UCLA
UCLA Electronic Theses and Dissertations

Title
The Accelerated Cauchy Estimator: A Paradigm for Parallelization

Permalink
https://escholarship.org/uc/item/8gg64001

Author
Sanpakit, Chirawat Chriss

Publication Date
2020

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

The Accelerated Cauchy Estimator:

A Paradigm for Parallelization

A thesis submitted in partial satisfaction

of the requirements for the degree Master of Science

in Aerospace Engineering

by

Chirawat Chriss Sanpakit

2020

© Copyright by

Chirawat Chriss Sanpakit

2020

ABSTRACT OF THE THESIS

The Accelerated Cauchy Estimator:

A Paradigm for Parallelization

by

Chirawat Chriss Sanpakit

Master of Science in Aerospace Engineering

University of California, Los Angeles, 2020

Professor Jason L. Speyer, Chair

This thesis presents a paradigm for accelerating the Cauchy Estimator using high performance computing for discrete linear systems with scalar process and measurement noises. The Cauchy Estimator is developed in such a way that is conducive to parallelization and thus presents an opportunity to leverage the graphics processing unit (GPU) to bring the algorithm closer to a real time environment. This work compartmentalizes the estimator into smaller routines and presents algorithms that were developed to accelerate the computations across several independent terms. First, the serial implementation is given for comparison and to help build intuition. Then, the GPU implementation is given along with the parameter sending and receiving requirements for each function. Simulations demonstrate that parallelization does eventually outpace the serial implementation along with an almost linear scaling in time after each subsequent measurement.

The thesis of Chirawat Chriss Sanpakit is approved.

Robert M'Closkey

Tsu-Chin Tsao

Jason L. Speyer, Committee Chair

University of California, Los Angeles

2020

DEDICATION

To Jill

For your tireless support, love, and inspiration. To whom this thesis, and everything it means to me, would not be possible without.

To my Beloved Family

Thank you for your endless love and sacrifice over the years. I will always be grateful and remember all you have given me.

To the Memory of my Late Father

While I wish you could see where I am today, I hope this thesis helps complete one of the many dreams you sought out for me.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION – THE CAUCHY ESTIMATOR.......................... 1

SECTION 1.1: INTRODUCTION 1

SECTION 1.2: THE CAUCHY ESTIMATOR 2

SECTION 1.3: PARALLEL PROGRAMMING APPLIED TO STATE ESTIMATION 3

CHAPTER 2: PARALLELIZATION AND THE GPU 5

SECTION 2.1: GPU PROGRAMMING: THE NEED FOR SPEED 5

SECTION 2.2: CUDA AND THE GPU KERNEL.. 6

SECTION 2.3: CUPY, PYTHONIC PARALLELIZATION... 7

SECTION 2.4: DATA TRANSFERS 8

CHAPTER 3: THE ACCELERATED CAUCHY ESTIMATOR ALGORITHM..... 9

SECTION 3.1: PROBLEM FORMULATION 9

SECTION 3.2: THE CAUCHY ESTIMATOR, OVERVIEW.. 11

SECTION 3.3: TIME PROPAGATION 12

3.3.1: Introduction and Serial Implementation... 12

3.3.2: Changes Made to the Measurement Update.. 14

3.3.3: The Parallelized Time Propagation Algorithm.. 15

3.3.4: "Time Propagation" of the Parent Terms... 16

SECTION 3.4: MEASUREMENT UPDATE 17

3.4.1: The Original Measurement Update 17

3.4.2: Changes Made to the Measurement Update.. 19

3.4.3: The CPU Implementation of the Measurement Update 20

3.4.4: The GPU Implementation of the Measurement Update *24*

SECTION 3.5: EVALUATING PARAMETERS FOR THE UCPDF 30

3.5.1: Calculating the G's and Y_{ei} *30*

3.5.2: CPU Implementation of Calculating G's and Y_{ei} *31*

3.5.3: Sending Requirements for Calculating G's and Y_{ei} *35*

3.5.4: The GPU Implementation of Calculating G's and Y_{ei} *35*

SECTION 3.6: COALIGNMENT INTRODUCTION AND SERIAL IMPLEMENTATION 39

3.6.1: Coalignment Introduction *39*

3.6.2: Coalignment Serial Implementation *41*

SECTION 3.7: COALIGNMENT PARALLEL IMPLEMENTATION 39

3.7.1: Inputs into Parallel Coalignment *43*

3.7.2: Coalignment Serial Implementation *44*

3.7.3: Parallel Coalignment – Removing Repeated Hyperplanes *46*

3.7.4: Parallel Coalignment – Parallel Reduction of Coefficients *47*

3.7.5: Parallel Coalignment – Global Versus Shared Memory *48*

SECTION 3.8: S-EXPANSION AND LEAST NORMS - INTRODUCTION 52

3.8.1: Kernelizing the S-Expansion - Motivation *52*

3.8.2: S-Expansion Nomenclature and Incremental Enumeration *53*

3.8.3: S-Expansion Preliminaries *54*

3.8.4: CPU Implementation of S-Expansion Algorithm and Least Norms Solution ... *54*

3.8.5: Challenges in the GPU Implementation of the S-Expansion Algorithm *55*

SECTION 3.9: PARALLEL S-EXPANSION AND LEAST NORMS – IMPLEMENTATION 56
3.9.1: CPU Refactoring to Enable Contiguous Inputs to the GPU........................... 56
3.9.2: Using CUDA Streams for Asynchronous Data Transfers 58
3.9.3: The S-Expansion Kernel – Introduction and Thread-Blocks Chosen 60
3.9.4: The S-Expansion Kernel – Computing the Pseudoinverse 68

CHAPTER 4: EXPERIMENTS AND RESULTS FROM PARALLELIZATION .. 69

SECTION 4.1: SIMULATION SETUP AND TESTING 69
SECTION 4.2: TIME PROPAGATION 70
SECTION 4.3: MEASUREMENT UPDATE 72
SECTION 4.4: COMPUTING G, Y_{EI} 73
SECTION 4.5: TERM-COALIGNMENT 74
SECTION 4.6: PARALLEL S-EXPANSION AND COMPUTING THE LEAST NORMS 77

CHAPTER 5: DISCUSSION AND CONCLUSION.. 78

SECTION 5.1: TIME PROPAGATION 78
SECTION 5.2: MEASUREMENT UPDATE 78
SECTION 5.3: COMPUTING G, Y_{EI} 79
SECTION 5.4: TERM-COALIGNMENT 80
SECTION 5.5: PARALLEL S-EXPANSION AND COMPUTING THE LEAST NORMS 81
SECTION 5.6: IMPLEMENTING THE GPU WITH A SLIDING WINDOW 82
SECTION 5.7: CONCLUSION 84

ACKNOWLEDGEMENTS

When I started my graduate studies, I thought for sure I wanted to pursue control theory. That is, until I took my first course in Probability and Stochastic Processes in Dynamic Systems with Professor Speyer. I found his overwhelming passion and love for this field infectious, ultimately propelling my journey into Estimation Theory.

First and foremost, I absolutely must thank Nat Snyder for the countless hours he put into supporting me and my thesis. His patience, willingness to work with me, and many insightful discussions has made this work possible. Secondly, I must thank Professor Speyer for his guidance, trust in me, encouragement, and for his love of teaching. I am eternally grateful for the opportunity to work on the Cauchy Estimator and for my newfound love of Navigation Engineering. To say that I am constantly inspired and humbled by you both would not begin to capture my sentiments.

Of course, I must thank my thesis committee: Professor M'Closkey and Professor Tsao. The time you both dedicate towards cultivating your students is what has made my time at UCLA incredibly gratifying and why I have found the campus to be such an intellectually vibrant environment (I most certainly will not forget my first midterm in Linear Dynamic Systems).

Finally, where would I be without the network of people who have supported me throughout my education, trials of my life, and through this pandemic: Xavier Hernandez, Akila Ganlath, Audrey Der, Timothy Lam, Steven Luna, John-Pierre Sawma, Luis Miranda, Gustavo Perez, Josef Bustamante, and Benjamin Sommerkorn. You have all truly made my life full.

And finally, this list would be incomplete without my undergraduate advisor, Dr. Marko Princevac, who has inspired my love of the sciences and engineering.

CHAPTER 1

Introduction – The Cauchy Estimator

1.1. Introduction

In 1960, Rudolph E. Kalman published a seminal paper on a recursive solution to the discrete time linear filtering problem [12, 20]. The Discrete-Time Kalman filter consists of essentially four equations that describe a predictor-corrector algorithm [9]. Moreover, two of these equations, the apriori and posteriori covariance are not functions of the measurement and in theory, can be computed offline. On the other hand, the posteriori state estimate is a linear, affine function of the measurement [17]. These results coalesce to a series of matrix operations that modern day computers can calculate with remarkable efficiency. The Cauchy Estimator developed by Moshe Idan and Jason Speyer do not share the same computational luxuries because the number of operations required along with the large dimensionality of the problems grow with each subsequent measurement. Practically speaking, each measurement adds a large amount of data that the algorithm must operate on. This thesis presents a paradigm for applying parallel computing to accelerate the estimator and offers a path forward towards bringing the Cauchy Estimator to a real-time implementable environment.

Chapter 1 begins with a select set of literature review relevant to the Cauchy Estimator and general applications of parallel programming to estimation algorithms. Chapter two then moves into some relevant background for GPU programming before delineating the algorithm in Chapter 3. Chapter 3 focuses heavily on the exact design and considerations for the parallel algorithms implemented for the Cauchy Estimator before concluding with the results and discussion in Chapter 4 and 5.

1.2. The Cauchy Estimator

Many engineering, economic, and telecommunications experience an underlying random process that exhibits volatility that Gaussian distributions do not properly capture [23]. Rather than light-tailed Gaussian distributions, heavy-tailed distributions like the Cauchy distributions better represent these random processes. Examples of engineering applications include radar, sonar sensors, and turbulent air [29]. This motivates the need for a filtering technique for linear dynamic systems with Cauchy distributions.

Results from the Cauchy Estimation paradigm designed by M. Idan and J. Speyer demonstrates the ability of the filter to respond to such impulsive and heavy-tailed stochasticity. This filter has seen applications in attitude estimation using a star tracker corrupted by an impulsive radiation background [38] or drag estimation for low earth orbiters [19]. Here, this sequential nonlinear estimator demonstrates some quintessential property that differs from the Kalman Filter. Unfortunately, the Cauchy processes do not have a well-defined first moment and has infinite second moment [21]. Nevertheless, it does have an analytical form for the conditional probability density function, given the measurement history and has been shown to have a finite and data dependent conditional first- and second moment [26,27]. In other words, the state estimates and error variance are functions of the measurement history and must be computed online. This also translates to surprising results, most notably state estimates that adapt to significant process noise as well as error variances that grow with that same measurement [16].

The estimator itself is generated by using the characteristic function of the unnormalized cpdf, yielding a closed-form expression for the minimum variance estimate of the states and conditional variance for the n-state problem [28]. Also, unlike the Kalman Filter, the conditional pdf for the Cauchy Filter exhibits a multimodal, non-symmetric process [26]. By contrast, the

Kalman filter has a very rigid, unimodal, and symmetric structure for its probability density function [17].

1.3. Parallel Programming Applied to State Estimation

CPU's have begun to stagnate in performance gains over the years due to the increased power consumption that comes with higher clock rates. The trend of exponential CPU growth begun to slow down, and serial computing has thus reached its limits. However, as CPU's have begun to reach their performance ceilings, multicore processing and GPU's began to grow exponentially due to their ability to conduct massively parallelized operations [6]. Broadly speaking, the GPU is designed for a class of problems that exhibit large computational requirements, substantial parallelism, and prioritization of throughput over latency [15].

Parallel programming has made its way to some classes of state estimation algorithms. Hendeby and Karlsson describe exploiting parallelism during the resampling phase to accelerate computation using the GPU [11]. Similarly, Lopez, Zhang, and Mok leverages the CUDA programming model to accelerate a particle filter for real-time application to a manufacturing process [10]. Karimipour and Dinavahi describe a parallel dynamic state estimator that is based on the Extended Kalman Filter that also leverages the GPU to compute matrix-matrix products and underlying linear algebraic operations for Linear Solvers over large data-sets [14]. Generally speaking, sequential filters like the discrete-time Kalman Filter is not well suited to GPU programming. However, when massive independent operations can occur simultaneously, such as in the case of a particle filter, the estimator can benefit significantly from parallelism.

Related disciplines include the fields of Computer Vision and Robotics, which have also benefitted from the parallelism offered by the GPU, resulting in massive efficiency gains in

decision-making or real-time image processing. Examples include Brian Fulkerson’s and Stefano Soatto’s Quick Shift Algorithm that uses the GPU to accelerate image segmentation [5], classification to identify walls, ceilings, and so on for robotics applications [13], and parallel algorithms to accelerate collision queries [18]. Similarly, massive commercialization efforts have been made by NVIDIA [32] and Tesla [3] to democratize autonomous driving by accelerating processing via the GPU. So overtime, the hardware necessary to produce accelerated algorithms has become more widely available.

The above set of applications share a similar challenge to the Cauchy Estimator. There exist many parallelizable operations that must occur to bring them into a real-time environment. Numerous examples of accelerating computation in time sensitive applications like autonomous driving or collision avoidance suggests that the future of programming has moved towards leveraging parallelism to improve the viability of data-dependent algorithms.

CHAPTER 2

Parallelization and the Graphics Processing Unit (GPU)

2.1. GPU Programming: The Need for Speed

The graphics processing unit (GPU) provides consumers with the primary means of visualizing anything on computers. But as it turns out, outside of rendering the graphics for video games or YouTube videos, the GPU also enables a readily accessible way to accelerate programs through massive parallel computing. This democratization has gotten to the point that an average user can go on Amazon and purchase all the parts needed, for under $2000, to build a computer that can perform tens of teraflops of calculations in a matter of seconds [7].

The GPU has many, many more "simplistic" cores than a CPU. While the CPU optimizes latency (i.e, the time it takes to compute a single computation), the GPU's primary advantage comes from its huge gains in throughput, or the number of possible simultaneous instructions. To drive the point home, a GPU is analogues to a wide city road that several slower cars can drive over at once whereas the CPU is like the hyperloop, designed to transport an individual to their destination much quicker [7].

However, like a city road, a GPU can run into traffic jams or dependencies on other vehicles. For example, a family might go on a road trip and take several cars. One car might drive a little slower, thereby arriving at the destination after everyone else and delaying the vacation. On top of that, how do you divide the workload or exploit parallelism? Ultimately, these questions translate into small design decisions or "best practices" when designing parallel algorithms for the Cauchy estimator. This is further delineated in the following sections but for now, this chapter

serves to acknowledge the areas where a GPU should be deployed, and a sample of the complications that arises when designing an efficiently parallelized algorithm.

2.2. CUDA and the GPU Kernel

CUDA is a framework developed by NVIDIA for general-purpose GPU (GPGPU) programming and what's used in this work to accelerate the Cauchy Estimator. It is a mature, free and open source (FOS) platform that provides access to a set of GPU accelerated mathematical libraries. Further, there are additional readily available Python libraries that interface with CUDA, like CuPy or PyCuda, that provide function calls to predeveloped kernels. In other words, CUDA extends on traditional coding by giving aspiring GPU programmers a means to interface with underlying parallelization tools.

This segues into the primary means by which a CPU practically interfaces with the GPU. A GPU *Kernel* is code that executes on the GPU. It looks virtually identical to serial CPU code but with the caveat that each *thread* runs it *several times over.* In this example adapted from Cupy documentation [4], a kernel computes the squared difference of $f(x,y) = (x-y)^2$, given x and y.

```
 squared_diff = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

At the beginning of a kernel launch, the programmer decides how many threads to deploy. If 512 threads are chosen, then the above code executes 512 times.

For the purpose of this thesis, there are two more primary levels of concurrency, *blocks* and *grids*. Each *block* contains up to 1024 *threads* and each *grid* can contain 2^{31} - 1 in the x-direction and 65535 *blocks* in the y and z-direction [30]. Blocks and threads also have directionality, affording programmers some intuition when deciding how to parallelize code. Thus, in any given program, there are several blocks running at the same time and within each block, several threads that perform simple operations, resulting in millions of operations happening concurrently at any given time. Note, that while it is theoretically possible to run 2^{31}-1 x 65535 x 65535 blocks per grid (all with 1024 threads), saturation often occurs in other forms, such as GPU memory. Figure 1 visualizes the primary levels of concurrency. Finally, *streams* allow course-grain concurrency that performs asynchronous data transfers to the GPU as well as kernel launches [7].

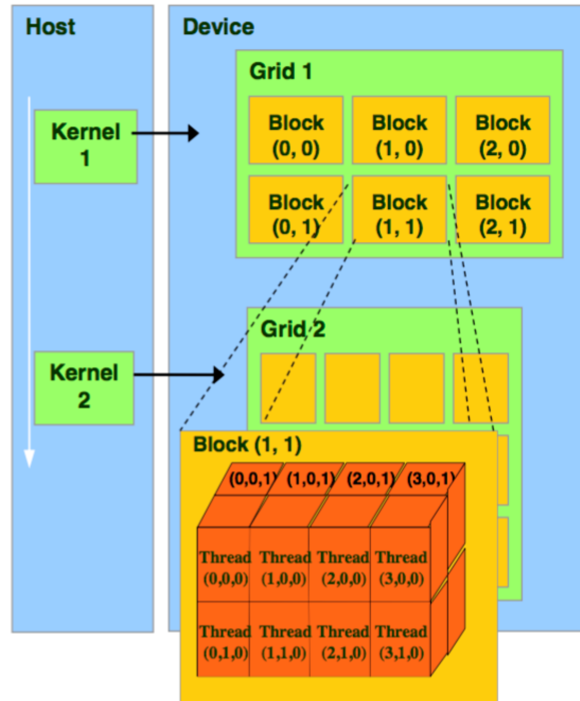

Figure 1: Host refers to the CPU while Device refers to the GPU [Image Source: NYU]

2.3. CuPy, Pythonic Parallelization

With the basic definitions of the GPU and how it operates differently from the CPU aside, this thesis now introduces CuPy as the primary development tool used to interface with the GPU and CUDA. CuPy is a python library that allows programmers to abstract away algorithms into function calls rather than develop custom GPU kernels for every parallel task, as in the element-wise multiplication example above. CuPy was chosen to facilitate future adoption of the algorithm

and because of its natural transition from interpreted languages like, MATLAB. Thus, instead of writing kernels and worrying about how to allocate threads, blocks or grids (and many other programmatic challenges like array broadcasting), the focus shifts to developing the higher-level algorithm and working on exploiting parallelism. For example, if a dot product is required, typically a GPU programmer would allocate several threads per element in a vector to perform a multiplication and perhaps a block if the vector has more than 1024 elements. Then, they would aggregate the results into a sum and assign it into some new variable. Instead of unnecessarily worrying about these lower level complications, CuPy allows programmers to simply type:

```
z = x*y or z = cupy.dot(x,y)
```

Nevertheless, some portions of the Cauchy Estimator cannot rely on CuPy functions alone and requires kernelization. These kernels are detailed in their corresponding sections.

2.4. Data Transfers

A data transfer occurs when the CPU needs to send data (say a variable, a matrix, or a measurement in the case of an estimator) to the GPU. Data transfer represents some of the largest bottlenecks when designing highly parallelized algorithms and should be minimized whenever possible [15]. In fact, entire operations on the GPU can be shorter when compared to a single data transfer. A prudent design will perform a transfer once, do several computations, then transfer the data back to the CPU as needed. In that vein, the Accelerated Cauchy Estimator (ACE) performs several algorithmic steps (time propagation, measurement update, etc.) on the GPU, thereby taking full advantage of the capability of the GPU before necessitating any additional data transfers.

Chapter 3

The Accelerated Cauchy Estimator Algorithm

3.1. Problem Formulation

Consider the single-input-single-output multivariable linear system

$$x_{k+1} = \Phi x_k + \Gamma \omega_k, \quad z_k = H x_k + v_k$$

with state vector $x_k \in \mathbb{R}^n$, scalar measurements z_k , and known matrices $\Phi \in \mathbb{R}^{nxn}$, $\Gamma \in \mathbb{R}^{nx1}$ and $H \in \mathbb{R}^{1xn}$. The noise inputs are assumed to be Cauchy distributed random variables where w_k is zero median and has a scaling parameter $\beta > 0$. Likewise, the noise input to the sensor measurement, v_k, has a Cauchy probability density function (pdf) with a median of zero and scaling parameter $\gamma > 0$.

The goal is to develop an algorithm that can compute the minimum variance estimate of x_k given the measurement history, $y_k = \{ z_1, z_2, \ldots z_k\}$ in real-time. This thesis summarizes the results from "Multivariate Cauchy Estimator with Scalar Measurement and Process Noise", by Idan and Speyer [28], and presents additional insight conducive to parallel programming. The method proposed in this thesis entails sending the minimum number of parameters required at every algorithmic step to minimize the overhead associated with GPU transfers. Further, this work takes a best practice approach to perform as many parallel computations as possible to maximize the Cauchy estimator's throughput in the process, resulting in a paradigm for high performance computing.

Some Nomenclature for the Cauchy Estimator

- let **d** be the state dimension.
- let $\boldsymbol{l} \in \mathbf{L}$, where **L** is the set of currently generated hyperplane arrangement sizes (i.e: 5, 6, 7, hyperplanes) over all terms.
- let $\boldsymbol{m}$, represent the current hyperplane $\boldsymbol{m} \in [1, 2\ldots, \boldsymbol{l}]$.
- let $\boldsymbol{n_t}$ represent the total number of terms generated at the current step. The number of hyperplanes in all arrangements is given by the set M
- $\mathbf{A_i} \in \mathbb{R}^{l\ x\ d}$ be the matrix which holds term i's hyperplanes, of dimension d.
- $\mathbf{A_{im}} \in \mathbb{R}^{1\ x\ d}$ represents the m-th hyperplane (row of A_i)
- $\mathbf{A_{il}} \in \mathbb{R}^{l\ x\ d}$ represents the i-th term for the $\boldsymbol{l}$**-th** arrangement
- $\mathbf{A_{iml}} \in \mathbb{R}^{1\ x\ d}$ represents the m-th hyperplane for the $\boldsymbol{l}$**-th** arrangement
- $\mathbf{p_i} \in \mathbb{R}^{m}$ represents the coefficient of the ith term's hyperplane arrangement within the exponential of the Cauchy's characteristic function.
- $\mathbf{q_i} \in \mathbb{R}^{m}$ represent the coefficients to the ith term's hyperplane arrangement, within the g-coefficient of the characteristic function

3.2. The Cauchy Estimator, Overview

The Cauchy Estimator currently covers 7 subroutines, shown in figure 2. The new development regarding Reverse Search, Flattening, and Reduction are not covered in this thesis. Instead, this work covers the routines in the first N-State Estimator derived by M. Idan and J. Speyer. That said, some of the subroutines discusses calculating certain parameters that are a direct result of this new development. So, it is important to be generally aware that some of the results are a direct consequence of the time it takes to compute the supporting parameters needed for, for example, Flattening.

This chapter will first begin my discussing the serial implementation of the different subroutines since the "improvements" reported by parallelization are with respect to the provided CPU code. Then, the GPU implementation including any required data transfers, memory management considerations, and other relevant designed decisions are discussed. Finally, there is an intermediate section that occurs right after coalignment called the S-Expansion kernel discussed in section 3.8 (not shown below).

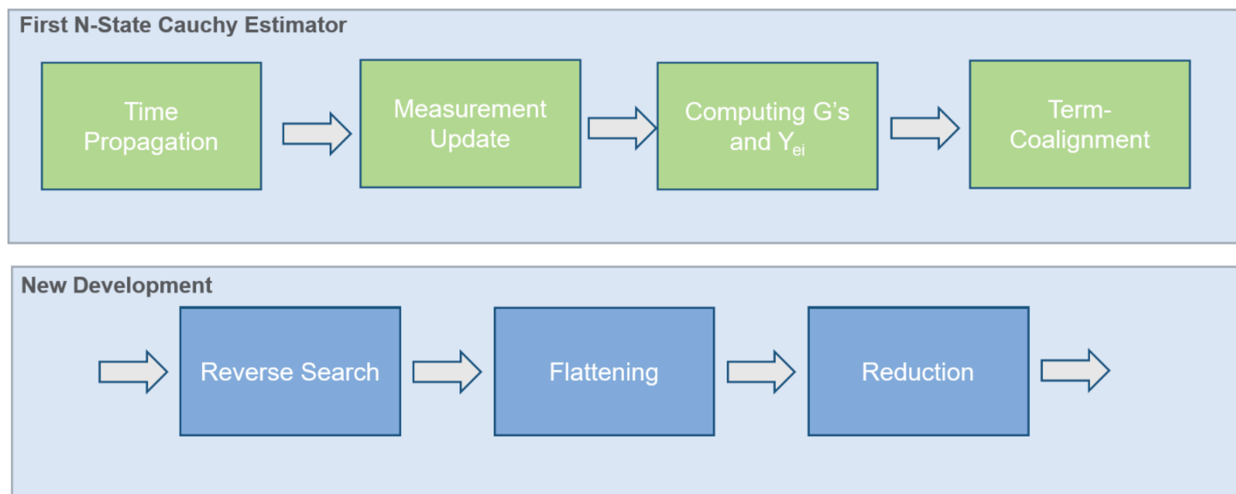

Figure 2: General flow diagram of the Cauchy Estimator and each separate function. The algorithms are compartmentalized into 7 routines and this thesis covers parallelization of the first 4.

3.3. Time Propagation

3.3.1 Introduction and Serial Implementation

Idan and Speyer derived the generalized time propagation characteristic function of the ucpdf of x_{i+1} given the measurement history $y_k = \{z_1\ z_2\ \ldots\ z_k\}$ [28]. The relevant results to parallel programming for Time Propagation are summarized by the following:

First, at any time step the characteristic function of the ucpdf of the state x_k is given by:

$$\bar{\phi}_{X_k|Y_k}(v) = \sum_{i=1}^{n_t^{k|k}} g_i^{k|k}\,(y_{gi}^{k|k}(v))\exp\,(y_{ei}^{k|k}(v)) \tag{3.3.1}$$

where

$$y_{gi}^{k|k}(v) = \sum_{l=1}^{n_{ei}^{k|k}} q_{il}^{k|k} sgn(< a_{il}^{k|k}, v >) \tag{3.4}$$

$$y_{ei}^{k|k}(v) = -\sum_{l=1}^{n_{ei}^{k|k}} p_{im}^{k|k} | < a_{il}^{k|k}, v \rangle | + j < b_i^{k|k}, v > \tag{3.3.2}$$

and where $g_i^{k|k}$ will be defined later in section 3.5. In other words, the current state given the measurement history is a function of several parameters that we compute in the actual estimation algorithm. Also worth noting is that $y_{ei}^{k|k}(v)$ and $y_{gi}^{k|k}(v)$ are term dependent parameters, hence the subscripted i-s. After performing a time propagation, the characteristic function is then given by:

$$\bar{\phi}_{X_k|Y_{k-1}}(v) = \bar{\phi}_{X_k|Y_{k-1}}(v)(\Phi^T v)\ \bar{\phi}_W(\Gamma^T v)$$
$$= \sum_{i=1}^{n_t^{k|k-1}} g_i^{k|k-1}\left(y_{gi}^{k|k-1}(\Phi^T v)\right) \exp\,(y_{ei}^{k|k-1}(\Phi^T v) - \beta| < \Gamma, v > |) \tag{3.3.3}$$

Where the above arguments in the exponent can be redefined as:

$$y_{gi}^{k+1|k}(v) = y_{gi}^{k|k}(\Phi^T v) = \sum_{l=1}^{n_{ei}^{k|k}} q_{il}^{k|k} sgn(< \Phi\, a_{il}^{k|k}, v >) \qquad (3.3.4)$$

$$y_{ei}^{k+1|k}(v) = y_{ei}^{k|k}(\Phi^T v) - \beta| < \Gamma, v > | \qquad (3.3.5)$$

$$= -\sum_{l=1}^{n_{ei}^{k|k}} p_{il}^{k|k} | < \Phi\, a_{il}^{k|k}, v > | - \beta| < \Gamma, v > | + j < b_i^{k|k}, v >$$

Thus, from the algorithmic standpoint the time propagation performs a set of matrix multiplications with Φ to compute the propagated arguments of *g* and y_{ei}. while c, *d*, *q* and *p* remain unchanged. For comparison, the pseudocode for the serialized CPU implementation taking advantage of the Numba library JIT (just-in-time) compiler is given in algorithm 1.

Algorithm 1: Serial CPU Implementation of the Time Propagation

1: #Begin CPU Time Propagation

2: #Inputs: Φ, Γ, *β*, a, b, p

3: For i ∈ [1,…,n_t]:

4: $a^{k+1|k} \leftarrow A\, \Phi^T$, #where $a^{k+1|k} \in \mathbb{R}^{l\ x\ d}$

5: $a^{k+1|k} \leftarrow$ Append Γ^T to $a^{k+1|k}$ to form $[a^{k+1|k}\ ; \Gamma^T]$, # $a^{k+1|k} \in \mathbb{R}^{l+1\ x\ d}$

6: $b^{k+1|k} \leftarrow b\, \Phi^T$, # $b_i^{k+1|k} \in \mathbb{R}^{l+1\ x\ d}$

7: $p^{k+1|k} \leftarrow$ append original ps

8: $p^{k+1|k} \leftarrow$ append β to last column of $p^{k+1|k}$ prop to form $[p^{k+1|k}\ \beta]$, # $p^{k+1|k} \in \mathbb{R}^{l\ x\ d+1}$

9: end for

10: Return $A^{k+1|k}$, $b^{k+1|k}$, $p^{k+1|k}$

12: end CPU time propagation

For the CPU implementation of the time propagation, each parameter's matrix individually multiplies or appends within a loop. For notational convenience, the propagated parameters are denoted with a k+1|k, for the unnormalized characteristic function. A, b and p are the parameters for the unnormalized characteristic function in the previous step while $\Phi \in \mathbb{R}^{nxn}$represents the known, state transition matrix. Finally, $\Gamma \in \mathbb{R}^{nx1}$ and β are known vectors and scaling parameters > 0 for Cauchy distributed noise, respectively. Similarly, v_k, the scalar measurement, has a Cauchy pdf with a scaling parameter $\gamma > 0$. While the number of terms (g's and yei's to compute) have stayed the same at the time propagation step, the number of hyperplanes has increased by one.

3.3.2 Parameter Sending Requirements for the GPU

For the parallel implementation, the time propagation needs to send 3 parameters to the GPU: A, b, and p. Each of these parameters that make up the argument of the exponential and the g's defined in equation 3.3.1 change dynamically as the estimator progresses and thus necessitates reallocation to the GPU at each measurement. It is worth noting that the reason this occurs at this stage of the algorithm is because reduction has not been developed for the GPU. Future work in this area would address this and thus save 3 parameter transfers.

All other parameters, Φ, γ, Γ and β are functions of the linear system dynamics and the known statistics. As a result, these parameters transfer once to the GPU at the start of the algorithm and stay there for the duration of the estimator, saving 3 additional transfers. Outside of these 6 parameters, no additional parameters need to exist on the GPU at this point in the algorithm to successfully perform the time propagation. In the case of a linear time varying system, the dynamics would update on the GPU itself instead of resending the parameters to the device at every time step.

3.3.3 The Parallelized Time Propagation Algorithm

First, note that the time propagation step consists of a weighted sum of parameters independent from one another, resulting in an inherently parallelizable structure. In other words, the matrix Φ^T multiplies to each a_i, forming $a_i^{k+1|k}$, irrespective of the other values $i \exists [1,\ldots,n_t]$. Similarly, all matrix multiplications in $b_i^{k+1|k} = \Phi\, b_i^{k|k}$ can occur simultaneously. Also, note that the parameters $p_i^{k+1|k} = p_i^{k|k}$ remains unchanged, though both a and p append an additional parameter to reflect the additional hyperplane. Finally, implementation-wise, whenever operations between two variables occur, they must exist on the same hardware. For example, CPU variables can only interact with CPU variables while GPU variables can only operate with GPU variables. With that in mind, any variables that exist on the GPU will be denoted with the subscripted $_{gpu}$.

These observations suggest the formulation of the parallelized time propagation algorithm, given in Algorithm 2, with the Python and CuPy implementation presented in the Appendix A. This eliminates the inner for loop that the CPU required to perform the matrix multiplication to each $a_i^{k+1|k}{}_{gpu}$, $b_i^{k+1|k}{}_{gpu}$ and reassignment for $p_i^{k+1|k}{}_{gpu}$. Further, all intermediate operations like the transpose, appending, tiling and so occur in parallel as well. For example, it is worth noting that in this implementation of the time propagation, the appending of β and Γ^T occur in parallel.

The operations happen in parallel over the number of terms while the CPU's for loop handles the different hyperplane arrangements. In other words, the time propagation of the hyperplane arrangement currently happens serially while operations on these hyperplanes occur in parallel. This is the paradigm for ACE for each section of the algorithm, whether it is the time propagation, measurement update, or computing the arguments for the characteristic function.

Algorithm 2: GPU Implementation of the Time Propagation

1: Begin GPU Time Propagation

2: GPU ← a, b, and p to GPU

3: For ∀ Hyperplane Arrangements, #conduct this loop over the CPU

4: **In parallel,** compute $A^{k+1|k}$, $b^{k+1|k}$ and $p^{k+1|k}$ matrix ∀ i ∃ [1,…,n_t]

5: $a^{k+1|k}{}_{gpu} \leftarrow a_{gpu}\ \Phi^{T}{}_{gpu}$

6: $a^{k+1|k}{}_{gpu} \leftarrow$ Assign $\Gamma^{T}{}_{gpu}$ to the *all* row of propagated matrices

7: $b^{k+1|k}{}_{gpu} \leftarrow b_{gpu}, \Phi^{T}{}_{gpu}$

8: $p^{k+1|k}{}_{gpu} \leftarrow$ append original $p^{k+1|k}$

9: $p^{k+1|k}{}_{gpu} \leftarrow$ append β_{gpu} to *all* last row of $p^{k+1|k}$, dimension has grown by 1

10: **return** $a^{k+1|k}{}_{gpu}$, $b^{k+1|k}{}_{gpu}$, $p^{k+1|k}{}_{gpu}$

11: end for

12: end GPU time propagation

3.3.4 "Time Propagation" of the Parent Terms

An intermediate step occurs in between the time propagation and the measurement update. Specifically, the parameters that result from the reverse search, α, Bp, and Ss append to a Python list for future use. These parameters can exist on the GPU by the time this propagation happens, but that requires parallelization of reduction. Otherwise, this requires one transfer of the alphas at this step of the algorithm. For the purposes of this work, this section was not timed and currently exists on the CPU; but note this assignment can occur simultaneously.

3.4. Measurement Update

3.4.1. The Original Measurement Update

After performing the generalized time propagation given in section 3.3, Idan and Speyer then compute the measurement update using $z_{k+1} = Hx_{k+1} + v_{k+1}$ to determine $\bar{\phi}_{X_{k+1}|Y_{k+1}}(v)$, where $y_{k+1} = \{z_1\ z_2\ \ldots\ z_{k+1}\}$ [28]. Again, the relevant results to parallel programming for the measurement update as well as the changes made after implementing the reverse search are summarized here. Like the time propagation, the measurement update is conducive to parallel programming and can be implemented with some alterations.

Beginning with the ucpdf of the state given the measurement history in Idan and Speyer [28]:

$$\bar{\phi}_{X_{k+1}|Y_{k+1}}(v) = \sum_{i=1}^{n_t^{k+1|k+1}} g_i^{k+1|k+1}\left(y_{gi}^{k+1|k+1}(v)\right)\exp\left(y_{ei}^{k+1|k+1}(v)\right) \quad (3.4.1)$$

Where, originally, g_i and y_{ei} were given by:

$$y_{gi}^{k+1|k+1}(v) = \sum_{l=1}^{n_{ei}^{k+1|k+1}} q_{il}^{k+1|k+1} sgn(< a_{il}^{k+1|k+1}, v >) \quad (3.4.2)$$

$$y_{ei}^{k+1|k}(v) = -\sum_{l=1}^{n_{ei}^{k+1|k+1}} p_{il}^{k+1|k+1} | < a_{il}^{k+1|k+1}, v > | + j < b_i^{k+1|k+1}, v > \quad (3.4.3)$$

Further, note that some of these parameters are themselves a function of other variables that must be computed as a part of the measurement update. This nested set of parameters is delineated below as well as summarized in the Table 1. The way these parameters are evaluated during the measurement update has not changed between this algorithm and the one presented in the original paper [28].

Starting from equation 3.4.2, the update at measurement 2 for a and b are defined as:

$$a_{iml} = \mu_{il} - \mu_{im} = \begin{cases} \frac{a_{il}^{2|1}}{Ha_{il}^{2|1}} - \frac{a_{im}^{2|1}}{Ha_{im}^{2|1}}, & m \neq n_{ei}^{2|1} + 1, l \neq n_{ei}^{2|1} + 1 \\ \frac{a_{il}^{2|1}}{Ha_{il}^{2|1}}, & m = n_{ei}^{2|1} + 1 \\ -\frac{a_{im}^{2|1}}{Ha_{im}^{2|1}} & l = n_{ei}^{2|1} + 1 \end{cases}$$

$$b_{im} = \zeta_i \mu_{im} + b_i^{2|1} = \begin{cases} (z_2 - Hb_i^{2|1}) \frac{a_{im}^{2|1}}{Ha_{im}^{2|1}} + b_i^{2|1}, & m \neq n_{ei}^{2|1} \\ b_i^{2|1}, & m = n_{ei}^{2|1} + 1 \end{cases}$$

Further note that the imaginary parameter, ζ , and ρ have been donated as c_i and d_i, respectively. Finally, q_{il} is defined as $\rho \; \forall \; l \nexists$ m.

a_{im}	$[a_i^{k+1\|k}; 0] < [a_i^{k+1\|k}; 0], H > \forall \; l \neq$ m
b_{im}	$(z - H * b_{im}^{k+1\|k}) * \mu_{im} + b_{im}^{k+1\|k}$
p_i	$[p_i^{k+1\|k}; \gamma_i] *< [a_i^{k+1\|k}; 0], H > + \gamma_i \; \forall \; l \neq$ m
ρ	$[p_i^{k+1\|k}; \gamma_i] *< [a_i^{k+1\|k}; 0], H >$
ζ	$z - H * b_i^{k+1\|k}{}_{im}$
c_i	ζ
d_i	ρ_i
q_i	$\rho_i \; \forall \; l \neq$ m

Table 1: The nested set of parameters that are computed during the measurement update are delineated above. Note the column term, l, has been dropped.

Note the indexing scheme and the hyperplane exclusions. The measurement update has an indexing exclusion such that the same hyperplane does not subtract from itself (resulting in a zero during the update). While looping through the current hyperplane arrangement, care needs to be taken so that this does not happen, both for the CPU and the GPU implementation of the algorithm.

3.4.2 Changes Made to the Measurement Update

Neither the argument for the exponential nor the denominator is evaluated for the g's has changed. However, the y_{gi} variables or the numerators is now computed as the dot product between the combinatorial S expansion and the alpha parameters that results from the reverse search and flattening routine, respectively. In fact, the left (positive) side of the numerators for the g's are computed using positive S's – reflecting the positive side of the hyperplanes – while the right (negative) side of the numerator is computed using negative S's, reflecting the negative side of the hyperplanes. This is delineated further in the following section when this thesis discusses parallelization of evaluating the g's and y_{ei}'s

One last important note is that the measurement update function, implementation-wise, focuses on computing the parameters needed to evaluate the g's and the argument for the exponential. The actual computation of the g's and y_{ei} occurs in a different function. This helps readability of the code, especially because the reverse search necessitates a combinatorial S expansion before evaluating the g's and y_{ei}.

3.4.3 The CPU Implementation for the Measurement Update

In essence, the CPU implementation of the measurement update computes the parameters needed to evaluate the g's over each hyperplane (in the current arrangement) and over each term in that hyperplane. This necessitates a double for loop to properly compute the required parameters. That said, some parameters such as ϱ or ζ can efficiently be vectorized by using, for example, numpy's broadcasting feature. This thesis uses the same notion as Idan and Speyer [28] and notes that the A, b, and p within the measurement update is not to be confused with the same variables within the time propagation.

The second essential task for the measurement update is to perform a combinatorial S expansion and generate coefficients needed to compute the g's in the next routine. The simplest implementation involves performing this expansion for each term *per* hyperplane in the current arrangement. Implementation-wise, this is conducted over a double for loop. Algorithm 2 depicts the pseudocode for the CPU implementation of the measurement update. Similar to the time propagation, this section is mainly provided as a comparison to the GPU implementation because "improvements" reported by the GPU are with respect to the provided serial implementation.

Algorithm 2: CPU Implementation of the Measurement Update

1: #Compute over all hyperplane arrangements, over all terms $\boldsymbol{i} \in [\mathbf{1}, \mathbf{2}, \dots, \boldsymbol{n_t}]$

2: # Let m ∈ M, where M is the set of currently generated hyperplane arrangement sizes

3: Input: z, A_i, p_i, b_i, $\boldsymbol{\gamma}$, H, mask #where A_i, p_i, and b_i are from the time propagation

4: Begin CPU Measurement Update

5: gams ← create n_t copies of γ **,** # gams $\in \mathbb{R}^{n_t \, x \, 1}$

6: ϱ ← horizontally stack [p_i , gams] , # $\varrho \in \mathbb{R}^{n_t \, x \, m+1}$

7: zeros ← create 0's $\in \mathbb{R}^{n_t \, x \, 1 \, x \, d}$

8: μ ← Append 0's [A_i, 0_i] # where $i \in [1, 2, \dots, n_t]$, $\mu \in \mathbb{R}^{n_t \, x \, (m+1) \, x \, d}$

9: $\mu_h \leftarrow \sum_{c=1}^{d} \mu_{irc} \circ H_c^T$, #where $H^T \in \mathbb{R}^{1 \, x \, d}$, $r \in [1, 2 \dots m+1]$, and $\mu_h \in \mathbb{R}^{n_t \, x \, (m+1)}$

10: $\rho \leftarrow \varrho \circ |\mu_h|$, #where $\rho \in \mathbb{R}^{n_t \, x \, (m+1)}$

11: $\rho \leftarrow \rho[:, m] + \gamma$, #and note γ is a scalar so this is elementwise addition

12: assert μ_h as 3D such that $\mu_h \in \mathbb{R}^{n_t \, x \, (m+1) \, x \, 1}$

13: $\mu \leftarrow \mu[:, 1: end-1, :] \oslash \mu_h[:, 1: end-1, :]$, #$\mu_h \in \mathbb{R}^{n_t \, x \, (m+1)}$

14: $\zeta \leftarrow z - \sum_{c=1}^{d} (H^T \circ b_i)_c$, #where $\zeta \in \mathbb{R}^{n_t \, x \, 1}$

