic parallelism support. An overall architecture of our implementation is shown in Figure 2.6.

The parareal algorithm part of the algorithm calls the EDD algorithm at every time step. The coarse iterations in parareal algorithm are done in a sequential manner, and the fine iterations are done in parallel. Thus the EDD algorithm is called in parallel by the fine iterations, and each call to EDD further calls the TDMA implementation to solve tridiagonals in parallel.

After every step of the parareal algorithm, the diagonals and RHS of the original matrix are updated. After every iteration of the entire parareal algorithm, the error between the coarse and fine iterations is computed and used to update the next set of solutions. This procedure is repeated until convergence.

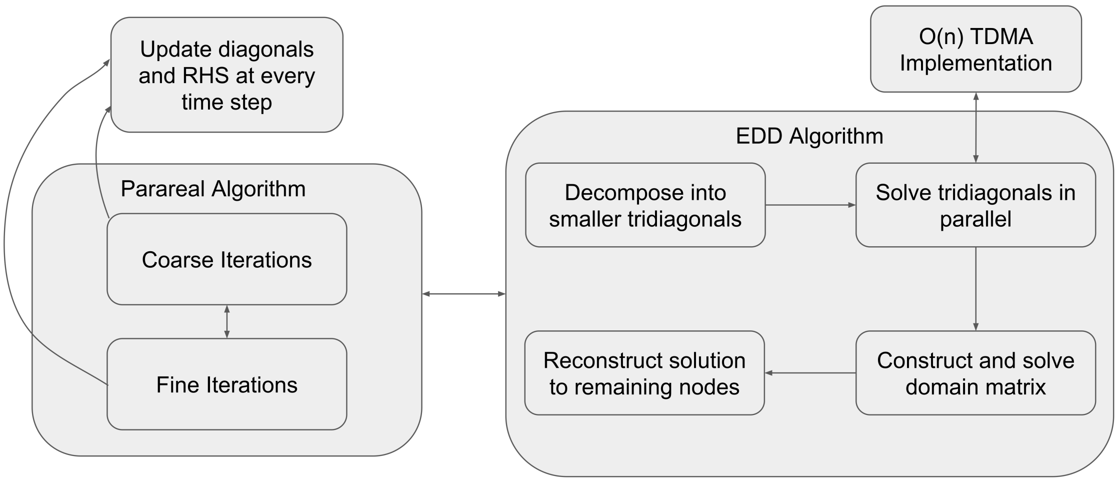

Update diagonals and RHS at every time step
Parareal Algorithm
Coarse Iterations
Fine Iterations
O(n) TDMA Implementation
EDD Algorithm
Decompose into smaller tridiagonals
Solve tridiagonals in parallel
Reconstruct solution to remaining nodes
Construct and solve domain matrix

Figure 2.6: Architecture of the implementation. The parareal part of the algorithm calls EDD algorithm implementation at every time step.

2.7.1 Matrix storage details

In this subsection, we will look at how the original hines matrix is stored. The main matrix is stored in 3 vectors D, U, and P. The D vector stores the main diagonal elements and the U vector stores the only non-zero element that exists after the main diagonal. The P vector stores the column index of the values stored in U vector. An illustration of the D and U vector stored is shown in Figure 2.7

2.8 Results

Figure 2.8 shows the correctness of the algorithm, which was obtained by substituting the correct values on the Hodgkin-Huxley equation. The points of spike in the voltage shows the points where the external current was applied. This plot shows the correctness of the values used in the experiment.

Figure 2.9 shows the error rate convergence on applying the parareal algorithm for time parallel code. As you can see, the error rate goes down to 10^{-6}, in the 6^{th} iteration. The number of iterations for the parareal algorithm for convergence could depend on the actual values applied, but typically number of iterations between 5 to 8 is considered acceptable.

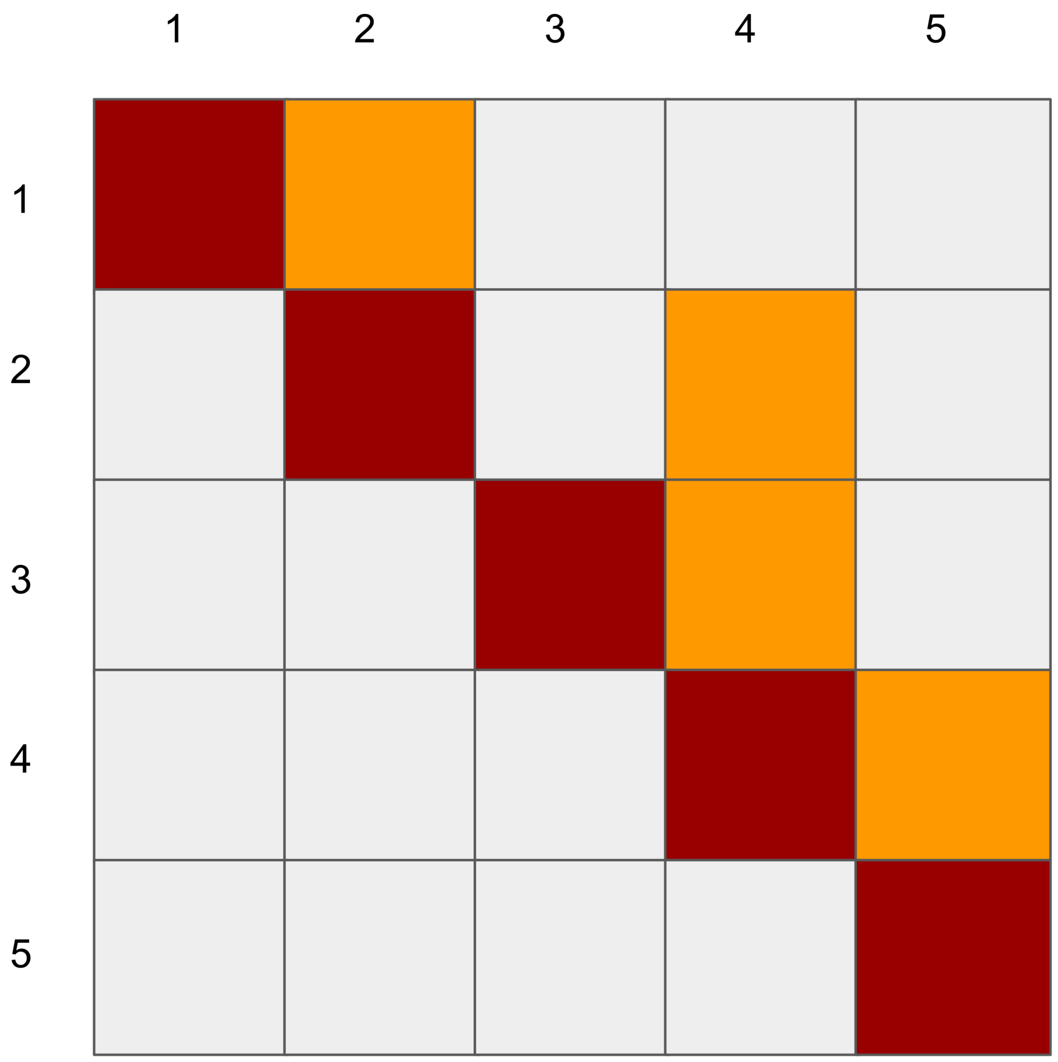


Figure 2.7: Shows the D and U vector stored. Since Hines matrix is symmetric, the lower triangular values need not be stored.

