Parallelization of an Odd-Even Solver Using CUDA

Clarence Lin[1]

Abstract— **Tridiagonal matrix systems are essential in solving constrained optimization problems, which appear in fields like robotics, control systems, and machine learning. However, traditional solvers like the Thomas algorithm and Python's NumPy library are inherently sequential or CPU-based, making them unable to leverage parallelism and less efficient for large-scale problems. We developed a CUDA-based implementation of an odd-even solver, which comes from the cyclic reduction algorithm. Being optimized for GPU parallelism, our implementation ensures scalability and numeric stability for large matrix sizes compared to other algorithms we developed, which involved NumPy methods, the Thomas Algorithm, and sequential odd-even solvers. Our optimized solver achieves a ∼50x speedup compared to NumPy's $np.linalg.solve$ method and a ∼7x compared to CPU-based odd-even solvers in C.**

I. Introduction

If a matrix A is block tridiagonal within the equation $Ax = b$, where b is any possible solution vector, the simplistic and symmetrical properties that arise from the matrix allow for efficient solutions. Solving such systems quickly is critical for real-time applications and large-scale systems such as trajectory optimization, model predictive control, and real-time path planning [1]. However, approaches to these issues typically involve mathematical endeavors or using general-purpose methods that utilize CPUs and sequential algorithms [2], [3]. Few talk about the effectiveness of parallelizing odd-even solvers.

To address these limitations, various parallel algorithms have been developed. For instance, the SPIKE algorithm offers a hybrid parallel approach for banded linear systems, including tridiagonal matrices, by dividing the problem into smaller subproblems that can be solved concurrently [4]. Despite these advancements, challenges remain in achieving both numerical stability and high performance. Another paper introduced a numerically stable tridiagonal solver using diagonal pivoting on GPUs, addressing some of these concerns [5]. However, existing methods often face challenges in achieving a balance between numerical stability, scalability, and computational efficiency [5].

In this work, we design and compare four different methods of solving tridiagonal matrix systems: a NumPy, a tridiagonal matrix algorithm (also known as Thomas algorithm), a sequential odd-even solver, and a parallel odd-even solver implementation. The primary focus is on the parallel odd-even solver built using CUDA.

Our main contribution is a successful design of an odd-even solver using the cyclic reduction algorithm while managing memory for matrix equations efficiently. We ensure this parallel implementation enhances scalability and runtime efficiency for large tridiagonal matrix systems. In doing so, we demonstrate the effectiveness of stride-based parallel reduction processes that iteratively reduce matrices into smaller, independent systems. Overall, by utilizing threading and GPU-specific synchronization, we show a 50x speedup over simple Python sequential solvers and a 7x speedup over CPU-based odd-even solvers written in C for 10,000-dimensional matrices.

We compare these different algorithms across matrices of increasing dimensionality, thereby demonstrating the CUDA-based odd-even solver achieves better runtime efficiency and scalability compared to the other methods as well as equivalent numerical stability. Our source code can be found at `github.com/1Clarence3/Parallel-Odd-Even-Solver`.

[1]Clarence Lin is with Columbia University. `cl4317@columbia.edu`
[2]Lixia Chen Wu is with Barnard College. `lc3716@barnard.edu`.

II. Related Work

Traditional methods, such as Gaussian elimination and LU decomposition, have been widely used but often lack the parallelism required for modern hardware architectures [6]. To address this limitation, researchers have developed parallel algorithms that can leverage multi-core CPUs and GPUs.

One notable approach is the cyclic reduction algorithm, also known as an odd-even reduction algorithm due to the algorithm's repeated elimination of variables at "odd" and "even" positions in the system [7], [8]. Although this method has shown promising results in terms of performance and scalability, its parallel implementation for block-tridiagonal systems remains an area of active research [9].

To elaborate, many studies have focused on enhancing mathematical and theoretical aspects of odd-even and cyclic reduction solvers. For instance, one piece of work established the equivalence between cyclic reduction and Gaussian elimination without pivoting on a permuted system, providing insights into the method's stability for certain matrix classes [10]. In another case, research on incomplete cyclic reduction demonstrated the potential for mathematically approximating solutions with early termination [10].

