SPECIAL PERTURBATIONS ON THE JETSON TX1 AND TX2 COMPUTERS

THESIS

Tyler M. Moore, Captain, USAF

AFIT-ENG-MS-18-M-047

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-047

SPECIAL PERTURBATIONS ON THE JETSON TX1 AND TX2 COMPUTERS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Tyler M. Moore, BS

Captain, USAF

March 2018

AFIT-ENG-MS-18-M-047

SPECIAL PERTURBATIONS ON THE JETSON TX1 AND TX2 COMPUTERS

Tyler M. Moore, BS

Captain, USAF

Committee Membership:

Col Dane F. Fuller, PhD
Chair

Dr. William E. Wiesel, PhD
Member

Dr. Douglas D. Hodson, PhD
Member

Abstract

Simplified General Perturbations Number 4 (SGP4) has been the traditional algorithm for performing Orbit Determination (OD) onboard orbiting spacecraft. However, the recent rise of high-performance computers with low Size, Weight, and Power (SWAP) factors has provided the opportunity to use Special Perturbations (SP), a more accurate algorithm to perform onboard OD. This research evaluates the most efficient way to implement SP on NVIDIA's Jetson TX series of integrated Graphical Processing Units (GPUs). An initial serial version was implemented on the Jetson TX1 and TX2's Central Processing Units (CPUs). The runtimes of the initial version are the benchmark that the runtimes of the other versions were compared against. A second version of SP was implemented using compiler optimizations to increase the speed of the program. A third version was developed to utilize the Jetsons' 256-core GPU for parallel processing to reduce the runtimes of the program. Runtimes of the different versions were then analyzed to determine the most efficient way to implement SP on the Jetson TX series of computers.

Acknowledgments

I would like to express my sincere appreciation to my thesis committee, Col Dane Fuller, Dr. William Wiesel, and Dr. Douglas Hodson, for their guidance and support throughout the course of this thesis effort. The insight and experience was certainly appreciated. I would, also, like to thank my fiancé for her support and understanding throughout this process. I would not have been able to do this without you.

Tyler M. Moore

Table of Contents

List of Figures

Page

List of Tables

Page

SPECIAL PERTURBATIONS ON THE JETSON TX1 AND TX2 COMPUTERS

1. Introduction

This chapter introduces this research and discusses the motivation for accomplishing it. The assumptions made and limitations that exist are then presented and discussed. A brief overview of the entire thesis is also contained in this section.

1.1 Background Information

The space domain is becoming increasingly congested as additional countries launch satellites into orbit (Colliot et al., 2012). Therefore, maintaining Space Situational Awareness (SSA) is imperative in ensuring U.S. space assets remain operational. SSA is loosely defined as enabling the description of the location and function of all resident space objects (RSOs) (McCall et al., 2014); thus, determining the position of a space object is an absolute necessity. This is accomplished through a process known as Orbit Determination (OD).

OD is the practice of determining the two primary components of an orbiting object's state vector, position and velocity, at a specific moment in time (Wiesel, 2003). This is accomplished by using an initial guess of the state vector to determine a preliminary orbit. This preliminary orbit does not take into account any external forces, or *perturbations*, such as variations in the potential of Earth's gravity field or atmospheric drag that are measured through ground- or space-based observations. Hence, a set of

equations of motion can be encapsulated in a dynamics model to more accurately represent the object's physical environment (Vetter, 2007). The orbit is then propagated forward in time to estimate the state vector at a future epoch. This can be accomplished *analytically* through a general perturbations method such as Simplified General Perturbations Number 4 (SGP4) or *numerically* through a Special Perturbations (SP) method (Vetter, 2007).

Computers first started being used to perform OD in the mid-twentieth century. At this time, their computational power was insufficient for producing precise orbit propagations. Thus, less computationally-intensive methods were required to perform OD on near-Earth space objects. General perturbation techniques such as SGP4, which assume that there are only small deviations from the two-body problem, were developed to meet this need (Wiesel, 2003).

While ground-based radar and optical sensors continue to be the primary pillars of SSA, they are limited by weather, solar blindspots, and their geography (Baird, 2013). Performing OD onboard the spacecraft can mitigate these limitations. Due to the Size, Weight, and Power (SWAP) constraints of space vehicles (SVs), SGP4 has been a natural fit for onboard OD. Its light-weight design can be implemented on small, energy-efficient computers.

However, SGP4 has its drawbacks; namely, it sacrifices precision for computational efficiency in order to provide a light-weight approach to OD for near-Earth space objects. Due to this trade-off, the accuracy of SGP4 is typically on the order of one kilometer (Vallado et al., 2006). This fact, paired with the increased capability of modern computing, has recently caused the use of SGP4 for SSA tasks to be called into question (Oltrogge et al., 2014).

The SP model, on the other hand, is a more accurate OD method that uses numerical integration to calculate ephemerides for Earth-centered space objects (Air Force Space Command (AFSPC), 2012). This method was first utilized by Cowell and Crommelin in the mid-nineteenth century when they numerically integrated the equations of motion for Halley's Comet to predict its 1910 passing of Earth to within three days (Crommelin, 1911). This method, paired with modern computing resources, can be used to more precisely determine an Earth-centered satellite's position (Pelaez et al., 2007).

Because SP integrates over definite integrals, perturbing forces must be calculated at each step, with the most expensive perturbing force to compute being the geopotential. This makes SP extremely computationally expensive. Historically, computers powerful enough to implement SP have been too large to use onboard a spacecraft. However, as Moore's Law has predicted, computing resources have become increasingly powerful while decreasing their SWAP factors (Moore, 1965). Thus, implementing SP has finally become a viable option for onboard OD.

1.2 Motivation

The Space Object Self-Tracker (SOS) is an experimental payload developed by the Air Force Research Laboratory (AFRL) and the Air Force Institute of Technology (AFIT) as part of the Payload Alert Communications System (PACS) (Bastow, 2013). The objective of PACS is to reduce uncertainty when calculating the positions of space objects. This improves the accuracy of collision avoidance analyses performed by the Joint Space Operations Center's (JSpOC). SOS was designed to be a low SWAP solution to precisely tracking an SV through onboard OD (Perry, 2014).

Figure 1 depicts the SOS concept of operations under normal operating conditions. The payload collects GPS position and velocity data every 10 minutes. Every 24 hours, the Single Board Computer (SBC) performs OD to estimate and propagate the orbit of the SV. Once OD is complete, the SBC sends the orbit parameters and associated telemetry to Air Force space operations units on the ground via the Iridium network (Perry, 2014).

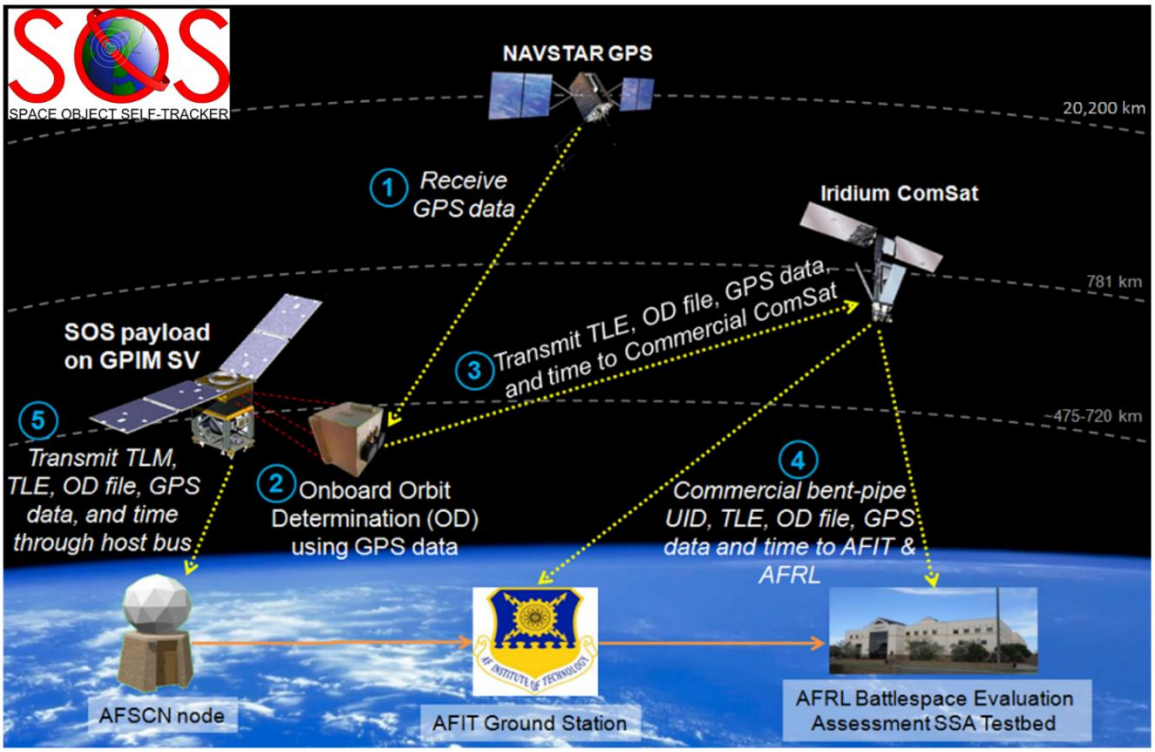


Figure 1. SOS Concept of Operations (CSRA, 2014)

An implementation of SP has been developed for SOS by Wiesel (2015) and tested by Flamos (2016), producing sub-meter level accuracy (Flamos, 2016). However, the low throughput of SOS's single-core SBC, which has a maximum clock speed of 200 MHz (Technologic, 2010), is insufficient in running SP in a timely manner. For this reason, SOS currently uses the SGP4 model. The SBC is powerful enough to run this SGP4 model; however, it produces an error that grows at a rate of 2 kilometers per day (Flamos, 2016).

Because of the limitations of the SBC, more powerful computers with small SWAP parameters were investigated to determine a replacement. The Jetson TX1 and TX2 integrated Graphical Processing Units (GPUs) were chosen as potential candidates. The Jetson TX1 has a quad-core CPU, with each individual processor on the CPU having a clock speed of 1.73 GHz (NVIDIA, 2016). The Jetson TX2 has a six-core CPU, with each core having a clock speed of 2.0 GHz (NVIDIA (A), 2017). Both Jetsons have 256-core GPUs that can be used to compute tasks in parallel. The computing power of these computers presents an opportunity to replace the SGP4 OD algorithm used by SOS with SP, which would reduce the error from the order of a kilometer to below a meter.

1.3 Research Focus

The ultimate goal of the research presented in this thesis was to determine an efficient way, in terms of runtime, to implement SP on the Jetson TX series of computers. Two primary approaches to optimizing SP were taken. The first approach taken was to use compiler flags to optimize the SP code running in serial on the Jetsons' CPUs. The second was to develop a parallel geopotential model that could utilize the Jetsons' GPUs. The runtimes from these two approaches were analyzed and compared to determine which is the most efficient in implementing SP on the Jetson TX1 and TX2.

The success of this research faced three primary challenges. First, the SP software developed for SOS was designed to run on the Windows operating system. The Jetson TX1 and TX2, however, run the Linux operating system and initial attempts to compile the SP application on the Jetson TX1/TX2 proved unsuccessful. Thus, the SP software had to be ported to run on a Linux machine.

Second, it was unknown which compiler flags, if any, would best optimize the serial code. Because there are several hundred compiler flags from which to choose, a guide had to be used to determine which compiler flags were most likely to benefit the application. Once the list of compiler flags was reduced, tests had to be run to determine which combination of compiler flags produced the fastest executable.

Third, the feasibility of using parallel computing to increase the speed of the SP model was uncertain. The SOS codebase uses the Pines Method (Pines, 1973) for computing the geopotential. This is the most time-consuming component of the code; hence, it was the most likely to benefit from parallel computing. However, the Pines Method depends on recursion to calculate several of the primary variables it uses to compute the geopotential (Pines, 1973). This makes a large portion of it inherently serial, meaning that it was not particularly amenable to parallelization.

Furthermore, it was unknown at what point, if any, computing the geopotential in parallel would reduce its runtime when compared to the initial serial version. The SP software uses the Earth Gravity Model 1996 (EGM96) as input for the geopotential routine. This model consists of two lower triangular matrices of harmonic coefficients, C and S, of degree and order 360 (Lemoine, 2005). Depending on the accuracy requirements, the granularity of the model can be scaled up or down by varying the degree and order of the model. Because C and S are lower triangular matrices, the number of elements included in the model is equal to the summation from zero to the degree and order plus one:

$$\sum_{i=0}^{(D\&O)+1} i = Number\ of\ Elements$$

That is, if degree and order of $10\ X\ 10$ is desired, then both matrix C and matrix S will consist of $0 + 1 + 2 + 3 + \cdots + 10 + 11 = 66$ elements. If degree and order of $360\ X\ 360$ is desired, matrix C and matrix S will consist of $0 + 1 + 2 + 3 + \cdots + 360 + 361 = 65{,}341$elements.

Figure 2. Lower Triangular Matrix

The number of elements included in the C and S matrices correlates to the number of threads launched on the GPU. Since the benefit of using the GPU scales as a factor of the number of threads launched, the degree and order of the geopotential directly affects the potential benefit of using it. However, the accuracy requirements of the implementation of SP used for this research only necessitate the use of degree and order of 50 or less.

Therefore, the most number of threads that could be launched when calculating SP was the summation from zero to 51, or 1,326.

The first task that had to be completed for this research was to update the SP software developed on a Windows machine so that it could compile and run on the Jetsons' Linux operating system. Once the solution produced by SP running on the Jetson TX1 and TX2 was verified for correctness, the following two hypotheses could be tested:

1. There is a combination of compiler flags from the chosen list that will result in reduced runtimes compared to the initial serial version of SP running on the Jetson TX1/TX2.
2. Computing portions of the geopotential model in parallel will result in reduced runtimes compared to the initial serial version of SP running on the Jetson TX1/TX2.

The results from these two hypotheses were used to determine the answer to the ultimate question being investigated in this thesis: What is the most efficient way to implement SP on the Jetson TX1 and TX2?

Before the two hypotheses could be tested, the existing SP software was reconfigured to run on the Jetson TX1 and TX2. This initial serial version included no optimizations or parallelization. Success in completing this task was achieved if the application developed for both Jetsons converged to the same solution as the Windows version.

To test the first hypothesis, different compiler optimizations were applied to the SP software. The runtimes of the resulting applications were compared to the initial serial

version of SP running on the Jetsons to determine if any combination of compiler optimizations reduced the runtimes. Success was achieved if the runtimes of this optimized serial version were less than those of the initial serial version for any degree and order less than or equal to 50.

For the second hypothesis, the Assess, Parallelize, Optimize, and Deploy (APOD) software development cycle was applied to the existing SP codebase. A stand-alone, parallel version of the geopotential model was developed and underwent several optimization steps to improve its runtime. It was then integrated into the SP codebase. Success was achieved if the runtime of SP using the parallel geopotential model was less than that of the initial serial version for any degree and order less than or equal to 50.

Once the two hypotheses were tested, the most efficient way to implement SP on the Jetson TX series of computers could be determined. Both the optimized serial version and the parallel version of SP were compared to initial serial version running on the Jetson TX1 and TX2 to determine the most efficient implementation. The performance benefit of running SP on the Jetson TX2 over the TX1 was also analyzed.

1.4 Assumptions and Limitations

Several assumptions and limitations are associated with this research. Namely, the hardware and software constraints, testing assumptions, and the use of the Pines Method for computing the geopotential.

It was assumed that the Jetson TX1 and TX2 must be used. This meant this research was limited by the capabilities of the Jetson TX1 and TX2, such as the amount of active threads each can handle and the amount of on-chip storage each GPU has. Because

NVIDIA's Jetson TX1 and TX2 had to be used, it was assumed any parallel code developed would be completed using NVIDIA's Compute Unified Device Architecture (CUDA) C/C++ language extension. CUDA was specifically developed for parallel computing with NVIDIA GPUs; thus, it was assumed to be the most logical choice for implementing parallel code on the Jetson TX1/TX2.

The testing and development accomplished in this research was completed using a single test case. The initial state vector was given in terms of Earth-Centered Inertial (ECI) coordinates and it used all terms of the geopotential, not just the zonal coefficients. These and other conditions of the test case guided its path through the SP software such that it only used certain functions. However, this test case used all the primary functions that the payload is likely to use under normal operating conditions. Thus, it was assumed to be a sufficient test case. This research was also limited to using runtime efficiency as the primary measure of performance. Power consumption and other factors were considered to be outside of the scope and were not considered when measuring performance.

Different types of geopotential models such as Mass Concentration (Mascon) and 3D interpolation have been developed in order to sidestep the problems presented by the recursion found in the Pines Method (Russell, 2012; Arora, 2016). However, due to the time constraints of the academic program, the scope of the research presented in this thesis was limited to parallelizing the existing SP codebase. Therefore, it was assumed Pines Method must be used for computing the geopotential.

1.5 Thesis Overview

This thesis consists of five chapters. Chapter I provides an introduction to the topic of the research and the background information relevant to it. Chapter II provides an in-depth look at the relevant subject matter required to complete the research presented in this thesis. Subjects include: the existing codebase of SOS, geopotential modeling, compiler optimizations, CUDA, the Jetson TX1 and TX2, and the APOD design cycle. Chapter III presents the methodology used to complete the research accomplished in this thesis. The two hypotheses discussed in Section 1.3 guide this section. In Chapter IV, the results from all experiments conducted to test the hypotheses are analyzed and discussed. Finally, Chapter V summarizes the results of this research and makes recommendations for future work.

1.6 List of Terms

Table 1. List of Terms

AFSPC	Air Force Space Command
APOD	Assess, Parallelize, Optimize, and Deploy
ARM	Acorn RISC Machine
CERN	Conseil Européen pour la Recherche Nucléaire
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
D&O	Degree and Order (of the Geopotential)
DRAM	Dynamic Random-Access Memory
ECI	Earth-Centered Inertial

EGM96	Earth Gravity Model 1996
GNU	GNU's Not Unix!
GPGPU	General-Purpose GPU
GPU	Graphical Processing Unit
GSFC	Goddard Space Flight Center
HBM2	High-Bandwidth Memory 2
HPC	High-Performance Computing
LEO	Low-Earth Orbit
MJD	Mean Julian Day
NGA	National Geospatial-Intelligence Agency
NIMA	National Imagery and Mapping Agency
NRL	Naval Research Laboratory
NVCC	NVidia C Compiler
OD	Orbit Determination
PCI-e	Peripheral Component Interconnect-express
RISC	Reduced Instruction Set Computing
RMS	Root-Mean-Square
RSO	Resident Space Object
SGP4	Simplified General Perturbations 4
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SOS	Space Object Space-Tracker

SP	Special Perturbations
STK	Systems Tool Kit
SV	Space Vehicle
SWAP	Size, Weight, And Power

2. Literature Review

This chapter presents a literature review of relevant background information needed to perform the research including the SOS software, gravity modeling, and GNU's Not Unix! (GNU) compiler flags. NVIDIA's parallel programming language extension, CUDA, the Jetson TX1 and TX2 integrated GPUs, and the APOD design cycle are also discussed in detail.

2.1 Space Object Self-Tracker Software

The SOS program currently utilizes SGP4; however, an SP implementation has been developed by Wiesel et al. (Flamos, 2016). It was written in C++ and compiled to run on a Windows machine. This algorithm numerically integrates the equations of motion and the equations of variation of a space object and propagates them to predict its state at a future epoch. This research investigates the feasibility of replacing SGP4 with SP; hence, only the SP algorithm is discussed in this section.

The SP software used is comprised of the files in Table 2. The *main* function is located in the SPLstSq.cpp file. SPLstSq.cpp also accomplishes Blocks 1-3 in Figure 3. The *main* function calls the *hamming* routine in Hamming.cpp to begin the least squares iteration. Throughout this process, the Dynamics model contained in EarthTruth.h/EarthTruth.cpp is applied to the state vector, which uses routines in Atmosphere.cpp and Geopotential.h/Geopotential.cpp to account for perturbing forces. Once these perturbing forces are applied to the state vector, it is propagated forward in time using the *interp* function in the Interp.cpp file. This process is repeated until the least squares method converges.

Table 2. List of Files that comprise the SP Software

calendar	C++ Header file
EarthTruth	C++ Header file
Ephemeris	C++ Header file
Geopotential	C++ Header file
Interp	C++ Header file
JulianDay	C++ Header file
LinearEquations	C++ Header file
ludcmp	C++ Header file
numerical	C++ Header file
ObservingSite	C++ Header file
SingularValue	C++ Header file
svd	C++ Header file
TwoBodyProblem	C++ Header file
Atmosphere	C++ Source file
EarthTruth	C++ Source file
Geopotential	C++ Source file
Hamming	C++ Source file
Interp	C++ Source file
Observation	C++ Source file
ReadInertialRData	C++ Source file
ReadMyTruth	C++ Source file
ReadSTK	C++ Source file
SPLstSq	C++ Source file

The SP program first reads in observational data and ensures it is in the correct format (Figure 3, Blocks 1-3). It then uses the least squares method to propagate the initial state, applying a dynamics model to account for perturbing forces (Figure 3, Blocks 4-6). The algorithm calculates the position residuals for each observation and determines the magnitude of error (Figure 3, Block 7-8). It then uses the error calculations to correct the reference trajectory (Figure 3, Block 9-10) and iterates through this process until the reference trajectory has converged.

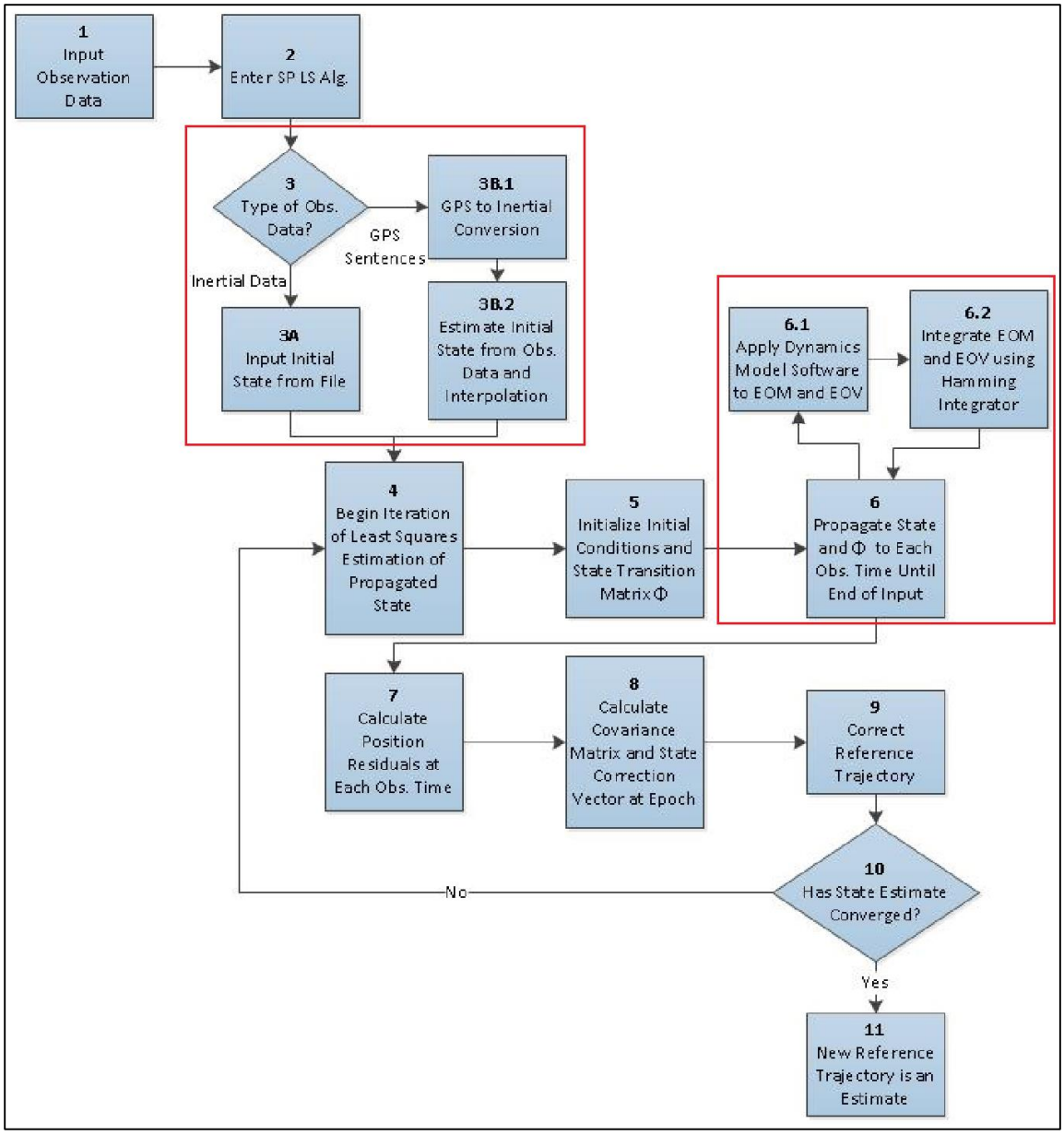


Figure 3. Special Perturbations Algorithm (Flamos, 2016)

Figure 4 below is a package diagram illustrating the relationships between these files, namely which files include routines from other files.

