4-20-2023

GPU Accelerated Solvers for Toeplitz Systems

Atherton George Mook

GPU Accelerated Solvers for Toeplitz Systems

by

Atherton George Mook

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Certificate

to

The Honors College

of

The University of Alabama in Huntsville

April 19th, 2023

Honors Capstone Director: Michael M. Wise, Ph.D.

Atherton Mook	4/20/2023
Student	Date
Michael M. Wise	4/20/2023
Director	Date
	4/20/2023
Department Chair	Date
William Wilkerson Digitally signed by William Wilkerson Date: 2023.04.21 14:35:43 -05'00'	
Honors College Dean	Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

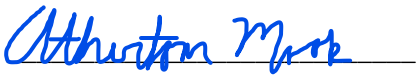
Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Atherton Mook
_____

Student Name (printed)



Student Signature

4/20/2023
_____

Date

TABLE OF CONTENTS

Abstract

Toeplitz matrices appear in a wide range of computational science and engineering applications ranging from the numerical analysis of certain differential and integral equations to digital signal processing. As a result, efficient solvers for systems of linear equations involving Toeplitz matrices are critical to the performance of not only modeling and simulation applications, but also mission-critical real-time applications. Graphics processing units (GPUs) have become the device of choice for accelerating parallelizable algorithms due to their ubiquity, their relatively easy and quick development enabled by general purpose frameworks (e.g., CUDA, HIP), and the fact that many times–with proper design–performance gains can be obtained by simply upgrading to the next generation of GPU hardware.

In this work, we develop optimized GPU-accelerated implementations for solving Toeplitz systems. As we iterate through the Assess-Parallelize-Optimize-Deploy (APOD) design cycle [6], we guide the reader through the development of our implementations and provide justifications for our optimizations. We examine performance results obtained when executing our implementations using representative benchmark problems and hardware and highlight how our implementations could be improved for future systems.

Introduction

Toeplitz matrices are matrices whose values on each diagonal are constant. A matrix is considered Toeplitz if there exist scalars $\vec{r} = \{r_{-n+1}, r_{-n+2}, \dots, r_0, r_1, \dots, r_{n-1}\}$ such that $a_{ij} = r_{j-i}$ [10]. For example, the following matrix is Toeplitz:

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} \\ a_1 & a_0 & a_{-1} \\ a_2 & a_1 & a_0 \end{bmatrix}$$

Toeplitz matrices are very useful for representing time-series data in the form of a matrix. The elements of the time series will correspond to the elements of vector $\vec{r}$ above. Solving the system

$$Ax = b \text{ [eq. 1]}$$

is often of particular interest, where the column vector b is known, and the vector x is yet to be calculated. Furthermore, many practical applications (e.g., digital array processing) give rise to the "batched problem" of concurrently solving the linear systems involving the single matrix *A* with various right hand sides *b*.

Toeplitz Matrices are useful for a variety of applications including digital signal processing and time-series forecasting. In digital signal processing and a variety of other domains, the discrete convolution operation is fundamental. Toeplitz matrices allow for discrete convolution to be represented as matrix multiplication. For example, take the following equation:

$$y = h * x$$

To convolve the two vectors, the problem can be expressed using Toeplitz matrices:

$$y = \begin{bmatrix} h_1 & 0 & 0 \\ h_2 & h_1 & 0 \\ h_3 & h_2 & h_1 \\ 0 & h_3 & h_2 \\ 0 & 0 & h_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

From equation 1, the vector *b* is the result of $a_t * x_t$. In addition, given the commutative properties of convolution and noting that discrete convolution coincides with applying a discrete filter, solving the related system where the places of h and x are switched can provide a very basic filter design. Furthermore, the system that is solved in the generation of the Wiener filter is Toeplitz due to the autocorrelation matrix of a received noisy estimate of the desired signal.

In the general case, when the matrix A can only be assumed to be non-singular, Gaussian Elimination, or an equivalent method, is used to solve linear systems involving A. This algorithm requires O(n^3) arithmetic operations and, consequently, can become prohibitively expensive as the size of A increases. However, when matrix A has a certain structure or possesses certain properties, these aspects can often be exploited to develop far more affordable numerical methods. Toeplitz matrices are a class of matrices that enable exceptional performance gains when their unique structure is exploited.

Levinson recursion is one such method requiring only O(n^2) arithmetic operations. Since its introduction in 1947, Levinson recursion has been among the most widely used algorithms for the solution of Toeplitz systems. Nevertheless, while Levinson's method is very useful practically due to its low computational complexity and low memory requirements, it can perform poorly from a stability perspective for certain problems. This has led to a significant amount of research into related methods with either better stability properties or better computational complexities. In recent years, related methods such as Schur decomposition methods and "superfast" algorithms have arisen offering additional improvements: both in terms of the number of operations and stability properties [2,17]. These "superfast" algorithms only require O($n \log n$) arithmetic operations, but they are significantly more complicated and thus harder to implement [2]. The PCG algorithm requires O($n \log^{5/2} n$) arithmetic operations [5], but the realistic operation count is very dependent upon the condition number of the Toeplitz matrix.

The computing landscape has changed and grown vastly since the introduction of Levinson's method. However, numerical methods introduced in the era of vacuum tubes and slide rules often manage to still perform effective work in the age of exascale computers [13]. Nevertheless, the differences in organization, architecture, accelerators, size, and scale offered by modern computers sometimes require that users select (or invent) algorithms designed to fully make use of these exceptionally capable machines. While many innovations have occurred in the past two and a half decades, two technologies particularly paved the way for the exascale era: multicore processors and graphics processing units [9, 15]. Computing practitioners noted the applicability of graphics processing units (GPU) to general purpose computing problems. Soon thereafter, the age of general-purpose GPU computing, or GPGPU computing as coined by Mark Harries, was born with the introduction of the Compute Unified Device Architecture or CUDA in 2007 [1].

CUDA is a C++ extension and application programming interface for effective offloading of general work to the GPU as a general-purpose accelerator. Other frameworks such as HIP, OpenACC, and SYCL seek to accomplish the same goal but emphasize portability across a variety of GPUs and accelerators. In the modern era of mature software frameworks, powerful hardware, and the democratization of high-performance computing via free industry conferences like NVIDIA's GTC, developers have more opportunities than ever before to experiment with new algorithms and architectures. Older algorithms may suffer greatly when embedded into an accelerated workflow due to data dependencies or an implicit serial perspective used when the algorithm was devised. While the number of arithmetic operations is to be considered, an algorithm parallelizability, especially in the sense of how well it will map to a given parallel architecture, is extremely important. A parallelizable algorithm using GPUs can often drastically shorten the runtime of costly computational efforts.

In this paper, we investigate algorithms for the solution of Toeplitz systems of linear equations as a case study of this topic. In particular, we follow the course set by Strang, Chan, and others [4] focusing on positive definite Toeplitz systems. We seek to investigate how mathematical structure and properties of these matrices can best be mapped to general-purpose parallel hardware to offer significant runtime improvements over traditional methods designed to run on a single core or be computed on paper. While Levinson recursion is a natural choice for a low runtime, low memory implementation that does not make use of parallelism, the structure of the algorithm and the synchronization required can make it difficult to implement in parallel to take full advantage of the device. Furthermore, superfast algorithms and other Levinson-related algorithms introduced to improve runtimes or numerical properties many times are quite complicated to implement.

Due to the parallelizability of Krylov subspace methods such as Conjugate Gradient, Strang suggested the consideration of their viability as a compromise algorithm offering reasonably decent runtimes and mathematical properties [3]. However, depending on the condition number of the matrix in question, conjugate gradient may not converge in floating point arithmetic; though in exact arithmetic, convergence is guaranteed in a number of iterations less than or equal to the number of distinct eigenvalues [3]. Consequently, the system must be preconditioned and the auxiliary system $\mathrm{PA}x = \mathrm{P}b$ must be solved to obtain a solution. Naturally, this seems to suggest additional overhead and with matrix vector multiplications (usually an O(n^2) operation) occurring each iteration, it seems that any hope of improvement might be lost, even with opportunities to exploit parallelism. However, as Strang noted, preconditioned conjugate gradient can be implemented for these systems in O($n \log n$) due to the fact that Toeplitz matrices can be multiplied in O($n \log n$) time by embedding them in a circulant matrix. Furthermore, he offered a circulant preconditioner to decrease the condition number of the auxiliary system that could be applied with no increase in the order of magnitude of the number of operations. Additional

research has been done in recent years by Chen, Strange, and others [3, 4, 5]. The fast Fourier transform (FFT) library cuFFT provided by NVIDIA makes calculation of these multiplications quite simple [8].

This paper is organized as follows: In Section 1, we introduce the GPU architecture, CUDA, and discuss optimizations relating to the GPU's architecture. Section 1 ends with a description of the APOD design cycle [6]. In Section 2, we introduce Levinson recursion, and give a description of the algorithm. Then, we walk through the APOD design cycle as applied to the algorithm, giving insight into how testing was conducted and presenting relevant performance metrics. In Section 3, we introduce PCG, and present its algorithm. Then, we showcase two implementations and show performance metrics, comparing PCG to Levinson recursion. In Section 4, we apply both GPU implementations to a real-world example. Finally, we conclude the paper and discuss future work. We provide relevant source code as an appendix.

Section 1: GPU Introduction

This section introduces the basic architecture of the targeted GPU device—the NVIDIA A100-SXM4-40GB. After this architectural discussion, we introduce the CUDA framework with which it will be programmed along with relevant techniques to make best use of the hardware. GPUs are based around the concept of many slower cores running the same instruction in parallel. This paradigm is called SIMD (Single Instruction Multiple Data), and Nvidia specifically uses a slightly different model called SIMT (Single Instruction Multiple Threads). In CUDA, threads are logically divided into logical groupings: the warp, the block, and the grid. The user launches their work onto the GPU using an (up to) three-dimensional grid of (up to) three-dimensional blocks of lightweight hardware threads that are scheduled together in warps: the fundamental unit of parallel execution. Blocks are assigned for execution on each of the streaming multiprocessors (SMs) of the GPU and subdivided into warps for scheduling. When a warp of threads (usually 32) is launched on the hardware, each thread of execution within a warp executes the exact same instruction—just on different data. A block is a collection of warps, and a grid is a collection of blocks. Blocks and grids can be logically organized in three axes: x, y, and z. For example, to process a large 4K image that is 2,160 pixels tall and 3,840 pixels wide, that image could be subdivided starting from the top left into many rectangles of size 16 by 32. We could then map the rectangles of pixels to a block and assign each thread within that block a singular pixel to process. Finally, we could assign the entire collection of rectangles to a 135 by 120 grid.

Each grouping (warp, block, grid) has different synchronization and memory sharing capabilities. A block can be easily synchronized, and it has shared memory—so threads within the same block can easily communicate and synchronize. However, to share data between blocks, global memory must be used, which is far slower. As mentioned above, warps are scheduled to execute on hardware called a Streaming

Multiprocessor (SM). All warps from the same block are scheduled onto the same SM. Here is a graphic from Nvidia's A100 whitepaper showcasing a SM [12]:

Figure 1: Nvidia GA100 SM Architecture

From this graphic, each SM in the GA100 can execute 4 warps at the same time—for a total of 128 threads of concurrent execution. As shown, each warp shares an instruction cache, and the SM contains shared memory for the block to use. The A100's SMs contains 64 CUDA cores, so it can execute 2 warps concurrently. The A100 contains 108 SMs, so it is very clear why GPUs are so useful for acceleration: if an algorithm is parallelizable, it can be worked on by thousands of threads at the same time.

Given this architecture, there is a multitude of design practices by which one can optimize algorithms and improve performance, many of which are explained in the CUDA C++ Best Practices Guide [6]. Device utilization refers to how much of the GPU device is used. If more of the device is used, more threads execute in parallel, so each thread does less work. Occupancy is defined as the ratio of active

warps on an SM to the maximum number possible. In computing, memory accesses are very often a bottleneck for computing, as it can take many clock cycles for a memory request to be serviced. To hide this latency, when one warp is stalled on a memory request, another warp can be switched in and executed. So, higher occupancy can allow for faster execution. Another important optimization is to minimize control divergence. With the SIMD paradigm, when an if statement/branch is reached, some of the warp might need to execute different instructions than other parts. To achieve this behavior, all the threads go down both paths of the branch, and the threads that do not need to be executed are masked off—so they are doing no useful work. So, minimizing control divergence ensures that threads are doing useful work. There are many other important optimizations that we will not explicitly cover: such as memory coalescing and thread coarsening, but they are also important and detailed in the Best Practices Guide.