Implementation of these methods on parallel architectures presents several challenges. Some odd-even solvers have reported to suffer from numerical instability, which can result in unexpected oscillations or rapid error growth, even with small time steps [11], [12], [13]. The effectiveness of these solvers is also highly dependent on the condition number of the matrix. Ill-conditioned matrices, which are not uncommon in trajectory optimization problems, can lead to significant rounding errors [14].

Furthermore, existing work on this topic often strays away from trajectory optimization and instead focuses on tridiagonal matrices in the context of solving Poisson equations, conjugate gradient-like algorithms, and least-squares problems [15], [16].

These factors contribute to the relative scarcity of literature on practical, parallel implementations of odd-even solvers, which provides the main motivating factor for our research.

III. Background

A. Cyclic Reduction Algorithm

To understand how the odd-even solver works, we first need to see how to set up a tridiagonal matrix. It is a square matrix in which all nonzero elements are confined to the main diagonal, the diagonal above it (superdiagonal), and the diagonal below it (subdiagonal). We can visualize this by considering the equation $Ax = b$:

$$\begin{bmatrix} b_1 & c_1 & & \cdots & & \\ a_2 & b_2 & c_2 & \cdots & & \\ \vdots & . & . & . & & \\ & & . & . & . & \\ & & & . & . & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} \tag{1}$$

We can then derive the following equation, where $a_1 = 0$ and $c_n = 0$.

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \quad i = 1, \ldots, n, \tag{2}$$

The variable a represents entries along the subdiagonal, b represents those on the main diagonal, and c represents those in the superdiagonal. The key idea is that if we only consider the odd rows in the matrix, we can express all odd unknowns as linear combinations of even unknowns. This eliminates them and reduces the system by half. We can repeat this process called forward reduction until a single equation remains, which can then be solved directly. Once the smallest system is solved, the back substitution process is used to recover the values of the eliminated variables. The following equations, originally presented by Cox and Knisely express entries along the three diagonals using their neighbors [17]. Note that the superscript j represents the coefficients after the jth step while the subscript i represents the element in the ith row:

Algorithm 1 Forward Reduction

$$a_i^{(j+1)} = \begin{cases} -\frac{a_{2i}^{(j)} a_{2i-1}^{(j)}}{b_{2i-1}^{(j)}}, & \text{if } i \neq 1, \\ 0, & \text{if } i = 1 \end{cases}$$

$$b_i^{(j+1)} = b_{2i}^{(j)} - \frac{a_{2i}^{(j)} c_{2i-1}^{(j)}}{b_{2i-1}^{(j)}} - \frac{c_{2i}^{(j)} a_{2i+1}^{(j)}}{b_{2i+1}^{(j)}}$$

Algorithm 1 Forward Reduction (continued)

$$c_i^{(j+1)} = \begin{cases} -\frac{c_{2i}^{(j)} c_{2i+1}^{(j)}}{b_{2i+1}^{(j)}}, & \text{if } i \neq \frac{n+1}{2^j}, \\ 0, & \text{if } i = \frac{n+1}{2^j} \end{cases}$$

$$d_i^{(j+1)} = d_{2i}^{(j)} - \frac{a_{2i}^{(j)} d_{2i-1}^{(j)}}{b_{2i-1}^{(j)}} - \frac{c_{2i}^{(j)} d_{2i+1}^{(j)}}{b_{2i+1}^{(j)}}$$

Moving onto the back substitution algorithm, we can now individually solve each entry in our solution vector x using our variables a, b, c, d from before [17].