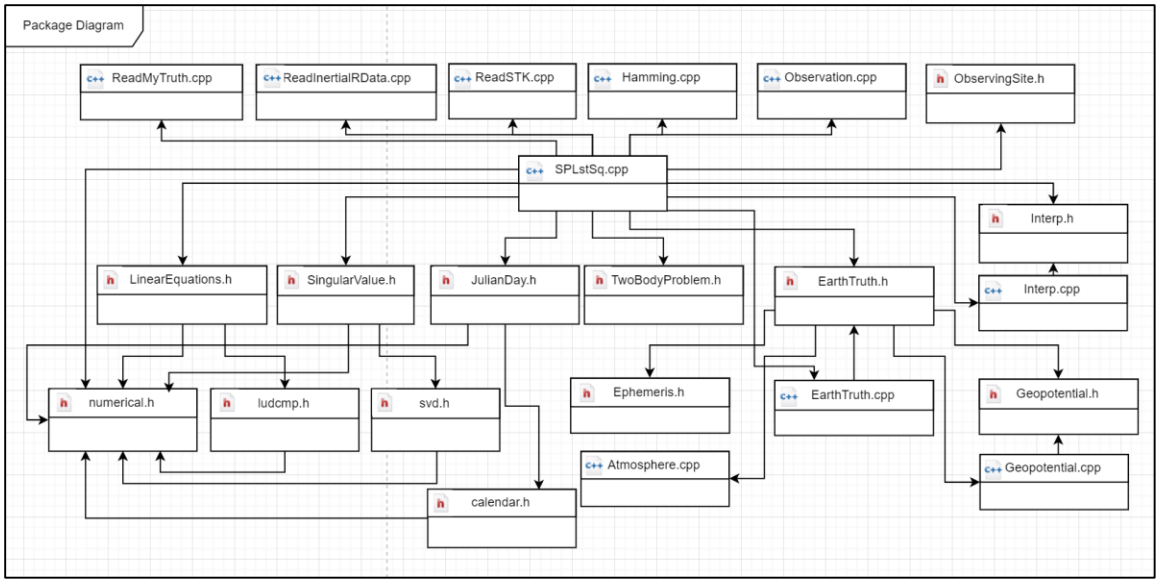


Figure 4. Original Package Diagram of the SP Software

The dynamics model contained in the EarthTruth files calls the geopotential routine thousands of times per iteration of SP. Because calculating the geopotential is very computationally expensive, this takes up a significant amount of SP's total runtime. The geopotential is explained in further detail in Section 2.2.

2.2 Earth's Gravitational Field

Traditionally, orbital mechanics has focused on the two-body problem, concerning two masses interacting through Newtonian point mass gravity (Wiesel, 2003). This is because naturally occurring celestial objects with relatively large masses such as comets and planets are typically separated by enough distance that the gravitational forces of other N-order objects are negligible. However, since the advent of manmade spacecraft, other perturbing forces must be taken into account when considering near-Earth space objects in Low-Earth Orbit (LEO). Because of Earth's rotation and variations in its surface density, the potential energy created by Earth's gravity field, or the *geopotential*, varies, especially

for RSOs in LEO. Objects at these altitudes also encounter atmospheric drag and space environment effects such as solar radiation that can have an impact on their orbits over time (Wiesel, 2003), but since this research is primarily concerned with the geopotential, only it is discussed in further detail.

2.2.1 Accounting for Variations in the Geopotential

The magnitude of variation in the geopotential can be large at lower altitudes (Wiesel, 2003). Earth's rotation causes it to bulge about its equator, making it an oblate rather than a homogeneous spheroid. This added mass about its equator increases the geopotential in this zone. This is shown in Figure 5, where the horizontal radius is larger than the vertical. Further deviations from a perfectly spherical gravity field are attributed to variations in the Earth's density; for example, the geopotential is generally stronger over a mountain range and weaker over an ocean basin. Because of these irregularities, the geopotential must be modelled as distribution of points in LEO instead of as a singular point mass at Earth's center as with higher altitude orbits (Wiesel, 2003).

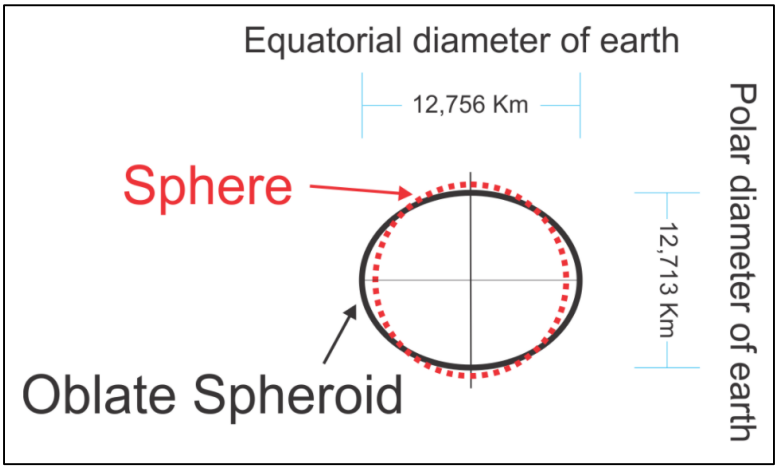


Figure 5. Earth's Oblateness (SandBox Science, 2017)

Earth's gravitational potential at a specific point can be expressed as a distribution of mass. Using polar coordinates, the geopotential, V, is expressed as $V(r, \alpha, \lambda)$, where r is the object's distance from Earth's center, α is its geocentric longitude, and λ is its geocentric colatitude. The geopotential can be derived by expanding the following infinite series known as the *geopotential expansion*:

$$V(r, \alpha, \lambda) = \frac{\mu}{r} \cdot \left\{ 1 + \sum_{n=1}^{\infty} \left(\frac{a}{r}\right)^n \sum_{m=1}^{n} P_n^m(\sin \alpha) \left(C_n^m \cdot \cos m\lambda + S_n^m \cdot \sin m\lambda\right) \right\}$$

Equation 1. Geopotential Expansion (Wiesel, 2003)

where μ is the gravitational constant; a is the Earth's equatorial radius; C and S are the spherical harmonic coefficients; and n and m are the degree and order, respectively. P represents the Associated Legendre Function (ALF), which is the zonal harmonic solution to the Legendre differential equation (Wiesel, 2003).

The spherical harmonic coefficients, or geopotential coefficients, of C and S are obtained through measurements and observations to account for Earth's oblateness and density variations. They represent the actual shape of the gravity field; therefore, they are the primary elements of a geopotential model (Wan Aziz et al., 1998).

2.2.2 Earth Gravitational Model 1996

In the 1990s, the National Imagery and Mapping Agency (NIMA), now known as the National Geospatial-Intelligence Agency (NGA), led a joint effort along with the NASA Goddard Space Flight Center (GSFC) and The Ohio State University to develop a high-fidelity geopotential model. This collaboration resulted in the Earth Gravitational

Model 1996 (EGM96), an improved geopotential model with degree and order of 360 (Lemoine, 2005).

This project required the collection of an immense amount of surface gravity data in order to accurately account for Earth's oblateness and density variations. The Naval Research Lab (NRL) conducted airborne gravity surveys over Greenland and parts of the Artic and Antarctica, while NIMA partnered with gravity collection projects from nations around the globe to cover land areas (Lemoine, 2005). Their efforts resulted in more than 30 million gravity points being recorded. These values were used to interpolate Earth's gravity field by computing point gravity anomalies using the geopotential expansion (Lemoine et al., 1998).

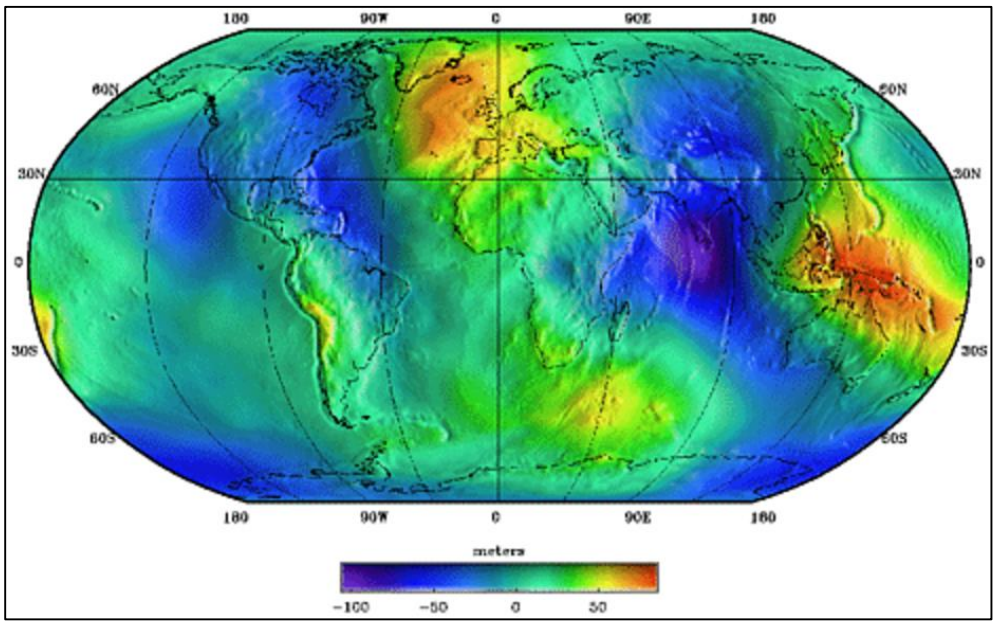


Figure 6. EGM96 Geoid (Lemoine, 2005)

These results were augmented by an extensive series of satellite tests. The NASA GSFC partnered with the U.S. Navy and European space agencies to launch the GEOSAT, TOPEX/POSIEDON, and ERS-1 missions. The direct altimetry collected by these satellites, paired with data collected from tracking the orbits of more than 20 other satellites, was used to verify and validate the surface collections. These efforts produced the high-fidelity EGM96 geopotential data that attained an accuracy on the magnitude of several milligals (Lemoine, 2005).

2.2.3 Pines Method

Because the traditional method of calculating Earth's geopotential uses spherical coordinates, it does not account for singularity about the Earth's poles. Thus, Pines introduced the uniform representation of the geopotential in which the geopotential expansion (Equation 1) was modified to overcome this singularity (Pines, 1973). The formulation of Pines Method is presented in this section.

The spherical coordinates, r, α, λ, are represented in directional-cosine, Cartesian coordinates as the position vector of $\boldsymbol{R} = \begin{Bmatrix} x \\ y \\ z \end{Bmatrix}$ in which:

$$x = \cos\alpha \cdot \cos\lambda$$

$$y = \cos\alpha \cdot \sin\lambda$$

$$z = \sin\alpha$$

And the scalar vector: $$r = \sqrt{x^2 + y^2 + z^2}$$

Pines then proposed a three-component unit vector $\widehat{\boldsymbol{R}} = \begin{Bmatrix} s \\ t \\ u \end{Bmatrix}$, where:

$$s = \frac{x}{r}$$

$$t = \frac{y}{r}$$

$$u = \frac{z}{r}$$

And, $$s^2 + t^2 + u^2 = 1$$

Furthermore, the ALFs, $P_n^m(u)$, were modified to become the derived Legendre polynomials (DLFs):

$$A_n^m(u) = \frac{1}{2^n n!} \cdot \frac{d^{n+m}}{du^{n+m}} \cdot (u^2 - 1)^n$$

The complex variable recursion relationships are defined as follows:

$$c_{real_m} = s \cdot c_{real_{m-1}} - t \cdot c_{imag_{m-1}}$$

$$c_{imag_m} = s \cdot c_{imag_{m-1}} + t \cdot c_{real_{m-1}}$$

Further recursion relationships are formed:

$$\rho = \frac{a}{r}$$

$$\rho_0 = \frac{\mu}{r}$$

$$\rho_1 = \rho \cdot \rho_0$$

$$\rho_n = \rho \cdot \rho_{n-1}$$

Finally, the coefficient mass functions are defined as follows:

$$D_n^m = C_n^m \cdot c_{real_m} + S_n^m \cdot c_{imag_m}$$

$$E_n^m = C_n^m \cdot c_{real_{m-1}} + S_n^m \cdot c_{imag_{m-1}}$$

$$F_n^m = S_n^m \cdot c_{real_{m-1}} + C_n^m \cdot c_{imag_{m-1}}$$

$$G_n^m = C_n^m \cdot c_{real_{m-2}} + S_n^m \cdot c_{imag_{m-2}}$$

$$H_n^m = S_n^m \cdot c_{real_{m-2}} + C_n^m \cdot c_{imag_{m-2}}$$

Thus, transforming the geopotential expansion to the following:

$$V(r,s,t,u) = \sum_{n=0}^{\infty} \rho_n \sum_{m=0}^{n} A_n^m \cdot D_n^m(s,t)$$

The first partial coefficients of acceleration are then derived:

$$a_1 = \frac{1}{r} \cdot \frac{\delta V}{\delta s} = \sum_{n=0}^{\infty} \frac{\rho_{n+1}}{a} \sum_{m=0}^{n} A_n^m(u) \cdot m \cdot E_n^m$$

$$a_2 = \frac{1}{r} \cdot \frac{\delta V}{\delta t} = \sum_{n=0}^{\infty} \frac{\rho_{n+1}}{a} \sum_{m=0}^{n} A_n^m(u) \cdot m \cdot F_n^m$$

$$a_3 = \frac{1}{r} \cdot \frac{\delta V}{\delta u} = \sum_{n=0}^{\infty} \frac{\rho_{n+1}}{a} \sum_{m=0}^{n} A_n^{m+1}(u) \cdot D_n^m$$

And the coefficient of $\widehat{\boldsymbol{R}}$:

$$a_4 = \frac{\delta V}{\delta r} - \left(\frac{s}{r}\frac{\delta V}{\delta s}\right) - \left(\frac{t}{r}\frac{\delta V}{\delta t}\right) - \left(\frac{u}{r}\frac{\delta V}{\delta u}\right) = -\sum_{n=0}^{\infty} \frac{\rho_{n+1}}{a} \sum_{m=0}^{n} A_{n+1}^{m+1}(u) \cdot D_n^m$$

These first partial derivatives are used to find the acceleration force vector $\boldsymbol{F}$ as follows:

$$\boldsymbol{F} = a_1\hat{\boldsymbol{\imath}} + a_2\hat{\boldsymbol{\jmath}} + a_3\widehat{\boldsymbol{k}} + a_4\widehat{\boldsymbol{R}}$$

Where $\hat{\boldsymbol{\imath}} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix}$; $\hat{\boldsymbol{\jmath}} = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix}$; and $\widehat{\boldsymbol{k}} = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$.

The second partial acceleration coefficients are derived similarly to form:

$$a_{11} = \sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} m(m-1) A_n^m G_n^m$$

$$a_{12} = \sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} m(m-1) A_n^m H_n^m$$

$$a_{13} = \sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} m \cdot A_n^{m+1} E_n^m$$

$$a_{14} = -\sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} m \cdot A_{n+1}^{m+1} E_n^m$$

$$a_{23} = \sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} m \cdot A_n^{m+1} F_n^m$$

$$a_{24} = -\sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} m \cdot A_{n+2}^{m+1} D_n^m$$

$$a_{33} = \sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} m \cdot A_{n+1}^{m+1} F_n^m$$

$$a_{34} = -\sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} A_{n+1}^{m+2} D_n^m$$

$$a_{44} = \sum_{n=0}^{\infty} \frac{\rho_{n+2}}{a^2} \sum_{m=0}^{n} A_{n+2}^{m+2} D_n^m$$

The first and second partial coefficients are then combined to compute the gradient of $\boldsymbol{F}$, P:

$$P_{11} = a_{11} + s^2 a_{44} + \frac{a_4}{r} + 2s a_{14}$$

$$P_{12} = P_{21} = a_{12} + st a_{44} + s a_{24} + t a_{14}$$

$$P_{13} = P_{31} = a_{13} + su a_{44} + s a_{34} + u a_{14}$$

$$P_{22} = -a_{11} + t^2 a_{44} + \frac{a_4}{r} + 2t a_{24}$$

$$P_{23} = P_{32} = a_{23} + tua_{44} + ua_{24} + ta_{34}$$

$$P_{33} = a_{33} + u^2 a_{44} + \frac{a_4}{r} + 2ua_{34}$$

(Pines, 1973)

This method removes the singularity about the Earth's poles. It uses recursive relationships to compute the acceleration and gradient of the geopotential, providing a solution that is relatively easy to code. However, it does not provide a particularly computationally efficient implementation, as it creates a considerable increase in the number of function evaluations required (Casotto et al., 2007).

2.3 Compiler Optimizations

GNU compilers have built-in functionality to optimize the execution of binaries in terms of speed. These compiler flags number in the hundreds, with each having the potential to decrease the runtime of a program. Therefore, the European Organization for Nuclear Research, or *Conseil Européen pour la Recherche Nucléaire* (CERN) in French, partnered with industry to form CERN openlab to investigate which compiler flags are most likely to improve the runtime of C++ code running on a CPU (Botezatu, 2012). This study produced a list of 17 compiler flags that are most likely to improve the performance of various programs by at least 1% in comparison to code compiled with the "-O2" optimizations enabled. The resulting list, along with high-level descriptions, is presented in Table 3:

Table 3. Compiler flags with description (Botezatu, 2012)

- O3	-O2 optimizations plus more aggressive optimizations for maximum speed like: • Loop unrolling and instruction scheduling • Code replication to eliminate branches • Padding the size of power two arrays to allow more efficient cache use
-ipo	Enables interprocedural optimizations between files. When this flag is enabled, the compiler performs inline function expansion for calls to functions defined in separate files.
-opt-ra-region-strategy=routine	The register allocator creates a single region for each routine.
-ip	Enables additional interprocedural optimizations for single-file compilations.
-opt-ra-region-strategy=block	The register allocator partitions each routine into one region per basic block.
-funroll-all-loops	Unroll all loops even if the number of iterations is uncertain when the loop is entered.
-nolib-inline	Disables inline expansion of standard library or intrinsic functions.
-inline-forceinline	Specifies that an inline routine should be inlined whenever the compiler can do so.

-opt-class-analysis	Determines whether C++ class hierarchy information is used to analyze and resolve C++ virtual function calls at compile time.
-opt-streaming-store-always	Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound.
-ansi-alias	Assumes that the program adheres to ISO C Standard aliasing rules. This allows the compiler to optimize more aggressively. If the code does not adhere to these rules then it can cause the compiler to generate incorrect code.
-opt-prefetch=4	Enables prefetch insertion optimization, with optprefetch=4 being more aggressive.
-falign-functions	A align functions on an optimal byte boundary.
-unroll-aggressive	This option enables aggressive, complete unrolling for loops with small constant trip counts.
-fno-inline-functions	It is the opposite of finline-functions which is enabled in O2 and O3.
-opt-block-factor=16	Loop-blocking factor=16. Loop blocking optimization is part of the High Level Optimizations in Intel compiler.
-opt-block-factor=2	Loop blocking factor = 2.

The “-O” series of compiler flags contains several different optimizations, making it particularly useful. It consists of three primary compiler flags: “-O1”, “-O2”, and “-O3”, with each level including all optimizations of its predecessor. That is, the “-O2” flag contains all optimizations of the “-O1” flag, and the “-O3” flag contains all optimizations of the “-O1” and “-O2” flags (Free Software Foundation, 2017).

Passing any version of the “-O” flag to compiler enables several optimizations that maximize the speed of the executable. The compiler will determine if a function can be *inlined*. When this occurs, the compiler replaces a function call in the code with a copy of the function itself. Loops are also optimized. The instruction and memory accessing overhead of iterating through loops can be decreased be adding code to the body of the loop through a process known as *loop peeling*, which enables the loop to iterate over larger increments. Consider the following example. The original **`for`** loop iterates 16 times, incrementing by one each time. The optimized version, however, iterates only four times because it increments by four each time.

Original:

```
for(int i = 0; i < 16; i++)
{
X[i] = i*i;
}
```

Optimized:

```
for(int i = 0; i < 16; i+4)
{
X[i]   = i*i;
X[i+1] = (i+1)*(i+1);
X[i+2] = (i+2)*(i+2);
X[i+3] = (i+3)*(i+3);
}
```

Loops are also optimized by *predictive commoning*. The predictive commoning optimization enabled by the "-O3" compiler flag tells the processor to reuse computations calculated by the N^{th} iteration in a loop for the $(N+1)^{st}$ iteration (Free Software Foundation, 2017). All 14 optimizations included in the "-O3" compiler flag, along with high-level descriptions, are presented in Table 4:

Table 4. "-O3" Compiler Optimizations with Descriptions (Free Software Foundation, 2017)

-finline-functions	Considers all functions for inlining.
-funswitch-loops	Moves branches with loop invariant conditions out of the loop.
-fpredictive-commoning	Reuses computations (especially memory loads and stores) performed in previous iterations of loops.
-fgcse-after-reload	Performs redundant load elimination pass after reload.
-ftree-loop-vectorize	Performs loop vectorization on trees.
-ftree-loop-distribution	Improves cache performance on big loop bodies and allows for further loop optimizations.
-ftree-loop-distribute-patterns	Performs loop distribution of patterns that can be code generated with calls to a library.

-floop-interchange	Improves cache performance on loop nest and allows for further loop optimizations.
-fsplit-paths	Improves dead code elimination and common subexpression elimination.
-ftree-slp-vectorize	Performs basic block vectorization on trees.
-fvect-cost-model	Alters the cost model used for vectorization.
-ftree-partial-pre	Makes partial redundancy elimination more aggressive.
-fpeel-loops	Peels loops for which there is enough information that they do not roll much.
-fipa-cp-clone	Performs function cloning to make interprocedural constant propagation stronger.

