UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

ACCELERATING DISTRIBUTED SYNTHETIC APERTURE RADAR DATA SIMULATIONS VIA CUDA

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

ERIC RACKELIN
Norman, Oklahoma
2022

ACCELERATING DISTRIBUTED SYNTHETIC APERTURE RADAR DATA SIMULATIONS VIA CUDA

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Chongle Pan, Chair

Dr. Nathan Goodman

Dr. Martin Kong

© Copyright by Eric Rackelin 2022

All Rights Reserved.

Table of Contents

Table of Contents iv

List of Tables vi

List of Figures vii

Abstract ix

1 Introduction 1

1.1 Motivation 2

1.2 Thesis Outline 3

2 Radar Background 4

2.1 Geometry 5

2.2 Range Detection 7

2.3 Cross-range Detection 14

2.4 Image Formation 16

2.5 DSAR 20

2.6 Signal Model 22

3 Programming Model Background 28

3.1 MATLAB 29

3.2 CUDA 30

3.3 MATLAB and CUDA 36

4 Implementation **40**

4.1 Preparation 41

4.2 Supporting Code for Datacube Calculations 47

4.3 Datacube Calculations 52

4.3.1 Separate Distance and Output Kernels 54

4.3.2 Monolithic Kernel 58

4.3.3 Shared Memory Usage 61

4.3.4 Eliminating Repeated Distance Calculations 63

5 Results **65**

5.1 Comparisons 65

5.2 Conclusions 69

5.3 Future Work 70

References **74**

List of Tables

5.1 Execution times of all implementations 67

List of Figures

2.1 Radar platform geometry 6

2.2 Radar data matrix 9

2.3 Radar datacube 10

2.4 An example SAR image 17

2.5 Types of reflections 18

3.1 Typical CUDA workflow 32

3.2 CUDA thread and processor hierarchy 33

3.3 Parallel thread execution on different inputs 36

3.4 Interpreting a multidimensional MATLAB variable as an array in CUDA 38

4.1 Simulated SAR image of a single point target 44

4.2 Simulated SAR image of a checkerboard pattern of targets 46

4.3 DSAR simulation input data as stored in the MATLAB environment 46

4.4 Example scattering patch partition 50

4.5 (a) The subsets of the output datacube and (b) the scattering patch segments of an output datacube subset 51

4.6 Propagation distances between satellites and a single scattering patch 53

4.7 Pseudocode of the supporting code for the two-kernel approach . . . 55

4.8 Output kernel of the two-kernel approach 57

4.9 Pseudocode of the supporting code for the monolithic kernel approach 59

4.10 The monolithic kernel 59

5.1 Comparisons of implementations with and without shared memory usage for both CUDA based approaches 67
5.2 Comparing the execution times of different DSAR data collection simulation implementations 68

Abstract

A general simulation of distributed synthetic aperture radar (DSAR) data is useful to evaluate the theoretical performance of a DSAR system and its underlying algorithms without a deployed system in place. Simulating DSAR is a computationally intensive task due to the size and complexity of the resultant data, but a simulation must complete within a reasonable amount of time to be a useful tool in practice. Through the use of both MATLAB and CUDA, a DSAR simulation can be flexible and modifiable while benefiting from efficiently implemented GPU acceleration. Multiple simulation programs have been developed using these programming languages to explore techniques of parallelizing and optimizing the performance of DSAR data simulation.

Chapter 1

Introduction

Synthetic aperture radar (SAR) is a radar based imaging technology that is useful for many civilian and military applications. One common use case is to survey the Earth's surface with radars on board orbital satellites. The main innovative feature of SAR is to take advantage of the radar's motion to synthesize a large antenna array. This means that unlike a real aperture radar, the cross-range resolution of a SAR is not limited by the physical size of its antenna [1].

However, despite not limiting the resolution, the physical antenna size does limit performance. A SAR's image quality depends on its signal to noise ratio (SNR), which is partly influenced by the size of the antenna [2]. Low SNR results in a noisy image in which important details may be obscured. Conversely, high SNR results in clear, sharp images. Compared to a small antenna, a large antenna will capture more of a signal's energy, meaning its measurements will have a higher SNR. Additionally, signals are attenuated as they propagate over long distances, so targets at longer ranges will be observed with a lower SNR. Therefore, a larger antenna is capable of capturing high quality images from longer ranges.

For practical reasons, the size and weight of any spaceborne platform is limited. Therefore, antenna size is limited. In theory it is possible to mount a very large antenna to a satellite for long-range imaging capabilities, but in reality the cost of

building and deploying such a system is prohibitive. Satellites must be built to fit on a launch vehicle, and larger, heavier payloads are more expensive to launch and deploy in orbit. Additionally, the cost of developing any new satellite radar is very high. A more cost efficient approach is needed for better SAR systems to be affordable.

Distributed synthetic aperture radar (DSAR) is an innovation in SAR technology that aims to improve the range at which scenes can be mapped without increasing the physical antenna size on one platform. The main idea is to use multiple smaller SAR satellites as one system to achieve performance equivalent to that of a single larger SAR. This approach is beneficial as there are commercially available SAR satellites developed by aerospace companies designed to fit within specific payload sizes. Compared to developing a new, larger, monolithic satellite, using a few satellites of an existing design is less expensive, as it is already a tested, proven SAR design that fits within a tested, proven launch vehicle [3]. DSAR improves on the performance of SAR while remaining practical.

1.1 Motivation

Before deploying any DSAR system, it is necessary to determine how it will be configured. The formation of satellites, known as a constellation, and radar parameters such as antenna and signal characteristics all influence performance and thus need to be tuned to maximize image quality. A general simulation of a DSAR is necessary to to evaluate potential configurations before applying them to a real-world system. The goal of creating a simulation is to enable quick evaluations of DSAR configurations and to support development of DSAR algorithms through analysis of their theoretical performance under different scenarios.

Specifically, the data collection of a DSAR system must be simulated, which is very computationally intensive. The size of the simulation under typical scenarios is too large for a CPU to execute within a reasonable amount of time in many cases. Alternatively, a graphics processing unit (GPU) can significantly accelerate computations via parallel processing and improve the simulation speed.

A GPU-accelerated DSAR simulation was previously implemented using MATLAB's built-in GPU support, but its speed is slower than desired. This thesis describes newer simulation implementations that were developed in attempts to improve the simulation speed even further so that DSAR could be simulated in a reasonable amount of time. The described implementations use a programming model with low level control over GPU memory and individual GPU threads, allowing for more efficient use of the GPU hardware that the previous implementation.

1.2 Thesis Outline

Chapter 2 introduces the basic theory of operation of SAR/DSAR and the signal model used to simulate its data collection. Chapter 3 introduces the different programming models used in the implementation of the simulation. Chapter 4 explains how the entire simulation is implemented and all successful and unsuccessful attempts of developing a faster data collection simulation routine. Chapter 5 analyzes the performance of each implementation and compares them to one another.

Chapter 2

Radar Background

Unlike optical systems that image a scene by measuring any light emitted or reflected from external sources, SAR systems make observations by emitting radar waves towards a scene and measuring the waves reflected back to the radar. This fundamental difference reveals two initial advantages. First, SAR can see through some objects that a camera cannot, such as clouds in the Earth's atmosphere. This ability is due to the relatively long wavelengths of radar waves, as longer wavelengths pass through matter with more ease than shorter wavelengths such as those on the visible spectrum. Second, SAR is self-illuminating. Because the electromagnetic energy captured by the radar is initially emitted from the radar itself, it does not have to rely on the presence of a light source to capture useful images.

These advantages prove very useful to surveillance satellites since they are set on predetermined orbit paths. A satellite's orbit is not simple to change and may have a long period of time between repetitions. For a satellite to survey a specific point of the Earth, it must first wait until it reaches a position in its orbit from which the desired point is visible. When the satellite is aligned with the desired scene, if there is no sunlight at the scene or the scene is overcast, it is a missed opportunity for an optical imaging system, but not for SAR. The nature of a satellite's orbit path is also why the extended range capability of DSAR is beneficial for surveillance.

With a longer maximum distance at which an image can be captured, more locations along the SAR's orbit path align properly with the desired scene, meaning there are more opportunities within a single orbit to image a particular area.

SAR captures recognizable images similar in some ways to what one would expect from a camera, although SAR images appear different and SAR operation is much less intuitive than a typical optical system. Many of SAR's additional advantages are due to the principles of radar technology and signal processing. To discuss these advantages and how to simulate SAR/DSAR any further, it is important to understand how SAR collects and processes data. This chapter begins with the basic operation of a SAR system and how its data are used to form images. Next, the operation of DSAR and its differences are discussed. Finally, the signal models required to simulate data collection of both SAR and DSAR are introduced.

2.1 Geometry

SAR produces images of static scenes from a moving platform – usually an aircraft or spacecraft. SAR is commonly configured as a side-looking radar, meaning its illuminating beam is aimed perpendicular to its flight path and oblique to the ground so that the radar observes an area on the ground located to the side of the platform. As mentioned previously, the feature that sets SAR apart from a real aperture radar is the utilization of the platform's motion to increase the cross-range resolution without increasing the physical antenna size. By making observations from different positions along the flight path over time, multiple observations can be combined to synthesize the data of a single observation made by a larger antenna array that has a finer cross-range resolution [1].

To describe the geometry of the radar, its beam, and its motion, some vocabulary

must be introduced. There are a few important directions relative to the flight path and the ground. The direction the platform moves along is known as the flight-track or the along-track. The slant range, or more simply, range, is the distance from the radar to a surface illuminated by the beam. Targets observed by the radar all have their own independent range. A target's ground-range is its slant range projected onto the ground. The cross-range dimension is the direction perpendicular to the slant and ground ranges, which is also parallel to the flight track. There are also some important angles determined by characteristics of the radar's antenna, including how it is driven and how it is physically constructed. The beam width is the angle of the main lobe in the antenna's emission pattern. The depression angle is the angle at which the center of antenna's beam is aimed down from the horizontal. Finally, the swath is the width of the beam on the ground in the range dimension. Figure 2.1 illustrates SAR geometry and identifies its relevant features.

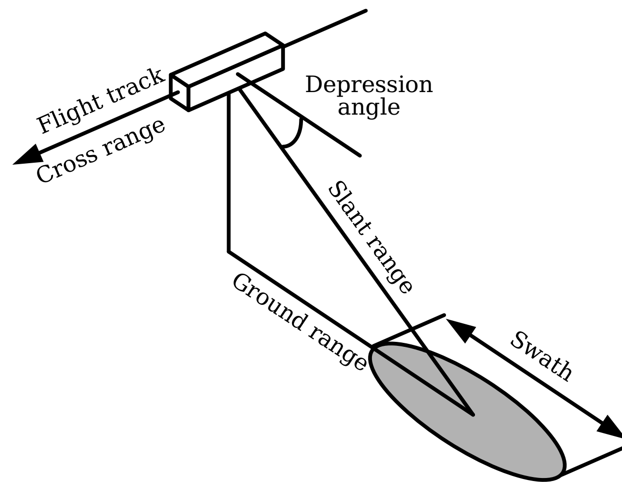

Figure 2.1: Radar platform geometry

SAR has several different imaging modes, all of which are useful for various situations [4]. Stripmap SAR is the most common imaging mode in which the radar's illuminating beam is aimed in a fixed direction relative to the platform. As

the platform travels along its flight path, the illuminated area follows, and each observation corresponds to a different cross-range position in the output image. Squinted SAR is similar to stripmap imaging, except the beam is steered ahead and to the side of the platform rather than looking out perpendicularly. This allows the radar to form images far ahead of the platform at the sacrifice of image quality.

Spotlight SAR is an imaging mode in which the illuminating beam is steered in the cross-range direction to illuminate a single, fixed target area as the platform travels along its flight path. Compared to stripmap SAR, spotlight SAR captures images of higher quality and finer resolution at the sacrifice of the imaged area's size. Circular SAR is an imaging mode in which the platform flies in a circular path centered around the target scene, illuminating the same area for the entire duration of the data collection. Similar to spotlight SAR, circular SAR captures higher quality, finer resolution images of smaller areas. Additionally, circular SAR can be used to create three dimensional images and track the motion of moving targets, such as vehicles, over time [5].

This list of SAR imaging modes is not exhaustive, but does illustrate what is possible with SAR. For the purpose of simulating satellite DSAR, this project focuses on stripmap, though the conclusions are applicable to all SAR modes.

2.2 Range Detection

SAR is a pulsed radar, meaning it periodically emits finite-length signals carried by radio waves to detect targets and measure their ranges. The most basic type of pulse has no modulation and takes the form

$$\bar{x}(t) = a(t)\cos(2\pi f t + \phi) \tag{2.1}$$

where $a(t)$ is the amplitude function used to pulse the signal and the cosine term is the carrier signal and global phase. The variable f represents the carrier frequency and ϕ represents the phase offset of the carrier signal. The bandpass signal $\bar{x}(t)$ represents the actual voltage transmitted or received by the antenna, but it is often useful to consider the pulsed waveform without the carrier frequency, which in this case is only the complex envelope

$$x\left(t\right) = a\left(t\right) e^{j\phi}. \tag{2.2}$$

To pulse the carrier signal, the amplitude function it is multiplied with switches between on and off. If the amplitude function is equal to the rect function

$$\mathrm{rect}\left(t\right) = \begin{cases} 1 & -0.5 \leq t \leq 0.5 \\ 0 & \text{otherwise} \end{cases}, \tag{2.3}$$

then the carrier signal will be present only for the duration of the rect function. The resultant signal is a single pulse. If the amplitude function contains periodically repeating rect functions, sometimes called a train of rect functions, the carrier signal is switched on and off periodically, resulting in multiple individual pulses. The constant rate at which these pulses are transmitted is known as the pulse repetition frequency (PRF), and inversely, the time between pulses is the pulse repetition interval (PRI).

To detect targets, the radar transmits pulses and measures their echoes. As a pulse travels away from the radar, it will reflect off any objects it hits, potentially directing some amount of the pulse's energy back to the radar. When this happens, the target may be detected. Multiple targets reflecting the same pulse each produce their own echo and can be detected individually. A target's range, the distance

between the radar and the target, can be determined from the time delay of the echo τ. The pulse travels to the target and back at the speed of light, c, so the target's range R is equal to

$$R = \frac{\tau c}{2}. \tag{2.4}$$

As pulses are emitted, the reflection data are recorded by an analog to digital converter (ADC) measuring the antenna voltage signal over time. The data are arranged into a two dimensional matrix of discrete complex values. Figure 2.2 shows the axes of the matrix, labeled fast time and slow time. The fast time axis corresponds to the time within the measurement of a single pulse's returns, which is the time within a single PRI, and the length of this axis is equal to the number of samples collected within a PRI. The slow time axis corresponds to the time across multiple pulses and its size is equal to the number of PRIs throughout the duration of data collection.