Algorithm 2 Back substitution

$$x_{2i}^{(j)} = x_i^{(j+1)}, \quad i = 1, \ldots, \frac{n+1}{2^j} - 1$$

$$x_{2i-1}^{(j)} = \frac{d_{2i-1}^{(j)} - a_{2i-1}^{(j)} x_{2i-2}^{(j)} - c_{2i-1}^{(j)} x_{2i}^{(j)}}{b_{2i-1}^{(j)}},$$

$$i = 1, \ldots, \frac{n+1}{2^j}, \quad j = k-1, \ldots, 1$$

Though these equations may seem complex, they lay out a solid mathematical foundation to follow for possible code implementation. The recursive nature of this algorithm may raise the question of how this could be solved in parallel. However, this is resolved when considering that we want to parallelize all the i rows at any given step j, instead of trying to parallelize several steps at once, which would be impossible. The exact inner workings of such an algorithm will be discussed in detail later in the design implementation section.

B. Sequential Algorithms

In order to benchmark our CUDA implementation of this algorithm with several baselines, we looked at a few common sequential approach for solving tridiagonal systems. One can utilize the Python library NumPy's $np.linalg.solve()$. This general-purpose linear algebra solver uses LU decomposition. While versatile, it does not exploit the tridiagonal structure, resulting in an $O(n^3)$ time complexity and the inability for further parallelization to be explored [18].

Another algorithm called the Thomas algorithm is a specialized form of Gaussian elimination for tridiagonal systems. It achieves $O(n)$ time complexity by leveraging similar steps of forward reduction and back substitution [2]. However, instead of eliminating alternate rows at each step like in an odd-even solver, it eliminates lower diagonal elements, transforming the matrix into an upper triangular form. Hence, the Thomas Algorithm cannot be parallelized effectively because each step depends on the previous one but makes for an adequate baseline [19].

Overall, these sequential methods, while efficient for small to medium-sized systems, become less practical for large-scale problems and large matrices, as we will see in the results section.

IV. C and CUDA Implementation

By using just the standard `stdio.h`, `stdlib.h`, and `time.h` libraries in C, we successfully implemented a sequential odd-even solver. A major reason for coding it in C first was to allow for an intermediate performance benchmark between simpler sequential algorithms in Python from before and the more complex parallel CUDA implementation later. Additionally, given their similar low-level control over memory and computations, extending from C to CUDA with GPU-specific constructs like kernels and memory management would be much smoother.

As mentioned previously, the cyclic reduction algorithm systematically reduces the system size through a divide-and-conquer strategy. A key step within our code began by determining the maximum number of reduction iterations required based on the size of the tridiagonal matrix. This is computed using a helper function that calculates the number of binary shifts needed to reduce the matrix to a single equation. Later on, the value is used in a forward reduction and back substitution kernel to limit the amount of iterations done. Below is the pseudocode for it:

```
int m = 0;
int cnt = 0;
while (binary != 0) {
    m += binary % 2;
     binary = binary >> 1;
     cnt++;
}
cnt--;
if (m > 1) {
    m = 0;
}
return (cnt - m);
```

Fig. 1. Code for counting the max iterations using bit shifts

Subsequently, in the main solver method, we initialized and allocated memory to the lists $alist$, $blist$, $clist$, $dlist$, and $xlist$ which represent the variables in our forward reduction and back substitution (Algorithm 1 and 2).

A final method $gen\_tridiagonal$ along with the variable $diagSz$ specifying the matrix dimensions gave the user the freedom to input custom matrices to test our odd-even solver. With this overview of our code, we can now investigate how to take advantage of parallel computations within the algorithm.

To understand how we parallelized the cyclic reduction algorithm, we can realize the potential of solving multiple equations at the same time at each algorithmic step. In the graphic below, where e stands for "equation", we can see how parallel cyclic reduction (PCR) recursively halves the number of equations simultaneously [20].

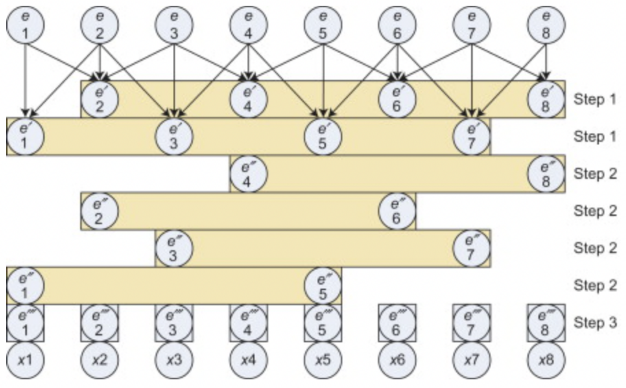


Fig. 2. Design flow of reducing equations in PCR.

