Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By THOMAS ANTONY

Entitled
RAPID INDIRECT TRAJECTORY OPTIMIZATION ON HIGHLY PARALLEL COMPUTING ARCHITECTURES

For the degree of Master of Science in Aeronautics and Astronautics

Is approved by the final examining committee:

MICHAEL J. GRANT

WILLIAM A. CROSSLEY

JAMES M. LONGUSKI

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

MICHAEL J. GRANT

Approved by Major Professor(s): _____

Approved by: WEINONG CHEN 11/24/2014

Head of the Department Graduate Program Date

RAPID INDIRECT TRAJECTORY OPTIMIZATION ON HIGHLY PARALLEL COMPUTING ARCHITECTURES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Thomas Antony

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Aeronautics & Astronautics

December 2014

Purdue University

West Lafayette, Indiana

To my parents, Antony and Susheela.

To my sister, Asha.

Your love and support make my dreams possible.

"For once you have tasted flight, you will forever walk the earth with your eyes turned skywards, for there you have been and there you will long to return."

– Leonardo Da Vinci

ACKNOWLEDGMENTS

First and foremost, I would not be where I am today without the constant encouragement and support from my family, which often goes unacknowledged. You have always urged me to follow my dreams, and this work is dedicated to you.

I would like to express my utmost gratitude to Professor Michael Grant, for believing in me and for all the time he dedicated for advising me and providing me with guidance. His deep understanding of fundamental concepts and openness to new ideas have greatly influenced my style of research. His support and encouragement have been invaluable in getting me to where I am today. I am very grateful for having the opportunity to work with you. I also express my sincere gratitude to Professors James Longuski and William Crossley for serving on my committee and reviewing my thesis. Your classes taught me the fundamentals that form the foundation of my research.

I would like to thank Nikhil Varma for his camaraderie throughout the ups and downs of my graduate studies at Purdue. I would like to thank Shreyas Subramanian, Kshitij Mall, Harish Saranathan, and everyone in my research group for their valuable insights on my work and lightening up the atmosphere at the office. I would also like to thank my significant other, Beth Kashon, for always being there for me.

I offer my gratitude to the faculty and staff at the School of Aeronautics and Astronautics for doing so much to provide us with an exceptional environment for research and education. I also acknowledge Sarag Saikia, who gave me the opportunity to collaborate on many interesting research problems.

Finally, I wish to thank the Charles Stark Draper Laboratory, who funded me through the University Research and Development Program, as well as Purdue University for financial support during my graduate studies. Their generosity has been instrumental in the completion of this degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SYMBOLS

A_{ref}	Reference area, m^2
C_D	Coefficient of drag
C_L	Coefficient of lift
H	Hamiltonian
h_{scale}	Scale height, m
I_N	Identity matrix of dimension NxN
J	Cost function or Jacobian matrix
$\frac{L}{D}_{max}$	Peak lift-to-drag ratio
m	Mass, kg
q	Dynamic pressure, N/m^2
R_E	Radius of the Earth, m
r	Radial position, m
r_n	Nose radius, m
t_f	Time of flight, s
V	Velocity, m/s
$\boldsymbol{x}$	State vector
α	Angle of attack, rad
β	Ballistic coefficient, kg/m^2
γ	Flight path angle, rad
$\boldsymbol{\epsilon}$	Residual error vector
θ	Longitude, rad
$\boldsymbol{\lambda}$	Costate vector
μ	Standard gravitational parameter, m^3/s^2
$\boldsymbol{\nu}_0$, $\boldsymbol{\nu}_f$	Lagrange multipliers for initial and terminal point constraints, respectively

$\boldsymbol{\pi}$	Lagrange multipliers for interior point constraints
ρ	Density of the atmosphere, kg/m^3
ρ_0	Atmospheric density at sea-level, kg/m^3
σ	Bank angle, rad
Φ	State transition matrix
Φ_k	State transition matrix for trajectory segment k
ϕ	Latitude, rad
ψ	Azimuth, rad
ω	Rotation rate of the Earth, rad/s

ABBREVIATIONS

BLAS	Basic Linear Algebra Subroutines
BVP	Boundary Value Problem
CPU	Central Processing Unit
CUBIN	CUDA Binary
CUBLAS	CUDA Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
DOF	Degree Of Freedom
GB/s	Gigabytes per second
GFLOPS	Giga Floating Point Operations per Second
GPU	Graphics Processing Unit
L-BFGS-B	Limited Memory Broyden-Fletcher-Goldfarb-Shanno with Boundaries
MCPI	Modified Chebyshev-Picard Iteration
MPBVP	Multi-Point Boundary Value Problem
NLP	Nonlinear Programming
PTX	Parallel Thread Execution
RK4	Runge-Kutta 4th order
SIMD	Single Instruction, Multiple Data
SM	Streaming Multiprocessor
SP	Streaming Processor
STM	State Transition Matrix
TPBVP	Two-Point Boundary Value Problem

ABSTRACT

Antony, Thomas M.S.A.A., Purdue University, December 2014. Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures. Major Professor: Michael J. Grant.

Trajectory optimization is a field which can benefit greatly from the advantages offered by parallel computing. The current state-of-the-art in trajectory optimization focuses on the use of direct optimization methods, such as the pseudo-spectral method. These methods are favored due to their ease of implementation and large convergence regions while indirect methods have largely been ignored in the literature in the past decade except for specific applications in astrodynamics. It has been shown that the shortcomings conventionally associated with indirect methods can be overcome by the use of a continuation method in which complex trajectory solutions are obtained by solving a sequence of progressively difficult optimization problems.

High performance computing hardware is trending towards more parallel architectures as opposed to powerful single-core processors. Graphics Processing Units (GPU), which were originally developed for 3D graphics rendering have gained popularity in the past decade as high-performance, programmable parallel processors. The Compute Unified Device Architecture (CUDA) framework, a parallel computing architecture and programming model developed by NVIDIA, is one of the most widely used platforms in GPU computing. GPUs have been applied to a wide range of fields that require the solution of complex, computationally demanding problems.

A GPU-accelerated indirect trajectory optimization methodology which uses the multiple shooting method and continuation is developed using the CUDA platform. The various algorithmic optimizations used to exploit the parallelism inherent in the indirect shooting method are described. The resulting rapid optimal control

framework enables the construction of high quality optimal trajectories that satisfy problem-specific constraints and fully satisfy the necessary conditions of optimality.

The benefits of the framework are highlighted by construction of maximum terminal velocity trajectories for a hypothetical long range weapon system. The techniques used to construct an initial guess from an analytic near-ballistic trajectory and the methods used to formulate the necessary conditions of optimality in a manner that is transparent to the designer are discussed. Various hypothetical mission scenarios that enforce different combinations of initial, terminal, interior point and path constraints demonstrate the rapid construction of complex trajectories without requiring any a-priori insight into the structure of the solutions. Trajectory problems of this kind were previously considered impractical to solve using indirect methods. The performance of the GPU-accelerated solver is found to be 2x–4x faster than MATLAB's *bvp4c*, even while running on GPU hardware that is five years behind the state-of-the-art.

1. INTRODUCTION

Trajectory optimization problems can be solved using a wide variety of techniques [1]. Since the dawn of modern computing, research by the trajectory design community has focused on direct optimization methods [2–6]. The pseudo-spectral method and other collocation methods are commonly used direct methods. A historical collocation method involves discretizing the trajectory into a number of nodes and optimizing the trajectory assuming a cubic interpolation between the nodes [6]. Current state-of-the-art optimization software such as GPOPS [7] and DIDO [8] use pseudo-spectral methods which implement a more efficient quadrature scheme such as Legendre-Guass-Lobatto [3] with the assumption that the trajectory can be approximated as a polynomial. Both of these methods require a non-linear programming (NLP) solver such as SNOPT [9] and are very computationally intensive for large optimization problems.

One factor that these approaches rarely consider, is the computing platform on which they are executed. While Moore's Law [10] has remained relevant decades after it was originally described, resulting in the packing of more and more transistors into smaller semiconductor devices, the clock-rate at which computer processors operate have essentially peaked [11]. Computing hardware is now transitioning towards highly parallel architectures rather than powerful monolithic processors [12, 13].

Graphics processing units (GPU) were originally designed to be used as dedicated processors for rendering 3D graphics on computers. Efforts to exploit the GPU for general purpose computing applications (GPGPU) have been underway since the early 2000's [14, 15]. One of the earliest demonstrations of GPU computing was a matrix-matrix multiplication algorithm [16]. The availability of floating point operations on GPU hardware allowed the implementation of more advanced computational methods on the GPU [17–19]. Over the last decade, the computing power of GPUs has grown

exponentially as compared to CPUs as shown in Fig. 1.1 and Fig. 1.2 [20]. The modern GPU is a highly capable parallel processor with peak arithmetic and memory bandwidth that substantially outpaces its CPU counterparts [21].

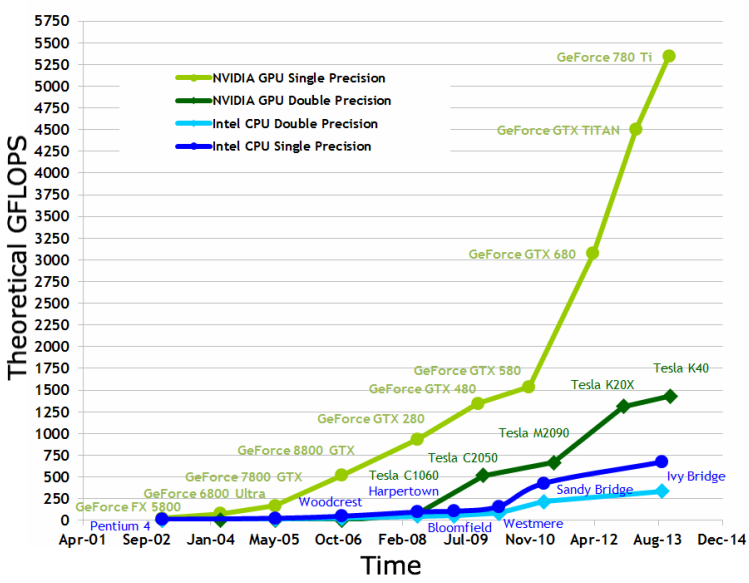

Figure 1.1. CPU vs. GPU performance – Theoretical GFLOPS [20]

While early efforts at GPGPU computing demonstrated great speedups for many applications, it required the programmer to possess intimate knowledge of graphics APIs and GPU architectures. Many basic programming features such as random memory reads/writes and double precision floating point operations were not supported. NVIDIA's Compute Unified Device Architecture (CUDA) is a framework that allows the use of graphics processing units as highly parallel general purpose computing processors. CUDA transforms the graphics processing unit into a powerful parallel processor with thousands of cores. GPUs are now used for accelerating scientific computation in a wide range of fields [22–29].