15: # It is worth noting that these operations are vectorized across the terms, $\mathbf{n_t}$

Algorithm 2: CPU Implementation of the Measurement Update (CONT.)

16: $\dot{n}_t$ ← (m + 1) * n_t **,** #calculate new number of child terms generated during update

17: zero_indices ← [-1 -1 -1 …] ∈ $\mathbb{R}^{\dot{n}_t \, x \, 1}$

18: n = 0, n ∈ $[1, 2, \ldots, \dot{n}_t]$

19: #Begin loops to compute measurement update parameters across hyperplanes

20: For $i \in [1, 2, \ldots, n_t]$:

21: **For** j ∈ $[1, 2, \ldots, m + 1]$:

22: e_{ll} ← [f for f in range(m+1) if f does not equal j], #create iterables

23: a_{im} ← $\mu[i][e_{ll}]$

24: μ_{tmp} ← μ[i][j][:] and **assert** as 1xd vector

25: A_{new}[n] ← $a_{im} - \mu_{tmp}$, #where A_{new}[n] ∈ $\mathbb{R}^{m \, x \, d}$

26: p_{new}[n] ← ρ[i][e_{ll}], #where p_{new}[n] ∈ $\mathbb{R}^{1 \, x \, m}$

27: b_{new}[n] ← ζ[i] * μ[i][j] + b[i], #where b_{new}[n] ∈ $\mathbb{R}^{1 \, x \, d}$

28: c_{new}[n] ← ζ[i], #where c_{new}[n] ∈ $\mathbb{R}^1$

29: d_{new}[n] ← ρ[i][j], #where d_{new}[n] ∈ $\mathbb{R}^1$

30: q_{new}[n] ← ρ[i][e_{ll}], #where d_{new}[n] ∈ $\mathbb{R}^{1xm}$

31: **#Note the double for loop continues onto the next page**

Algorithm 2: CPU Implementation of the Measurement Update (CONT.)

32: **#begin computing flattening terms**

33: $A_h \leftarrow$ <A[i], H>

34: $\tilde{\alpha} \leftarrow$ get_S(sign(A_h), mask) #conduct row-wise sign expansion

35: $\tilde{\alpha}_p$[n] $\leftarrow$ store $\tilde{\alpha}$ for future processing

36: $A_f \leftarrow A[i] \oslash A_h^T$, #row-wise element division $A_f \in \mathbb{R}^{mxd}$

37: **if** j < m:

38: $\bar{A}$ [n] $\leftarrow A_f$ **-** A_f[j]

39: zero_indices[n] = j

40: **else**:

41: $\bar{A}$ [n] = A_f

42: zero_indices[n] = -1

43: n = n + 1

44: **end For**

45 end For

46: return A_{new}, p_{new}, b_{new}, c_{new}, d_{new}, q_{new}, $\bar{A}$, $\tilde{\alpha}_p$, zero_indices

47: end measurement update function

3.4.4 The GPU Implementation of the Measurement Update

For the parallel implementation of the measurement update, the main goal is to eliminate any loops needed to compute the updates over the different terms. Similar to the time propagation, because term for each parameter $\boldsymbol{i} \in [\mathbf{1}, \mathbf{2}, \ldots, \boldsymbol{n_t}]$, is independent from one another, each evaluation of the update can occur simultaneously. Because the number of terms grow exponentially every update along with an increasing number of hyperplanes, this methodology provides the best practice approach to leveraging the GPU. In other words, this eliminates the outer for loop needed to compute the measurement update in the CPU implementation.

The inner for loop that iterates over the current hyperplane unfortunately could not be parallelized at the time of writing, notwithstanding the hyperplanes being independent from one another. Attempting to leverage Python libraries such as multipool results in an initialization error due to the incompatibility of using CUDA in just the parent process. Instead, spawn or forkserver start methods are required to use CUDA in subprocesses [34]. The overhead of spawning children processes and reinitializing these GPU parameters in real-time would likely render any speed benefits obtained during the measurement update moot. However, this represents a section where CUDA streaming could occur instead if the current computer had enough resources to not saturate the hardware. This is further elaborate in the discussion section of the thesis. All that said, the inner for loop is far less computationally expensive than parallelizing over the number of terms because the number of hyperplanes is at most 11 at the 9th measurement.

The indexing scheme to assign the arrays has also changed from the CPU implementation. In the serial implementation of the measurement update, the i_{th} term of the current hyperplane is evaluated, until $\dot{n}_t$ terms are computed. For example, in a (5,3) hyperplane arrangement, the number of old terms is 20 while the number of new terms is 84. The current indexing scheme results in the first 5 indices in A_{new} as being the first updated term for each hyperplane in the current arrangement. Because the GPU implementation parallelizes the term computation, it also has to appropriately reassign the indices. In other words, because the outer for loop restarts at index 6 in this example, then it means assignment has to occur in parallel for terms $i \in [0, 6, 12 \ldots \dot{n}_t - 6]$ in a Python indexing scheme. Note that this parallel assignment is often a biproduct of parallel programming. Lastly, a formula was used to conduct the reassignment using the inner for loop

Let the number of indices to reassign at once be:

num_idx = $\dot{n}_t$ / (m+1)

Next, Let the starting index to assign the first hyperplane's updated term be:

stack_index = range(0, num_idx)*(m+1)

The algorithm then increments with the inner for loop such that each updated term *for each hyperplane* is computed simultaneously and reassigned into their appropriate index.

Take for example:

A_{new}[stack_index + j]. As j increasements to m+1 hyperplane, then stack_index + j indices are accessed and assigned the updated term, where $j \in [0, 1, \ldots$ m] for Python.

The next change made to the measurement update is the parallel computation of the S expansion algorithm. In the original CPU implementation, the S expansion occurs for each hyperplane *per* term. First note that the mask is a constant matrix throughout the entire measurement update. Secondly, each input is a function of the current term of A in the current hyperplane arrangement. Thus, the entire $\tilde{\alpha}_p_{\mathrm{gpu}}$ can be found by running a parallel implementation of the S expansion algorithm for each hyperplane in the current arrangement and make memory copies of the results $\dot{n}_t$ times. Alternatively, the S expansion algorithm can occur *once* over every term and *also* in parallel for each hyperplane in the current arrangement. The current implementation makes use of the former method. This parallel implementation of the S expansion represents one of the key algorithmic contributions to this thesis and is detailed further in section 3.3.8. Along with this change comes the parallelization of the flattening terms as well, like the indexing scheme described in the regular update.

Technically, parallelization can also occur over supporting parameters such as μ (the parameters evaluated outside of the double for loop). For example, concatenation, multiplications, or operations in general occur using parallel methods in CuPy's built-in library. However, because the CPU can efficiently conduct the vectorization over each term, there is not as much of a speed improvement when done over the GPU. That said, it is prudent to conduct operations on the GPU anyway because otherwise, it would necessitate at least 3 more parameter transfers for ζ, μ, and ρ, especially because these are exactly the parameters that are functions of previously allocated variables to the GPU. These results and observations form the basis for the parallelized measurement update, presented in Algorithm 4. Similar to the time propagation, parallelization occurs on the terms whereas the CPU for loop handles the different hyperplane arrangements themselves, i.e a (4,3) arrangement is processed separately from the (5,3) and so on.

Algorithm 4: GPU Implementation of the Measurement Update

1: #Compute over all hyperplane arrangements, over all terms $\boldsymbol{i} \in [\mathbf{1}, \mathbf{2}, \ldots, \boldsymbol{n_t}]$

2: # Let m ∈ M, where M is the set of currently generated hyperplanes

3: Input: z_{cpu}, A_{i_gpu}, p_{i_gpu}, b_{i_gpu}, γ_{gpu}, H_{gpu}, mask

4: Begin GPU Measurement Update

5: $z_{gpu} \leftarrow z_{cpu}$, **#send measurement to GPU**

6: $gams_{gpu} \leftarrow$ create n_t copies of γ_{gpu} **,** # gams $\in \mathbb{R}^{n_t \, x \, 1}$

7: $\varrho_{gpu} \leftarrow$ horizontally stack $[p_{i_gpu}, gams_{gpu}]$, # $\varrho \in \mathbb{R}^{n_t \, x \, m+1}$

8: $zeros_{gpu} \leftarrow$ create 0's $\in \mathbb{R}^{n_t \, x \, 1 \, x \, d}$, #creates variables on device but quite inexpensive

9: $\mu_{gpu} \leftarrow$ Append 0's $[A_{i_gpu}, 0_i]$ # where $i \in [1, 2, \ldots, n_t]$, $\mu \in \mathbb{R}^{n_t \, x \, (m+1) \, x \, d}$

10: $\mu_{h_gpu} \leftarrow \left(\sum_{c=1}^{d} \mu_{irc} \circ H_c^T\right)_{gpu}$, #where $H^T \in \mathbb{R}^{1 \, x \, d}$, $r \in [1, 2 \ldots m+1]$, and $\mu_{h_gpu} \in \mathbb{R}^{n_t \, x \, (m+1)}$

11: $\rho_{gpu} \leftarrow \varrho_{gpu} \circ \left|\mu_{h_gpu}\right|$, #where $\rho_{gpu} \in \mathbb{R}^{n_t \, x \, (m+1)}$

12: $\rho_{gpu} \leftarrow \rho_{gpu}[:, m] + \gamma_{gpu}$, #and note γ_{gpu} is a scalar so this is elementwise addition

13: assert μ_{h_gpu} as 3D such that $\mu_{h_gpu} \in \mathbb{R}^{n_t \, x \, (m+1) \, x \, 1}$

14: $\mu_{gpu} \leftarrow \mu_{gpu}[:, 1{:}\,end-1, :] \oslash \mu_{h_gpu}[:, 1{:}\,end-1, :]$, #$\mu_h \neq 0 \; \forall$ elements, and $\mu_h \in \mathbb{R}^{n_t \, x \, (m+1)}$

15: $\zeta_{gpu} \leftarrow z_{gpu} - \sum_{c=1}^{d} (H^T \circ b_i)_{c_gpu}$, #where $\zeta_{gpu} \in \mathbb{R}^{n_t \, x \, 1}$

16: # note that this occurs the same way as the CPU implementation but leverages CuPy

Algorithm 4: GPU Implementation of the Measurement Update (CONT.)

17: $\dot{\boldsymbol{n}}_{\boldsymbol{t}} \leftarrow (m + 1) * n_t$ **,** #calculate new number of child terms generated during update

18: #now allocate empty space on GPU. Required due to parallel assignment of indices

19: A_{new_gpu} = empty matrix on GPU $\in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}\ x\ m\ x\ d}$

20: p_{new_gpu} = empty matrix on GPU $\in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}\ x\ m}$

21: b_{new_gpu} = empty matrix on GPU $\in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}\ x\ 1\ x\ d}$

22: q_{new_gpu} = empty matrix on GPU $\in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}\ x\ m}$

23: c_{new_gpu} = empty matrix on GPU $\in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}}$

24: d_{new_gpu} = empty matrix on GPU $\in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}}$

25: $\bar{A}_{gpu\ =}$ empty matrix on GPU $\in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}\ x\ m\ x\ d}$

26: zero_indices $\leftarrow [-1\ -1\ -1\ \ldots\ ]_{gpu} \in \mathbb{R}^{\dot{\boldsymbol{n}}_{\boldsymbol{t}}\ x\ 1}$

27: stack_index = create array of range($\dot{n}_t/(m+1)$) * $(m + 1)$, #create fancy indexing

28: #compute flattening terms once and repeat per term

29: $A_{h_gpu} \leftarrow \langle A_{gpu}[i], H_{gpu}\rangle$

30: $\tilde{\alpha}_{gpu} \leftarrow$ get_S_gpu(sign(A_{h_gpu}), mask) #conduct row-wise sign expansion

31: #$\tilde{\alpha}_p \in \mathbb{R}^{n_t\ \ x\, s_d(m)}$ where $s_d(m)$ = mask.shape[0]+1, so each parent's sign expansion

32: #was computed once, whereby hyperplane 1:m has the same sign expansion per term

33: $\tilde{\alpha}_p \leftarrow repeat(\tilde{\alpha})$ $(m + 1)$ times row-wise and store for future processing,

#$\tilde{\alpha}_p \in \mathbb{R}^{1\ x\ (m+1)*n_t\ \ x\, s_d(m)}$

34: #Because of line 36, this algorithm can now repeat the sign expansion m+1 times

35: $\tilde{\alpha}_p \leftarrow$ reshape $\tilde{\alpha}_p$ to $\in \mathbb{R}^{(m+1)*n_t\ x\ 1\ x\ s_d(m)}$

Algorithm 4: GPU Implementation of the Measurement Update (CONT.)

36: #Begin for loop to simultaneously compute all term updates across hyperplanes

37: For j ∈ $[1, 2, \dots, m+1]$:

38: e_{ll} ← [f for f in range(m+1) if f does not equal j], #create iterables

39: $a_{im_gpu} \leftarrow \mu_{gpu}[:, e_{ll}, :]$

40: $\mu_{tmp_gpu} \leftarrow \mu_{gpu}[:, j, :]$ and **assert** as 1xd vector

41: A_{new_gpu}[stack_index+j]← $a_{im_gpu} - \mu_{tmp_gpu}$, #where $A_{new} \in \mathbb{R}^{n_t \, x \, m \, x \, d}$

42: p_{new_gpu} [stack_index+j] ← $\rho[:, e_{ll}]$, #where $p_{new} \in \mathbb{R}^{n_t \, x \, m}$

43: b_{new_gpu} [stack_index+j] ← $\zeta[:] * \mu[:, j] + b[:]$, #where $b_{new} \in \mathbb{R}^{n_t \, x \, d}$

44: c_{new_gpu} [stack_index+j] ← $\zeta[:]$, #where $c_{new} \in \mathbb{R}^{n_t}$

45: d_{new_gpu} [stack_index+j] ← $\rho[:, j]$, #where $d_{new} \in \mathbb{R}^{n_t}$

46: q_{new_gpu} [stack_index+j] ← $\rho[:, e_{ll}]$, #where $q_{new} \in \mathbb{R}^{n_t x m}$

47: $A_{f_gpu} \leftarrow A_{gpu} \oslash A_{h_gpu}^{T}$, #row-wise parallel element divide, $A_{f_gpu} \in \mathbb{R}^{mxd}$

48: **if** j < m:

49: $\bar{A}_{gpu}$ [stack_index+j] ← A_{f_gpu} **-** $A_{f_gpu}[j]$

50: $zero_indices_{gpu}$ [stack_index+j] = j

51: **else**:

52: $\bar{A}_{gpu}$ [stack_index+j] = A_{f_gpu}

53: $zero_indices_{gpu}$ [stack_index+j] = -1

54: end For

55: return $\mathbf{A_{new_gpu}}$, $\mathbf{p_{new_gpu}}$, $\mathbf{b_{new_gpu}}$, $\mathbf{c_{new_gpu}}$, $\mathbf{d_{new_gpu}}$, $\mathbf{q_{new_gpu}}$, $\bar{A}_{gpu}$, $\tilde{\alpha}_p_{gpu}$, **zero_indices**$_{gpu}$

56: end GPU measurement update function

3.5. Evaluating Parameters for the ucpdf

3.5.1 Calculating the G's and Y_{ei}

This routine occurs separately from the measurement update and its main objective is to evaluate the parameters for the ucpdf, $\bar{\phi}_{X_{k+1}|Y_{k+1}}(v)$, itself. The wonder behind the parallelized version is that the algorithm can eliminate every for loop within the function with one temporary parameter transfers necessary to the GPU. Note that from here on out, the gpu designation is dropped because everything is now assumed to exist on the device.

After the measurement update, the g's are now evaluated as follows:

$$g_f^i = \frac{1}{2\pi}\left[\frac{S^+(b_{if}^p(z_i))^T\bar{\alpha}_i^p}{jc_i + d_i + y_{gi}} - \frac{S^-(b_{if}^p(z_i))^T\bar{\alpha}_i^p}{jc_i - d_i + y_{gi}}\right] \quad (3.5.1)$$

where the following parameters are defined as:

$$g^i = S(B_i)\alpha_i, \text{ and thus, } \alpha_i = S(B_i)^\dagger g^i \quad (3.5.2)$$

Equation 3.5.1 illustrates a high-level computing of the g-value per face, f, of i hyperplane arrangement. Equation 3.5.2 demonstrates that it is possible to find the g-value by describing the faces of an arrangement as a linear relation between a real basis matrix S_i and a complex g-coefficient vector α_i. Note that the superscript p stands for "parent".

The argument of the exponential for the ucpdf, Y_{ei}, is still evaluated as in equation 3.4.3:

$$y_{ei}^{k+1|k}(v) = -\sum_{l=1}^{n_{ei}^{k+1|k+1}} p_{il}^{k+1|k+1} | < a_{il}^{k+1|k+1}, v > | + j < b_i^{k+1|k+1}, v >$$

3.5.2 CPU Implementation of Calculating G's and Y_{ei}

In the current implementation there exists five major loops or operations that occur before the final G's or Yei's occur. In the first loop, the signs for the flattening term, $\bar{A}$ and the original A from the measurement update is found over each term, $\dot{\boldsymbol{n}}_{\boldsymbol{t}}$ using the root point**,** $\boldsymbol{\nu}$**.** $\bar{A}$ is used to calculate the sign basis for the G's while A is used for the argument of the exponential.

The second set of loop's main logic is now to assign positive 1's and negative 1's (which is used to evaluate the positive and negative side of the G's, respectively) wherever there are 0's that occurred from not finding a sign from the operation above. This first loop occurs over the number of terms by first checking whether there is a zero in the current row and then using another loop to find that 0's index and assign it the appropriate signed 1. Otherwise, if there does not exist a zero in the current hyperplane term, then these values are simply reassigned to a new variable.

The third loop is used to calculate y_{gi} for equation 3.5.1, as shown in section 3.5.1. This is simply an element-wise multiplication and a summation between q and the sign basis. Again, this is evaluated for every term.

The fourth loop represents the main computational overhead for this routine. First, an S expansion must occur for both the positive and negative S's (i.e S^+ and S^- as depicted in equation 3.5.1). This is further compounded by the fact that this must occur for every new child term generated from the measurement update routine. Because this serial implementation of the sign expansion consists of a double for loop, there exists a triple nested for loop. Afterwards, g^i is evaluated by computing the dot product between both S+ and α as well as S- and α.

After the G's are computed using the results from the fourth major loop, the y_{ei}'s are computed over every term as well. Unlike the g's, the y_{ei}'s are evaluated the same way as Idan and Speyer's methodology [28]. The CPU implementation of the algorithm is detailed below, mainly for comparison to the parallelized implementation.

Algorithm 5: CPU Implementation of Evaluating the G's and Y_{ei}

1: Input: A, $\bar{A}$, p, q, b, c, d, α_p, $\widetilde{\alpha_p}$, ν, masks

2: Begin CPU Evaluation of G's and Yei

3: _s_sgn_basis ← [0 0 0…; 0 0 0…] , #preallocate zeros to _s_sgn_basis ∈ $\mathbb{R}^{n_t \, x \, m}$

4: _sgn_basis ← [0 0 0…; 0 0 0…] , #preallocate zeros to _s_sgn_basis ∈ $\mathbb{R}^{n_t \, x \, m}$

5: For i ∈ $[1, 2 \ldots n_t]$: **#Loop 1, grab sign basis**

6: _s_sgn_basis[i] = sign(< $\bar{A}_i$, ν >), # _s_sgn_basis[i] ∈ $\mathbb{R}^{1 \, x \, m}$

7: _sgn_basis[i] = sign(<A_i , ν>), #_sgn_basis[i] ∈ $\mathbb{R}^{1 \, x \, m}$

8: End For

9: _sbp ← [0 0 0…; 0 0 0…], #preallocate zeros to _sbp ∈ $\mathbb{R}^{n_t \, x \, m}$

10: _sbm ← [0 0 0…; 0 0 0…], #preallocate zeros to _sbm ∈ $\mathbb{R}^{n_t \, x \, m}$

11: For i ∈ $[1, 2 \ldots n_t]$: **#Loop 2, assign positive and negative 1's to 0 indices**

12: **if** ∃ 0 $for\ any$ _s_sgn_basis[i]:

13: gate ← true, idx ← 0

14: **while** gate:

15: **if** _s_sgn_basis[0][idx] == 0:

16: gate = False

17: **else:**

18: indx += 1

Algorithm 5: CPU Implementation of Evaluating the G's and Y_{ei} (CONT.)

20: **end while**

21: sb_pos ← copy(_s_sgn_basis[i]) **,** #avoid pass by reference in Python

22: sb_neg ← copy(_s_sgn_basis[i])

23: sb_pos[idx] ← 1, sb_neg[idx] ← -1, #assign 1's to the indices of 0's

24: _sbp[i] ← sb_pos, _sbm[i] ← sb_neg, #form new row of signs

25: **else:** #∄ 0 $for\ any$ _s_sgn_basis[i]:

26: _sbp[i] ← _s_sgn_basis[i], **,**_sbm[i] ← _s_sgn_basis[i], # ∈ $\mathbb{R}^{1\,x\,m}$

27: End For

28: For i ∈ $[1, 2 \ldots n_t]$: **#Loop 3, compute y_{gi}**

29: ygi[i] ← $\sum_{i=1}^{n_t}(q[i] *\ _sgn_basis[i])$, #where ygi ∈ $\mathbb{R}^{1\,x\,m}$

30: End For

31: For i ∈ $[1, 2 \ldots n_t]$: **#Loop 4, obtain sign expansion and compute Sα**

32: sip_cut ← _sbp[i][: -1] , sim_cut ← _sbm[i][: -1] , #grab all columns to m-1

33: mask ← masks[(m-1, d)] , #Index masks dict using (m-1, d)

34: Sip ← get_S(sip_cut, mask), Sim ← get_S(sim_cut, masks) # sign expansions

35: $\alpha = \alpha_p[i] *\ \tilde{\alpha}[i],\ \alpha \in \mathbb{R}$

36 $\alpha_list = \alpha$, #store alpha for future use

37: g_p = Sip*α , g_m = Sim*α **, #compute coefficient for G's**

38: end For

39: Compute: $g_f^i = \frac{1}{2\pi}\left[\frac{S^+(b_{if}^p(z_i))^T\overline{\alpha}_i^p}{jc_i + d_i + y_{gi}} - \frac{S^-(b_{if}^p(z_i))^T\overline{\alpha}_i^p}{jc_i - d_i + y_{gi}}\right]$, $\#g_f^i \in \mathbb{R}^{n_t}$

Algorithm 5: CPU Implementation of Evaluating the G's and Y_{ei} (CONT.)

40: For i ∈ $[1, 2 \ldots n_t]$: **#Loop 5, compute y_{ei}**

41: _tmp_yei[i] = $\sum_{k=1}^{m}(A[i] * p[i] * _sgn_basis[i])$) , #sum across hyperplanes

42: end For

43: yei = 1j*b - _tmp_yei

44: **Return G's, yei, α_list**

45: End CPU Evaluation of G's and Yei's

3.5.3 Sending Requirements for Calculating G's and Y_{ei}

If the measurement update has been implemented on the GPU, then no parameter transfers are necessary during the evaluation of the ucpdf except for a mask used to generate the sign expansion. Regardless, this is very inexpensive transfer and this thesis notes that this is temporary only because reduction has not yet been implemented on the GPU. Otherwise, all parameters such as A, p, and so on were already on the GPU because of parallelizing the measurement update. Further, the root point is determined at the beginning of the algorithm, so it can be allocated to the GPU apriori, saving one additional real-time transfer. In total, by parallelizing the measurement update, this saves 10 real-time transfers with the potential to save 1 more after reduction has been implemented on the GPU.

3.5.4 The GPU implementation of Calculating G's and Y_{ei}