In the above example, an eight-equation system is reduced to systems with four-equation systems in step 1, followed by systems with two-equation systems. At each step the number of such systems doubles. Each yellow bar indicates the ability for a thread to simplify variables in an entire row of a matrix at a given step (Figure 2). Once we reach the final step of having just single variables, we begin the back substitution phase in a similar vein. That is, after solving the smallest subsystems, the solutions are propagated back to reconstruct larger subsystems. At each level of the hierarchy, the computation of each subsystem's solution is independent of each other, allowing the work to be parallelized. This lends itself to PCR only requiring $O(log(n))$ steps [21].

Now considering the code, we divided the task into a forward reduction kernel and a back substitution kernel that are as follows:

```
int threads_per_block = 256;
int blocks_per_grid = (diagSz+
    threads_per_block -1)/threads_per_block
    ;
for (int stride=1; stride<diagSz; stride
    <<=1) {
     forward_red<<<blocks_per_grid,
          threads_per_block>>>(a1, b1, c1,
          d1, stride, diagSz);
     cudaDeviceSynchronize();
}
```

Fig. 3. CUDA Forward Reduction Code

```
int threads_per_block = 256;
int blocks_per_grid = (diagSz+
    threads_per_block -1)/threads_per_block
    ;
for (int stride=diagSz/2; stride>0; stride
    >>=1) {
     back_sub<<<blocks_per_grid,
          threads_per_block>>>(a1, b1, c1,
          d1, x1, stride, diagSz);
     cudaDeviceSynchronize();
}
```

Fig. 4. CUDA Back Substitution Code

The code used a thread block size of 256 threads, which we chose due to it being a multiple of 32 (warp size), which aligns their memory accesses and maximizes efficiency. Additionally, online literature has indicated this provides a good balance between shared memory usage and the number of active threads. The number of blocks is calculated essentially by taking a ceiling function of $diagSz/threads_per_block$, which ensures that enough blocks are allocated to cover all elements in the matrix.

These variables became useful within both kernel methods because we could index the threads by using a common formula of $idx = blockIdx.x * blockDim.x + threadIdx.x$. In terms of what the threads represent in the forward reduction pseudo code, each thread is assigned to an equation in the system while the stride parameter determines which equations interact in each iteration. As the threads simultaneously update a, b, c, and d values for their assigned equations, $syncthreads()$ ensures all threads complete their updates before proceeding. On the other hand, in back substitution, we start from the largest stride corresponding to the equations that were eliminated last during the forward reduction. Similar to a dynamic programming approach, this furthers maximum parallelism because as we progress to smaller strides, more unknowns can be computed simultaneously. From there, each thread updates its part of the solution vector independently.

Shifting the focus to the memory aspect of this implementation, memory for each of the arrays was allocated on the host using $cudaMalloc()$. They were then copied to their corresponding device arrays using $cudaMemcpy()$ prior to the execution of the kernels. After the computation was complete, the results of our x vector were stored back to the host for further visualization.

V. Results

The performance comparison of four solvers — NumPy's np.linalg.solve, the Thomas algorithm, the odd-even method, and the CUDA-based odd-even solver — were evaluated across matrices of dimensions 100, 1,000, and 10,000. This analysis examines both runtime performance and numerical stability to assess the trade-offs and advantages of the methods in handling tridiagonal matrix systems.

A. Methodology

Our CUDA-based odd-even solvers were executed in a GPU-accelerated environment using Google Colab. This gave us access to the NVIDIA Tesla T4 GPU. The software was compiled and executed using CUDA Toolkit's nvcc compiler driver. The CPU-based methods using NumPy and the Thomas Algorithm were executed directly in Python while our sequential odd-even solver was written in C.