Direct methods for trajectory optimization, specifically pseudo-spectral and collocations methods, convert the trajectory optimization problem into a non-linear programming problem that involves many sequential, iterative operations instead of large, independent parallel operations. Some of the attempts at implementing NLP

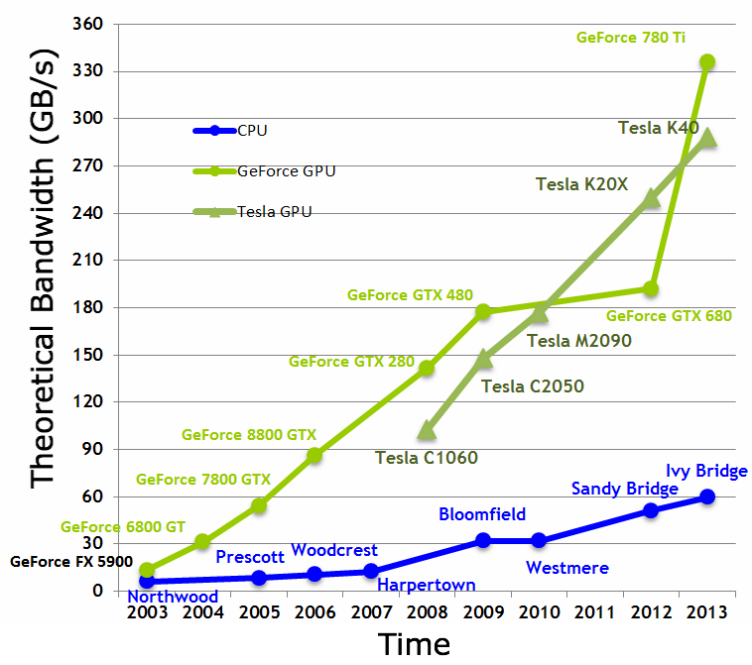

Figure 1.2. CPU vs. GPU performance – Theoretical memory bandwidth (GB/s) by year [20]

algorithms on GPUs have noted 2x-3x speedups over their CPU counterparts for applications in machine learning [30] and radio interferometry [31]. Another recent implementation is that of the L-BFGS-B nonlinear optimization algorithm on a GPU which shows significant speedups, but for only extremely large optimization problems, with dimensions numbering in the millions [32]. The authors of Ref. 32 evaluated the performance of their L-BFGS-B GPU algorithm using the Elastic-Plastic Torsion problem and presented the timing comparisons. As shown in Fig. 1.3, the GPU does not start showing any benefits in speed for that algorithm until the NLP problem size reaches around 6400. Even with a problem size of 40000, the speedup can be observed to be only around 5x.

All these examples consist of NLP problems with extremely large dimensionality. This seems to indicate that the speedups in these examples were produced by virtue of the problems being large enough to take advantage of the large number of processors on the GPU, rather than any inherent parallelism in the methods being used. The

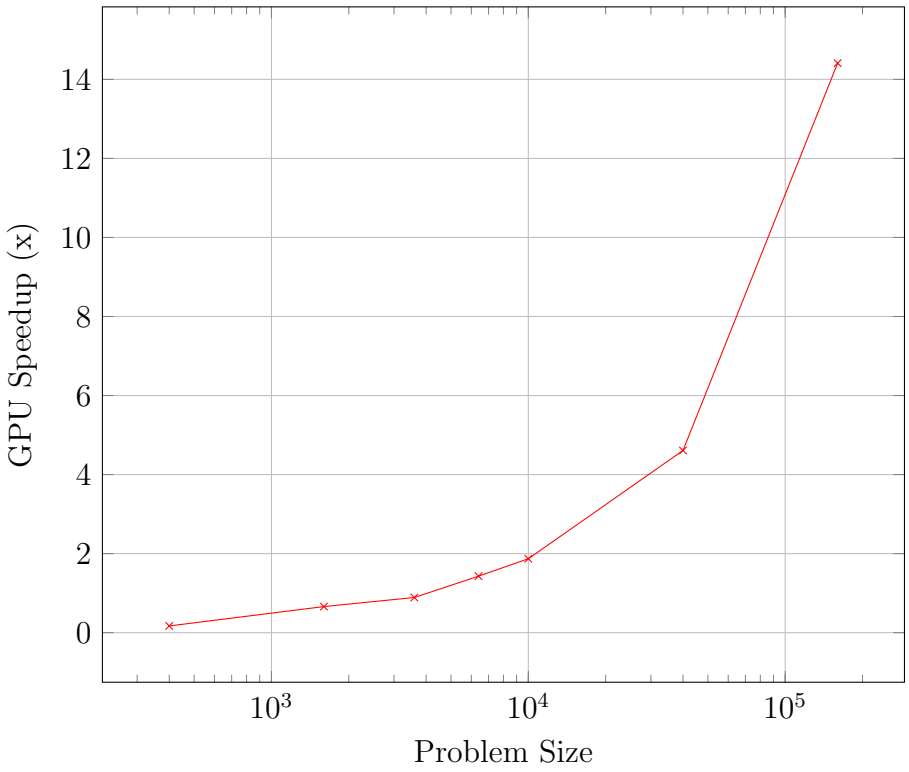

Figure 1.3. Speedup of L-BFGS-B algorithm on GPU for the Elastic-Plastic Torsion problem [32].

NLP problems resulting from the application of pseudo-spectral methods to trajectory problems usually have dimensions in the high hundreds or low thousands. Unless other radically different algorithms with a high degree of parallelism are developed, it seems reasonable to infer that current direct pseudo-spectral optimization algorithms may not benefit from the significant advantages associated with implementation on parallel architectures.

Global search methods such as Genetic Algorithms (GA) and Particle Swarm Optimizers (PSO) are other methods used in direct optimization that may enjoy significant advantages from implementation on a parallel architecture [33–37]. However, GA and PSO methods are zero-order methods and the lack of optimality conditions in these methods lead to lower quality solutions and difficulty in directly handling constraints. GA and PSO are also much more computationally intensive and time

consuming than gradient based algorithms. Prior work has shown that the incorporation of trajectory constraints into the PSO process through penalty functions or as additional objectives greatly increases the difficulty of the design process, while indirect methods are able to precisely satisfy trajectory constraints without this added difficulty [38].

Alternatively, indirect methods use calculus of variations to formulate the optimization problem as a boundary value problem (BVP) [39]. The BVPs can be solved using a suitable numerical solver, and the shooting method is a popular and fast solver [40, 41]. As discussed in the upcoming chapters, the shooting method has many features that make it suitable for implementation on a highly parallel computing architecture such as the GPU. It can be shown that even for a problem with small dimensionality (e.g., an augmented state vector of size 6–24 in a typical hypersonic optimization problem), the shooting method enables a considerable speedup even while running on GPU hardware that is five years behind the state-of-the-art. Indirect methods also have the advantage of generating high quality solutions that satisfy the necessary conditions of optimality. [39].

Historically, three major reasons have been cited as being the reasons for avoiding the use of indirect methods for real-world optimization problems [6].

1. The use of indirect methods requires knowledge of optimal control theory and involves many mathematical derivations that have to be performed prior to the actual solution process.

2. Incorporating path inequality constraints requires a-priori knowledge of the sequence of constrained and unconstrained segments.

3. A very good initial guess is required for the process to converge to a solution. This is often difficult, especially for the mathematical "costate" variables.

All these drawbacks can be overcome using different strategies, enabling rapid, complex, high quality trajectory optimization using indirect methods. Modern ad-

vancements in symbolic computation enables the automated derivation of the equations and the necessary conditions of optimality. The second and third problems listed above can be mitigated by starting the solution process with a short trajectory outside the design space and then using a continuation method to incrementally change the problem to ultimately solve the problem of interest [42]. The same process can be used to incorporate path constraints into the trajectory.

The boundary value problem resulting from using optimal control theory can be solved using a numerical method such as multiple shooting [43] or a solver such MATLAB's *bvp4c* [44] (which uses a collocation method). The shooting method was selected because of its rapid convergence characteristics and low memory use, as well as the inherent parallelism that makes it ideal for implementing on a GPU. Shooting methods convert the boundary value problem into a series of initial value problems. The dynamic equations of the system are propagated using a numerical integration scheme such as 4th order Runge-Kutta (RK4) with guesses for the unknown initial values. The errors in the final value are then corrected iteratively until the desired tolerance is satisfied. This is discussed in further detail in Sections 2.4 and 2.5.

Gradient based algorithms such as the shooting methods depend on the availability of accurate sensitivity information to converge towards the solution. The calculation of the sensitivity information is the most computationally intensive part of the process. This part of the algorithm can be accelerated using an NVIDIA GPU to obtain considerable speedups over a conventional CPU. The computational effort required for computing sensitivity information needed for gradient based optimization algorithms increases exponentially with problem complexity. Given a dynamic system of N equations, the computation of first-order sensitivities has a computational complexity of $O(N^2)$ [45]. Computing the sensitivity information for multiple segments of the trajectory (as is the case in multiple shooting) further increases the amount of computation to be performed. Existing CPU architectures are unable to exploit the massive parallelism inherent in this problem.