Using a combination of CuPy and kernelizing the sign expansion, it is possible to eliminate all major loops during this portion of the algorithm. First, finding the sign basis between each term can happen independently, just like the time propagation. In other words, the dot product between the root point $\boldsymbol{\nu}$ and each term $\dot{\boldsymbol{n}}_{\boldsymbol{t}}$ in $\bar{A}$ and the original A from the measurement update occurs simultaneously.

The second loop that handles the assignment for the 1's and -1's to the location of zero's in the sign basis can be parallelized. This involves using a GPU function that finds the indices of zeroes within the sign basis variable. Afterwards, using python's fancy indexing, the algorithm can efficiently assign 1's and -1's, thereby relegating a double loop to effectively 3 lines of code.

Parallelization of the third loop used to calculate y_{gi} for equation 3.5 is similar to how parallelization occurred in finding the sign basis. Independent element-wise multiplication can efficiently occur between terms as well as the summation needed to compute the y_{gi}.

The fourth loop that comprises the largest overhead is now relegated to two calls to the kernelized sign expansion function. This function finds the sign expansion for all terms that compromise the g vector. Two calls are necessary because one finds S+ and another finds S-. It may be possible to stream these two calls or perform both expansions simultaneously using separate grids on the GPU (assuming the hardware had not been saturated), but the author has not explored this in depth. Afterwards, g^i is evaluated by computing the dot product in parallel between all terms for S+ and α as well as S- and α.

After the g's are computed using the results from the α and sign expansions, the y_{ei}'s are computed over every term as well using a parallel implementation of element-wise multiplication and summation. This is near identical to the way y_{gi} is computed except there are some intermediate operations such as making the p's into a 3D vector to allow for parallel operations across stacks of terms. Note that these operations entirely depend on how the designer decides to store the variables in practice when implementing the estimator.

Algorithm 6: GPU Implementation of Evaluating the G's and Y_{ei}

1: Input: $\mathbf{A_{gpu}}$, $\bar{A}_{gpu}$, $\mathbf{p_{gpu}}$, $\mathbf{q_{gpu}}$, $\boldsymbol{b}_{gpu}$, $\mathbf{c_{gpu}}$, $\mathbf{d_{gpu}}$, $\boldsymbol{\alpha_p}_{gpu}$, $\widetilde{\boldsymbol{\alpha_p}}_{gpu}$, $\boldsymbol{\nu}_{gpu}$, **masks**

2: Begin GPU Evaluation of G's and Yei

3: _s_sgn_basis ← [0 0 0…; 0 0 0…] , #preallocate zeros to _s_sgn_basis ∈ $\mathbb{R}^{n_t\ x\ m}$

4: _sgn_basis ← [0 0 0…; 0 0 0…] , #preallocate zeros to _s_sgn_basis ∈ $\mathbb{R}^{n_t\ x\ m}$

5: In Parallel, for i ∈ $[1, 2 \dots n_t]$: **#grab sign basis across all terms**

6: $\quad$ _s_sgn_basis$_{gpu}$[i] = sign(< $\bar{A}_{gpu}$, ν >), # _s_sgn_basis$_{gpu}$ ∈ $\mathbb{R}^{n_t\ x\ m}$

7: $\quad$ _sgn_basis$_{gpu}$ [i] = sign(<A_{gpu} , ν>), #_sgn_basis$_{gpu}$ ∈ $\mathbb{R}^{n_t\ x\ m}$

8: #note the use of gpu copy below to avoid pass by reference

9: _sbp$_{\mathbf{gpu}}$ ← copy(_s_sgn_basis_gpu), #preallocate zeros to _sbp ∈ $\mathbb{R}^{n_t\ x\ m}$

10: _sbm$_{\mathbf{gpu}}$ ← copy(_s_sgn_basis_gpu), #preallocate zeros to _sbm ∈ $\mathbb{R}^{n_t\ x\ m}$

11: row, column ← where(_s_sgn_basis_gpu == 0), # rows and column index of 0's

12: _sbp$_{gpu}$[row, column] ← 1, _sbm$_{gpu}$ [row, column] ← -1, #vectorize assignment

13: In Parallel, for i ∈ $[1, 2 \dots n_t]$: **compute ygi across all terms**

14: $\quad$ ygi$_{gpu}$ ← $\sum_{i=1}^{n_t}(q_{gpu} * _sgn_basis_{gpu})$, #ygi$_{gpu}$ ∈ $\mathbb{R}^{n_t\ x\ m}$

15: mask ← masks[(m-1, d)] , #Index masks dict using (m-1, d)

16: #Invoke 2 calls to parallel implementation of sign expansion algorithm on GPU

17: Sip$_{gpu}$ ←get_S_gpu(_sbp[:, :-1], mask), Sim$_{gpu}$ ← get_S_gpu(_sbm[:, :-1], mask)

18: **In Parallel ,** for i ∈ $[1, 2 \dots n_t]$:

19: $\quad$ $\alpha_{gpu} = \alpha_{p_gpu} * \alpha_{gpu}$,

20: $\quad$ g_{p_gpu} = Sip$_{gpu}$*α_{gpu} , g_{m_gpu} = Sim$_{gpu}$*α_{gpu} **, #compute coefficient for G's**

21: Compute: $g_f^i = \frac{1}{2\pi}[\frac{S^+(b_{if}^p(z_i))^T\bar{\alpha}_i^p}{jc_i + d_i + y_{gi}} - \frac{S^-(b_{if}^p(z_i))^T\bar{\alpha}_i^p}{jc_i - d_i + y_{gi}}]$

Algorithm 6: GPU Implementation of Evaluating the G's and Y_{ei} (CONT.)

22: yei_{gpu} = 1j*b_{gpu}

23: In Parallel, for i ∈ $[1, 2 \ldots n_t]$: **#Compute yei across all terms**

24: yei_{gpu} -= $\sum_{k=1}^{m}(A_{gpu} * p_{gpu} * _sgn_basis_{gpu})$) , #sum across all hyperplanes

25: Return G_{gpu}, yei_{gpu}, $\boldsymbol{\alpha_{gpu}}$

26: End GPU Evaluation of G's and Yei's

3.6. Coalignment Introduction and Serial Implementation

3.6.1 Coalignment Introduction

The Coalignment Algorithm removes any hyperplanes that coalign with another hyperplane within the same term. In other words, Coalignment seeks to remove all repeated hyperplanes. Moreover, this algorithm operates term-by-term such that the results of one term in the hyperplane arrangement does not depend on the preceding results. Although not immediately apparent, this sets the precedence for parallelism and enables significant speed improvements over its serial counterpart.

Like the preceding algorithms, this thesis will cover the serial implementation before delineating the parallel implementation. First, a methodology to parallelize hyperplane comparisons using GPU threads is shown. By using knowledge of these thread's ID, the algorithm implicitly determines what hyperplanes coalign. Then coalignment is recast into a form that allows implementation of a partial GPU scan operation to combine coefficients within the exponential of the Cauchy's characteristic function using the same thread-block pairs that operated on coaligned hyperplanes. Finally, the output from the kernelized coalignment goes through post processing to remove repeated hyperplanes.

Both the CPU and GPU implementation of term-coalignment saves off ancillary results. In particular, the direction vectors point with respect to each other and the particular hyperplanes that coalign are stored in a look-up table format. This is a necessary step since flattening necessitates expanding the enumerated hyperplanes to perform a sign expansion and ultimately calculate the g's using a linear basis. These stored results from term-coalignment give a blueprint to tell the flattening algorithm which indices to insert hyperplane arrangements.

Coalignment Nomenclature

- let $\boldsymbol{d}$ be the state dimension
- let $l \in L$, where L is the set of currently generated hyperplane arrangement sizes over all terms
- let n_t represent the total number of terms generated at the current step. Note that the number of hyperplanes in all arrangements n_t is given by the set L
- let $i \in [1, 2, \dots n_t]$ be the index for the i-th term of the n_t term
- Let $A_i \in \mathbb{R}^{mxd}$ be the matrix which holds term I's hyperplanes, of dimension d
- $A_{im} \in \mathbb{R}^{1xd}$ represents the m-th hyperplane (row of A_i) for term i.
- Note that all hyperplanes are assumed to be normalized
- let $p_i \in \mathbb{R}^m$ be the coefficients to the i-th term's hyperplane arrangement, within the exponential of the Cauchy's characteristic function.
- let $q_i \in \mathbb{R}^m$ be the coefficients to the i-th term's hyperplane arrangement, within the exponential of the Cauchy's characteristic function.

3.6.2 Coalignment Serial Implementation

The serial or CPU implementation of term-coalignment occurs term-by-term. Intuitively speaking, a sequential implementation must check each term and then check each hyperplane within that term against all other hyperplanes. In this thesis' implementation, a term within the hyperplane arrangement goes into the term-coalignment function and computes a dot product. If vectors are coaligned, then the dot product should equal close to 1. Thus, the algorithm uses a parameter, $\mathcal{E}$, to see whether it is numerically close enough to zero when subtracting from 1. Additionally, the sign mapping between hyperplanes (i.e. whether the dot product process a -1 for a 1) is also computed and stored along with the indices the coaligned planes. This is used later in Flattening to reinsert enumerated hyperplanes (results from Reverse Search) into their original locations since term-coalignment inherently changes the size of hyperplanes.

In the below formulation, A_c represents the current hyperplane and A_r is the reference hyperplane being compared to:

$$|\,(1.0 - |A_c * A_r|)\,| < \mathcal{E} \qquad (3.6.1)$$

If this conditional pass, then term-coalignment proceeds to add coefficients within the argument of the exponential. Similar to above, a for loop must add all of these parameters to the current reference coefficient (the p_i or q_i that corresponds to A_c) [28]: Where the θ_l below is some nonzero constant related by $A_{i\overline{m}}^{k+1|k} = \theta_m A_{im}^{k+1|k}$, and $\overline{m} \neq m$.

$$\overline{q_{\imath m}} = q_{im}^{k+1|k} + \text{sign}(\theta_m)\, q_{i\overline{m}}^{k+1|k},\ \ \overline{p_{\imath m}} = p_{im}^{k+1|k} + |\theta_m|\, p_{i\overline{m}}^{k+1|k} \qquad (3.6.2)$$

Otherwise, if equation 3.6.1 does not pass, then hyperplanes do not coalign and the algorithm proceeds with checking all other hyperplanes within the same term. This repeats for all $i \in [1, 2, \ldots n_t]$. The serial implementation of term coalignment is given in algorithm 11.

Algorithm 11: CPU Coalignment Algorithm

the following algorithm is computed over all all $i \in [1, 2, \ldots n_l]$.

1: Input: A_i, p_i, q_i

2: m ← number of hyperplanes in arrangement i

3: $\mathcal{E}$ ← some acceptably small positive number

4: F ← set to all True, # $F \in B^m$

5: Declare: $maps_i$ = { }, $sign_maps_i$ = { }

6: For j = 1 to m-1 **do:**

7: **if** F[j] **then**:

8: #testing hyperplane j for coalignment

9: $a_r = A_i$[j,:] # $a_r \in \mathbb{R}^d$

10: $A_c = A_i$[j+1 : m, :] #$A_c \in \mathbb{R}^{m-j \, x \, d}$

11: indxs = argwhere($|\ (1.0 - |A_c * A_r|)\ | < \mathcal{E}$)

12: **if** len(indxs) ≠ 0 **then**

13: maps[j] = indxs + j;

14: sign_maps[j] = zeros(len(indx))

15: k = 0

16: **For** indx in indxs **do**

17: s = sign(a_r *A_i[indx,:]) #find sign mapping

18: p_i[j] = p_i[j] + p_i[indx] #combine terms

19: q_i[j] = q_i[j] + s * qi[indx]

20: sign_maps[j][k] = s #store sign mapping

21: F[indx] = False;

22: k = k+1

23: **end For**

24: **else:**

25: #hyperplanes are unique, so no action is taken

26: **end if-else**

27: **end if**

28: end For

29: Return A_i, p_i, q_i, for **True** indices of *F,* and $maps_i$, $sign_maps_i$

30: End CPU Term Coalignment

3.7. Coalignment Parallel Implementation

Like the GPU S-Expansion Algorithm, this thesis separates the parallel design of Term-Coalignment to aid readability and to better highlight the considerations made when designing the kernel. First, the algorithm takes into GPU-Coalignment stacked hyperplane arrangements and their corresponding coefficients. For example, the hyperplanes might take the form of 4x3, 5x3, or 6x3 matrices; so, the inputs to GPU-coalignment should be a 15x3 matrix here. The coefficients of the argument of the exponential should align with this newly stacked form as well. Then, parallel coalignment simultaneously checks all hyperplanes against each other using a dot product and an appropriately small epsilon value. These results then tell the next phase how to combine the arguments of the exponential using a method based off a GPU-Scan (Parallel Prefix) before finally returning the results in the same stacked format.

3.7.1 Inputs into Parallel Coalignment

When looking at the serial implementation of term-coalignment, parallelization seems difficult on the surface since each hyperplane must evaluate against others in the current arrangement. However, when the algorithm preprocesses the hyperplanes and coefficients into a stack before inputting them to the kernelized coalignment, this opens some possibilities. Hyperplanes have the property that their columns always reflect the dimension of the problem. Because this thesis considers a 3-state system, the matrices will always be m x 3 where m represents the hyperplane number in the current arrangement. This allows stacking along the vertical axis as seen in figure 3. This affords GPU designers two distinct advantages. This tall matrix maximizes parallelization by maintaining a contiguous format. Secondly, the algorithm can now operate on every hyperplane, not just across terms but across all arrangements. This contributes to substantial efficiency gains when compared to its serial counterpart.

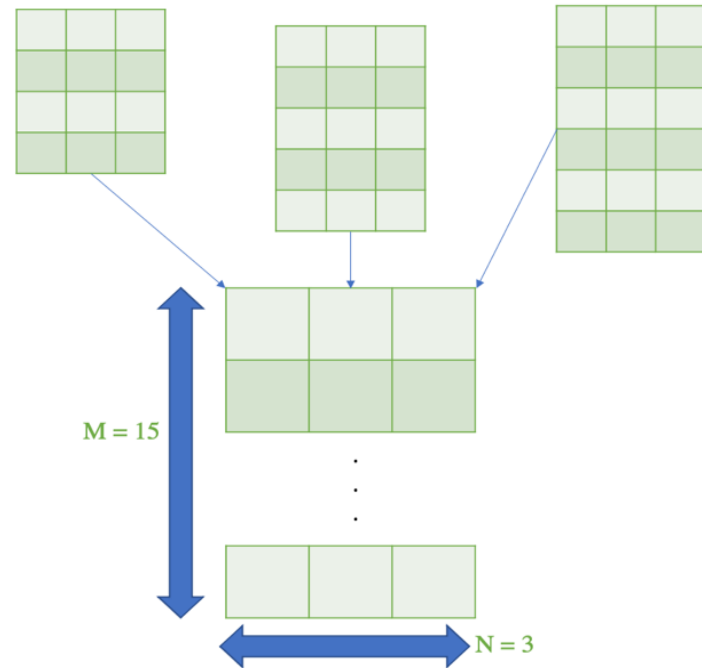

Figure 3: Visualization of stacking preceding arrangements into one large matrix

Because all preceding portions leading up to point – from time propagation to computing the g's – were parallelized, this allows zero new transfers to the GPU for coalignment, further reducing any unnecessary bottlenecks in the current implementation. Also note that the kernel assumes the matrices have been normalized apriori. Further, the kernel requires some supporting inputs such as the differently sized hyperplane arrangements and the start indices for each new arrangement. Fortunately, CuPy allows some simple function calls to parallelize all the aforementioned operations.

3.7.2 Parallel Coalignment – Choosing Thread-Block Pairs

Figure 4 demonstrates how the kernel allocates threads to compares hyperplane arrangements against each other. This gives some insight into the minimum number of threads should ACE allocate before calling the coalignment kernel. The minimum number of blocks allocated should equal the total number of terms across all hyperplane arrangements.

The following equation reads, the number of minimum blocks to allocate equals the sum of the number of terms in all hyperplane arrangements:

$$\text{Number Blocks to Allocate} = \sum_{k}^{\max(k)} A_{n_t k}$$

In the following figure, Tx represents threads allocated in the x-direction, Ty represents threads allocated in the "y-direction", and k represents iterations in a loop that point over the columns. To be clear, Ty threads are not actually pointing in the y-direction in this instance but are actually being used to independently assess other hyperplanes against threads in the x-direction. $A_{current}$ and $A_{reference}$ are equal to each other, and a dot product is computed between rows. For example, Tx_0 computes a dot product against the row that Ty_0 points to, Ty_1, and Ty_2. Thus, to simultaneously compare all hyperplanes, this kernel requires at least (m-1)x(m-1) threads where m equals the number of hyperplanes in the current arrangement.

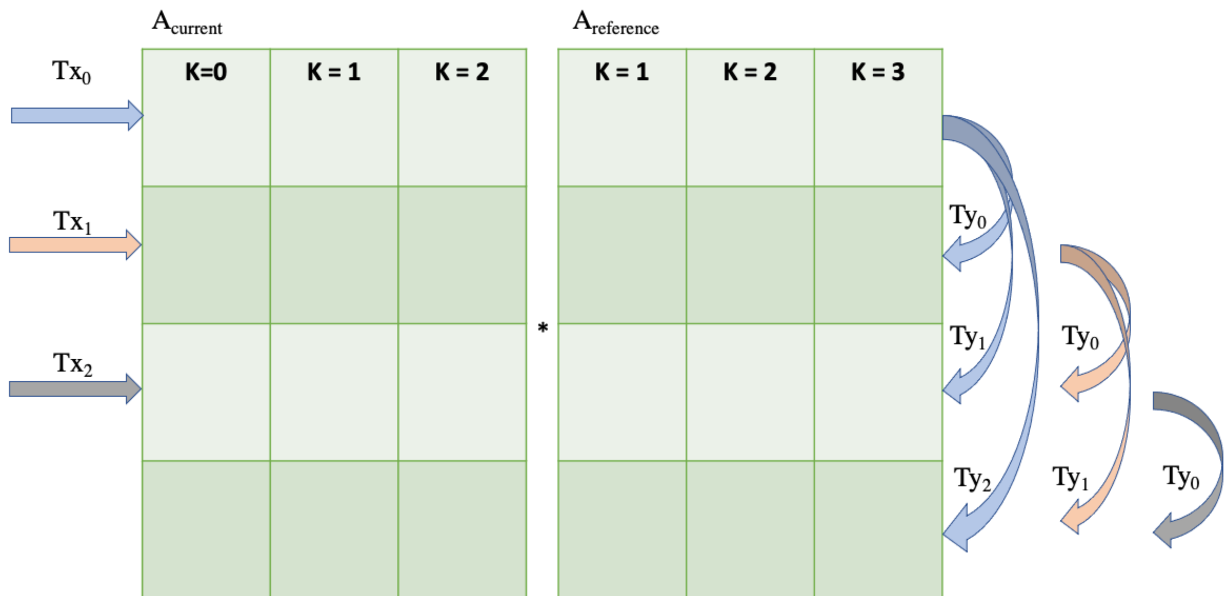

Figure 4: This visualizes how threads point at different hyperplanes. This particular term requires 9 threads. Note that each thread in the x has a corresponding set of threads in the y.

Although Figure 4 demonstrates that a 4x3 term requires at least 9 threads, ACE launches threads in pairs of 32 to take advantage of the CUDA programming model's ability to synchronize Warps [31]. A Warp is a pair of 32 threads that NVIDIA GPU's launch simultaneously. Consequently, if any of these threads reach a conditional or a loop, then thread divergence occurs [31]. Programming techniques such as avoiding branching or loop unrolling is leveraged to minimize this phenomenon.

3.7.3 Parallel Coalignment – Removing Repeated Hyperplanes

In the current implementation, if a thread pair (say Tx_0 and Ty_1) finds that a corresponding hyperplane coaligns, then it uses the threads in the y-direction to set all indices in that vector to zero. Later, ACE uses a CuPy implementation of argwhere() to find all columns that have a zero and fancy indexes to remove these arrangements. Figure 5 demonstrates this access and removal of repeated arrangements. The thread indices that correspond to the coaligned hyperplanes are then used to combine terms for the coefficients in the exponential of the characteristic function.

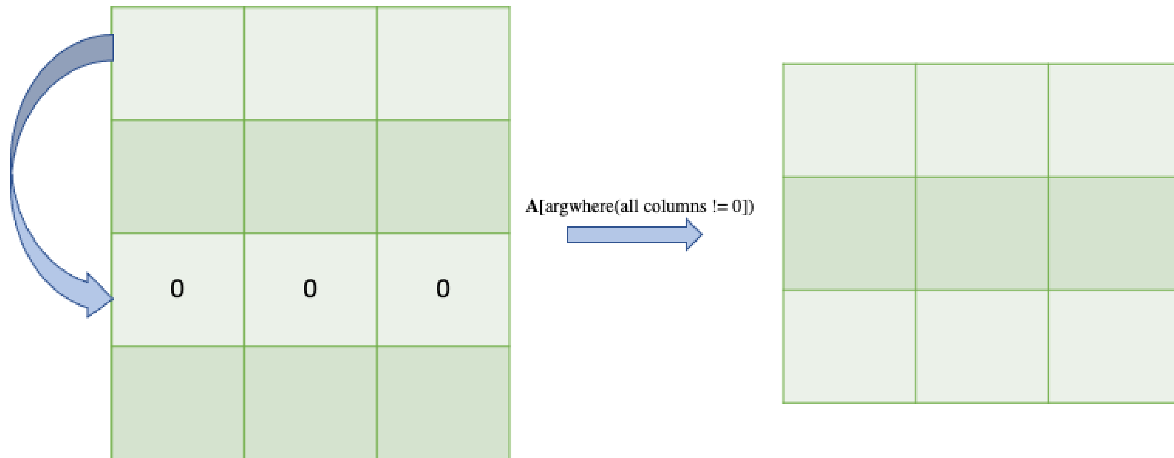

Figure 5: Let A represent the original 4x3 hyperplane arrangement. If the first hyperplane coaligns with the third hyperplane, then those indices are zeroed out and removed in parallel, across all terms, in postprocessing.

3.7.4 Parallel Coalignment – Parallel Reduction of Coefficients

As in the case of the serial implementation, the coefficients of the characteristic function, **p** and **q** must also combine with the coefficient that does not coalign. Algorithmically speaking, this also looks difficult to parallelize since the serial implementation has to loop and add coefficients together one-by-one. Figure 6 illustrates a serial reduction of values.

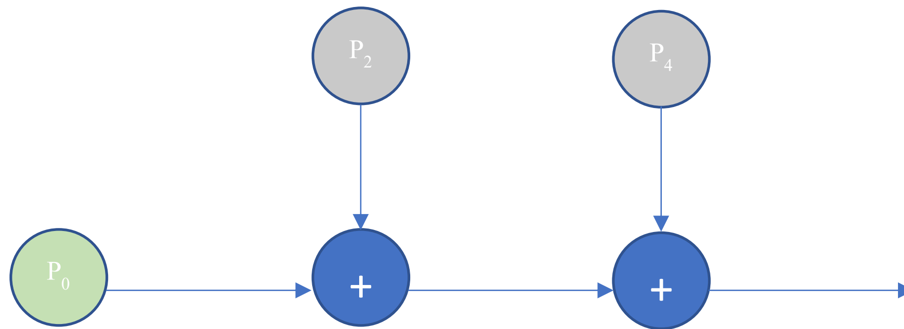

Figure 6: Visualization of a serial reduction of terms for the coefficient, P. Here, P combines with the 3rd and 5th hyperplane arrangement.

Interestingly, the reduction of p and q falls under a class of problems known as the Prefix-sum. The Prefix-sum algorithm represents one of the most important building blocks in modern day parallel computation, such as in radix-sort, solving tri-diagonal linear systems, or quicksort [36]. Essentially, the algorithm computes the sum of preceding terms, similar to what must occur for these coefficients. After the introduction of Hillis and Steele Jr.'s work [37], a parallel implementation of the Prefix Sum was introduced by Daniel Horn [6]. Thus, ACE is inspired by the work of these authors and recasts the reduction of p and q into a partial Parallel Prefix-Sum algorithm. In addition, the algorithm deploys sequential addressing for coalesced memory access to further improve performance [31]. Figure 7 showcases ACE's implementation and notes that the same method applies for both p and q, except q has a potential sign change depending on hyperplane direction.

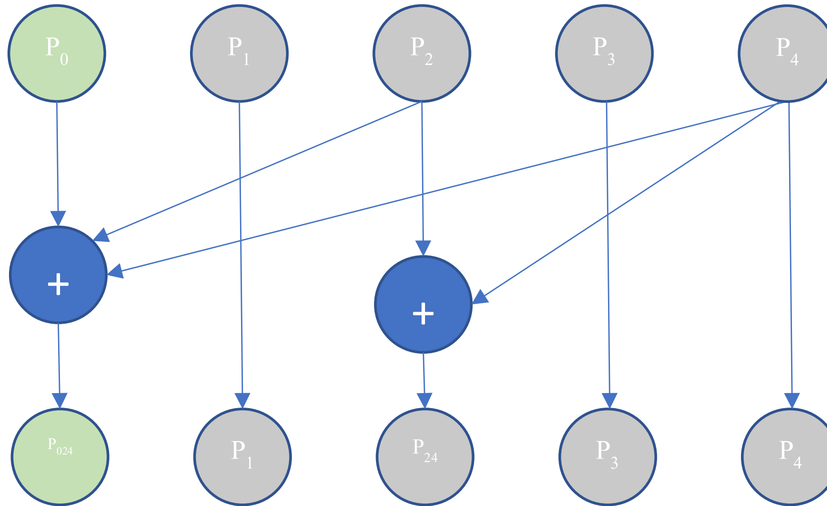

Figure 7: Visualization of a parallel reduction of terms for the coefficient, P. Here, P combines with the 3rd and 5th hyperplane arrangement. Each arrow represents a different thread.

One important thing to note is that in this implementation, the 3rd and 5th hyperplane also add together. Nevertheless, these indices are discarded in the post processing phase and thus become a non-issue. Similar to the hyperplane arrangements, these coefficients also occur on a per-block basis, so each block uses these interleaved threads to perform the parallel reduction

3.7.5 Parallel Coalignment – Global Versus Shared Memory

The last design decision involves how to allocate the variables in GPU memory. For the purposes of this work, there exists 3 primary modes of memory management, Global, Shared, and Local Memory. Global memory has global scope which just means all threads across blocks have access to variables stored here. However, it is also the slowest memory type and because memory sending or retrieving (IO) dominates runtime, ACE should attempt to access global memory as little as possible [35] Shared memory, or the L1 Cache, is very fast when compared to global memory but only threads within a block have access to variables stored here. Finally, Register or Local memory, is private to each thread and is even faster than shared memory [35].

For the parallel implementation of coalignment, the hyperplanes that have zeroed out vectors exist in global memory whereas the hyperplanes that perform the dot product exist in local memory. On the other hand, the coefficients of the exponential exist in shared memory since they require interthread communication. Independence of each term contributes to the logic behind this methodology. Because threads among terms do not have to communicate, this allows the storing of p and q on shared memory where threads within the same blocks have access [35]. On the other hand, the hyperplane "strip" must exist on global memory since threads outside of the same block must access and zero out their corresponding vectors. Figure 8 depicts a high-level view of how memory storage and communication occur for GPU-Coalignment.

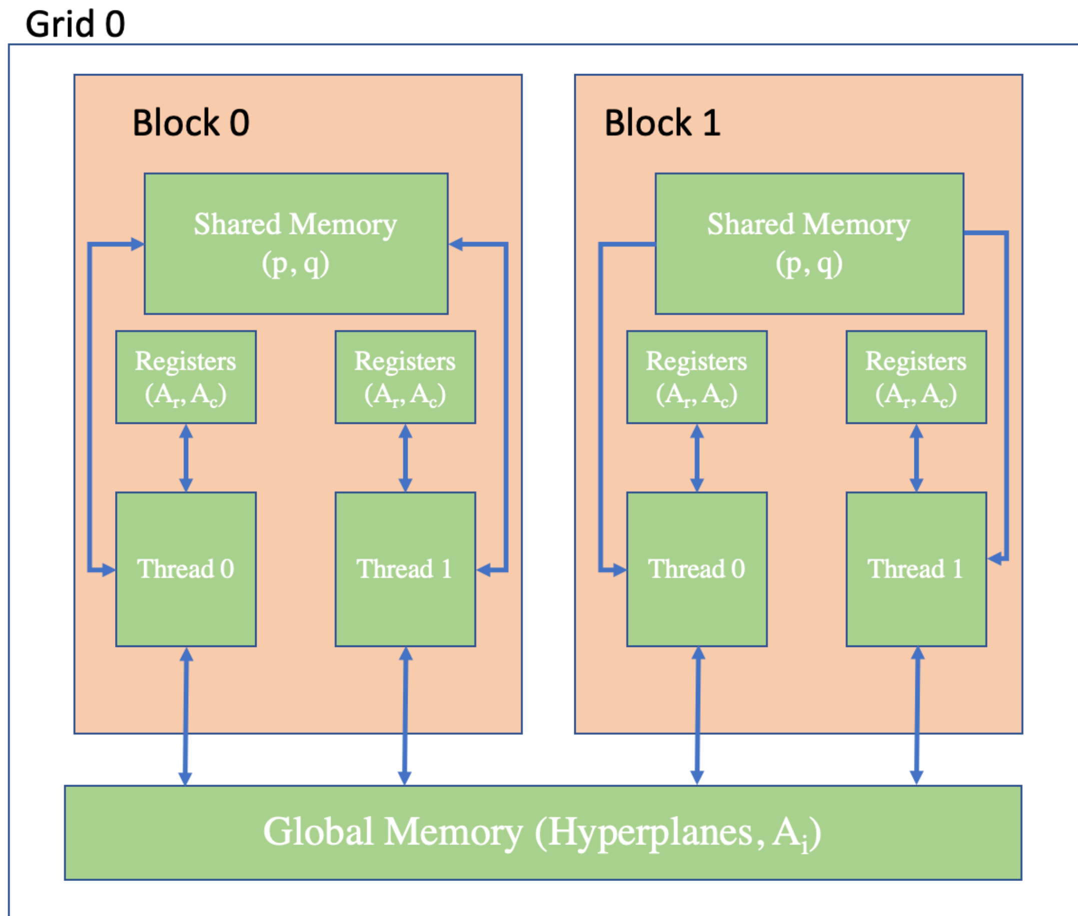

Figure 8: High level view of the interoperability between different memory types and how hyperplanes or coefficients are stored. In this example, only two blocks and two threads are shown over a single grid.

Algorithm 12: GPU Coalignment Algorithm