In the CUDA Best Practices Guide, NVIDIA recommends the Assess-Parallelize-Optimize-Deploy (APOD) design cycle [6]. For a given algorithm or codebase, the first step is to assess the code, taking note of the most expensive portions of the computation and how these portions might be refactored to take advantage of available parallelism. Next, the parallelizable sections must be implemented in parallel. When using the GPU, many times this can be using a combination of custom kernels (user code directly interacting with a GPU) or through library calls that utilize their own kernels behind their API. Next, by keeping in mind the optimizations above and from the guide, the developed code should be optimized to improve performance. To complete this step, the developer must measure frequently to decide what to optimize, as it is usually most advantageous to prioritize optimizations on the most expensive portions of the codebase before focusing on optimizing components that contribute only a small amount to the total runtime. Finally, it is important to deploy the new code and test it in a realistic application. This allows for a deeper understanding of critical optimizations, highlights, and edge cases that might not have been considered, and it propels the developer toward the next cycle of this incremental engineering process.

During the development of the code for this paper, the APOD design cycle was put into practice with most of our development time being spent in the optimize and deploy steps.

Section 2: Levinson-Durbin Recursion

Levinson-Durbin Recursion is often considered the industry standard for solving Toeplitz systems. It has a time complexity of $O(n^2)$ [11]. Here is the algorithm for performing Levinson recursion:

$x \leftarrow \text{LevinsonRecusion}(A, b)$	
1	Initialize V_f, V_b, V_x to zero vector of length N
2	Initialize $V_f(0), V_b(0), V_x(0)$ to $1/A_{0,0},\ 1/A_{0,0}, 1/A_{0,0}, b_0/A_{0,0}$ respectively
3	WHILE k < N
4	Compute $\text{Eps}_f, \text{Eps}_b, \text{Eps}_x$
5	Use $\text{Eps}_f, \text{Eps}_b, \text{Eps}_x$ to calculate new V_f, V_b, V_x
6	Increment k
7	END WHILE

Levinson-Durbin Recursion GPU Implementation

To develop the GPU implementation, we referenced a Matlab implementation created by user RoyiAvital on GitHub [16]. We verified this reference implementation using the Matlab ' \ ' operator ($x = A\backslash b$). This implementation was very accurate and served as a very useful reference when developing our GPU implementation.

Implementing Levinson Recursion to take full advantage of a GPU's hardware presents significant challenges. Given the recursive nature of the algorithm, each step of the algorithm has a data dependency from the previous iteration. Also, for each step in the recursion, the vectors V_f, V_b, V_x are only of length *k*, where *k* is the current iteration. Consequently, in the early stages, very few threads are capable of being utilized. Furthermore, the algorithm also requires frequent synchronization between the steps. In parallel

computing, synchronizations are often very expensive because it requires computational units to wait, performing no useful work while other units finish executing. Given the need for synchronization, we completed a simple first-pass at an accelerated implementation using a single CUDA block. Inter-block synchronization semantics (e.g., __syncthreads()) are significantly less expensive and less programmatically intensive compared to intra-block cooperation paradigms. This first version operates on 64-bit doubles, and it has a block size of 512 threads. Here are the results from the very first version:

Table 1: Levinson Single Block Metrics

n	t (ms)	Error
1024	2.14241	1.61E-12
2048	4.51092	1.62E-12
4096	11.9181	1.65E-12
8192	52.3956	1.66E-12
16384	296.548	1.67E-12
32768	1209.73	1.73E-12

The error term refers to the relative error. To test the error, the algorithm outputs a csv file of the resulting *x* vector. The resulting vector was multiplied by the input Toeplitz matrix and the result compared to the original b vector from equation 1. The calculated error is relative error, defined here:

$$\text{error} = \frac{\|\vec{g} - \vec{b}\|}{\|\vec{b}\|}, \vec{g} = \text{gpu output}; \vec{b} = \text{expected output}$$

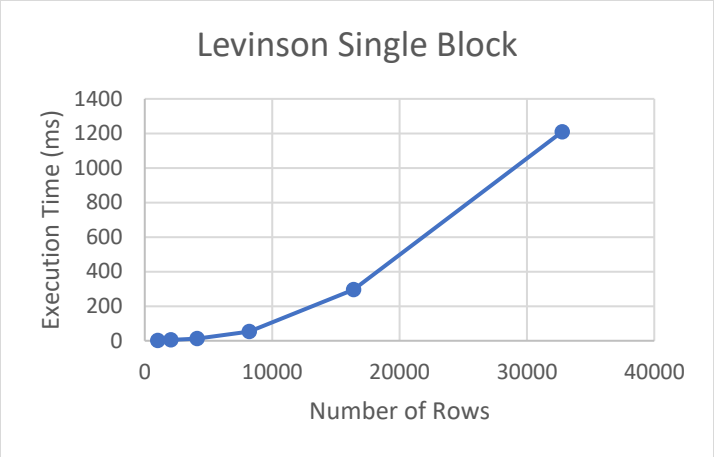


Figure 2: Lev Single Block Execution Time

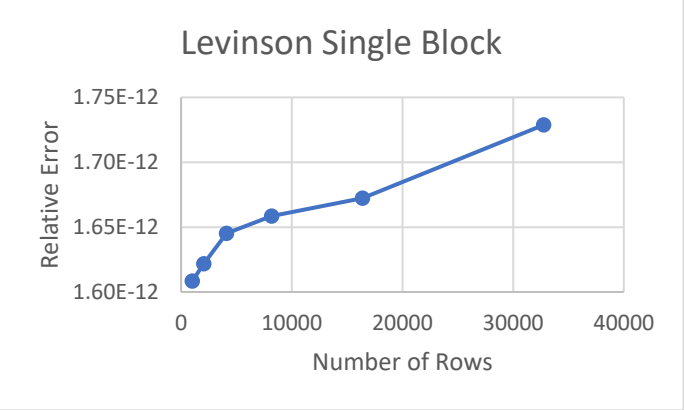


Figure 3: Lev Single Block Error

As the problem size increases, the execution time also grows very large. The error is relatively small for small problem sizes, and it grows larger as the problem size increases. However, for very large problems, the error term reduces. Here are some additional metrics from Nvidia Nsight Compute for running a problem size of n=16384:

Compute (SM) Throughput [%]	0.23 (+0.00%)
Memory Throughput [%]	0.53 (+0.00%)

Figure 4: Levinson Single Block Throughput

The first metric shows how much computational throughput is achieved compared to theoretical maximums. The second metric shows the memory throughput compared to the theoretical maximums.

Since only one block is used, very little of the device ends up being utilized—leading to the very low throughput values of 0.23% and 0.53%.

Now, the next logical step for better performance is utilizing more of device—instead of just one block. Traditionally, synchronizing between blocks on a GPU was not a natively supported feature, and required “hacky” solutions. However, our version of CUDA supports inter-block synchronization using Cooperative Groups. Syncing between blocks, unfortunately, is very expensive. Despite the overhead of that synchronization, the ability to utilize the entire device is well worth the overhead.

Now, each key step from the Levinson algorithm presented above is split across multiple blocks. This also introduced extra communication needed to reduce summations between blocks. To accomplish this reduction, each block computes its own partial sum, and then atomically adds its result to global memory accessible by every block.

This version, which we refer to as Levinson Multiblock, operates on 64-bit double precision floating point values, and it has a block size of 512 threads. The error calculations were the same as the above version, and the same inputs were supplied. Here is the execution data:

Table 2: Levinson Multi Block Metrics

n	t (ms)	Error
1024	7.2129	1.61E-12
2048	14.484	1.62E-12
4096	29.377	1.65E-12
8192	59.992	1.66E-12
16384	122.94	1.67E-12
32768	297.36	1.73E-12

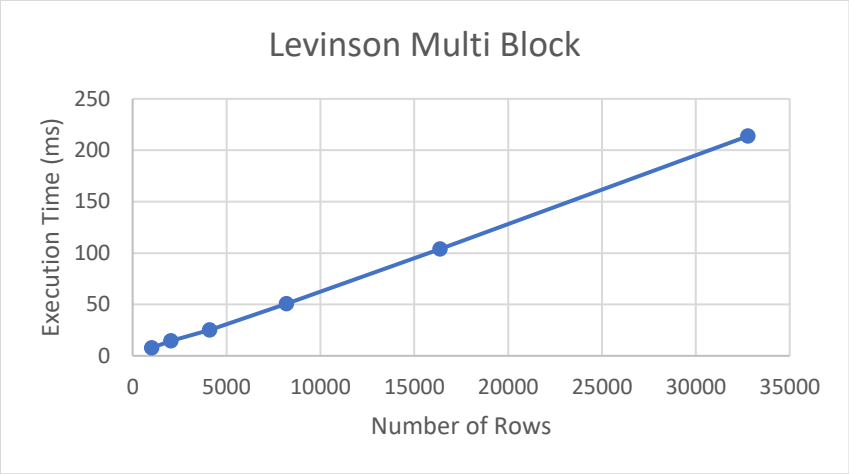


Figure 5: Levinson Multiblock Execution Time

For this version, the execution time seems to vary linearly with the problem size despite the number of operations being $O(n^2)$. This could be due to the large number of threads that the A100 supports. Consequently, as the problem size increases, more threads are allocated to the problem. This version is significantly faster for large problem sizes, but for small problem sizes it is slower. Logic could be introduced to select the appropriate kernel based on which is faster for a given problem size and architecture.

Table 3: Single Block vs Multi Block

n	E.T. Single Block	E.T. Multi Block	Speedup
1024	2.14241	6.9857	0.306685
2048	4.51092	11.506	0.392049
4096	11.9181	23.479	0.507607
8192	52.3956	46.787	1.119875
16384	296.548	96.329	3.078491
32768	1209.73	234.68	5.154807

When analyzed with Nsight Compute, for the problem size of n=16384, these metrics were generated:

Compute (SM) Throughput [%]	0.93 (+313.90%)
Memory Throughput [%]	3.01 (+471.90%)

Figure 6: Levinson Groups Throughput

This figure has the first Levinson version as a baseline. So, the compute throughput has increased by 313%, and the memory throughput by 471%. However, these metrics are still very, very low. This is due to the synchronization required by Levinson recursion. To achieve higher throughput, more threads need to be actively computing something useful, rather than waiting for synchronizations or not being used at all. This leads to our next iteration: batched Levinson.

The next version developed supports batched inputs. As mentioned at the start of this section, in Levinson recursion, the length of the vector operations is equal to the current iteration, so lots of threads end up doing nothing. By batching, there will be lots of smaller problem sizes happening at the same time, so this issue will not be as large substantial. To implement batched, two main ideas came to mind: the first is to iterate over every batch at each stage in the recursion and the second is for many concurrent kernels to be launched, where each kernel handles a small number of batches. For example, with the first method when calculating the epsilon values, the kernel would perform the calculations for each batch before moving on. We opted for the latter option, where the cooperative groups kernel (previous iteration) was modified such that each kernel can loop over the number of batches allocated to it; this ended up being four batches. To justify the second method, we will first discuss the implications of the first method. In the first method, for the first iteration, each batch only needs one element operated upon. Given how the problem is logically divided, a single thread would calculate all 128 first elements while every other thread remains stagnant. If the batches could be logically divided better, this problem would go away; this is essentially what the second option does. So, using the second option with each kernel handling 4 batches, 32 kernels need to be launched in parallel. So, at the first iteration, the 128 first elements are divided among 32 threads—a huge improvement over 1 thread. Streams were used to achieve this functionality. The number of blocks is set as follows:

$$\text{Grid Size} = \text{ceil}(n/(2*\text{block\_size})), \text{block\_size} = 512 \text{ threads}$$

By allocating fewer blocks for each kernel, more kernels can run on the GPU at the same time. Here are the performance metrics of running with 128 batches:

Table 4: Levinson Batched Metrics

n	t (ms)	Error
1024	21.0285	1.61E-12
2048	40.4799	1.62E-12
4096	88.8928	1.65E-12

The error is the average of every batch's error. Using Nsight Compute to compare throughput between batched and non-batched presents difficulties, because Nsight Compute produces metrics for a single kernel, and given our batched implementation executes many kernels in parallel, the metrics in Figure 8 do not offer as significant of insight into performance as they do in other cases we present.