Previous work [45], has shown that it is possible to use the GPU to accelerate the computation of sensitivity information for the dynamic system. Following on from that work, many GPU specific optimizations were devised to make the algorithm much faster. The algorithm and the data structures used were designed in such a way as to obtain maximum performance from the GPU. This GPU-based sensitivity algorithm was also designed to be used as part of a larger automated optimization framework. The framework computes the necessary conditions of optimality transparently to the end-user and uses the multiple shooting method to solve the resulting boundary value problem. This optimal control solver, which utilizes indirect methods and GPUs, is capable of solving complex hypersonic trajectory optimization problems with a runtime on the order of tens of seconds on an obsolete GPU (*Fermi* architecture as opposed to the newer and more powerful *Kepler* and *Maxwell* architectures).

Trajectory optimization problems are considered complex because of their infinite-dimensional nature and the presence of discontinuities and constraints in the design space. Indirect methods give solutions of the highest quality and the goal of this research is to implement a solver that can rapidly construct complex, optimal, constrained hypersonic trajectories. Even with the exponential increase in computing power available over the last few decades, there is yet to be any aerospace platform that is capable of performing on-board, real-time trajectory optimization for atmospheric flight. Some scenarios where this can be applied is in ascent/abort guidance for future launch vehicles, adaptive guidance software on military aircraft that can rapidly react to adverse situations, and autonomous mission analysis of any kind on future aerospace platforms. This research demonstrates that trajectory optimization problems can be solved using a GPU to obtain sufficient speedups for supporting future applications for on-board real-time optimization.

2. OVERVIEW OF INDIRECT TRAJECTORY OPTIMIZATION

2.1 Calculus of Variations

Calculus of variations is a field of mathematics that has found applications in fields ranging from optics to quantum mechanics to aerospace engineering, and is the progenitor of modern optimal control theory. While early descriptions of relevant problems can be traced back as far as 300 A.D. [46], the most famous problem associated with calculus of variations is the Brachistochrone problem, posed by Johann Bernoulli in *Acta Eruditorum* in 1696 [47]. The word "brachistochrone" originates from the Greek words for "shortest" and "time". Bernoulli's original problem statement was,

> Given two points A and B in a vertical plane, what is the curve traced out by a point acted on only by gravity, which starts at A and reaches B in the shortest time?

This seemingly simple problem attracted the attention of such great minds as Newton, Lagrange, and Leibniz and eventually resulted in the rise of a field of mathematics known as calculus of variations. Lagrange approached the problem by considering sub-optimal trajectories close to the optimal path. Euler and Lagrange independently developed the differential equation that is now known as the Euler-Lagrange Equation [48]. By following Lagrange's technique, a more generalized set of necessary conditions of optimality can be formulated, using the Euler-Lagrange theorem [49]. This theorem can be used to solve optimal control problems, in which the path resulting from a control variable that optimizes a cost functional is computed, and forms the foundation of indirect trajectory optimization.

Trajectory optimization problems involve the calculation of the time-history of the control variable(s) associated with a system that optimizes a given performance index while satisfying problem-specific constraints at the initial point, terminal point and interior points as well as path constraints. Hypersonic trajectory optimization refers to specific case of trajectories of vehicles flying at hypersonic velocities through an atmosphere. This is generally more complicated than solving pure spaceflight trajectories which, in some cases, may have closed-form analytical solutions [50]. A trajectory optimization problem is generally expressed in the form given in Eq. (2.1). J is the cost functional or performance index to be optimized, with ϕ being the terminal cost and $\int_{t_0}^{t_f} L(x,u,t)\,\mathrm{d}t$ being the path cost. There are also initial and terminal constraints, Ψ and Φ respectively, that are to be satisfied simultaneously.

$$\text{Min } J = \phi(\boldsymbol{x}(t_f), t_f) + \int_{t_0}^{t_f} L(\boldsymbol{x}, \boldsymbol{u}, t)\,\mathrm{d}t \tag{2.1}$$

Subject to :

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t) \tag{2.2}$$

$$\boldsymbol{\Psi}(\boldsymbol{x}(t_0), t_0) = 0 \tag{2.3}$$

$$\boldsymbol{\Phi}(\boldsymbol{x}(t_f), t_f) = 0 \tag{2.4}$$

$$t_0 = 0$$

2.2 Necessary Conditions of Optimality

Indirect methods optimize the cost functional J shown in Eq. (2.1) by formulating a multi-point boundary value problem that represents the necessary conditions of optimality. If these boundary conditions are satisfied, the solution will be locally optimal in the design space. In order to do this, the dynamic equations of the system are augmented with a set of costates, and the necessary conditions of optimality are formulated by applying the Euler-Lagrange equation [49].

The Hamiltonian is defined as shown in Eq. (2.5), where $\boldsymbol{\lambda}$ is the costate vector with its corresponding dynamic equations defined in Eq. (2.6). The optimal control law, $\boldsymbol{u}(t)$ is obtained as a function of the states and costates by solving Eq. (2.7). The initial and terminal boundary conditions on the costates are specified in Eqs. (2.8) and (2.9), where ν_0 and ν_f are sets of undetermined parameters which are used to adjoin these boundary conditions to the cost functional. The time of flight of the trajectory is determined by the free-final time condition in Eq. (2.10). The necessary conditions of optimality are defined by Eqs.(2.6–2.10, and they form a well-defined Two-Point Boundary Value Problem (TPBVP) that can be solved rapidly using the shooting method.

$$H = \; L(x,u,t) + \boldsymbol{\lambda}^T(t)\boldsymbol{f}(\boldsymbol{x},\boldsymbol{u},t) \tag{2.5}$$

$$\dot{\boldsymbol{\lambda}} = \; -\frac{\partial H}{\partial \boldsymbol{x}} \tag{2.6}$$

$$\frac{\partial H}{\partial \boldsymbol{u}} = 0 \tag{2.7}$$

$$\boldsymbol{\lambda}(t_0) = \boldsymbol{\nu_0}^T \frac{\partial \boldsymbol{\Psi}}{\partial \boldsymbol{x}(t_0)} \tag{2.8}$$

$$\boldsymbol{\lambda}(t_f) = \left(\frac{\partial \phi}{\partial \boldsymbol{x}(t_f)} + \boldsymbol{\nu_f}^T \frac{\partial \boldsymbol{\Phi}}{\partial \boldsymbol{x}(t_f)} \right) \tag{2.9}$$

$$\left(H + \frac{\partial \phi}{\partial t} + \boldsymbol{\nu_f}^T \frac{\partial \boldsymbol{\Phi}}{\partial t} \right)_{t=t_f} = 0 \tag{2.10}$$

2.3 Path Constraints & Interior Point Constraints

The presence of path constraints and interior point constraints further complicates the boundary conditions by introducing corner conditions in certain costates and effectively splitting the trajectory into multiple arcs. Path constraints are usually of the form shown in Eq. (2.11). To obtain the control history for the constrained arc, time derivatives of the path constraints are taken until the control variable appears explicitly. If this happens with the q^{th} derivative, the Hamiltonian is augmented as

shown in Eq. (2.12), and the control law for the constraint boundary is obtained by solving $\boldsymbol{S}^{(q)} = 0$.

$$\boldsymbol{S}(\boldsymbol{x}, t) \leq 0 \tag{2.11}$$

$$H = L + \boldsymbol{\lambda}^T \boldsymbol{f} + \boldsymbol{\mu}^T \boldsymbol{S}^{(q)} \tag{2.12}$$

The addition of path constraints also modifies the dynamic equations of the costates along the constrained arcs as shown in Eq. (2.13), where the multipliers $\boldsymbol{\mu}$ are calculated by solving Eq. (2.14).

$$\dot{\boldsymbol{\lambda}} = -\frac{\partial H}{\partial \boldsymbol{x}} = -L_x - \boldsymbol{\lambda}^T \boldsymbol{f}_x - \boldsymbol{\mu}^T \boldsymbol{S}_{\boldsymbol{x}}^{(q)} \tag{2.13}$$

$$\frac{\partial H}{\partial \boldsymbol{u}} = L_u + \boldsymbol{\lambda}^T \boldsymbol{f}_u + \boldsymbol{\mu}^T \boldsymbol{S}_{\boldsymbol{u}}^{(q)} = 0 \tag{2.14}$$

The states are continuous at the entry (t_1) and exit (t_2) of the constrained arc as shown in Eq. (2.15). Corner conditions on costates are chosen such that the costates are continuous at the exit of the constrained arc as shown in Eq. (2.16). The tangency conditions described in Eq. (2.17) and corner conditions in Eq. (2.18) and Eq. (2.19) apply at the entry point of the constrained arc.

$$\begin{aligned} \boldsymbol{x}(t_1{}^+) &= \boldsymbol{x}(t_1{}^-) \\ \boldsymbol{x}(t_2{}^+) &= \boldsymbol{x}(t_2{}^-) \end{aligned} \tag{2.15}$$

$$\begin{aligned} \boldsymbol{\lambda}(t_2{}^+) &= \boldsymbol{\lambda}(t_2{}^-) \\ H(t_2{}^+) &= H(t_2{}^-) \end{aligned} \tag{2.16}$$

$$\boldsymbol{N}(\boldsymbol{x},t) = \begin{bmatrix} S(\boldsymbol{x},t) \\ S^{(1)}(\boldsymbol{x},t) \\ S^{(2)}(\boldsymbol{x},t) \\ . \\ . \\ . \\ S^{(q-1)}(\boldsymbol{x},t) \end{bmatrix} \tag{2.17}$$

$$\boldsymbol{\lambda}({t_1}^+) = \boldsymbol{\lambda}({t_1}^-) + \boldsymbol{\Pi}^T \boldsymbol{N}_{\boldsymbol{x}} \tag{2.18}$$

$$H({t_1}^+) = H({t_1}^-) + \boldsymbol{\Pi}^T \boldsymbol{N}_t \tag{2.19}$$

Interior point constraints are very similar to the tangency conditions in a path constraint, described in Eq. (2.17). It can be considered to be a case where the constrained arc is infinitesimally small. The states remain continuous across the "junction", but the costates and the Hamiltonian may have jump conditions imposed on them. An interior point constraint is defined as shown in Eq. (2.20). Introducing interior point constraints into the problem results in the continuity and corner conditions in Eq. (2.21), where $\boldsymbol{\Pi}$ is a vector of unknown parameters.

$$\boldsymbol{N}(\boldsymbol{x}(t_1), t_1) = 0 \tag{2.20}$$

$$\begin{aligned} \boldsymbol{x}({t_1}^-) &= \boldsymbol{x}({t_1}^+) \\ \boldsymbol{\lambda}({t_1}^-) &= \boldsymbol{\lambda}({t_1}^+) + \boldsymbol{\Pi}^T \boldsymbol{N}_{\boldsymbol{x}} \\ H({t_1}^-) &= H({t_1}^+) + \boldsymbol{\Pi}^T \boldsymbol{N}_t \end{aligned} \tag{2.21}$$

The Multi-Point Boundary Value Problem (MPBVP) resulting from applying the necessary conditions of optimality discussed above, is solved using the multiple shooting method.