To test the runtime, tridiagonal matrices of dimensions 100, 1,000, and 10,000 were generated dynamically using a custom Python function. The main diagonal and off-diagonal entries were defined to ensure consistency across solvers, with the right-hand side b vector generated using a simple increasing numerical sequence from 1 to the current dimension.

For CPU-based solvers, Python's $time$ module was used to measure the runtime by recording the start and end times of each method. For the CUDA-based solver, the $cudaEventRecord$ function was used to accurately measure the execution time on the GPU. Each test was then repeated five times to minimize variability, and the average runtime was reported.

For efficiency purposes, the first ten entries of the solution vectors from each solver were compared across each matrix size. Numerical deviations between methods were analyzed to assess stability and identify any potential rounding errors. As a ground truth benchmark, we validated our solutions against the NumPy method, which we treated as the baseline for accuracy.

The results were visualized using Matplotlib's bar graphs to compare runtimes as well as with line plots with logarithmic scales to highlight scalability trends.

B. Analysis

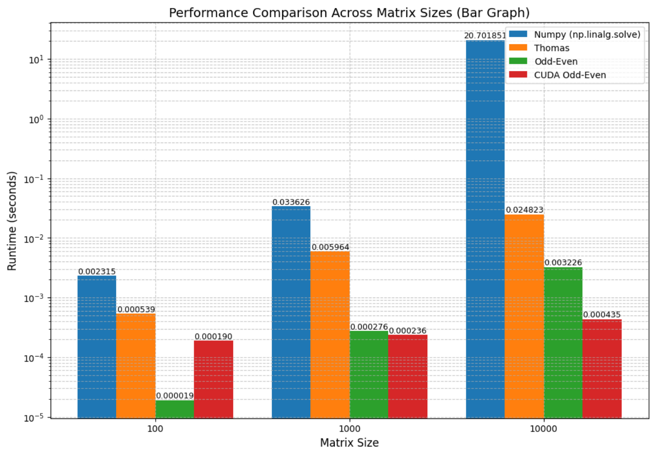


Fig. 5. Runtimes across 4 algorithms on 3 different matrix sizes.

By looking at the bar graph, it is clear that across all matrix sizes, dimensionality is positively correlated with runtime. Another trend is that the CUDA-based odd-even solver has a significant advantage over the other methods. At a matrix dimension of 100, its time of 0.00019 seconds over the baseline NumPy's 0.00231 second runtime is equivalent to a 12-fold increase (Figure 5). As the matrix size increases to 1,000 and 10,000 the difference between the runtime of this parallel implementation against all other methods scales almost exponentially. This is exemplified in the matrix of dimensionality 10,000, where the CUDA odd-even solver computes at 0.000435 seconds, far outpacing the CPU odd-even method at 0.00323 seconds and NumPy at 20.702 seconds. Overall, it seems that the algorithmic performance ranked in ascending order is as follows: NumPy solver, Thomas algorithm, CPU-based odd-even solver, and the CUDA-based Odd-Even solver. This aligns with our background research that the memory allocation and fast low-level computations of C-based implementations' should be advantageous compared to Python sequential implementations while parallelized odd-even solvers would perform the best.

An important note is that for small dimensions like 100, the CUDA odd-even solver at 0.000190 seconds is slower than the CPU-based odd-even algorithm's runtime of 0.000019 seconds. This is likely due to the fixed overheads of CUDA parallelization, including kernel launch setup, resource allocation, and CPU-GPU memory transfers, which dominate for smaller problem sizes with lighter computational workloads [22]. However, as matrix size increases, the workload grows, and these overheads become negligible. Consequently, the CUDA odd-even solver can fully capitalize on the GPU's parallel processing capabilities, achieving superior performance and scalability for larger matrices.