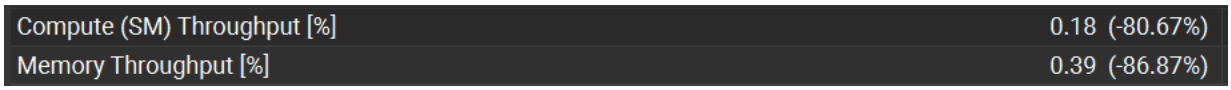


Figure 7: Levinson Batched Throughput

For a single kernel, out of the 32 executing in parallel, the compute throughput decreases by 80%. This is also due to the problem size being decreased down to 1024 per batch, so each kernel has less to do. One metric to compare batched to non-batched is to compare the execution time per element. For non-batched, the number of elements is simply the number of rows. For batched, it is the number of rows multiplied by the batch size. So, in terms of execution time per element, batched outperforms non-batched by a significant margin as shown by the table below.

Table 5: Batched and Non-batched Comparison

	Total number of elements		E.T. total		E.T. per element	
number rows	Non-batched	batched	Non-batched	Batched	Non-batched	Batched
1024	1024	131072	7.7789	21.0285	7.60E-03	1.60E-04
2048	2048	262144	14.627	40.4799	7.14E-03	1.54E-04
4096	4096	524288	25.003	88.8928	6.10E-03	1.70E-04
8192	8192		50.583		6.17E-03	
16384	16384		103.96		6.35E-03	
32768	32768		213.75		6.52E-03	

Section 3: Preconditioned Conjugate Gradient

The conjugate gradient algorithm is (in non-exact arithmetic) an iterative method for solving Hermitian positive-definite systems, i.e., systems $Ax = b$ where the $n \times n$ complex-valued matrix A is Hermitian ($A = A^H$, where $A^H = conj(A^T)$) and positive-definite, i.e., for any complex-valued vector x , $x^H Ax > 0$. For simplicity, we restrict ourselves to the case of real systems, though the implementation discussed herein can easily be modified to support complex data.

The number of iterations in which conjugate gradient converges in exact arithmetic is bounded above by the number of distinct eigenvalues [3]. However, in non-exact arithmetic such as the floating-point variety used here, the number of iterations necessary to converge to a certain tolerance depends strongly on the condition number of the matrix. For a given norm $\text{norm}(\cdot)$, the condition number is given by $\kappa(A) = \text{norm}(A)\text{norm}(A^{-1})$. In this work, we use the matrix norm induced by the Euclidean vector norm $\text{norm}_2(x) = \sqrt[2]{\sum_{i=0}^{n-1} x_i^2}$. In general, the value of this particular matrix norm coincides with the largest singular value and is also called the spectral norm. For the class of symmetric positive definite matrices, we note that the value of this norm is then given by the maximum eigenvalue of the matrix. Consequently, in this special case, the condition number is simply the ratio of the largest and smallest eigenvalues of the matrix. Indeed, if this condition number grows large enough, i.e., the largest eigenvalue is far removed from the smallest, the conjugate gradient method can fail to converge in non-exact arithmetic.

Since the condition number directly impacts the practicality of the method, one usually applies the conjugate gradient to the preconditioned system $PAx = Pb$. The idea is to choose the matrix P such that the product PA has a condition number significantly smaller than that of A . If the condition number is made small enough, conjugate gradient will converge to a given tolerance in a significantly smaller number of iterations, even in cases that before failed to converge altogether. To be practical, the preconditioner must

also be sufficiently efficient to compute. Indeed, there is a balance to be struck between the cost of the application of the preconditioner and its quality, how small it makes the condition number of PA .

In the case of Toeplitz systems, the structure of the matrix can be exploited to create an extremely efficient PCG scheme. Strang noted that by embedding the Toeplitz matrix in a circulant matrix, the matrix vector products at the core of the conjugate gradient method can be done in O($n \log n$) time using the Fast Fourier Transform [3]. If a circulant preconditioner is chosen, preconditioned conjugate gradient can be executed in O($n \log^{5/2} n$) time [5]. Chen et al. explain this property further, and the algorithm presented below is outlined in their paper, and during development we used the Matlab implementation developed by user zhwufd on Github as a reference. [5, 18]. We use the same circulant preconditioner from Chen et al., the T. Chan preconditioner [4]. We present the PCG algorithm below, customized for our case of Toeplitz matrices. Subsequently, we define the subalgorithms utilized in the definition of the PCG algorithm.

$x \leftarrow \mathrm{PCG}(a_c, \mathrm{a}_r, b, \text{tolerance}), a_c = \text{first column of } A; \mathrm{a}_r = \text{first row of } A$	
1	$\mathrm{c} = \ \mathrm{precondition}(a_c, \mathrm{a}_r)$
2	$p_1 = \mathrm{circulant\_solve}(c, b)$
3	$y_0 = p_1;\ r_0 = b$
4	WHILE $\mathrm{norm2}(r_0 > \mathrm{tolerance})$
5	$z = \mathrm{toeplitz\_mul}(a_c, a_r, p_1)$
6	$\alpha = (y_0 \cdot r_0)/(p_1 \cdot z)$
7	$x_0 = x_0 + \alpha p_1$
8	$r_1 = r_0 - \alpha p_1$
9	$p_1 = \mathrm{circulant\_solve}(c, r_1)$
10	$\beta = (y_1 \cdot r_1)/(y_0 \cdot r_0)$

11	$p_1 = y_1 + \beta p_1$
12	$y_0 = y_1; r_0 = r_1$
13	END WHILE
14	$x = x_0$

$x \leftarrow \text{circulant\_solve}(c, b)$	
1	$e = \text{length}(c) * \text{ifft}(c)$
2	$x = \text{Re}(\text{fft}(\text{ifft}(\text{b})./e))$

$b \leftarrow \text{toeplitz\_mul}(a_c, a_r, x)$	
1	$n = \text{length}(a_c)$
2	$t_k = \begin{cases} a_c(k) & \text{for } k < n \\ 0 & \text{for } k = n, 0 \le k < n \\ a_r(n-k+1) & \text{for k > n} \end{cases}$
3	$xx_k = \begin{cases} x(k) & \text{for } k < n \\ 0 & \text{for } k \ge n \end{cases}, \quad 0 \le k < 2n$
4	$e = 2n$*ifft(t)
5	$b = \text{Re}(\text{fft}(e.* \text{ifft}(xx)));$

PCG GPU Implementation

Compared to Levinsons, PCG offers significantly more opportunity for parallelization Rather than lots of quick iterations, PCG has fewer iterations that take more time—meaning fewer dependency issues between iterations. Also, PCG makes use of FFTs—a highly optimized library function—to perform lots of

its operations. Unlike the developed Levinson implementation that uses one custom kernel, our implementation of PCG makes use of many library functions from cuBLAS and cuFFT.

It was also developed in a more object-oriented manner, where the major components of the algorithm have their own class, implemented as functors or, loosely, C++ classes that implement the () operator. The three main components are the circulant solver, the Toeplitz multiplier, and the overall PCG solver. The T. Chan preconditioner was used, and given its simplicity, a custom kernel was made for it. The constructor for each class allocates the needed device memory and sets up the required cuBLAS handles and cuFFT plans. This construction is expensive, but it only needs to happen once, so the timing cost was excluded from the results. Here are the results from operating on 64-bit doubles, using the relative error from the corresponding Levinson experiments as a stopping criterion to enable fair comparison:

Table 6: PCG Metrics

n	iterations	t (ms)	error
1024	8	2.6604	1.61E-12
2048	8	3.0853	1.62E-12
4096	9	3.6659	1.64E-12
8192	9	4.8845	1.65E-12
16384	10	5.4036	1.65E-12
32768	11	4.2619	1.65E-12

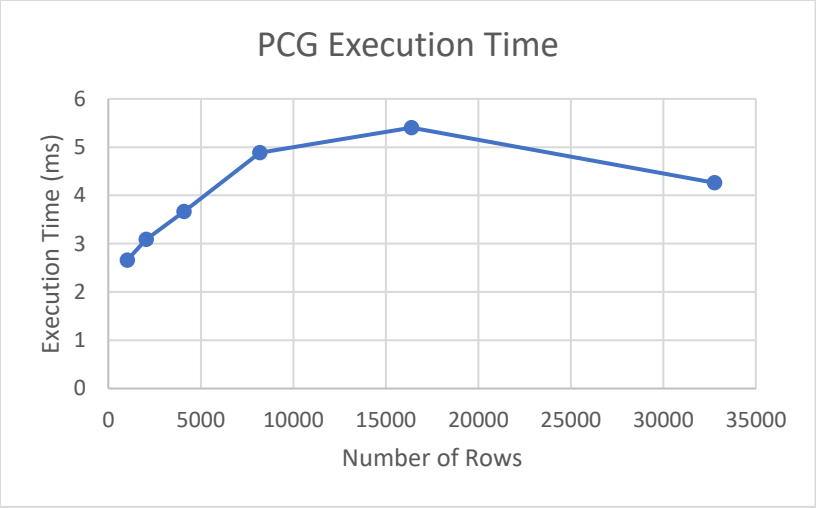


Figure 8: PCG Execution Time

The execution time of PCG grows slightly, but it remains very low for large problem sizes. Like Levinson recursion, smaller problem sizes are not capable of utilizing the entire GPU. While larger problem sizes require more arithmetic operations, more of the GPU can be used to solve the problem. To reach an error comparable to Levinson recursion, PCG requires 8 to 11 iterations. However, the execution time is significantly better for problem sizes greater than 4096. Here is a chart comparing the execution time of PCG to that of multiblock Levinson:

Table 7: Comparison of Levinson and PCG

n	E.T. Lev	E.T. PCG	Speedup
1024	6.9857	2.6604	2.625808
2048	11.506	3.0853	3.729297
4096	23.479	3.6659	6.404703
8192	46.787	4.8845	9.578667
16384	96.329	5.4036	17.82682
32768	234.68	4.2619	55.06464

Next, a batched version of PCG was implemented. The library cuFFT offers very easy support for batched input, so changing the circulant solver and the Toeplitz multiplier was straightforward given the FFTs were the bulk of those two functors. The batched implementation went through several iterations: a

naïve approach simply iterating through each batch at a given step, a version using streams to execute some aspects in parallel, and a final version using library calls in a different way to support batched.

The first iteration used the updated circulant solver and the Toeplitz multiplier, but between the usage of those two functions, some intensive computations take place: dot products and axpys. An axpy performs the following calculation on vectors:

$$y = \alpha x + y$$

For the first iteration, each dot product and axpy was performed b times in a for loop, where b is the number of batches. Without using streams, this essentially serialized execution of these operations, and most of the GPU was unused. Here are some metrics for this first version:

Table 8: Batched PCG Naive Metrics

n	iterations	t (ms)	Error
1024	2	30.762	0.0011
2048	2	211.301	0.0042
4096	2	208.906	0.0126

The execution time for this version is very high, and Levinson batched is much faster. The error shown is also significantly worse due to the fact that we artificially stopped the number of iterations as runtime was already exceeding that of Levinson's. So, in reality, the execution time would be significantly worse due to the extra iterations needed for comparable accuracy.

The cuBLAS functions to perform the axpy and the dot product did not have batched versions, so a different approach needed to be used. The easiest approach would be to use CUDA streams so the smaller operations can be executed concurrently. To achieve streamed execution in cuBLAS, we simply created a handle for each batch, and set each handle to run on its own stream. Some other less-intensive operations (such as converting between complex and real datatypes) were also converted to use streams. Like batched Levinson recursion, all the following problem sizes are for 128 batches. Here are the metrics for this version:

Table 9: Batched PCG v2 Metrics

n	iterations	t (ms)	Error
1024	5	25.4046	8.43E-04
2048	5	25.3372	0.0017
4096	6	30.9871	0.0066

Just by applying streams, the execution time significantly decreases. For larger problem sizes it is clearly superior to the naïve implementation. For the n=1024, it performs faster with more iterations. The error, however, was still not comparable to Levinson recursion batched, likely due to an implementation bug.