2.4 Single Shooting Method

If $\boldsymbol{X}$ is the augmented state vector consisting of both the states and costates as shown in Eq. 2.22a, a TPBVP resulting from a trajectory optimization problem takes the form described by Eqs. 2.22b–2.22c.

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{\lambda} \end{bmatrix} \tag{2.22a}$$

$$\boldsymbol{g}(\boldsymbol{X}_0, \boldsymbol{X}_f) = 0 \tag{2.22b}$$

$$\dot{\boldsymbol{X}}(t) = \boldsymbol{f}(\boldsymbol{X}, t) \tag{2.22c}$$

where $\boldsymbol{X}_0$ and $\boldsymbol{X}_f$ are the values of $\boldsymbol{X}$ at the times t_0 and t_f respectively.

The single shooting method root-solves for the values of $\boldsymbol{X}_0$ and $\boldsymbol{X}_f$ that satisfy these conditions. In order to do so, an initial guess for $\boldsymbol{X}_0$ is used to propagate the dynamic equations of the system ($\dot{\boldsymbol{X}}$). Along with the dynamic equations we also propagate equations describing the sensitivity of the system which form the state transition matrix (STM), Φ. $\Phi(t_f)$ and $\boldsymbol{X}_f$ are obtained by propagating Eqs. (2.22c) and (2.23) and are used to evaluate the residual error ($\boldsymbol{\epsilon}$) in the boundary conditions (value of $\boldsymbol{g}(\boldsymbol{X}_0, \boldsymbol{X}_f)$). T

$$\dot{\Phi} = F \cdot \Phi, \quad F = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{X}}, \quad \Phi(t_0) = I_N \tag{2.23}$$

his residual is then used to compute a correction ($\Delta \boldsymbol{X}_0$) to the initial guess which will drive this error to zero as shown in Eqs. (2.24a–2.24e).

$$\boldsymbol{g}(\boldsymbol{X}_0, \boldsymbol{X}_f) = \boldsymbol{\epsilon} \tag{2.24a}$$

$$\boldsymbol{g}(\boldsymbol{X}_0 + \Delta\boldsymbol{X}_0, \boldsymbol{X}_f) = 0 \tag{2.24b}$$

$$\implies \boldsymbol{g}(\boldsymbol{X}_0 + \Delta\boldsymbol{X}_0, \boldsymbol{X}_f) - \boldsymbol{g}(\boldsymbol{X}_0, \boldsymbol{X}_f) = -\boldsymbol{\epsilon} \tag{2.24c}$$

Using a first-order expansion of the above expression,

$$\boldsymbol{g}(\boldsymbol{X}_0, \boldsymbol{X}_f) + \frac{\mathrm{d}\boldsymbol{g}}{\mathrm{d}\boldsymbol{X}_0}\Delta\boldsymbol{X}_0 - \boldsymbol{g}(\boldsymbol{X}_0, \boldsymbol{X}_f) = -\boldsymbol{\epsilon} \tag{2.24d}$$

$$\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{X}_0} + \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{X}_f}\frac{\partial \boldsymbol{X}_f}{\partial \boldsymbol{X}_0}\right)\Delta\boldsymbol{X}_0 = -\boldsymbol{\epsilon} \tag{2.24e}$$

The correction vector $\Delta\boldsymbol{X}_0$ can be thus be obtained by solving the following linear system in Eq. (2.25).

$$(M + N\Phi)\Delta\boldsymbol{X}_0 = -\boldsymbol{\epsilon} \tag{2.25}$$

where M and N are the Jacobian matrices, obtained by taking partial derivatives of the boundary conditions $\boldsymbol{g}$ with respect to $\boldsymbol{X}_0$ and $\boldsymbol{X}_f$ respectively, and Φ is the sensitivity matrix of the system. This is called the single shooting method.

2.5 Multiple Shooting Method

When a trajectory optimization problem contains path inequality or interior point constraints, the trajectory is split into multiple arcs, with the possibility of discontinuities at the junctions. These boundary value problems are called multipoint boundary value problems (MPBVP). The problem may also contain scalar parameters such as the Lagrange multipliers, $\boldsymbol{\nu}_0$, $\boldsymbol{\nu}_f$ and $\boldsymbol{\pi}$, that have to be solved along with the state and costate trajectories. In such cases, the correction vector is computed using an extension of the single shooting method, called the multiple shooting method. The general form of a MPBVP is as follows:

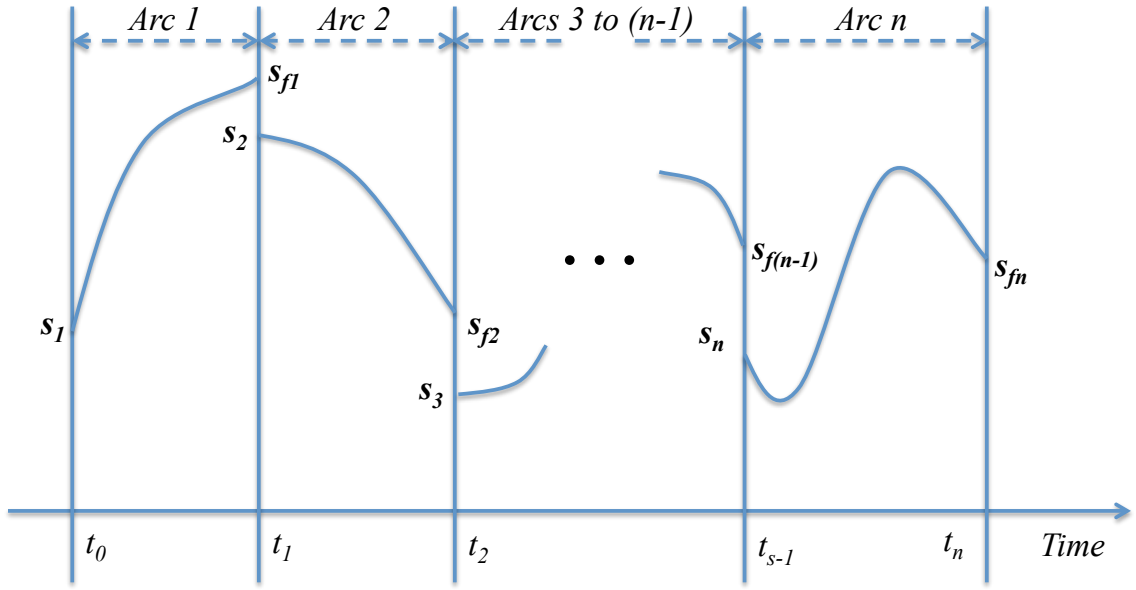

Figure 2.1. An example multi-point boundary value problem.

$$\boldsymbol{g}(\boldsymbol{s}_1, \boldsymbol{s}_{f1}, \boldsymbol{s}_2, \boldsymbol{s}_{f2} \ldots \boldsymbol{s}_n, \boldsymbol{s}_{fn}, \boldsymbol{p}) = 0$$

$$\dot{\boldsymbol{s}} = \begin{cases} \boldsymbol{f}_1(t, \boldsymbol{s}) & \text{if } t_0 < t < t_1 \\ \boldsymbol{f}_2(t, \boldsymbol{s}) & \text{if } t_1 < t < t_2 \\ \qquad\vdots & \\ \boldsymbol{f}_n(t, \boldsymbol{s}) & \text{if } t_{n-1} < t < t_n \end{cases}$$

where $\boldsymbol{s}_1$, $\boldsymbol{s}_2$ $\ldots$ $\boldsymbol{s}_n$ are the values at the left endpoints of the arcs, $\boldsymbol{s}_{f1}$, $\boldsymbol{s}_{f2}$, $\ldots$ $\boldsymbol{s}_{fn}$ are the values at the right endpoints of the arcs as shown in Fig. 2.1, and $\boldsymbol{p}$ is the set of scalar parameters.

To solve this problem, we first compute a Jacobian matrix J, which is of the form:

$$J = \begin{bmatrix} M_1 + N_1\Phi_1 & M_2 + N_2\Phi_2 & \cdots & M_n + N_n\Phi_n & J_p \end{bmatrix} \tag{2.26}$$

$$
\begin{aligned}
M_1 = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{s}_1}, M_2 = \frac{\partial \boldsymbol{g}}{\boldsymbol{s}_2}, \cdots \; M_n = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{s}_n}, \\
N_1 = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{s}_{f1}}, N_2 = \frac{\partial \boldsymbol{g}}{\boldsymbol{s}_{f2}} \; \cdots \; N_n = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{s}_{fn}}, \\
J_p = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}}
\end{aligned}
\tag{2.27}
$$

The sensitivity matrices for each arc is computed separately as Φ_1,Φ_2, $\ldots \Phi_n$. The correction vector $\Delta \boldsymbol{s}$ will consist of corrections to $\boldsymbol{s_1}$, $\boldsymbol{s_2}$ $\ldots \boldsymbol{s_n}$ and $\boldsymbol{p}$, and it is computed by solving the following linear system.

$$
J \Delta \boldsymbol{s} = -\boldsymbol{\epsilon} \tag{2.28}
$$

3. PARALLEL SENSITIVITY COMPUTATIONS ON THE GPU

3.1 Overview of the NVIDIA *Fermi* Architecture

The general layout of an Intel CPU and that of an NVIDIA Fermi [51] GPU are shown in Fig. 3.1 and Fig. 3.2, respectively. In the CPU shown in Fig. 3.1, there are four processing cores and a large cache memory block. In the Fermi GPU (Fig. 3.3), each of the long green rectangles represent a Streaming Multiprocessor (SM), that contain up to 32 individual processing cores or Streaming Processors (SP) for a total of up to 512 processing cores per GPU (Fig. 3.3). The newer Kepler [52] and Maxwell [53] architectures support up to 2880 cores per GPU.