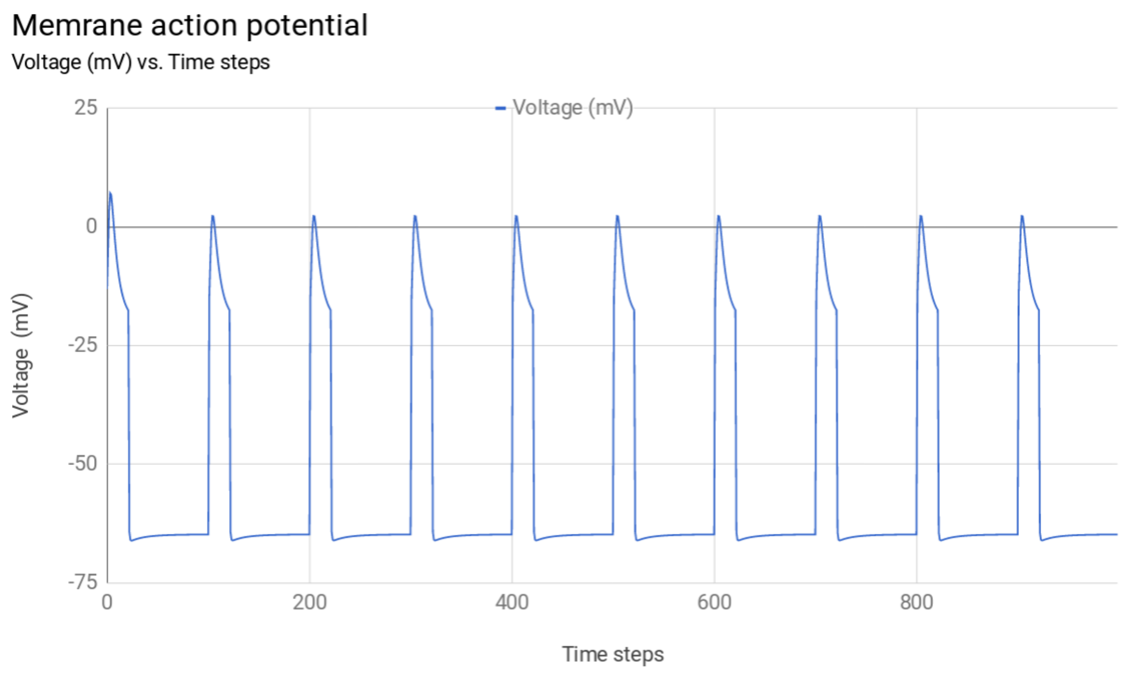


Figure 2.8: Action potential showing neuron firing, when external current is applied

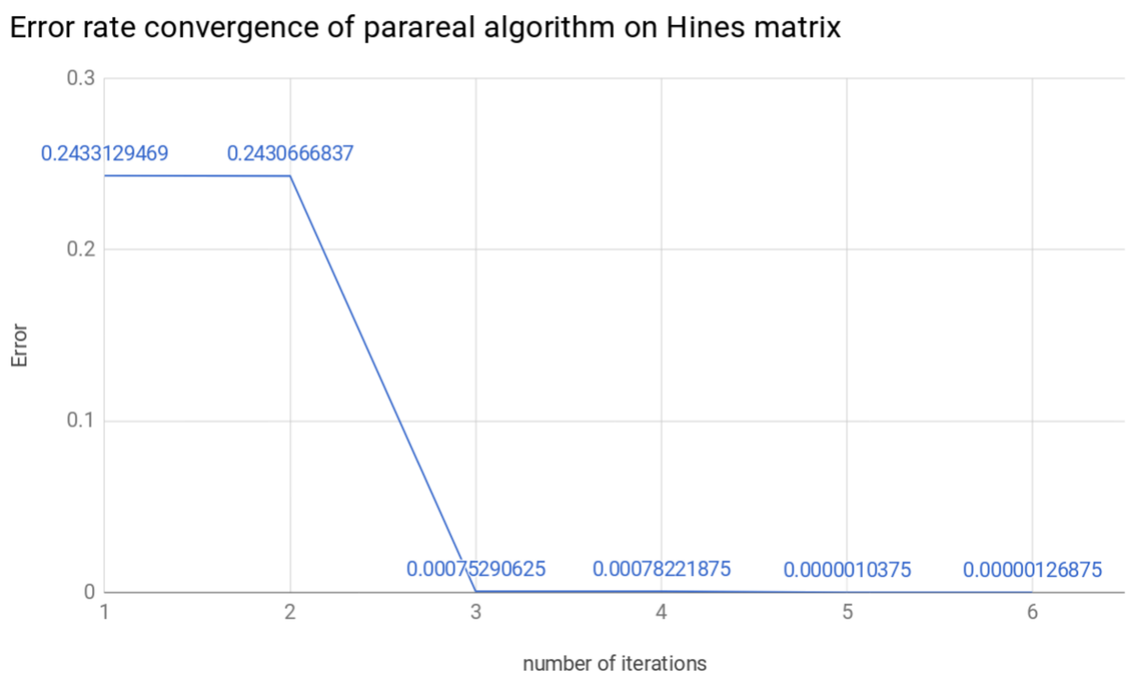


Figure 2.9: Error rate convergence when solving using parareal algorithm

Figure 2.10 shows speedup in GPU with respect to CPU, when performing the multi-level parallel code. As normally known in literature and shown in the plot below, when the size of the Hines matrix is comparatively small (i.e. less than 1000), it is better to run the algorithm on CPU, but when the size of the Hines matrix is large, the speedup when compared to CPU is considerably good.

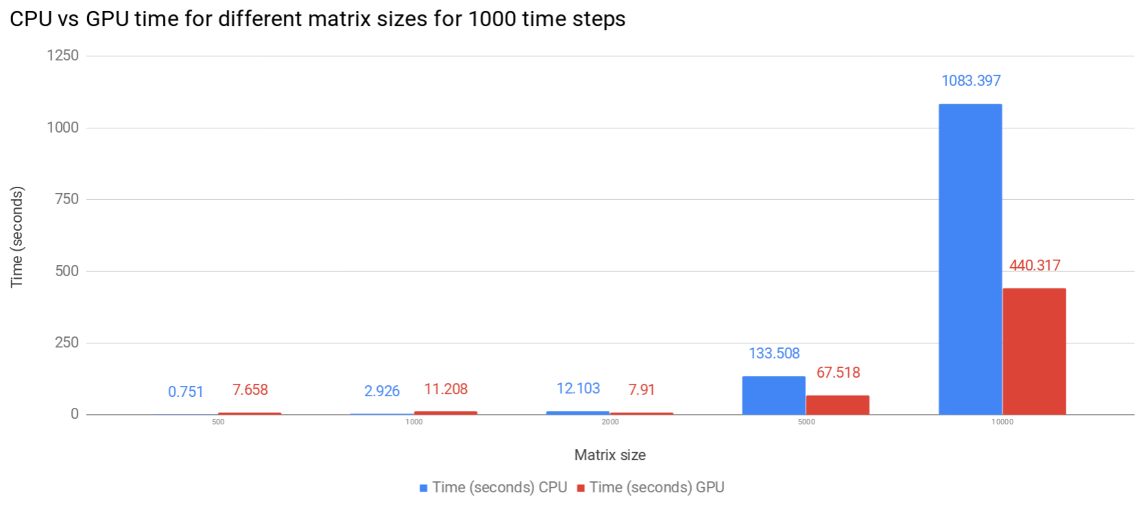


Figure 2.10: Speedup in terms of seconds when compared to CPU code

2.9 Future Work

There are a lot of opportunities for improvement in this model of solving the Hodgkin-Huxley equation. One of the major improvements that is possible would be to try to use other parallel in time algorithms like PITA [25], PFASST [26] [27], etc.

Another major avenue to try would be to explore ways of avoiding the coarse computations after fine computations in every time step of Parareal algorithm. This is important because avoiding the coarse computation at every time step helps avoid the repeated creation and destruction of threads in the GPU. The kernel call can be made for Fine computations once and the created threads can be used to compute the entire solution.

This work helps Action Potential accurately for a single neuron by modeling it using the multi-compartment model. However, to understand the brain's working we need to model a network of (possibly millions) neurons. modeling networks of neurons would capture a complete picture of the electrical characteristics of the entire brain. So far, we modeling a network of neurons is normally done using what are known as "Integrate-and-Fire" (IF) [28] models. These models do not try to action potential of the neuron, instead, they assume that as soon as a threshold is reached, the neuron immediately fires. The differential equations used in IF models are comparatively much simpler and can be analytically solved.