```
// the following algorithm is computed over all i ∈[1, 2, … n_t], in parallel.
// A_shapes are the shapes of every stacked term's hyperplane
//start_indx are the indices of where each new hyperplane begins, row-wise
1: extern "C" __global__
2: void coalign_main(A_stack, p_stack, q_stack, A_shapes, start_indx){
3:      int Tx, Ty ← threadIdx.x, threadIdx.y; //declare thread and block indices
4:      int Bx, By ← blockIdx.x, blockIdx.y;
5:      coalign_and_reduce(A_stack, p_stack, q_stack, A_shapes, start_indx, Tx, Ty, Bx, By)
6:      }
7: end GPU Coalignment

8: extern "C" __device__
9: void coalign_and_reduce(A_stack, p_stack, q_stack, A_shapes, start_indx, Tx, Ty, Bx, By){
10:     int hp_num ← start_indx[Bx]; // grabs desired hyperplanes
11:     int max_num_hp ← A_stack[Bx];
12:     store on shared memory ← p_shared[];
13:     store on shared memory ← q_shared[];
14:     p_shared[tx] ← p_stack [tx+hp_num]; // copy global to shared memory
15:     q_shared[tx+max_num_hp] ← q_stack; //offset to use same memory space
16:     float *A_r, A_c ← A_stack //make local variable copies
17:     float *p_r ← p_stack
18:     float *q_r ← p_stack
19:     __syncthreads(); // finish all memory copies before proceeding
20:     If ((tx+ty+1) < max_num_hp) { // double check we don't over index
21:             For(int k = 0; k < d; k ++) { // loop across columns (d = 3 in this work)
22:                     float result_of_dot_prod += A_r[tx*d+k+d*hp_num]…
                                                  *A_c[(tx+ty+1)*d+k+d*hp_num];
23:             }
24:             if ( | 1-|result_of_dot_prod| | < eps){ // set to 1e-7 here
25:                     Set all column indices in A_mega[(tx+ty+1)*d + d*hp_num] to 0;
```

Algorithm 12: GPU Coalignment Algorithm (Reduction of Coefficients in Exponential)