For the final version of PCG batched we changed the axpy and dot product operations to use batched versions, rather than streams. For the axpy, a simple batched kernel was developed. Since batches are contiguous in memory, the only detail that changes from a traditional axpy kernel is that the alpha value is different from each batch. So, the index of the grid stride loop determines the alpha value. For dot products, the cuBLAS function "gemmStridedBatched" was used. This function performs a batch of generic matrix multiplies. Given that each element of the output matrix is a dot product of the row and column of the two input matrices, this function can be used to compute a batched dot product. For the equation $AB = C$, A is a column vector, and B is a row vector, so C is the resulting dot product. Here are the resulting metrics:

Table 10: Batched PCG Optimized Metrics

n	iterations	t (ms)	Error
1024	4	7.08938	1.62E-12
2048	5	8.72483	1.62E-12
4096	5	8.90954	1.64E-12

This version executes significantly quicker than the previous version, and the error is comparable to that of Levinson recursion. Here is a table comparing the performance of PCG batched to regular PCG based on per element execution time.

Table 11: PCG Batched vs Non-Batched Comparison

	Total number of elements		E.T. total		E.T. per element	
number rows	Non-batched	batched	Non-batched	Batched	Non-batched	Batched
1024	1024	131072	2.6604	7.08938	2.60E-03	5.41E-05
2048	2048	262144	3.0853	8.72483	1.51E-03	3.33E-05
4096	4096	524288	3.6659	8.90954	8.95E-04	1.70E-05
8192	8192		4.8845		5.96E-04	
16384	16384		5.4036		3.30E-04	
32768	32768		4.2619		1.30E-04	

Like Levinson recursion, batched execution sees a speedup on a per element basis. Here is a table comparing batched PCG to batched Levinsons for batch size 128:

Table 12: PCG and Levinson Batched Comparison (batch size 128)

n	E.T. Lev	Lev Error	E.T. PCG	PCG Error	Speedup
1024	21.0285	1.61E-12	7.08938	1.62E-12	2.966197
2048	40.4799	1.62E-12	8.72483	1.62E-12	4.63962
4096	88.8928	1.65E-12	8.90954	1.64E-12	9.97726

Consequently, PCG sees a reasonable speedup for batched run—as was the case for Levinson.

Section 4: Applications to Real-World Problems

In this section, we emphasize the importance of the last step of the APOD design cycle, the deploy step. There are a host of problems from digital signal processing in which Toeplitz or block Toeplitz matrices arise. Of these, we select that of constructing a Wiener filter. The causal FIR Wiener filter $h[n]$ of order N offers minimum mean squared error between the quantity $x[n] = \text{filter}(h[n],\ 1,\ w[n])$ and a signal $s[n]$ based on the statistical properties of the "reference" signal $s[n]$ and an "alternate" received signal $w[n]$, under some basic assumptions of stationarity, etc. After some mathematics, the problem of finding the coefficients of $h[n]$ amounts to the solution of another system $Ax = b$ where A is the (truncated) Hermitian positive definite autocorrelation matrix of the signal $w[n]$ and b is the (truncated) cross-correlation between the signal $w[n]$ and the signal $s[n]$ [14]. For a practical example, one may know that two received signals contain versions of the same signal corrupted by additive noise and perhaps a delay. In this example, we constructed a parametrizable Python program of this scenario, allowing us to specify signals $s[n]$ and $w[n]$ with a given SNR generated as time-delayed versions of some known sinusoid. Using functions available in numpy and scipy, we created a reference for the calculation of the Wiener filter. Finally, we applied the resulting filter to realign the signals in time. We then tested our Levinson and PCG implementation to compute the same filter taps and applied the resulting filter. The

results are presented in Figure 9. Levinson recursion was used on the left, and PCG on the right. Note that both implementations produce the expected results from the top of each graph.

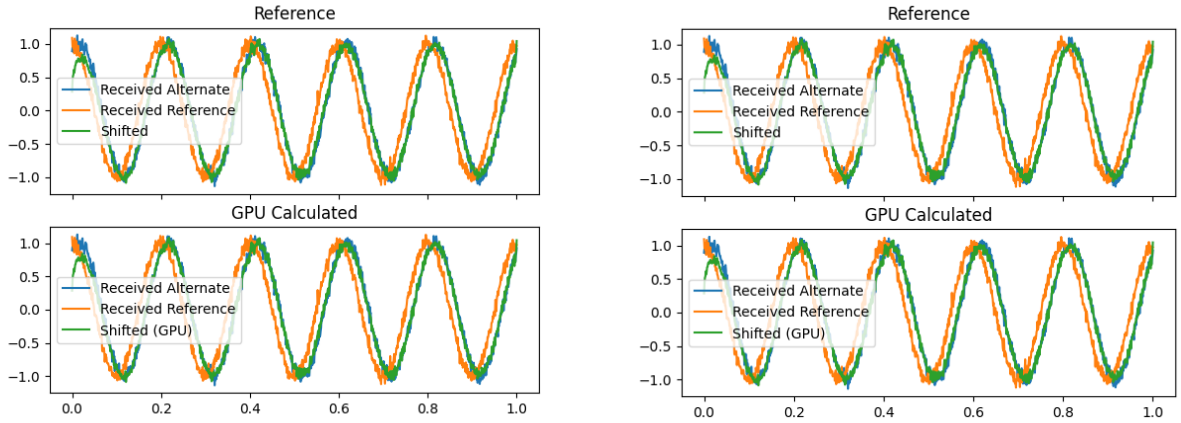


Figure 9: Shifted Time Signal using Levinsons (left) and PCG (right)

Note that the condition numbers for the autocorrelation matrices generated here were at least 3.6e5, suggesting our solvers are useful beyond the limited test cases used in development, a key feature of the Deploy phase of the APOD development cycle.

Conclusion

In this work, we studied the implementation of key solvers for Toeplitz matrices using GPUs as a general-purpose accelerator. We compared and contrasted implementations and performance of Levinson Recursion, a relatively classic algorithm for the solution of Toeplitz systems, and the Preconditoned Conjugate Gradient (PCG) method, an iterative method widely used throughout computational science and engineering. We showcased that Levinson recursion is indeed possible to implement on a GPU, but the algorithm structure hinders it from taking full advantage of the device's parallel architecture. Iterating through the APOD cycle with Levinson recursion, we showed how basic optimizations can be made to improve performance, and how by extending the use case (i.e. targeting batched implementation suggested by real-world problems like array processing), performance can be improved even more on a per-element basis.

With PCG, the importance of parallelizability became evident. For larger problem sizes, PCG vastly outperformed Levinson recursion. This was also with a relatively unoptimized version of PCG. In future work, we will plan to perform additional optimizations, perhaps most notably via a type of kernel fusion called cuFFT callbacks [7]. These callbacks allow a user to perform pre- and post-processing kernels that would normally occur before an FFT along with the initial load or final store of the FFT. This makes the need for custom data realignment, zero padding memcpys, and scaling kernels altogether unnecessary. We will also perform the minimal extensions needed to extend our code to work on real and complex single- and double-precision input. Finally, it would be beneficial to extend the suite of solvers developed to include the more stable superfast and Schur decomposition-based approaches as well as similar iterative methods that could take advantage of the fast matrix-vector multiplications supported by the Toeplitz structure (e.g., GMRES).

In conclusion, in an era of exceeding capable hardware and novel architectural changes facilitating high-performance, parallel execution, it is critical not only to select algorithms with good mathematical properties, low numbers of arithmetic operations, and memory requirements, but also to select algorithms that maintain these properties while mapping well to available hardware resources. Furthermore, it is critical to use knowledge of the architecture, organization, and the underlying physical components when designing and implementing algorithms that take full advantage of such capable devices. While CPUs and serial execution are still very important, GPUs and hardware accelerators will only continue to grow, so its time to think parallel.

Reference List

[1] “Author: Mark Harris.” *NVIDIA Technical Blog*, https://developer.nvidia.com/blog/author/mharris/.

[2] Brent, Richard P. “Parallel Algorithms for Toeplitz Systems.” *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, 1991, pp. 75–92., https://doi.org/10.1007/978-3-642-75536-1_4.

[3] Chan, Raymond H., and Gilbert Strang. “Toeplitz Equations by Conjugate Gradients with Circulant Preconditioner.” *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 1, 1989, pp. 104–119., https://doi.org/10.1137/0910009.

[4] Chan, Tony F. “An Optimal Circulant Preconditioner for Toeplitz Systems.” *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 4, 1988, pp. 766–771., https://doi.org/10.1137/0909051.

[5] Chen, Willa W, et al. “On the Correlation Matrix of the Discrete Fourier Transform and the Fast Solution of Large Toeplitz Systems for Long-Memory Time Series.” *Journal of the American Statistical Association*, vol. 101, no. 474, 2006, pp. 812–822., https://doi.org/10.1198/016214505000001069.

[6] *CUDA C++ Best Practices*, 19 Apr. 2023, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[7] “Cuda pro Tip: Use CUFFT Callbacks for Custom Data Processing.” *NVIDIA Technical Blog*, 21 Aug. 2022, https://developer.nvidia.com/blog/cuda-pro-tip-use-cufft-callbacks-custom-data-processing/.

[8] “CuFFT Docs.” *CuFFT*, Nvidia, 19 Apr. 2023, https://docs.nvidia.com/cuda/cufft/.

[9] Dally, William J., et al. “Evolution of the Graphics Processing Unit (GPU).” *IEEE Micro*, vol. 41, no. 6, 2021, pp. 42–51., https://doi.org/10.1109/mm.2021.3113475.

[10] Golub, Gene H., and Van Loan Charles F. “Classical Methods for Toeplitz Systems.” *Matrix Computations*, Johns Hopkins Univ Press, Baltimore, 2013, pp. 208–219.

[11] Levinson, Norman. “The Wiener (Root Mean Square) Error Criterion in Filter Design and Prediction.” *Journal of Mathematics and Physics*, vol. 25, no. 1-4, 1946, pp. 261–278., https://doi.org/10.1002/sapm1946251261.

[12] "NVIDIA A100 Tensor Core GPU Architecture" nvidia.com. Nvidia. 4/19/2023. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[13] “ORNL’s Frontier First to Break the Exaflop Ceiling.” *TOP500*, https://www.top500.org/news/ornls-frontier-first-to-break-the-exaflop-ceiling/.

[14] Plett, Gregory, and Max Kamenetsky. “Lecture 12 Wiener Filtering.” EE264.

[15] “Power 4 The First Multi-Core, 1GHz Processor.” *IBM100 - Power 4 : The First Multi-Core, 1GHz Processor*, https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/#:~:text=In%202001%2C%20IBM%20introduced%20the,more%20than%20170%20million%20transistors.

[16] RoyiAvital. “Projects/Levinsonrecursion.” *GitHub*, https://github.com/RoyiAvital/Projects/tree/master/LevinsonRecursion.

[17] Stewart, Michael. “A Superfast Toeplitz Solver with Improved Numerical Stability.” *SIAM Journal on Matrix Analysis and Applications*, vol. 25, no. 3, 2003, pp. 669–693., https://doi.org/10.1137/s089547980241791x.

[18] Zhwufd. “Zhwufd/Toeplitz: Fast Solver for Symmetric Positive Definite Toeplitz System with Preconditioned Conjugate Gradient Method.” *GitHub*, 2014, https://github.com/zhwufd/toeplitz.