To capture a realistic working of neurons in a neural network, the Action Potential should be modeled, which can be done using the Hodgkin-Huxley model. Thus the problem of considering Hodgkin-Huxley for a network of neurons is a much more important problem, but due to the high number of neurons in the brain and the numerical solutions needed for the HH based differential equations, this problem is computationally much more expensive. So, trying to use the methods in this work, to help model a network models using HH model would be a much more important and interesting work. Please look at [29] for more details.

2.10 Conclusion

The multi-level parallel solution given in this report could be massaged to be reused to solve other differential equations. One example that comes to mind is the solutions to 3D heat equation, which contains an underlying tridiagonal matrix at every time step, which can also be solved using EDD as shown in 2.5. The work in [30] shows usage of parareal type algorithm on 3D heat equation.

Chapter 3

Optimizing Higher Order Tensor Renormalization Group

3.1 Introduction

Optimizing high dimensional tensor computations is of big importance in major areas of scientific. Tensor computations are used in a lot of scientific problems in quantum physics, quantum chemistry, and many other fields. Some of the major computations on tensors are tensor addition, tensor contraction, cross product, etc. Among these computations, tensor contraction is heavily used in this algorithm. We will discuss about tensors and tensor contraction in the next section.

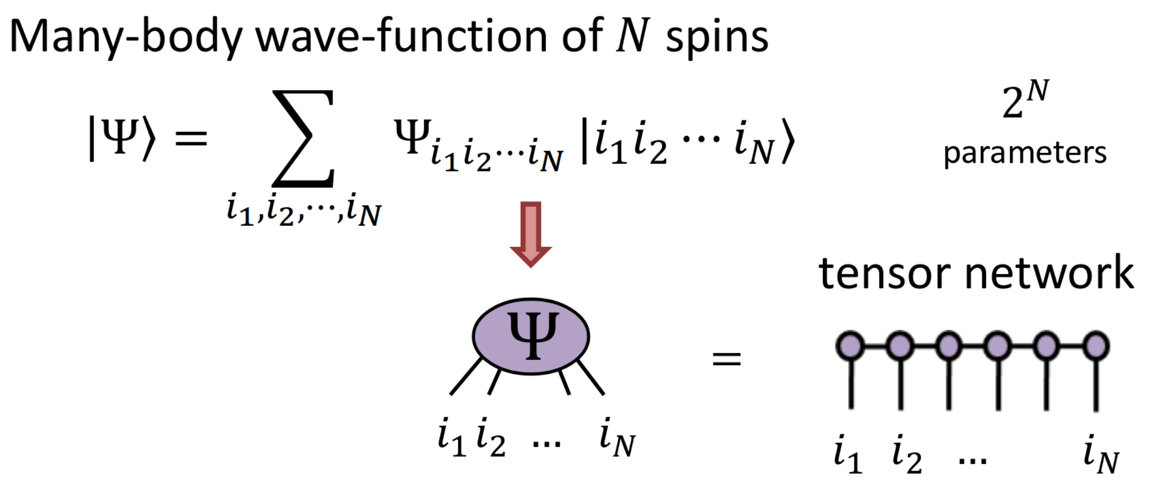
This section discusses about the optimization and parallelization challenges for a tensor contraction based algorithm, that tries to compute the emergent phenomena in quantum many-body systems. The big picture of the problem being solved is given a bunch of electrons, atoms, etc. how do they organize themselves to form a metal, insulator, etc. This is done by identifying the “ground state” of the wave function.

The initial work on Tensor renormalization was by Levin and Nave [31]. This work gave the initial algorithm for tensor renormalization group which generalized the Density Matrix Renormalization Group (TRG) [32] [33] algorithm to two-dimensional classical lattice models using Singular Value Decomposition.

This was followed by Xie et al. [34], where they improved the accuracy of the original TRG algorithm called Second Renormalization Group (SRG). Finally, Xie et al. [35] also came up with a version of the Algorithm called Higher Order Tensor Renormalization

Group (HOTRG) and Higher Order Second Renormalization Group (HOSRG), which is applicable to 3D and 4D lattice based on Higher Order Singular Value Decomposition (HOSVD).

Mathematically, the algorithm approximates a tensor having large rank with a network of tensors of smaller ranks.



Many-body wave-function of N spins
$|\Psi\rangle = \sum_{i_1,i_2,\cdots,i_N} \Psi_{i_1 i_2 \cdots i_N} |i_1 i_2 \cdots i_N\rangle$
2^N parameters
tensor network
Ψ
$i_1\, i_2\, \ldots\, i_N$
=
$i_1\, i_2\, \ldots\, i_N$

Figure 3.1: Large tensor that represents the wave function approximated using tensor network. Image Source: Vidal, Burke institute, Caltech

Computationally, the problem to be solved is to optimize the memory and computing resource usage when computing tensor contraction for tensors of high dimensions.

3.2 Tensor computations

Tensors are a collection of values, possibly multi-dimensional. A tensor has a rank associated with it, which specifies the number of dimensions/directions in which values exist. A single value/scalar is a tensor of rank 0. A vector is tensor of rank 1. A matrix is a tensor of rank 2. When there are values in k dimensions, it is known as a tensor of rank 'k'. Tensors can also be thought of as multi-dimensional arrays with the number of dimensions being the rank of the tensor.

Tensors are denoted with dot with k legs, where k is the rank of the tensor. For example, the figure 3.2 shows the tensors with ranks 3 and 4.

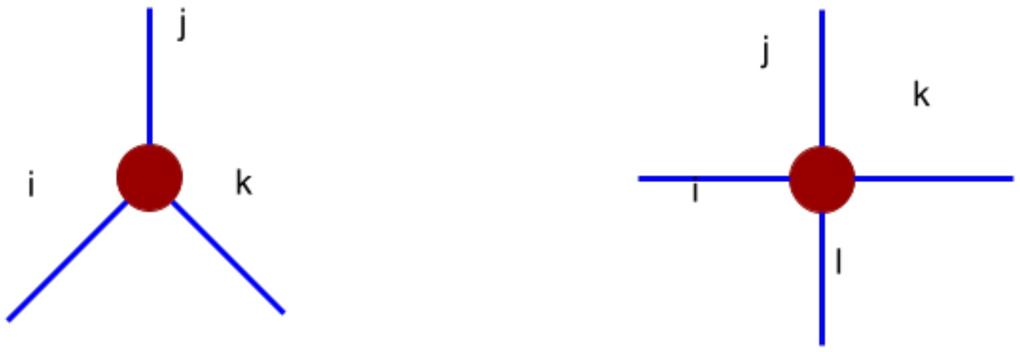


Figure 3.2: Tensor notation using dot and legs

3.2.1 Tensor networks

Tensor networks are a collection of tensors with common indices. They are denoted by connecting two dots (representing the tensors) with a common leg as shown in figure 4.1.

3.3 Operations on Tensors

Tensor Addition: Adding two tensors involving adding the values of the tensors which have matching indices. This operation requires that the two tensors have the same rank and same number of elements in each dimension.

Tensor product: Tensor product is an operation that takes two tensors say A and B and gives a product tensor whose rank is the sum of ranks of A and B. This operation is similar to the cross product operation on matrices/vectors. The operation is pairwise multiplication along the indices, the values in those indices

Tensor Contraction: Contraction is an operation that produces a tensor whose rank is 2 lesser that the ranks of the input tensors. The two input tensor contain a common index, which is the the index along which the operation is performed. The main operation is a sum of products of the two tensors along the common index.