The reason for the discrepancy in performance between CPUs and GPUs is that the GPU is specialized for compute-intensive, highly parallel operations. On the other hand, CPUs were designed with more transistors dedicated to data caching, and flow control, rather than data processing. Hence, a GPU is especially suited to problems which can be expressed as data-parallel computations [20], with a high ratio of arithmetic operations to memory operations.

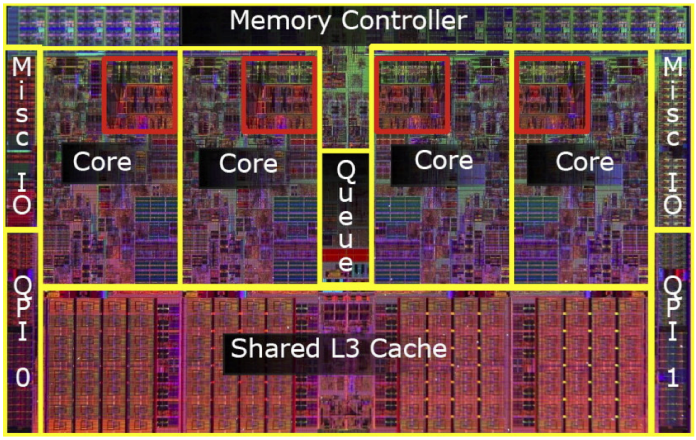

Figure 3.1. Intel Core i7 *Nehalem* architecture [54].

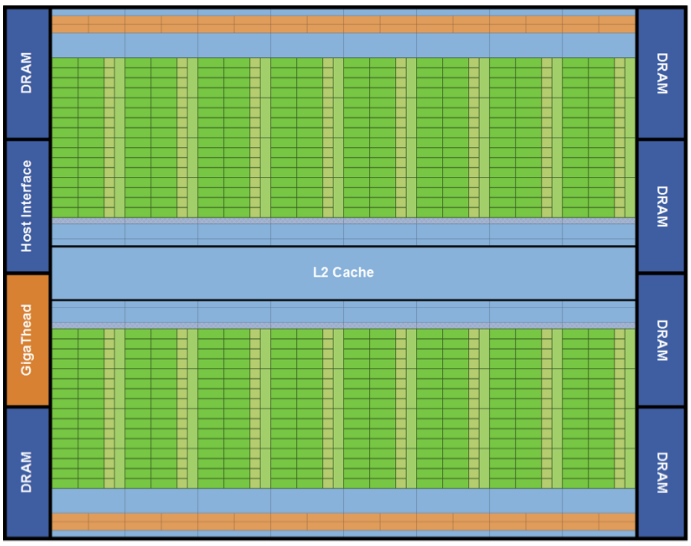

Figure 3.2. NVIDIA *Fermi* architecture [51].

3.2 Overview of the NVIDIA CUDA framework

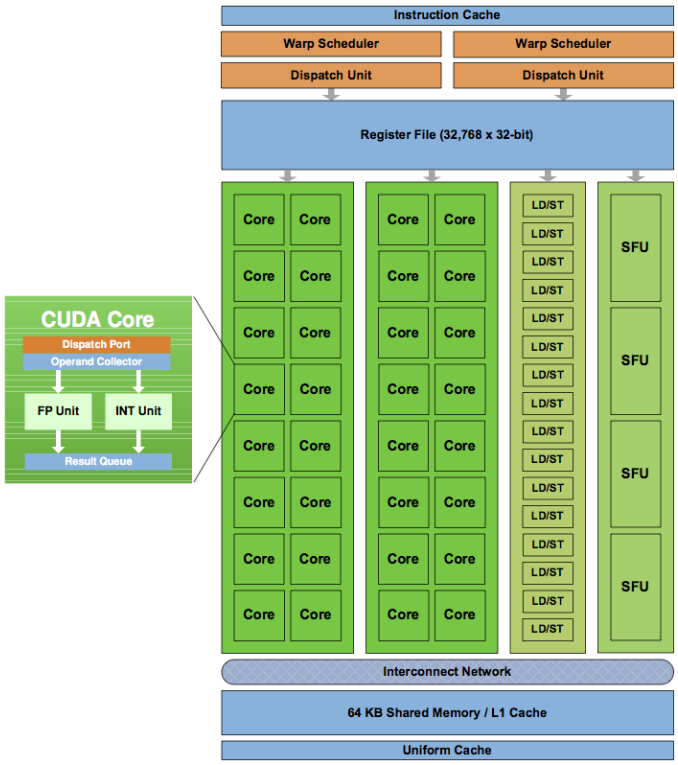

Figure 3.3. *Fermi* streaming multiprocessor [51].

The Compute Unified Device Architecture (CUDA) is a parallel computing framework and architecture developed by NVIDIA for enabling general purpose computing on their GPUs. CUDA implements extensions to several industry standard programming languages including C, C++, and Fortran and comes with an extensive library of commonly used parallel programming constructs. It is particularly useful for solving problems which are more computationally intense as opposed to memory or data intense due to the relatively slow nature of GPU memory. CUDA programs have been able to give speedups of 5x to 500x over their CPU counterparts in many applications ranging from molecular simulation [25] to modeling of aircraft traffic [26]. Struc-

turing the problem and the data in the right manner is key to obtaining maximum performance in GPU computing.

The parallel functionality in CUDA is implemented in units called kernels. When a kernel is invoked, it is copied onto the thousands of cores on the GPU and executed simultaneously. Typically, the thousands of threads that are spawned perform the same set of operations on different sets of data. This is called the Single Instruction Multiple Data or SIMD paradigm. Algorithms that are able to follow the SIMD paradigm as much as possible are able to obtain maximum performance from GPU computing. It is to be noted that the concept of a "thread" on a GPU is different from what it means on a CPU. Generating and scheduling threads in CUDA is much faster than doing the same on a CPU, but each individual processor on a GPU will not be as powerful as a CPU core.

When a kernel is invoked, a large number of threads are launched, collectively called a grid. The grid is divided into thread blocks and each thread block consists of the individual threads. Each thread-block is scheduled to execute on a single SM where all of its threads are executed simultaneously. The manner in which CUDA schedules and executes the individual threads plays an important role in structuring the problem for maximum performance. This is examined in detail in Chapter 4.

NVCC, the NVIDIA C Compiler compiles the CUDA C code into a binary format that can be interpreted by the GPU. The code is separated into "host" and "device" code, with the former running on the CPU and the latter on the GPU. The two parts of the code are compiled separately and then combined. In CUDA, there can be no GPU device code that runs by itself. There is always some host code which launches the kernels and then collects the results for post-processing. NVCC can also generate device code that can be loaded by CUDA at runtime. This is very useful in cases where problem-specific information (such as the dynamic equations) have to be changed at run-time.

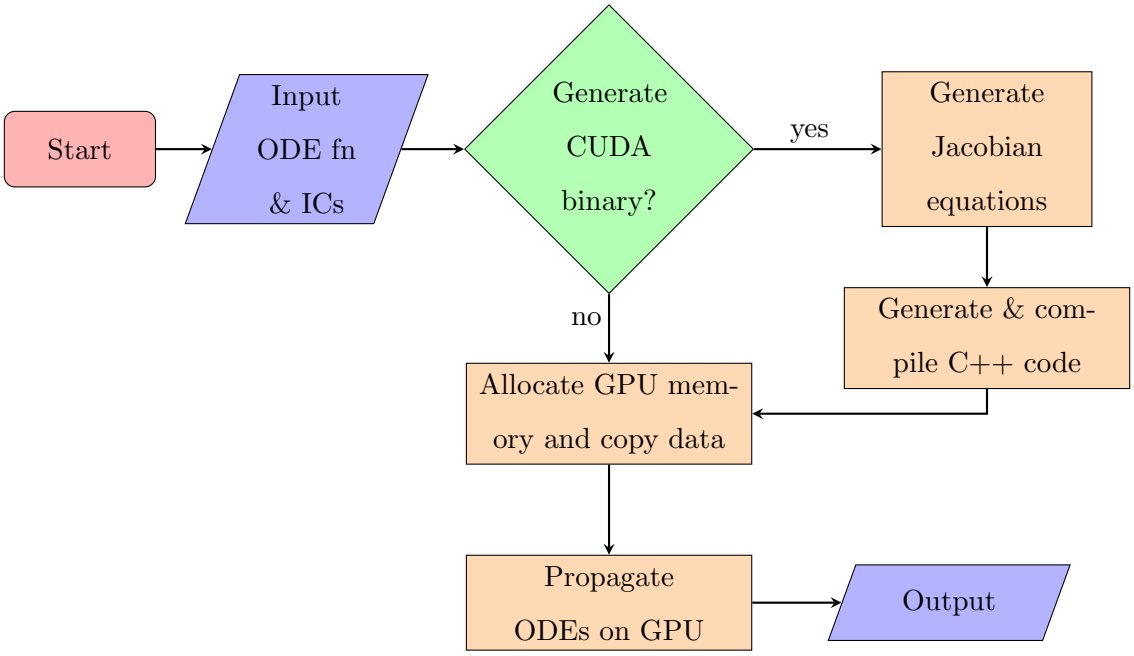

Figure 3.4. Flowchart of GPU accelerator implementation.

3.3 GPU-Accelerated Solver Implementation

The GPU-accelerated optimal control solver implements a MATLAB interface. The host-code which controls the overall process and launches the GPU kernel is pre-compiled into a MATLAB *mex* library. The solver uses the Mathematica [55] to derive the necessary conditions of optimality and generates and compiles CUDA C code transparently to the designer. This produces standalone device code in a binary format which can be loaded at run-time by CUDA (described further in Section 5.4). This process is to be executed only once per problem. Once the CUDA binaries are available, changing problem parameters doesn't require recompilation. Figure 3.4 outlines this process.