2.4 Parallel Computing

Most computer programs are designed to execute code in a serial manner. Single-core CPUs usually employ this type of architecture. This changed, however, when the theoretical limits of the CPU began to be reached (Frank, 2002). This caused focus to shift away from improving CPU performance towards using many-core processors to execute code in parallel (Asanovic et al., 2006).

According to Williams (2012), there are two primary ways to employ parallel computing. The first is *task parallelism*. When using this method, a single task is divided into discrete, independent tasks that can be computed simultaneously. Task parallelism is useful when a single data set must be operated on by N different instructions, resulting in N different data sets. The second method for computing in parallel is *data parallelism*. This method is used when multiple pieces of data must be operated upon by a single instruction. Most vector and array operations fall under this category. Both methods utilize a machine's multiple processors to decrease the overall runtime of an application (Williams, 2012).

In the early decades after its inception, the only way to perform parallel processing was to manually code applications for concurrent execution using vendor-supplied, nonstandard libraries or language extensions. This meant that developers would have to invest significant time, effort, and costs into reengineering complex software applications to utilize parallel architectures without knowing whether desired efficiencies and reduced runtimes would be achieved. To overcome this barrier, programming languages had to be adapted and extended to support multithreaded functionality (Hack, 1989).

2.4.1 CUDA

In 2006, NVIDIA developed a computer architecture for data parallelism called Single Instruction, Multiple Threads (SIMT) which combined multithreading with an array of multiprocessors (NVIDIA (B), 2017). The first system of this kind was also invented by NVIDIA and used the G80 GPU. While the GPU was initially intended to render three-dimensional images on a display for the gaming industry, its scalable array of processors was a natural fit for problems that could be solved using data parallelism. Researchers

began employing GPUs for more general high-performance computing (HPC) tasks, coining the term General-Purpose GPU (GPGPU) (NVIDIA (B), 2017).

Soon after, NVIDIA released their Compute Unified Device Architecture (CUDA) based on the SIMT paradigm. CUDA, coupled with the NVIDIA CUDA compiler (nvcc), was the first C language extension that gained widespread traction among users wishing to take advantage of GPUs for general purpose computing. It allowed engineers to disregard the underlying graphical concept for which the GPU was originally intended and instead utilize it as a true GPGPU (NVIDIA (B), 2017).

In CUDA, the CPU and GPU are known as the Host and Device, respectively. Maintaining the logical distinction between Host and Device as two separate entities enables CUDA to employ a *heterogeneous programming model* where threads are executed on a physically separate device. This model assumes that the Host and the Device maintain their own separate memory and that the Host directs the Device on which functions and data to operate. In other words, the Host begins the program, configures the number of threads to be executed, and then calls parallelized Device functions for the Device to process (NVIDIA (B), 2017).

CUDA extends C/C++ to allow users to define functions, or *kernels*, that are executed on the Device, meaning they can utilize the GPU's array of processors. When called, kernels can launch thousands of threads simultaneously, instead of being executed as a single thread as in a serial implementation. Kernels can use two different declaration specifiers, **`__global__`** or **`__device__`**. Kernels using the global declaration specifier are called from the Host and executed on the Device, while kernels using the device declaration specifier are called and executed on the Device (NVIDIA (B), 2017).

To manage its heterogeneous programming model, CUDA adopts three fundamental abstractions: a hierarchy of thread groups, a hierarchy of GPU memory, and barrier synchronization. CUDA's thread hierarchy allows users to divide complex problems into finer-grained sub-problems that can be managed, branched, and executed differently depending on their place in the hierarchy. The memory hierarchy allows the user to manage what can and cannot be accessed by code running on the GPU. CUDA also employs barriers that ensure no single thread goes beyond a certain specified point, wherein doing so would result in an attempt to access or manipulate data that is dependent on other threads. These three core capabilities aide in partitioning tasks into smaller sub-problems that can be solved cooperatively by multiple threads (NVIDIA (B), 2017).

2.4.1.1 CUDA's Thread Hierarchy

CUDA organizes threads into a hierarchy of threads, thread blocks, and grids. Threads are the lowest level in the hierarchy. The CUDA built-in keyword, **`threadIdx`**, is used for indexing each thread launched on the Device. It returns a three-component vector enabling individual threads to be identified in up to three dimensions. The next tier is referred to as a thread block, and is a collection of multiple threads that are executed independently. Thus, thread blocks are required to be structured such that all threads within a given block can be executed in any order or in parallel. This requirement allows CUDA programs to scale to the number of Streaming Multiprocessors (SMs) on a given Device. Thread blocks are indexed using **`blockIdx`** which is also a three-component vector. The multi-dimensional thread blocks are organized into *grids*, which are the highest level in the thread hierarchy (NVIDIA (B), 2017).

The total number of threads being operated upon by a kernel depends on the number of grids, blocks, and threads launched. For instance, if a kernel is launched with number of grids, g, number of blocks per grid, b, and number of threads per block, t, the total number of threads, T, is given by the following expression:

$$T = g \times b \times t$$

Grids, thread blocks, and threads are illustrated in Figure 7, in which a single grid has six blocks, with 12 threads per block, resulting in 72 total threads:

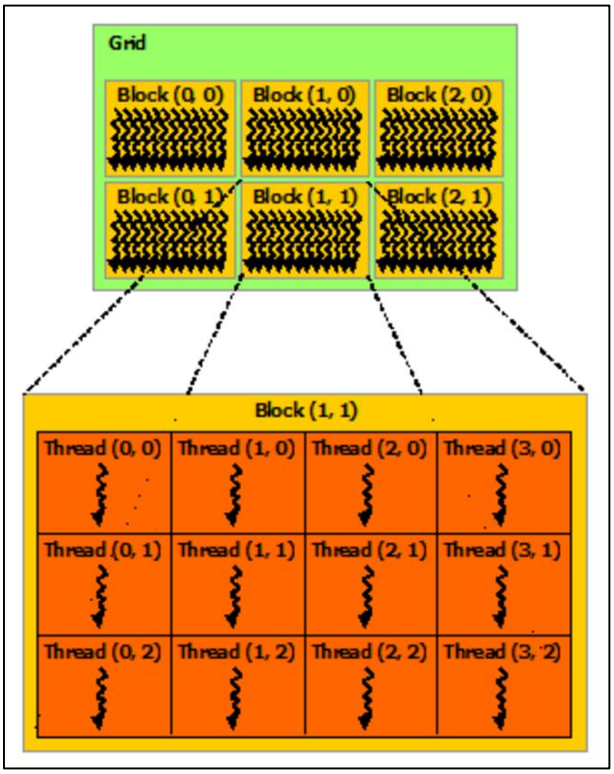


Figure 7. CUDA's Thread Hierarchy (NVIDIA (B), 2017)

It is important to note that upon launching a kernel, the Device creates, schedules, and executes threads in groups of 32 called *warps*. When an SM is given a thread block to execute, it divides it into warps and uses its warp scheduler to schedule each one. SMs manage threads in groups of 32 regardless of the number of threads per block; therefore, it

is important to adjust block size into multiples of 32 whenever possible (NVIDIA (B), 2017). Configuring the most efficient number of grids, blocks, and threads is discussed in further detail in Section 2.4.4.3.

2.4.1.2 CUDA's Memory Hierarchy

Device memory is divided up into three primary tiers, with the lowest level being *local memory*. Each individual thread has its own local memory. Despite what its name implies, local memory's default physical location resides off-chip, making it inefficient to access. However, local variables can be moved to registers located on-chip. The number of these 32-bit registers is of course finite, so care must be taken to not exceed the amount of registers available. The next level of Device memory is *shared memory.* Each thread block has its own shared memory space that each thread within a block can access. This allows thread blocks to work together to perform interdependent tasks such as summations by storing and accessing data using shared variables across all threads in a thread block. Because multiple threads use shared memory to collaborate, it is low latency. Therefore, shared memory resides on-chip, and should be used whenever possible. The highest level in CUDA's memory hierarchy is *global memory*. Each thread, thread block, and grid can access the Device's global memory. Global memory is the largest memory space, but it is inefficient to access due to it being off-chip. In addition to the three primary types of Device memory, there are also two read-only memory spaces called *texture* and *constant* memory that can be used in the same way as global memory (NVIDIA (B), 2017). CUDA's primary memory hierarchy is shown in Figure 8:

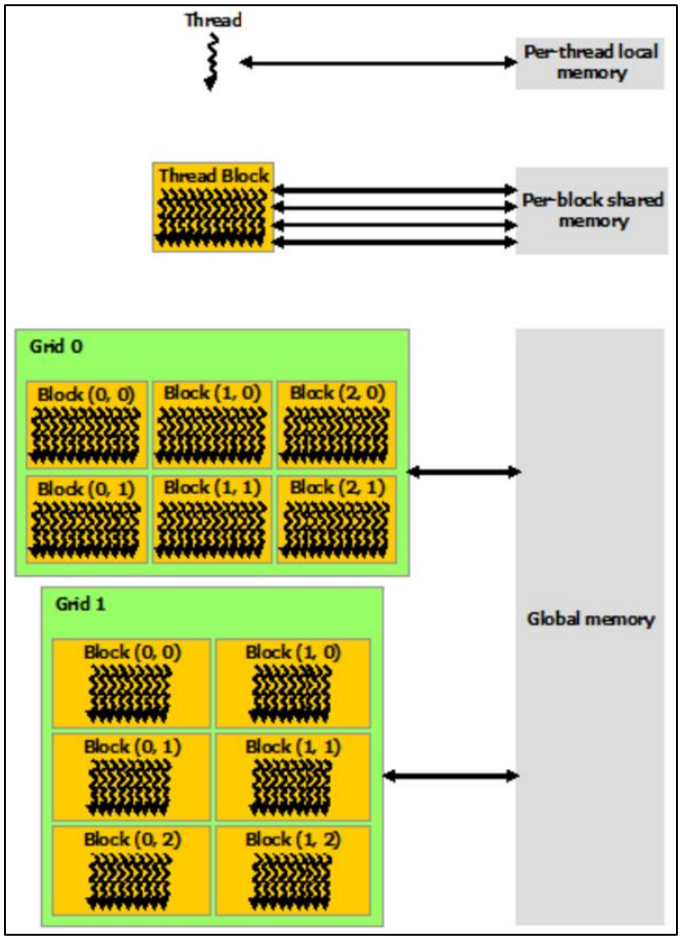


Figure 8. CUDA's Memory Hierarchy (NVIDIA (B), 2017)

2.4.1.3 Thread Synchronization

The sharing of data by shared and global memory introduces the same synchronization problems that arise in multi-threaded applications. Because threads working cooperatively to solve a problem are often dependent on data produced by another thread, certain threads can attempt to access this data before it has actually been computed. Thus, CUDA employs a built-in function, **`__syncthreads()`**. It acts as a barrier that no thread can go beyond until all threads within a block or grid have reached it. This barrier synchronization is necessary for the collaborative capability of CUDA (NVIDIA (B), 2017).

2.4.2 NVIDIA Jetson TX1

Part of the research presented in this thesis investigates runtime performance gains that can be achieved by porting existing SP and Geopotential code bases to NVIDIA's Jetson TX1 system-on-module. It consists of a tightly-coupled CPU and GPU on the same board, both of which are discussed in this section.

2.4.2.1 Jetson TX1 CPU

The Jetson TX1 module employs a quad Acorn Reduced Instruction Set Computing (RISC) Machine (ARM) Cortex-A57 CPU that can achieve an operating frequency of 1.73 GHz. It has 80 KB of L1 cache per core, resulting in 320 KB of total L1 cache. It also has 2 MB of shared L2 cache between its four cores. The Cortex-A57 processor core utilizes a SIMT architecture, ARMv8-A, that enables the Jetson TX1's four CPU cores to perform multithreaded operations (Otterness et al., 2017). Furthermore, the Jetson TX1's L2 cache is optimized for multithreaded applications by allowing multiple processors to access the L2 simultaneously (NVIDIA, 2016).

2.4.2.2 Jetson TX1 GPU

The Jetson TX1's GPU consists of two Maxwell SMs, each of which contains an array of 128 processors, or CUDA cores. The Maxwell architecture improves NVIDIA's control logic partitioning, workload balancing, clock-gating granularity, compiler-based scheduling, and number of instructions issued per clock cycle (NVIDIA, 2016). It has 4 GB of Dynamic Random-Access Memory (DRAM) that can be accessed at 25.6 GB/s. The Maxwell architecture also devotes a full 64 KB of L1 cache per SM to shared memory, decreasing the time cost in algorithms that depend on sharing variables across thread

blocks. Both Maxwell SMs on the Jetson TX1 have a clock speed of 998 MHz. Moreover, the Maxwell architecture features 3,072 KB of L2 cache which is larger than previous designs and results in fewer high-cost accesses of global memory (NVIDIA, 2016).

GPUs are typically divided into two categories: *discrete* or *integrated*. A discrete GPU is a stand-alone device that must be manually connected to a CPU in order for it to be utilized for parallel computing. The Jetson TX1 falls into the integrated category, where the CPU and GPU are built onto a single board and share DRAM. The Jetson TX1's 4GB of shared DRAM give it a wider range of mechanisms to transfer data to and from the Device (Otterness et al., 2017).

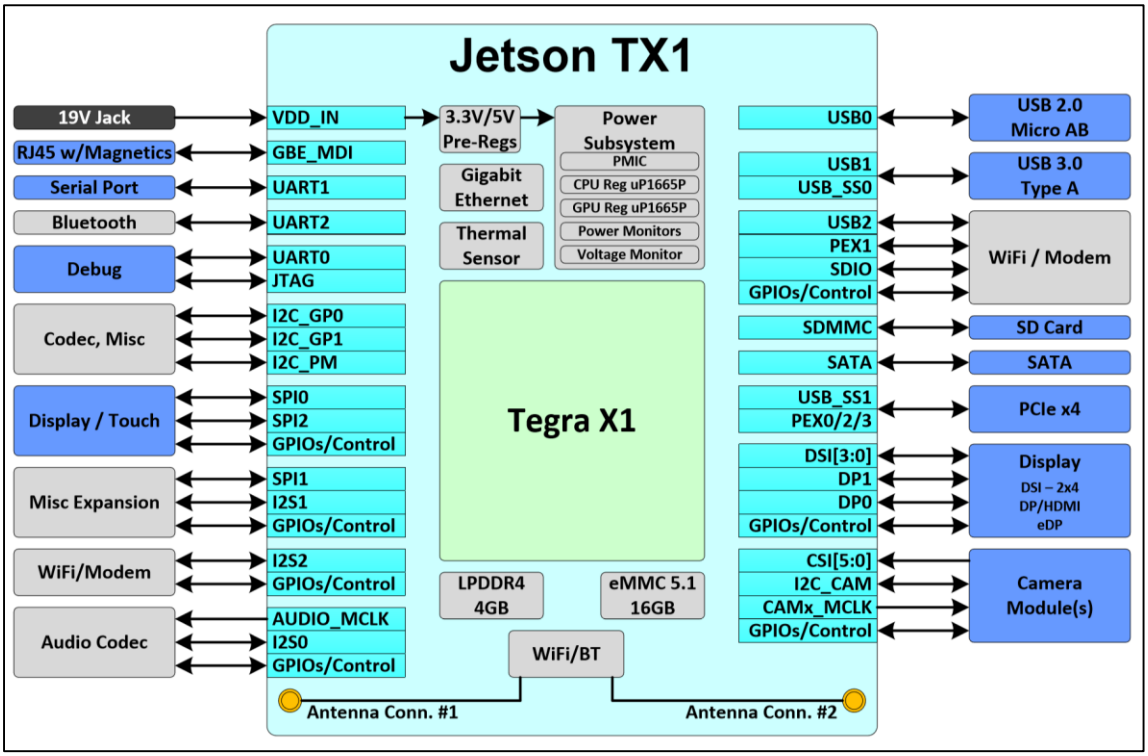


Figure 9. Block Diagram of the Jetson TX1 Development Kit (NVIDIA, 2016)

The Jetson TX1 has a compute capability of 5.3. This is not to be confused with the version of CUDA that is deployed on the Jetson TX1 (i.e., CUDA 5.5, CUDA 6.0, etc.), as

the compute capability of a device represents its SM version, not its CUDA version. Thus, the compute capability specifies the capabilities supported by the GPU hardware implementation (NVIDIA (B), 2017).

A compute capability of 5.3 tells the compiler that the Device has 128 CUDA cores per SM and four warp schedulers at its disposal. When a kernel is launched, the SM distributes all warps between the four schedulers. Every time an instruction is issued, each individual scheduler issues the instruction to the next warp in the queue (NVIDIA (B), 2017).

A device's compute capability also dictates how kernels can be configured and executed. Devices of compute capability 5.3 can have a maximum of 16 grids present on the Device. It allows for up to 1024 threads per thread block, and up to 2048 threads on a single SM at a time. A complete list of technical specifications for compute capability 5.3 can be found in Appendix A.

2.4.3 NVIDIA Jetson TX2

The software being evaluated for this thesis will also be deployed on the Jetson TX2. It is the newest release of the Jetson TX series. The following sections highlight the primary differences between the Jetson TX1 and TX2.

2.4.3.1 Jetson TX2 CPU

The CPU on the Jetson TX2 consists of six cores. Four of the six cores are the same as the Jetson TX1 CPU cluster, and the remaining two cores are the Denver 2 dual-core CPU cluster. The Denver 2 cluster is optimized for single-thread performance. The Denver 2 cores are also linked together via high-performance coherent interconnect fabric that

allows for seamless multiprocessing. All six of these cores have a clock speed of 2.0 GHz (NVIDIA (A), 2017).

2.4.3.2 Jetson TX2 GPU

The Jetson TX2's integrated GPU consists of two Pascal SMs, each consisting of 128 CUDA cores. The Pascal SM architecture employed by the TX2 improves upon the Maxwell architecture. Each CUDA core on the Jetson TX2's GPU operates at a frequency of 1.3 GHz. The TX2 also increases the amount of DRAM from 4 GB to 8 GB and more than doubles the memory bandwidth from 25.6 GB/s to 59.7 GB/s as compared to the TX1. The size of the L2 cache was increased to 4096 KB. Each Pascal SM comes equipped with 32 CUDA cores specifically designed for double-precision computing. The Jetson TX2's GPU has a compute capability of 6.2. A complete list of technical specifications for compute capability 6.2 can be found in Appendix A.

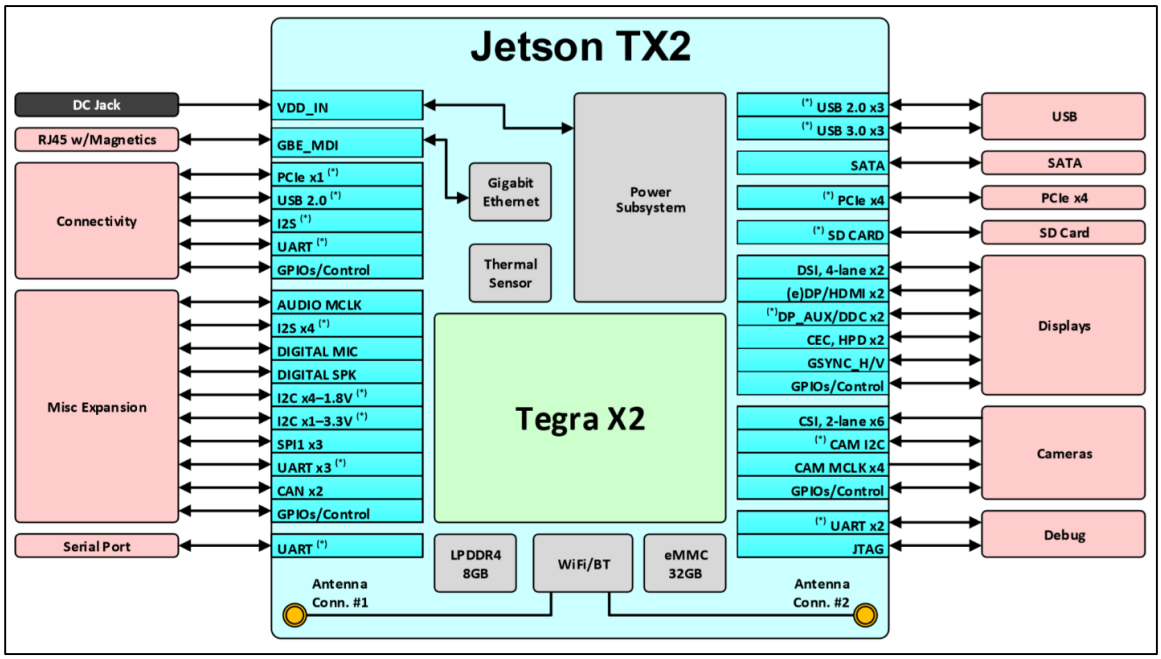


Figure 10. Block Diagram of the Jetson TX2 Development Kit (NVIDIA (A), 2017)

2.4.4 APOD Design Cycle

Even though CUDA was designed to be intuitive to C/C++ programmers, parallelizing existing software applications can still be a challenging task requiring developers to front significant development efforts with little guarantee of return on investment. Therefore, NVIDA introduced an iterative software development cycle to guide programmers in efficient development of parallel applications. The Assess, Parallelize, Optimize, Deploy (APOD) design cycle enables developers to identify aspects of their code that could see performance gains from GPU acceleration, realize those gains, and begin deploying the GPU-accelerated software into operational systems as early as possible (NVIDIA (C), 2017).

2.4.4.1 Assess

The first step in reengineering an existing software application to benefit from parallel computing is to determine which portions of code are most time-intensive (NVIDIA (C), 2017). Developers should create profiles to identify bottlenecks, or *hotspots*, in the program that can be analyzed to determine their suitability to be parallelized. This step in the design cycle prevents developers from investing time parallelizing portions of code that would likely have minimal impact on the overall performance of the application (NVIDIA (C), 2017).

An application profile details the functions where a program spends its time. This allows the developer to identify which routines are most time consuming, which guides the developer in determining which aspects of a program are good candidates for parallelization (NVIDIA (C), 2017).