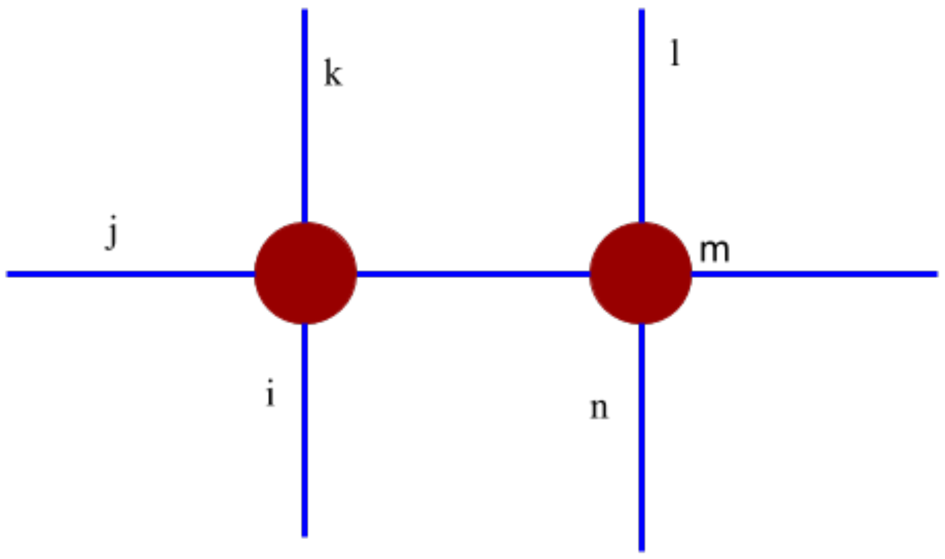


Figure 3.3: Tensor network consisting 2 tensors

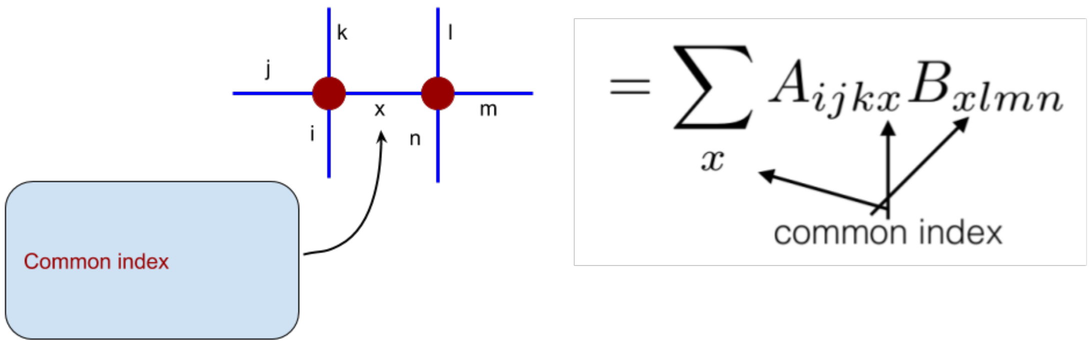


Figure 3.4: Tensor contraction on a tensor network. The common index is x

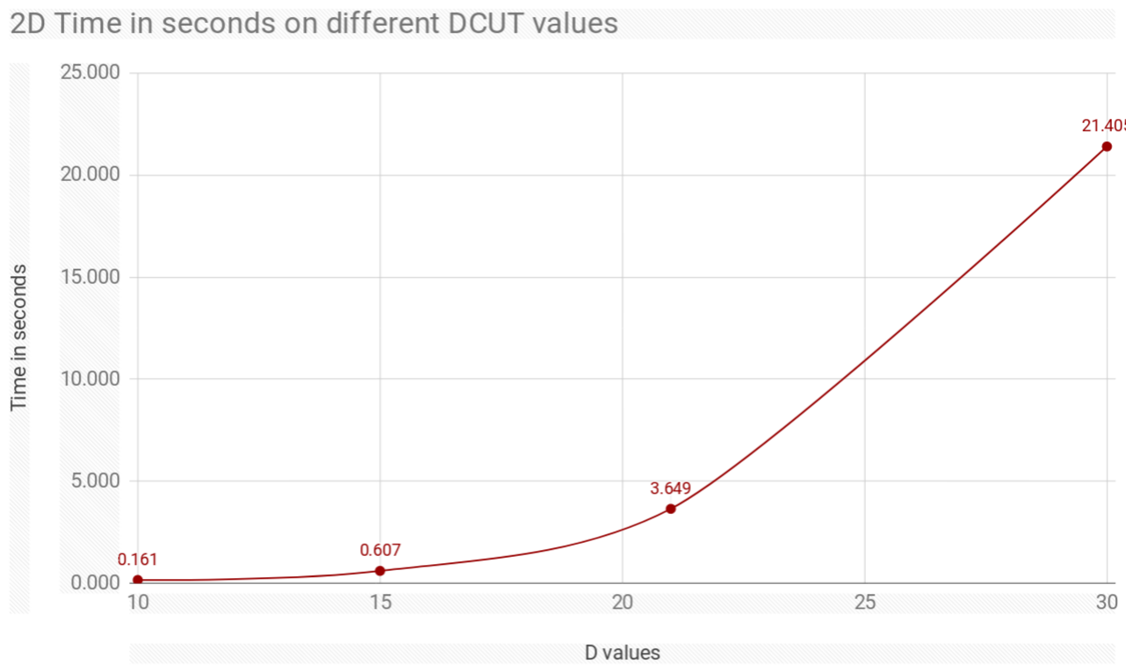


Figure 3.5: Result in 2D initial values

3.4 Initial Optimizations and results

The algorithm takes multiple parameters including the type of lattice model, and a parameter D, which specifies the maximal dimension for each index of the tensors. This parameter D specifies the truncation threshold, which is connected to the accuracy with which the original tensor is approximated with the tensor contraction. The initial runtimes for the HOTRG algorithm for 2D, 3D and 4D algorithms for varying values of D are shown in 3.5 3.6 3.7.

As you can see from the initial results above, the time taken for the entire algorithm is calculated for varying D values and for the 4D lattice case, the time taken in seconds for a D value of 5 increases to 7231 seconds. Thus the increase in time is clearly exponential with increasing D value. Hence we analyze the major type of computation that needs to be optimized.

There are two major types of computations performed in the algorithm. The first type of computation is the re-arrangement of values across the dimensions. The computations involve copying the values of the original tensor into new tensor, but rearranged as shown in 3.8.

The second important computation is the tensor contraction, which was introduced above.