```
// The following algorithm performs sequential addressing and a reduction-type operation
// continuing from line 25 above
26:                 int s = signum(result_of_dot_prod);
27:                 //force serialized write (additions) to same address space
28:                 atomicAdd(&p_shared[tx],p_r[tx+ty+1+hp_num]);
29:                 atomicAdd(&q_shared[tx+max_num_hp],s*q_r[tx+ty+1+hp_num]);
30:                 // note that q_shared is offset by max_num_hp
31:                 p_stack [tx+hp_num] ← p_shared[tx]; //reassign shared back to global
32:                 p_stack [tx+hp_num] ← q_shared[tx+max_num_hp];
33:           }
34:     }
35: }
36: End coalign_and_reduce device function call
```

3.8. S-Expansion and Least Norms - Introduction

3.8.1 Kernelizing the S-Expansion - Motivation

After coalignment, reverse search enumerates the unique hyperplane arrangements. These enumerations from reverse search then has to expand using an "S-expansion" algorithm such that it can be used as the least norm solution in flattening. This section details kernelizing the GPU implementation of the S-expansion as well as when the matrices are inverted for the least norm solution. ACE uses this algorithm several times throughout the entire estimator; most notably during the measurement update, initial G calculation, right after the reverse search, and during flattening. As a result, this routine serves as one of the most important portions of the algorithm to parallelize.

Further, the results presented here served as the initial motivation for parallelizing the rest of the Cauchy Estimator. After conducting an initial code review to see where bottlenecks exist, this section was identified as one of the most promising areas to pursue. This led to the development of S-expansion algorithm for the first custom kernel to see whether applying GPU programming was worthwhile.

At the time of writing, reverse search has not yet been developed for the GPU. As a result, code refactoring had to occur in order to support a GPU implementation of the S-expansion algorithm. Practically speaking, another CPU algorithm was developed to enable the GPU to take in as inputs the differently sized matrices that occurred as a result of coalignment. In particular, the CPU portion that occurs before the GPU expansion uses a list comprehension to identify and group similarly sized matrices along with its initial ordering due to the CUDA programming model requiring contiguous arrays as inputs. Then, ACE uses CUDA streams to asynchronously transfer

data to the GPU before conducting an expansion in parallel, across several terms and hyperplane arrangements. These streams improve performance by allowing for course-grain concurrency and are applied again after the expansion to stream required results back to the CPU.

3.8.2 S-Expansion Nomenclature and Incremental Enumeration

- Let $S_d(\sigma_m)$ be the function which provides a combinatorial expansion of m-singletons σ_m, up to dimension d.
- Let $s_d(m)$ be the count of elements generated from conducting a combinatorial 'S' expansion of m-singletons in dimension d, for central hyperplane arrangements
- S is overloaded to expand a matrix $Z \in R^{F \times m}$ of sign sequences row-wise to column length $s_d(m)$.
- Let $Bi \in R^{Fi \times mi}$ be m_i-singleton sign sequences. This output from reverse search describes how to locate a particular face $f \in [1, 2, ...F_i]$ for the $i \in [1, 2, ...n_t]$:

$$B_i = IncEnu(A_i) = \begin{bmatrix} b_{i1} \\ b_{i2} \\ \vdots \\ b_{iF_i} \end{bmatrix}$$

The above cell-enumeration algorithm utilized for this thesis is Incremental Enumeration (IncEnu), which is a variation on the reverse search method [8] outlined in [25]. IncEnu will find all unique faces of a hyperplane arrangement. This requires that coalignment removes aligned hyperplanes. Otherwise, the algorithm fails. As shown above, IncEnu returns a matrix $Bi \in R^{Fi \times mi}$ for a hyperplane arrangement $A_i \in R^{mi \times d}$, where F denotes the number of faces found in arrangement A_i and here, m_i denotes the number of hyperplanes in A_i (e.g 4, 5, and so on). The singleton sequence described above indicates which side of each hyperplane the *f*-th face lies.

3.8.3 S-Expansion Preliminaries

The S-expansion algorithm at its core takes the results of the reverse search and performs a combinatoric sign sequence. For example, a sign sequence, $b = [\sigma_1\ \sigma_2\ \sigma_3]$, with $\sigma \in \{-1, 1\}$, with a state dimension of $d = 3$, would have a sign expansion shown in equation 3.8.

$$S(b) = S([\sigma_1\ \sigma_2\ \sigma_3]) = [1, \sigma_1, \sigma_2, \sigma_3, \sigma_1\sigma_2, \sigma_1\sigma_3, \sigma_2\sigma_3, \sigma_1\sigma_2\sigma_3] \qquad (3.8)$$

Because the resulting matrix results in more columns than rows, a pseudoinverse results in the least norm solution when solving for $\alpha_i \in C^{sd(mi)}$.

3.8.4 CPU Implementation of S-Expansion Algorithm and Least Norms Solution

The CPU implementation leveraging JIT is relatively straightforward. Essentially, a nested for loop is used to conduct the expansion dimension d times, over the number of combinations minus 1. The minus 1 is because the first column of the expansion is skipped. The resulting S is then formed a column at a time.

Algorithm 7: CPU Implementation of the S-Expansion Algorithm

```
@njit()
1: Begin CPU S-expansion
2: Input: B, masks
2: number_faces ← number of rows in B
3: number_combos = number of rows in masks + 1
4: S ← ones(number_faces, number_combos), # S ∈ R^(F× s(m))
5:      For i in range(number_combos – 1):