NVIDIA's profiling tool, **`nvprof`**, was used to support this research. **`nvprof`** is NVIDIA's version of **`gprof`** and can produce several different application profiles, with the *flat profile* being the most applicable. The flat profile lists a program's total execution time by function. An example flat program is presented in Figure 11:

```
$ gcc -O2 -g -pg myprog.c
$ gprof ./a.out > profile.txt
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  genTimeStep
 16.67      0.03     0.01      240     0.04     0.12  calcStats
 16.67      0.04     0.01        8     1.25     1.25  calcSummaryData
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

Figure 11. Flat Profile Produced by nvprof (NVIDIA (C), 2017)

Figure 11 shows that the *genTimeStep* function took an average of 0.02 seconds to execute. However, this information alone does not indicate whether this function is a potential hotspot. Since *genTimeStep* was called 7,208 times, it makes up the largest portion of time spent by the program, and is a potential candidate for parallelization (NVIDIA (C), 2017).

2.4.4.2 Parallelize

Once hotspots have been identified, software developers can parallelize the code (NVIDIA (C), 2017). The objective of this step is not to produce a perfectly optimized parallel implementation, but to investigate whether a certain hotspot has the potential for parallelization (NVIDIA (C), 2017).

Serial code is often structured in such a way that does not expose its inherent parallelism. Therefore, developers must restructure their routines to expose their inherent parallelism, if any exists. For example, recursive loops can often be restructured to use a deterministic solution, disabling dependence of previous iterations (NVIDIA (C), 2017).

Verification must also be accomplished in this step to ensure the hotspot was properly parallelized. Developers must verify parallel implementations yield identical results or results within some error bound. Unexpected results often arise from floating-point values due to how they are computed and stored; thus, identical results are often unattainable in these instances and some small epsilon can be used depending on the application's accuracy requirements (NVIDIA (C), 2017).

2.4.4.3 Optimize

Poorly structured parallel programs often result in slower runtimes than serial implementations, or fail to compile at all. Liberal usage of expensive memory transfers and accesses, improperly partitioning tasks, or having an incorrect understanding of the Device's hardware architecture are often the culprits behind these instances. In order to ensure parallel code is being implemented effectively, developers must take full advantage of all techniques, features, and tools available at their disposal (NVIDIA (C), 2017).

Parallelized kernels may not be properly structured to take full advantage of the GPU's computing power. Thus, after the parallelization of a hotspot has been shown to be feasible it must be optimized to improve performance (NVIDIA (C), 2017). Unlike the 'Parallelize' step, the 'Optimize' step, itself, is iterative. Meaning that for each portion of newly parallelized code, the developer should attempt to optimize the code, verify for

correctness, and record any performance gains. Optimizations can be as high-level as overlapping data transfers between the Host and Device or as granular as fine-tuning individual arithmetic operations (NVIDIA (C), 2017).

Memory Optimization

As stated in Section 2.4.1.2, each access to global memory incurs an expensive time cost. Hence, mitigating the cost of these accesses is often the single most important performance consideration when optimizing a CUDA application (NVIDIA (C), 2017). Programmers can lessen the impact of accessing data from memory by *coalescing* multiple memory accesses into single transactions and by storing data in the Device's more-efficient shared memory (NVIDIA (B), 2017).

Coalescing groups of reads or writes of multiple data items into a single operation distributes the memory access cost across the entire group, versus having individual cost for each memory access. This technique is demonstrated below by using the simple access pattern (NVIDIA (C), 2017). In this access pattern, the k^{th} thread accesses the k^{th} data element. Thus, if the threads of a warp access adjacent 4-byte floating-point variables, which equals a single 128B L2 cache line, the processor will service all 32 threads with a single memory access. In Figure 12, the red rectangle indicates a single 128-byte L2 cache line that can be coalesced into a single memory transaction:

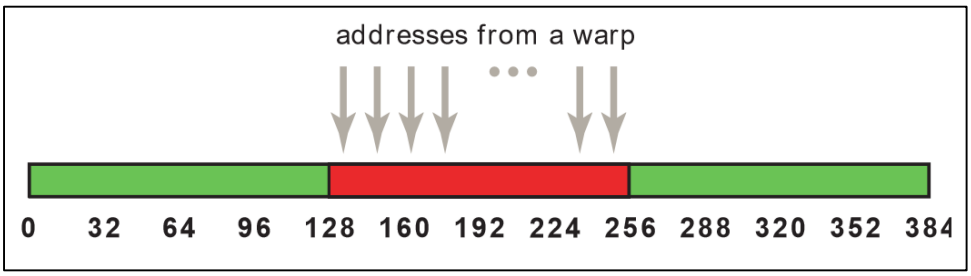


Figure 12. Coalesced Global Memory Access (NVIDIA (C), 2017)

The second technique in mitigating the time cost of accessing data from memory is to use shared memory. Shared memory is designed with high bandwidth and low latency, as it is used by multiple threads to cooperate across thread blocks. To achieve this, it is divided into 32 equal-sized banks that can be accessed simultaneously, allowing all 32 threads in a warp to access the same data at the same time. Furthermore, when multiple threads within a block need to access the same global memory addresses, shared memory can be used to access global memory only once and in an automatically coalesced fashion. This resulting efficiency makes shared memory the most preferred memory type when optimizing a kernel's memory accesses (NVIDIA (C), 2017).

When utilizing shared memory, developers must be careful to minimize bank conflicts (NVIDIA (B), 2017). Bank conflicts occur when *n* threads attempt to access the same memory bank simultaneously, causing the memory accesses to be serialized, decreasing the bandwidth by a factor of *n*. Threads in a single warp, however, are an exception. When threads in the same warp attempt to access the same shared memory bank, copies of the data being accessed are broadcasted to each thread requesting it. This is another reason blocks should be organized into multiples of 32 threads whenever possible (NVIDIA (C), 2017).

Because constant memory is stored in an on-chip cache, it is very efficient under certain conditions. If every thread within a warp accesses a single or a few memory locations in the constant cache, a broadcast occurs, which can be as fast as a register access. However, accesses to different memory locations in the constant cache are serialized; thus, if each thread must access a different memory location, it would take about 32 times as long (NVIDIA (C), 2017).

Minimizing the Impact of Data Transfers

Even on integrated GPUs that share DRAM with the CPU, data must be transferred to the Device through a Peripheral Component Interconnect – express (PCI-e) bus (Otterness et al., 2017), which typically has relatively low bandwidth. Therefore, minimizing the time cost incurred when data is transferred from Host to Device and vice versa is a high priority when optimizing an application. Strategies such as minimizing the number of data transfers, using asynchronous memory copies, and using zero-copy memory can help lessen the impact of costly data transfers (NVIDIA (C), 2017).

The most direct way to minimize the total time cost incurred from data transfers is to do fewer of them. In some instances, functions should be run on the Device even when no performance gains are realized, strictly to refrain from transferring data between Host and Device. It is up to the developer to experiment with their code in order to determine the most efficient way to manage data transfers (NVIDIA (C), 2017).

Developers utilizing a GPU will inevitably need to transfer data between the Host and Device. *Pinned memory* allows for *asynchronous* transfers that can be used to minimize the impact of these transfers. Pinned memory can be allocated without copying data into a separate buffer, resulting in a simplified transfer process. It is important to note that pinned memory is a scarce resource and must be used sparingly. Furthermore, pinned memory takes much longer to allocate. Allocating pinned memory takes on the order of three to five orders of magnitude longer than allocating regular Device memory (Boyer, 2013). The conventional method of transferring data between the Host and Device using the CUDA function **`cudaMemcpy()`** is a *blocking* transfer, meaning the CPU remains ideal until the memory transfer is complete. Conversely, asynchronous transfers using

`cudaMemcpyAsync()`,are *non-blocking*, meaning that the CPU can continue to do work while the transfer is being executed. Asynchronous transfers must be used in tandem with pinned memory; hence, they are limited by the amount of pinned memory available (NVIDIA (C), 2017).

Depending on the program structure, data can often be broken into independent chunks and transferred to the Device asynchronously. Since the Host is free to do work while a data transfer is being executed, the Host can launch kernels that will be queued up to execute immediately after the data on which they are dependent is transferred. Furthermore, some devices can perform asynchronous memory transfers concurrently with kernel executions. When this occurs, the k^{th} kernel executes while the data needed by the $k^{\text{th}} + 1$ kernel is being transferred. Overlapping kernel execution and data transfers can result in faster completion times, as illustrated in Figure 13:

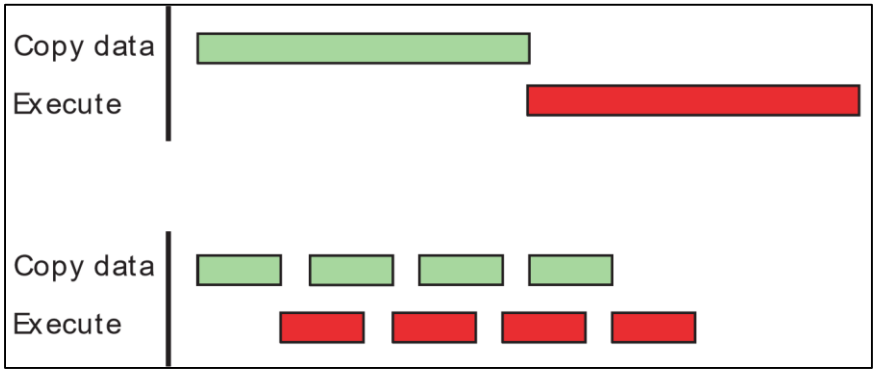


Figure 13. Concurrent Data Copy and Kernel Execution (NVIDIA (B), 2017)

The top "Copy data, Execute" represents the conventional sequential blocking method, in which the Host remains idle until the data transfer is complete. The bottom shows the concurrent method in which kernels can execute while other data is being

transferred, resulting in increased runtime performance. The transfers should still be combined whenever possible, as each transfer has intrinsic overhead (NVIDIA (C), 2017).

On integrated GPUs like the Jetson, the GPU has access to the CPU's DRAM. This shared DRAM allows developers to utilize a feature called *zero-copy memory*. Zero-copy memory enables the passing of pointers to shared memory where data used by the kernel is located, which eliminates the need to explicitly transfer data to and from the Host and Device (NVIDIA (B), 2017). Zero-copy does not allow for the caching of data on the Device, meaning each time the data is accessed through zero-copy, it must be accessed off-chip in DRAM. Therefore, zero-copy should only be used for data that is used sparingly on the Device (Otterness et al., 2017).

Maximizing Occupancy

In order to maximize performance, the multiprocessors of the Host and Device should be kept as busy as possible. A poorly structured application with multiple idle processors will likely result in sub-optimal performance (NVIDIA (C), 2017). Therefore, it is imperative to organize an application to use threads and blocks in such a way that achieves the maximal *occupancy* of the available hardware. Occupancy can be summarized as the ratio between the number of active warps per multiprocessor and the maximum number of possible active warps. Consider compute capability 5.3, which can support up to 64 active warps. This means 64 active warps per SM must be present in order for the SM to be fully occupied. Several factors can improve occupancy, such as using concurrent kernel executions, using the proper number of threads per block and registers per thread, minimizing register dependencies, and using the proper amount of shared memory per block (NVIDIA (C), 2017).

To minimize the number of idle processors on an SM, independent kernels can be executed concurrently to ensure occupancy is being maximized. If a kernel is only using 50% of an SM's processors, another kernel can be launched to utilize the remaining processors. Overlapping kernel execution enable SMs to be fully occupied, even when single kernels only use a fraction of an SM's resources. Note that Compute Capability 5.3 supports up to 16 kernels executing on an SM simultaneously (NVIDIA (C), 2017).

Because warps are groups of 32 threads, SMs are designed to handle multiples of 32 threads at once. For example, Compute Capability 5.3 and 6.2 can each handle up to 2048 (i.e., 32×64) threads per SM. This design pattern requires the number of threads per block to be in multiples of 32 to fully maximize an SM's occupancy (NVIDIA (C), 2017).

The number of 32-bit registers on an SM can be a limiting factor when maximizing occupancy. Because register storage enables the low-latency access of local variables by keeping them on-chip, it is tempting to partition blocks such that they use enough registers to store all of their local variables. However, registers are a limited asset; if too many registers are being used by a thread, the number of resident thread blocks on the SM is lowered, which lowers occupancy. Therefore, blocks must be partitioned in a way that they take advantage of registers' low-latency for local variables while still maintaining the highest occupancy possible. It is also important to note that since registers are 32 bits, a single register can store a single **`int`** (32 bits long) while it takes two registers to store a single **`double`** (64 bits long) (NVIDIA (C), 2017).

Register dependencies can also adversely affect occupancy. A register dependency occurs when an impending instruction requires the result of a calculation stored in a register. Because the current latency on CUDA-enabled devices is 24 cycles, threads must

wait 24 cycles before accessing the data stored on a register. Thus, register dependencies can force threads to stall as they await the data on a register to become available (NVIDIA (C), 2017).

Shared memory is also a potential limiting factor when calculating occupancy. Much like using registers to store local variables, shared memory should be utilized to the maximum extent possible due to its low-latency memory access. It too, however, is a scarce resource and can limit the amount of resident warps on an SM. Hence, developers should consider the amount of shared memory available when determining block size (NVIDIA (C), 2017).

Achieving maximum occupancy through trial and error would be an exhaustive task. Therefore, the CUDA Occupancy Calculator should be used to determine the optimal number of threads per block, registers per thread, and the amount of shared memory used per block. Figure 14 shows the occupancy of the Jetson TX1. This example uses 256 threads per block, 32 registers per thread, and 8192 bytes of shared memory per block (NVIDIA (C), 2017).

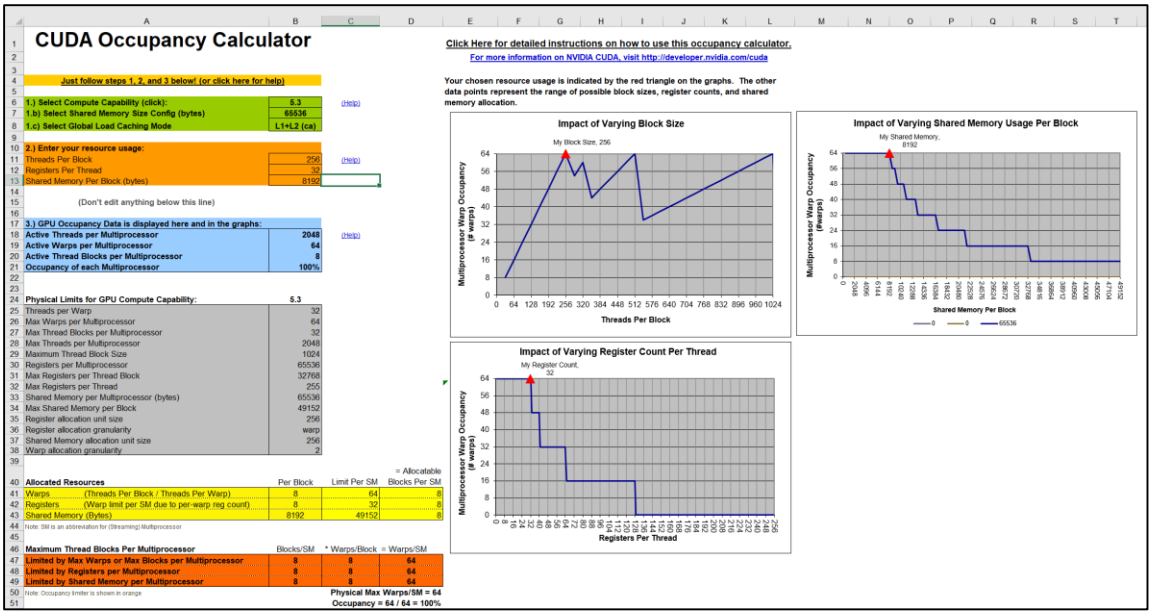


Figure 14. CUDA Occupancy Calculator

Minimizing Thread Divergence

Because all 32 threads within a warp execute one common instruction at a time, full efficiency cannot be achieved if the 32 threads within a warp do not have a common execution path (NVIDIA (B), 2017). Control flow instructions such as **`if`**, **`switch`**, **`do`**, **`for`**, and **`while`** can cause threads within a warp to diverge by steering them down different execution paths. When this occurs, the warp serially executes each branch path taken, disabling all other threads within the warp until the threads converge, resulting in slower kernel execution times (NVIDIA (C), 2017).

Control flow statements are sometimes necessary in parallel computing, however, often utilizing the **`threadIdx`** keyword to direct specific threads to perform specific tasks. Kernels that require thread-ID-dependent control flow statements should be constructed to minimize the number of divergent paths per warp. This can be accomplished

by partitioning a thread's execution path based on the warp to which it belongs, instead of its specific thread ID (NVIDIA (C), 2017).

Instruction Optimization

Division and modulo instructions are particularly expensive to perform on a GPU. Thus, replacing these instances with equivalent *shift* operations can result in performance gains. In the case where n is a power of 2, (i/n) is equivalent to $(i \gg log2(n))$. For modulo operations where n is a power of 2, $(i \,\%\, n)$ is equivalent to $(i \,\&\, n-1)$. These optimizations are considered low-priority, but can provide significant reductions in runtimes if a kernel uses a large number of division and/or modulo operations (NVIDIA (C), 2017).

2.4.4.4 Deploy

Before moving on to the next hotspot identified in the 'Assess' step, the developer should deploy the partially reengineered code onto a test system. This allows users to see partial performance gains as early as possible and minimizes risk by isolating new bugs introduced to the software by providing evolutionary versions of the application (NVIDIA (C), 2017).

When integrating CUDA files with the ".cu" extension with ".c/.cpp" C/C++ files, function names become mangled. This can be avoided by using the **`extern "C"`** wrapper on relevant functions within the ".cu" file, which ensures the function names remain demangled. Once the functions are declared in the C++ header, the functions inside the ".cu" files can be called from within the ".cpp" files (Oak Ridge National Laboratory, 2013).

When integrating C/C++ files and CUDA into a single application, the process of *separate compilation* shown in Figure 15 is used. The ".cu" files contain all CUDA code. These files include functions executed on the Host as well as functions executed on the Device; thus, nvcc must be used for ".cu" files. All ".cpp" files can be compiled with a standard compiler, such as g++. The object files created by these are then linked together using nvcc to create the executable.

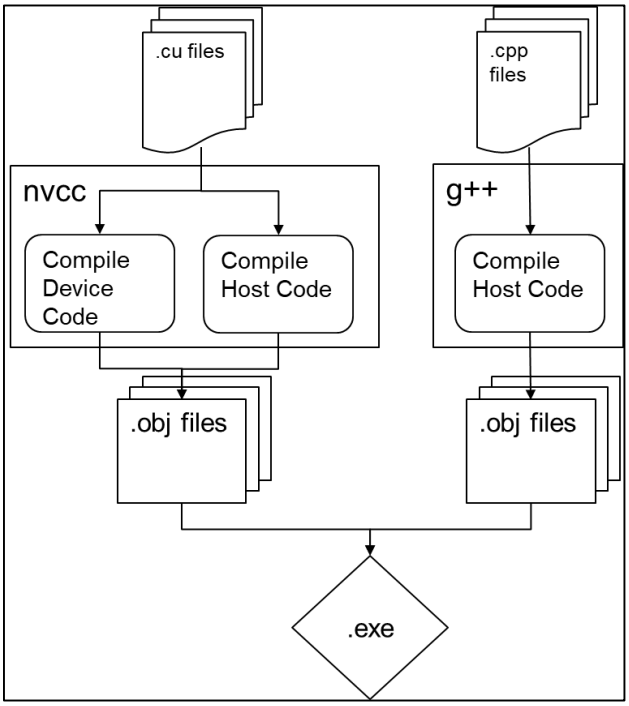


Figure 15. Separate compilation process used to combine ".cu" and ".cpp" files

3. Methodology

This chapter describes the methodology used to implement SP on the Jetson TX1/TX2 and the methodology used to test the two hypotheses presented in Section 1.3. The steps taken to successfully compile the SP software on the Jetson TX1 and TX2 are discussed first, followed by the methods to determine the validity of its solution. The second section discusses the methods for determining the optimal combination of compiler flags to apply to SP. The development of a parallel geopotential model and the methods used to determine its validity are then discussed. The final section presents the methods used to determine the most efficient implementation of SP on the Jetson TX1/TX2.

3.1 Implementing Special Perturbations on the Jetson TX1 and TX2

The first task in completing this research was to implement SP on the Jetson TX1 and TX2 and verify that it produced the same solution as the Windows version of the software. The first step taken to accomplish this was to reorganize the package diagram presented in Section 2.1 to reduce redundant dependencies and resolve linkage errors so the SP software could compile and run on the Jetson TX1/TX2. The validity of the Linux version of SP was then determined to ensure it converged to the correct solution.

3.1.1 Compiling Special Perturbations on the Jetson TX1 and TX2

The package diagram presented in Section 2.1 shows two primary issues that had to be overcome in order to run SP on the Jetson TX1 and TX2. First, several ".cpp" files are directly included in other files. Second, redundant dependencies unintentionally add to the complexity of the code by requiring more files than necessary. These issues can result