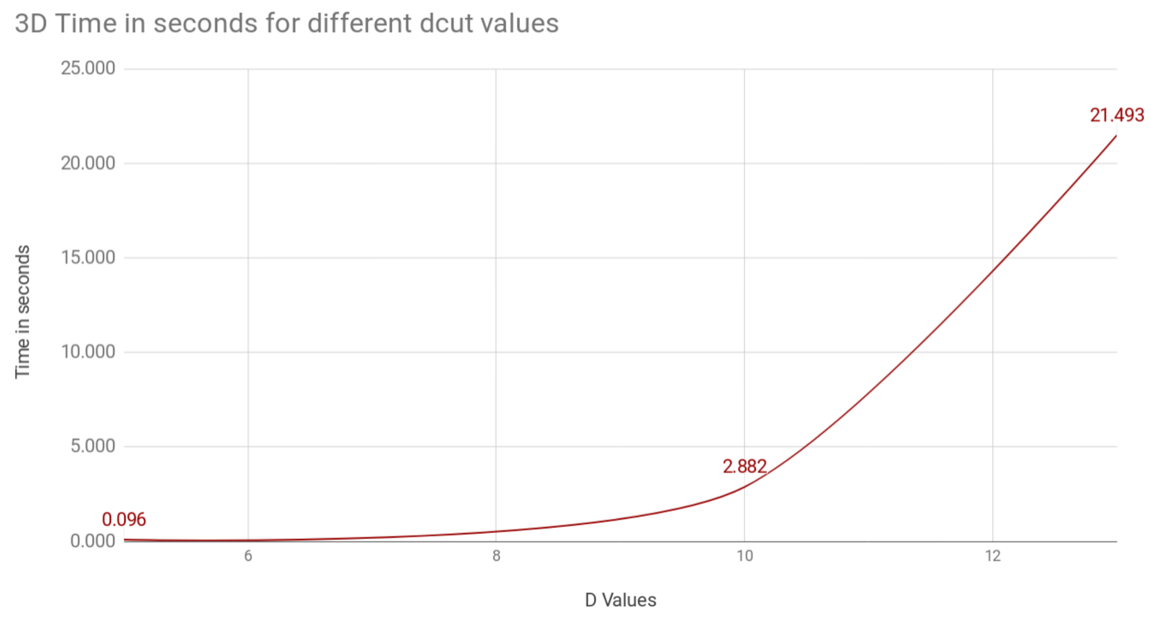


Figure 3.6: Result in 3D initial values

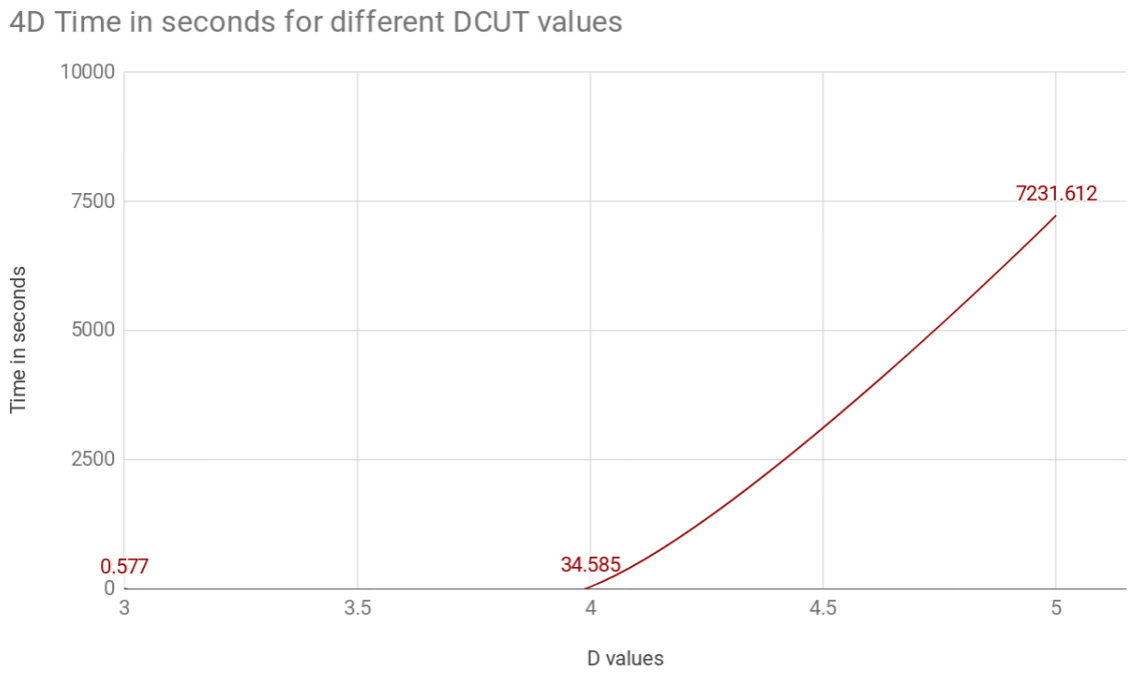
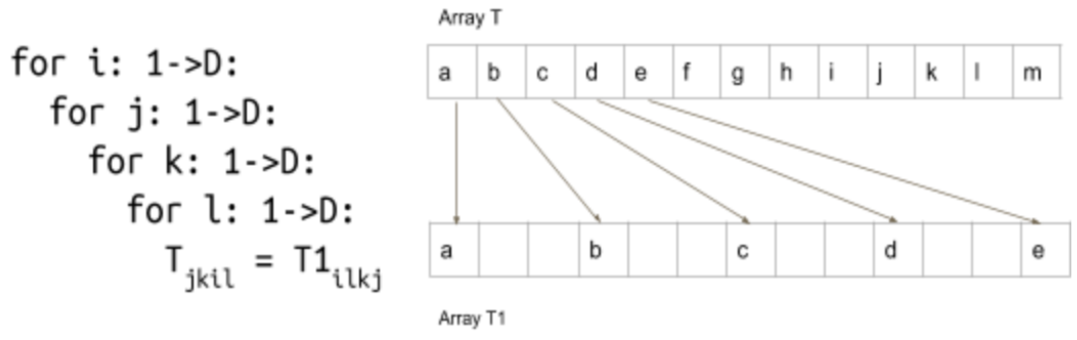


Figure 3.7: Result in 4D initial values



```
for i: 1->D:
  for j: 1->D:
    for k: 1->D:
      for l: 1->D:
        T_jkil = T1_ilkj
```

Figure 3.8: Initial reordering computation

In this particular algorithm, the tensor contraction is between two tensors, one of high dimension and another with lower dimension, but they share only one index. Hence, this computation is similar to Matrix multiplication between two matrices.

$$\sum_{x} T_{lmnxijk} * U_{oxp} \tag{3.1}$$

If the values in the tensor are stored in row major order, This computation would access non-contiguous elements in the array, which would cause cache misses for every element of the main array that stores the tensor. One major way of avoiding the cache misses is to interchange the loops, so that the array accesses for the tensor in one of the arrays are contiguous, to avoid cache misses.

The problem with loop interchange is that, if there are dependences across accesses in the array, the dependences have to be maintained in the interchanged code. We used the polyhedral optimization tool pluto to optimize the code, but the cost model of the tool did not concentrate on the loop interchange. Hence we manually performed loop optimizations and the initial performance improvements are shown in 3.9.

In the graph above names below the bars represent the function names which contain the computations. The getu* functions contains the re-arrangement of values, and gett* functions contains the Tensor contraction.