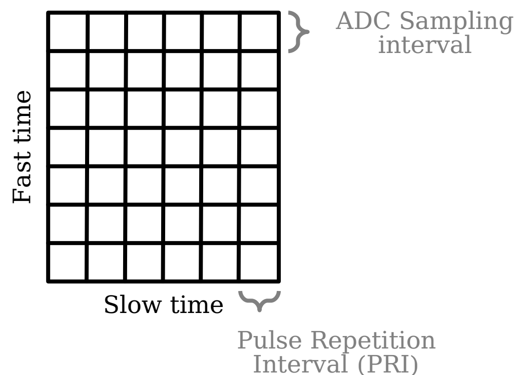

Figure 2.2: Radar data matrix

If there are multiple antennas with separate receiving channels, the data matrix is extended into a third dimension representing the different channels, forming a datacube, which is essentially multiple data matrices stacked on one another. Figure 2.3 shows a datacube with three receiving channels.

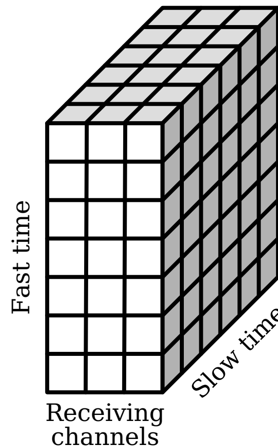

Figure 2.3: Radar datacube

The data are arranged into a matrix or cube because it is useful to visualize how the data are processed. For example, in a static pulsed radar, analyzing the data over fast time can reveal a target's range at a specific point in time. The time an echo is recorded at in a PRI corresponds to the range of a detected target, and the amplitude of the echo corresponds to the brightness or reflectivity of the target. Analyzing the data over slow time can reveal the target's motion by showing the change in range over multiple PRIs.

In general, pulsed radars have one glaring potential issue in range measurement. If the time between the radar transmitting a pulse and receiving its echo is longer than one PRI, then a second pulse will be transmitted before the echo due to the first pulse is received. When the echo of the first pulse is received, there is no way to differentiate whether that echo is caused by the first pulse reflecting off of a distant target or the second pulse reflecting off of a close target. This phenomenon is known as range ambiguity. In spaceborne SAR applications, there are several pulses in flight at the same time, so it is assumed that all received echoes propagated over a distance corresponding to the radar's altitude and beam direction in relation to the Earth's surface.

Another issue in range detection is signal attenuation. For an ideal radar, the radar range equation relates the power, P_r, of a pulse returned by a target to the transmitted pulse power, P_t, the target's range, R, and other certain physical parameters [1].

$$P_r = \frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 R^4} \tag{2.5}$$

The other parameters include the signal's wavelength λ, the target's radar cross section σ, and the antenna gain G. The antenna gain is a measure of how concentrated the signal power is in the antenna's main lobe and depends on the physical construction of the antenna.

An important interpretation of (2.5) is that the received power of an echo is inversely related to R^4, so it drops off dramatically as the target range, R, increases. This is problematic for practical applications, as receiver noise will be present. At long ranges, it is common for the noise to have more power than the signal itself. An SNR this low will produce unrecognizable images, so the signal drop off must be mitigated.

Fortunately, filtering techniques can be used to significantly improve the SNR. Matched filtering is a technique used to extract the times at which echoes are received from a noisy signal. A matched filter corresponds to one specific signal waveform. It is represented in the time domain by its impulse response $h(t)$. For a signal $x(t)$, the impulse response of its matched filter is equal to the conjugated, time reversed signal

$$h(t) = x^*(-t). \tag{2.6}$$

The filter $h(t)$ can be applied to the original signal $x(t)$ by convolution of the two, which produces the correlation between $x(t)$ and $x^*(t)$. The correlation between $x(t)$ and $x^*(t)$ is the autocorrelation of $x(t)$, which is not useful by itself,

but the context in which it appears in a cross-correlation function is useful. The cross-correlation function is the result of applying the matched filter of a pulse signal to some measured signal and is a measure of how similar the pulse signal is to the measured signal over time. The correlation of the pulse signal and the radar's measurement within a PRI will contain a time-shifted copy of $x(t)$ and $x^*(t)$'s correlation function whenever an echo of the pulse is received. The random noise in the measured signal is not likely to strongly resemble the pulse, so the noise in the cross-correlation function will have nearly zero amplitude. Thus, the filtered signal has a much higher SNR than the unfiltered signal, even when the noise has more power than the pulse in the original measured signal. Each peak in the matched filter output likely corresponds to an echo representing a target. The peak's time shift corresponds to the range of the target it represents and its amplitude is related to how strongly the target reflected the pulse.

Since the radar data are discrete, the time domain is broken up into different time slots. After the matched filter has been applied, each time slot in the fast time axis of the data matrix or datacube is associated with a range bin. An echo recorded in a certain time slot represents a target at a range that falls somewhere within the limits of the associated range bin.

Resolution is the ability to differentiate separate targets that are close to one another. A radar's range resolution of a radar is the minimum difference in range required to resolve targets. Range resolution is based on the ability to differentiate the peaks representing different targets in the matched filter output. When a single pulse hits multiple targets, if they are too close to each other, then their representative autocorrelation instances in the matched filter output will overlap. Their peaks combine coherently and appear as one echo from that of a single larger or "brighter" target. Pulse waveforms with wider autocorrelation functions will result in worse

range resolutions since they will overlap each other at greater range differences. The basic pulse modeled in (2.1) has a relatively wide autocorrelation function and does not perform well. In fact, its autocorrelation function is wider than the pulse itself.

Narrowing the width of the autocorrelation function improves the range resolution. Narrower peaks in the matched filter output allow echoes to be distinguished from one another with less time between their occurrences. The width of a peak is inversely related to the bandwidth of the pulse waveform, so the range resolution can be expressed as [1]

$$\Delta R = \frac{c}{2\beta} \tag{2.7}$$

where ΔR is range resolution, c is the speed of light, and β is the bandwidth of the pulse. A larger bandwidth will produce a finer range resolution. It is possible to increase the bandwidth of the simple pulse seen in (2.1) by shortening its duration, but the signal's energy will also decrease. Lower energy signals result in a lower SNR in the cross correlation function, and therefore harm performance. Increasing the transmitting power can compensate for the shorter pulse's lower energy, but in many cases, the amount of power needed to achieve good range resolution and image quality with a simple pulse is impractically large. Instead, a more complicated pulse waveform is typically used.

Pulse compression is a technique in which the pulse waveform is modulated in some way to increase its bandwidth without shortening its duration. There are multiple types of modulation that can be used in pulse compression, such as binary phase coding, in which the phase of the signal pseudo-randomly flips between 0 and π radians, or linear frequency modulation, where the pulse's frequency is a linear function of time [2]. In any type of pulse compression, the pulse's bandwidth is not

solely dependent on the length of the pulse. Therefore, the pulse's autocorrelation function can be narrowed without decreasing the signal's energy. This allows for the range resolution to be improved without increasing the transmitting power or decreasing the SNR, resulting in high-quality, fine-resolution range detection.

2.3 Cross-range Detection

Similar to the range resolution, the cross-range resolution of a radar system is the minimum distance required between two targets for them to be distinguishable in the cross-range direction. In real aperture radars, the cross-range resolution is based solely on the physical antenna's beamwidth in the cross-range direction. If two targets are within the beam simultaneously, their cross-range positions are not distinguishable. Radars with narrower beams have smaller, better resolutions. As mentioned previously, the beamwidth is an angle determined by the physical construction of the transmitting antenna, which presents two bottlenecks. The first is the resolution's dependence on range. Since the beamwidth is a constant angle, the illumination area of the beam is larger when further away from the antenna. This means the cross-range resolution increases with distance. The second bottleneck is the size of the antenna. Larger antennas generally have narrower beamwidths and, therefore, better cross-range resolutions. However, due to the magnitude of the distance between a satellite and the Earth's surface, an antenna capable of sub-meter cross-range resolutions would be too large to be practical on a spacecraft.

SAR has an advantage over real aperture radars because it can circumvent the previously mentioned bottlenecks by combining multiple observations collected over time. A physical antenna array has multiple transmitting and receiving elements that all transmit and receive a pulse simultaneously to make simultaneous

observations. A larger antenna with more elements produces a narrower beam. A synthetic aperture radar is different, as it uses a much smaller antenna with a wider beam and mimics the spacing of individual elements on a much larger array by capturing multiple observations from different locations over slow time while moving. Multiple observations can then be combined coherently into observations equivalent to that of a larger real aperture antenna, effectively synthesizing a large aperture with a small beamwidth.

The synthetic aperture length L is equal to

$$L = vT_a \tag{2.8}$$

where v is the platform velocity and T_a is the aperture time, the amount of time spent collecting data. The cross-range resolution ΔCR is equal to [1]

$$\Delta CR = \frac{\lambda R}{2L} = \frac{\lambda R}{2vT_a}. \tag{2.9}$$

As discussed before, in stripmap imaging, the illumination beam is fixed relative to the platform and the platform travels along its flight path, the illumination footprint on the observed surface follows. Since the beamwidth is defined by constant angles in both the range and cross-range directions, the longer the range is, the larger the illumination footprint is in both directions. A radar observing a scene at a longer range will illuminate a single cross-range slice of the scene for a longer period of time and within more pulses. Therefore, the effective aperture time is directly related to range, which balances out the direct relation to range in (2.9). As a result, the cross-range resolution of SAR is independent of range.

The most significant interpretation of (2.9) is that the cross-range resolution is

inversely related to the synthetic aperture length L. Since the length of the aperture is not limited by the practicality of construction, a SAR can achieve much finer cross-range resolutions than any practical real aperture radar by increasing the path length over which observations are collected.

2.4 Image Formation

The two-dimensional images produced by SAR look similar to an overhead image captured by an optical sensor, although there is one important distinction. The dimensions of a SAR image are mapped to slant-range and cross-range positions, not the dimensions of the target scene's ground plane. Figure 2.4 shows an example of a SAR image captured by a radar observing a building surrounded by trees looking rightward from the left side of the image. The vertical axis of the figure is the cross-range dimension and the horizontal axis is the range dimension. The image shown is the result of using an existing SAR image from Sandia National Laboratories as an input to the simulation program. The image is still recognizable, but the slant-range mapping causes some visual abnormalities in one of the dimensions [6].

One effect of the slant-range mapping, called foreshortening, causes slopes to appear compressed. For example, the side of a mountain or hill facing the radar will look shorter than the side facing away from the radar. This occurs because the distance between the base and the peak of the mountain in the slant-range direction is smaller than the distance between the base and peak in the ground-range direction.

Another effect, known as layover, is a more extreme case of foreshortening. If an object is tall and narrow enough and the radar's depression angle is steep enough, the top of the object can be closer than the bottom of the object in the slant

Figure 2.4: An example SAR image

range. For example, a mountain peak may appear ahead of the slope leading up to it that faces the radar. Both foreshortening and layover become more pronounced at steeper viewing angles.

Lastly, shadowing is an effect that appears behind tall objects. If the depression angle is shallow enough, a tall object illuminated by the radar's beam will cast a shadow behind itself, obscuring any objects in its shade. In the image, the shadowed regions appear completely dark. Shadowing becomes more pronounced at shallower viewing angles.

The brightness of an object in the image is dependent on how pulses reflect off of it. A mostly smooth, flat surface such as a parking lot or a calm body of water will reflect a pulse once, like a mirror. When a pulse hits this type of target, almost none of its energy is returned towards the radar. For this reason, smooth areas

appear dark. A corner between flat ground and a vertical wall, commonly found on the edges of buildings, reflect a pulse twice. When a pulse collides with such a corner, the double bounce will cause the pulse's new trajectory to be nearly parallel to and opposite of its original trajectory, resulting in a strong echo with a relatively high amount of energy. This phenomenon causes corners to appear bright. Finally, rough surfaces like a grassy field or the foliage of a tree will diffuse a pulse's energy, scattering it in many directions. Some fraction of the pulse's energy will be returned back to the radar, resulting in a dim, but not dark appearance.

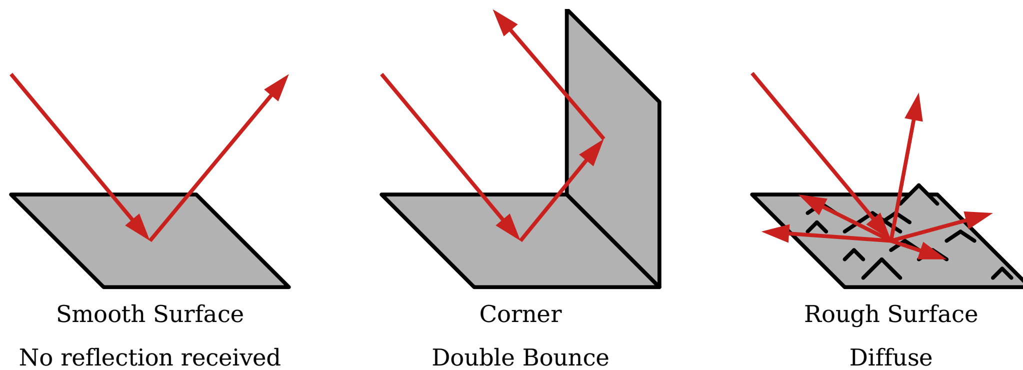

Figure 2.5: Types of reflections

Once collected, the SAR data can then be processed by an image forming algorithm to produce a map of the observed scene. Regardless of which algorithm is used, the output images are useful to check the correctness of simulated data and to evaluate the performance of the simulated DSAR. In this project, the time domain backprojection algorithm [4] is used to form images from the simulated data. Backprojection has many advantages over other SAR imaging algorithms, including its inherent compensation for any non-linear platform motion and its applicability to multiple imaging modes. These advantages are very useful when simulating DSAR under many different configurations.

SAR imaging requires the radar's geometry to be known precisely at the time of image formation, including the radar's location at every slow-time interval. In the case of an aircraft platform, these locations can be measured from various sensors such as GPS and inertial measurement units as the radar collects data. In the case of a satellite platform, the locations can be calculated from the predetermined orbit and the time at which the observations were made. In SAR data simulation, all platform locations are known for every pulse simply because they are defined as part of the simulation scenario.