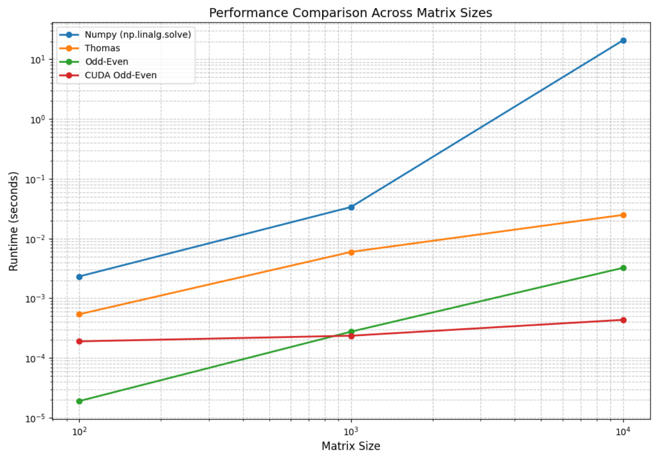
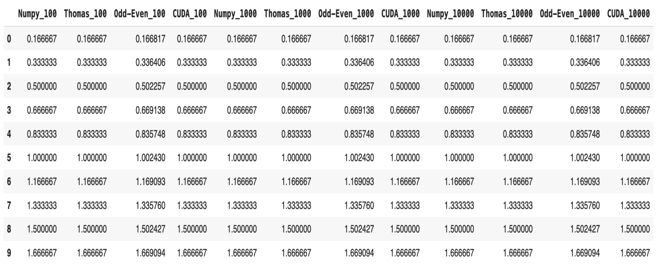


Fig. 6. Runtimes across 4 algorithms on 3 different matrix sizes to visualize scalability efficiency with runtime performance of each solver.

Transitioning to scalability instead of focusing on raw runtime numbers, the NumPy's algorithm shows poor scalability with a steep increase in runtime as matrix size grows, making it unsuitable for large-scale or real-time applications (Figure 6). The Thomas algorithm, though more efficient than NumPy, follows a linear growth pattern and remains limited by its sequential nature, which corroborates our time complexity findings of $O(n)$ from earlier [2]. The CPU-based odd-even method shows a better runtime over the Thomas algorithm at the start, though their performances likely converge as matrix size increases. Lastly, the CUDA-based odd-even solver scales the slowest, making it highly advantageous for large matrices.

From a numerical stability standpoint, we can reference the figure here:



	Numpy_100	Thomas_100	Odd-Even_100	CUDA_100	Numpy_1000	Thomas_1000	Odd-Even_1000	CUDA_1000	Numpy_10000	Thomas_10000	Odd-Even_10000	CUDA_10000
0	0.166667	0.166667	0.166817	0.166667	0.166667	0.166667	0.166817	0.166667	0.166667	0.166667	0.166817	0.166667
1	0.333333	0.333333	0.336406	0.333333	0.333333	0.333333	0.336406	0.333333	0.333333	0.333333	0.336406	0.333333
2	0.500000	0.500000	0.502257	0.500000	0.500000	0.500000	0.502257	0.500000	0.500000	0.500000	0.502257	0.500000
3	0.666667	0.666667	0.669138	0.666667	0.666667	0.666667	0.669138	0.666667	0.666667	0.666667	0.669138	0.666667
4	0.833333	0.833333	0.835748	0.833333	0.833333	0.833333	0.835748	0.833333	0.833333	0.833333	0.835748	0.833333
5	1.000000	1.000000	1.002430	1.000000	1.000000	1.000000	1.002430	1.000000	1.000000	1.000000	1.002430	1.000000
6	1.166667	1.166667	1.169093	1.166667	1.166667	1.166667	1.169093	1.166667	1.166667	1.166667	1.169093	1.166667
7	1.333333	1.333333	1.335760	1.333333	1.333333	1.333333	1.335760	1.333333	1.333333	1.333333	1.335760	1.333333
8	1.500000	1.500000	1.502427	1.500000	1.500000	1.500000	1.502427	1.500000	1.500000	1.500000	1.502427	1.500000
9	1.666667	1.666667	1.669094	1.666667	1.666667	1.666667	1.669094	1.666667	1.666667	1.666667	1.669094	1.666667

Fig. 7. The first 10 entries of the solution vectors for each algorithm rounded to 6 decimal points across 3 matrix sizes.