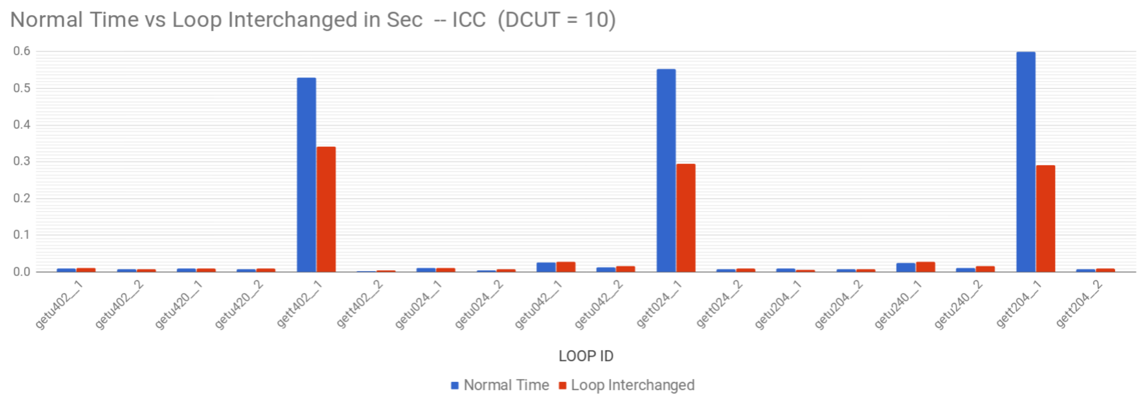


Figure 3.9: Improvement in time after loop interchange

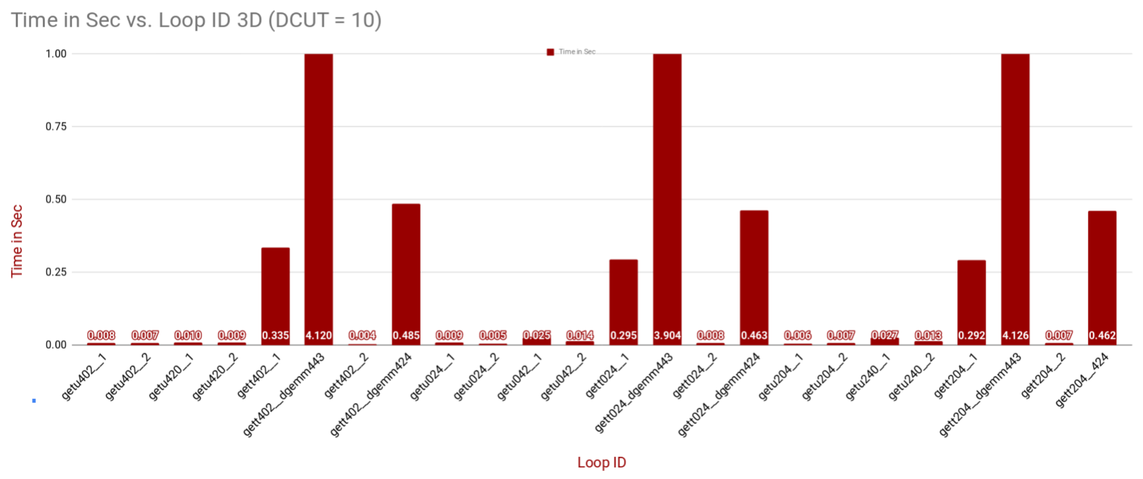


Figure 3.10: Runtimes across different procedures showing the procedure with bottleneck

3.5 GPU Optimization

After loop interchange we record the runtimes of each function in the algorithm, to determine the bottleneck computation. The graph in 3.10 shows the runtimes for each function.

As one can see from the plot above, the most time consuming part of the functions was the *gett* $*_d$ *gemm* calls. Here dgemm is BLAS (Basic Linear Algebra Subprograms) call to do general matrix multiplication on double values. The BLAS library used here was the Intel MKL library.

Since GEMM calls were the most time consuming part, we tried to optimize this call by offloading the computation to GPU. We did this by replacing the Intel MKL BLAS call to dgemm with cuBLAS library. cuBLAS is a BLAS library by nvidia which contains optimized library calls to GPU. cuBLAS also contains a library known as cuBLAS xt API,

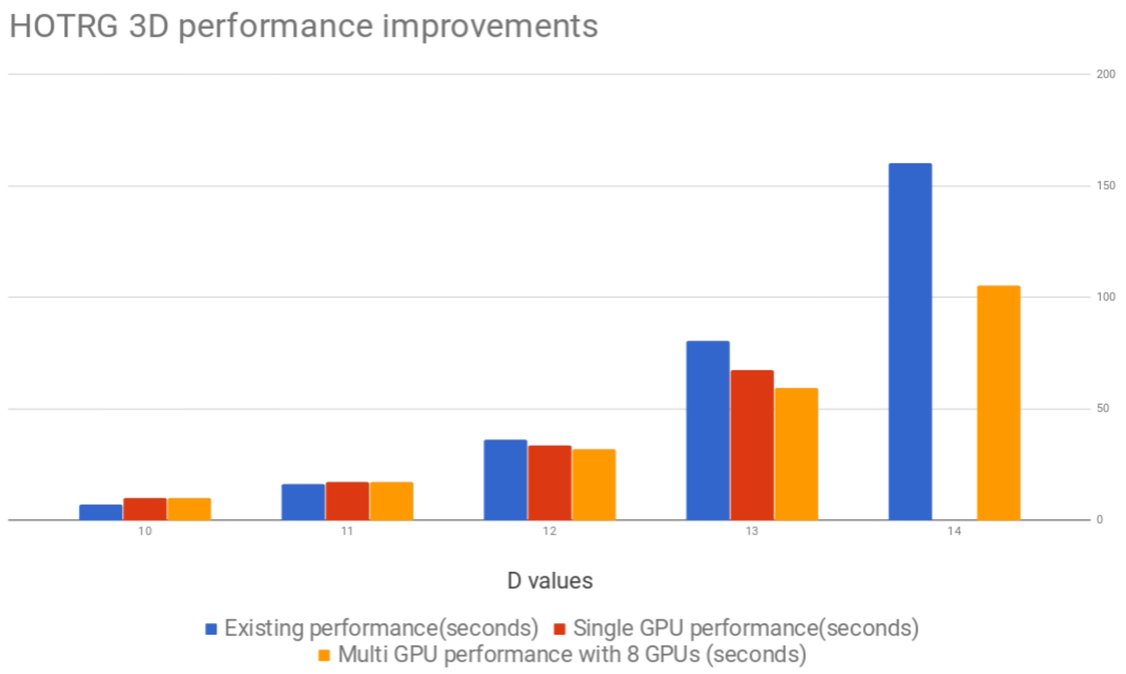


Figure 3.11: Speedup in time when run on single and multiple GPUs compared to CPU

which allows BLAS calls to multiple GPUs. We also used this test the performance on multiple GPUs. The experiments were done on nVidia cluster which had Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz with 20 cores for CPU and 8 * Nvidia Tesla K80 with 12GB RAM GPUs. The results are as shown in 3.11.

Note: DCUT 14 requires 33 GB memory, so it cannot be run on single GPU

The above plot the speedup in percentage between a single GPU and multi-GPU version of the code. You can notice from the graph, that the speedup is negative for smaller D values. This is because the overhead of splitting the matrices to be computed in multiple GPUs and the communication overhead is more than the gain. But, as the D values increase, the speed increases.

3.6 Exact Computation and possible solutions using TCE

So far, we only tried to optimize a single tensor contraction operation with loop interchange and on GPUs. But the actual algorithm for the Higher Order Tensor Renormalization contains a sequence of contraction operations. For example, for the 4D lattice case, the each tensor has to be contracted along 3 of its directions with 3 different smaller tensors. The

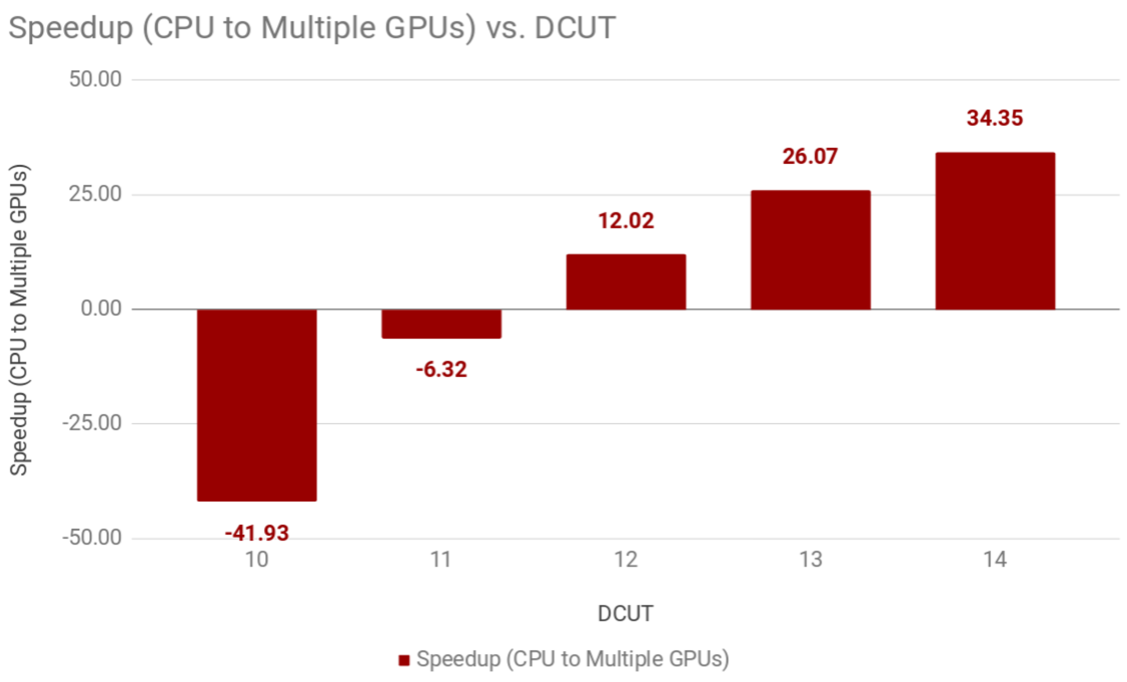


Figure 3.12: Speedup between single and multiple GPU

exact sequence of contractions are shown below:

$$T_1 = \sum_x T_{lmoxijk} * U2_{nxp} \tag{3.2}$$

$$T_2 = \sum_x T1_{lmnoixk} * U1_{xjp} \tag{3.3}$$

$$A = \sum_x T2_{lxnoijk} * U0_{mpx} \tag{3.4}$$

From the equations above, we can see that there are

1. Four input tensors a rank-8 tensor T, and 3 rank-3 tensors U0, U1 and U2
2. Two rank-8 intermediate tensors T1 and T2 and
3. One rank-8 output tensor A