The radar geometry is used to determine the location of every pixel in the output image. Since the radar's flight path, altitude, depression angle, and beamwidth are all known, the location of the target scene relative to the flight path and the size of the target scene can be calculated. A grid is superimposed onto the scene to represent the pixels of the image. Each cell in the grid encompasses the area on the ground represented by a single pixel and is associated with a complex value initialized to zero. The exact topology of the scene is unknown, so it is assumed to be flat. The ground-range coverage of the grid is based on the antenna's beamwidth in the elevation direction and the ground-range length of each grid cell is derived from the radar's range resolution. Similarly, the cross-range length of the grid is based on the antenna's beamwidth in the azimuth direction and the length of the platform's flight path while the cross-range length of each grid cell is equal to the radar's cross-range resolution.

After performing pulse compression on the SAR data, the brightness of each pixel can be determined based on location. At a single point in slow time, the range of every pixel can be determined by a simple Cartesian distance calculation from the radar's location to the pixel's location and back. When a pixel's range falls within a range bin of the fast time axis, the complex value of the pulse compressed

return signal in that range bin is phase-corrected and added to the pixel. A single range bin contributes to multiple pixels at once, and the set of pixels influenced by a single range bin is different for every point in slow time. As more pulse reflections are processed and added to the image, the image is unrolled in the cross-range dimension and becomes sharper. The brightness of each pixel in the output image is equal to the amplitude of its final complex value.

2.5 DSAR

DSAR works under the same principles as SAR but has multiple radars on separate platforms collaborating to produce one image. Every radar in the system is an independent, fully capable SAR. The only additional functionality required for an individual SAR to be in a DSAR system is the ability to synchronize its PRI with other platforms, so using commercially available satellite radars is possible with some modification. All satellites within a DSAR fly in a predetermined formation, or constellation, capturing synchronized observations.

When collecting data, not necessarily all radars emit pulses, but all radars measure all returns. In other words, there are N_{TX} transmitters and N_{RX} receivers such that $N_{TX} \leq N_{RX}$. The collected DSAR data are arranged into a datacube in which every receiving channel corresponds to a different radar. At the start of a PRI, each transmitter emits a unique pulse waveform. Throughout the PRI, each receiver records measurements through its receiving channel. All N_{RX} receiving channels are synchronized in the fast-time domain. The echoes of all transmitters' pulse waveforms overlap one another in every receiver's channel, but they are separable from one another via matched filtering due to their unique modulations [7, 8].

Applying a pulse waveform's matched filter to a receiving channel's fast-time

measurement will isolate the reflections due to the corresponding transmitter. After a single observation has been made, all N_{RX} receiving channels can be separated into N_{TX} different reflection measurements in post processing. With all of the DSAR's collected data, a different image can be formed for every combination of transmitter and receiver for a total of $N_{RX} \cdot N_{TR}$ images. A transmitter and receiver pair corresponding to a single platform composes a monostatic setup, while the remaining pairs corresponding to two platforms compose a bistatic setup. The quality of each image corresponds to the performance of just one of the radars in the system if it were acting independently, assuming all radars are equivalent. The images are not all identical due to the different positions of the platforms, but the distances between platforms are small enough compared to the distance to the target scene for the images to be virtually the same. To produce a single output image, all individual images are coherently combined through addition.

Using multiple individual radars as a DSAR is comparable to using one radar with more power and a larger antenna. As seen from (2.5), the power of a pulse return P_r is directly related to the transmitted pulse power P_t and the square of the antenna gain G. In a DSAR with multiple transmitters, the individual pulse signals are not transmitted with any more power than they would be from a standalone SAR, but there is more transmission power in total when considering multiple transmitted signals. Additionally, the antenna gain G has a direct relation to the area of the receiving antenna, and together, multiple radars have more total antenna area than any single radar in the DSAR. However, the reflection of a single pulse waveform is not received with any more power in the individual receiving channels. Rather, it is the summation of all individual images where the power is increased. In an aggregate DSAR output image, every pixel now has contributions from the amplitudes of $N_{RX} \cdot N_{TR}$ signals, which raises the image's amplitude higher above the

noise floor. Thus, the SNR of the DSAR image is higher than the SNR of an image produced by a single radar in the system.

One potential problem of DSAR is signal contamination. Matched filtering is capable of removing some of the undesired signals and noise, but cannot remove them absolutely. There will always be some remaining amount of noise and cross-correlation from other waveforms. When applying the filter to fast time measurements, the undesired pulse waveform, if received strongly enough, can cause aberrations in the output image. At relatively scene ranges, the R^4 term in (2.5) is too small to attenuate the signals enough to prevent cross-contamination. However, the solution is simple. Using fewer transmitters at short ranges will result in less signal contamination. This is an acceptable workaround since the shorter range inherently improves SNR.

A DSAR system is more capable than any individual SAR within it. The resolution is not any different, but the SNR, and therefore the image quality, is improved. Images of scenes at further ranges can be captured clearly, which is very useful as explained at the beginning of this chapter. Under urgent situations, locations need to be mapped as soon as possible and extending the maximum range at which the radar can capture quality images increases the number of opportunities in the system's orbit to map a location. Overall, a DSAR composed of small radars has comparable performance and capabilities to a larger monolithic SAR.

2.6 Signal Model

Simulation of a SAR data collection calculates its data matrix. To simulate the data of a single PRI, the time-delayed, amplitude-scaled pulse reflected by every point of the target scene must be calculated and summed. In theory, a single pulse's

total reflection is an integration of reflections across a continuous area of the Earth's surface, but for simulation, reflections are approximated by dividing the target scene into discrete patches of scattering targets. These calculations are performed in the frequency domain as described below.

The time delay of each discrete reflection is determined by its propagation distance. As explained in Section 2.2, for a monostatic radar, the range of a target can be determined from the measured time delay of its echo as shown in (2.4). When simulating SAR, the range R of a scattering patch is known and its time delay τ is calculated as

$$\tau = \frac{2R}{c}. \tag{2.10}$$

Given a reflection's base pulse waveform $x(t)$, its time delay can be modeled in the time domain as $x\left(t-\tau\right)$. Applying the Fourier transform produces its frequency domain representation, where the time shift is modeled as a phasor that rotates versus frequency according to

$$x\left(t-\tau\right) \iff X\left(F\right)\exp\left(-j2\pi F\tau\right) \tag{2.11}$$

The frequency representation is easier to use in discrete numerical calculations, as there is no need for explicit interpolation in the discrete-time domain.

The real transmitted waveform $\bar{x}(t)$ has a constant carrier frequency F_0. The carrier signal's phase is also dependent on τ and is represented as

$$\exp\left(-j2\pi F_0\tau\right), \tag{2.12}$$

so the complex representation of the measured echo is

$$\bar{x}(t-\tau) \iff X(F)\exp(-j2\pi F\tau)\exp(-j2\pi F_0\tau). \tag{2.13}$$

After simplification, the delayed signal is

$$\bar{x}(t-\tau) \iff X(F)\exp(-j2\pi(F_0+F)\tau). \tag{2.14}$$

The model in (2.14) represents the time-delayed complex voltage signal of a reflection, which is needed to compute the data matrix.

The amplitude scaling of each discrete reflection is based on the patch that returned it. Every scattering patch has a complex reflectivity coefficient α representing the combined strength and phase shift of the reflections from every point within it. The reflection's signal model with the scattering patch's influence is simply

$$\alpha X(F)\exp(-j2\pi(F_0+F)\tau). \tag{2.15}$$

The complete reflection of a pulse is the sum of all discrete scattering patch reflections.

The frequency domain data matrix has a frequency axis in place of the time domain data matrix's fast-time axis. The length of the frequency axis, N_f, dependents on the sampling rate of the radar's ADC and the radar's PRI. Each vector along the frequency axis stores the frequency spectrum of a complete reflection. The other dimension of the frequency domain data matrix is still the slow time axis, which has N_{pulse} complete reflection vectors, one for each PRI, arranged along it. If the target scene is divided into N_{patch} discrete scattering patches, then every element in

the frequency domain data matrix is modeled as

$$Z\left(F_p, q\right) = X\left(F_p\right) \sum_{i=1}^{N_{patch}} \alpha_i \exp\left(-j2\pi\left(F_0 + F_p\right)\tau_{q,i}\right) \tag{2.16}$$

where q is the index along the slow time axis identifying the pulse and p is the index along the frequency axis identifying the frequency component of the pulse reflection represented by the element $Z\left(F_p, q\right)$. It is worth noting that the echo's time delay $\tau_{q,i}$ for scattering patch i is different for every pulse since the platform is moving throughout slow time.

A practical radar collects data in the time domain, but the simulated data matrix modeled by (2.16) is in the frequency domain. Performing an inverse discrete Fourier transform over the frequency axis for every pulse in the data collection is the only step required to convert it to the time domain data matrix. After this step, the calculation of the simulated data is complete.

Similar to SAR, a simulation of DSAR data collection calculates its datacube. Although it is more practical to push limits of SAR performance by building DSAR systems, it is also more complicated. Luckily, they both follow the same principles to simulate. In fact, SAR simulation can be considered a special case of DSAR simulation in which there is only one platform that both transmits and receives. To simulate DSAR data, the SAR data matrix must be calculated for every combination of transmitter and receiver. All matrices corresponding to the same receiving satellite are summed to produce a single slice of the datacube taken along the receiving channel axis. Every vector along the fast-time dimension is the sum of the discrete reflections of every transmitter's pulse in one PRI measured by one receiver.

The signal model for DSAR simulations follows the same principles as the SAR model in (2.16), but has a slightly more complicated time delay for each echo. Some

receiver and transmitter pairs will form a bistatic radar path, so the time delay of an echo depends on both the scattering patch's distance to the transmitting platform d_{TX} and the scattering patch's distance to the receiving platform d_{RX},

$$\tau = \frac{d_{TX} + d_{RX}}{c}. \tag{2.17}$$

Additionally, there are more discrete reflections to calculate. Instead of summing together only N_{patch} time-delayed, amplitude-scaled pulses for a single element in the datacube, there are $N_{patch} \cdot N_{TX}$ reflections, all with independent time delays, summed together.

If there are N_{TX} transmitters and the target scene is divided into N_{patch} discrete scattering patches, then the frequency domain datacube is modeled as

$$Z\left(F_p, m, q\right) = \sum_{n=1}^{N_{TX}} X_n\left(F_p\right) \sum_{i=1}^{N_{patch}} \alpha_i \exp\left(-j2\pi\left(F_0 + F_p\right)\tau_{m,n,q,i}\right) \tag{2.18}$$

where q is the index along the slow-time axis identifying the pulse, m is the index along the receiving channel axis identifying the receiver, and p is the index along the fast-time axis identifying the frequency component of the pulse reflection represented by the element $Z\left(F_p, m, q\right)$. It is worth noting that the echo's time delay $\tau_{m,n,q,i}$ for scattering patch i is different for every pulse, transmitter, and receiver combination since the entire constellation moves throughout slow time. As with the SAR data, an inverse discrete Fourier transform will convert the frequency domain DSAR datacube to the time domain DSAR datacube.

The model in (2.18) is the essence of the data simulation. The typical dimensions of the output have the number of transmitters N_{TX} and the number of receivers N_{RX} in the single digits. The number of pulses N_{pulse} is typically in the

hundreds or low thousands, the size of the fast-time/frequency axis N_f is often in the tens of thousands, and finally, if simulating a full scene, the number of scattering patches N_{patch} can be in the range of hundreds of thousands. In addition to the large dimensions of the output, DSAR simulations have added complexity, making it more computationally intensive than SAR simulation. The goal of this project is to accelerate the simulation so that it can execute within a reasonable amount of time.

Chapter 3

Programming Model Background

As discussed in Chapter 1, the goal of this project is to shorten the execution time of DSAR data collection simulations. The original implementation based entirely in MATLAB does make use of GPU hardware to accelerate the computations, but it is limited in speed by the language used to implement it. Many programming languages and models exist, all with their own strengths and weaknesses. MATLAB has a place in this project but is not the ideal language for all aspects of the simulation as it does not provide low-level access to the hardware that would enable more optimizations. Now that an understanding of DSAR has been established in Chapter 2, the requirements of a performant DSAR data collection simulation can be postulated. This project calls for a programming language that can take advantage of the highly-parallel GPU hardware and is suited for high-performance computing. This chapter explains MATLAB and its strengths, then introduces the CUDA C language and parallel programming model, and finally goes in depth about how both languages can be used in the same program.

3.1 MATLAB

MATLAB is a programming language designed to make complicated mathematics easy to implement. It has native support for matrix, array, and complex math operations along with standard features of other languages such as user defined functions and classes. Although there are tools available to generate C/C++ code or compile a standalone executable from MATLAB code, MATLAB is frequently used as a scripting language contained entirely within its own development environment. Unlike other languages that programmers use to create applications or tools to be used, programmers mainly use MATLAB to quickly create scripts for evaluating mathematical models. Typically, it is used as a numeric computing environment for processing or visualizing data and algorithms. This paradigm is supported through the structure of the MATLAB language itself and its development environment.

The language's syntax allows complicated mathematical formulas and algorithms to be written quickly. A single variable can represent a scalar, vector, matrix, or multidimensional array, all of which work with simple operators. Matrix math works intuitively with vectors and matrices of all sizes, assuming the sizes are compatible for each respective operation. Additionally, scalar operations can be applied to every element within a variable via a single operator. Many built in functions work on variables of arbitrary dimensions, too. For example, given a vector, the fast Fourier transform (FFT) function will return the Fourier transform of that vector. Given a matrix, the same function returns a matrix containing the Fourier transform of each column in the input. For most transforms and operations, there is no need for the programmer to explicitly iterate over every element of a variable. These features allow formulas to be implemented with relatively few lines of code, arguably making it easier for humans to interpret, which significantly reduces

development time.

MATLAB's development environment provides a lot of features to help the programmer. Plotting data is a trivial task thanks to the many built in graphing methods for several types of coordinate spaces. 3D meshes, images, complex numbers, and many more types of plots can all be created with relatively few commands. There are many toolboxes (an analog to libraries in other languages) available from MathWorks that each add more functionality for a specific area of study. For example, the control system toolbox provides a high-level interface to create and define models for control system architectures. When using this toolbox, there is no need for the programmer to implement all the underlying math to run their control system simulations. Other toolboxes cover topics such as deep learning, antennas, finance, and many more, all of which would normally require a lot of area-specific knowledge and time to implement. These features make MATLAB a useful analysis tool to scientists and engineers across many disciplines.

MATLAB is appropriate to use when rapid development is a higher priority than performance. When developing simulations or models, experimental results may reveal bugs within the program, which guide new code changes as they emerge. Different algorithms may be proposed after examining initial results. In either case, the code will be changing frequently. MATLAB allows the programmer to quickly identify and make appropriate changes without sinking time into the low level of implementations of mathematical operations or loops.