Appendix 1: Batched Levinson Code

Lev_batched kernel.cu

```
#include <cub/cub.cuh>
#include <stdio.h>
#include <iostream>

#define LOCAL_B (batch%BATCHES_PER_PARTITION)
namespace cg = cooperative_groups;
struct Vects {
    DATA_TYPE vF[NUM_ROWS];
    DATA_TYPE vB[NUM_ROWS];
    DATA_TYPE vX[NUM_ROWS];

    DATA_TYPE vF_Prev[NUM_ROWS];
    DATA_TYPE vB_Prev[NUM_ROWS];

    DATA_TYPE epsF, epsB, epsX;
};

__device__ void sync_device(cg::grid_group* grp) {
    // Function to sync the entire cooperative group
    cg::sync(*grp);
}
__global__ void levRecursionKernel(DATA_TYPE *mT, DATA_TYPE *vY, DATA_TYPE *vX_out,
Vects* vects, int numRows) {
    // This kernel is performs Levinson recursion on multiple batches.

    // Each batch needs its own epsilon values
    DATA_TYPE epsF_local[BATCHES_PER_PARTITION];
    DATA_TYPE epsB_local[BATCHES_PER_PARTITION];
    DATA_TYPE epsX_local[BATCHES_PER_PARTITION];
    DATA_TYPE scalingFctr[BATCHES_PER_PARTITION];

    unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int block_id = threadIdx.x;

    cg::grid_group group = cg::this_grid();

    unsigned int gsl_inc = blockDim.x * gridDim.x;

    // Block reducer
    typedef cub::BlockReduce<DATA_TYPE, BLOCK_SIZE> BlockReduce;
    __shared__ typename BlockReduce::TempStorage temp_storageEpsF;
    __shared__ typename BlockReduce::TempStorage temp_storageEpsB;
    __shared__ typename BlockReduce::TempStorage temp_storageEpsX;

    // Initilize memory
    for (int batch = 0; batch < BATCHES_PER_PARTITION; batch++) {
        for (int index = id; index < NUM_ROWS; index += gsl_inc)  {
            vects[batch].vF[index] = 0.0;
            vects[batch].vB[index] = 0.0;
            vects[batch].vX[index] = 0.0;
            vects[batch].vF_Prev[index] = 0.0;
            vects[batch].vB_Prev[index] = 0.0;
        }

        // Setup iteration=0  elements
```

```
55.         if (id == 0) {
56.             vects[batch].vF[0] = 1.0 / mT[NUM_ROWS-1]; //mt[0]
57.             vects[batch].vB[0] = 1.0 / mT[NUM_ROWS-1];
58.             vects[batch].vX[0] = vY[batch*NUM_ROWS] / mT[NUM_ROWS-1]; //vY 0
59.             vects[batch].epsF = 0.0;
60.             vects[batch].epsX = 0.0;
61.             vects[batch].epsB = 0.0;
62.         }
63.     }
64.     //mt: A of i,j is a i - j. i is row. j is column.
65.     // ends up being mt[row - column + NUM_ROWS]
66.
67.     sync_device(&group);
68.     // Perform the recursion
69.     for (int iteration = 1; iteration < numRows; iteration++) {
70.         // Calculate local epsilon values for each batch
71.         for (int batch = 0; batch < BATCHES_PER_PARTITION; batch++) {
72.             epsF_local[LOCAL_B] = 0.0;
73.             epsB_local[LOCAL_B] = 0.0;
74.             epsX_local[LOCAL_B] = 0.0;
75.             // First step of dot products:
76.             for (int index = id; index < iteration; index += gsl_inc) {
77.                 epsF_local[LOCAL_B] += mT[iteration - 1 - index + NUM_ROWS] *
    vects[batch].vF[index];
78.                 epsB_local[LOCAL_B] += mT[NUM_ROWS - index - 2] *
    vects[batch].vB[index];
79.                 epsX_local[LOCAL_B] += mT[iteration - 1 - index + NUM_ROWS] *
    vects[batch].vX[index];
80.             }
81.         }
82.         sync_device(&group);
83.         for (int batch = 0; batch < BATCHES_PER_PARTITION; batch++) {
84.             // Perform reductions
85.             epsF_local[LOCAL_B] =
    BlockReduce(temp_storageEpsF).Sum(epsF_local[LOCAL_B]); // THIS CAN PROBABLY BE
    OPTIMIZED
86.             epsB_local[LOCAL_B] =
    BlockReduce(temp_storageEpsB).Sum(epsB_local[LOCAL_B]);
87.             epsX_local[LOCAL_B] =
    BlockReduce(temp_storageEpsX).Sum(epsX_local[LOCAL_B]);
88.             if (block_id == 0) { //Only thread 0 gets the result. Send to other
    threads.
89.                 atomicAdd(&vects[batch].epsF, epsF_local[LOCAL_B]);
90.                 atomicAdd(&vects[batch].epsB, epsB_local[LOCAL_B]);
91.                 atomicAdd(&vects[batch].epsX, epsX_local[LOCAL_B]);
92.             }
93.         }
94.         sync_device(&group);
95.         for (int batch = 0; batch < BATCHES_PER_PARTITION; batch++) {
96.             epsF_local[LOCAL_B] = vects[batch].epsF;
97.             epsB_local[LOCAL_B] = vects[batch].epsB;
98.             epsX_local[LOCAL_B] = vects[batch].epsX;
99.
100.              for (int index = id; index < NUM_ROWS; index += gsl_inc)  {
101.                 vects[batch].vF_Prev[index] = vects[batch].vF[index];
102.                 vects[batch].vB_Prev[index] = vects[batch].vB[index];
103.             }
104.         }
105.         sync_device(&group);
```

```
106.          for (int batch = 0; batch < BATCHES_PER_PARTITION; batch++) {
107.              scalingFctr[LOCAL_B] = 1.0 / (1.0 - epsF_local[LOCAL_B] *
    epsB_local[LOCAL_B]);
108.
109.              // Calculate new forward and backwards vectors
110.              for (int index = id; index <= iteration; index += blockDim.x *
    gridDim.x)  {
111.                  if (index == 0) {
112.                      vects[batch].vF[index] = scalingFctr[LOCAL_B] *
    vects[batch].vF[index];
113.                      vects[batch].vB[index] = -(epsB_local[LOCAL_B] *
    scalingFctr[LOCAL_B]) * vects[batch].vF_Prev[index];
114.                  } else {
115.                      vects[batch].vF[index] = (scalingFctr[LOCAL_B] *
    vects[batch].vF[index]) - ((epsF_local[LOCAL_B] * scalingFctr[LOCAL_B]) *
    vects[batch].vB_Prev[index - 1]);
116.                      vects[batch].vB[index] = (scalingFctr[LOCAL_B] *
    vects[batch].vB_Prev[index - 1]) - ((epsB_local[LOCAL_B] * scalingFctr[LOCAL_B]) *
    vects[batch].vF_Prev[index]);
117.                  }
118.                  vects[batch].vX[index] = vects[batch].vX[index] + ((vY[batch*NUM_ROWS
    + iteration] - epsX_local[LOCAL_B]) * vects[batch].vB[index]);
119.              }
120.          }
121.          sync_device(&group); // was device
122.          for (int batch = 0; batch < BATCHES_PER_PARTITION; batch++) {
123.              if (id == 0) {
124.                  vects[batch].epsF = 0.0;
125.                  vects[batch].epsX = 0.0;
126.                  vects[batch].epsB = 0.0;
127.              }
128.          }
129.      }
130.      // Write shared memory vX to output vX
131.      for (int batch = 0; batch < BATCHES_PER_PARTITION; batch++) {
132.          for (int index = id; index < NUM_ROWS; index += blockDim.x * gridDim.x)  {
133.              vX_out[batch*NUM_ROWS + index] = vects[batch].vX[index];
134.          }
135.      }
136.      sync_device(&group);
137.  }
138.
139.  void levRecursion(DATA_TYPE *mT, DATA_TYPE *vY, DATA_TYPE *vX_out, Vects* vects, int
    numRows) {
140.      // mT holds the time series data of the Toeplitz matrix
141.      // vY is the RHS. Different batches are contigous in memory.
142.      // vX_out will store the resulting output. Resulting batches will be contigous.
143.      // vects is a pointer to an array of global memory needed for each batch
144.      // numRows is the number of rows in the Toeplitz matrix
145.
146.      dim3 dimBlock(BLOCK_SIZE, 1, 1);
147.      dim3 dimGrid(NUM_BLOCKS, 1, 1);
148.
149.      // Create a stream for each kernel
150.      cudaStream_t stream[NUM_PARTITIONS];
151.      for (int i = 0; i < NUM_PARTITIONS; ++i) {
152.          cudaStreamCreate(&stream[i]);
153.      }
154.
```

```
155.      cudaDeviceSynchronize(); // TODO: Create streams ahead of time. Remove
    synchronize
156.      for (int i = 0; i < NUM_PARTITIONS; i++) {
157.          Vects* vects_ptr = vects + i*BATCHES_PER_PARTITION;
158.          DATA_TYPE* vY_ptr = vY + i*BATCHES_PER_PARTITION*NUM_ROWS;
159.          DATA_TYPE* vX_out_ptr = vX_out + i*BATCHES_PER_PARTITION*NUM_ROWS;
160.          void* kernelArgs[] = {
161.              (void*) &mT, // Remains constant for all batches
162.              (void*) &vY_ptr,
163.              (void*) &vX_out_ptr,
164.              (void*) &vects_ptr,
165.              (void*) &numRows
166.          };
167.          cudaLaunchCooperativeKernel((void *)levRecursionKernel, dimGrid,
168.                                      dimBlock, kernelArgs, NULL, stream[i]);
169.      }
170.  }
```

Lev_Batched Driver

```
1.  #ifndef NUM_ROWS
2.      #define NUM_ROWS 1024
3.  #endif
4.  #define BLOCK_SIZE 256
5.  #define NUM_BLOCKS ((NUM_ROWS/2 + BLOCK_SIZE - 1)/BLOCK_SIZE)
6.  #ifndef NUM_BATCHES
7.      #define NUM_BATCHES 128
8.  #endif
9.  #define BATCHES_PER_PARTITION 4
10. #define NUM_PARTITIONS ((NUM_BATCHES + BATCHES_PER_PARTITION -
    1)/BATCHES_PER_PARTITION)
11. #define DATA_TYPE double
12. #define NUM_RUNS 1
13.
14. #include <cooperative_groups.h>
15. #include "kernel.cu"
16. #include <iostream>
17. #include <fstream>
18. #include <cstdlib>
19. #include <iomanip>
20.
21. #include <cuda_runtime.h>
22. #include <helper_cuda.h>
23.
24. #include "/home/uahclsc0014/assignment1/project/Timer/include/Timer.hpp"
25.
26. namespace cg = cooperative_groups;
27.
28. float randomFloat()
29. {
30.     return (float)(rand()) / (float)(RAND_MAX);
31. }
32.
33. void fill_array(DATA_TYPE* arr, std::string filename, int size);
34.
```

```
35. int main(int argc, char **argv) {
36.     DATA_TYPE ac[NUM_ROWS]; // 1st column of Toeplitz matrix
37.     DATA_TYPE ar[NUM_ROWS]; // 1st row of Toeplitz matrix
38.     DATA_TYPE vX[NUM_BATCHES*NUM_ROWS]; // Result placed here
39.     DATA_TYPE vY[NUM_BATCHES*NUM_ROWS]; // RHS
40.
41.     DATA_TYPE* mt_d;
42.     DATA_TYPE* vX_d;
43.     DATA_TYPE* vY_d;
44.     Vects* vects_d;
45.
46.     Timer timer1;
47.
48.     fill_array(ac, "/home/uahclsc0014/lev/metrics/ac.dat", NUM_ROWS);
49.     fill_array(ar, "/home/uahclsc0014/lev/metrics/ar.dat", NUM_ROWS);
50.     fill_array(vY, "/home/uahclsc0014/lev/metrics/b.dat", NUM_BATCHES*NUM_ROWS);
51.
52.     // Convert toeplitz matrix into time-series like format
53.     DATA_TYPE mt_time[2*NUM_ROWS-1];
54.     for (int i = 0; i < NUM_ROWS; i++) {
55.         mt_time[i] = ar[NUM_ROWS-i-1];
56.         mt_time[NUM_ROWS- 1 + i] = ac[i];
57.     }
58.
59.     // Copy mt over
60.     checkCudaErrors(cudaMalloc(&mt_d, (2*NUM_ROWS-1)*sizeof(DATA_TYPE)));
61.     checkCudaErrors(cudaMemcpy(mt_d, mt_time, (2*NUM_ROWS-1)*sizeof(DATA_TYPE),
    cudaMemcpyHostToDevice));
62.
63.     // Copy vY over
64.     checkCudaErrors(cudaMalloc(&vY_d, NUM_BATCHES*NUM_ROWS*sizeof(DATA_TYPE)));
65.     checkCudaErrors(cudaMemcpy(vY_d, vY, NUM_BATCHES*NUM_ROWS*sizeof(DATA_TYPE),
    cudaMemcpyHostToDevice));
66.     // Allocate vX
67.     checkCudaErrors(cudaMalloc(&vX_d, NUM_BATCHES*NUM_ROWS*sizeof(DATA_TYPE)));
68.     // Allocate global memory for forward and backward vectors
69.     checkCudaErrors(cudaMalloc(&vects_d, NUM_BATCHES*sizeof(Vects)));
70.
71.     // Call the kernel.
72.     float time_avg = 0.0;
73.     for (int cnt = 0; cnt < NUM_RUNS; cnt++) {
74.         Timer my_timer;
75.         my_timer.start();
76.         levRecursion(mt_d, vY_d, vX_d, vects_d, NUM_ROWS);
77.         checkCudaErrors(cudaGetLastError());
78.         my_timer.stop();
79.         cudaDeviceSynchronize();
80.         time_avg += my_timer.elapsedTime_ms() / (double) NUM_RUNS;
81.     }
82.
83.     std::cout << NUM_ROWS << "," << time_avg << std::endl;
84.
85.     checkCudaErrors(cudaMemcpy(vX, vX_d, NUM_BATCHES*NUM_ROWS*sizeof(DATA_TYPE),
    cudaMemcpyDeviceToHost));
86.
87.     #ifdef WRITE_CSV
88.     std::ofstream outfile;
89.     outfile.open("lev_n" + std::to_string(NUM_ROWS) + ".csv",std::ofstream::out);
90.     for (int batch = 0; batch < NUM_BATCHES; batch++) {
```

```
91.         for (int i = 0; i < NUM_ROWS; i++) {
92.             outfile << std::fixed << std::setprecision(16) << vX[batch*NUM_ROWS + i]
    << ",";
93.         }
94.         outfile << "\n";
95.     }
96.     outfile.close();
97.     #endif
98. }
99.
100. void fill_array(DATA_TYPE* arr, std::string filename, int length) {
101.     std::ifstream input_file_y;
102.     DATA_TYPE temp;
103.     int i = 0;
104.     input_file_y.open(filename);
105.     if (input_file_y.is_open()) {
106.         while (i < length && input_file_y >> temp)
107.         {
108.             arr[i] = (DATA_TYPE) temp;
109.             i++;
110.         }
111.     } else {
112.         std::cout << "Unable to open " << filename << std::endl;
113.     }
114.     // If input file is too small, fill it up with other values.
115.     int size = i;
116.     while (i < NUM_ROWS) {
117.         arr[i] = arr[i%size];
118.         i++;
119.     }
120. }
```