A naive implementation of this computation contains first computing T1 and then T2 and finally A. This would requires 3 nine dimensional loop nests written one after the other. Since, we don't need the values of the intermediate tensors completely, if we can compute T1 and T2 partially, we could use that to compute appropriate values of A. This requires merging the 3 loop nests. But, since the actual tensor contractions happen over different dimensions, this requires careful re-writing of tensor accesses.

Removing the intermediate tensors would help save both time because of merging of loops and memory because the 2 rank-8 intermediate tensors never have to be completely kept in memory. This would provide significant savings since if the value of D is 16, then the number of elements in a rank-8 tensor whose size of dimension is D is 168. Since the values of the arrays are doubles, each values requires 8-bytes of stored. Thus the total memory required to store a single rank-8 tensor of size D=16 is 168*8 bytes, which is equal to 32GB. Thus avoiding the storage of 2 such tensors saves 64GB memory. This is just an example, the actual memory required increases exponentially with increasing D values.

3.7 Conclusion

The optimizations that were done on HOTRG algorithm, mentioned in this chapter were just initial optimizations. There are other major performance bottlenecks to be handled. For example, when the D value for the threshold is increased, the memory requirements increase exponentially, and this would create major bottlenecks.

One way of saving memory would be to approximate the tensors involved in tensor contraction of the HOSVD performed before tensor contraction. Another option would be to implement openMPI code, and distribute the computation over a large distributed network, thus getting access to large amounts of memory that are typically not available in a single computer. This would not directly solve the problem as openMPI based computation would pose its own set of challenges regarding communication minimization and efficient resource usage. Thus, there are a lot of other optimization avenues that can be explored

Chapter 4

Legup: High Level Synthesis with LLVM

4.1 Introduction

Field Programmable Gate Arrays are devices that contain configurable blocks of arrays. They contain blocks of memory and interconnects. They can be programmed to work for any required purpose. The programming of FPGAs is normally done using low level Hardware Definition Languages (HDLs) or Register Transfer Languages (RTLs) such as Verilog, VHDL and systemC. The HDL code is then synthesized and programmed in to the desired hardware.

Programming in HDL/RTL type of languages was easy initially, but as devices have become more and more complicated, and the requirements of algorithms for encryption, deep learning, etc. on hardware becomes more and more important, the need for programming the algorithms in high level languages and porting them to HDL type languages is becoming more important. This gives rise to tools which perform High Level Synthesis.

4.2 High Level Synthesis

High Level Synthesis (HLS) also known as C synthesis, electronic system-level (ESL) synthesis, algorithmic synthesis, or behavioral synthesis, is the process of writing the hardware circuit definitions in high level languages such as C/C++ and generating HDL/RTL code is called High level Synthesis. Due to the ease of programming with high level languages, complex algorithms now can be written with much more ease, giving rise to design of hardware for some very complex algorithms like deep learning algorithms. The overall structure of High Level Synthesis constitutes three major steps.

1. Scheduling
2. Allocation
3. Binding

4.2.1 Scheduling

Scheduling is the process of dividing the control flow of the algorithm into parts that can be used to define the states of a finite state machine. The finite state machine is used to define the the steps of the computation, which is then synthesized. Each part/step is used to perform one small step of the algorithm, which can be performed in a single clock cycle. This step basically defines steps analogous to timing control of the circuit.

4.2.2 Allocation

Allocation is the process of assigning components of hardware to the parts of the program. This includes allocating memory components such as memory blocks or registers, interconnects, arithmetic units and other hardware specific components.

4.2.3 Binding

Binding as the name specifies, binds the allocated components of hardware to the control steps scheduled during scheduling. This provides final structure for the RTL code to be generated

For more details see [36] and [37]

4.3 Related Tools

There are a variety of high level synthesis tools that are available in the community. There are proprietary versions available from FPGA vendors such as VivadoHLS for Xilinx FPGAs, IntelHLS for Altera based FPGAs, etc. There are also some tools that are available for simulating and synthesizing RTL/HDL code such as modelsim from mentor graphics, etc. Most of these tools only work with their specific FPGA devices.

One of the most famous open source versions of HLS tools is Legup. Legup is an LLVM based tool to generate verilog code for FPGAs. They support a wide range of FPGA devices. The code base however is not maintained and the version of LLVM used was 3.5, compared to the current version of LLVM, which is 7.0

4.4 Basic architecture of the tool

Legup contains a verilog backend for LLVM which is used to generate verilog tool. The backend code generator generates verilog modules for the functions/methods available in high level language. The data is stored mainly in two different memories, one is Local memory and the other is Global memory which are stored on chip. These two together contain the all the major data in the program.

If the program is not completely generated on hardware, but rather as a hybrid HW-SW version, there are also cache and off-chip memories which are not stored on the FPGA

Any variables which are specific to a single function are allocated in local memories, while variables/arrays, which are allocated and used across multiple functions are stored in global memory. The local memory can be accessed in parallel, and they do not need expensive multiplexing, which saves area.

Global memory is created inside a memory controller. The memory controller helps drive the memory access to the right block RAM during runtime. An illustration of the memory controller inside Legup is shown below, which is taken from Legup official documentation.

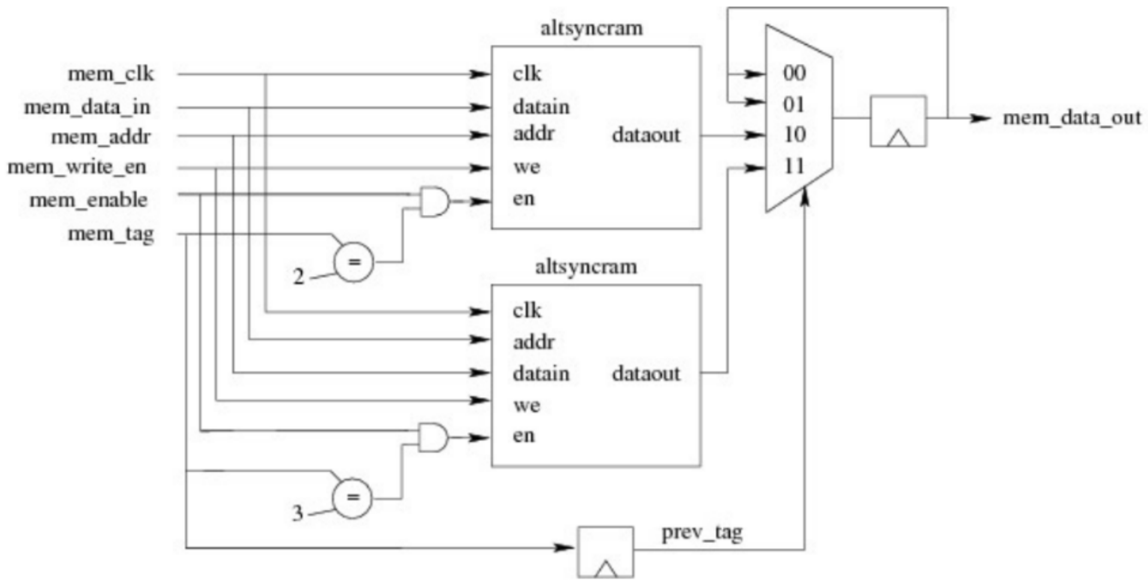


Figure 4.1: Memory controller used in Legup

Legup supports the conversion of structs, multi-dimensional arrays, function calls, dynamic memory allocations, etc. into FPGA code. It however does not support some constructs such as function pointers. For more details please see [38]

4.5 My work

The major issue with legup was that it was not maintained and the last released version was using LLVM 3.5. LLVM 3.5 is seven versions ahead of the latest version of LLVM which is version 7.0. Let's first see some major details of the differences between LLVM 3.5 and LLVM 7.0

The first major difference was that LLVM 3.5 used autoconf based configure scripts build the source. Compared to that LLVM 7.0 has completely ported to CMAKE style build scripts, which are much more versatile. So, porting a backend written for LLVM 3.5 to 7.0 required modifying the appropriate build scripts to be similar to the changes done in the original version.