in increased compilation times or compilation failure; therefore, several components of the software had to be restructured.

The SP software includes three different routines used to read the three primary types of data needed for testing and operation. The *ReadMyTruth, ReadIneritalRData,* and *ReadSTK* routines were each contained in their own ".cpp" file. These routines are used in the *main* function to read observational data, meaning their files had to be included in SPLstSq.cpp. Since each of these files only contains a single function, they were easily changed to header files so they could be included in SPLstSq.cpp without issue.

Routines contained in Hamming.cpp and Observation.cpp are also used in the main function. These files, however, both contain multiple functions and are relatively large. For these reasons, Hamming.h and Observation.h header files were added so the functions contained in Hamming.cpp and Observation.cpp could be included in SPLstSq.cpp.

The two routines developed to account for the effects of air drag were contained within the Atmosphere.cpp file. These functions are used in the dynamics model which is comprised of the EarthTruth.cpp and EarthTruth.h files. In order to prevent the inclusion of the Atmosphere.cpp in another file, the two air drag routines it contained were added to EarthTruth.cpp and EarthTruth.h. This removed the necessity of including a .cpp file in another file.

The SPLstSq.cpp file originally included two header files, LinearEquations.h and SingularValue.h, that only included other files. The LinearEquations.h file included the numerical.h and ludcmp.h files, while the SingularValue.h file included the numerical.h and svd.h files. These two files were removed from the codebase and the files they contained were included directly in SPLstSq.cpp.

The TwoBodyProblem.h and Observartion.h files contained the class definitions and functions in the same file. Therefore, the TwoBodyProblem.cpp and Observartion.cpp files were added to separate the two. Furthermore, include guards (e.g., **`#ifndef`**, **`#endif`**) were added to each header file used in the software. Include guards prevent duplicate expansion that can result in linkage errors. These changes are summarized in the following restructured package diagram:

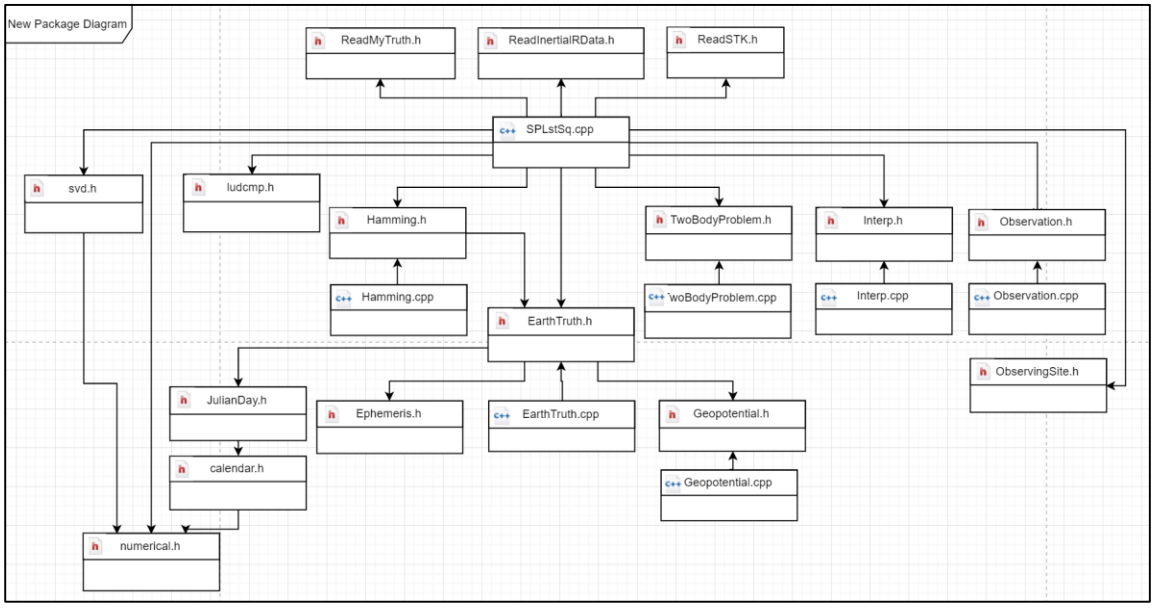


Figure 16. Updated Package Diagram of the SP Software

3.1.2 Verifying for Correctness

Once the SP software successfully compiled and ran on the Jetson TX1 and TX2, its accuracy in comparison to the Windows version had to be verified. The state vector of the spacecraft is comprised of seven elements. The first three elements are the position component, given in XYZ dimensions. The second three elements are the velocity component, also given in XYZ dimensions. The seventh element is the *B** air drag

coefficient. The following parameters were used as inputs for both the Windows version and the version running on the Jetson TX1/TX2:

Table 5. Initial State Vector

Position X:	-3.71191588114069e+3
Position Y:	-5.86581648105739e+3
Position Z:	2.94244366723117e-1
Velocity X:	5.63178840625886e+0
Velocity Y:	-3.56186511007924e+0
Velocity Z:	3.6161019950714e+0
*B** (Air Drag):	0.00e+0

Table 6. Additional Input Parameters

Epoch Time (MJD):	1.51995746598202e+4
D&O of Geopotential:	20
Sea Level Pressure:	1.01325e+5
Rejection Limit (km):	1.00e+4

Through each iteration of SP, each element of the state vector is adjusted until they converge on the solution. Since this test case converges on the fourth iteration, there are four additional state vectors in the Windows version. These state vectors were used as the success criteria for determining if SP produced the correct solution. Each dimension of the state vector's position and velocity vectors output by the Linux version, along with *B**, was

compared to the output of the Windows version to determine if they converged to the same solution. These results are presented in Section 4.1.

3.2 Optimizing the Serial Version of Special Perturbations

Due to the large number of compiler flags available, testing each one to determine if it benefited SP on the Jetson TX1/TX2 was impractical. Thus, only a small subset of compiler flags was considered for this research. This section presents the methodology and the reasoning used to determine which combination of compiler flags resulted in the fastest runtimes.

3.2.1 Determining the Optimal Combination of Compiler Flags

The list produced by CERN openlab (Section 2.3, Table 3) was used to down-select from all possible compiler optimizations. The "-nolib-inline" compiler flag was not recognized by the compilers used for the Jetson TX1 and TX2; for this reason, it was not considered. Furthermore, the "-O2" compiler flag was added to the list and compared to the initial, unoptimized version of SP running on the Jetson TX1/TX2 instead of using the "-O2"-optimized version as the initial condition, as in the CERN openlab study (Botezatu, 2012).

The method for assessing the significance of the speedup was taken from the CERN openlab study. That is, the benefit of a certain compiler optimization was determined to be 'significant' if it resulted in a $\geq 1\%$ reduction in runtime when compared to previous versions. This experiment was considered a success if any combination of compiler optimizations were found to reduce the runtime of SP by at least 1%.

The runtime of SP on the both Jetsons was very stable. Therefore, accurate timing statistics could be measured by running the SP program only 50 times. Degree and order of 20 was used to test all compiler flags and the standard CPU timer, **`clock()`**, was used to accurately measure runtimes.

3.2.2 Testing the First Hypothesis

The first hypothesis presented in this thesis was to determine if a combination of the selected compiler flags could be applied to SP such that it decreased its runtime compared to the initial serial version on the Jetson TX1/TX2. Once the optimal combination of compiler flags was determined, this hypothesis was tested by running both versions of SP at multiple degrees and orders of the geopotential. The degrees and orders used started at 10 and increased in increments of 5 until degree and order of 50 was reached, with SP being run 50 times at each degree and order. This was completed for the initial serial and optimized serial versions to attain accurate runtime statistics. Success was achieved if the optimized serial version was faster than the initial serial version for any degree and order less than or equal to 50.

3.3 Applying APOD to Special Perturbations

The second hypothesis presented in this thesis questions whether the geopotential model employed by the SP software could be implemented using parallel computing. To accomplish this, the APOD software development cycle is applied to the SP codebase to develop a stand-alone parallel version of the geopotential model on the Jetson TX1. Degree and order of 50 was used for this experimentation. Next, the parallel geopotential model was integrated into the SP software to determine if it converged to the same solution as the

Windows version. The runtimes of SP using the parallel geopotential model were then compared to the runtimes of the initial serial version running on the Jetson TX1/TX2 to determine if any speedup was attained.

3.3.1 Assessing Special Perturbations for Hotspots

The first step taken in applying the APOD software development cycle was to assess SP to determine the most time-consuming components of the code. This was accomplished using the profiling tool, **`nvprof`**. The **`nvprof`** profiling tool produced the following flat profile:

Table 7. Flat Profile of SP Produced by `nvprof`

% Time	Cum Seconds	Self Seconds	Calls	Self μs/call	Total μs/call	Name
84.68	51.22	51.22	615950	83.15	83.15	Geopotential::geoECR
8.96	56.64	5.42	615950	8.80	95.08	Dynamics::Rhs
3.16	58.55	1.91	307940	6.20	196.39	hamming
2.00	59.76	1.21	615950	1.96	85.11	Geopotential::geoECI
0.60	60.12	0.36	615950	0.58	0.58	Dynamics::Atm
0.51	60.43	0.31	615950	0.50	85.70	Dynamics::Hder
0.07	60.47	0.04	615950	0.06	0.06	GreenwichSiderialTime
0.02	60.48	0.01	616453	0.02	0.02	SecondsToJulian

The profile shows that the function used by the geopotential model, *goeECR*, is by far the most time-consuming routine, taking just under 85% of the total runtime. Therefore, it was the primary focus when attempting to use parallel computing to reduce the runtime.

3.3.2 Parallelizing the Geopotential Model

The second step of the APOD cycle, 'Parallelize,' was then applied to the geopotential model. CUDA was used to develop a stand-alone version of the *geoECR* routine. The results of the parallel version were then compared to those of the initial serial version to verify their correctness.

The recursion used by Pines Method to calculate the $A, \rho, c_{real}, c_{imag}$ arrays presented a challenge when attempting to parallelize the model. Deterministic solutions for each array were developed to unravel this recursion so each element of the arrays could be calculated independently and in parallel. However, the non-recursive methods proved far less efficient than the recursive versions. Each non-recursive version used a **`do-while`** loop to calculate its elements, with the loop iterating up the particular thread's index. This created race conditions that required barriers to synchronize the threads, greatly reducing the efficiency of this method. Hence, attempts to calculate these particular arrays on the GPU were abandoned, meaning they had to be calculated on the Host and transferred to the Device each time the geopotential model was called.

The initial parallel implementation followed these general steps:

Table 8. Steps of Parallel Geopotential Model

1	Calculate s, t, u, r variables
2	Calculate $A, \rho, c_{real}, c_{imag}$ vectors
3	Allocate Device memory (x19)
4	Transfer inputs from Host to Device (x5)
5	Launch kernel
6	Transfer outputs from Device to Host (x14)
7	Sum outputs
8	Calculate $fECR$ and P matrices

Where the inputs consisted of the $A, \rho, c_{real}, c_{imag}$ vectors and the r variable and the outputs consisted of the potential U and the partially summed coefficients $fac1$ through $fac44$.

The size of the arrays used to compute the geopotential are determined by the degree and order of the model. These matrices are lower triangular with their upper halves only containing zeros. However, for ease of implementation, the initial parallel kernel was configured such that each thread block corresponded to a particular row and each thread to a particular element in that row. In other words, for a degree and order of nine, the initial parallel version launched $9 + 1 = 10$ blocks with 10 threads each, with almost half of the threads remaining idle. This is shown in Figure 17:

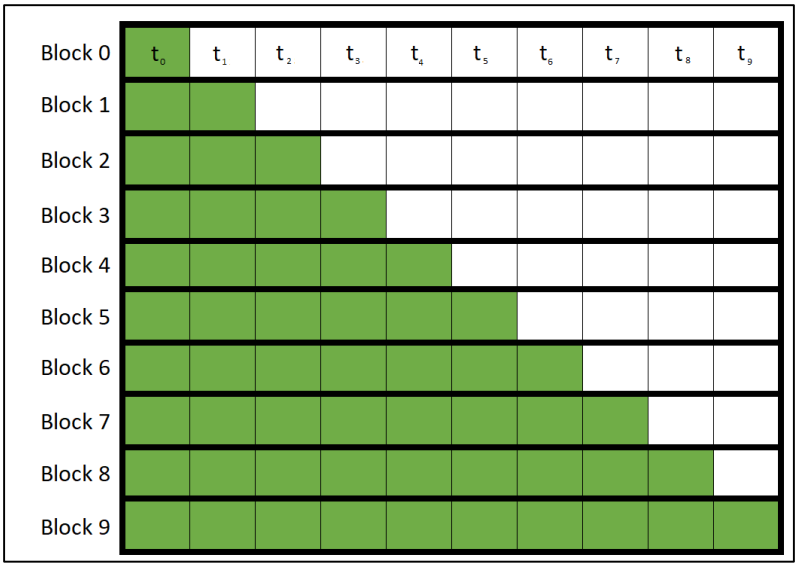


Figure 17. Representation of the Original Parallel Geopotential Model

Device memory was allocated using the **`cudaMalloc()`** function. For the initial kernel, the Device memory for each individual input and output was allocated separately, for a total of 19 separate calls to **`cudaMalloc()`**. The inputs were then transferred from Host memory to Device memory using **`cudaMemcpy()`**.

The kernel also required the use of the spherical harmonic coefficients of C and S. Since these matrices remain constant, they were allocated using **`__device__`** memory, meaning that they only needed to be allocated and transferred once. The kernel used these two matrices along with the inputs to calculate the remaining arrays (e.g., the $D, E, F, G,$ and H) needed to compute the geopotential. These arrays were used to calculate the outputs of the kernel. All the outputs within a single block were summed to a single value on the Device. The outputs were individually transferred from the Device to the Host, where they were further summed up and used to calculate the $fECR$ and P matrices.

The potential U variable and the $fECR$ and P matrices are the primary outputs of the geopotential model. Thus, these elements were compared to the results of the serial version to determine if the parallel version was correct.

3.3.3 Optimizing the Parallel Geopotential Model

Like the APOD process as a whole, the 'Optimize' step, itself, is iterative. This step was repeated until all optimization strategies were exhausted. The runtimes of each optimization were recorded to measure any performance gains and the results were verified against the initial serial version to ensure correctness.

As stated in Section 2.4.4.3, CUDA enables the use of a technique called *concurrent execution* that allows the CPU to continue doing work while transferring data between the Host and Device or while a kernel is being executed. There was little work for the CPU to accomplish while the geopotential kernel was executing; however, there was some potential for the CPU to remain busy while transferring data. This method of using asynchronous data transfers required the use of pinned memory, which must be allocated using **`cudaMallocHost()`**. However, allocating pinned memory is much slower than allocating pageable memory. This offset any time saved through concurrent execution. Therefore, these efforts were abandoned.

3.3.3.1 First Iteration: Minimizing Effects of Data Transfers

For the first iteration of the 'Optimize' step, the focus was to minimize the impact of data transfers between the Host and Device. First, all inputs were combined into a single vector to reduce the number of allocations and transfers. The same was done for all outputs. Second, the use of **`__device__`** memory was prioritized to allow data to be allocated

only once throughout the lifetime of the application. This design was preferred because the SP program calls the geopotential routine thousands of times per iteration.

Even though the same amount of data was being allocated and transferred, minimizing the discrete number of allocations and data transfers is important because of the inherent overhead in each allocation and transfer. The $s, t, u,$ and r variables were combined with the $A, \rho, c_{real},$ and c_{imag} matrices into a single vector. Similarly, the potential U and all partial coefficients of acceleration were combined into a single output vector. This decreased the total number of memory allocations and data transfers needed each time the geopotential routine was called from nineteen to two.

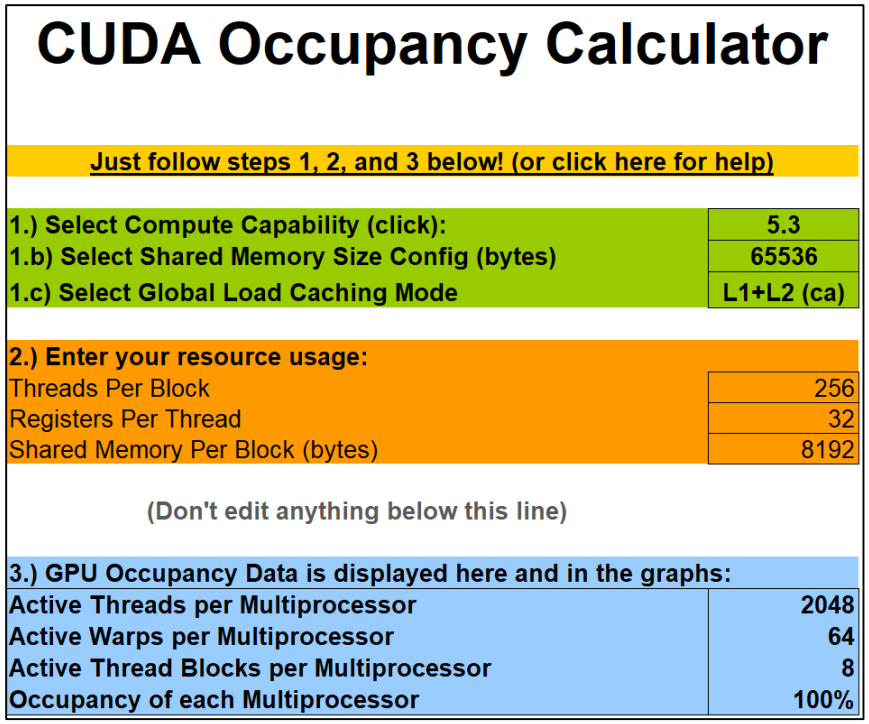
Because the SP algorithm computes the geopotential thousands of times over the course of a single iteration, the memory used for the input and output vectors could remain allocated and be used multiple times. This approach was already being used for the C and S matrices and was extended to the input and output vectors by declaring them as `__device__` memory. Unlike C and S, which remain constant, the input and output vectors had to be transferred to the Device each time the geopotential was calculated. However, this still prevented the need to re-allocate memory for the inputs and outputs.

3.3.3.2 Second Iteration: Maximizing Occupancy

The second iteration of the optimization cycle focused on maximizing the occupancy of the GPU. The kernel was restructured to account for the use of lower triangular matrices, which left almost half of the threads idle in the original kernel. The CUDA Occupancy Calculator was then used as a guide to ensure proper utilization of the GPU's computing resources.

In the original parallel kernel, each row corresponded to a thread block and each element in the row corresponded to an individual thread. Because these matrices are lower triangular, the first row only contained one element, with the rest being zeros. Thus, in the original configuration, only one thread in the first thread block accomplished any work. This was changed to eliminate the idle threads. However, the vectors still needed to be traversed as though they were square matrices. Therefore, the $rowIdx$ and $colIdx$ vectors were created to allow each thread to recall its original position in the matrix. Since these vectors remain constant throughout the lifetime of the application, they were declared as **`__device__`** vectors so they only had to be allocated and transferred to the Device once.

The CUDA Occupancy Calculator was used to ensure the maximum occupancy was achieved within the given hardware limitations. The first limiting factor was the number of physical CUDA cores on the GPU. GPUs of Compute Capability 5.3 and 6.2 both have two SMs with 128 CUDA cores each, for a total of 256. This is the maximum number of active threads possible. By design, it is also a multiple of 32, meaning it is aligned with the amount of threads per warp. For these reasons, the number of threads per block was changed to 256.



CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	5.3
1.b) Select Shared Memory Size Config (bytes)	65536
1.c) Select Global Load Caching Mode	L1+L2 (ca)
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	32
Shared Memory Per Block (bytes)	8192
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

Figure 18. Improved Occupancy of Parallel Geopotential Model for the Jetson TX1

The second limiting factor was the amount of `__shared__` memory available for each thread block. This type of memory is the most efficient and should be used as much as possible. However, if the recommended amount of shared memory is exceeded, the GPU will prevent other blocks from executing until shared memory is freed, thus reducing occupancy.

Figure 19 shows how occupancy is affected by the amount of shared memory when each block has 256 threads. When block size is 256, up to 8,192 bytes of shared memory can be used and 100% occupancy still be achieved. If each block uses 14,864 bytes of shared memory, only 50% occupancy is possible. For this reason, the amount of shared memory allowed per block was not allowed to exceed 8,192 bytes.

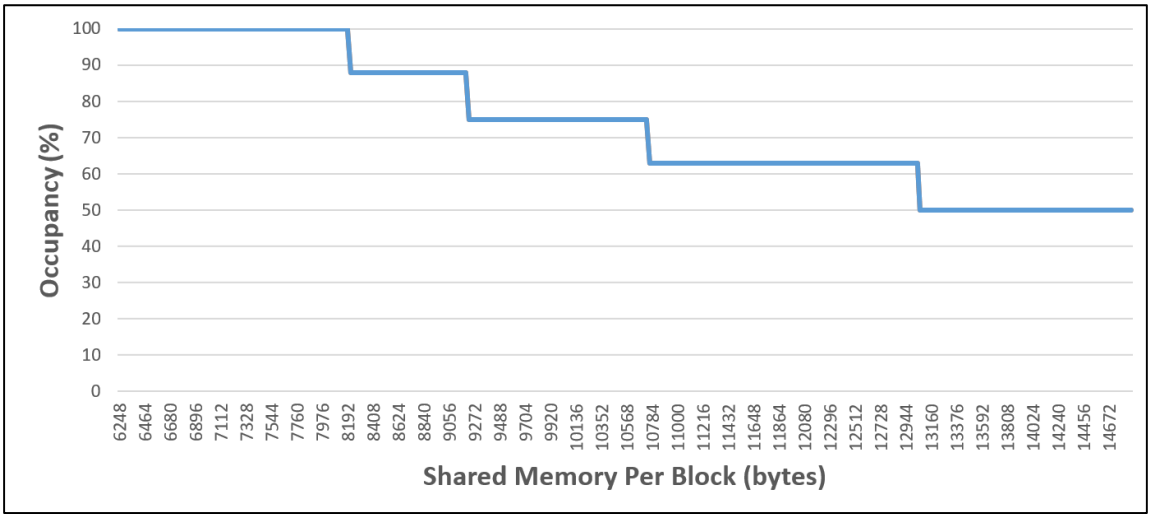


Figure 19. Occupancy as a Function of Shared Memory

These 8,192 bytes of shared memory were divided into four 2,048 byte chunks. Since the summation operation requires the use of shared memory, three of the chunks were used to allocate shared workspaces to sum up the factors of the partial coefficients. These workspaces are reused to calculate the 14 outputs. The input vectors $\rho, c_{real},$ and c_{imag} are used repeatedly, so the remaining chunk of shared memory was used to store these vectors.

The third limiting factor of occupancy was the amount of local registers used by each thread. With 256 threads per block, 100% occupancy was only achieved when each thread used 32 registers or fewer. This was accomplished by passing the **`–maxrregcount=32`** flag to the compiler that prevented threads from decreasing the occupancy by using more than 32 registers.

3.3.3.3 Third Iteration: Applying Reduction Operation

The partial factors calculated on the GPU had to be summed to form the final partial coefficients of acceleration. This summation operation was able to be performed in parallel on the GPU by using a reduction operation. The following simple parallel reduction routine was developed that sums all of the partial factors elements within a given thread block:

```
int n = 8;
int Exp = 1;
for(int i = 0; i < n; i++)
{
  Exp = Exp*2;
  if(threadIdx.x % Exp == 0)
  {
        facA[threadIdx.x] =
facA[threadIdx.x]+facA[threadIdx.x+(Exp/2)];
        facB[threadIdx.x] =
facB[threadIdx.x]+facB[threadIdx.x+(Exp/2)];
        facC[threadIdx.x] =
facC[threadIdx.x]+facC[threadIdx.x+(Exp/2)];
      }
}
```

Because the blocks are of size 256, the **`for`** loop must iterate $\log_2 256 = 8$ times. This routine reduced the total number of sequential double-precision arithmetic operations required per block from 3,584 to 112. It was further optimized by *unrolling* the **`for`** loop. This prevented the need to initialize and calculate intermediary values such as i and n.

3.3.3.4 Fourth Iteration: Minimizing Thread Divergence

In order to calculate the partial coefficients, each thread requires the use of the n^{th}, $n-1^{\text{st}}$, and $n-2^{\text{nd}}$ elements from the c_{real} and c_{imag} vectors, where n is the thread's column index. Thus, **`if`** statements had to be implemented to ensure any threads with a column index of two or less did not attempt to access elements outside of the bounds of the vector. This introduced thread divergence that negatively affected the efficiency of the kernel.

This thread divergence was minimized by *zero-padding* the c_{real} and c_{imag} vectors. That is, the first two elements of these vectors were changed to zeros, and the rest of the elements vector were offset by two. This meant that a thread with a column index of zero could access the $0 - 2 = -2^{nd}$ element without conflict. In the original parallel kernel, eight **`if`** statements that diverged based on a thread's column index were required. Through zero-padding, this was reduced to three.

3.3.3.5 Fifth Iteration: Instruction Optimization

The reduction operation developed for the kernel used the modulo operator a total of 40 times per thread. Although a single invocation of the modulo operator per thread is relatively insignificant to the overall runtime of the kernel, 40 invocations of the modulo operator can hamper performance. Therefore, each of these instances was changed to the equivalent shift operation. Even though this optimization is low-level, the benefit increases with the number of threads being executed.

3.3.3.6 Measuring Performance when Optimizing the Parallel Geopotential Model

To ensure each optimization strategy resulted in adequate performance gains, accurate runtime statistics had to be calculated for the stand-alone parallel geopotential model. To accomplish this, the setup, execution, and post-processing for the parallel geopotential model was wrapped in a **`for`** loop and iterated 1,000 times. Each time the **`for`** loop iterated, the runtime was output to a spreadsheet. A shell script was used to repeat this process 100 times for each version of the geopotential model, and resulted in 100,000 data points per version. *cudaEvents* were used to accurately measure runtime.

3.3.4 Deploying the Parallel Geopotential Model

Once the parallel geopotential was optimized, it had to be integrated into the SP codebase in order to determine if it converged to the same solution as the serial version. This was accomplished by using **`extern "C"`** to ensure the function names in the CUDA file were able to be read by the C++ files and by compiling the CUDA file separately from the C++ files.

The *CudaConstructor* and *CallGeoEcrKernel* functions were wrapped in **`extern "C"`** and declared in the Geopotential.h header file. The *CudaConstructor* function transfers the $C, S, rowIdx,$ and $colIdx$ vectors to the Device. Because these inputs remain constant throughout the lifetime of the application, they only need to be transferred once. Thus, the *CudaConstructor* function is only called a single time. The *CallGeoEcrKernel* function calculates the $A, \rho, c_{real},$ and c_{imag} vectors, transfers them to the Device, calls the kernel, and performs post-processing on the results. The kernel itself, however, was not required to use the **`extern "C"`** wrapper since the kernel was both declared and called from within the CUDA file. The ".cu" file containing the *CudaConstructor* and *CallGeoEcrKernel* functions were compiled with nvcc, and all ".cpp" files used by SP were compiled using g++. The resulting ".obj" files were then linked together with nvcc to form the executable.

3.3.5 Verifying the Parallel Version of Special Perturbations

To determine if the parallel geopotential model worked properly, SP had to converge to the correct solution using the parallel geopotential model. This verification was accomplished using the same methodology described in Section 3.1.2. The solution of

the parallel version was compared to the truth data produced by the Windows version of SP to determine if the parallel version converged to the correct solution.

3.3.6 Testing the Second Hypothesis

The second hypothesis presented in this thesis was to determine if portions of SP could be implemented in parallel on the Jetsons' GPUs such that it reduced the runtimes of SP compared to the initial serial version. This hypothesis was tested using the same methodology described in Section 3.2.2, but used *cudaEvents* in lieu of CPU timers to accurately measure the parallel version of the code. Success was achieved if the parallel version of SP was faster than the initial serial version for any degree and order less than or equal to 50.

3.4 Determining the Most Efficient Implementation

Once the initial serial version of SP was optimized via compiler flags and the parallel geopotential model was integrated into the SP codebase, the average runtimes of each version of SP on the two Jetsons were compared. Accurate runtime statistics of the serial versions of SP were taken using the same methodologies described in Sections 3.2.2 and 3.3.6.

Because the implementation of SP used in this research uses geopotential of up to degree and order of 50, the degree and order was started at 10 and incremented by 5 until 50 was reached for the initial serial, optimized serial, and parallel versions of SP. The input parameters presented in Section 3.1.2 were used. The results were then compared to one another to determine the best way to implement SP on the Jetson TX1 and TX2. The

percentage of improvement of the Jetson TX2 over the Jetson TX1 was also compared for each version of SP.

4. Analysis and Results

This chapter presents the results and analysis of the experiments described in Chapter III. The first section analyzes the state vectors produced by the initial serial version of SP on the Jetson TX1/TX2 compared to the state vectors produced by the Windows version. The results of applying different combinations of compiler optimizations to SP are discussed in the second section. The third section presents the results of developing and optimizing the parallel geopotential model. The results of SP using the parallel geopotential model are also analyzed to show that it converged to the correct solution. The final section in this chapter compares the runtimes of the initial serial version of SP to the optimized serial and parallel versions on the Jetson TX1 and TX2 to determine the most efficient way to implement SP on each Jetson. This section also compares the performance of the Jetson TX2 over the TX1.

4.1 Implementing Special Perturbations on the Jetson TX1 and TX2

The first task completed for this research was to implement the SP software on the Jetson TX1 and TX2 such that both converged to the correct solution. This section presents the resulting state vectors produced by SP on the Jetson TX1/TX2 compared to truth data produced by the Windows version.

The test case used in this research iterated through SP a total of four times before it converged to the correct solution. Thus, each component of the state vectors produced by SP on the Jetson TX1 and TX2 was compared to the truth data at each iteration to determine the extent of the deviation.

The root-mean-square (RMS) of the position and velocity components of the state vectors, as well as the air drag coefficients, over each iteration of SP are shown in Figures 20-22. Figure 20 shows the RMS for the position component of the state vectors at each iteration of SP. Figures 21 and 22 show the RMS for the velocity components of the state vectors and the air drag coefficients, respectively. Each of these components of the state vectors converge to the solution on the fourth iteration of SP.

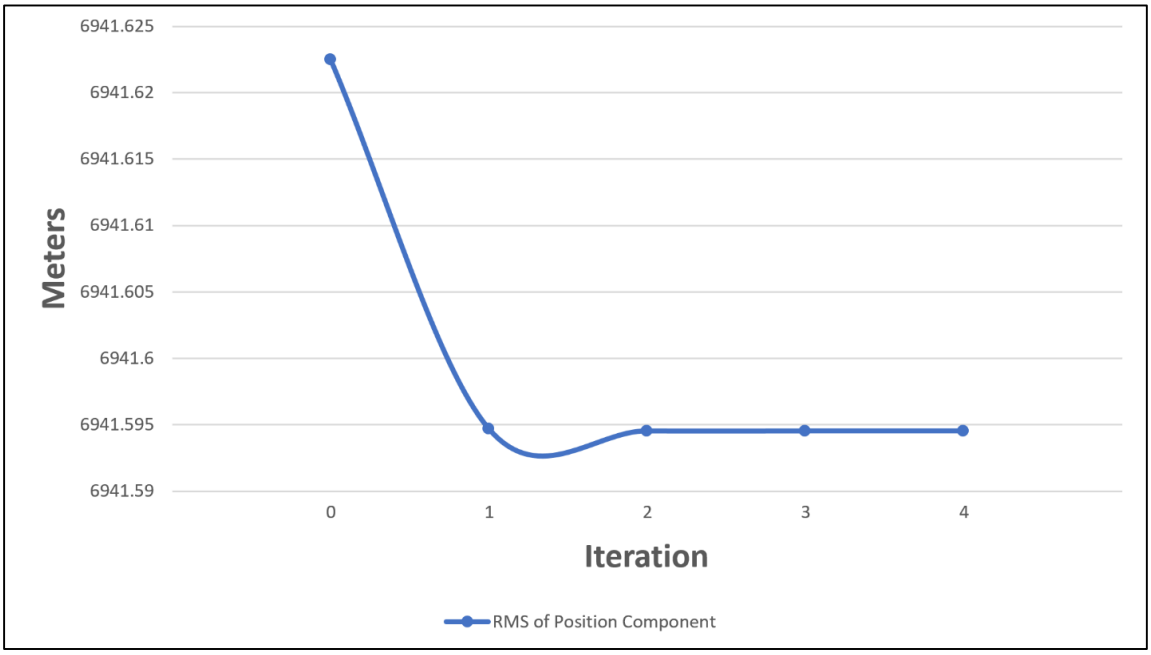


Figure 20. RMS of Position Components Converge, D&O = 20

Figure 21. RMS of Velocity Components Converge, D&O = 20

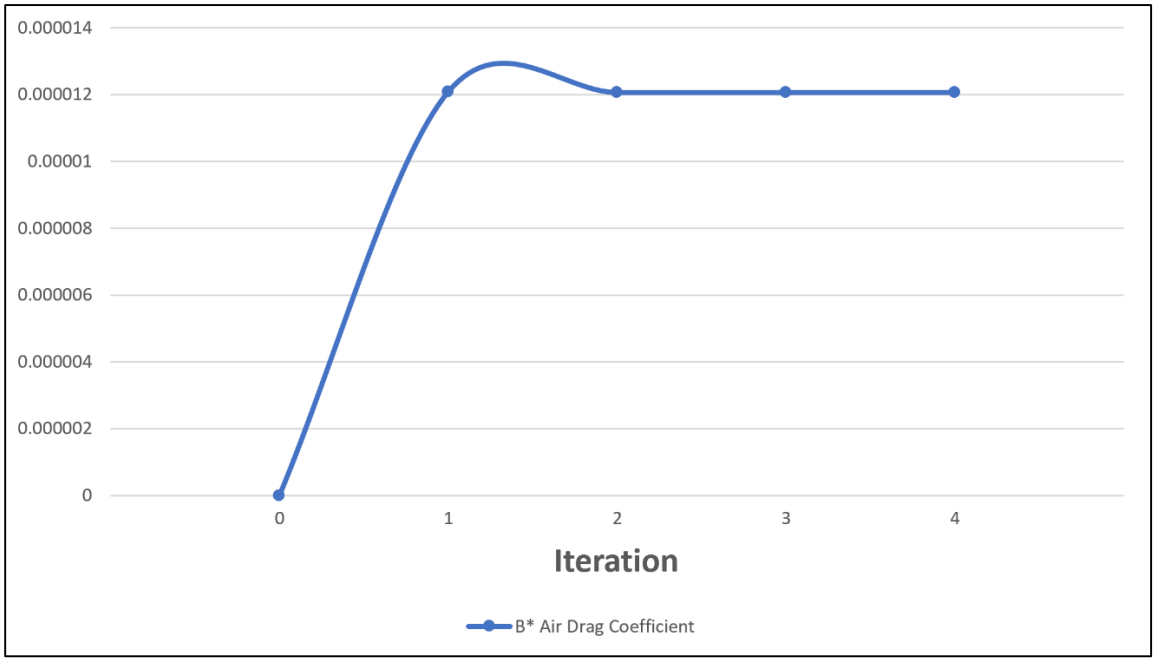


Figure 22. Air Drag Coefficients Converge, D&O = 20

Table 9 presents the final state vectors produced by SP on the Jetsons. The X- and Y- components of the position vector produced by SP on the Jetson TX1/TX2 agree with the truth data to the eighth decimal place, while the Z- component agrees with the truth data to the seventh decimal place. The X- and Y- components of the velocity vector agree with the truth data to the tenth decimal place, while Z- components agree to the eleventh decimal place. Finally, the air drag coefficients agree to the eleventh decimal place.

Table 9. Converged State Vectors Produced by SP

	Truth Data	Jetson TX1/TX2
Position X:	-3.71202483895854e+3	-3.71202483895187e+3
Position Y:	-5.86571445840261e+3	-5.86571445840703e+3
Position Z:	-2.16592610235187e-1	-2.16592605867213e-1
Velocity X:	5.63172597015199e+0	5.63172597015723e+0
Velocity Y:	-3.56203592928780e+0	-3.56203592927947e+0
Velocity Z:	3.61614435881406e+0	3.61614435881398e+0
*B** (Air Drag):	1.206715481898e-5	1.20671556188568e-5

The difference between the converged state vector produced by the Jetson TX1/TX2 and the truth data produced by the Windows version is presented in Table 10. The accuracy of the Jetson TX1/TX2 solution is also given as a percentage of the truth data, with the largest deviation from a perfect solution being the Z- component of the position vector.

Table 10. Accuracy of Jetson TX1/TX2 Converged State Vector

	Difference	%
Position X:	-6.66977939545177e-9	99.9999999998203
Position Y:	4.42014425061643e-9	100.000000000075
Position Z:	-4.36797401026645e-9	99.9999979833227
Velocity X:	-5.24025267623074e-12	100.000000000093
Velocity Y:	-8.32978130915762e-12	99.9999999997661
Velocity Z:	7.99360577730113e-14	99.9999999999978
*B** (Air Drag):	-7.99876800265751e-13	100.000006628545

These results show that the initial serial version of SP implemented on the Jetson TX1/TX2 converged to the correct solution. Once this task was completed, the two hypotheses could be tested.

4.2 Optimizing the Serial Version of Special Perturbations

This section presents the results of applying different combinations of compiler flags to the initial serial version of SP on the Jetson TX1 and TX2. Only the flags that reduced the runtime of SP by at least 1% were included in the final version. Table 3 in Section 2.3 was used as the starting point, with the "-nolib-inline" flag deleted and the "-O2" added to the list. Once the optimal combination of compiler flags was determined, the runtimes of the optimized serial version of SP were compared to the runtimes of the initial serial version to test the first hypothesis.

4.2.1 Determining the Optimal Combination of Compiler Flags

Since the impact of the results from this test was the same for the TX1 and TX2, only the results from the TX1 are shown in Figure 23. The red line is the runtime of the initial serial version of SP, with the "-O2" and "-O3" versions shown in green. For both the Jetson TX1 and TX2, only the "-O2" and "-O3" compiler flags resulted in a significant reduction in the runtime of SP.

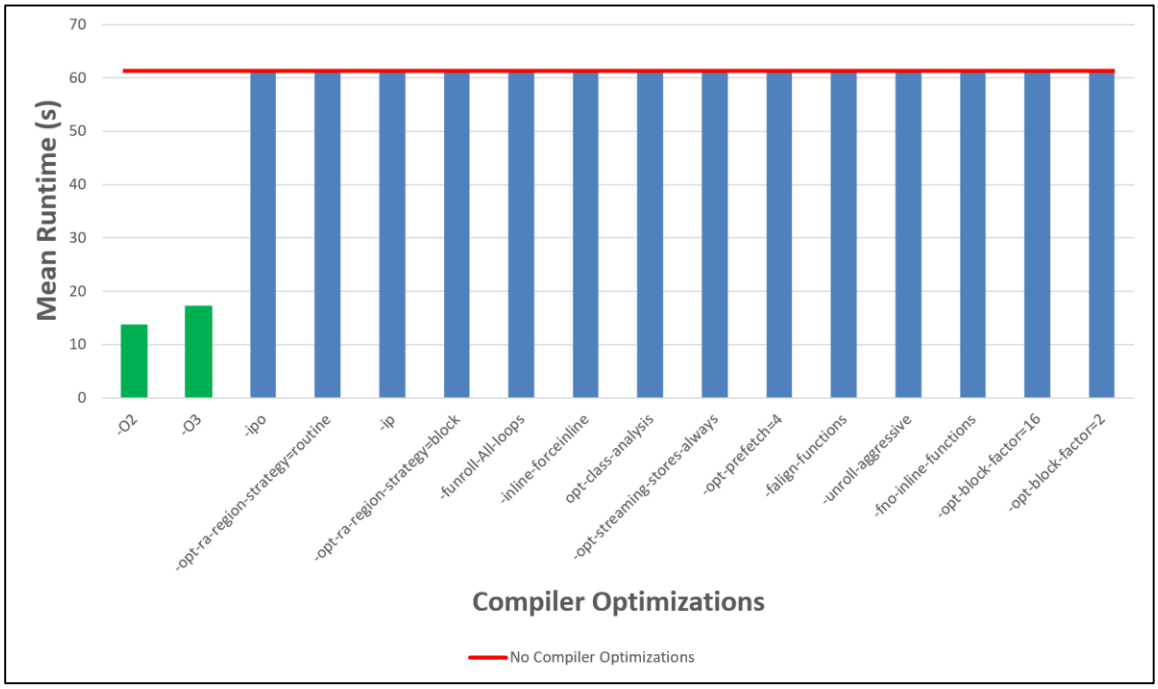


Figure 23. Compiler Optimizations Applied to SP on Jetson TX1, D&O = 20

Table 11 shows the results from using the compiler flags recommended by the CERN openlab study on both the Jetson TX1 and TX2. Applying the "-O2" flag reduced the speed by approximately 78% on both Jetsons, while the "-O3" flag only reduced the speed by 72%.

Table 11. Compiler Optimizations Applied to SP on Jetson TX1/TX2

Compiler Flag	Mean Runtime on TX1 (s)	Speedup on TX1 (%)	Mean Runtime on TX2 (s)	Speedup on TX2 (%)
None	61.32	--	53.73	--
-O2	13.70	77.66	11.43	78.73
-O3	17.13	72.06	15.04	72.01
-ipo	61.35	-0.06	53.78	-0.08
-opt-ra-region-strategy=routine	61.35	-0.04	53.36	0.70
-ip	61.31	0.01	53.82	-0.15
-opt-ra-region-strategy=block	61.36	-0.07	53.73	0.01
-funroll-all-loops	61.41	-0.15	53.67	0.12
-inline-forceinline	61.35	-0.05	53.63	0.19
-opt-class-analysis	61.30	0.03	53.92	-0.34
-opt-streaming-stores-always	61.32	-0.01	53.80	-0.13
-opt-prefetch=4	61.35	-0.06	53.82	-0.16
-falign-functions	61.30	0.03	53.58	0.28
-unroll-aggressive	61.35	-0.05	53.87	-0.26
-fno-inline-functions	61.35	-0.05	53.73	-0.01
-opt-block-factor=16	61.34	-0.03	53.57	0.31
-opt-block-factor=2	61.39	-0.11	53.58	0.29

The “-O3” flag consists of all the optimizations of the “-O2” flag plus 14 others (Section 2.3, Table 4). This inferred that one or more of the additional optimizations included in the “-O3” flag was negatively affecting the performance of the application. For this reason, another similar test was run in which each optimization of the “-O3” flag was individually paired with the “-O2” flag and compared to the “-O2”-optimized version to determine its effects on runtime. These results were slightly different for the Jetson TX1 and TX2; therefore, both sets of results are presented.

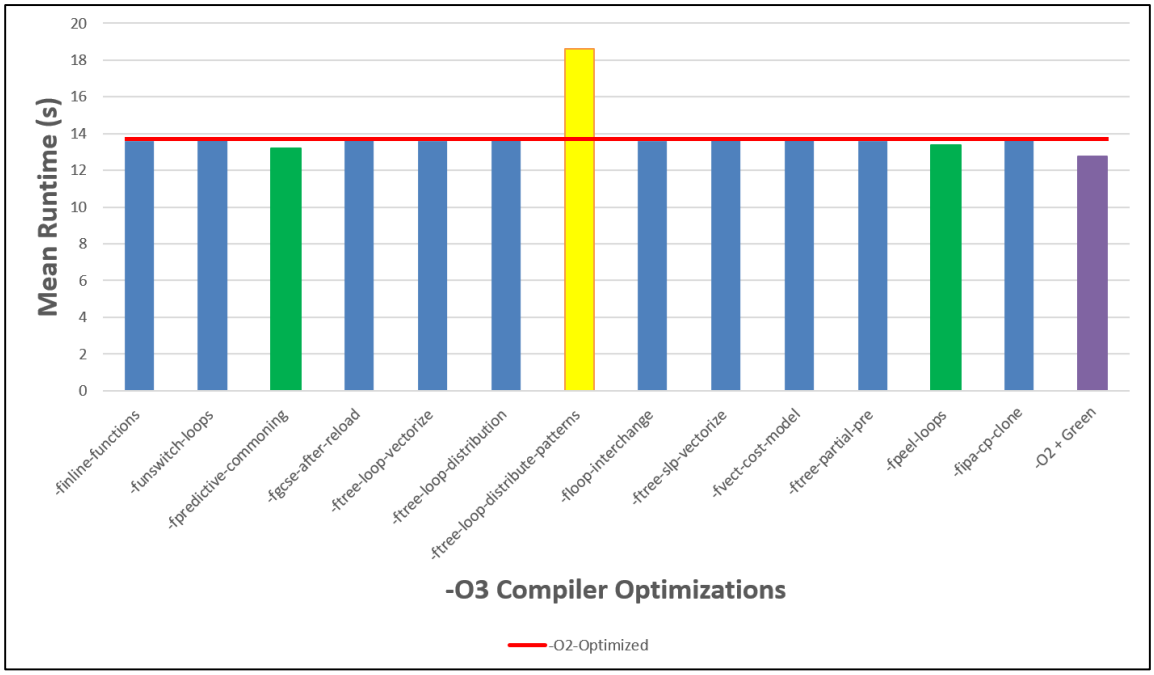


Figure 24. “-O2/O3” Optimizations Applied to SP on Jetson TX1, D&O = 20

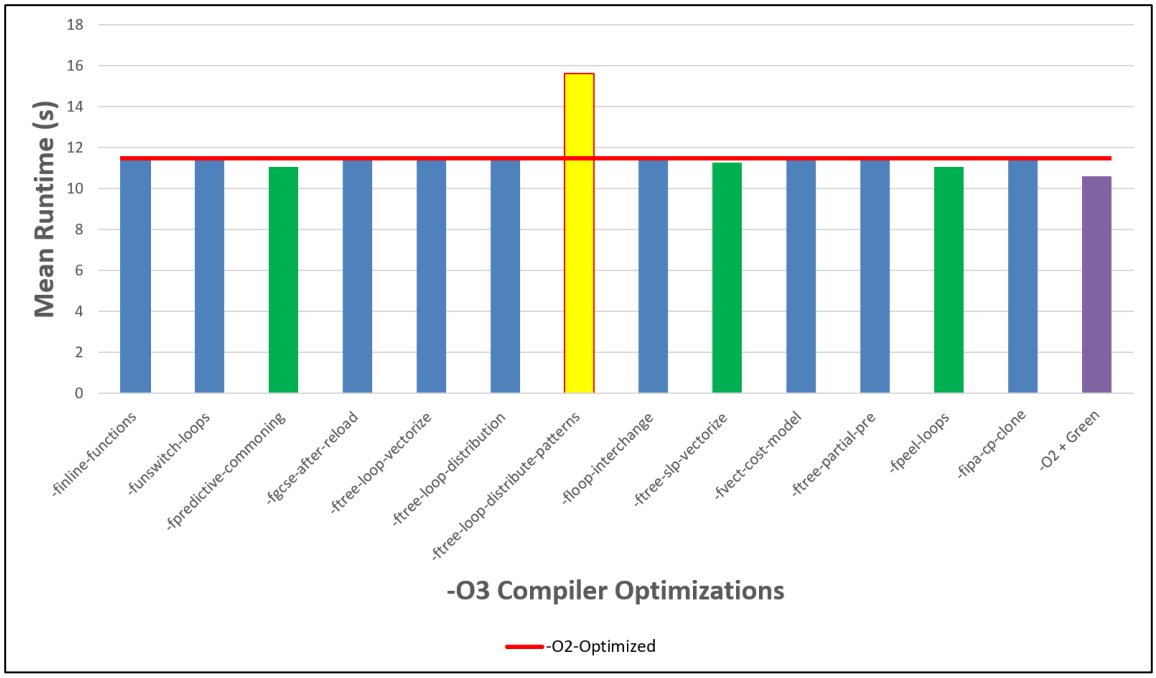


Figure 25. "-O2/O3" Optimizations Applied to SP on Jetson TX2, D&O = 20

Table 12. "-O2/O3" Optimizations Applied to SP on Jetson TX1/TX2, D&O = 20

Compiler Flag	Mean Runtime on TX1 (s)	Speedup on TX1 (%)	Mean Runtime on TX2 (s)	Speedup on TX2 (%)
-O2	13.70	--	11.48	--
-finline-functions	13.60	0.70	11.38	0.82
-funswitch-loops	13.61	0.64	11.42	0.48
-fpredictive-commoning	13.18	3.79	11.06	3.69
-fgcse-after-reload	13.67	0.25	11.45	0.25
-ftree-loop-vectorize	13.58	0.88	11.44	0.35
-ftree-loop-distribution	13.70	0.05	11.48	-0.01

-ftree-loop-distribute-patterns	18.61	-35.87	15.63	-36.12
-floop-interchange	13.60	0.76	11.48	-0.01
-ftree-slp-vectorize	13.69	0.09	11.36	1.00
-fvect-cost-model	13.68	0.12	11.47	0.04
-ftree-partial-pre	13.58	0.88	11.55	-0.61
-fpeel-loops	13.35	2.55	11.08	3.51
-fipa-cp-clone	13.67	0.24	11.48	0.00
-O2 + Green	12.74	7.00	10.61	7.55