The dynamic equations of the system are propagated on the CPU using compiled MATLAB code and only the sensitivity information is propagated on the GPU. It was done in this manner because the number of dynamic equations (usually around 6-24 for re-entry problems), does not offer a big enough problem to justify the overheads

of using the GPU. Benchmarks showed that a hybrid method where the states are propagated on the CPU and the sensitivities on the GPU gave the best performance with current hardware. The solver was tested on an NVIDIA Tesla M2090 GPU (*Fermi* architecture) with an Intel Xeon E5-2670 2.60GHz processor CPU.

4. GPU OPTIMIZATIONS

At the most basic level, computing the sensitivity matrix or the State Transition Matrix (STM) involves propagating N^2 extra differential equations for a dynamic system of N equations. The naive way of porting this over to a GPU would involve assigning each equation (from both the original system of equations, as well as the STM) to a separate thread on the GPU, and launching a kernel to solve the problem. While this is very simple to implement, it is also very inefficient. In fact, benchmarking showed that this made the process twice as slow as performing the same operation on a CPU. In order to optimize the code for maximum performance on the GPU, it is necessary to understand how the threads are scheduled and executed by CUDA.

4.1 GPU Occupancy and Thread Divergence

Modern GPUs have a large quantity of device memory on the order of gigabytes. However, accessing this memory is typically slower than memory access by a CPU. CUDA's thread scheduling process is key to overcoming this problem. At the hardware level, the threads from a kernel are executed in groups called "warps", each with 32-128 threads depending on architecture. When the threads in a warp request data from memory and are waiting for it, the whole warp is replaced with a new one for execution on the SM. This way, as long as there are enough threads to schedule on the processor, CUDA can very efficiently hide the latency of memory access. However, this is also the main caveat in GPU computing: the problem has to be large enough and have enough parallel units running simultaneously to hide the memory latency.

All the threads in a single thread-block will always be executed on the same SM. This allows the threads to communicate with each other, if required, using high speed shared memory. Every thread in a warp must execute the same code in order for them

to be executed in parallel. If this is not the case, the threads will incur a penalty by executing in a serial manner. This is called the thread divergence penalty.

Both the memory latency and thread divergence penalties can be addressed by splitting the trajectory into smaller segments. This is possible because of a unique feature of the STM, which makes it possible to compute its value for the entire trajectory by independently computing the STMs for segments of the trajectory and combining them. If Φ_1, $\Phi_2 \ldots \Phi_P$ are the STMs of the individual trajectory segments, the STM for the whole trajectory, Φ can be computed as shown in Eq. (4.1). It is to be noted that these trajectory segments are different from the multiple arcs described in Section 2.5.

$$\Phi = \Phi_P \cdot \Phi_{P-1} \cdot \Phi_{P-2} \cdot \Phi_{P-3} \cdot \ldots \cdot \Phi_2 \cdot \Phi_1 \tag{4.1}$$

Since the GPU used in our tests supports launching up to 512 threads per thread-block, the trajectory is split into a maximum of 512 segments, each integrating a fixed number of RK4 time steps. The work is assigned such that every thread in a thread-block processes the *same* element of the STM for different segments of the trajectory. This way the algorithm follows the SIMD paradigm, since every thread in one thread-block will integrate the *same* equations for different trajectory segments. Thus, the thread divergence penalty can be avoided.

It is possible to maximize the occupancy of the GPU cores by scheduling the mos number of threads possible. This in turn speeds up the algorithm by having enough threads scheduled to hide the memory access latency. If the trajectory is setup to use 512 RK4 time steps (which is sufficient for a typical hypersonic trajectory problem) and then split into 512 segments, each thread propagates a different time step of the STM propagation in parallel, making it much faster than propagating the 512 time steps sequentially on a CPU.

This effect is further magnified when the multiple shooting method from Section 2.5 is used. In this case, the trajectory is separated into multiple arcs, with the possibility of some of them following different dynamic equations (e.g., in the case

of constrained arcs). In such a situation, each trajectory arc can be assigned to a different set of thread-blocks, with each individual arc again being split into smaller segments, enabling the parallel computation of the sensitivity information for multiple trajectory arcs.

The propagation of the dynamic equations of the trajectory is still performed on the CPU since the segmented approach is not possible for propagation of those equations. The data from the CPU-propagated trajectory including the intermediate RK4 stages are preprocessed into the segmented form that is required for the propagation of the STM on the GPU.

4.2 Memory Access Coalescing

GPU device memory is accessed in chunks of 32-, 64-, or 128-byte data blocks called memory transactions. When a warp executes an instruction that accesses device memory, it coalesces the memory accesses of the threads within the warp into one or more transactions. Depending on the distribution of the memory addresses accessed by the threads, it may have to issue more memory transaction requests to access the same amount of data. In order to optimize this, the data stored on the GPU memory should be stored such that adjacent threads in a thread-block access consecutive memory addresses when reading or writing to memory. This allows memory to be fetched using minimum number of transactions, resulting in higher data throughput.

In the case of the optimal control solver, this is implemented by storing the STM data in a particular manner. STMs for all the different segments of the trajectory are combined into one large data structure, as shown in Figure 4.1. The diagram represents the storage of the elements of the STM for a trajectory with P segments and N dynamic equations. $\underset{i,j}{\Phi_k}$ represents element (i, j) of the STM for trajectory segment k. The data is stored in column-major format. Each thread-block propagates the same element of the STM for all the trajectory segments. For example, the first column of the data structure, stores element $\underset{1,1}{\Phi}$ for all the trajectory segments,

followed by the element $\underset{2,1}{\Phi}$ and so on. The first thread-block propagates the equations for element $\underset{1,1}{\Phi}$ for all the segments, and accesses the contiguous memory block from $\underset{1,1}{\Phi_1}$ to $\underset{1,1}{\Phi_p}$. Thus, each thread-block propagates the same dynamic equations for different segments of the trajectory and every thread in the block accesses memory elements adjacent to each other, thereby achieving memory access coalescing.

$\Phi^1_{1,1}$	$\Phi^1_{1,2}$	$\cdots$	$\Phi^1_{1,N-1}$	$\Phi^1_{1,N}$
$\Phi^2_{1,1}$	$\Phi^2_{1,2}$	$\cdots$	$\Phi^2_{1,N}$	$\Phi^2_{1,N}$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$\Phi^{P-1}_{1,1}$	$\Phi^{P-1}_{1,2}$	$\cdots$	$\Phi^{P-1}_{1,N-1}$	$\Phi^{P-1}_{1,N}$
$\Phi^P_{1,1}$	$\Phi^P_{1,2}$	$\cdots$	$\Phi^P_{1,N-1}$	$\Phi^P_{1,N}$
$\Phi^1_{2,1}$	$\Phi^1_{2,2}$	$\cdots$	$\Phi^1_{2,N-1}$	$\Phi^1_{2,N}$
$\Phi^2_{2,1}$	$\Phi^2_{2,2}$	$\cdots$	$\Phi^2_{2,N-1}$	$\Phi^2_{2,N}$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$\Phi^P_{2,1}$	$\Phi^{P-1}_{2,2}$	$\cdots$	$\Phi^{P-1}_{2,N-1}$	$\Phi^{P-1}_{2,N}$
$\Phi^P_{2,1}$	$\Phi^P_{2,2}$	$\cdots$	$\Phi^P_{2,N-1}$	$\Phi^P_{2,N}$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$\Phi^1_{N-1,1}$	$\Phi^1_{N-1,2}$	$\cdots$	$\Phi^1_{N-1,N-1}$	$\Phi^1_{N-1,N}$
$\Phi^2_{N-1,1}$	$\Phi^2_{N-1,2}$	$\cdots$	$\Phi^2_{N-1,N-1}$	$\Phi^2_{N-1,N}$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$\Phi^{P-1}_{N-1,1}$	$\Phi^{P-1}_{N-1,2}$	$\cdots$	$\Phi^{P-1}_{N-1,N-1}$	$\Phi^{P-1}_{N-1,N}$
$\Phi^P_{N-1,1}$	$\Phi^P_{N-1,2}$	$\cdots$	$\Phi^P_{N-1,N-1}$	$\Phi^P_{N-1,N}$
$\Phi^1_{N,1}$	$\Phi^1_{N,2}$	$\cdots$	$\Phi^1_{N,N-1}$	$\Phi^1_{N,N}$
$\Phi^2_{N,1}$	$\Phi^2_{N,2}$	$\cdots$	$\Phi^2_{N,N-1}$	$\Phi^2_{N,N}$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$
$\Phi^{P-1}_{N,1}$	$\Phi^{P-1}_{N,2}$	$\cdots$	$\Phi^{P-1}_{N,N-1}$	$\Phi^{P-1}_{N,N}$
$\Phi^P_{N,1}$	$\Phi^P_{N,2}$	$\cdots$	$\Phi^P_{N,N-1}$	$\Phi^P_{N,N}$

Figure 4.1. Combined STM data structure in GPU memory.

4.3 Parallel Matrix Reduction

One of the main advantages of using CUDA as opposed to other GPU programming technologies, is the availability of built-in GPU-optimized libraries for performing common operations in scientific computing. The NVIDIA CUDA Basic Linear Algebra Subroutines (*CUBLAS*) library is a GPU-accelerated version of the standard BLAS library that offers GPU-optimized versions of many common linear algebra and matrix operations.

CUBLAS was originally designed for operations on large matrices and was optimized for such operations. In older versions, it was possible to use CUDA *streams* to launch multiple kernels that run concurrently to perform operations on a large number of smaller matrices. However, starting with CUDA Toolkit 5.0, CUBLAS offers a batched matrix multiplication API that is meant for efficient multiplication of a large number of smaller matrices on the GPU.