The next big difference was the LLVM 3.5 was using a pass manager, which has become the LegacyPassManager in the current version. This change includes adding wrapper classes for most of the LLVM passes and other changes. These changes had to be fixed for the newer version.

There were also other smaller breaking changes among the versions, which required fixing. Apart from these code changes, Legup's verilog backend depends on configuration files read in the form TCL scripts, which contain the hardware configurations for specific FPGA hardwares. These scripts were still needed with the latest version, and this required working on the opt and llc tools of LLVM, and allowing it to take command line option to read configuration scripts.

After the changes, we built and tested the verilog backend with test examples given as part of the original Legup source. We verified the correctness of the backend.

References

[1] J. Nievergelt. Parallel Methods for Integrating Ordinary Differential Equations. *Commun. ACM* 7, (1964) 731–733.

[2] W. L. Miranker and W. Liniger. Parallel methods for the numerical integration of ordinary differential equations. *Mathematics of Computation* 21, (1967) 303–303.

[3] V. Martin. An optimized Schwarz waveform relaxation method for the unsteady convection diffusion equation in two dimensions. *Applied Numerical Mathematics* 52, (2005) 401–428.

[4] H. A. L. and H. A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology* 117, (1952) 500–544.

[5] M. L. Hines, H. Eichner, and F. Schürmann. Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. *Journal of Computational Neuroscience* 25, (2008) 203–210.

[6] M. L. Hines, H. Markram, and F. Schürmann. Fully implicit parallel simulation of single neurons. *Journal of Computational Neuroscience* 25, (2008) 439–448.

[7] M. Mascagni. A parallelizing algorithm for computing solutions to arbitrarily branched cable neuron models. *Journal of Neuroscience Methods* 36, (1991) 105 – 114.

[8] D. T. Vooturi, K. Kothapalli, and U. S. Bhalla. Parallelizing Hines Matrix Solver in Neuron Simulations on GPU. In 2017 IEEE 24th International Conference on High Performance Computing (HiPC). IEEE, 2017 .

[9] G. H. Golub and C. F. Van Loan. Matrix Computations (3rd Ed.). Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[10] L. W. Ehrlich. A Numerical Method of Solving a Heat Flow Problem with Moving Boundary. *J. ACM* 5, (1958) 161–176.

[11] H. S. Stone. An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. *J. ACM* 20, (1973) 27–38.

[12] H. S. Stone. Parallel Tridiagonal Equation Solvers. *ACM Trans. Math. Softw.* 1, (1975) 289–307.

[13] E. Polizzi and A. H. Sameh. A Parallel Hybrid Banded System Solver: The SPIKE Algorithm. *Parallel Comput.* 32, (2006) 177–194.

[14] E. Polizzi and A. Sameh. SPIKE: A parallel environment for solving banded linear systems. *Computers & Fluids* 36, (2007) 113–120.

[15] E. Gallopoulos, B. Philippe, and A. H. Sameh. Parallelism in Matrix Computations. 1st edition. Springer Publishing Company, Incorporated, 2015.

[16] M. Hines. Efficient computation of branched nerve equations. *International Journal of Bio-Medical Computing* 15, (1984) 69 – 76.

[17] P. Dayan and L. F. Abbott. Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems. The MIT Press, 2005.

[18] J.-L. Larriba-Pey, M. Mascagni, A. Jorba, and J. J. Navarro. An Analysis of the Parallel Computation of Arbitrarily Branched Cable Neuron Models. In PPSC. 1995 .

[19] r. ben shalom, g. liberman, and a. korngreen. accelerating compartmental modeling on a graphical processing unit. *frontiers in neuroinformatics* 7, (2013) 4.

[20] W. Hackbusch. Parabolic Multi-grid Methods. In Proc. Of the Sixth Int'L. Symposium on Computing Methods in Applied Sciences and Engineering, VI. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1985 189–197.

[21] A. J. Christlieb, C. B. Macdonald, and B. W. Ong. Parallel High-Order Integrators. *SIAM J. Sci. Comput.* 32, (2010) 818–835.

[22] S. Güttel. A Parallel Overlapping Time-Domain Decomposition Method for ODEs. In R. Bank, M. Holst, O. Widlund, and J. Xu, eds., Domain Decomposition Methods in Science and Engineering XX. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013 459–466.

[23] J.-L. Lions, Y. Maday, and G. Turinici. A parareal in time discretization of PDEs. *C.R. Acad. Sci. Paris, Series I* 332, (2001) 661–668.

[24] M. J. Gander and E. Hairer. Nonlinear Convergence Analysis for the Parareal Algorithm. In Lecture Notes in Computational Science and Engineering, 45–56. Springer Berlin Heidelberg, 2008.

[25] J. Cortial and C. Farhat. A time-parallel implicit method for accelerating the solution of non-linear structural dynamics problems. *International Journal for Numerical Methods in Engineering* 77, (2009) 451–470.

[26] M. Emmett and M. Minion. Toward an efficient parallel in time method for partial differential equations. *Communications in Applied Mathematics and Computational Science* 7, (2012) 105–132.

[27] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. Minion, M. Winkel, and P. Gibbon. A Massively Space-time Parallel N-body Solver. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12. IEEE Computer Society Press, Los Alamitos, CA, USA, 2012 92:1–92:11.

[28] L. Lapicque. Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. *J Physiol Pathol Gen* 9, (1907) 620–635.

[29] C. J. Lobb, Z. Chao, R. M. Fujimoto, and S. M. Potter. Parallel event-driven neural network simulations using the Hodgkin-Huxley neuron model. In Workshop on Principles of Advanced and Distributed Simulation (PADS'05). 2005 16–25.

[30] S. Robert, R. Daniel, E. Matthew, B. Matthias, and K. Rolf. A space-time parallel solver for the three-dimensional heat equation. *Advances in Parallel Computing* 25, (2014) 263–272.

[31] M. Levin and C. P. Nave. Tensor Renormalization Group Approach to Two-Dimensional Classical Lattice Models. *Physical Review Letters* 99.

[32] S. R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters* 69, (1992) 2863–2866.

[33] U. Schollwöck. The density-matrix renormalization group. *Reviews of Modern Physics* 77, (2005) 259–315.

[34] Z. Y. Xie, H. C. Jiang, Q. N. Chen, Z. Y. Weng, and T. Xiang. Second Renormalization of Tensor-Network States. *Physical Review Letters* 103.

[35] Z. Y. Xie, J. Chen, M. P. Qin, J. W. Zhu, L. P. Yang, and T. Xiang. Coarse-graining renormalization by higher-order singular value decomposition. *Physical Review B* 86.

[36] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers* 26, (2009) 18–25.

[37] P. Coussy and A. Morawiec. High-Level Synthesis: From Algorithm to Digital Circuit. 1st edition. Springer Publishing Company, Incorporated, 2008.

[38] LegUp 4.0 Documentation¶.