3.2 CUDA

CUDA is a programming model created by NVIDIA for general purpose computing on GPUs. Historically, the GPU was designed and used exclusively for

graphical computations. Filling many pixels on a screen at once for displaying bitmap textures requires high memory bandwidth. Transforming sets of vertices for 3D graphics requires quick linear algebra operations. The CPU, which processes data sequentially, was not well suited for these tasks, so the GPU, which processes data in parallel, was developed to handle them [9]. Eventually, data scientists discovered that the GPU hardware could also accelerate certain computationally intensive tasks other than rendering graphics by using the existing graphics libraries in ways they were not originally designed for. It was possible to perform general computations with a GPU, but it was unintuitive and difficult for programmers without knowledge of computer graphics. NVIDIA eventually took notice of the trend among data scientists and created CUDA to give programmers access to their GPU hardware with a more general interface for use in high-performance computing [9]. CUDA can be used through a variety of languages, libraries, and compiler directives. In the scope of this project, it is utilized through CUDA C, an extension to the C language supported by NVIDIA's C compiler, nvcc.

Programs accelerated by CUDA use a mix of traditional sequential programming and parallel programming. The CPU, known as the host, executes standard C code sequentially and the GPU, known as the device, executes accelerated functions in parallel, both of which do so in their own independent sets of memory. An accelerated function, known as a CUDA kernel, is a single function that many GPU threads execute simultaneously. Each thread executes the same code, but typically each thread will run the code on different inputs from different memory locations.

Figure 3.1 shows the typical workflow of a GPU accelerated program. The program initially runs on the host and continues to do so until it reaches code that is executed in parallel on the device. Here, the typical CUDA workflow starts with transferring input data for the kernel from host memory to device memory. Next,

the host launches the kernel, which then is executed on the device. Once the kernel has finished executing, the output data are transferred from the device memory to the host memory, which can then be accessed by the rest of the program. The code executing in a kernel cannot interact with the user at all, so the sequentially executed code is responsible for collecting inputs and providing outputs.

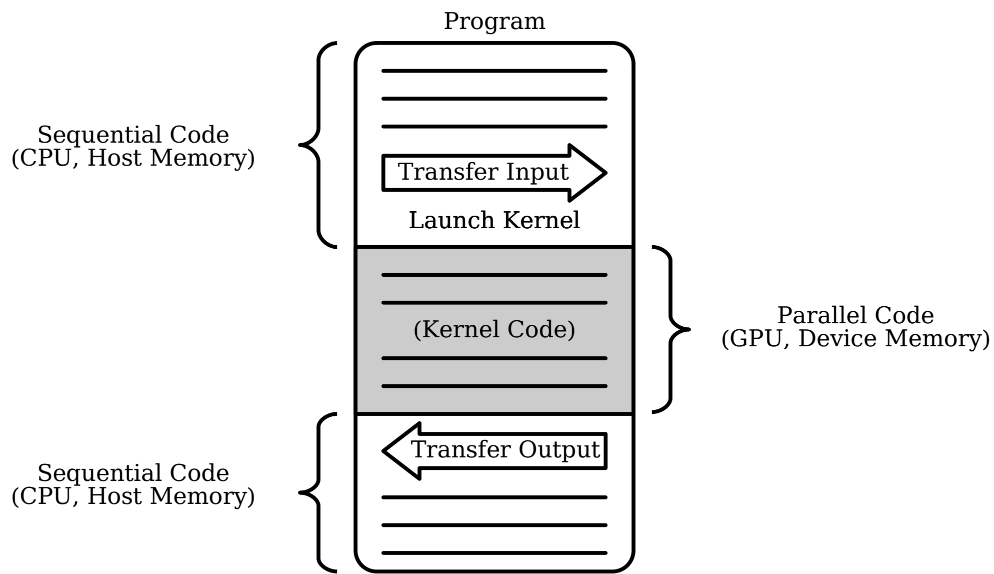

Figure 3.1: Typical CUDA workflow

The parallel code follows what NVIDIA describes as a single instruction, multiple thread (SIMT) programming model [10]. Figure 3.2 shows the hierarchies of CUDA's corresponding hardware and thread organizations.

At the lowest level, a single thread executes on a CUDA core, also known as a streaming processor. Each thread has its own set of registers and private local memory that cannot be accessed by other threads. Threads are organized in groups of 32 called warps. If there are not enough threads to fill a warp, it will be padded out with dummy threads. In a single execution step, all threads within a warp execute the same instruction. If 32 CUDA cores are available, the warp will execute in

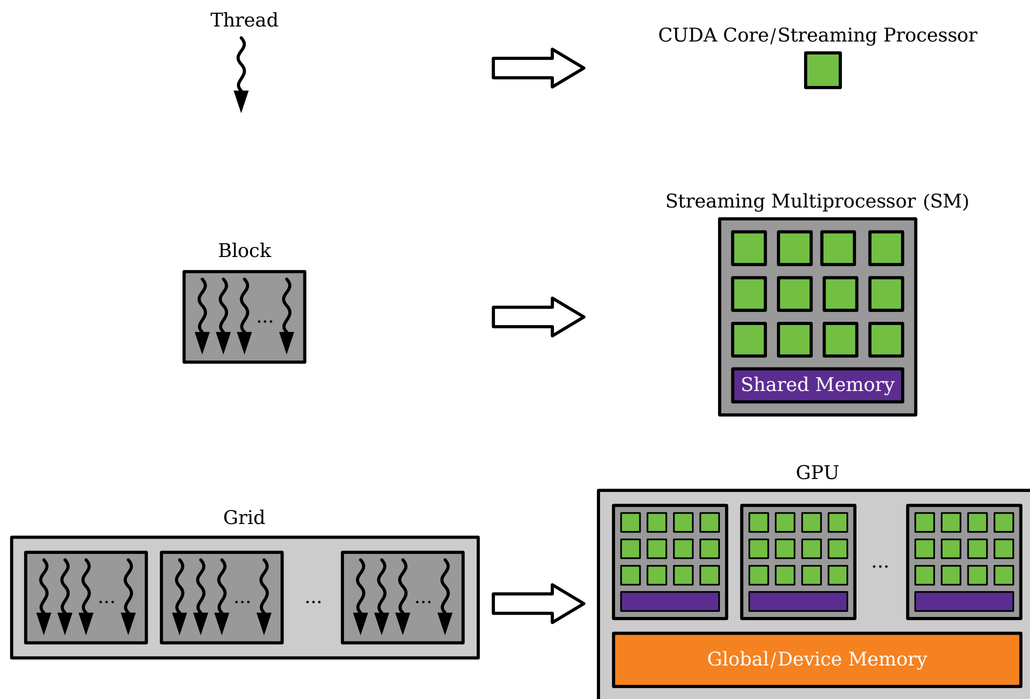

Figure 3.2: CUDA thread and processor hierarchy

one clock cycle, but if not, the warp will take multiple clock cycles for all threads to execute the instruction. Additionally, if the code branches, the threads in the warp may not all have the same instruction to execute within the step. In this situation, known as control divergence, the warp execution must be split across multiple clock cycles. The threads are organized into groups by their instruction such that one group of threads execute the same instruction for each clock cycle of the step. For example, when an if-else statement causes some threads within a warp to take action A while the rest of the threads in that same warp take action B, the warp will execute each branch sequentially. First, the threads/cores performing action A execute in parallel while the threads/cores performing action B do not perform any meaningful work. Next, the threads/cores performing action B execute in parallel while the first group of threads wait. Overall, when thread divergence occurs within a warp, more clock cycles are required to execute that work, so it is best to avoid

this phenomenon when possible.

The second level in the thread hierarchy is the thread block, which executes on a streaming multiprocessor. A thread block is a group of threads, and a streaming multiprocessor is a group of CUDA cores that share a small bank of memory. The programmer can arbitrarily set the size and shape of a thread block and index the threads in one, two, or three dimensions, but there is a limit to the total number of threads per block. Within a single kernel launch, all blocks have the same number of threads. A streaming multiprocessor can execute multiple blocks simultaneously, but there are separate limits on both how many threads and how many blocks it can execute simultaneously. Every thread within a block has access to the same portion of the multiprocessor's shared memory, but cannot access shared memory used by other blocks. The shared memory can be used along with synchronization primitives to communicate between threads. A multiprocessor's shared memory is faster than the GPU's global memory, so it can additionally be used to speed up some algorithms via caching. It is best practice to set the block size be a multiple of the warp size (currently, 32 for every CUDA-enabled GPU) while maximizing the number of threads concurrently on a single multiprocessor. This maximizes the multiprocessor utilization, reducing the total execution time.

At the highest level, the grid is executed on the entire GPU. In a single kernel launch, the grid is the organization of all thread blocks. Similar to thread blocks, the programmer can arbitrarily set the size and shape of the grid, which can index all the blocks within it in one, two, or three dimensions. As the kernel runs, the grid's thread blocks are assigned to multiprocessors with enough free cores to hold all of a block's threads. When a thread block has finished executing, a new block will be assigned to the multiprocessor, taking the place of the completed block. All threads in all thread blocks in the grid have access to the GPU's global memory bank. The

host can also directly access the global device memory, so it is where the kernel's inputs and outputs are located immediately before and after the kernel's execution.

When writing a program accelerated by CUDA, the programmer writes the parallel section as a kernel to run on the GPU and writes the parallel section's supporting code to run on the CPU. The block size, grid size, and all of the kernel's arguments are defined in the host code. The size of the grid and its thread blocks are typically determined by the sizes of the inputs or outputs for a particular launch. While executing, each GPU thread is able to determine its own unique global index by accessing both the thread's index inside the block and the block's index inside the grid. Figure 3.3 shows how the global index allows each thread to access a different input from global memory despite running the same code. A common CUDA kernel thread layout will have each thread access just one element from multiple large input vectors in global memory to run computations on. Every thread runs the same computation on a different element in the inputs. With many threads running on many CUDA cores simultaneously, many computations can be executed at once. As a result, GPU based processing can have much higher throughput than CPU based processing for certain problems.

Certain algorithms can achieve a massive speedup when performed on a GPU instead of a CPU. CPUs are generally designed for low latency and GPUs are designed for high throughput [9]. Part of the GPU's design is a relatively slow clock speed, so when running only a single thread, a CPU will be faster. However, a GPU can have thousands of threads running at once. If all threads are performing the same computations independently, large scale problems can be processed much faster on a GPU due to the sheer number of threads simultaneously executing. However, the high throughput focused design is not useful for all situations. If the number of threads executing at once is too small, running the problem on the

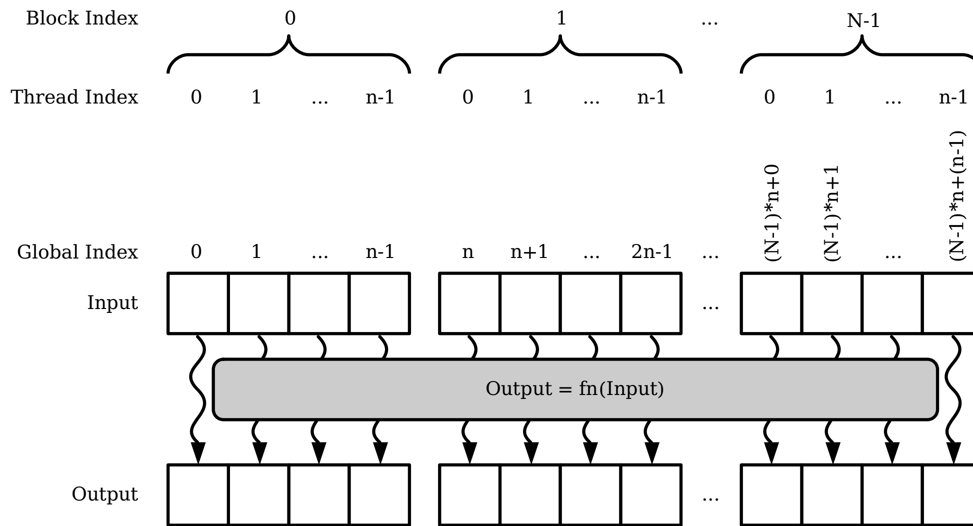

Figure 3.3: Parallel thread execution on different inputs

GPU will be slower than a CPU due to the the slower clock speed and the overhead from launching a kernel and transferring data between host and device memory. Additionally, any algorithms that must be computed in specific sequential steps like the Fibonacci sequence are not suitable for parallel processing. The best use case for a GPU is a massively parallel computation where thousands of threads all perform relatively small, independent tasks. Most of a GPU accelerated program's speedup comes from the sheer number of threads executing at once, not the speed of the threads.

3.3 MATLAB and CUDA

MathWorks's Parallel Computing Toolbox provides a variety of options for parallel computing in a MATLAB program, such as general purpose GPU programming. When using the toolbox to take advantage of GPU hardware, the principal requirement is to store variables used in parallel computations in device memory.

This is accomplished by creating *gpuArray* objects, which act like any other multidimensional array variable in MATLAB. They can be accessed from any point in the code, but their actual data are not stored in the host memory. To copy data from device memory to host memory, the *gather* function will take in a *gpuArray* object and return an array object with a copy of the data to be stored in host memory. With the ability to transfer data between host and device, the parallel computations can be launched in a variety of methods.

The simplest method of running MATLAB calculations on the GPU is to use *gpuArray* objects like any other MATLAB variable. There is usually no need for the user to rewrite any functions or to overload any operators when working with *gpuArray* objects, as parallelized implementations of these functions are already built into MATLAB. As a result, an existing MATLAB program can be modified to use GPU acceleration with very few changes. The programmer can continue using the same standard functions and operators already present in the environment, such as matrix multiplication, FFTs, and more. While this will produce noticeably faster scripts in many cases, this is not the fastest way to implement GPU parallelization in MATLAB.

A more complicated method of accessing the GPU that provides even better results than the built in *gpuArray* functions is to launch user written CUDA kernels directly from within the MATLAB script [11]. The nvcc compiler can produce a file containing parallel thread execution (PTX) instruction set architecture (ISA) code, which is a low level language for a general purpose GPU virtual environment. MATLAB can load the resultant PTX file and launch individual kernels defined in its CUDA C source code on the GPU. Scalars, vectors, and multidimensional variables can all be passed between the two languages as inputs and outputs of the kernel.