Appendix 2: Batched PCG Code

CirculantSolver.cu

```
1.  #include "pcg.hpp"
2.
3.  #include "Kernels.hpp"
4.
5.  #include <cufft.h>
6.  #include <cublas_v2.h>
7.
8.  #include <cuda_runtime.h>
9.  #include <helper_cuda.h>
10.
11. #include <iostream>
12.
13. #include "cufft_errors.cpp"
14.
15.
16. class CirculantSolver::Implementation {
17.     public:
18.     Implementation(int num_rows, int batch_size) {
19.         this->num_rows = num_rows;
20.         this->batch_size = batch_size;
21.
22.         // cufft decleration
23.         cufftSafeCall(cufftPlan1d(&this->fft_plan, num_rows, MY_FFT_TYPE, 1));
24.         cufftSafeCall(cufftPlan1d(&this->fft_batched_plan, num_rows, MY_FFT_TYPE,
    batch_size));
25.
26.
27.         // Cublas handle
28.         cublasCreate(&handle);
29.
30.         // Allocate global memory needed for intermediate operations
31.         checkCudaErrors(cudaMalloc(&complex_c, num_rows*sizeof(COMPLEX_TYPE)));
32.         checkCudaErrors(cudaMalloc(&complex_b,
    batch_size*num_rows*sizeof(COMPLEX_TYPE)));
33.     }
34.
35.     ~Implementation() {
36.         // Deallocate memory
37.         cufftDestroy(fft_plan);
38.         cufftDestroy(fft_batched_plan);
39.         checkCudaErrors(cudaFree(complex_c));
40.         checkCudaErrors(cudaFree(complex_b));
41.
42.         // Free cublas handle
43.         cublasDestroy(handle);
44.
45.     }
46.
47.     // Solving Ax = b where A is circulant and b is a vector
48.     //  - ar holds a row of the circulant matrix with which we wish to solve
49.     //  - b_batch holds a set of vectors of length n, we'll solve the matrix
50.     //  - numberOfRHS specifies how many vectors there are
51.     void solve(
52.         REAL_TYPE * ar, // device memory
53.         REAL_TYPE * b_batch, // device memory
```

```
54.         REAL_TYPE * x, // result
55.         cudaStream_t* streams
56.     ) {
57.         // Copy input reals into complex array
58.         real_to_complex(ar, complex_c, this->num_rows);
59.
60.         // do an ifft (scale it)
61.         // e=length(c)*ifft(c);%eigenvalue of circulant matrix.
62.         cufftSafeCall(call_cufftExecC2C(fft_plan, complex_c, complex_c,
    CUFFT_INVERSE)); // CUFFT_INVERSE
63.
64.         // do another ifft
65.         // Convert b to complex
66.         real_to_complex(b_batch, complex_b, batch_size*num_rows);
67.
68.         cufftSafeCall(call_cufftExecC2C(fft_batched_plan, complex_b, complex_b,
    CUFFT_INVERSE)); // CUFFT_INVERSE
69.
70.         // do an element wise division for each batch
71.         for (int b = 0; b < batch_size; b++) {
72.             element_wise_division_complex(complex_b + b*num_rows, complex_c,
    complex_b + b*num_rows, this->num_rows, streams[b]);
73.         }
74.         cudaDeviceSynchronize();
75.
76.         // take the fft of the result
77.         cufftSafeCall(call_cufftExecC2C(fft_batched_plan, complex_b, complex_b,
    CUFFT_FORWARD)); // CUFFT_FORWARD
78.
79.         // return the real part
80.         complex_to_real(complex_b, x, batch_size * num_rows);
81.
82.         // Scale to adjust for fft's not being normalized. :-)
83.         REAL_TYPE alpha = 0.1;
84.         call_cublas_scale(handle, batch_size * num_rows, &alpha, x, 1);
85.     }
86.     private:
87.
88.     int num_rows;
89.     int batch_size;
90.     // FFT plans
91.     cufftHandle fft_plan;
92.     cufftHandle fft_batched_plan;
93.
94.     // Cublas stuff
95.     cublasHandle_t handle;
96.     // Device memory
97.     COMPLEX_TYPE* complex_c;
98.     COMPLEX_TYPE* complex_b;
99.
100. };
101.
102.
103. CirculantSolver::CirculantSolver(
104.     int num_rows,
105.     int batch_size
106. ) : implementation_(std::make_shared<Implementation>(num_rows, batch_size)) {}
107.
108. void CirculantSolver::operator() (
```

```
    REAL_TYPE * ac,
    REAL_TYPE * b_batch, // device memory
    REAL_TYPE * x,
    cudaStream_t* streams
) {
    implementation_->solve(ac, b_batch, x, streams);
}
```