Since the computation of STMs from a segmented trajectory involves a long chain of matrix multiplication operations shown in Eq. (4.1), the new batched matrix multiplication API is found to be useful for making the process even faster. In order to do this, the process in Eq. (4.1) is transformed into a series of parallel steps called matrix reduction, an example of which is shown in Eq. (4.2).

$$\left.\begin{aligned}
\Phi &= \left(\Phi_{512}\cdot\Phi_{511}\right)\cdot\left(\Phi_{510}\cdot\Phi_{509}\right)\cdot\ldots\cdot\left(\Phi_{4}\cdot\Phi_{3}\right)\cdot\left(\Phi_{2}\cdot\Phi_{1}\right)\\
&= \left(\Phi_{512\cdot511}\cdot\Phi_{510\cdot509}\right)\cdot\ldots\cdot\left(\Phi_{4\cdot3}\cdot\Phi_{2\cdot1}\right)\\
&= \Phi_{512\cdot511\cdot510\cdot509}\cdot\ldots\cdot\Phi_{4\cdot3\cdot2\cdot1}\\
&\;\;\vdots\\
&= \Phi_{512\cdot511\cdot\ldots\cdot2\cdot1}
\end{aligned}\right\}\quad log_2512 = 9 \text{ steps} \tag{4.2}$$

The number of trajectory segments is chosen to be a power of 2 so that the matrix reduction process results in two matrices at the end, which are multiplied to give the complete STM. Each of the bracketed operations in Eq. (4.2) were performed

independently of each other in parallel. If there are P segments in the trajectory, the complete STM can be obtained in $log_2 P$ steps of parallel matrix reduction as against P separate sequential matrix multiplications when using serial processing.

5. SOLUTION STRATEGY

5.1 A Hypersonic Trajectory Optimization Problem

Trajectory optimization of a hypothetical unpowered long range weapon is used to demonstrate the benefits of the GPU-accelerated optimal control solver. In the following examples, a hypothetical US warship is located near the Gulf of Oman. A hypothetical high-value target is located in a cave within a mountain range inside Afghanistan. The long range weapon must be delivered with maximum velocity to the target. The vehicle is assumed to be released at a specified boost condition with a certain altitude, latitude, longitude, and velocity. A terminal position is specified in terms of altitude, latitude, and longitude. These common entry/impact conditions are specified in Table 5.1.

There may also be other constraints at the initial or terminal point as well as different combinations of path constraints (such as heat rate) and/or interior point constraints (e.g., country overflight constraints) along the trajectory depending on the scenario being analyzed. These conditions represent fictitious but relevant mission scenarios.

A vehicle-centric polar coordinate system and 3-DOF dynamic model [56] are used to develop the equations of motion for the problem as shown in Eqs. (5.2–5.7). Angle of attack, α, and bank angle, σ, are used as the control variables in the optimization problem. A spherical Earth gravity model with an exponential atmosphere is assumed with the parameters shown in Table 5.2. A high performance hypersonic vehicle model with a mass of 340 kg and peak L/D of around 2.5 is selected. The optimal control problem is formally stated in Eqs. (5.1)–(5.9).

Table 5.1. Post-boost staging and impact conditions.

State	*Post-Boost Condition*	*Impact Condition*
Altitude, h	80,000 m	4,570 m
Velocity, v	6000 m/s	free
Flight-Path Angle, γ	free	-60 deg
Latitude, θ	23.14 deg	33.66 deg
Longitude, ϕ	64.07 deg	67.63 deg

Table 5.2. Environment parameters.

Parameter	*Value*
Scale Height, h_{scale}	7500 m
Surface Density, ρ_o	1.2 kg/m^3
Gravitational Parameter, μ	3.986e14 m^3/s^2
Earth Radius, r_e	6378000 m
Rotation Rate, ω	7.292×10^{-5} rad/s
Heat Rate Coefficient, k	1.742×10^{-4} $\sqrt{kg}/m$

$$\text{Min } J = -V(t_f)^2 \tag{5.1}$$

Subject to :

$$\dot{r} = V\sin(\gamma) \tag{5.2}$$

$$\dot{\theta} = \frac{V\cos(\gamma)\cos(\psi)}{r\cos(\phi)} \tag{5.3}$$

$$\dot{\phi} = \frac{V\cos(\gamma)\sin(\psi)}{r} \tag{5.4}$$

$$\begin{aligned}\dot{V} = &-\frac{qC_DA_{ref}}{m} - \frac{\mu\sin(\gamma)}{r^2} \\ &+ \omega^2 r\cos(\phi)\Big(\sin(\gamma)\cos(\phi) - \cos(\gamma)\sin(\phi)\sin(\psi)\Big)\end{aligned} \tag{5.5}$$

$$\begin{aligned}\dot{\gamma} = &\frac{qC_LA_{ref}\cos(\sigma)}{mV} - \frac{\mu\cos(\gamma)}{Vr^2} + \frac{V\cos(\gamma)}{r} + 2\omega\cos(\phi)\cos(\psi) + \\ &\omega^2 r\cos(\phi)\Big(\cos(\gamma)\cos(\phi) + \sin(\gamma)\sin(\phi)\sin(\psi)\Big)\end{aligned} \tag{5.6}$$

$$\begin{aligned}\dot{\psi} = &\frac{qC_LA_{ref}\sin(\sigma)}{mV\cos(\gamma)} - \frac{V\cos(\gamma)\cos(\psi)\tan(\phi)}{r} \\ &+ 2\omega\Big(\tan(\gamma)\cos(\phi)\sin(\psi) - \sin(\phi)\Big) - \frac{\omega^2 r\sin(\phi)\cos(\phi)\cos(\psi)}{V\cos(\gamma)}\end{aligned} \tag{5.7}$$

where

$$\begin{aligned}q &= \frac{1}{2}\rho V^2 \\ \rho &= \rho_0\exp(-(r - R_E)/h_{scale})\end{aligned} \tag{5.8}$$

$$\begin{aligned}C_L &= C_{L1}\alpha + C_{L0} \\ C_D &= C_{D2}\alpha^2 + C_{D1}\alpha + C_{D0}\end{aligned} \tag{5.9}$$

5.2 Development of Necessary Conditions

In order to use indirect methods to solve trajectory problems, it is first required to derive the necessary conditions for all the possible scenarios in the problem. This includes corner conditions for interior point and path constraints, control laws, and boundary conditions. The algorithm derives these using Mathematica [55]. The

control law is derived by symbolically root-solving the control law equation, also using Mathematica. The control law is root-solved numerically if complete analytical formulations of the dynamics of the system are not available. This can be the case when tables of aerodynamic data are used to model the vehicle's behavior.

5.3 Selection of Control Options

The algorithm derives all the possible control combinations by applying optimal control theory. For example, in the test scenarios in Chapter 6, root-solving for the control variables results in four distinct combinations of angle of attack and bank angle. This happens because root-solving trigonometric functions results in multiple answers.

During the optimization process, the correct control branch is selected based on Pontryagin's Minimum Principle described by Eq. (5.10), where "*" refers to the optimum solution. Essentially, the Hamiltonian from Eq. (2.5) is evaluated for all the possible control options, and the one which gives the minimum value is selected as the optimal control, and this process is repeated for every point of time in the trajectory. This selection process is automated and performed fully transparently to the designer and is performed for both the constrained and unconstrained trajectory segments.

$$H(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{\lambda}^*(t), t) \leq H(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{\lambda}x(t), t) \tag{5.10}$$

5.4 Compilation of Problem-Specific Files

Executing code on the GPU requires compilation of the code into special binary formats that are readable by the GPU. This has to be done for the code that contains the dynamic equations (Eq. (5.1)) specific to the problem. This is a one-time process, as long as no further changes are made to the equations of motion. The dynamic equations for the state transition matrix, shown in Eq. (2.23), involve partial derivatives of the equations of motion. These are numerically computed using either the

finite difference method or complex-step derivative method as shown in Eqs. (5.11) and (5.12) depending on the designer's choice. The complex-step derivative method, while being very accurate to arbitrary precision, cannot be used in the cases where there are complex solutions for the control law. Numerical derivatives are required mainly because it is not possible to take analytical derivatives of the dynamic equations due to the presence of the control selection process described in Section 5.3. The use of numerical derivatives also ensures that it is possible to use non-analytic formulations of the dynamic system's behavior (e.g., high-fidelity CFD aerodynamic tables). For the test cases described in later Sections, finite difference derivatives were used.

$$f'(x) \approx \frac{\mathcal{I}\big(f(x+ih)\big)}{h} \tag{5.11}$$

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \tag{5.12}$$

The dynamic equations of the states, costates, and the sensitivity matrix are written as C++ source code and compiled using NVIDIA's *nvcc* compiler into Parallel Thread Execution (PTX) or CUDA Binary (CUBIN) format. In later runs of the optimization process, these files can be directly loaded on the GPU. The boundary conditions are written as MATLAB source files and are compiled into MEX binary files for faster processing.

5.5 Dynamic Scaling

The different states and costates in the MPBVP resulting from using indirect methods can vary from each other by several orders of magnitude. This presents a problem while solving the MPBVP using numerical methods. For example, it may be impractical to enforce a tight error tolerance (e.g., 10^{-10}), on a state that has values on the order of 10^{9}. In order to mitigate this issue, the states, costates, constants, parameters, constraints, and the independent variable (time) are dynamically scaled during every iteration of the continuation method. It is generally difficult to identify

scaling factors for all these parameters for complex, hypersonic problems. By starting with a simple problem and evolving it into more complex problems (as discussed later in Sections 5.6 and 5.7), it is possible to evolve the scaling factors based on the solution history of the past iterations during the continuation process. This scaling methodology is fully automated and the designer only has to specify the scaling factor associated with each of the fundamental units. Dynamic scaling is key to enabling solution convergence in the test cases described in Chapter 6.

5.6 Construction of the Initial Guess

In order to seed the continuation process, it is necessary to supply it with a relatively simple optimization problem as an initial guess. Since the objective is to maximize the velocity at impact, a simple trajectory that can be solved is one that flies nearly straight down from the assumed post-boost staging condition. The optimal trajectory for reaching a target almost directly underneath the staging location, with maximum velocity, would be a near-ballistic trajectory that minimizes the drag coefficient of the vehicle. Hence, the Allen and Eggers trajectory solution [57] for ballistic trajectories can be used to construct a high quality initial guess to this optimization problem.