Within the MATLAB code, all inputs and outputs of the kernel will be interpreted as *gpuArray* objects. For compatibility, the CUDA C kernel must be declared so that all variables are either scalars or pointers to one-dimensional C-style arrays. Additionally, the types of all arguments must match across the source code of both languages. For example, if the CUDA C kernel declaration describes an input as a float, then MATLAB must pass the kernel a scalar single from its perspective. Similarly, if the kernel has an array of double2 values as an input, MATLAB must pass a non-scalar variable with the type complex double. MATLAB can pass any variable as long as the data type matches what the kernel expects. The sizes and number of dimensions of that variable do not matter, as it will be converted from column-major ordering to a one-dimensional array for the CUDA code to interpret as shown in Figure 3.4.

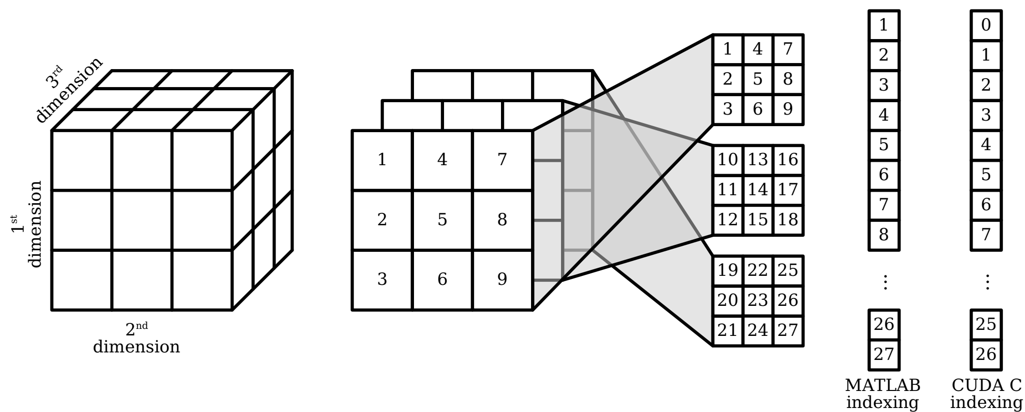

Figure 3.4: Interpreting a multidimensional MATLAB variable as an array in CUDA

The combination of MATLAB and CUDA C was chosen for the DSAR simulation implementations described later in Chapter 4, as both languages are appropriate for their respective sections of the program. Some portions of the simulation are subject to frequent change. The placement of scattering patches, the transmitted

pulses and their matched filters, and image formation algorithms are all examples of what may be modified at some point to test different scenarios. Interpretability and flexibility are necessary for the script's users to make these changes quickly. MATLAB's collection of native parallel functions provides a useful speedup while retaining all the advantages of MATLAB code, so the simpler *gpuArray* technique is used for the scene setup, signal processing, and all other supporting routines. While the speedup is reasonable for relatively small-scale calculations, higher performance is necessary for the larger scale calculations. Computing the DSAR datacube as modeled in (2.18) is the most computationally intensive portion of the simulation. With only native MATLAB GPU acceleration used in the data collection calculations, it is not uncommon for the simulation to take up to a week to execute. Writing a custom CUDA kernel will likely provide an appropriate speedup, as launching external CUDA C code is currently the fastest method of implementing GPU acceleration within a MATLAB script. CUDA code is not nearly as simple to understand and write, so it is not always appropriate to use. However, kernels implemented in CUDA C circumvent interpreted language overhead and provide lower-level control, giving it a very high potential for implementing fast simulations.

Chapter 4

Implementation

Previously, Chapter 2 described the fundamental concepts behind DSAR data collection and the signal model used to simulate it. Chapter 3 described the programming models and languages used in the simulation's implementation. Now, the implementation of the simulation program itself can be discussed. The simulation has been developed specifically for satellite DSAR and does not use a simplified flat Earth model. Instead, the more accurate ellipsoidal model of the Earth, WGS84, is used [12]. However, this is still an approximation, as the Earth's topography is ignored. The main sections of the program are the scene preparation, the datacube calculation, and lastly, the image formation. This chapter describes the implementations of the routines needed to compute simulated DSAR data. First, the simulation's input scenario and its preparation are introduced. Next, the supporting code for the main calculation is explained. Finally, all implementations of the datacube calculation are discussed, including both successful and unsuccessful attempts of exceeding the speed of the native MATLAB GPU acceleration. The performance of all implementations will be numerically analyzed later in Chapter 5 for comparison.

4.1 Preparation

The first step in simulating DSAR is defining the scenario for data to be simulated, which serves as the input to the datacube calculation. The input scenario includes multiple variables, namely the flight paths of the satellite platforms, the pulse waveforms emitted by the transmitters, and the target scene on the Earth's surface to image. Typically, the preparation of this input is computationally light in comparison to other tasks within the simulation program, but it is not simple. It must be flexible enough to define various simulation scenarios. Additionally, there is a high likelihood that new, currently unsupported scenarios will be proposed for simulation in the future, so the code responsible for the preparation must also be flexible. The requirements favor flexible code over high performance, so MATLAB is the chosen language for the simulation preparation as it will make complicated modifications easier and faster to implement than they would be for other languages.

The constellation must be defined generally enough to reflect any possible formation. To accomplish this, each satellite's flight path is independently defined as a list of Earth-centered, Earth-fixed (ECEF) coordinates. The ECEF coordinate system is a three-dimensional Cartesian coordinate system in which the origin is located at the Earth's center, the $+x$ axis intersects with the equator and the prime meridian, the $+y$ axis intersects with the equator and is 90 degrees East of the prime meridian, and the $+z$ axis intersects with the North Pole [12]. The ECEF coordinate system rotates with the Earth, which eliminates the need to account for its rotation explicitly in the simulation. Each coordinate in a list represents the location of one satellite during one PRI. All lists are arranged in chronological order and synchronized, mapping to the slow-time axis of the datacube.

The independently defined flight paths allow for both static and dynamic con-

stellations to be simulated. Even in static formations, the independence is useful, as it allows deviations from a planned flight path to be modeled on a platform-specific basis. All satellite paths are generated from a formula based on user defined parameters, such as velocity, altitude, and PRI. Such parameters are quick to change, allowing the user to tune the configuration for better image quality. The formula is specific to one type of formation, such as a “string of pearls” in which all satellites move in tandem, forming a straight line parallel to their direction of travel. MATLAB’s flexibility allows for new formulas to be added fairly quickly.

The transmitters and their unique pulse waveforms are also defined in the simulation preparation. All satellites will be receivers, but any subset of the satellites can be configured as transmitters. Similar to the satellite flight paths, every transmitter’s pulse waveform is defined independently. These signals can be created in the time domain initially, but need to be stored in the frequency domain for use in the datacube calculation. All signals are defined along the same frequency axis as used to represent the pulse returns in the frequency domain datacube. Again, similar to the satellite flight paths, all independent signals are typically generated based on a formula with input parameters including bandwidth, pulse width, and carrier frequency. New signal formulas are easy to implement for new simulation scenarios.

The last major part of the simulation scenario to prepare is the target scene, which includes the pixel grid for backprojection and the scattering patches for the data simulation. The backprojection pixel grid for imaging is not used in the data simulation, but it is prepared alongside the scattering patches since they both depend on the same geometry. Since the simulation uses an ellipsoidal model of the Earth, the illumination area is a curved surface. The location and size of the illumination area are calculated from the midpoint of the average flight path of all satellites,

how the illumination beam is aimed, and the approximated surface of the Earth. Locations for the pixel grid cells and the scattering patches are calculated very similarly.

The pixel grid is part of the precisely known geometry required for the backprojection algorithm described in Chapter 2 and is not used until image formation, which takes place after the data simulation. In memory, the backprojection pixel grid is stored as a list containing the location of the center of every cell in ECEF coordinates. Conceptually, the grid fills the entire illumination area and is fitted the curved surface of the Earth. To calculate the positions of the grid's cells, a plane that contains the points at which the maximum and minimum range of the illumination beam intersect with the Earth' surface and is parallel to the constellation flight path at its midpoint is calculated. Every cell is placed on this plane at a constant, not necessarily square spacing corresponding to the ground-range and cross-range resolutions of the radar to form a two-dimensional grid. The center of each cell is then projected onto the Earth's surface to make the grid match its curvature. This placement is not mathematically perfect, but it is an acceptable approximation considering the size of the target scene in comparison to the size of the Earth. A mathematically perfect representation would involve ellipsoidal integrals, which are very computationally intensive since they can only be solved numerically.

The scattering patches compose the part of the target scene used in the data collection simulation. Regardless of how they are arranged, they are all stored and interpreted in the same way. Every scattering patch is stored as an ECEF coordinate representing the location of its center and a complex number representing its reflectivity coefficient. The position of every scattering patch is calculated in the same as the pixel grid cells, by placing them on a two-dimensional plane and projecting them onto the Earth's surface. Currently, scattering patches can be arranged

to represent either individual point targets or a two dimensional grid.

Individual point targets are useful to examine how a target will be imaged at different locations within the view of the radar. Any number of targets can be placed at arbitrary two-dimensional offsets from the center of the illumination area plane before fitting to the Earth. All point targets have the same normalized reflection coefficient with no phase offset. Figure 4.1 shows an example output image from a simulation with a single point target located at the center of the target scene.

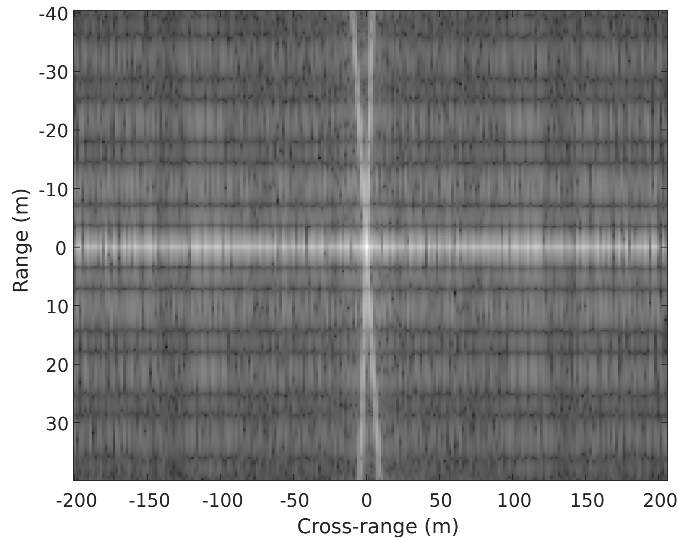

Figure 4.1: Simulated SAR image of a single point target

An arrangement of targets depicting a two-dimensional image is useful to examine performance at a high level. To accomplish this, scattering patches are placed in a grid on the illumination area plane before fitting to the Earth based on a user defined input image. The number of rows and columns in the scattering patch grid match the dimensions of the input image so that every scattering patch corresponds to single pixel of it. The size of the scattering patch grid is unrelated to the size of the backprojection grid, so may not fill the illumination area entirely or may be bigger than the illumination area. The magnitude of every scattering patch's

reflectivity coefficient is determined by the brightness of its corresponding pixel, ignoring color. Completely dark pixels in the input image correspond to a reflectivity coefficient of zero. The phase of every scatterings patch's complex coefficient is randomly set to recreate the speckled look of real SAR images. The spacing of the scattering patches depends on the radar's resolution and whether or not it will be oversampled. Without oversampling, the dimensions of each scattering patch are equal to the ground-range and cross-range resolutions, just like the backprojection grid. With oversampling, the dimensions of each scattering patch are divided by an oversampling factor, making the scattering patch placement more dense.

Typically, the input image will either be a simple test pattern or an image similar to what could be captured by a real SAR. Test patterns like a checkerboard or concentric rings are helpful to plainly show visual aberrations such as signal contamination, while a real SAR image is useful to show what the simulated parameters may produce in practical applications. Figure 4.1 shows an example output image from a simulation with targets arranged in a checkerboard pattern at the center of the target scene.

All of the simulation input data are created and stored in the MATLAB environment. Figure 4.3 shows how all parts of this input are stored across multiple variables.

The satellite flight paths are stored in three matrices. There is one matrix of size $N_{pulse} \times N_{RX}$ for each axis of the ECEF coordinates. The columns correspond to individual receiving satellites and the rows correspond to individual pulses or PRIs. All matrices are aligned such that a satellite's position during one pulse is stored in the same coordinate across all three matrices.

The transmitter information is stored in a matrix containing the pulse waveform spectra and a column vector of frequencies. The matrix has a size of $N_f \times N_{TX}$

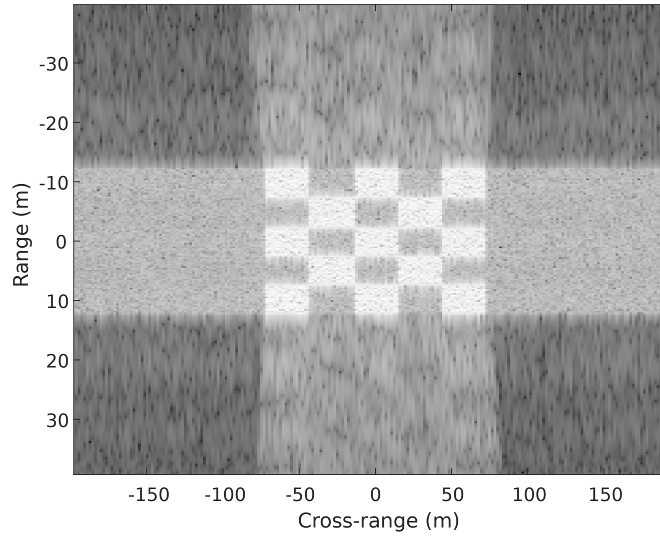

Figure 4.2: Simulated SAR image of a checkerboard pattern of targets

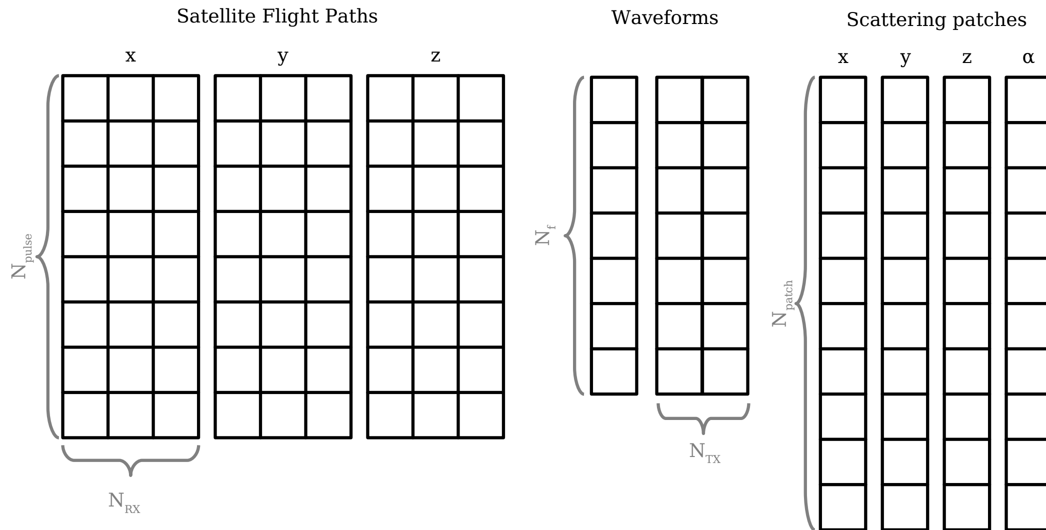

Figure 4.3: DSAR simulation input data as stored in the MATLAB environment

where each column represents a different transmitter's pulse waveform spectrum. Every element in the waveform spectrum is a complex value representing the amplitude and phase of a frequency component in the signal. The frequency vector has a length of N_f and is aligned with the matrix to represent the frequencies associated with the frequency components in the waveform spectra.