Despite the runtime differences, all solvers deliver consistent and stable solutions with minor numerical deviations, such as the 5th entry being 1.00243 for the CPU-based odd-even method compared to 1.0 for the other solvers (Figure 7). Overall, looking across all entries, the table reflects our goal of obtaining a parallel structure for odd-even methods while maintaining solution stability. It seems that even for large matrices the CUDA-based odd-even solver is still able to maintain its numerical stability.

VI. Conclusion and Future Work

In this work, we developed a CUDA-based parallel implementation of the odd-even solver, inspired by the cyclic reduction algorithm, to address the computational challenges associated with solving tridiagonal matrix systems. Our GPU optimized solver demonstrated significant scalability and efficiency, achieving an approximate 50x speedup over NumPy's np.linalg.solve method and an approximate 7x improvement compared to CPU-based odd-even solvers. Our results also indicated that the difference in runtime growth between our parallel solver and CPU-based solvers in C or sequential Python solvers was notable. The results of scalability and numerical consistency also highlight the potential of leveraging GPU parallelism for this approach in fields like robotics, control systems, and machine learning.

Areas of future work include finding a threshold point (matrix dimension) at which the observed memory overhead on our CUDA-based code impacts efficiency for smaller matrix sizes. Addressing this issue by optimizing memory utilization or exploring hybrid CPU-GPU approaches could further enhance performance [23]. Additionally, a big challenge in code development was the limitation of GPU credits. While we evaluated scalability up to moderately large matrix sizes, testing on significantly larger matrices of 10^6 or 10^8, comparing values across all entries, and using different matrix values would provide deeper insights into the exact growth rate in runtime compared to theoretical time complexities.

Other directions involve applying this solver to real-time trajectory optimization problems, such as cost optimization or model predictive control in robotics [24]. This would validate its effectiveness in real-world scenarios but also highlight potential areas for combining multiple algorithms [25]. With these extensions, our work could pave the way for more computationally demanding domains.

VII. Contributions

The authors Clarence Lin and Lixia Chen Wu were involved in the complete code development and paper writing for this project. Clarence worked on researching existing work done on sequential odd-even algorithms and converting the CPU-based code written in C to a parallel implementation in CUDA using threading and memory allocation. In writing this paper, he mainly wrote the related works, background, design/implementation, and conclusion sections. Lixia helped code the CPU-based algorithms, which involved Python's NumPy library, the Thomas algorithm, and the CPU-based sequential cyclic reduction algorithm. She largely wrote the abstract, introduction, and results sections. In terms of setting up the coding environment, debugging Python and C code, and creating a presentation, there was a significant overlap in contribution.

References

[1] Y. Wang and S. Boyd, "Fast Model Predictive Control Using Online Optimization," *IEEE Transactions on Control Systems Technology*, vol. 18, no. 2, pp. 267–278, Mar. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5153127/

[2] "Thomas Algorithm - an overview | ScienceDirect Topics." [Online]. Available: https://www.sciencedirect.com/topics/computer-science/thomas-algorithm

[3] "(PDF) The cyclic reduction algorithm," *ResearchGate*, Oct. 2024. [Online]. Available: https://www.researchgate.net/publication/225220867_The_cyclic_reduction_algorithm

[4] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: The spike algorithm," *Parallel Computing*, vol. 32, no. 2, pp. 177–194, 2006.

[5] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W. mei W. Hwu, "A scalable, numerically stable, high-performance tridiagonal solver using gpus," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–11.

[6] M. Stabrowski, "Parallel real-world LU decomposition: Gauss vs. Crout algorithm," *Open Computer Science*, vol. 8, no. 1, pp. 210–217, Dec. 2018, publisher: De Gruyter Open Access. [Online]. Available: https://www.degruyter.com/document/doi/10.1515/comp-2018-0020/html

[7] "Cyclic Reduction - an overview | ScienceDirect Topics." [Online]. Available: https://www.sciencedirect.com/topics/computer-science/cyclic-reduction

[8] M. T. Heath and E. Solomonik, "Parallel Numerical Algorithms - Chapter 4 – Sparse Linear Systems Section 4.2 – Banded Matrices."