Kernels.cu

```
#ifndef __MY_KERNELS
#define __MY_KERNELS

#include "Kernels.hpp"
#include <iostream>

__global__ void element_wise_division_kernel_real(REAL_TYPE* v1, REAL_TYPE* v2, REAL_TYPE* out, int size) {

    unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

    for (int index = id; index < size; index += blockDim.x * gridDim.x) {
        out[index] = v1[index] / v2[index];
    }
}

__global__ void element_wise_division_kernel_complex(COMPLEX_TYPE* v1, COMPLEX_TYPE* v2, COMPLEX_TYPE* out, int size) {

    unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

    for (int index = id; index < size; index += blockDim.x * gridDim.x) {
        auto a = v1[index].x;
        auto b = v1[index].y;
        auto c = v2[index].x;
        auto d = v2[index].y;
        // Find (a+bi)/(c+di)
        auto real = (a*c + b*d)/(c*c + d*d);
        auto img = (b*c - a*d)/(c*c + d*d);
        out[index].x = real;// = v1[index] / v2[index];
        out[index].y = img;
    }
}

__global__ void element_wise_mul_kernel_complex(COMPLEX_TYPE* v1, COMPLEX_TYPE* v2, COMPLEX_TYPE* out, int size) {

    unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

    for (int index = id; index < size; index += blockDim.x * gridDim.x) {
        auto a = v1[index].x;
        auto b = v1[index].y;
        auto c = v2[index].x;
        auto d = v2[index].y;
        // Find (a+bi)/(c+di)
        auto real = a*c - b*d;
        auto img = b*c + a*d;
```

```
45.         out[index].x = real;// = v1[index] / v2[index];
46.         out[index].y = img;
47.     }
48. }
49.
50.
51. __global__ void compex_to_real_kernel(COMPLEX_TYPE* v_in, REAL_TYPE* v_out, int size)
    {
52.     unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
53.
54.     for (int index = id; index < size; index += blockDim.x * gridDim.x) {
55.         v_out[index] = v_in[index].x;
56.     }
57. }
58.
59.
60. __global__ void real_to_complex_kernel(REAL_TYPE* v_in, COMPLEX_TYPE* v_out, int size)
    {
61.     unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
62.
63.     for (int index = id; index < size; index += blockDim.x * gridDim.x) {
64.         v_out[index].x = v_in[index];
65.         v_out[index].y = (REAL_TYPE) 0;
66.     }
67. }
68.
69. __global__ void precondition_kernel(REAL_TYPE* ac,REAL_TYPE* ar,REAL_TYPE* out, int
    size) {
70.     unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
71.     for (int index = id; index < size; index += blockDim.x * gridDim.x)  {
72.         if (index == 0) {
73.             out[index] = ac[0];
74.         }  else {// In matlab, index rangess from 2:n. Here, it ranges from 1:(n-1)
75.             out[index] = ((size-(index+1)+1)*ac[index]+(index)*ar[size-index])/size;
76.         }
77.     } // c(i)= ((n-i+1)*ac(i)+(i-1)*ar(n-i+2))/n;
78. }
79.
80. __global__ void batched_axpy_kernel(REAL_TYPE* x, REAL_TYPE* alpha, REAL_TYPE* y, int
    size, int batch_size) {
81.     unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
82.     REAL_TYPE* alpha_ptr = alpha;
83.
84.     int len = batch_size*size;
85.     int gsl_inc = blockDim.x * gridDim.x;
86.
87.     for (int index = id; index < len; index += gsl_inc) {
88.         y[index] = alpha_ptr[index/size] * x[index] + y[index];
89.     }
90. }
91.
92. __global__ void negate_kernel(REAL_TYPE* in, REAL_TYPE *out, int size) {
93.     unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
94.
95.     for (int index = id; index < size; index += blockDim.x * gridDim.x) {
96.         out[index] = -1.0 * in[index];
97.     }
98. }
99.
```

```
100.
101.
102.  void precondition(REAL_TYPE* ac,REAL_TYPE* ar,REAL_TYPE* out, int size) {
103.      int blockSize = 512; // Number of threads in a block
104.      int gridSize; // number of blocks
105.
106.      //Round up according to array size
107.      gridSize = (size + blockSize - 1) / blockSize;
108.
109.      // call the kernel
110.      precondition_kernel<<<gridSize, blockSize>>> (ac, ar, out, size);
111.
112.      checkCudaErrors(cudaGetLastError());
113.  }
114.
115.  void element_wise_division_complex(COMPLEX_TYPE* v1, COMPLEX_TYPE* v2, COMPLEX_TYPE*
    out, int size, cudaStream_t stream) {
116.      int blockSize = 512; // Number of threads in a block
117.      int gridSize; // number of blocks
118.
119.      //Round up according to array size
120.      gridSize = (size + blockSize - 1) / blockSize;
121.
122.      // call the kernel
123.      element_wise_division_kernel_complex<<<gridSize, blockSize, 0 ,stream>>> (v1, v2,
    out, size);
124.
125.      checkCudaErrors(cudaGetLastError());
126.  }
127.
128.  void element_wise_division_real(REAL_TYPE* v1, REAL_TYPE* v2, REAL_TYPE* out, int
    size, cudaStream_t stream) {
129.      int blockSize = 512; // Number of threads in a block
130.      int gridSize; // number of blocks
131.
132.      //Round up according to array size
133.      gridSize = (size + blockSize - 1) / blockSize;
134.
135.      // call the kernel
136.      element_wise_division_kernel_real<<<gridSize, blockSize, 0 ,stream>>> (v1, v2,
    out, size);
137.
138.      checkCudaErrors(cudaGetLastError());
139.  }
140.
141.  void element_wise_mul_complex(COMPLEX_TYPE* v1, COMPLEX_TYPE* v2, COMPLEX_TYPE* out,
    int size, cudaStream_t stream) {
142.      int blockSize = 512; // Number of threads in a block
143.      int gridSize; // number of blocks
144.
145.      //Round up according to array size
146.      gridSize = (size + blockSize - 1) / blockSize;
147.
148.      // call the kernel
149.      element_wise_mul_kernel_complex<<<gridSize, blockSize,0 , stream>>> (v1, v2, out,
    size);
150.
151.      checkCudaErrors(cudaGetLastError());
152.  }
```

```
153.
154.
155.  void real_to_complex(REAL_TYPE* v_in, COMPLEX_TYPE* v_out, int size, cudaStream_t
    stream) {
156.      int blockSize = 512; // Number of threads in a block
157.      int gridSize; // number of blocks
158.
159.      //Round up according to array size
160.      gridSize = (size + blockSize - 1) / blockSize;
161.
162.      // call the kernel
163.      real_to_complex_kernel<<<gridSize, blockSize, 0, stream>>> (v_in, v_out, size);
164.
165.      checkCudaErrors(cudaGetLastError());
166.  }
167.
168.
169.  void complex_to_real(COMPLEX_TYPE* v_in, REAL_TYPE* v_out, int size, cudaStream_t
    stream) {
170.      int blockSize = 512; // Number of threads in a block
171.      int gridSize; // number of blocks
172.
173.      //Round up according to array size
174.      gridSize = (size + blockSize - 1) / blockSize;
175.
176.      // call the kernel
177.      compex_to_real_kernel<<<gridSize, blockSize, 0, stream>>> (v_in, v_out, size);
178.
179.      checkCudaErrors(cudaGetLastError());
180.  }
181.
182.  void negate(REAL_TYPE* in, REAL_TYPE* out, int size) {
183.      int blockSize = 512;
184.      int gridSize = (size + blockSize - 1) / blockSize;
185.
186.      negate_kernel<<<gridSize, blockSize>>> (in, out, size);
187.
188.      checkCudaErrors(cudaGetLastError());
189.  }
190.
191.  void print_reals(REAL_TYPE* ptr_d, int size) {
192.      cudaDeviceSynchronize();
193.      REAL_TYPE* tmp = new REAL_TYPE[size];
194.      checkCudaErrors(cudaMemcpy(tmp, ptr_d, size*sizeof(REAL_TYPE),
    cudaMemcpyDeviceToHost));
195.
196.      for (int i = 0; i < size; i++) {
197.          std::cout << tmp[i] << ",";
198.      }
199.      std::cout << "\n";
200.
201.      delete[] tmp;
202.      checkCudaErrors(cudaGetLastError());
203.  }
204.
205.  void print_complex(COMPLEX_TYPE* ptr_d, int size) {
206.      cudaDeviceSynchronize();
207.      COMPLEX_TYPE* tmp = new COMPLEX_TYPE[size];
```

```
208.      checkCudaErrors(cudaMemcpy(tmp, ptr_d, size*sizeof(COMPLEX_TYPE),
    cudaMemcpyDeviceToHost));
209.
210.      for (int i = 0; i < size; i++) {
211.          std::cout << tmp[i].x << " + " << tmp[i].y << "i ,";
212.      }
213.      std::cout << "\n";
214.
215.      delete[] tmp;
216.      checkCudaErrors(cudaGetLastError());
217.  }
218.
219.  __global__ void reverse_array_kernel(REAL_TYPE* in, REAL_TYPE* out, int size) {
220.      unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
221.      REAL_TYPE temp;
222.      for (int index = id; index < (size/2); index += blockDim.x * gridDim.x)  {
223.          temp = in[index];
224.          out[index] = in[size-index-1];
225.          out[size-index-1] = temp;
226.      }
227.  }
228.
229.  void reverse_array(REAL_TYPE* in, REAL_TYPE* out, int size) {
230.      int blockSize = 64; // Number of threads in a block
231.      int gridSize; // number of blocks
232.
233.      //Round up according to array size
234.      gridSize = (size + blockSize - 1) / blockSize;
235.
236.      // call the kernel
237.      reverse_array_kernel<<<gridSize, blockSize>>> (in, out, size);
238.
239.      checkCudaErrors(cudaGetLastError());
240.  }
241.
242.  void batched_axpy(REAL_TYPE* x, REAL_TYPE* alpha, REAL_TYPE* y, int size, int
    batch_size) {
243.      int blockSize = 512;
244.      int gridSize = (batch_size*size + blockSize - 1) / blockSize;
245.
246.      batched_axpy_kernel<<<gridSize, blockSize>>> (x, alpha, y, size, batch_size);
247.
248.      checkCudaErrors(cudaGetLastError());
249.  }
250.
251.  void call_cublas_scale(cublasHandle_t handle, int num_rows, float* alpha, float* x,
    int inc) {
252.      cublasSscal(handle, num_rows, alpha, x, 1);
253.  }
254.
255.  void call_cublas_scale(cublasHandle_t handle, int num_rows, double* alpha, double* x,
    int inc) {
256.      cublasDscal(handle, num_rows, alpha, x, 1);
257.  }
258.
259.  void call_cublas_nrm2(cublasHandle_t handle, int num_rows, float* r0_d, int
    inc,float* norm) {
260.      cublasSnrm2(handle, num_rows, r0_d, 1, norm);
261.  }
```

```
262.
263.  void call_cublas_nrm2(cublasHandle_t handle, int num_rows, double* r0_d, int
    inc,double* norm) {
264.      cublasDnrm2(handle, num_rows, r0_d, inc, norm);
265.  }
266.  //cublasSdot(handle, num_rows, y0_d, 1, r0_d, 1, &v1_numerator);
267.  void call_cublas_dot(cublasHandle_t handle, int num_rows, float* arr1, int inc1,
    float* arr2, int inc2, float* out) {
268.      cublasSdot(handle, num_rows, arr1, 1, arr2, 1, out);
269.  }
270.
271.  void call_cublas_dot(cublasHandle_t handle, int num_rows, double* arr1, int inc1,
    double* arr2, int inc2, double* out) {
272.      cublasDdot(handle, num_rows, arr1, 1, arr2, 1, out);
273.  }
274.
275.  cublasStatus_t call_cublas_axpy(cublasHandle_t handle, int n, const float *alpha,
    const float *x, int incx, float *y, int incy) {
276.      return cublasSaxpy(handle, n, alpha, x, incx, y, incy);
277.  }
278.
279.  cublasStatus_t call_cublas_axpy(cublasHandle_t handle, int n, const double *alpha,
    const double *x, int incx, double *y, int incy) {
280.      return cublasDaxpy(handle, n, alpha, x, incx, y, incy);
281.  }
282.
283.  cufftResult call_cufftExecC2C(cufftHandle plan, cufftComplex *idata, cufftComplex
    *odata, int direction) {
284.      return cufftExecC2C(plan, idata, odata, direction);
285.  }
286.
287.  cufftResult call_cufftExecC2C(cufftHandle plan, cufftDoubleComplex *idata,
    cufftDoubleComplex *odata, int direction) {
288.      return cufftExecZ2Z(plan, idata, odata, direction);
289.  }
290.
291.
292.  #endif
```

Toeplitz Multiplier

```
1.  #include "pcg.hpp"
2.
3.  #include "Kernels.hpp"
4.
5.  #include <cufft.h>
6.  #include <cublas_v2.h>
7.
8.  #include <cuda_runtime.h>
9.  #include <helper_cuda.h>
10.
11. #include <iostream>
12.
13. #include "cufft_errors.cpp"
14.
```

```
15.
16. class ToeplitzMultiplier::Implementation {
17.     public:
18.     Implementation(int num_rows, int batch_size) {
19.         this->num_rows = num_rows;
20.         this->batch_size = batch_size;
21.         checkCudaErrors(cudaMalloc(&t_d, 2*num_rows*sizeof(COMPLEX_TYPE)));
22.         checkCudaErrors(cudaMalloc(&xx_d,
    batch_size*2*num_rows*sizeof(COMPLEX_TYPE)));
23.
24.         checkCudaErrors(cudaMalloc(&ar_flipped_d, num_rows*sizeof(COMPLEX_TYPE)));
25.
26.         // Used for fft of ac and ar
27.         cufftSafeCall(cufftPlan1d(&fft_plan, num_rows*2, MY_FFT_TYPE, 1));
28.
29.         // Create another plan for batched use of x
30.         cufftSafeCall(cufftPlan1d(&fft_batched_plan, num_rows*2, MY_FFT_TYPE,
    batch_size));
31.
32.         cublasCreate(&handle);
33.     }
34.
35.     ~Implementation() {
36.         cufftDestroy(fft_plan);
37.         cufftDestroy(fft_batched_plan);
38.         checkCudaErrors(cudaFree(t_d));
39.         checkCudaErrors(cudaFree(xx_d));
40.         checkCudaErrors(cudaFree(ar_flipped_d));
41.     }
42.
43.     void solve(
44.         REAL_TYPE* ac,
45.         REAL_TYPE* ar,
46.         REAL_TYPE* x,
47.         REAL_TYPE* out,
48.         cudaStream_t* streams // TODO: Change to std::vector. Use round robin
49.     ) {
50.         // n=length(x);
51.         // t=[ac;0;ar(end:-1:2)']; //Construct t vector
52.         real_to_complex(ac, t_d, num_rows);
53.         reverse_array(ar, ar_flipped_d, num_rows);
54.         real_to_complex(ar_flipped_d, t_d+num_rows+1, num_rows-1);
55.         cudaMemset((REAL_TYPE*) (t_d+num_rows), (REAL_TYPE) 0, sizeof(COMPLEX_TYPE));
    // Set middle element to complex 0
56.
57.         // xx=[x;zeros(n,1)];%pad zero to x.
58.         for (int b = 0; b < batch_size; b++) {
59.             real_to_complex(x+ b*num_rows, xx_d + b*2*num_rows, num_rows,
    streams[b]);
60.         }
61.         for (int b = 0; b < batch_size; b++) {
62.             cudaMemsetAsync((REAL_TYPE*) (xx_d + num_rows + b*2*num_rows),
    (REAL_TYPE) 0, num_rows*sizeof(COMPLEX_TYPE), streams[b]); // Complex type is float2.
    Set both to float 2.
63.         }
64.         cudaDeviceSynchronize(); // Sync streams together. //TODO: Loop over streams
    and call cuda stream sync
65.
66.         // e=2*n*ifft(t);%eigenvalue of circulant matrix.
```

```
67.         cufftSafeCall(call_cufftExecC2C(fft_plan, t_d, t_d, CUFFT_INVERSE)); //
    CUFFT_INVERSE
68.
69.         // yy=fft(e.*ifft(xx));
70.         cufftSafeCall(call_cufftExecC2C(fft_batched_plan, xx_d, xx_d,
    CUFFT_INVERSE)); // CUFFT_INVERSE
71.
72.         for (int b = 0; b < batch_size; b++) {
73.             element_wise_mul_complex(xx_d + b*2*num_rows, t_d, xx_d + b*2*num_rows,
    num_rows*2, streams[b]);
74.         }
75.         cudaDeviceSynchronize(); // Reunify streams
76.
77.         cufftSafeCall(call_cufftExecC2C(fft_batched_plan, xx_d, xx_d,
    CUFFT_FORWARD));
78.
79.         // y=real(yy(1:n));
80.         for (int b = 0; b < batch_size; b++) {
81.             complex_to_real(xx_d + b*2*num_rows, out+b*num_rows, num_rows,
    streams[b]);
82.         }
83.         cudaDeviceSynchronize();
84.         // Scale it
85.         REAL_TYPE alpha = 1.0/(2.0*num_rows); // cuffts are not normalized, so the
    result from an fft is scaled by n
86.         call_cublas_scale(handle, batch_size*num_rows, &alpha, out, 1);
87.     }
88.     private:
89.
90.     int num_rows;
91.     int batch_size;
92.
93.     REAL_TYPE* ar_flipped_d;
94.     COMPLEX_TYPE* xx_d;
95.     COMPLEX_TYPE* t_d;
96.     COMPLEX_TYPE* t_d_ifft;
97.     COMPLEX_TYPE* xx_d_ifft;
98.
99.     cufftHandle fft_plan;
100.     cufftHandle fft_batched_plan;
101.
102.     cublasHandle_t handle;
103.
104. };
105.
106.
107. ToeplitzMultiplier::ToeplitzMultiplier(
108.     int num_rows,
109.     int batch_size
110. ) : implementation_(std::make_shared<Implementation>(num_rows, batch_size)) {}
111.
112. void ToeplitzMultiplier::operator() (
113.     REAL_TYPE* ac,
114.     REAL_TYPE* ar,
115.     REAL_TYPE* x,
116.     REAL_TYPE* out,
117.     cudaStream_t* streams
118. ) {
119.     implementation_->solve(ac, ar, x, out, streams);
```

```
120. }
```