Finally, the scattering patch information is stored in four separate column vectors of length N_{patch}. Three of the vectors each store a different dimension of the ECEF coordinate location and the fourth vector stores the complex reflection coefficient. The four vectors are aligned so that every row of every vector corresponds to the same patch.

4.2 Supporting Code for Datacube Calculations

After the input has been created, the data collection can be simulated. Specifically, the signal model in (2.18) can be used to calculate the DSAR datacube. This section of the simulation program has different requirements than the scene preparation. The DSAR signal model itself will never change, and by accepting the previously described input format, the implementation of it will inherently be general enough to accept any potential simulation scenario. The code responsible for this calculation is much less likely to change than the simulation preparation. Additionally, the calculation is very computationally intensive. Input scenarios with a large amount of data produce large datacubes and require many computations. To run in reasonable time, a lot of throughput is required. Since this part of simulation requires performant, high-throughput code but does not require it to be easily modifiable, CUDA C was chosen as the programming language to implement the data collection simulation.

The CUDA C code is launched from the MATLAB script as explained in Chapter 3. Supporting code written in MATLAB is required to load the input data onto the GPU, offload the output data from it, and launch the kernel. However, for this simulation, simply loading all the input data at once is not feasible.

The complicating factor of running general DSAR simulations on GPU hard-

ware is the amount of input data required. A typical simulation is a very large scale computation. The number of PRIs in the data collection is usually in the range of thousands. The pulse waveforms have high bandwidths in the range of thousands or tens of thousands of frequency components (or fast-time samples), meaning the IFFT-length is in the thousands or tens of thousands. When the scattering patches are arranged to form an image, the number of patches can easily be in the range of hundreds of thousands. Together, these inputs can require hundreds of gigabytes of memory to store. This presents a problem, as GPU threads can only interact directly with device memory, which is very limited. By today's standards, a high-end workstation might have enough RAM to store the data of such a simulation in host memory, but even the best workstation graphics cards do not have enough video RAM to hold all the data in device memory.

To work around the device memory limitation, the supporting MATLAB code splits the inputs and outputs into subsets that fit within device memory. Instead of computing the entire datacube within one kernel launch, it is computed in multiple kernel launches. Each launch computes a different segment of the datacube, all of which are later combined within host memory to form the complete output. When necessary, the scattering patch, waveform, and/or satellite flight path variables split across rows into subsets. The entirety of each variable is stored in host memory at all times, but only one subset from each input will be stored in device memory at any point throughout the simulation. The size of the subsets are modeled as follows:

For an input x, given its total number of elements N_x and the number of subsets s_x to partition it into, the number of elements in all but the last subset is

$$n_{x,s'} = \left\lceil \frac{N_x}{s_x} \right\rceil \quad \forall\, s' \in \{1, 2, \ldots, s_x - 1\} \tag{4.1}$$

and the number of elements in the last subset is

$$n_{x,s_x} = N_x - \left\lceil \frac{N_x}{s_x} \right\rceil (s_x - 1) \tag{4.2}$$

where $n_{x,s}$ is the number of elements of the sth subset of x. If N_x is evenly divisible by s_x, then all subsets will be the same size. Otherwise, all subsets will be the same size except for the last subset, which will be smaller. The maximum possible difference in size between the last subset any other subset is $s_x - 1$.

It is worth noting that in (4.2), not all values of N_x and s_x will produce valid results. An invalid subset configuration will result in the size of the last subset being less than one. This never occurs when $s_x << N_x$, which is typically true. However, correcting invalid subset configurations algorithmically is as simple as checking if $n_{x,s_x} \leq 0$ after computing (4.2). If yes, then we subtract 1 from s_x, repeating until the subset configuration is valid. Figure 4.4 shows an example of a valid subset configuration of a scattering patch input with $N_{patch} = 10$ and $s_{patch} = 4$.

Roughly all subsets of a single input are equal size, which offers two main benefits. First, the equal size simplifies memory allocations. In MATLAB, it is possible to query the total amount of free device memory, but it is not possible to query which specific areas of device memory are free. Despite the appearance of enough free memory, a large allocation may fail due to memory fragmentation. In this event, increasing the number of subsets is an easy way to find an appropriate subset size so that a reasonable amount of elements from every input can be loaded at once. Additionally, equal subset sizes will cause every kernel launch to have roughly the same amount of thread utilization and roughly the same execution time. This makes it possible to extrapolate the total run time more accurately after timing just one kernel launch.

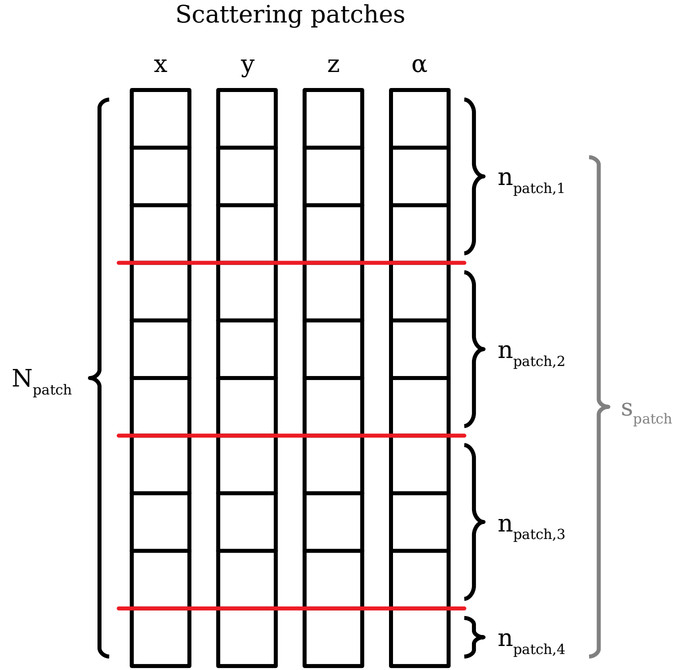

Figure 4.4: Example scattering patch partition

The datacube computation is broken into subsets determined by the subsets of its inputs. The datacube itself is split into segments to match the Cartesian product of all input subsets. Each individual segment is the output of an individual kernel launch. The datacube's frequency axis is split into subsets matching those of the transmitter waveform input. Similarly, the slow-time axis is split into subsets matching those of the satellite flight path input. The receiving channel axis, however, is not split into subsets, as it is already small. The resulting datacube subsets are split further into segments according to the subdivision of the scattering patch input. All segments of the same datacube subset represent different parts of the summation along scattering patches in the signal model. Figure 4.5 shows a datacube computation split into subsets by the input subset counts $s_f = 2$, $s_{pulse} = 2$, and $s_{patch} = 2$.

The final datacube is assembled in host memory from all of its segments com-

puted in device memory. Conceptually, each completed subset of the datacube is calculated by summing all of its segments like three-dimensional matrices. Then, all completed subsets are arranged side by side to form the completed datacube. Even though the entire computation does not fit in device memory, it is possible to complete it in separate segments.

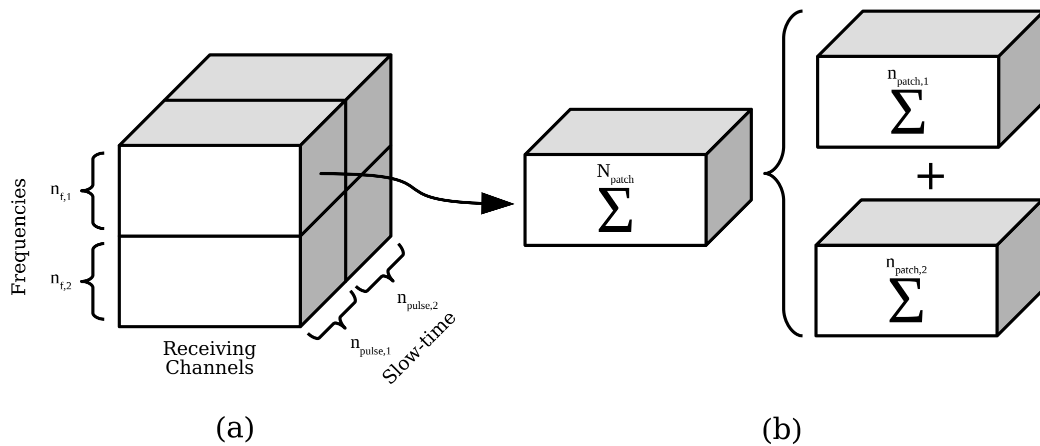

Figure 4.5: (a) The subsets of the output datacube and (b) the scattering patch segments of an output datacube subset

The subdivision of the output datacube is accomplished by a set of nested for loops in the supporting MATLAB code. In host memory, a MATLAB variable with the dimensions of the final output datacube is initially populated with zeros and acts as a running sum. Each loop iterates over the subsets of an input and loads them into device memory. In order from outermost to innermost, the for loops iterate over scattering patch subsets, satellite flight path and slow-time axis subsets, and pulse waveform and frequency axis subsets. Inside the innermost loop, the CUDA kernel operates on the loaded subsets to calculate a segment of the datacube. Immediately after the kernel has executed, its output is transferred from device memory to host memory and added to the running sum in the appropriate place. Once all loops are finished iterating, all segments have been added to the running sum, meaning the

frequency domain datacube is complete and can be used to form an image.

4.3 Datacube Calculations

With the problem split up into manageable segments, the GPU is now capable of performing all the signal model calculations. Following the signal model in (2.18) directly, every element in the datacube can be computed independently in $O(N_{TX} \cdot N_{patch})$ time. The entire datacube can be computed in $O(N_f \cdot N_{RX} \cdot N_{pulse} \cdot N_{TX} \cdot N_{patch})$ time. This section of the simulation is the main focus for improving performance.

Regardless of how it is implemented, the output will be the same. Conceptually, every complex value in the datacube represents one frequency component in the discrete Fourier transform of a single receiver's measurement within one PRI. This measured signal is the sum of all pulse waveforms originating from different transmitters, reflecting off of every scattering patch in the target scene, returning to one receiver. In the signal model, the inner summation adds all phasors from the discrete reflections of a single transmitter's pulse and the outer summation adds the sums of all transmitters together.

Every phasor within the calculation a single cell of the datacube has an independent time delay. The time delay is directly related to propagation distance, and for every scattering patch, each transmitter signal travels over a different distance. As shown in Figure 4.6, three transmitting satellites labeled A, B, and C will each have a different one-way distances, d_a, d_b, d_c, respectively, from their location to a single point on the Earth. When measuring the returns within a PRI, satellite A receives an echo of its own waveform that traveled a distance of $d_a + d_a$, an echo of Satellite B's waveform that traveled a distance of $d_a + d_b$, and an echo of Satellite

C's waveform that traveled a distance of $d_a + d_c$. Satellites B and C similarly receive the three waveforms, all at different distances. Since each output cell contains the value of one frequency component of one receiver's total measurement, there are $N_{TX} \cdot N_{patch}$ unique time delays used in the calculation of a single cell of the datacube.

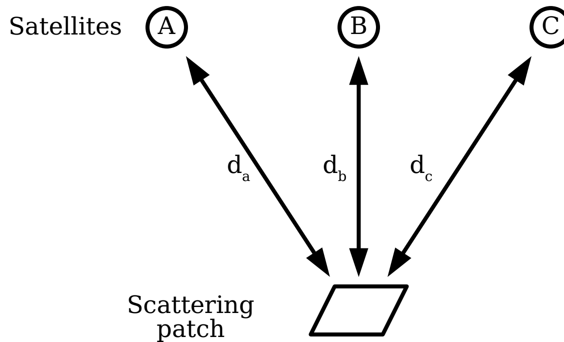

Figure 4.6: Propagation distances between satellites and a single scattering patch

Overall, simulating DSAR data is computationally intensive. Due to the size of the datacube and the fact that all cells within it can be calculated independently, the simulation can easily be implemented in a highly parallel model. However, in a typical scenario with a large number of scattering patches, the computation of an individual cell itself is relatively intensive, too. The signal model cannot be simplified any further to reduce the number of calculations, so optimizations must be found for the simulation to execute within a reasonable amount of time. Reusing intermediate calculations across multiple cells can potentially improve performance, but memory bottlenecks must be avoided for there to be a net gain in execution speed.

In the past, a known working datacube calculation routine written purely in MATLAB was developed. It used standard linear algebra operations to compute the simulated data and was accelerated with the native GPU support in MATLAB.

Multiple implementations of the datacube calculation have been created in CUDA C in an attempt to calculate the simulated data faster. All of them accept the previously described input format and, given the same input, produce the same datacube. The goal in creating multiple implementations is to evaluate multiple methods and identify the one that produces the correct result the fastest. A correct result is one that produces the correct SAR image of the input scenario, matching the output image of the purely MATLAB-based simulation program. The remainder of this chapter introduces and explains all implementations, each in their own subsection.

4.3.1 Separate Distance and Output Kernels

The first implementation is an attempt to minimize the amount of two-way distances calculated. The time delays $\tau_{m,n,q,i}$ within a single PRI have the highest potential for reuse since they are not influenced by the frequency axis. All values in a single vector along the frequency axis of the output datacube represent a single receiver's measurement at one position, so every value in the vector can be calculated with the same set of time delays. Reusing them significantly reduces the total number of operations in the scope of the entire datacube calculation, as only $N_{TX} \cdot N_{patch}$ two-way distances are calculated per frequency axis vector rather than $N_{TX} \cdot N_{patch} \cdot N_{freq}$ distances per vector.