[9] D. Heller, "Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems," *SIAM Journal on Numerical Analysis*, vol. 13, no. 4, pp. 484–496, 1976, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: https://www.jstor.org/stable/2156240

[10] W. Gander and G. H. Golub, "Cyclic Reduction – History and Applications."

[11] P. Yalamov and V. Pavlov, "Stability of the block cyclic reduction," *Linear Algebra and its Applications*, vol. 249, no. 1, pp. 341–358, Dec. 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0024379595003924

[12] C. C. K. Mikkelsen and B. Kågström, "Approximate Incomplete Cyclic Reduction for Systems Which Are Tridiagonal and Strictly Diagonally Dominant by Rows," in *Applied Parallel and Scientific Computing*, P. Manninen and P. Öster, Eds. Berlin, Heidelberg: Springer, 2013, pp. 250–264.

[13] M. Neuenhofen, "Review of Cyclic Reduction for Parallel Solution of Hermitian Positive Definite Block-Tridiagonal Linear Systems," Jul. 2018, arXiv:1807.00370 [math]. [Online]. Available: http://arxiv.org/abs/1807.00370

[14] P. A. Rubin, "Ill-conditioned Bases and Numerical Instability," Aug. 2010. [Online]. Available: https://orinanobworld.blogspot.com/2010/08/ill-conditioned-bases-and-numerical.html

[15] R. W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," *J. ACM*, vol. 12, no. 1, pp. 95–113, Jan. 1965. [Online]. Available: https://dl.acm.org/doi/10.1145/321250.321259

[16] R. Bramley and A. Sameh, "A robust parallel solver for block tridiagonal systems," in *Proceedings of the 2nd international conference on Supercomputing - ICS '88*. St. Malo, France: ACM Press, 1988, pp. 39–54. [Online]. Available: http://portal.acm.org/citation.cfm?doid=55364.55369

[17] C. L. Cox and J. A. Knisely, "A tridiagonal system solver for distributed memory parallel processors with vector nodes," *Journal of Parallel and Distributed Computing*, vol. 13, no. 3, pp. 325–331, Nov. 1991. [Online]. Available: https://www.sciencedirect.com/science/article/pii/074373159190079O

[18] "LAPACK Benchmark." [Online]. Available: https://www.netlib.org/lapack/lug/node71.html

[19] S. P. Hirshman, K. S. Perumalla, V. E. Lynch, and R. Sanchez, "BCYCLIC: A parallel block tridiagonal matrix cyclic solver," *Journal of Computational Physics*, vol. 229, no. 18, pp. 6392–6404, Sep. 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999110002536

[20] Y. Zhang, J. Cohen, A. A. Davidson, and J. D. Owens, "Chapter 11 - A Hybrid Method for Solving Tridiagonal Systems on the GPU," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed. Boston: Morgan Kaufmann, Jan. 2012, pp. 117–132. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123859631000113

[21] "Solving Tridiagonal Systems on Ensemble Architectures." [Online]. Available: https://epubs.siam.org/doi/epdf/10.1137/0908040

[22] "CUDA C++ Best Practices Guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[23] V. Esfahanian, B. Baghapour, M. Torabzadeh, and H. Chizari, "An efficient GPU implementation of cyclic reduction solver for high-order compressible viscous flow simulations," *Computers & Fluids*, vol. 92, pp. 160–171, Mar. 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S004579301300491X

[24] M. Mukadam, X. Yan, and B. Boots, "Gaussian Process Motion planning," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. Stockholm, Sweden: IEEE, May 2016, pp. 9–15. [Online]. Available: http://ieeexplore.ieee.org/document/7487091/

[25] M. Souri, P. Akbarzadeh, and H. M. Darian, "Parallel Thomas approach development for solving tridiagonal systems in GPU programming steady and unsteady flow simulation," *Mechanics & Industry*, vol. 21, no. 3, p. 303, 2020, number: 3 Publisher: EDP Sciences. [Online]. Available: https://www.mechanics-industry.org/articles/meca/abs/2020/03/mi170262/mi170262.html