For both Jetsons, the compiler flag "-free-loop-distribute-patterns" negatively affected the runtime of SP. Contrarily, the "-fpredictive-commoning" and "-fpeel-loops" flags positively affected performance on both computers. On the Jetson TX2, the "-ftree-slp-vectorize" optimization reduced the runtime of SP by just over 1%. None of the other compiler flags had any significant effect on either machine.

On the Jetson TX1, the "-fpredictive-commoning" and "-fpeel-loops" optimizations were combined with the "-O2" flag to form the "-O2 + Green" simulation shown in Figure 24, which reduced the runtime by 7% compared to the "-O2"-optimized version. On the TX2, the "-ftree-slp-vectorize" optimization was included in the "-O2 + Green" simulation. This reduced the runtime by 7.55% compared to the "-O2"-optimized version. These combinations of compiler optimizations formed the most efficient implementations of SP on each machine.

4.2.2 Testing the First Hypothesis

Once the optimal combination of compiler flags was applied, the first hypothesis could be tested to determine if the runtimes of the optimized serial version were less than those of the initial serial version. Figure 26 shows the resulting runtimes of the optimized serial version compared to the initial serial version for degrees and orders 10-50. At degree and order of 10, the optimized serial version converged in approximately five seconds on both machines, compared to the initial serial version which converged in 22-26 seconds. At degree and order of 50, the optimized serial version converged in 48-57 seconds, a full 200 seconds faster than the initial serial version.

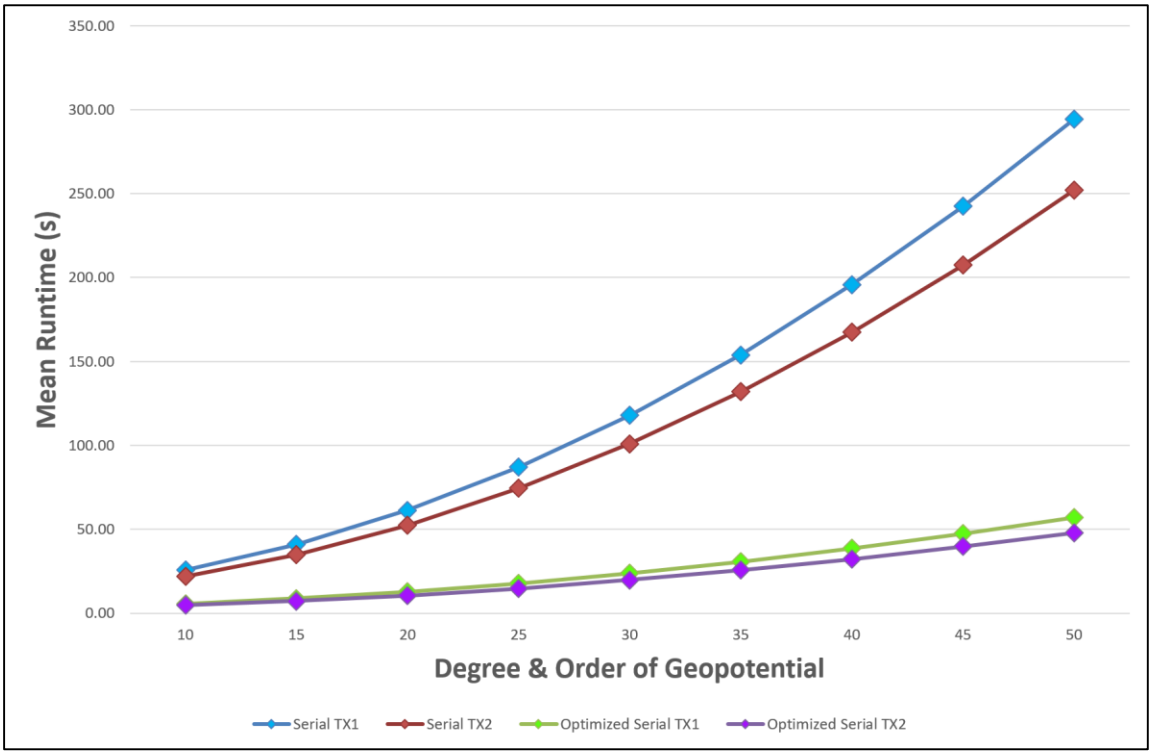


Figure 26. Initial Serial Version vs. Optimized Serial Version of SP

These results showed that the optimized serial version significantly outpaced the initial serial version for all degrees and orders tested. The combination of compiler optimizations used reduced the runtime of the initial serial version by an average of almost 80%. Thus, the evidence produced by these tests strongly supported the first hypothesis.

4.3 Applying APOD to Special Perturbations

This section analyzes the results of optimizing and deploying the parallel geopotential. The runtime of the parallel geopotential model was measured after the initial parallelization step was completed and measured again after each subsequent optimization step to ensure the optimization technique was beneficial. Because a large portion of the geopotential was still calculated on the Host, the compiler optimizations described in Section 4.2 were applied. Once all major optimization strategies were exhausted, the parallel geopotential model was integrated into the SP codebase and ran to ensure it converged to the correct solution.

4.3.1 Runtime Analysis of Optimizing the Parallel Geopotential Model

Once the results of initial parallel geopotential were verified for correctness, its runtime was measured before entering the optimization step. The mean runtime of the parallel geopotential was remeasured after each iteration of optimization to determine the performance gains of each optimization strategy. Figure 27 shows the mean runtime of the initial parallel geopotential model and the mean runtime after each iteration of the optimization step compared to the runtime of the serial version, all at degree and order of 50:

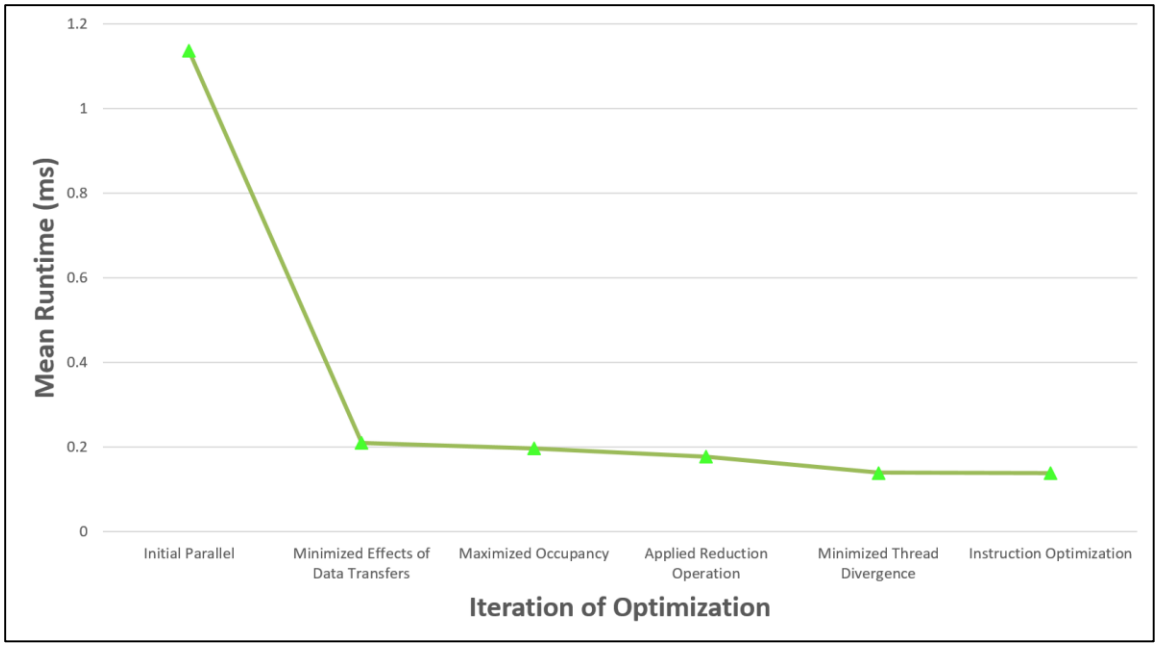


Figure 27. Runtime of Parallel Geopotential through Optimization Steps, D&O = 50

As described in Section 3.3.3.1, the first optimization technique applied to the parallel geopotential model was to minimize the impact of transferring data between the CPU and GPU. This optimization reduced the runtime of the parallel geopotential by more than 81%. The second optimization applied was to maximize the occupancy of the GPU. Although this only reduced the runtime by 6% over the previous version, it was essential to apply since the benefit of maximizing occupancy grows with the number of threads being run on the GPU. The third optimization strategy was to implement the reduction operation presented in Section 3.3.3.3. The initial implementation of the reduction operation used a **`for`** loop; however, the reduction operation was further optimized by unrolling the **`for`** loop. This reduced the runtime by 10% compared to the previous version. The next optimization strategy applied was to reduce thread divergence within the kernel. This reduced the runtime of the previous version by over 21%. The final optimization strategy was to implement instruction-level optimizations for all modulo

operations in the kernel. The benefit of this optimization was very small, which indicated the kernel was as optimized as possible in its current form. These five optimization strategies resulted in an overall runtime reduction of almost 88% over the initial parallel geopotential model. These results are presented in Table 13:

Table 13. Improvement of Parallel Geopotential through Optimization Steps, D&O = 50

Iteration	Description	Mean Runtime (ms)	% Improved
0	Initial Parallel Geopotential Model	1.1370	--
1	Minimized Impact of Data Transfers	0.2100	81.53
2	Maximized Occupancy	0.1977	5.87
3	Implemented Reduction Operation	0.1776	10.15
4	Minimized Thread Divergence	0.1395	21.43
5	Instruction Optimization	0.1391	0.30
		Total Runtime Improvement (%):	87.76

4.3.2 Verifying the Parallel Version of Special Perturbations

Once the parallel geopotential was optimized, it was integrated into the SP codebase and ran to ensure it converged to the correct solution. Table 14 contains the comparison of the truth data with the final state vector produced by SP using the parallel geopotential model. It shows that the position component of the state vector produced using the parallel geopotential model matches the truth data to the eighth decimal place. The velocity component matches the truth data to the eleventh decimal place, and the air drag coefficients match to the twelfth decimal place.

Table 14. Accuracy of State Vector Produced Using the Parallel Geopotential Model

	Difference	Percentage
Position X:	-3.57977114617825e-9	99.9999999999036
Position Y:	2.25008989218622e-9	100.000000000038
Position Z:	-2.39030001347729e-9	99.9999988964074
Velocity X:	-2.87947443666781e-12	100.000000000051
Velocity Y:	-4.38982183936787e-12	99.9999999998767
Velocity Z:	1.02140518265514e-14	99.9999999999997
*B** (Air Drag):	-9.99599700091786e-13	100.000008283640

4.3.3 Testing the Second Hypothesis

Once the parallel geopotential model was deployed into the SP codebase, the second hypothesis could be tested to determine if parallelizing the geopotential model would decrease the runtimes of the initial serial version of SP for any degree and order less than or equal to 50. Figure 28 shows the runtimes of the parallel version and the initial serial version using degrees and orders 10-50 on both Jetsons. At degree and order of 10, the initial serial versions converged in 22-26 seconds while the parallel versions took between 65-80 seconds. However, the parallel versions broke even with the initial serial versions at degree and order of 26 on the TX1 and 28 on the TX2. By degree and order of 50, the parallel version resulted in an average runtime reduction of $54 - 56\%$ over the initial serial version.

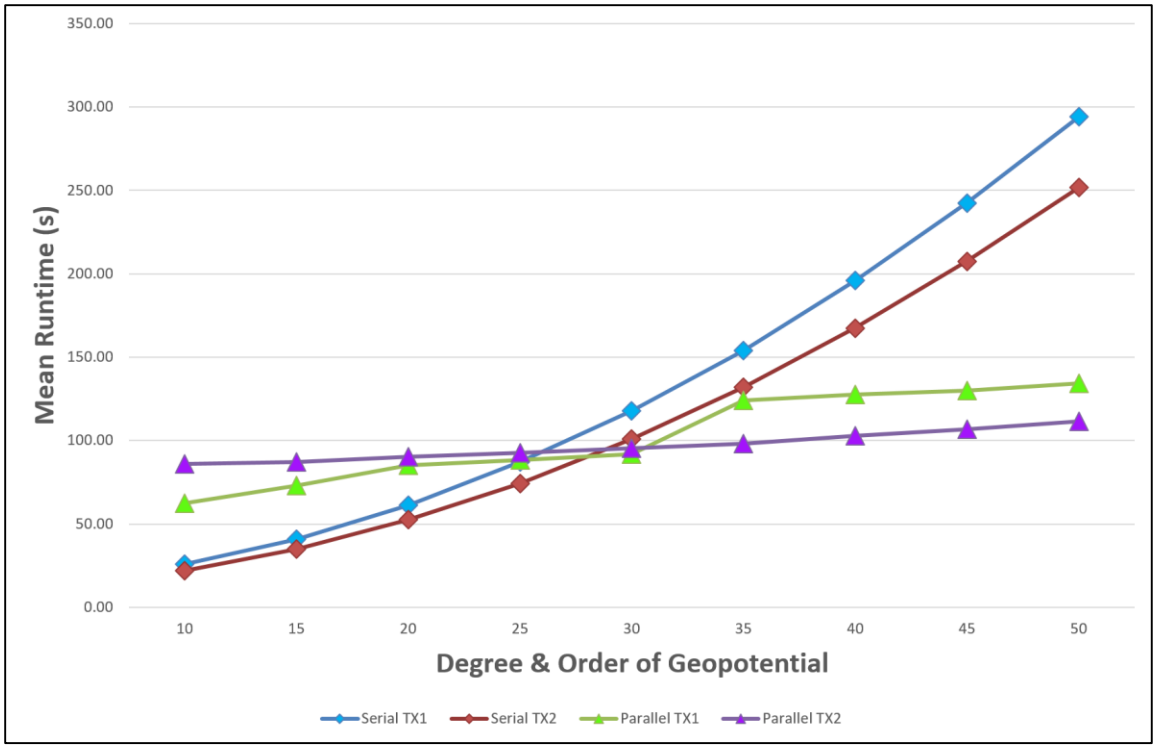


Figure 28. Initial Serial Version vs. Parallel Version of SP

The runtimes of the parallel version on the Jetson TX1 increase in increments instead of a steady incline like those on the TX2. The degrees and orders at which the runtimes on the TX1 increase correlate to the number of thread blocks being launched. When degree and order was set to 30, three thread blocks of 256 threads each were launched on the GPU. At degree and order of 35, four thread blocks were launched.

At degree and order of 10, the initial serial version was faster on both machines. However, the parallel version outpaced the initial serial versions for any degree and order greater than 26 on the TX1 and 28 on the TX2. Since the parallel version was faster than the initial serial version for degrees and orders less than 50, the results of this test strongly supported the second hypothesis.

4.4 Determining the Most Efficient Implementation

After the optimized serial version and the parallel version were implemented, the runtimes of the two versions were compared to the initial serial version of SP. The results of these tests concluded the best way to implement SP on both the Jetson TX1 and TX2 was the optimized serial version. This section presents these results and the accompanying analysis.

4.4.1 Special Perturbations on the Jetson TX1

Despite the parallel version being faster than the initial serial version for any degree and order higher than 26 on the TX1, the optimized serial version was the fastest overall. Figure 29 shows the comparison of the runtimes of the three versions of SP on the Jetson TX1. For all degrees and orders tested, the optimized serial version significantly outpaced the parallel version. The runtimes of the parallel version grew at a slower rate than those of the optimized serial version, meaning the parallel version would likely outpace the optimized serial version at higher degrees and orders. However, since the implementation of SP being used only uses degree and order of 50 or less, the optimized serial version remains the most efficient implementation.

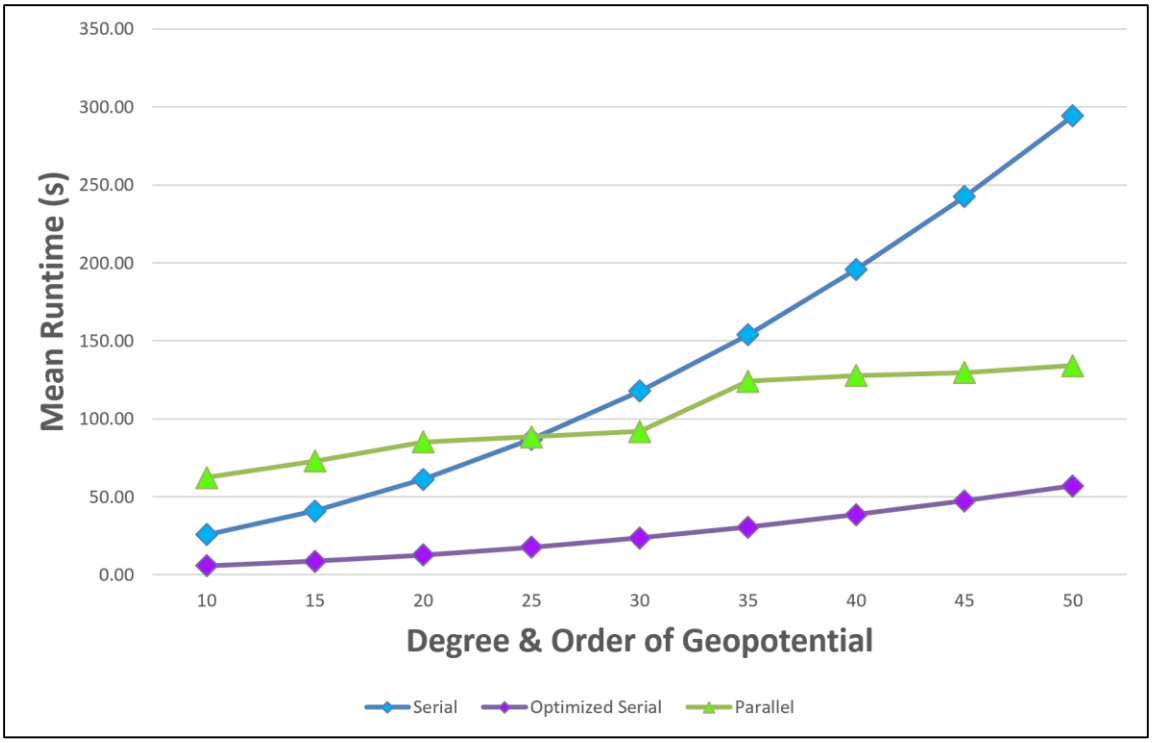


Figure 29. Initial Serial vs. Optimized Serial vs. Parallel Version of SP, Jetson TX1

These results are also presented in Table 15 for each degree and order tested. The percentage of speedup of the optimized serial and parallel versions is in relation to the initial serial version. At degree and order of 50, the optimized serial version is over 80% faster than the initial serial version, while the parallel version is only 54% faster, making the optimized serial version the most efficient way to implement SP on the Jetson TX1.

Table 15. Initial Serial vs. Optimized Serial vs. Parallel Version of SP, Jetson TX1

D&O of Geopotential	Initial Serial Version (s)	Optimized Serial Version (s)	Optimized Serial Speedup (%)	Parallel Version (s)	Parallel Speedup (%)
10	25.76	5.80	77.50	62.37	-142.11
15	40.92	8.77	78.56	73.07	-78.57
20	61.32	12.76	79.18	85.19	-38.93
25	87.00	17.74	79.61	88.45	-1.67
30	117.81	23.72	79.87	92.03	21.88
35	153.95	30.64	80.09	124.15	19.36
40	195.86	38.61	80.29	127.75	34.77
45	242.54	47.45	80.43	129.84	46.47
50	294.36	57.11	80.60	134.19	54.41

4.4.2 Special Perturbations on the Jetson TX2

As on the Jetson TX1, the optimized serial version was the most efficient way to implement SP on the TX2. The parallel version broke even with the initial serial version at degree and order of 28. However, the runtimes of the optimized serial version were much faster, as seen in Figure 30:

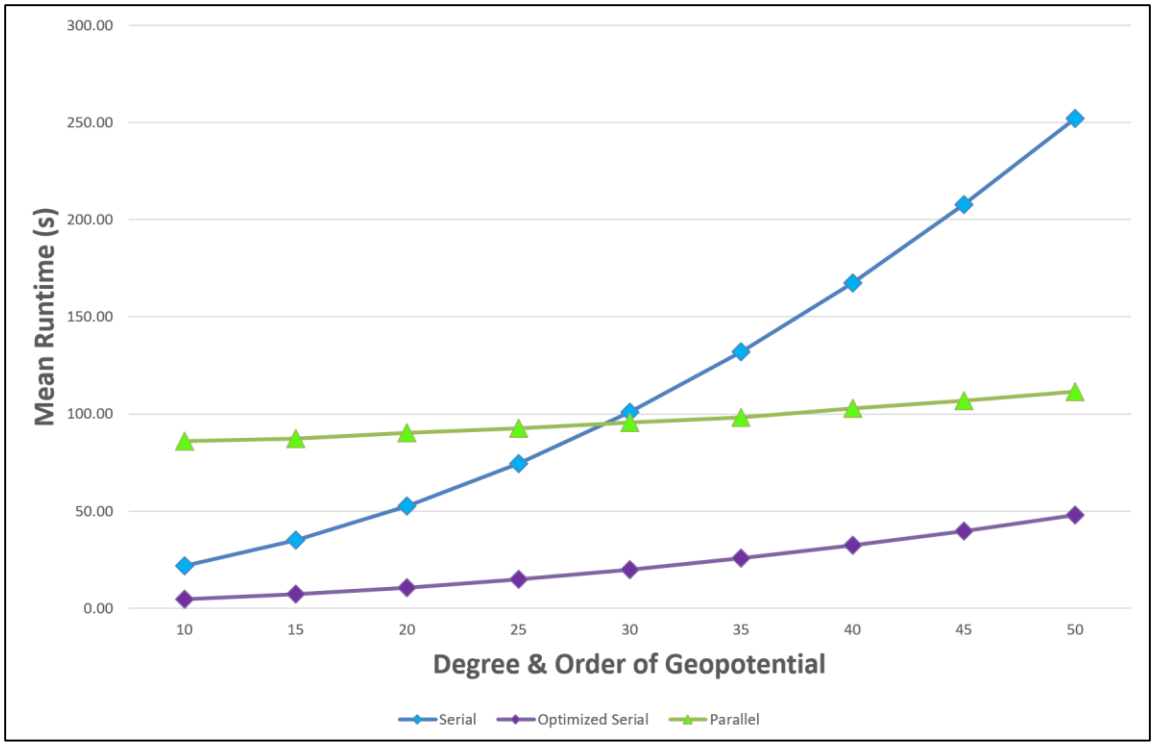


Figure 30. Initial Serial vs. Optimized Serial vs. Parallel Version of SP, Jetson TX2

At degree and order of 50, the optimized serial version is 81% faster than the initial serial version. The parallel version is only 56% faster at this degree and order. Thus, the optimized serial version was the most efficient way to implement SP on the Jetson TX2. This data is shown in Table 16:

Table 16. Initial Serial vs. Optimized Serial vs. Parallel Version of SP, Jetson TX2

D&O of Geopotential	Initial Serial Version (s)	Optimized Serial Version (s)	Optimized Serial Speedup (%)	Parallel Version (s)	Parallel Speedup (%)
10	21.93	4.75	78.33	85.94	-291.92
15	35.00	7.28	79.19	87.21	-149.17
20	52.48	10.62	79.76	90.34	-72.15
25	74.39	14.80	80.11	92.72	-24.65
30	100.98	19.80	80.39	95.49	5.44
35	131.92	25.66	80.55	98.17	25.59
40	167.52	32.30	80.72	102.81	38.62
45	207.58	39.74	80.85	106.71	48.59
50	251.97	47.87	81.00	111.50	55.75

4.4.3 Special Perturbations on the Jetson TX1 vs. the Jetson TX2

The extent to which SP performs better on the Jetson TX2 compared to the TX1 was also recorded for this research. The runtimes of SP on each machine used in Sections 4.4.1 and 4.4.2 were compared to determine how much faster the Jetson TX2's CPU and GPU performed compared to the TX1's. Figure 31 compares the runtimes of the optimized serial version on the TX2 and TX1.

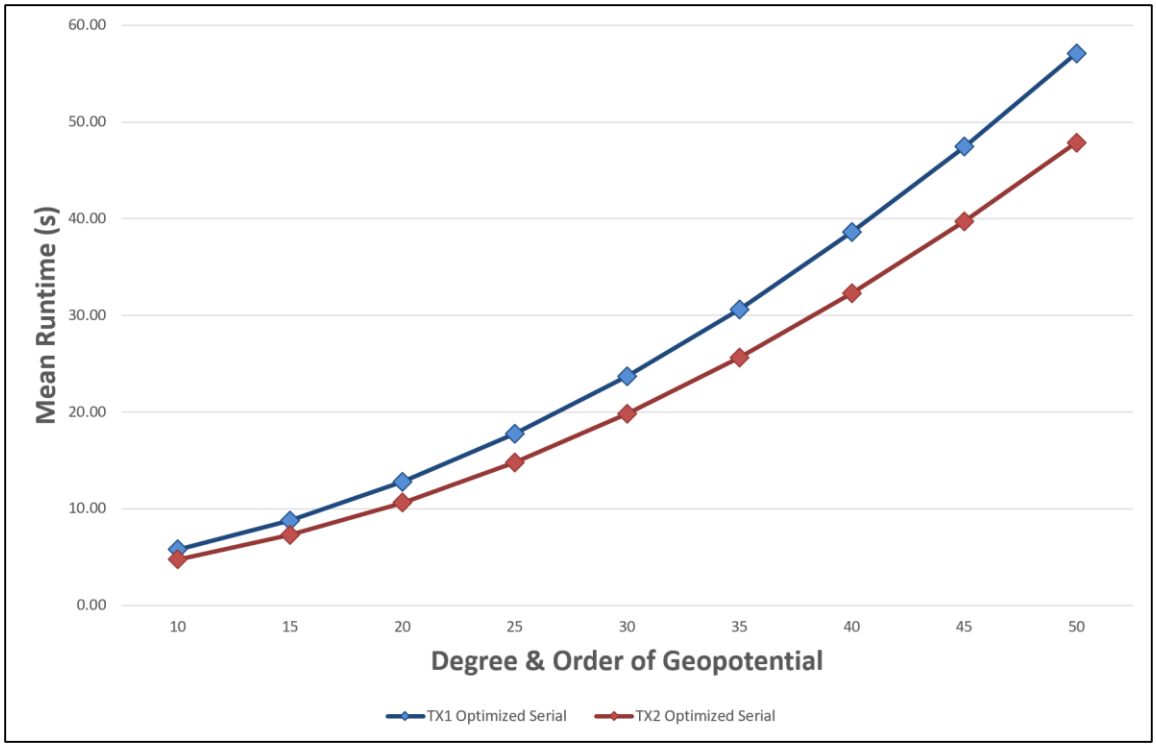


Figure 31. Optimized Serial Version of SP on Jetson TX1 vs. TX2

Table 17 compares the performance of the optimized serial version on each of the Jetsons' CPUs. At degree and order of 50, the Jetson TX2 outpaced the TX1 by just under 10 seconds. This equates to a speedup of 16%.

Table 17. Optimized Serial Version of SP on Jetson TX1 vs. TX2

D&O of Geopotential	Optimized Serial Version on TX1	Optimized Serial Version on TX2	Speedup of TX2 (%)
10	5.80	4.75	18.02
15	8.77	7.28	16.97
20	12.76	10.62	16.78
25	17.74	14.80	16.59
30	23.72	19.80	16.51

35	30.64	25.66	16.27
40	38.61	32.30	16.35
45	47.45	39.74	16.25
50	57.11	47.87	16.19

Next, the runtimes of the parallel versions on each machine were compared. Figure 32 and Table 18 below show the performance of the parallel version of SP on the Jetson TX2 compared to the TX1. At degree and order of 50, the Jetson TX2 converged almost 17% faster than the TX1.

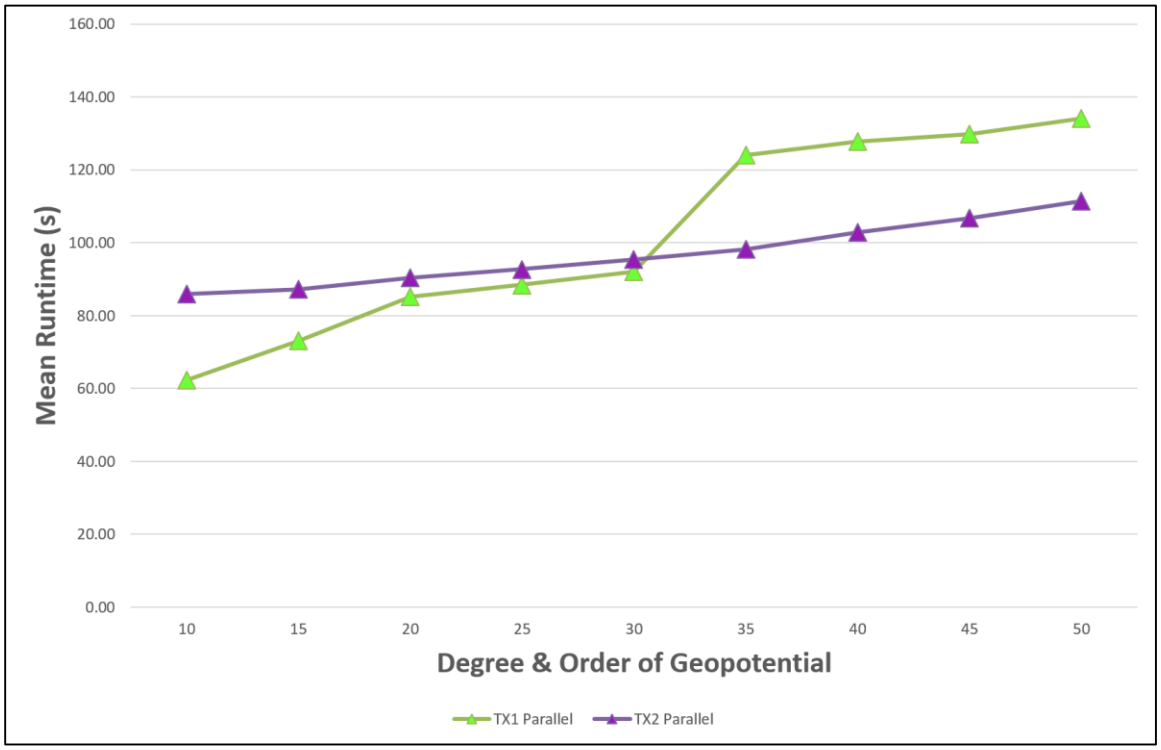


Figure 32. Parallel Version of SP on Jetson TX1 vs. TX2

Table 18. Parallel Version of SP on Jetson TX1 vs. TX2

D&O of Geopotential	Parallel Version on TX1	Parallel Version on TX2	Speedup of TX2 (%)
10	62.37	85.94	-37.80
15	73.07	87.21	-19.35
20	85.19	90.34	-6.04
25	88.45	92.72	-4.82
30	92.03	95.49	-3.76
35	124.15	98.17	20.93
40	127.75	102.81	19.52
45	129.84	106.71	17.81
50	134.19	111.50	16.91

5. Conclusions and Recommendations

This section presents the conclusions associated with this research. The results of two hypotheses are summarized and the significance of this research is discussed. Recommendations for future work are also presented.

5.1 Research Summary and Conclusions

The ultimate goal of the research conducted in this thesis was to determine the most efficient way to implement SP on the Jetson TX1 and TX2. Before that determination could be made, the implementation of SP developed for a Windows machine had to be ported over to the Linux operating system used by the Jetson TX series of computers and the two hypotheses had to be tested. The results of the first hypothesis determined the optimal combination of compiler flags to apply to SP that would reduce SP's runtime. The results of the second hypothesis showed that the Jetsons' GPUs could be used to reduce the runtime of SP. The results of the two hypotheses were then used to determine the most efficient way to implement SP on the Jetson TX1 and TX2.