There are two separate kernels at different positions in the nested subset loops for different stages in the calculations. There is also an additional loop between the slow-time subset loop and the frequency axis subset loop that iterates over every individual transmitter. This added loop causes each frequency axis iteration to load only one transmitter's pulse waveform into memory, so the datacube subsets are split further into segments by transmitting channel in addition to the scattering patch

subset segments. The first kernel is executed immediately inside the transmitting loop before the frequency axis loop. It calculates all two-way distances from the current transmitter to the loaded scattering patches to all receivers. The second kernel is executed immediately inside the innermost loop and uses the distances from the first kernel to calculate a segment of the datacube, which is then added to the running sum of the entire datacube. Within each iteration of the transmitter loop, the two-way distance kernel, the first kernel, is executed once and the output kernel, the second kernel, is executed N_{TX} times with the same set of pre-calculated two-way distances. Figure 4.7 shows a pseudocode representation of the supporting code for both kernels.

```
for all scattering patch subsets do
    Load a scattering patch subset into global memory
    for all pulse subsets do
        Load a pulse subset into global memory
        for all transmitters do
            Launch distance kernel
            for all frequency subsets do
                Load frequency axis subset into global memory
                Load pulse waveform subset of transmitter into global memory
                Launch output kernel
                Transfer output to host memory
                Add kernel output to total output
            end for
        end for
    end for
end for
```

Figure 4.7: Pseudocode of the supporting code for the two-kernel approach

When the two-way distance kernel is launched, the grid and blocks are arranged so that there is one GPU thread per combination of scattering patch, receiving satellite, and slow-time interval. Each thread first loads its scattering patch location, the location of the current transmitter at its slow-time interval, and the location of its

receiver at its slow-time interval from global memory. Each thread then computes the distance from the transmitter to its associated scattering patch, the distance from its associated scattering patch to its associated receiver, and the sum of the two distances, resulting in the desired two-way propagation distance. Finally, each thread writes its one output to an element of a three-dimensional matrix stored in global memory. This matrix of two-way distances has dimensions corresponding to the size the current slow-time subset, the number of receiving channels, and the size of the current scattering patch subset. All distances calculated in this kernel are used multiple times throughout each following launch of the output kernel.

The output kernel calculates a segment of the datacube subset corresponding to one transmitter waveform and all currently loaded input subsets, as shown in Figure 4.8. This kernel's grid and blocks are arranged so that there is a single thread for each element in the datacube subset segment it is to calculate. First, the thread loads the frequency for its element in the datacube and the phasor representing the transmitter waveform's sinusoid component at the same frequency. Next, the thread has a loop that iterates once for every scattering patch in the input subset. In every iteration, the scattering patch's complex reflectivity coefficient and the two-way distance for the current datacube element are loaded from global memory. The thread then calculates one phasor in the signal model's inner summation and adds it to a local running sum. After every iteration of this loop is complete, the sum of all scattering patch phasors is then multiplied by the transmitter waveform component phasor and stored in the output datacube segment stored in global memory. Each element in the output segment is similar to the original signal model, the only difference being that a subset of scattering patches were summed instead of all of them. When the MATLAB code running on the host sums all segments together, the signal model of the final output is equivalent to the original model.

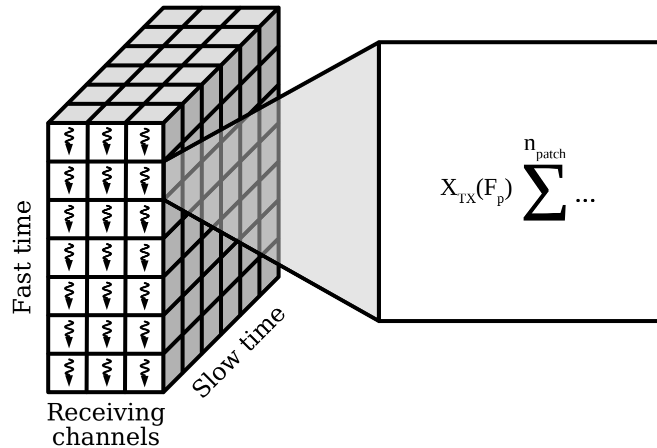

Figure 4.8: Output kernel of the two-kernel approach

With the two-way distance kernel launch placed in a shallower level of loop nesting than the output kernel, the number of two-way distance calculations performed is minimized. The same two-way distance matrix is used for every following iteration of the frequency axis subset loop. As a result, every unique two-way propagation distance that appears anywhere in the entire datacube computation is calculated only once.

This implementation produced the correct results, but was not faster than the approach based on native MATLAB GPU acceleration. The total number of one-way distance calculations could have been reduced even further, but the two-way distance kernel took a negligible amount time to execute compared to the output kernel, so the distance calculations were determined not to be the limiting factor of performance. It is likely the global memory loads used throughout the loops in each thread of the output kernel that degraded performance the most. When loading data from the three-dimensional matrix of two-way distances, the global memory access was strided, thus limited memory bandwidth. Since this approach did not perform well, the following approach was created as the next attempt at improving the DSAR data collection simulation performance.

4.3.2 Monolithic Kernel

The second implementation is an attempt to minimize the amount of one-way distances calculated without relying on global memory to store intermediate calculations. Within a single slow-time interval, there are $N_{TX} \cdot N_{RX}$ unique two-way distances associated with each scattering patch. However, all of these two-way distances can be calculated from just N_{RX} unique one-way distances. Since every transmitter is also a receiver, every transmitter and receiver pair has a two-way propagation distance equal to the sum of two satellite's one-way distances to the scattering patch.

Unlike the previous two-kernel approach, there is no loop iterating over transmitters and each segment contains the returns from all transmitters. In this implementation, there is only one kernel responsible for computing the datacube segments, which is launched once per iteration of the most deeply nested loop, the frequency axis subset loop. The kernel computes all the necessary propagation distances internally. Reducing global memory usage is a higher priority than reusing calculations, so time delay values are not shared across elements within the same frequency axis vectors. Every time the monolithic kernel is ran, it computes a single output datacube subset segment from the input subsets. Figure 4.9 shows the pseudocode for the monolithic kernel's supporting code.

When the monolithic kernel is launched, the grid and blocks are arranged so that there is a thread for every vector in the datacube along the receiving channel axis, which corresponds to one frequency and pulse pair in the datacube subset, as shown in Figure 4.10. Compared to the output computation of the two-kernel approach, every thread is responsible for significantly more work. One thread computes values for all elements in a receiving channel vector in the datacube.

for all scattering patch subsets **do**
  Load a scattering patch subset into global memory
  for all pulse subsets **do**
    Load a pulse subset into global memory
    for all frequency subsets **do**
      Load frequency axis subset into global memory
      Load pulse waveform subset of all transmitters into global memory
      Launch output kernel
      Transfer output to host memory
      Add kernel output to total output
    end for
  end for
end for

Figure 4.9: Pseudocode of the supporting code for the monolithic kernel approach

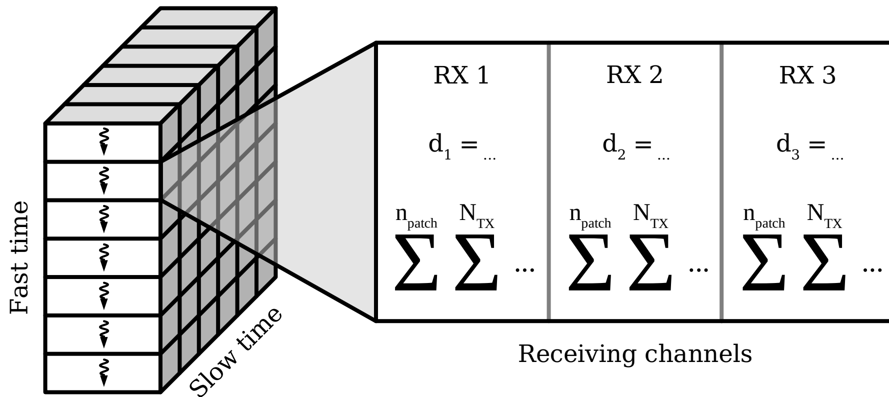

Figure 4.10: The monolithic kernel

Every thread loops over all scattering patches in the current output segment while tracking a separate running summation for every receiving channel. Within every iteration of this first loop, the thread immediately loads all N_{RX} satellite positions at the current slow-time interval and calculates the one-way distances between the current scattering patch and every satellite, storing them as variables in the scope of the iteration.

Next, still inside the first loop, a second loop iterates over every possible combination of transmitter and receiver. Within every iteration of this inner loop, two

one-way distances are added to compute a time delay, which is then used to compute a single phasor of the signal model's inner summation. This phasor is then multiplied by the transmitter waveform's component phasor corresponding to the thread's position along the frequency axis. The value computed in this inner loop is modeled as

$$X_n\left(F_p\right)\alpha_i \exp\left(-j2\pi\left(F_0+F_p\right)\tau_{m,n,q,i}\right). \tag{4.3}$$

After calculation, this value is added to the running sum of the appropriate receiver. Thus, each thread computes all N_{RX} separate outputs by the model

$$\sum_{n=1}^{N_{TX}}\sum_{i=1}^{n_{patch}} X_n\left(F_p\right)\alpha_i \exp\left(-j2\pi\left(F_0+F_p\right)\tau_{m,n,q,i}\right). \tag{4.4}$$

After all scattering patches in the loaded subset have been iterated over by the first loop, all N_{RX} running sums are written to the output datacube segment in global memory. The MATLAB code running on the host then adds the segment to the full datacube. When all kernel outputs have been added to the total output, the datacube is complete and matches the final signal model.

All threads corresponding to datacube elements occupying the same slow-time index calculate the same one-way distances. Computationally, this is inefficient, as it is possible to avoid repeating any distance calculations. However, this approach was focused on minimizing the number of one-way distance calculations without relying on sharing distance values across multiple threads. One beneficial side effect from this is that more global memory is free to store inputs and the output, so planning memory allocations is easier and the datacube can be computed in fewer segments.

The monolithic kernel is a significant improvement over both the two-kernel

approach and the native MATLAB GPU acceleration approach despite the amount of redundant distance calculations. The increased performance is most likely due to the difference in memory accesses and how memory is cached on the GPU. On current Nvidia GPU architectures, data loaded from global memory is routed through multiple caches. Each SM in the GPU has its own L1 cache with similar speed to the shared memory and the GPU has a single L2 cache between all SM's and global memory. Although accessing the global memory in the L1 cache is not quite as fast as accessing shared memory, it is still much faster than uncached global memory [13].

In the output kernel of the two-kernel approach, a thread loads a different scattering patch reflectivity coefficient and distance for every phasor it calculates. Comparatively, in the monolithic kernel, a thread will reuse the same scattering patch and satellite flight path data to calculate $N_{RX} \cdot N_{TX}$ phasors before loading new data. When the same data are accessed multiple times sequentially, there is a higher chance of a cache hit due to its temporal locality. Additionally, since the monolithic kernel only reads data from the scattering patch inputs, the cache is not invalidated throughout its execution. Overall, the monolithic kernel computes the correct DSAR datacube more quickly than any previous implementation.

4.3.3 Shared Memory Usage

In an attempt to attain even better performance than the monolithic kernel approach, both the two-kernel and monolithic kernel implementations were rewritten to use shared memory. In both new implementations, the shared memory is used as a cache to pre-load portions of scattering patch data from the subset in global memory. These attempts were motivated by the fact shared memory is faster than

global memory. It was thought that using shared memory could have potentially reduced the amount of time spent loading scattering patch data from its subsets.

Shared memory is used in the same way in both approaches. Only the kernels that calculate the datacube segment, the output kernel of the two-kernel approach and the one monolithic kernel of the other approach, pre-fetch scattering patch data. Conceptually, the subset of scattering patches loaded from host memory into the device's global memory is split further into even smaller subsets to be loaded from global memory into shared memory. These subsets within shared memory all have the same size equal to the number of threads in a block, although the last shared memory subset is smaller when the subset in global memory is not evenly divisible by the number of threads. The shared memory subsets are present only within a kernel launch, so the output is unchanged from the implementations without shared memory.

In code, the shared memory pre-fetching is implemented by adding another loop to the GPU threads. This new loop iterates over the subsets in shared memory and encompasses the original loop iterating over individual scattering patches. At the beginning of every iteration of this new loop, all threads in a block collaborate to load a subset of scattering patch data from the scattering patch subset in global memory. Every thread in the block loads one scattering patch based on its index. After the shared memory subset has been loaded, the threads then independently loop over all loaded scattering patches as before in the original implementations. When the scattering patches loaded in shared memory have been exhausted, the next iteration of the shared memory subset loop begins, repeating the process until the entire datacube segment has been completed.

Neither of these implementations performed better than their original counterparts, although they did not perform any worse. The use of shared memory did not

significantly change the execution time despite its advantage over global memory. Although the ratio of global memory accesses to computations within each thread was smaller, the increased speed of shared memory was not enough to overcome the additional overhead introduced by pre-fetching. Extra calculations and instructions were required to ensure the threads cooperate to load the correct subsets from global memory and loop over them. The overhead was too large for the shared memory speed to overcome, so the shared memory implementations were not faster than the approaches relying on the global memory cache. Since the performance did not change with this approach, shared memory techniques were not explored any further.

4.3.4 Eliminating Repeated Distance Calculations

Other potential approaches were explored to further reduce the number of repeated propagation distance calculations without storing data to global memory, but they all proved to be impractical. The central idea was to have threads that correspond to entire frequency axis vectors in the datacube. This arrangement would allow for all frequency components in a receiver's measured pulse reflection to use the same distance calculations. Theoretically, no one-way or two-way distance calculations would be repeated, thoroughly reducing the total amount of operations per kernel launch. Two approaches of implementing this idea were briefly considered, but after further consideration, both proved to have fundamental flaws.

The first potential implementation would have had a single thread per frequency axis vector. Each thread would compute the summation of phasors for every frequency component. This is impractical for two reasons. First, there would only be $N_{RX} \cdot N_{pulse}$ threads total. N_{pulse} is commonly in the range of 1,000 pulses in

typical simulation scenarios. Under these scenarios, there are not enough threads for a powerful GPU to be at maximum utilization. Second, every thread would have to write N_f complex values to global memory sequentially in every kernel launch. Since global memory access is very slow relative to other operations, this would decrease the overall speed dramatically.

The second potential implementation would have a group of multiple threads assigned to each frequency axis vector. Each thread would work on different subsets of scattering patches and all frequencies. In this case, there would likely be enough threads to achieve maximum GPU utilization. However, the fundamental flaw is that each thread would still must write N_f values to global memory. In addition, more overhead would be required to synchronize the threads to avoid race conditions since all threads assigned to the same vector each write to the same set of memory locations.

After considering these concepts, it was clear that they would not perform better than previous approaches. The monolithic kernel approach shared memory remains as the best implementation. More details and analysis of the performance of each implemented approach will follow in the next chapter.