Toeplitz Solver

```
1.  #include "pcg.hpp"
2.
3.  #include "Kernels.hpp"
4.
5.  #include <cufft.h>
6.  #include <cublas_v2.h>
7.
8.  #include <cuda_runtime.h>
9.  #include <helper_cuda.h>
10.
11. #include <iostream>
12.
13. #include "cufft_errors.cpp"
14.
15. class ToeplitzSolver::Implementation {
16. public:
17.   Implementation(int num_rows, int batch_size) : circinv_(num_rows, batch_size),
    toepmultip_(num_rows, batch_size) {
18.     this->num_rows = num_rows;
19.     this->batch_size = batch_size;
20.     checkCudaErrors(cudaMalloc(&c_d, num_rows*sizeof(REAL_TYPE)));
21.     checkCudaErrors(cudaMalloc(&x0_d, batch_size*num_rows*sizeof(REAL_TYPE)));
22.     checkCudaErrors(cudaMalloc(&r0_d, batch_size*num_rows*sizeof(REAL_TYPE)));
23.     checkCudaErrors(cudaMalloc(&r1_d, batch_size*num_rows*sizeof(REAL_TYPE)));
24.     checkCudaErrors(cudaMalloc(&p1_d, batch_size*num_rows*sizeof(REAL_TYPE)));
25.     checkCudaErrors(cudaMalloc(&z_d, batch_size*num_rows*sizeof(REAL_TYPE)));
26.     checkCudaErrors(cudaMalloc(&y0_d, batch_size*num_rows*sizeof(REAL_TYPE)));
27.     checkCudaErrors(cudaMalloc(&y1_d, batch_size*num_rows*sizeof(REAL_TYPE)));
28.
29.     checkCudaErrors(cudaMalloc(&v1_numerator_d, batch_size*sizeof(REAL_TYPE)));
30.     checkCudaErrors(cudaMalloc(&v1_d, batch_size*sizeof(REAL_TYPE)));
31.     checkCudaErrors(cudaMalloc(&mu_numerator_d, batch_size*sizeof(REAL_TYPE)));
32.     checkCudaErrors(cudaMalloc(&mu_d, batch_size*sizeof(REAL_TYPE)));
33.
34.     cublasCreate(&norm_handle);
35.     handles = new cublasHandle_t[batch_size];
36.     for (int i = 0; i < batch_size; i++) {
37.       cublasCreate(handles + i);
38.       cublasSetPointerMode(handles[i], CUBLAS_POINTER_MODE_DEVICE);
39.     }
40.
41.     // plan the ffts if you need them, create the circulant solver, etc.
42.   }
43.   ~Implementation() {
44.     checkCudaErrors(cudaFree(c_d));
45.     checkCudaErrors(cudaFree(x0_d));
46.     checkCudaErrors(cudaFree(r0_d));
47.     checkCudaErrors(cudaFree(r1_d));
48.     checkCudaErrors(cudaFree(p1_d));
49.     checkCudaErrors(cudaFree(z_d));
50.     checkCudaErrors(cudaFree(y0_d));
51.     checkCudaErrors(cudaFree(y1_d));
52.
```

```
53.     checkCudaErrors(cudaFree(v1_numerator_d));
54.     checkCudaErrors(cudaFree(v1_d));
55.     checkCudaErrors(cudaFree(mu_numerator_d));
56.     checkCudaErrors(cudaFree(mu_d));
57.
58.     for (int i = 0; i < batch_size; i++) {
59.       cublasDestroy(handles[i]);
60.     }
61.     delete[] handles;
62.   }
63.   int solve(
64.     REAL_TYPE * ar,
65.     REAL_TYPE * ac,
66.     REAL_TYPE * b_batch,
67.     cudaStream_t* streams,
68.     double tolerance
69.   ) {
70.       // Set cublas handles to different streams
71.       for (int i = 0; i < batch_size; i++) {
72.         cublasSetStream(handles[i],streams[i]);
73.       }
74.       int k = 0;
75.       precondition(ac, ar, c_d, num_rows);
76.       //x0=0;
77.       cudaMemset(x0_d, 0, batch_size*num_rows*sizeof(REAL_TYPE));
78.       //r0=b;
79.       checkCudaErrors(cudaMemcpy(r0_d, b_batch, batch_size*num_rows*sizeof(REAL_TYPE),
    cudaMemcpyDeviceToDevice));
80.
81.       //p1=circinv(c,b);% p1=M\b
82.       circinv_(c_d, b_batch, p1_d, streams);
83.
84.       //y0=p1;
85.       checkCudaErrors(cudaMemcpy(y0_d, p1_d,
    batch_size*num_rows*sizeof(REAL_TYPE),cudaMemcpyDeviceToDevice));
86.
87.       // while cublas_norm(residual_0) > tol
88.       while (k < 10) { // Cap on number of iterations
89.         REAL_TYPE norm;
90.         call_cublas_nrm2(norm_handle, batch_size*num_rows, r0_d, 1, &norm);
91.         if (norm < tolerance) {
92.           break; // Stop iterating if norm is low enough
93.         }
94.       //    z=toepmultip(ac,ar,p1);% z=A*p1
95.       toepmultip_(ac, ar, p1_d, z_d, streams);
96.
97.
98.       REAL_TYPE alpha = 1.0;
99.       REAL_TYPE beta = 0.0;
100.        // This Dgemm is used to take a batched dot product of
101.        // y0_d and r0_d.
102.        cublasDgemmStridedBatched(norm_handle,
103.                                  CUBLAS_OP_N, CUBLAS_OP_N,
104.                                  1, 1, num_rows, /*m, n, k*/
105.                                  &alpha,
106.                                  y0_d, 1, 1, /*A, lda, stride A*/
107.                                  r0_d, num_rows, 1, /*B, ldb, stride B*/
108.                                  &beta,
109.                                  v1_numerator_d, 1, 1, /*C, ldc, stride C*/
```

```
110.                                    batch_size);
111.        // Batched dot products
112.        cublasDgemmStridedBatched(norm_handle,
113.                                    CUBLAS_OP_N, CUBLAS_OP_N,
114.                                    1, 1, num_rows, /*m, n, k*/
115.                                    &alpha,
116.                                    p1_d, 1, 1, /*A, lda, stride A*/
117.                                    z_d, num_rows, 1, /*B, ldb, stride B*/
118.                                    &beta,
119.                                    v1_d, 1, 1, /*C, ldc, stride C*/
120.                                    batch_size);
121.
122.        cudaDeviceSynchronize();
123.        element_wise_division_real(v1_numerator_d, v1_d, v1_d, batch_size, streams[0]);
124.        cudaDeviceSynchronize();
125.
126.        //    x0=v1*p1 + x0;
127.        batched_axpy(p1_d, v1_d, x0_d, num_rows, batch_size); // Custom kernel
128.
129.        negate(v1_d, v1_d, batch_size);
130.        //r1=r0;
131.        checkCudaErrors(cudaMemcpy(r1_d, r0_d, batch_size*num_rows*sizeof(REAL_TYPE),
    cudaMemcpyDeviceToDevice));
132.
133.        batched_axpy(z_d, v1_d, r1_d, num_rows, batch_size);
134.
135.        cudaDeviceSynchronize(); // All streams need to finish
136.
137.        //    y1=circinv(c,r1);% y1=M\r1
138.        circinv_(c_d, r1_d, y1_d, streams);
139.
140.        // mu=(y1'*r1)/(y0'*r0);
141.        alpha = 1.0;
142.        beta = 0.0;
143.
144.        // Batched dot product of y1_d and r1_d
145.        cublasDgemmStridedBatched(norm_handle,
146.                                    CUBLAS_OP_N, CUBLAS_OP_N,
147.                                    1, 1, num_rows, /*m, n, k*/
148.                                    &alpha,
149.                                    y1_d, 1, 1, /*A, lda, stride A*/
150.                                    r1_d, num_rows, 1, /*B, ldb, stride B*/
151.                                    &beta,
152.                                    mu_numerator_d, 1, 1, /*C, ldc, stride C*/
153.                                    batch_size);
154.        // Batched dot product of y0_d and r0_d
155.        cublasDgemmStridedBatched(norm_handle,
156.                                    CUBLAS_OP_N, CUBLAS_OP_N,
157.                                    1, 1, num_rows, /*m, n, k*/
158.                                    &alpha,
159.                                    y0_d, 1, 1, /*A, lda, stride A*/
160.                                    r0_d, num_rows, 1, /*B, ldb, stride B*/
161.                                    &beta,
162.                                    mu_d, 1, 1, /*C, ldc, stride C*/
163.                                    batch_size);
164.        cudaDeviceSynchronize(); // All streams need to finish
165.        element_wise_division_real(mu_numerator_d, mu_d, mu_d, batch_size, streams[0]);
166.
167.        //    y0=y1;
```

```
168.        checkCudaErrors(cudaMemcpy(y0_d, y1_d, batch_size*num_rows*sizeof(REAL_TYPE),
    cudaMemcpyDeviceToDevice));
169.
170.        // p1= mu*p1 + y1;
171.        batched_axpy(p1_d, mu_d, y1_d, num_rows, batch_size);
172.
173.        cudaDeviceSynchronize(); // All streams need to finish
174.
175.        checkCudaErrors(cudaMemcpy(p1_d, y1_d, batch_size*num_rows*sizeof(REAL_TYPE),
    cudaMemcpyDeviceToDevice));
176.
177.        //    r0=r1;
178.        checkCudaErrors(cudaMemcpy(r0_d, r1_d, batch_size*num_rows*sizeof(REAL_TYPE),
    cudaMemcpyDeviceToDevice));
179.
180.          k += 1;
181.        }
182.
183.        // Copy to output
184.        //    b_batch = x0;
185.        checkCudaErrors(cudaMemcpy(b_batch, x0_d,
    batch_size*num_rows*sizeof(REAL_TYPE), cudaMemcpyDeviceToDevice));
186.
187.        // Return the number of iterations
188.        return k;
189.    }
190.  private:
191.    int num_rows;
192.    int batch_size;
193.
194.    REAL_TYPE* c_d;
195.    REAL_TYPE* x0_d; // Will be Output
196.    REAL_TYPE* r0_d;
197.    REAL_TYPE* r1_d;
198.    REAL_TYPE* p1_d;
199.    REAL_TYPE* z_d;
200.    REAL_TYPE* y0_d;
201.    REAL_TYPE* y1_d;
202.
203.    REAL_TYPE* v1_numerator_d;
204.    REAL_TYPE* v1_d;
205.    REAL_TYPE* mu_numerator_d;
206.    REAL_TYPE* mu_d;
207.
208.    // Cublas stuff
209.    cublasHandle_t norm_handle;
210.    cublasHandle_t* handles;
211.
212.    // Solvers
213.    ToeplitzMultiplier toepmultip_;
214.    CirculantSolver circinv_;
215.
216.  };
217.
218.
219.  ToeplitzSolver::ToeplitzSolver (
220.    int num_rows,
221.    int batch_size
222.  ): implementation_(std::make_shared<Implementation>(num_rows, batch_size)) {}
```

```
223.
224.  int ToeplitzSolver::operator() (
225.    REAL_TYPE * ar,
226.    REAL_TYPE * ac,
227.    REAL_TYPE * b_batch,
228.    cudaStream_t* streams,
229.    double tolerance
230.  ) {
231.    return implementation_->solve(ar, ac, b_batch, streams, tolerance);
232.  }
```