6:           For j in range(dimensions):
7:                if masks[i,j] != -1:
8:                     indx = masks[i][j]
9:                     S[:, i + 1] *= B[indx]
10:               end if
```

Algorithm 7: CPU Implementation of the S-Expansion Algorithm (cont.)

11: **end For**

12: **end For**

13: Return **S**

14: End CPU Implementation of the S-Expansion Algorithm

Note that in the above implementation that a variable "masks" was used. This mask can be obtained and prestored in the beginning of the algorithm. This is worth noting because this saves at least one real-time transfer for the GPU implementation.

3.8.5 Challenges in the GPU Implementation of the S-Expansion Algorithm

The parallel or custom kernelized implementation is an involved process in the sense that the asymmetric dimensions that result from reverse search results in non-contiguous inputs to the GPU. As a result, careful attention is needed to optimize the latency when performing a data transfer. Moreover, because this is a custom kernel, several layers of concurrency takes place when designing a GPU program and the designer needs to choose carefully what sections to loop, how to allocate shared or global memory, and how to distribute the thread-block cluster.

Although the algorithm itself at its core perform an expansion over a double for loop, it also provides a strong case study to apply fundamental GPU programming techniques. For example, it showcases quintessential parallel programming challenges, such as code refactoring, impact of data transfers, and fundamental considerations when designing algorithms using the CUDA programming model. The goal thus relegates to exploiting parallelism when seemingly difficult to do so. These techniques apply broadly to parallel programming and this work reiterates similar approaches when delineating the section on coalignment.

3.9. Parallel S-Expansion and Least Norms – Implementation

This thesis separates the parallel design of the S-expansion algorithm to aid readability and to help draw attention to considerations made when designing the kernel. Secondly, when implementing the Moore Penrose Pseudoinverse in calculating the least norm, this work uses a Cupy implementation of matrix multiplications and inverses. Unfortunately, at the time of writing, a parallel implementation of the singular value decomposition did not work when applied to 3-D matrices. Thus, this thesis invokes a simple theorem to show an equivalent implementation that is more suitable to GPU implementation.

3.9.1 CPU Refactoring to Enable Contiguous Inputs to the GPU

The first important design decision made when kernelizing the S-expansion involves how the CPU will group the enumerated hyperplanes such that it preserves contiguity. To maximize parallelism, the GPU should perform as many simultaneous operations as possible [24]. Forcing a

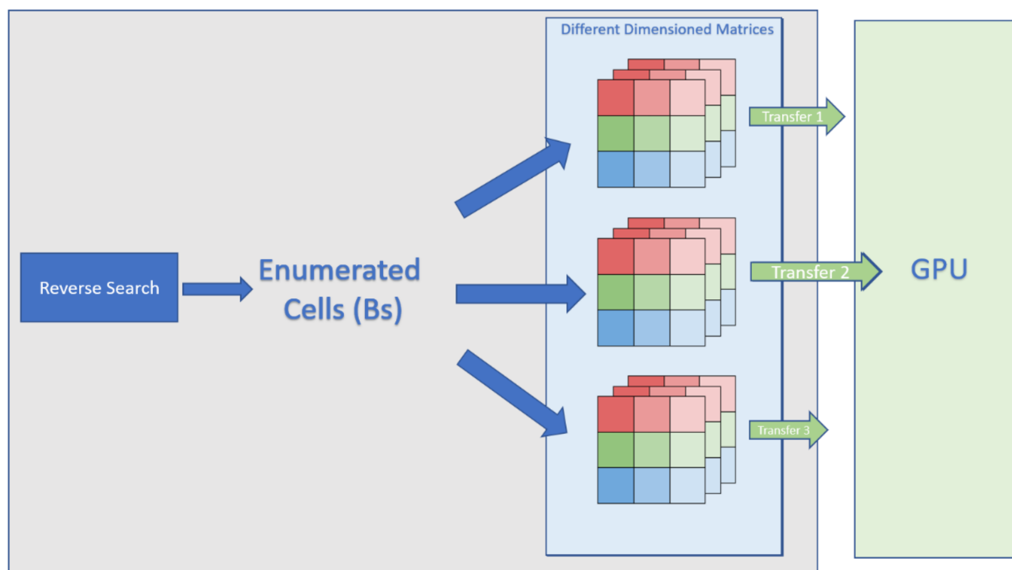

Figure 9: The aggregation of enumerated hyperplanes and how they flow asynchronously to the GPU due to CUDA streams.

loop to handle jagged matrices within the kernel compromises this and, as a result, nullifies any speed improvements that the hardware offers. This is precisely why designers should try to maximize how many operations happen in parallel by sending as large of a contiguous array as possible. Figure 9 showcases a high-level view of how the CPU portion of the algorithm groups the matrices into like dimensions for parallel expansion. Note the separation of the host and device hardware when looking at the flow of data as well as how the matrices are streamed to the GPU.

ACE begins by separating the enumerated faces for each hyperplane arrangement. So, enumeration for 4 hyperplane arrangements are grouped versus a 6 hyperplane arrangement. Afterwards, the question remains on how the CPU should combine the differently sized enumerated faces due to coalignment combining hyperplanes within terms – again because the GPU requires a contiguous input. This is done by extracting the unique enumerated faces across all the terms by looking at the number of rows multiplied to the number of columns. Afterwards, a loop runs over and stacks matrices that correspond to these unique dimensions, thus minimizing the amount of iterations required. The indices for teach term corresponding to their original locations are also stored to reassemble the expanded matrices back to their original locations.

Algorithm 8: CPU Implementation of B-Sort Algorithm

1: Begin CPU Sorting of Unique Dimensioned Hyperplane Terms

2: Input: B

3: Bs_unique_dim ← [size(b) for b in B] #where size is the row*columns

4: For vals **in** Bs_unique_dim:

5: idx[ii] = where(Bs_size == vals) #find indices corresponding to these sizes in B

6: stack_Bs[ii] = itemgetter(*idx[ii])(B) #use idx to sort B's into ordered stack

7: ii+= 1

8: end For

9: return stack_Bs, idx

For completeness, the CPU reordering algorithm for the original B's, once they have been expanded, is also given. This routine is a fair bit simpler and simply requires as inputs, the unsorted S-expanded matrices, and the indices of their original location (given in Algorithm 7). Algorithm 8 simply concatenates the unsorted S-expanded matrices into a list and then uses a zip function to sort the list based on the provided indices from algorithm 7.

Algorithm 8: CPU Implementation of S-expansion Reassembling Algorithm

1: Begin CPU Reassembling of Unique Dimensioned Hyperplane Terms

2: Input: S_expanded, idx

3: assembled_list = empty list of length S_expanded

4: For ii in range(dim[0] – 1) # dim is the number of unique dimensions from algorithm 7

5: assembled_list = list(S_expanded[ii+1])

6: **end For**

7: return [x for _, x in sorted(zip(idx, assembled_list))]

9: end CPU Reassembling

3.9.2 Using CUDA Streams for Asynchronous Data Transfers

After CPU refactoring to group matrices of like dimensions, this work leverages CUDA Streams for course-grained concurrency. One of the largest overheads for GPU programming, outside of a kernel launch itself, is the bottleneck induced by a data transfer to the GPU. Streams allow asynchronous transfer of data as well as kernel launches. Rather than serialize each transfer to the GPU and then invoking the GPU for a computation, it is much more prudent to further parallelize the calls to the hardware when data does not depend on each other's results. This is the case for the enumerated faces across each term in hyperplane arrangements. Figure 10 helps to visualize how a CUDA stream changes the way the GPU is invoked over a serializing the calls.

Serialized Calls Between CPU and GPU

CPU				
Stream 0				

Asynchronous Calls on Multiple CUDA Streams

CPU					
Stream 0					
Stream 1					

Time

Figure 10: The time it takes to complete operations on the GPU and CPU for the serialized versus streamed version is shown above. Note the time saved when staggering calls.

Algorithm 9: CUDA Streams in the Context of the Entire Algorithm

1: Begin Full Kernelized S-Expansion

2: Input: B #from Reverse Search

3: stack_Bs, idx = **B_sort**(B) #call algorithm 7

4: **For** ii **in** range(len(stack_Bs)): #use **CUDA STREAMS** here

5: **S_GPU[ii] = S_expanded_GPU**(stack_Bs, mask)

6: S_pseudo[ii] = **get_least_norm_GPU**(S_GPU[ii])

7: **end For**

8: S_final = B_reassemble(S_GPU)

9: S_pseudo_final = B_reassemble(S_pseudo)

10: return S_final, S_pseudo_final

11: End Kernelized S-Expansion

3.9.3 The S-expansion Kernel – Introduction and Thread-Blocks Chosen

After coalescing matrices of like dimensions and setting up the kernel for CUDA streams, the algorithm then moves into the actual kernel call. This work sets 1024 threads (the maximum amount of possible threads at the time of writing) at launch and found no considerable difference when experimenting with this value. That said, threads should launch in pairs of 32 as a best practice approach since the hardware automatically synchronizes them, thus maximizing speed. Note that the number of threads equals the chosen threads in the x times the threads in the y, times the threads in the z. Thus, 32 threads in the x and 32 threads in the y would reach the thread limit.

The kernelized S-expansion allocates the number of threads in the x direction as the number of enumerated faces and the threads in the y direction as the number of possible combinations. If the number of faces multiplied to the number of combinations is greater than the chosen number of threads, then the algorithm allocates additional blocks to complete the rest of the expansion. Higher measurement counts with the Cauchy Estimator necessitates this since the hardware limits the thread count at launch to 1024 [30].

The number of threads in the x are chosen using the formula:

$$\boldsymbol{Tx} = floor\left(\frac{\#\ Chosen\ Threads}{number\ of\ combinations}\right);\ \boldsymbol{Ty} = number\ of\ combinations$$

Where Tx and Ty are the threads in the x and y directions, respectively. Next, if Tx*Ty exceeds 1024, then the algorithms allocates additional blocks as such:9

$$\boldsymbol{total\ threads} = Tx * Ty$$

$$\boldsymbol{threads\ needed} = number\ of\ faces * number\ of\ combinations$$

$$\boldsymbol{Bx} = number\ of\ expansions\ required\ ;\ \mathbf{Bz} = ceil\left(\frac{threads\ needed}{total\ threads}\right)$$

Where Bx and Bz are the blocks required in the x and z, respectively. Bz captures how this work stacks the enumerated faces per hyperplane arrangement along the third dimension.

Recall that each thread runs simultaneously or launches their own instantiation of the kernel. Each block contains the set of (at most) 1024 threads. Thus, the total number of operations happening at any given time here is equal to the number of threads multiplied to the number of blocks. However, blocks also have a limit; blocks launched in the y and z direction are relegated to 65535 blocks while blocks in the x can reach as much as 2^{31}-1 for an NVIDIA 1080 Ti. These numbers vary depending on the hardware's compute capability [30]. Thus, the kernel allocates the blocks in the x direction to the third dimension in a matrix stack for computation since the estimator will exceed 65535 blocks for a particular hyperplane arrangement by the 9th measurement.

It is worth noting that although the hardware can technically launch 2^{31}-1 * 65535 *65535 blocks, and with it, 1024 threads per block (for an unbelievable amount of simultaneous operations), the GPU memory will likely saturate before leveraging that many thread-block pairs. Lastly, a device with a compute capability of 6.1, such as the 1080 TI, can launch at most 32 grids (each of which contain a thread-block pair). However, this work does not leverage parallelism on the grid level.

3.9.4 The S-expansion Kernel – Algorithm Design

The kernel is separated into 1 global main function called from the host side and 2 device functions that the GPU calls on the device side. The global function's task simply allocates the threads and blocks into variables which the kernel leverages to point (index) the matrix containing the enumerated hyperplanes. It also determines whether to distribute blocks or whether a single block is sufficient to complete the expansion; this explains the purpose of the 2 device functions. For this implementation, the kernel requires 9 inputs: the enumerated hyperplanes, a mask which tells the algorithm how to expand the enumeration, an empty matrix to allocate the results of the expansion, a temporary matrix containing 1's with the same dimensions as this expanded matrix,

the number of combinations, number of faces, the number of hyperplanes, the dimension of the problem, and the enumeration matrix stack depth.

Assume that the global function calls the kernel requiring additional blocks (the smaller S expansion has the same logic but a very slightly different indexing scheme). First, the threads are checked to insure overallocation does not occur, resulting in undefined behavior in the GPU. Then, a loop over the dimension of the problem occurs. Because the dimension of the problem here is 3, the loop does not result in a significant bottleneck. Then, each thread in the y direction simultaneously points to a different row in the mask, as shown in figure 11.

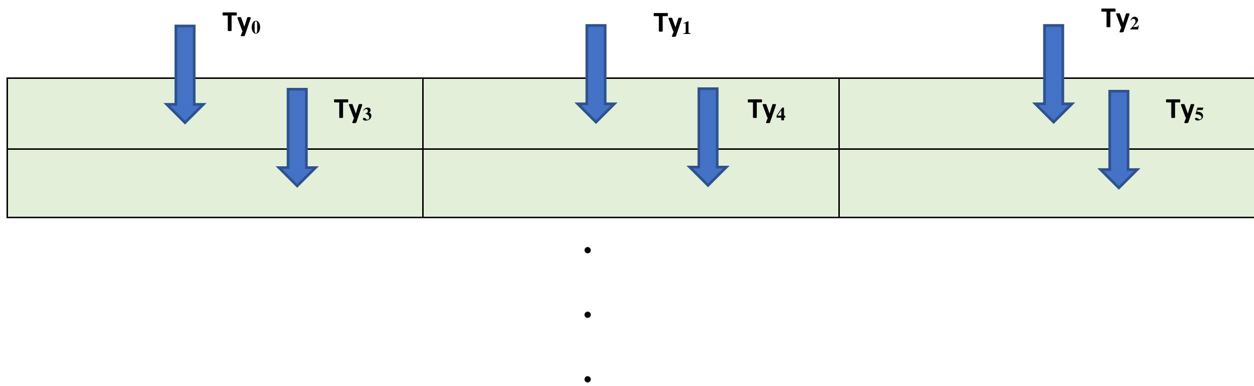

Figure 11: Visualization of how each thread simultaneously operates to access a value in memory

Like the CPU implementation, the GPU checks whether the algorithm should proceed with an expansion. In this case, a -1 indicates skipping the current set of faces. Moreover, the indexing scheme is drastically different than how a typical serial implementation occurs. This changes based on individual implementation. However, the key thing to note is that CUDA passes variables as 1-D, row-wise pointers [7]. This work's indexing scheme is noted in the algorithms section. As an example, masks indexes using ty*#dimensions + k. This results in the following:

	For k = 0	**For** k = 1	**For** k = 2
Ty = 0	0	1	2
Ty = 1	3	4	5
Ty = 2	6	7	8

Table 1: Indexing values for masks. Note that each row in the table happens concurrently whereas a loop covers each column serially.

After the conditional passes, the expansion occurs. Note that in this the enumerated hyperplanes exist on global memory where each thread in each block has access to. Similarly, it writes to the temporary matrix containing 1's before writing again to the empty matrix, all of which exist in global memory. Each thread, when checking the mask variable also reads from global memory. The author would like to note that it be more efficient to allocate the enumerated hyperplanes (results from reverse search) to local memory, masks to shared memory, and temporary matrix of 1's to shared memory before coalescing results to the empty matrix in global memory. This is possible because of the independence of each hyperplane from each other, and masks being the same for every enumeration. Thus, the reported results can potentially improve significantly due to the approximately 100x efficiency gain when accessing shared memory. To be clear, this does not mean the speed of the overall program improves by a factor of 100. Rather, the latency of accessing shared memory is approximately 100 times faster when compared to accessing global memory [24]. The full kernel along with the pseudoinverse of the expansion is delineated further in algorithm 10 along with a high-level flow of the code.

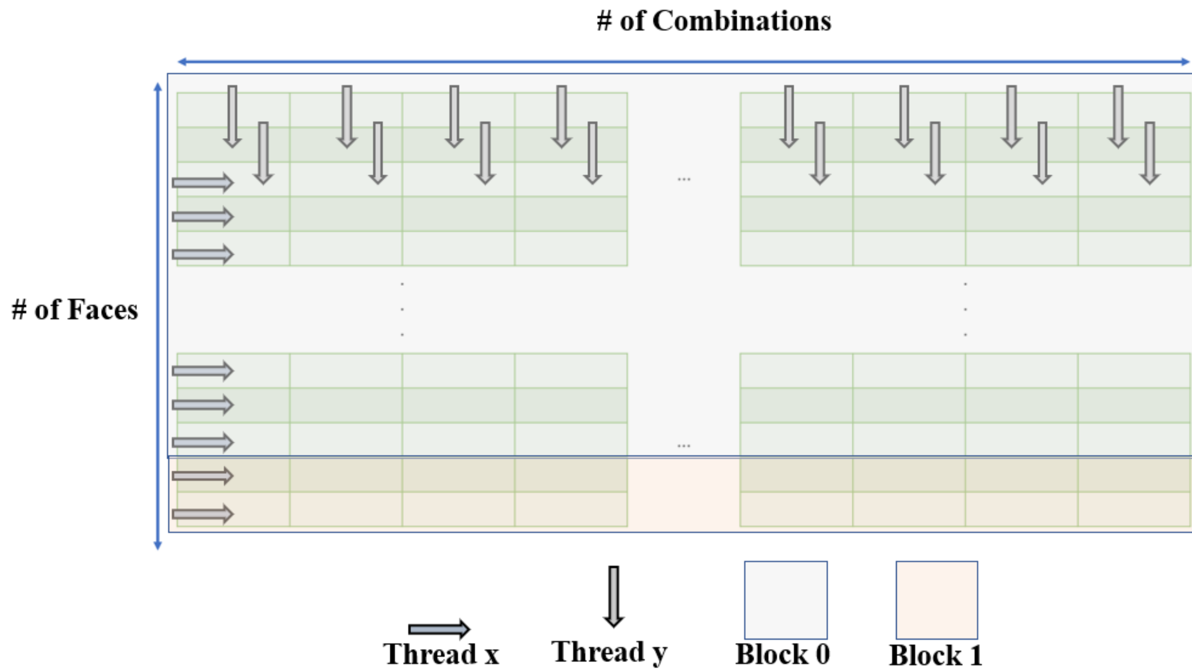

Figure 12: Example of how thread-block pairs operate on the expanded matrix, S. Note that each thread runs a copy of the kernel and that there exists many 3-D Tensors.

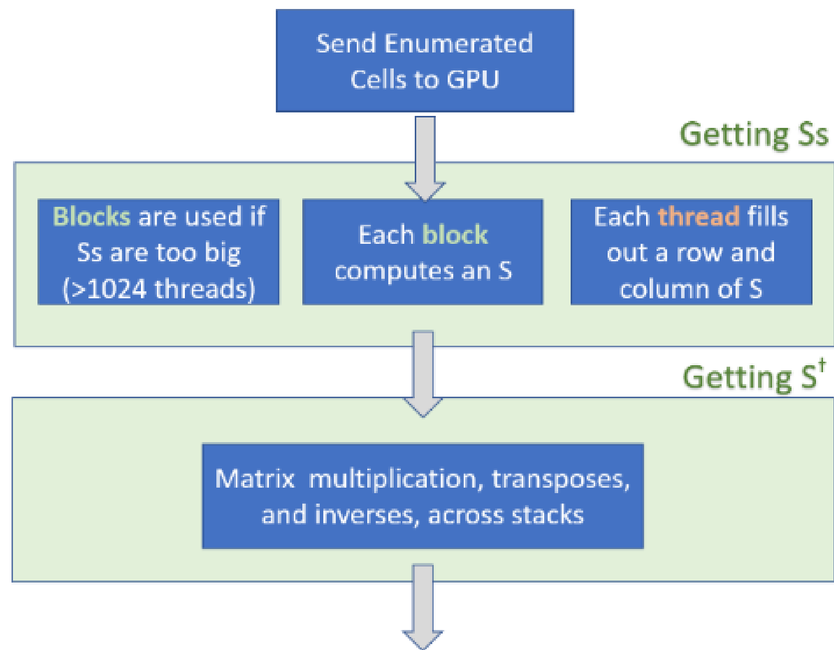

Figure 13: Sending the enumerated matrices to GPU can be done asynchronously through CUDA streams

Algorithm 10: S-Expansion Kernel

```
1: Begin GPU S-Expansion //note that Pseudocode now reflects C language

2: extern "C" __global__

3: void getS_kern_main(float *B, float *masks, float *S_GPU, float *S, int NC, int NF, int B_col,
int ndim, int stackDepth)
4: {
5:      //BlockDim.x,y,z, gives num of threads in a block , in particular direction (fixed scalar)
6:      // gridDim.x,y,z gives num of blocks in a grid, in a particular direction (fixed scalar)
7:      // threadidx.x gives the thread indices themselves (changes)

8:      int Tx ← threadIdx.x; , int Bx ← blockIdx.x; , int Bdx = blockDim.x;
9:      int Ty ← (threadIdx.y + blockIdx.y*blockDim.y); // initialize thread block indexing scheme
10:     int Tz ← (threadIdx.z + blockIdx.z*blockDim.z); , int Bz ← blockIdx.z;
12:     // blockDimx gives the proper thread count (max thread count)

13:     // Each thread runs a copy of the kernel
14:     if (bx < 1){ //i.e required thread count is less than 1024

15:             S_GPU[bz*NC*NF + NC*bdx*bx) < NF)  = getS_small(B, masks, S, tx, bx, ty, tz,
                                                     Bz, NC, NF, B_col, ndim, stackDepth)

        }
16:     Else if (Bx > 0 && (Tx + Bdx*Bx) < NF){
                S_GPU[bz*NC*NF + NC*bdx*bx + tx*NC + (ty+1)] = getS_slack(…) //same as 15
        }
17: }
18: End GPU S-Expansion
19: //In this implementation, the Bz axis is used to contain the third dimension of the expanded S-
20: //Matrix. This was done to support intuition. It is important to note that at the 9th measurement,
21: this can  //exceed the maximum block size of 65534. This can be resolved by replacing Bz with
22: Bx and make according changes
```

Algorithm 10: S-Expansion Kernel – Device Functions for < 1024 Threads

```
1: Begin GPU S-Expansion for < 1024 threads
2: extern "C" __Device__
3: float getS_small(//Note the inputs in line 16 of the above algorithm) {
4: //NC = number of combos (columns of S) and rows of mask
5: //NF = number of faces (rows of S) and rows of B
6: //Prevent over indexing with following conditional
7:     If( Tx < number of faces ) and (Ty < number of combos – 1) and (Tz < stack depth){
8:         For (int k = 0; k < state dimension; k++){
9:             __syncthreads();
10:            if(masks[Ty*state dimensions + k] != -1){ //all threads in y check a column
11:                int idx = masks[Ty*state dimension+k];
12:                S[bz*NC*NF+tx*NC + (Ty+1)] *= B[bz*B_col*NF+tx*B_col+idx];

13:            }Else If{
14:                S[Bz*NC*NF+Tx*NC + (Ty+1)] *= 1;
15:            }
16:        }
17:    }
18: return(S[Bz*NC*NF+Tx*NC + (Ty+1)]);
19: }
20: End getS_small
```

Algorithm 10: S-Expansion Kernel – Device Functions for > 1024 Threads

```
//Difference is the use of the block indices as pointers to index the remaining expansions.
1: Begin GPU S-Expansion for < 1024 threads
2: extern "C" __Device__
3: float getS_slack(//Note the inputs in line 16 of the S-expansion algorithm) {
4: //NC = number of combos (columns of S) and rows of mask
5: //NF = number of faces (rows of S) and rows of B
6: //Prevent over indexing with following conditional
7:      If( Tx < number of faces ) and (Ty < number of combos – 1) and (Tz < stack depth){
8:           For (int k = 0; k < state dimension; k++){
9:                __syncthreads();
10:               if(masks[Ty*state dimensions + k] != -1){ //all threads in y check a column
11:                    int idx = masks[Ty*state dimension+k];
12:                    int indx_scheme = Bz*B_col*NF + B_col*Bdx*Bx + Tx*B_col + idx;
13:                    S[bz*NC*NF+NC*Bdx*Bx + Tx*NC + (Ty+1)] *=B[indx_scheme];
14:               }Else If{
15:                    S[Bz*NC*NF+ NC*Bdx*Bx + Tx*NC + (Ty+1)] *= 1;
16:               }
17:          }
18:     }
19: return(S[Bz*NC*NF+ NC*Bdx*Bx + Tx*NC + (Ty+1)]);
20: }