Before the two hypotheses could be tested, the SP software had to be reconfigured to run on the Jetson TX1 and TX2. Through reorganizing the package diagrams, the SP codebase was able to be implemented on the Jetsons such that it converged to the same solution as the original Windows version.

Once the SP software was reconfigured to run on the Jetson TX1/TX2, the first hypothesis was able to be tested. The study conducted by CERN openlab provided a guide to which compiler flags should be included. Although the majority of the compiler optimizations suggested by the study had negligible affects, the "-O2" and "-O3" compiler

flags significantly reduced the runtime of SP. Since the benefit of using the "-O3" compiler flag was less than that of using the "-O2" compiler flag, an additional test was conducted for both the Jetson TX1 and TX2 to determine if certain individual compiler optimizations included in the "-O3" compiler flag would reduce the runtimes further. The "-fpredictive-commoning" and "-fpeel-loops" compiler optimizations, when combined with "-O2" compiler flag, resulted in the fastest runtimes on the Jetson TX1. On the TX2, the "-ftree-slp-vectorize" compiler flag was added to the combination of compiler flags to produce the optimal solution. The runtimes of the optimized serial versions were significantly less than the initial serial version, which strongly supported the first hypothesis.

The APOD software development cycle was then applied to the initial serial version of SP to test the second hypothesis of determining whether parallel computing using the Jetsons' GPUs could reduce the runtimes of SP. The 'Assess' step of the APOD cycle revealed that the majority of SP's runtime was calculating the geopotential. For this reason, the 'Parallel' step focused on this portion of the code. An initial parallel version of Pines Method for computing the geopotential was developed and verified. The 'Optimize' step of APOD was then applied to the initial parallel version. The parallel geopotential model underwent five optimization iterations, resulting in an 88% reduction in runtime compared to the initial parallel version. Once all major optimization strategies were applied, the parallel geopotential model was deployed into the SP codebase and verified to confirm it converged to the correct solution. The version of SP using the parallel geopotential was faster than the initial serial version at any degree and order higher than 26 on the Jetson TX1 and 28 on the TX2, which strongly supported the second hypothesis.

The results of these hypotheses were used to determine the most efficient way to implement SP on the Jetson TX1 and TX2. The optimized serial version and the parallel version of SP were compared to the initial serial version to determine which resulted in the fastest runtimes. Although the parallel version was faster than the initial serial version at higher orders of the geopotential, the optimized serial version resulted in the fastest runtimes by far. Therefore, the optimized serial version is the best version to use when implementing SP on the Jetson TX1 and TX2.

5.2 Research Significance

The results of the research conducted in this thesis have shown that SP has become a viable option for performing OD onboard a spacecraft. When implemented on the Jetson TX series of computers, SP can converge in as little as 5 seconds when degree and order of 10 is used for the geopotential. When degree and order of 50 is used, SP can converge in 47 seconds. This is significantly faster than the original Windows version on the SBC. This will allow the SOS payload to achieve much higher accuracy than that produced by using SGP4.

Knowing the precise location of a spacecraft at future epochs is paramount in avoiding conjunctions. This is especially important due to the increasing congestion of the space domain. Higher accuracy OD performed onboard the spacecraft via SP will enable SSA assets to maintain tight control over where spacecraft are located and reduce the likelihood of an unintended conjunction.

5.3 Recommendations for Future Work

The research presented in this thesis could be furthered in two primary ways. First and foremost, additional test cases could be developed and applied to SP on the Jetson TX1/TX2. Secondly, a different approach to parallelizing SP could be investigated to determine if the runtime could be further reduced.

One of the limitations accepted for this research was the use of a single test case. Although this test case was sufficient for preliminary experimentation, more rigorous testing is required to ensure the software would operate properly under all possible operating conditions. Once additional test cases are developed, the Systems Tool Kit (STK) could be used to simulate the spacecraft in a real-world environment. This experimentation should be completed before SP is integrated into the SOS payload.

The second recommendation for future work is to investigate a different approach to using parallel processing to run SP. SP uses numerical integration to estimate a spacecraft's position and velocity over small, consecutive integrals. Since these integrals are consecutive, calculating them in parallel cannot be accomplished in a straightforward way. However, there is a method for doing this that involves doing a first 'rough' pass with large intervals. This initial rough pass is shown in Figure 33:

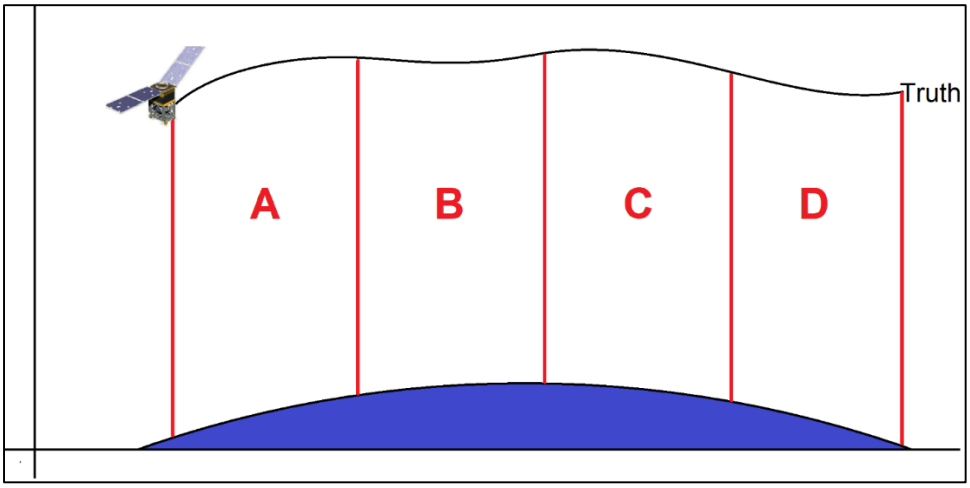


Figure 33. SP Using Large Intervals

A second pass of SP can then be applied that divides intervals A-D into smaller intervals, and computes them in parallel. This is shown in Figure 34:

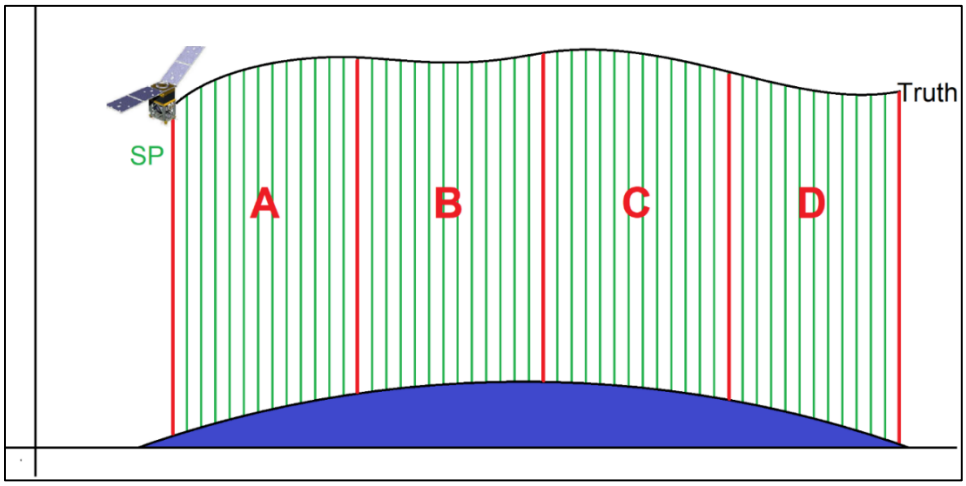


Figure 34. SP Performed on Large Intervals in Parallel

This approach is closer to typical multithreaded application instead of a massively parallel GPU application, meaning that the Jetson TX1/TX2's multi-core CPUs would likely be more applicable than the 256-core GPUs. However, this approach has the potential to further reduce the runtimes of SP running on the Jetson TX1 and TX2.

Appendix

Compute Capability Specifications		
Technical Specification	**5.3**	**6.2**
Maximum number of resident grids per device	16	16
Maximum dimensionality of grid of thread blocks	3	3
Maximum x-dimension of a grid of thread blocks	$2^{31} - 1$	$2^{31} - 1$
Maximum y- or z-dimension of a grid of thread blocks	65535	65535
Maximum dimensionality of thread block	3	3
Maximum x- or y-dimension of a block	1024	1024
Maximum z-dimension of a block	64	64
Maximum number of threads per block	1024	1024
Maximum number of resident blocks per multiprocessor	32	32
Maximum number of resident warps per multiprocessor	64	64
Maximum number of resident threads per multiprocessor	2048	2048
Number of 32-bit registers per multiprocessor	64 K	64 K
Maximum number of 32-bit registers per thread block	32 K	32 K
Maximum number of 32-bit registers per thread	255	255
Maximum amount of shared memory per multiprocessor	64 KB	64 KB

Maximum amount of shared memory per thread block	48 KB	48 KB
Number of shared memory banks	32	32
Amount of local memory per thread	512 KB	512 KB
Constant memory size	64 KB	64 KB
Cache working set per multiprocessor for constant memory	8 KB	8 KB
Maximum number of instructions per kernel	512 M	512 M

(Table derived from NVIDIA (B), 2017)

Bibliography

1. Air Force Space Command. “AFSPC Astrodynamic Standard Software.” 2012.

2. Arora, N., Russell, R. P. “Fast, Efficient and Adaptive Interpolation of the Geopotential.” *Journal of Guidance, Control, and Dynamics*, vol.39, no.1, pp. 128-143, 2016.

3. Asanovic, K., et al. “The Landscape of Parallel Computing Research: A View from Berkeley.” Electrical Engineering and Computer Science Department, University of California, Berkeley, 2006.

4. Baird, Col M. A., “Maintaining Space Situational Awareness and Taking it to the Next Level.” *Air & Space Power Journal*, Space Focus, pp. 50–72, 2013.

5. Bastow, L. B., “Modeling the Impact of the Payload Alert Communications System (PACS) on the Accuracy of Conjunction Analysis.” Ph.D. dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, 2013.

6. Botezatu, M. “A Study on Compiler Flags and Performance Events.” *Conseil Européen pour la Recherche Nucléaire* (CERN) openlab, 2012.

7. Boyer, M. *LAVA Lab CUDA Support: Memory Management Overhead*. School of Engineering and Applied Sciences, University of Virginia. www.cs.virginia.edu/~mwb7w/cuda_support. 2013. Accessed Nov 2017.

8. Casotto, S., et al. “Evaluation of Methods for Spherical Harmonic Synthesis of the Gravitational Potential and its Gradients.” *Advances in Space Research*, vol. 40, pp. 69-75, 2007.

9. Colliot, T., et al. “Space Risks: A New Generation of Challenges.” Allianz Global Corporate and Specialty (AGSC), 2012.

10. Crommelin, A. C. D., Cowell, P. H., “Investigation of the Motion of Halley’s Comet from 1759 to 1910.” *Greenwich Observations in Astronomy, Magnetism, and Meteorology made at the Royal Observatory*, vol. 71, series 2, pp. 1-84, 1911.

11. CSRA, “Space Object Self-Tracker Experiment ConOps.” 2014.

12. Flamos, S. M., “Space Object Self-Tracker On-board Orbit Determination Analysis.” M.S. thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, 2016.

13. Frank, M. P., “The Physical Limits of Computing.” *Computing in Science and Engineering*, vol. 4, issue 3, pp. 16-26, 2002.

14. Free Software Foundation, Inc. *Using the GNU Compiler Collection (GCC): Options that Control Optimization.* www.gcc.gnu.org/onlinedocs/gcc/index.html#SEC_Contents. 2018. Accessed Dec 2017.

15. Hack, J. J. "On the Promise of General-Purpose Parallel Computing." *Parallel Computing*, vol. 10, issue 3, pp. 261-275, 1989.

16. Lemoine, F. G., et al. "The Development of the Joint NASA GSFC and the National Imagery and Mapping Agency (NIMA) Geopotential Model EGM96." 1998.

17. Lemoine, F.G. *EGM96: The NASA GSFC and NIMA Joint Geopotential Model.* www.cddis.nasa.gov/926/egm96/contents.html. 2005. Accessed Jul 2017.

18. McCall, G. H., Darrah, J. H., "Space Situational Awareness: Difficult, Expensive - and Necessary," *Air & Space Power Journal*, Senior Leadership Perspective, pp. 6–16, 2014.

19. Moore, G. "Cramming More Components onto Integrated Circuits." *Electronics Magazine*, pp. 114-117, 19 Apr 1965.

20. NVIDIA Corporation. "Data Sheet: NVIDIA Jetson TX1 System-on-Module." *Version 1.1*, docs.nvidia.com, 2016.

21. NVIDIA Corporation (A). "Data Sheet [Preliminary]: NVIDIA Jetson TX2 System-on-Module." *Version 1.0*, docs.nvidia.com, 2017.

22. NVIDIA Corporation (B). "CUDA C Programming Guide." *Version 8.0*, docs.nvidia.com, 2017.

23. NVIDIA Corporation (C). "CUDA C Best Practices Guide." *Version 8.0*, docs.nvidia.com, 2017.

24. Oltrogge, D. L., et al. "Parametric Characterization of SGP4 Theory and TLE Positional Accuracy." *AMOS SSA Conference*, Maui HI, 2014.

25. Otterness, N., et al. "An Evaluation of the NVIDIA TX1 for Supporting Real-time Computer-Vision Workloads." *General Motors Research*, 2017.

26. Pelaez, J., Hedo, J., "A Special Perturbation Method in Orbital Dynamics." *Celestial Mechanics and Dynamical Astronomy*, vol.97, no.2, pp. 131–150, 2007.

27. Perry, D. A., “Space Object Self-Tracker Hardware Analysis and Environmental Testing.” Ph.D. dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, 2014.

28. Pines, S. “Uniform Representation of the Gravitational Potential and its Derivatives.” *The American Institute of Aeronautics and Astronautics,* vol. 11, no. 11, pp. 1508-11, 1973.

29. Russell, R. P., Arora, N. “Global Point Mascon Models for Simple, Accurate and Parallel Geopotential Computation.” *Journal of Guidance, Control, and Dynamics*, vol. 35, no. 5, pp. 1568-1581, 2012.

30. Sandbox Science. *1.0 Labeling Earth*. www.sandboxscience.com. Accessed Jun 2017.

31. Technologic Systems. “TS-7260 Hardware Manual.” 2010.

32. Vallado, D. A., et al. “Revisiting Spacetrack Report #3.” *AIAA Astrodynamics Specialist Conference,* 2006.

33. Vetter, J. R. “Fifty Years of Orbit Determination: Development of Modern Astrodynamics Methods.” *Johns Hopkins APL Technical Digest,* vol.27, no.3, pp. 239-52, 2007.

34. Wan Aziz, W.A., et al. “Evaluation of the EGM96 Model of the Geopotential in Peninsular Malaysia.” Seminar Geoinformation, University Technology Malaysia, Kuala Lumpur, 1998.

35. Wiesel, W. E., *Modern Orbit Determination*. Beavercreek OH: Aphelion Press, 2003.

36. Williams, A. *C++ Concurrency in Action.* Shelter Island NY: Manning Publications Co., 2012.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
23-03-2018	Master's Thesis	May 2017 – March 2018

TITLE AND SUBTITLE

Special Perturbations on the Jetson TX1 and TX2 Computers

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Moore ,Tyler M., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/ENG)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-18-M-047

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

(no sponsor enter: Intentionally left blank)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRUBTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Simplified General Perturbations Number 4 (SGP4) has been the traditional algorithm for performing Orbit Determination (OD) onboard orbiting spacecraft. However, the recent rise of high-performance computers with low Size, Weight, and Power (SWAP) factors has provided the opportunity to use Special Perturbations (SP), a more accurate algorithm to perform onboard OD. This research evaluates the most efficient way to implement SP on NVIDIA's Jetson TX series of integrated Graphical Processing Units (GPUs). An initial serial version was implemented on the Jetson TX1 and TX2's Central Processing Units (CPUs). The runtimes of the initial version are the benchmark that the runtimes of the other versions were compared against. A second version of SP was implemented using compiler optimizations to increase the speed of the program. A third version was developed to utilize the Jetsons' 256-core GPU for parallel processing to reduce the runtimes of the program. Runtimes of the different versions were then analyzed to determine the most efficient way to implement SP on the Jetson TX series of computers.

15. SUBJECT TERMS

Orbit Determination, Special Perturbations, Space Object Self-Tracker, NVIDIA, CUDA

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Col. Dane F. Fuller, AFIT/ENY
U	U	U	UU	119	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-6565, ext 4679 (NOT DSN) (dane.fuller@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18