Chapter 5

Results

The relative performance of each implementation of the DSAR data collection simulation is determined by how fast it can complete the datacube calculation. All implementations were introduced previously in Chapter 4. This chapter compares the measured execution time of each implementation, identifies the most important features related to their performance, and finally, describes potential future work to continue this research.

5.1 Comparisons

To evaluate the implementations discussed in Chapter 4, their execution times were measured and compared on the same machine with various input scenarios. Specifically, the real-world time spent in the frequency domain DSAR datacube computation and in the supporting MATLAB code was measured. The time spent before and after the datacube calculation, which includes the time spent on preparing the input scene and signal processing, was not considered. Since all implementations produce identical outputs, the best implementation is simply determined to be the fastest one.

To ensure all comparisons are fair, all tests were performed on the same hard-

ware. DSAR simulations are very computationally intensive, so despite being compatible with theoretically any CUDA-compatible GPU, the program will typically be executed on very powerful hardware. To mimic the typical use case, the tests were executed on a high-end desktop computer with an AMD Ryzen Threadripper 3960X CPU, 256 GB of RAM, and an NVIDIA RTX A6000 GPU. The GPU has 48 GB of video RAM, all of which is available to the MATLAB environment since the GPU is configured in the Tesla Compute Cluster mode. In this mode, the GPU is not used by the operating system to drive the display. The computer has another GPU for the display which is not involved in the simulation.

The performance tests consist of various input scenarios that are executed on every implementation of the simulation to compare execution times. In all scenarios, there are three satellites that all transmit and receive pulses, 19,582 frequency components in each pulse waveform, and 8,192 pulse observations. The number of scattering patches are varied across different scenarios. In all tests, the entire input and output fit in device memory, so the computation was not split into subsets. Different numbers of subsets were tested, but no significant changes in execution times were observed, so those tests were not included. Each kernel was launched with 512 threads per block to ensure maximum utilization of each SM in the test computer's GPU. Each SM can host up to 1,536 concurrent threads, so three thread blocks will completely occupy a single SM during execution. Table 5.1 shows the number of scattering patches and the execution times for each test.

One interesting observation drawn from Table 5.1 is that for both the monolithic kernel and two-kernel approaches, the implementations using shared memory show no improvement over the original implementations. In fact, the use of shared memory did not make any significant difference in execution time, as shown by Figure 5.1. For the two-kernel approach, the implementation with shared memory

	Execution Time (seconds)					
Num. Scattering Patches	Native MATLAB	Two-kernel	Monolithic Kernel	Two-kernel (Shared Mem.)	Monolithic Kernel (Shared Mem.)
3393	782.34	1336.84	947.74	1334.82	985.83
10000	2113.84	8365.83	1311.63	8699.01	1312.85
20301	4638.57	16926.15	2662.10	17606.13	2660.47
37500	8408.51	31521.30	4913.95	32781.60	4928.40
50000	10352.30	41817.30	6552.67	43497.03	6555.27

Table 5.1: Execution times of all implementations

took slightly longer on average to execute than the implementation without shared memory.

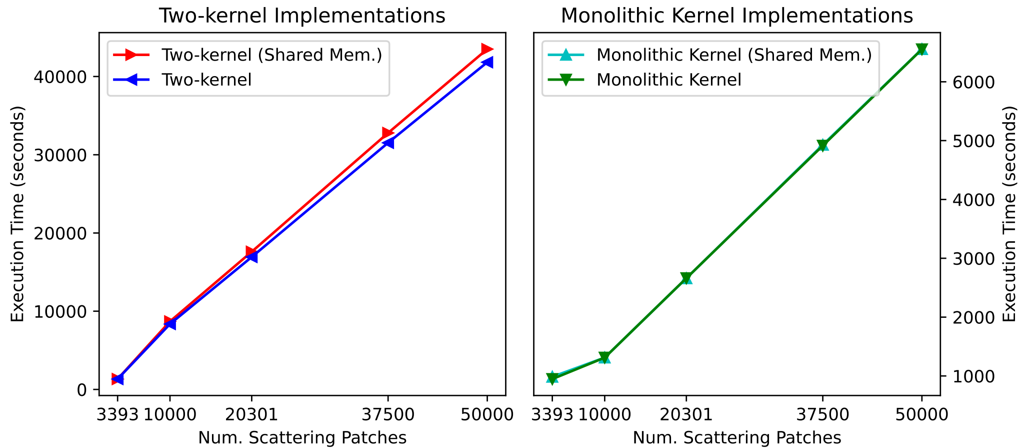

Figure 5.1: Comparisons of implementations with and without shared memory usage for both CUDA based approaches

The most significant results shown in Table 5.1 is how all the implementations without shared memory compare. The execution times of the native MATLAB, two-kernel, and monolithic kernel approaches are all plotted against the number of scattering patches for each test scenario in Figure 5.2. The two-kernel approach is the slowest implementation in all tests. Compared to the MATLAB implementation,

the two-kernel implementation is 3.84 times as slow. For the performance test with the fewest number of scattering patches, the native MATLAB implementation is the fastest. However, for all other performance tests, which are more representative of typical simulations due to the larger amount of scattering patches, the monolithic kernel approach performs the best. In tests with 10,000 or more scattering patches, the monolithic kernel is, on average, 1.66 times faster than the original MATLAB implementation, making it the fastest implementation of the simulation.

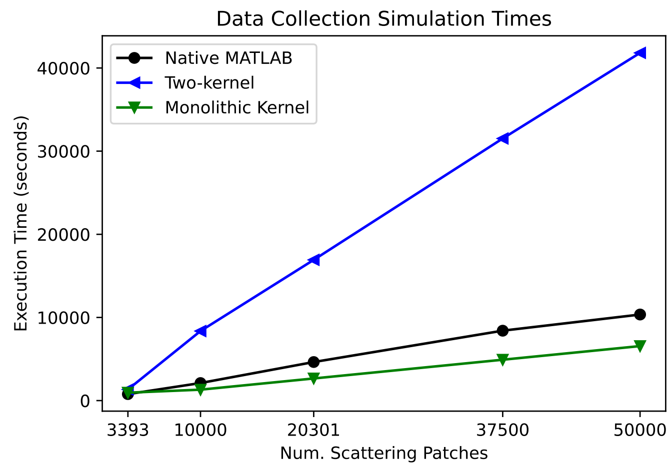

Figure 5.2: Comparing the execution times of different DSAR data collection simulation implementations

In addition to performance, the monolithic kernel has other advantages over the original native MATLAB GPU based approach due to how it uses memory. The first advantage is that the monolithic kernel uses a more sophisticated method of partitioning the datacube calculation into subsets. The original, purely MATLAB-based approach splits only the scattering patch input into subsets, so if any other input is too large to fit entirely within device memory, the scenario can not be simulated. Both CUDA based approaches can partition every simulation input parameter into subsets if necessary, so they can run any simulation that fits within host memory

regardless of how much device memory is available. The second advantage is that the monolithic kernel uses less of the device's global memory than other implementations. The purely MATLAB-based implementation uses global memory as working memory to store intermediate calculations for all of the matrix operations performed and the two-kernel approach uses global memory to store propagation distance calculations. Comparatively, the monolithic kernel approach uses only the registers and local memory of threads in the kernel to store intermediate calculations. As a result, more global memory is free for larger input and output subsets and the entire calculation can be performed in fewer kernel launches.

5.2 Conclusions

Implementing the approaches described in Chapter 4 and testing their performance as described here in Chapter 5 highlighted characteristics of the DSAR simulation problem. Although the problem presents a potential for a high level of parallelism, it difficult to map to a GPU.

The performance of all current implementations of the datacube calculation is bound by memory access speed, but making use of shared memory hardware did not improve performance. The two-kernel implementation was an attempt to minimize the amount of repeated computations in the signal model's summations, but was slower than any other implementation. The monolithic kernel implementation is different in that it repeats many computations to reduce how often threads load from memory and has significantly better performance. The reduced amount of global memory access appears to be the cause of the improved speeds. However, the test results show that loading data from global memory into shared memory does not improve speeds further because of how individual threads access that data.

Even in the pure MATLAB approach, which is based on linear algebra operations, shared memory is not very useful to this problem. Shared memory can be used to improve matrix multiplication speeds by tiling, a technique in which a subsets of the input matrices are loaded into shared memory to be computed individually. Each thread will access the same data as it would without tiling, but will access global memory fewer times. However, in the DSAR datacube calculation, there are never any two matrices multiplied together. Other operations, such as multiplying column and row vectors or multiplying vectors by matrices are used. Unfortunately, there are not many values that can be reused across different output elements, so shared memory is not useful here.

Overall, when computing on a GPU, a general DSAR simulation performs best when the number of global memory access is minimized, even if it means increasing the total number of mathematical operations performed in each thread. Typically, shared memory can be used to reduce global memory accesses, but the DSAR datacube calculation is difficult to apply shared memory techniques to.

5.3 Future Work

The obvious next step to increase the utility of this simulation is to explore methods that improve execution time even further. One potential approach that has not yet been implemented or tested would be to assign one or more thread blocks to a single element of the datacube and use a tree reduction pattern [9]. First, an individual thread would compute one phasor or the summation of a few phasors in the signal model. Then, all threads in the block would collaborate via shared memory to compute the summation of all threads' values. If there are fewer threads in a block than the number of phasors needed to calculate an element, there are

two options. The scattering patch input could be split into subsets of the same size as the thread blocks or multiple thread blocks could work on different segments of the same element with the tree reduction pattern running recursively. To ensure there are enough threads for every processor in the GPU to be utilized, multiple datacube elements, all with independent tree reductions, can be calculated within one kernel if there are not enough scattering patches loaded in global memory to populate threads with alone.

Compared to the implemented and tested approaches described previously, in this proposed approach there are fewer individual datacube cells computed simultaneously in every kernel launch, but the computation of each cell is done in parallel. As a result, less time is required to compute a single cell in the output datacube, although it is not immediately clear whether or not this approach will compute an entire datacube faster than the monolithic kernel approach.

The main benefit of the tree reduction in this application is the efficient use of memory. In each thread of the monolithic kernel, scattering patches and pulse locations are loaded from global memory multiple times at regular intervals throughout the thread's execution. This presents an issue, as arithmetic operations are relatively fast, but the processor must wait until the required operands have been loaded from global memory to perform the operation. Since global memory is slow, it can take many cycles before the addition, which only takes a few cycles, to complete. Comparatively, in this tree reduction approach, the threads will only load data from global memory at the beginning of the kernel launch. All the phasors in a single datacube cell's summation will be computed and all the required operands will be loaded at once instead of at multiple times throughout the execution of the thread. The operands of the addition instructions that sum all the phasors will be loaded from the shared memory, not global memory, so there will be less time spent wait-

ing for memory transactions to complete throughout the large summation.

The main drawback of using multiple tree reductions is the reduced overall thread utilization. At every step in a tree reduction, the number of addition operations and the number of working threads is halved. Fewer threads perform meaningful work at every iteration while the remaining threads effectively sit idly. Comparatively, the monolithic kernel approach has all threads performing meaningful work throughout the entire duration of the kernel. Whether or not the reduced utilization outweighs the benefits of this tree reduction approach is unclear. If this approach does perform better than the monolithic kernel, it is likely that the performance is is increased more for simulations with a large number of scattering patches than for simulations with fewer scattering patches. There is currently uncertainty around this tree reduction approach to the DSAR simulation, but it is worth exploring.

The utility of this simulation program can also be improved in areas outside of execution time. Support for a wider range of hardware could be beneficial. Since the simulation's input scenario can be split into subsets when necessary, the size of the simulation is not limited by the amount of device memory. However, the input scene currently must fit entirely within host memory for the simulation to run at all. Therefore, a large amount of RAM is required to run simulations of large scenes. A potential improvement is to take the subset concept a step further to eliminate the host memory limitation on simulation size. If the entire input scenario can fit on the computer's file system, then subsets of that input can be loaded from the file system into RAM. Then, smaller subsets of the RAM-hosted subsets can be loaded into device memory as the current implementations do. Temporarily storing the scenario's data in the file system would be relatively slow, but would enable machines to compute larger simulations that are currently supported. Implementing this would likely be relatively simple with MATLAB, too. MATLAB has native functionality to work

with out-of-memory data thanks to its Tall Array object. Although the simulation sizes are currently limited by the practicality of execution time, as either faster GPU hardware or faster algorithms are developed, this proposed technique prevents the amount of host memory from becoming the next limiting factor.

References

[1] M. A. Richards, *Fundamentals of Radar Signal Processing*. McGraw-Hill Education, 2014.

[2] P. Z. Peebles, *Radar Principles*. Wiley-Interscience, 1998.

[3] N. Crisp, K. Smith, and P. Hollingsworth, "Launch and deployment of distributed small satellite systems," *Acta astronautica*, vol. 114, pp. 65–78, 2015.

[4] M. I. Duersch, "Backprojection for synthetic aperture radar," 2013.

[5] J.-B. Poisson, H. M. Oriot, and F. Tupin, "Ground moving target trajectory reconstruction in single-channel circular sar," *IEEE transactions on geoscience and remote sensing*, vol. 53, no. 4, pp. 1976–1984, 2015.

[6] Y.-Q. Jin and F. Xu, *Polarimetric Scattering and SAR Information Retrieval*, 1st ed., ser. Wiley - IEEE. Singapore: Wiley-IEEE Computer Society Pr, 2013.

[7] J. Li, L. Xu, P. Stoica, K. Forsythe, and D. Bliss, "Range compression and waveform optimization for mimo radar: A cramÉr-rao bound based study," *IEEE Transactions on Signal Processing*, vol. 56, no. 1, pp. 218–232, 2008.

[8] W.-Q. Wang, *Multi-antenna synthetic aperture radar Wen-Qin Wang.* Boca Raton: Taylor & Francis, 2013.

[9] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, ser. Applications of GPU Computing Series. Burlington, MA: Morgan Kaufmann Publishers, 2010.

[10] NVIDIA Corporation, "Cuda c++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, March 2022.

[11] N. Ploskas and N. Samaras, *GPU Programming in MATLAB*. San Francisco: Elsevier Science & Technology, 2016.

[12] J. Zhu, "Conversion of earth-centered earth-fixed coordinates to geodetic coordinates," *IEEE transactions on aerospace and electronic systems*, vol. 30, no. 3, pp. 957–961, 1994.

[13] X. Mei and X. Chu, “Dissecting gpu memory hierarchy through microbenchmarking,” *IEEE transactions on parallel and distributed systems*, vol. 28, no. 1, pp. 72–86, 2017.