20: End getS_slack
```

3.9.5 The S-expansion Kernel – Computing the Pseudoinverse

At the time of writing, CuPy did not allow a singular value decomposition across a 3-D stack of matrices. However, transposes and matrix multiplications is possible. Thus, to compute the pseudoinverse of the expanded S, this work uses the following from Laub [1]:

Theorem 1:

$$A^{\dagger} = \lim_{\delta \to 0} A^T (AA^T + \delta^2 I)^{-1} \qquad (3.9)$$

Taking the limit as δ approaches 0 simply results in:

$$A^{\dagger} = A^T (AA^T)^{-1} \qquad (3.9.1)$$

Equation 3.9.1 is implemented on the GPU and is much more suitable to parallel computation. This is also possible because coalignment removed redundant hyperplane arrangements, resulting in linearly independent rows. This is a necessity condition for equation 3.9.1 [1].

CHAPTER 4

Experiments and Results from Parallelization

4.1 Simulation Setup and Testing

The performance of ACE was tested numerically in Python 3. This algorithm considers a 3-state system with the following parameters:

$$\Phi = \begin{bmatrix} 1.4 & -0.6 & -1.0 \\ -0.2 & 1.0 & 0.5 \\ 0.6 & -0.6 & -0.2 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} 0.1 \\ 0.3 \\ -0.2 \end{bmatrix}, \quad \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.08 \\ 0.05 \end{bmatrix}$$

$$H = [1.0 \quad 0.5 \quad 0.2], \qquad \beta = 0.1, \qquad \gamma = 0.2.$$

This system has stable eigenvalues at 0.7 ± 0.3j and 0.8. It is also observable and complies with a necessary condition for the estimator that $H\Gamma \neq 0$ [22].

For timing results, each algorithm within the Cauchy Estimator was independently parallelized and timed using CUDA events. CUDA events utilizes the concepts of streams and records a time stamp for the piece of GPU code between a designated start and stop. This methodology has a resolution of approximately one-half microsecond and avoids the slight overhead often associated with using CPU timing methods [22]. That said, serial implementations of the Cauchy Estimator are timed using CPU timing methods.

When appropriate, data transfers are also included in the GPU timing results to consider the worst-case scenario. Moreover, the number of terms for $i \in [1, 2, \ldots n_t]$ for each hyperplane arrangement is also reported since that contributes greatly to the algorithm's performance. Lastly, simulations were conducted on a Ryzen 1700x with a 1080 Ti, and 32 Gb of RAM at 3200 Mhz.

The timing results are summarized in table 2. Note that measurement one is skipped because the first step is a different formulation than the generalized time propagation of k+1|k and measurement update of k+1|k+1. Also, note that the table shows a comparatively long time for measurement 2, both for the CPU and the GPU. That is because the just-in-time (JIT) must compile the code on the first invocation before accelerating the subsequent runs [2]. Similarly, the GPU code must compile and link with NVIDIA's nvcc compiler the first time it's run [7]. Consequently, these results are discarded since they would skew the timing far in favor of the GPU and can be initialized at the start of the algorithm.

4.2 Time Propagation

The time it takes to complete the time propagation for each measurement is coalesced below. Note that the second measurement has been omitted in the plotting since it would skew the results largely in favor of the GPU. The results here demonstrate an almost linear scaling for the GPU whereas by the 7th measurement, the CPU begins to increase exponentially in time.

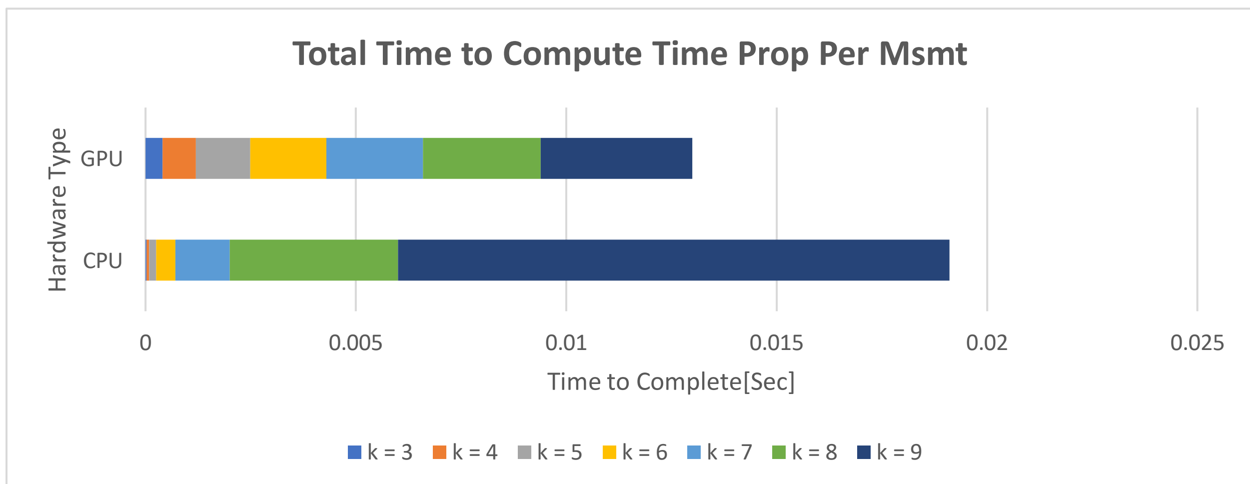

Figure 14: The total time it takes to finish 9 measurements, excluding measurement 1 and 2, is shown. Notice that the GPU outpaces the CPU by about the 7th to 8th measurement.

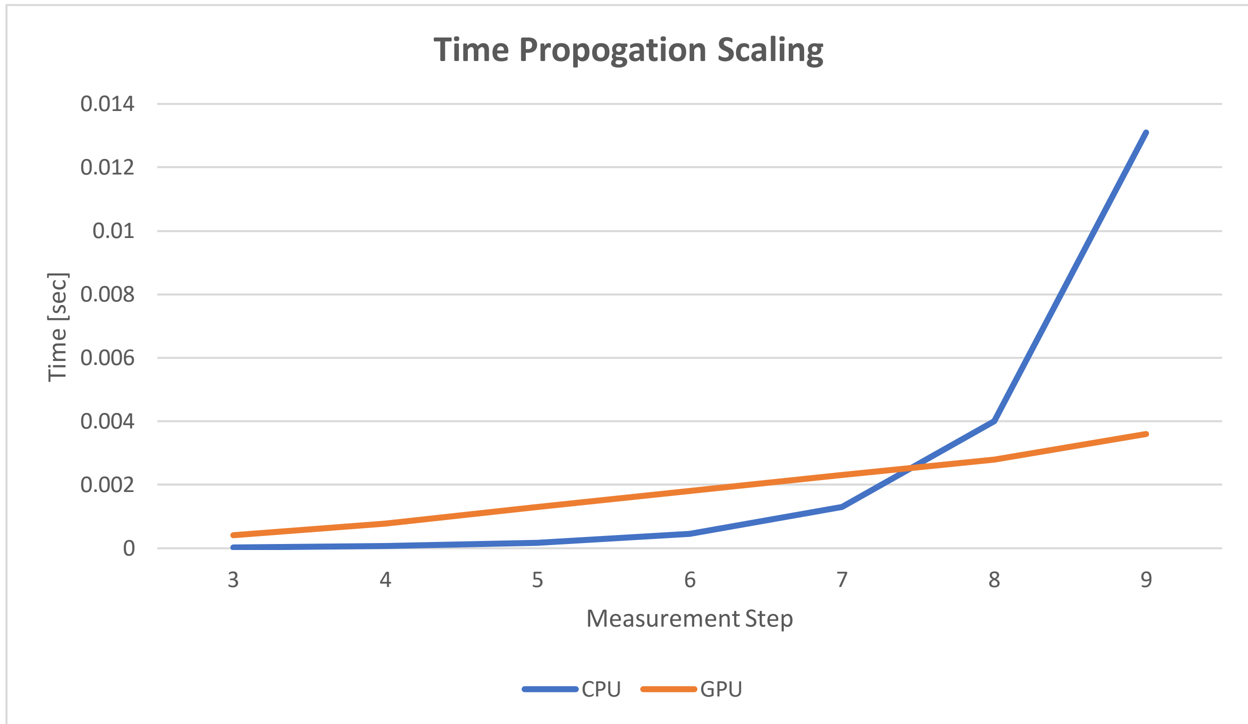

Figure 15: Results demonstrate that this implementation of ACE has an almost linear scaling per measurement.

Measurement Number	Number of Hyperplanes	CPU (seconds)	GPU (seconds)	Speed Improvements
2	**16**	0.718	0.127	**N/A**
3	**70**	2.58E-05	0.00041199	0.062502
4	**276**	0.000065804	0.00078273	0.08407
5	**1035**	0.0001626	0.0013	0.125077
6	**3704**	0.00044847	0.0018	0.24915
7	**12888**	0.0013	0.0023	0.565217
8	**85698**	0.004	0.0028	1.428571
9	**148740**	0.0131	0.0036	3.638889

Table 2: Tabulated timing results for each measurement step, including the number of hyperplanes that ACE must operate on during time propagation. Note that the speed improvements are relative to the CPU. A result of <1 means the GPU was slower by that factor.

4.3 Measurement Update

Like the time propagation, the results for the measurement update are summarized below while omitting the second measurement in the plotting routine. Similar to the time propagation, the measurement update also sees an almost linear scaling per measurement. The exception, however, is the ninth measurement where the time to completion seemingly doubles.

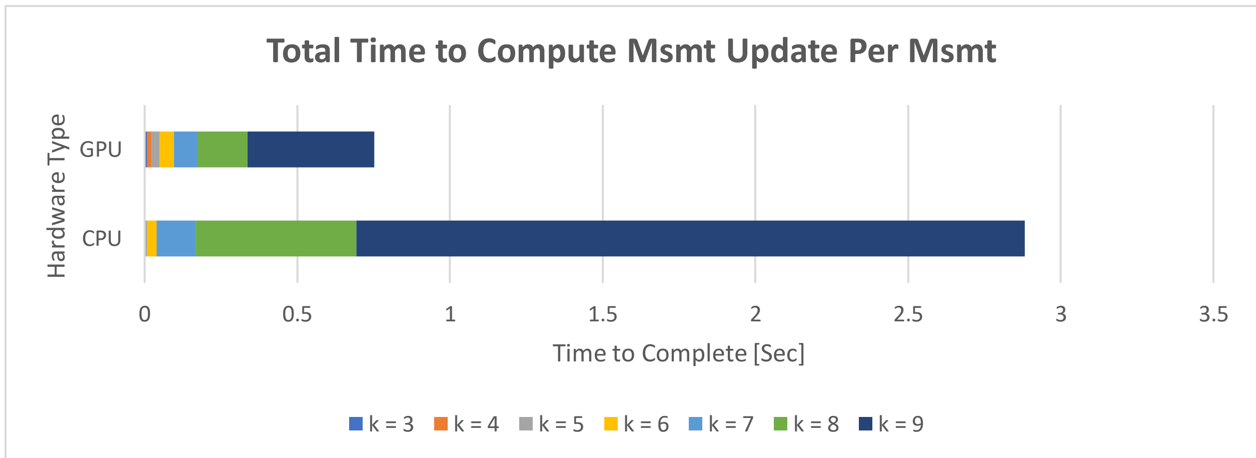

Figure 16: The total time it takes to finish 9 measurements, excluding measurement 1 and 2, is shown. Notice that the GPU outpaces the CPU by about the 6th measurement.

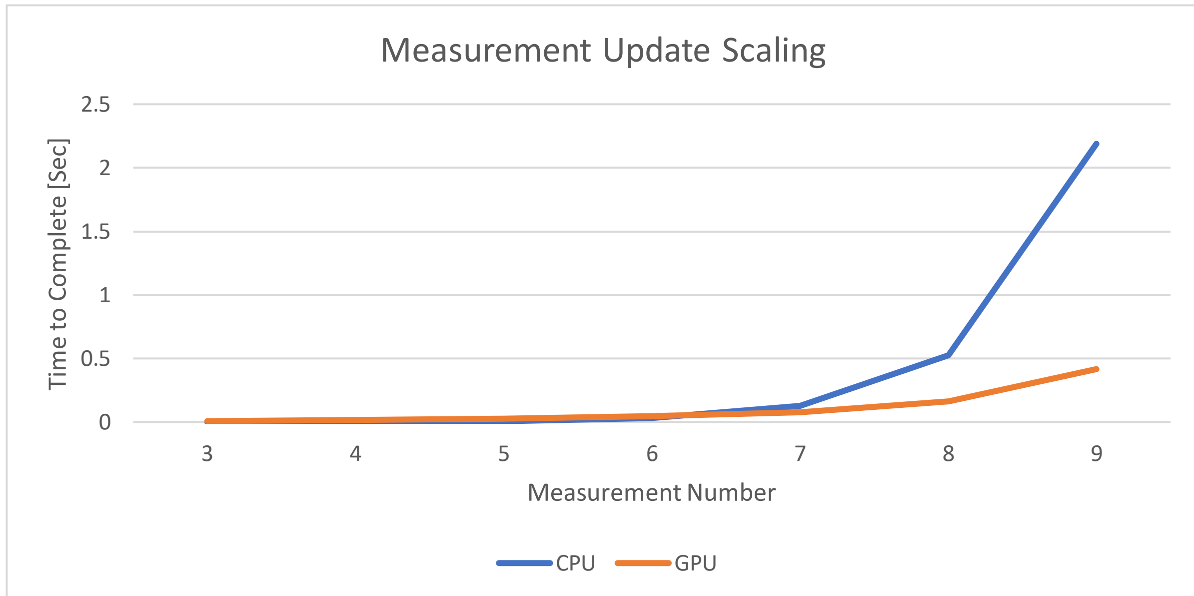

Figure 17: Results demonstrate that this implementation of ACE has an almost linear scaling per measurement until approximately the 9th measurement, which almost doubles in runtime.

Measurement Number	Number of Hyperplanes	CPU (seconds)	GPU (seconds)	Speed Improvements
2	**80**	4.975	0.0175	**N/A**
3	**420**	3.23E-04	0.0073	0.044288
4	**1896**	0.0016	0.0161	0.099379
5	**7790**	0.0067	0.0263	0.254753
6	**30190**	0.0302	0.0453	0.666667
7	**112294**	0.1278	0.0787	1.623888
8	**405836**	0.5259	0.1624	3.2383
9	**1437320**	2.1888	0.4164	5.256484

Table 3: Tabulated timing results for each measurement step, including the number of hyperplanes that ACE must operate on during measurement update.

4.4 Computing G, Y_{ei}

Computing the G and Y_{ei} is partitioned out into their own routine to help aid readability. For the most part, the results of this portion of the algorithm displays similar characteristics to the measurement update. Here, the run time for the ninth measurement grows considerably over the prior steps.

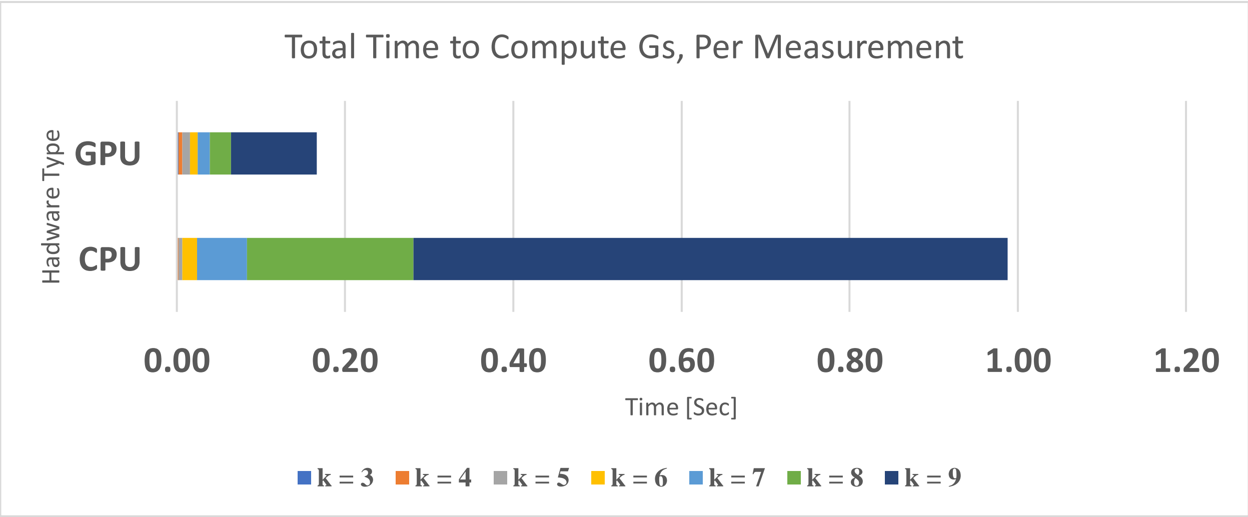

Figure 18: The total time it takes to finish 9 measurements, excluding measurement 1 and 2, is shown. Notice that the GPU outpaces the CPU by about the 6th measurement.

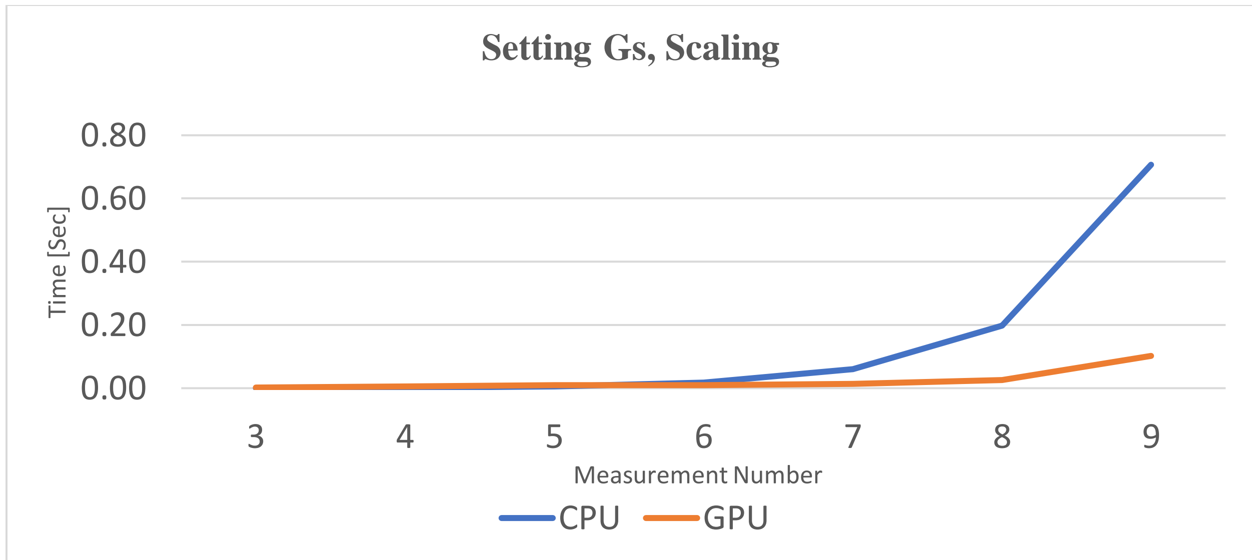

Figure 19: Results demonstrate that this implementation of ACE has an almost linear scaling per measurement until approximately the 9th measurement, which almost doubles in runtime.

Measurement Number	Number of Hyperplanes	CPU (seconds)	GPU (seconds)	Speed Improvements
2	**80**	5.3103	0.0133	**N/A**
3	**420**	4.00E-04	0.0021	0.190476
4	**1896**	0.0013	0.0046	0.282609
5	**7790**	0.0049	0.0088	0.556818
6	**30190**	0.0171	0.0096	1.78125
7	**112294**	0.06	0.0142	4.225352
8	**405836**	0.1975	0.0253	7.806324
9	**1437320**	0.7068	0.1021	6.922625

Table 4: Tabulated timing results for each measurement step, including the number of hyperplanes that ACE must operate when computing G's.

4.5 Term-Coalignment

GPU Term -Coalignment is almost incomparable to its serial counterpart. However, timing results do not fully capture the efficacy of parallelization for this section of the algorithm. Results for how many terms were coaligned on the GPU versus the CPU is also presented. Note, that the GPU had a closeness measure of 1e-7 due to the imprecision of operating on floating point 32 variables. This further explains why the GPU would coalign more over the CPU.

The results are presented as a logarithmic scaling since it gives a better sense of the scale that GPU programming outpaces the CPU for this custom kernel.

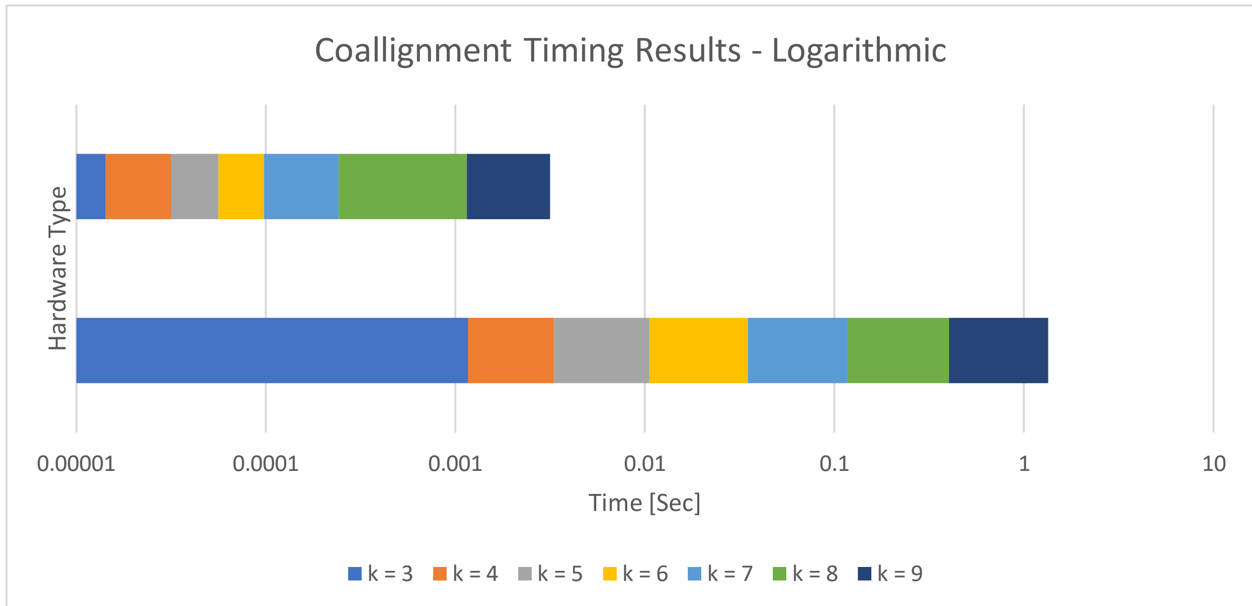

Figure 20: Results demonstrates that custom kernelization of coalignment completes 3 to 9 measurements in the same amount of time that the CPU completes 3 to 4 measurements.

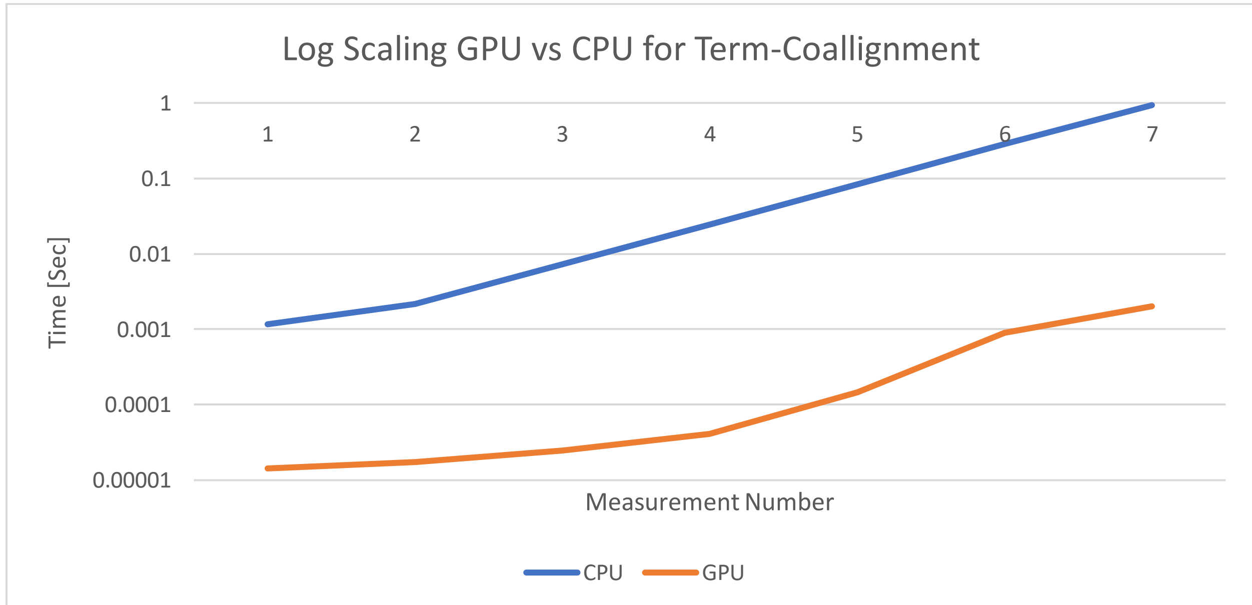

Figure 21: The logarithmic scaling demonstrates several orders of magnitude speed improvement per measurement

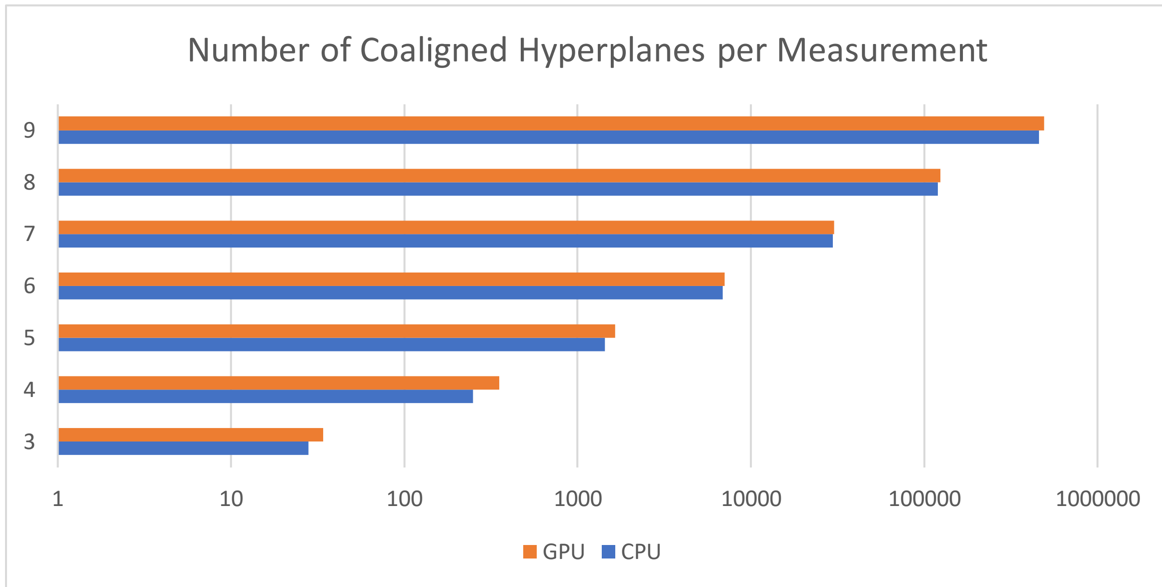

Figure 22: The logarithmic scaling showcases how many hyperplanes coaligned per measurement for both the GPU and CPU. For the chosen closeness measure, the hyperplanes consistently coaligned more on the GPU.

Measurement Number	Number of Hyperplanes	CPU (seconds)	GPU (seconds) FP(32)	GPU (seconds) (FP64)	Speed Improvements (FP32)	Speed Improvements (FP64)
2	**80**	3.788115501	0.00040122	0.000394	**N/A**	**N/A**
3	**420**	0.00116539	1.54E-05	1.54E-05	**75.71401**	**75.87175**
4	**1896**	0.002156019	1.64E-05	1.84E-05	**131.593**	**116.9715**
5	**7790**	0.007196665	2.25E-05	2.56E-05	**319.4542**	**281.4716**
6	**30190**	0.024286985	4.43E-05	4.48E-05	**547.9916**	**541.7333**
7	**112294**	0.082619429	0.00014848	0.000157	**556.4347**	**527.3401**
8	**405836**	0.285746336	0.00051814	0.001097	**551.4805**	**260.5501**
9	**1437320**	0.934037924	0.00182579	0.001871	**511.5796**	**499.2591**

Measurement Number	# Coaligned (CPU)	#Coaligned (GPU)	Difference
3	28	34	6
4	250	352	102
5	1433	1650	217
6	6861	7028	167
7	29630	30146	516
8	119638	124356	4718
9	461092	492514	31422

Table(s) 5 and 6: Note that measurement 2 is not shown because there were no coaligned hyperplanes for the given dynamics. The difference generally increases per measurement step.

4.6 Parallel S-Expansion and Computing the Least Norms

Recall that the S-expansion occurs right after evaluating the enumerated hyperplanes using Reverse Search. Occasionally, this kernelized function is also called throughout the estimator. For example, invoking this routine to parallelize the expansion per arrangement in the measurement update or when computing the G's is possible.

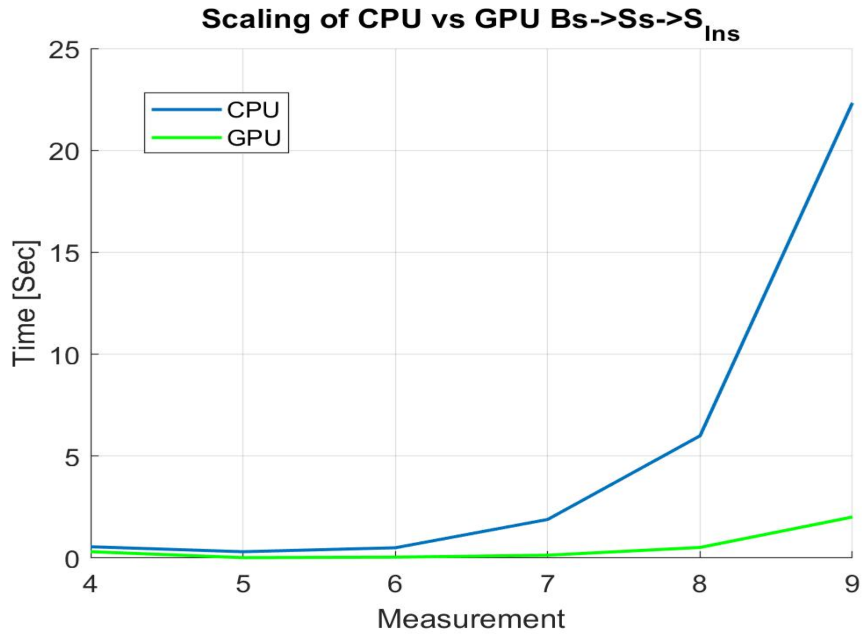

Figure 23: Note that measurement 1 to 3 is not shown because this kernel only becomes viable by the fourth measurement. Results include GPU data transfers and pseudoinverse calculations.

Measurement Number	CPU (seconds)	GPU (seconds)	Speed Improvements
4	0.5535	0.3112	1.7786
5	0.3104	0.0202	15.3663
6	0.5043	0.0449	11.2316
7	1.8930	0.1393	13.5894
8	5.9965	0.5198	11.5362
9	22.3283	2.0127	11.0937

Table 7: The timing results of the expansion is given. Note that when this data was collected, the 4th measurement was the first call to this function, so it is slower than the subsequent runs.

CHAPTER 5

Discussion and Conclusion

5.1 Time Propagation

The parallel time propagation computed using CuPy's built in functions performs well, demonstrating an almost linear scaling across each measurement. On the other hand, the jitted CPU implementation begins to scale exponentially and quickly becomes unviable at larger measurements. This makes sense considering the time propagation consists of matrix multiplications with readily scales across the GPU. However, while the number of operations can scale, there are additional calls to the GPU time propagation function because of the additional hyperplane arrangements after every measurement update.

However, note that the time propagation has a consistent dimension across its columns for parameters like A, the hyperplanes. Although the number of rows grow, the columns always consists of the dimension of the states. Thus, time propagation could simultaneously operate on all hyperplane arrangements at once, versus calling the GPU implementation each time. Alternatively, if multiple GPU's were used, the designer could also allocate the arrangements to each GPU or stream multiple kernel calls if memory footprint permits.

5.2 Measurement Update

For the measurement update, the parallel implementation begins to outpace the CPU implementation right around 7th measurement. Also similar to the time propagation, the measurement update exhibits an almost linear scaling until the 9th measurement where it almost doubles in compute time. As it turns out, an operation that repeats the α needed to compute the G's in the following routine is quite expensive due to creating memory copies; future work should

seek to implement this in a more efficient manner. Implementing the algorithm this way was simpler, but the computational overhead and scaling per measurement is likely not worth the trade off on higher measurements. Instead, a preallocation of appropriate α copies and a modification to get_S_gpu() (the kernelized S-expansion) to compute the same copies in parallel will likely be more efficient.

Although promising, the parallel implementation of the measurement update focuses on simultaneous computation across terms for each hyperplane arrangement. While this course level of parallelism should be maintained, the measurement update to each hyperplane arrangements is done serially across the CPU, just like the time propagation. To improve results, future work could focus on sending each arrangement to a separate GPU or instead, stack terms and arrangements across consistent dimensions so that a custom kernel could operate on a contiguous input. This will help improve the scaling of the measurement update considerably along with further speed improvements.

5.3 Evaluating G and Y_{ei}

Computing the G is very fast when compared to the measurement update. The parallel implementation does not begin to outpace the CPU implementation until the 6th measurement. Afterwards, the GPU implementation can compute 9 measurements in less time than the serial implementation can compute 8 measurements. Something worth pointing out is that the number of hyperplanes grow very quickly. From 80 in the second measurement to over 1.4 million by the 9th, the number of operations increases considerably. This partially explains why scaling towards the ninth measurement increases versus the almost linear scaling demonstrated in the first 8 measurements. This same reasoning also applies to the measurement update. Measurements 2 to 5 also demonstrates the quintessential problem of parallel programming; that unless there exists enough operations, the serial version will likely outcompete the GPU.

Again, parallelization happens across terms and not across hyperplane arrangements. Unfortunately, unlike coalignment, many of the operations like expanding the sign basis does not occur across consistent dimensions. As a result, serially computing each arrangement using parallel operations will likely have to occur unless the designer has access to multiple GPU's. Alternatively, overallocated arrays can be used to maintain a contiguous array for a custom kernel. Although memory inefficient, it would allow for simultaneous computing of G and Y_{ei} across hyperplane arrangements if only a single GPU is available.

5.4 Term-Coalignment

GPU term-coalignment was one of the last algorithms developed for this thesis and represents the coalescing of many GPU design techniques. This translated to many times speed improvement over the best CPU implementation at the time of this writing. The most important takeaway was getting creative with exploiting parallelism and recasting problems into a GPU implementable form in the case of the parallel-prefix sum.

Several other design techniques also contributed to the improvements, including using shared memory when possible, launching in batches of 32 threads, loop unrolling, and avoiding branching of conditionals when possible. However, the greatest improvement involved simultaneously operating across all hyperplane arrangements via a contiguous input. Although this might take some preprocessing to stack matrices in practice, the speed gains were well-worth the cost of these operations. It is worth noting that these speed improvements did not measure the aforementioned preprocessing done, just the time to execute the kernel itself.

This also motivates stacking or operating across several arrangements in parallel in the case of the time propagation, measurement update, and computing of G. In fact, when looking at the number of hyperplanes that had to be checked for coalignment, the same increase of 80 to 1.4 million terms can be seen. However, speed improvements were consistently large and time to

completion scaled well due to the ability to complete simultaneous operations across the growing number of hyperplanes. Moreover, the second table demonstrates that the GPU consistently coaligned more over its serial counterpart. The parallel implementation has the distinct advantage that when more hyperplanes have to coalign, the combining of the coefficients in the exponential, p and q, can also occur in parallel using coalesced memory access. As a result, the algorithm continues to scale if more hyperplanes must coalign whereas the serial implementation has to perform an inner loop to combine the coefficients. This further slows the serial implementation as more hyperplanes begin to coalign.

The largest challenge with this implementation of coalignment was the fact that all GPU operations were done using floating point 32. As a result, the precision of all results could only be matched to the 7th or so decimal place. This applies to all previously discussed GPU algorithms. This causes coalignment to remove more or less hyperplanes over its serial counterpart, depending on the closeness measured used. To circumvent this issue, future work should leverage GPU's designed for high precision scientific computing, like the NVIDIA Tesla, to maintain speed improvements over double precision and to pick an appropriately lower closeness measure.

5.5 Parallel S-Expansion and Computing the Least Norms

The parallel S-expansion and computing of the pseudoinverse also demonstrates appreciable speed improvements over the serial implementation. Further, it maintains the speed improvements of over 10x across higher measurements. Note, however, that the results include the time it took to perform data transfers to the GPU, the time to assemble and reassemble hyperplane arrangements using the CPU, as well as computing the pseudoinverse. Data transfers are very expensive here because the number of enumerated hyperplanes grow considerably across each

measurement and rearranging does not take a trivial amount of time as well. This necessitates parallelization of reverse search if ACE wants to avoid this data transfer.

Future work for this algorithm should focus on leveraging shared memory. In this work the expansion accesses and writes to global memory across all enumerated planes. This is unnecessary since expansions for each face is independent from one another and can be done at the block level using shared memory. However, the largest speed improvements and the greatest impact to scaling will come from eliminating the need to reorder the hyperplane arrangements and instead conduct as many simultaneous expansions as possible.

5.6 Implementing the GPU With a Sliding Window

Although not presented in detail for this thesis, early work to implement the GPU with a sliding window of measurements was conducted to validate a general paradigm for high performance computing (HPC). In particular, each core in the CPU can run its own instantiation of the estimator and process several measurements. The results of these measurements are then carried over and used in a different window once it reaches the appropriate measurement. For example, if core 0 finishes 6 measurements, then core 1 would be at 5 measurements. Once core 1 reaches 5 measurements, it has the results from core 0 to process the next step and so on. At the time of this writing, only the parent process can leverage the GPU without using methods from Python's multiprocessing library to initialize the device for each instantiation (child) of the estimator [34]. Although this is the simplest way to do this, this requires n-times the memory consumption of the GPU where n is the number of sliding windows. Notwithstanding this bottleneck, it is nevertheless possible to leverage the GPU for each sliding window. Alternatively, each window could have access to its own GPU. This paradigm of using the GPU per window or separate GPU's per window is illustrated in 23 and 24, respectively. Lastly, only the parent can

leverage the GPU and child processes can send required parameters for processing on the GPU in the parent. However, the overhead of the transfers however might negate any speed benefits gained from parallelization in the first place. This paradigm is illustrated in Figure 25.

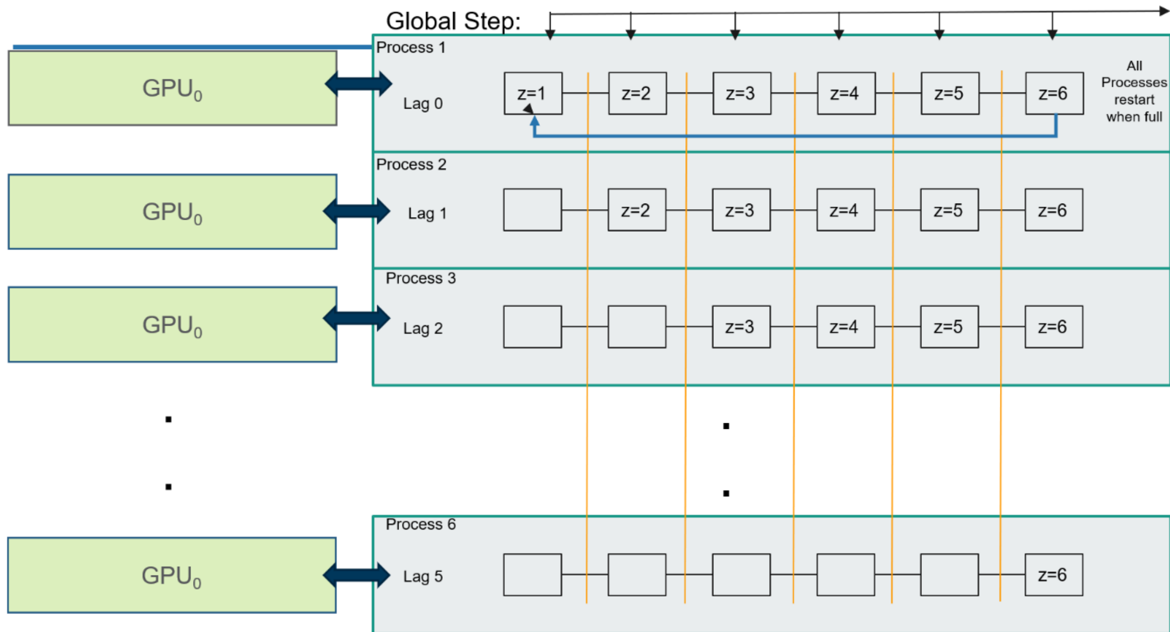

Figure 24: Each child process may leverage the same GPU using Python's multiprocessing library.

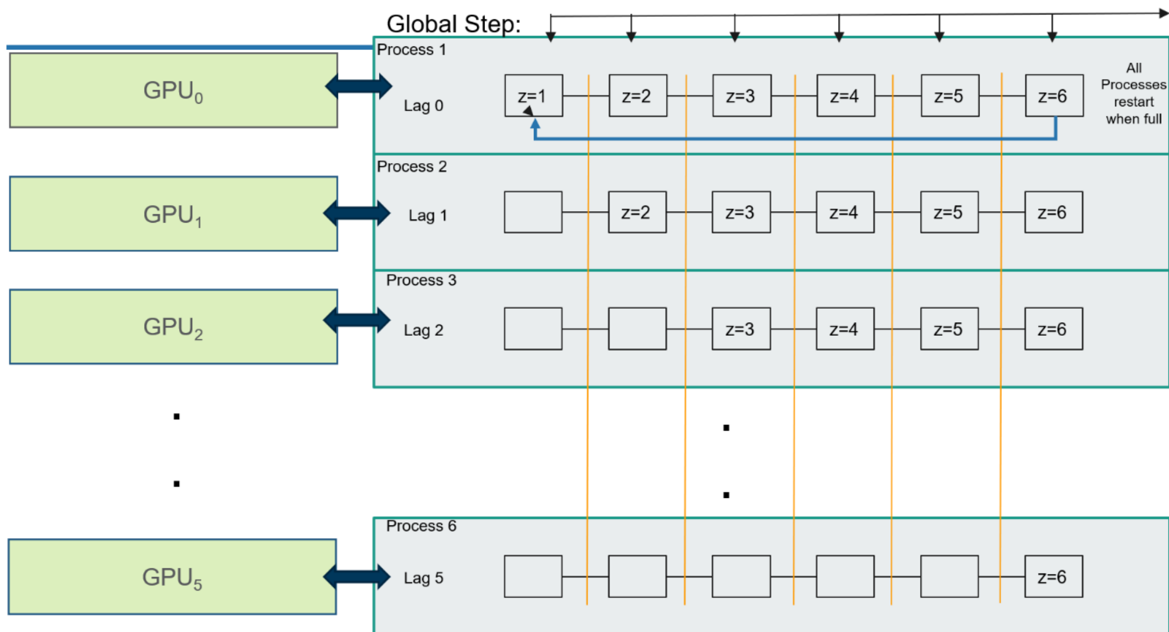

Figure 25: Alternatively, each process that computes the sliding window can have its own GPU. Image Credit: N. Snyder.

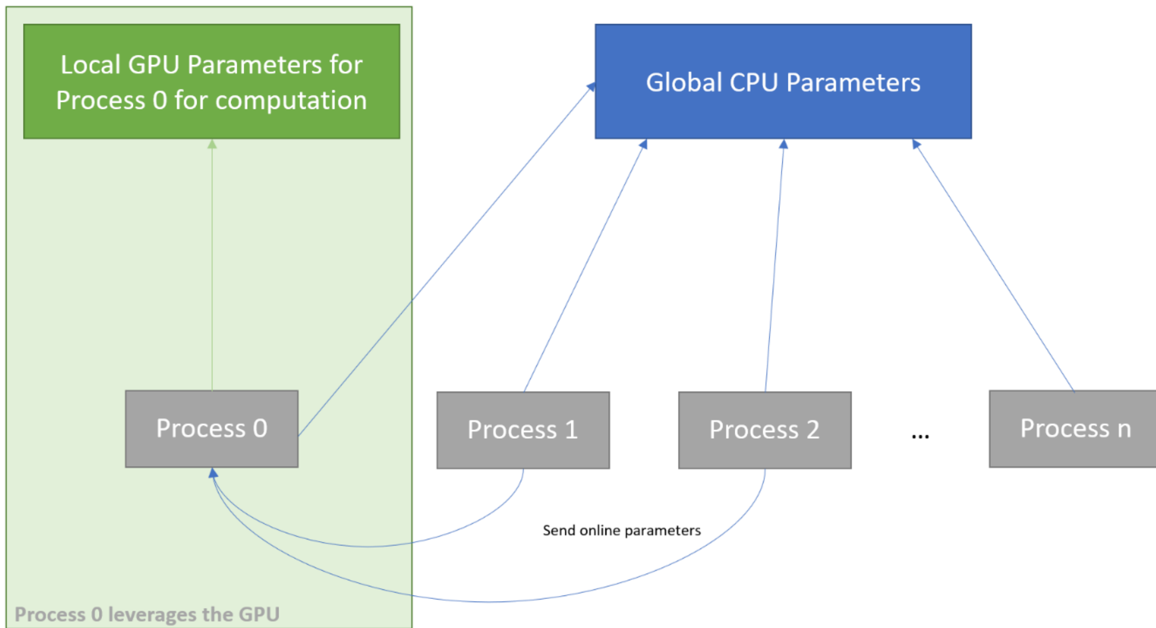

Figure 26: It is currently not possible for there to be a global GPU parameter space at the CPU level. Instead, one process may have access to local GPU parameters for computation and Process 1-to-n can send parameters to process 0 to accelerate computation, as necessary.

5.7 Conclusion

This thesis presents a parallelization paradigm for several key routines for the linear discrete-time system for the n-state Cauchy Estimator. Although the algorithms were written specifically for the 3-state estimator, they can also scale to the n-state estimator with modifications to how operations occur when the dimensions grow. Every subroutine demonstrates a speed improvement once the number of required parallel operations have grown past a certain amount. For the estimator, this takes place in the form of higher measurements.

It is shown that simultaneous operations across a contiguous array yields the greatest speed improvement (in the case of term-coalignment) and future work should focus on maximizing this level of concurrency as much as possible. At this stage of the algorithm, only the time propagation, measurement update, computing the G's, term-coalignment, and expanding the numerated planes have been parallelized. Three key routines remain: flattening, reduction, and reverse search. Before

fully realizing the Cauchy estimator, these routines must be completed as well. Nonetheless, these results demonstrate promise for leveraging HPC to accelerate the estimator.

Finally, a discussion on implementing the GPU in conjunction with a multicore GPU processing to continuously process measurements using a sliding window was provided. In this instance, the greatest bottleneck is memory footprint since each core must compute n-measurements each where n is the size of the sliding window. This problem can be circumvented by assigning each core of the CPU with its own GPU or reducing memory consumption in general. Nonetheless, early work has shown that aggregating the GPU and CPU for multilayer concurrency is possible and can be used to fully realize the estimator for a general class of problems.

CITATION

[1] A. Laub, *Matrix Analysis for Scientists and Engineers,* SIAM, USA, 2004.

[2] Anaconda, *Numba Documentation*, Release 0.51.2-py3.7-linux-x86_64.egg, Nov. 26, 2020, https://numba.readthedocs.io/en/stable/user/5minguide.html, Accessed, Nov. 2020.

[3] *Autopilot*, https://www.tesla.com/autopilotAI, Accessed Nov. 2020.

[4] *Basics of CuPy, https://docs.cupy.dev/en/latest/tutorial/basic.html,* Accessed Aug. 2020.

[5] B. Fulkerson and S. Soatto, *Really Quick Shift: Image Segmentation on a GPU*, ECCV 2010 Workshop, Part II, LNCS 6554, pp. 350 – 358, 2012.

[6] Brodtkorb, A.R., Hagen, T.R. and Saetra, M.L. (2013) Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing. Journal of Parallel and Distributed Computing, 73, 4-13. https://doi.org/10.1016/j.jpdc.2012.04.003

[7] B. Tuomanen, *Hands-On GPU Programming with Python and CUDA*, Packt Publishing Ltd. 2018.

[8] D. Avis, K. Fukuda, *Reverse Search for Enumeration*¸ Discrete Applied Mathematics, Vol. 64, Issues 1-3, pp 21-46, 1996.

[9] D. Horn, *Stream Reduction Operations for GPGPU Applications*, in GPU Gems 2, Ch. 36, 573-589, 2005.

[10] F. Lopez, L. Zhang, A. Mok, and J. Beaman, *Particle Filtering on GPU Architectures for Manufacturing Applications,* Computers in Industry, Vol. 71, Issue C, 2015.

[11] G. Hendeby, R. Karlsson, and F. Gustafsson, *Particle Filtering: The Need for Speed*, EURASIP Journal on Advances in Signal Processing, 2010.

[12] G. Welch, and G. Bishop, *An Introduction to the Kalman Filter*, ACM, Inc., 2001

[13] H. Balta, J. Bedkowski, S. Govindaraj, K. Majek, P. Musialik, D. Serrano, K. Alexis, R. Siegwart, and G.D. Cubber, *Integrated Data Management for a Fleet of Search-and-rescue Robots*, Journal of Field Robotics, 2016.

[14] H. Karimipour, V. Dinavahi, *Extended Kalman Filter-Based Parallel Dynamic State Estimation,* IEEE Transactions on Smart Grid, Vol. 6, Issue 3, 2015.

[15] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, *GPU Computing*, Proceedings of the IEEE, Vol. 96, No. 5, 2008.

[16] J.H. Fernandez, J.L. Speyer, and M. Idan., *Stochastic Estimation for Two-State Linear Dynamic Systems with Additive Cauchy Noises,* IEEE Trans. on Automat. Control, Vol. 60, No. 12, 2015.

[17] J. L. Speyer and W. H. Chung, *Stochastic Processes, Estimation, and Control,* SIAM, Philadelphia, 2008.

[18] J. Pan, D. Manocha, *GPU-Based Parallel Collision Detection for Fast Motion Planning,* International Journal of Robotics Research, 2012.

[19] J.R. Carpenter and A.K. Mashiku, *Cauchy Drag Estimation for Low Earth Orbiters.* AAS/AIAA Space Flight Mechanics Meeting, Williamsburg, VA, 2005.

[20] Kalman, R., A New Approach to Linear Filtering and Prediction Problems. ASME Journal of Basic Engineering, 82, 35-45, 1960

[21] L. N. Johnson, S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions,* Vol. 1, 2nd ed., John Wiley & Sons, New York, 1994.

[22] M. Harris, *How to Implement Performance Metrics in CUDA C/C++,* Nov. 7th, 2012, https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/, Accessed Nov 2020.

[23] M. Harris, *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*, Jan. 7th, 2013, https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/, Accessed Nov 2020.

[24] M. Harris, *Using Shared Memory in CUDA C/C++,* Jan. 28th, 2013, https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/, Accessed Nov 2020.

[25] M. Rada and M. Cerny, *A New Algorithm for Enumeration of Cells of Hyperplane Arrangements and a Comparison with Avis-Fukuda's Reverse Search¸* SIAM Journal on Discrete Mathematics, Vol. 32, No. 1, pp. 455-473, 2018.

[26] M. Idan and J.L. Speyer, *Cauchy Estimation for Linear Scalar Systems,* 47th IEEE Conference on Decision and Control, Mexico, 2008.

[27] M. Idan and J.L. Speyer, *Cauchy Estimation for Linear Scalar Systems,* IEEE Trans. Automat. Control, 55 (2010), pp. 1329-1342.

[28] M. Idan and J.L. Speyer, *Multivariate Cauchy Estimator with Scalar Measurement and Process Noises*, SIAM, Vol. 52, No. 2, pp. 1108-1141, 2014.

[29] N. N. Taleb, *The Black Swan: The Impact of the Highly Improbably,* Random House, New York, 2007.

[30] NVIDIA, *CUDA C++ Programming Guide,* PG-02829-001_V11.1, Oct. 2020, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Accessed Nov. 2020.

[31] NVIDIA, *CUDA C++ Best Practices Guide – Design Guide,* DG-05603-001_V11.1, Oct. 2020, *https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#optimizing-cuda-applications,* Accessed, Nov. 2020.

[32] *NVIDIA Drive*, https://www.nvidia.com/en-gb/self-driving-cars/drive-platform/, Accessed Nov. 2020.

[33] P. Reeves*, A Non-Gaussian Turbulence Simulation,* Technical Report AFFDL-TR-69-67, Air Force Flight Dynamics Laboratory, 1969.

[34] Pytorch, *Multiprocessing Best Practices*, https://pytorch.org/docs/master/notes/multiprocessing.html, Accessed Sept. 2020.

[35] G. Stathopoulos, E. Jaszewski, Alden R., *CS 179 GPU Programming*, Lecture 4, http://courses.cms.caltech.edu/cs179/, Accessed Nov. 2020.

[36] S. Sengupta, A.E. Lefohn, and J.D. Owens, *A Work-Efficient Step-Efficient Prefix-Sum Algorithm*, ResearchGate, 2006.

[37] W. D. Hillis, and G.L. Steele Jr, *Data Parallel Algorithms*, Communications of the ACM 29, 1170-1183, 1986.

[38] Y. Bai, J.L. Speyer, and M. Idan, *Planetary Flyby Attitude Estimation in an Intense Radiation Background Based on Cauchy Uncertainty*