Assuming that drag forces dominate and a nearly-constant flight path angle for the steep trajectory, the closed form expression for velocity as a function of altitude can be obtained as shown in Eq. (5.13) where V_0 is the post-boost initial velocity of the weapon. The latitude and longitude can be reasonably assumed to be linearly varying and the assumption of a constant flight path angle of -80° prevents singularities in the equations of motion.

$$V = V_0 \exp[Ce^{(-h/h_{scale})}], \text{where } C = \frac{\rho_0 h_{scale}}{2\beta sin(\gamma)} \tag{5.13}$$

The costates for this initial trajectory can be constructed by reverse integration from the terminal point. From optimal control theory, the costate corresponding

to velocity can be computed using Eq. (5.14) when maximizing the velocity at the terminal point. The costates for flight-path angle, latitude, longitude and azimuth are chosen to be zero. The costate for altitude can be computed by equating the expression for the Hamiltonian at the terminal point to zero and solving for the costate.

$$\lambda_{v,f} = -2v_f \tag{5.14}$$

5.7 Continuation Process

This initial guess trajectory, being very close to the optimum, rapidly converges to a solution. Starting with this solution, the targeted location is moved until it matches the desired terminal conditions as shown in Figs. 5.1 and 5.2. At the end of this process, a maximum terminal velocity trajectory connecting the post-boost staging location and the targeted impact location is obtained. In the following examples, this trajectory is used as the initial guess to construct more complicated, highly constrained trajectories for a range of scenarios.

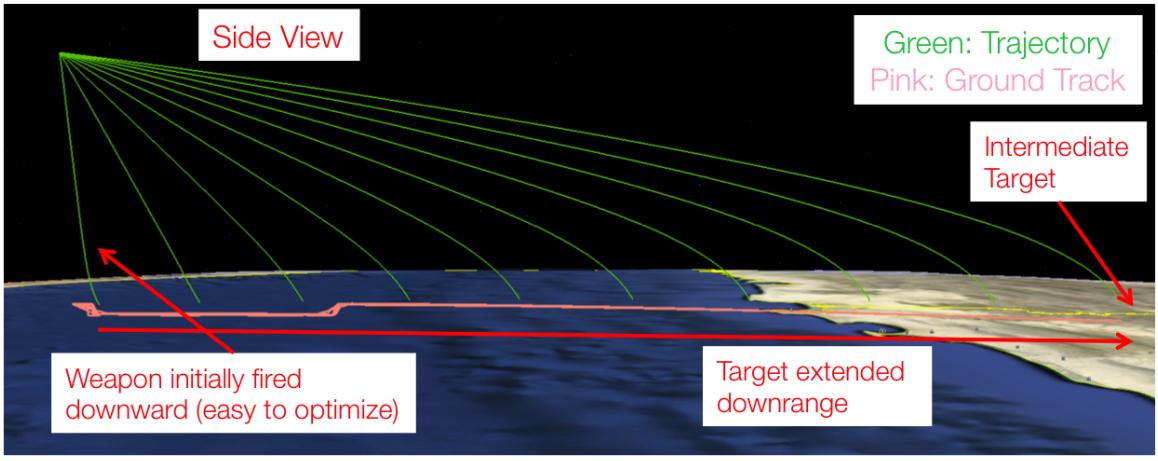

Figure 5.1. Side view of initial part of continuation process

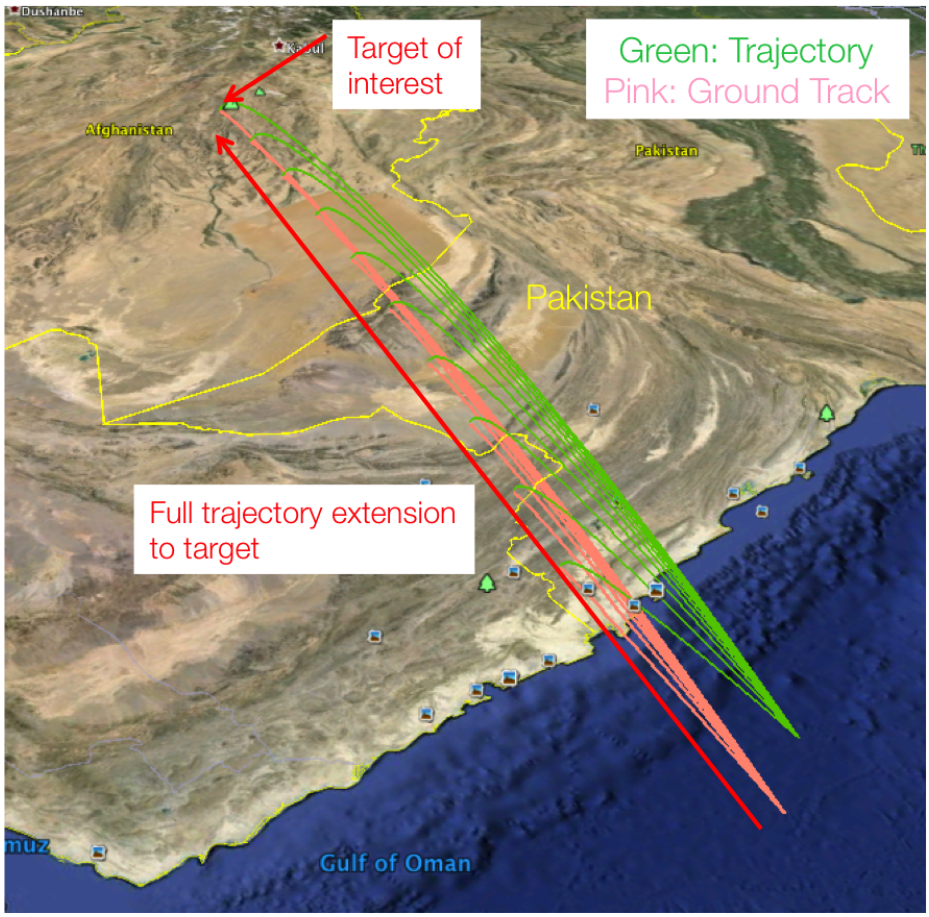

Figure 5.2. Full extension of trajectory to final target

6. TEST SCENARIOS AND BENCHMARKS

The hypersonic trajectory optimization problem defined in Chapter 5.1 is used to construct a number of hypothetical mission scenarios in the following sections, to demonstrate the application of the GPU-accelerated indirect solver for generating high quality optimal trajectories. The runtime performance of the solver is also compared to that of MATLAB's *bvp4c* for the same test cases. The examples described below demonstrate that it is possible to rapidly construct high quality, optimal, hypersonic trajectories using indirect methods.

6.1 Impact Geometry Constraints

This hypothetical example demonstrates user-specified constraints at the point of impact of the weapon in terms of position, flight path angle, and azimuth. These impact geometry constraints are dependent on the terrain and orientation of the cave entrance. In order to illustrate multiple impact geometries, trajectory solutions are computed for a range of terminal azimuth directions. The full sweep of optimal trajectories for different impact directions are shown in Fig. 6.1. Note that the trajectories are clustered together for the majority of the trajectory and do not separate until the weapon is very close to the target, as shown in Fig. 6.2.

In this test case, the terminal heading (ψ) is constrained to be a range of different values to showcase impacts on south-facing to north-facing targets. It can be seen from Fig. 6.2 that when the target is facing towards the shore, the trajectory involves flying straight to the target. As the target faces more and more west, the vehicle performs a banked dive to the target. When the target is facing directly opposite to the shore, the vehicle can be seen to fly directly over the target at around 6 km altitude before turning around. All these optimal trajectories maximize the velocity of impact, while

satisfying the impact geometry constraints. The entire process, including solving all the 124 intermediate trajectories, was completed in approximately 24 seconds using the GPU solver. No prior insight into the structure of the optimal solution was required in order to construct these complicated trajectories.

Fig. 6.3 illustrates the penalty in terminal velocity from performing aggressive maneuvers required to satisfy the impact geometry constraints. The banked dives and the increase in angle of attack required to perform the aggressive maneuvers can also be seen in the control history. It can also be seen that the very first trajectory which involves a direct flight to the target does not require any complex maneuvers. The corresponding well-behaved costates from the optimization problem are shown in Fig. 6.4. From optimal control theory, the costate corresponding to the longitude can be shown to be constant for this problem, since the equations of motion are independent of longitude. This is illustrated in the plot for λ_θ in Fig. 6.4.

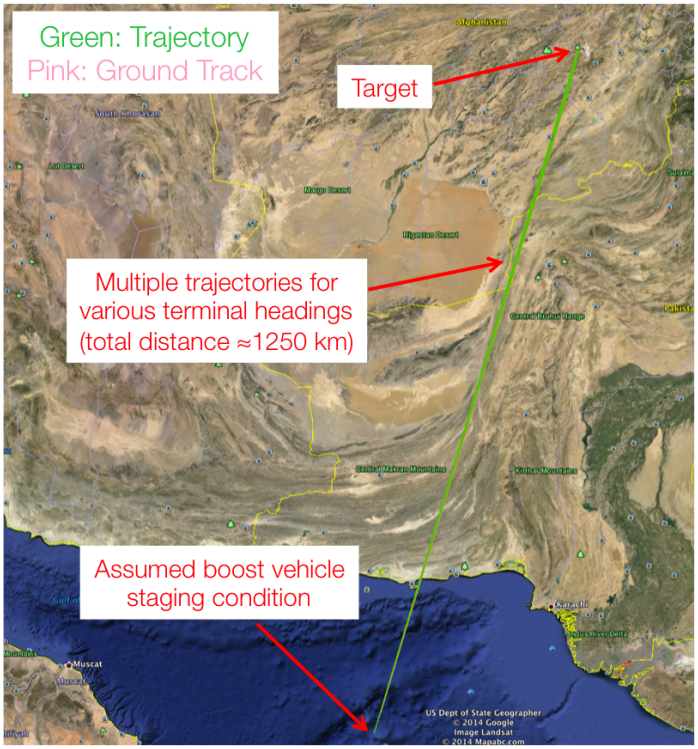

Figure 6.1. Impact geometry constraints – Overview

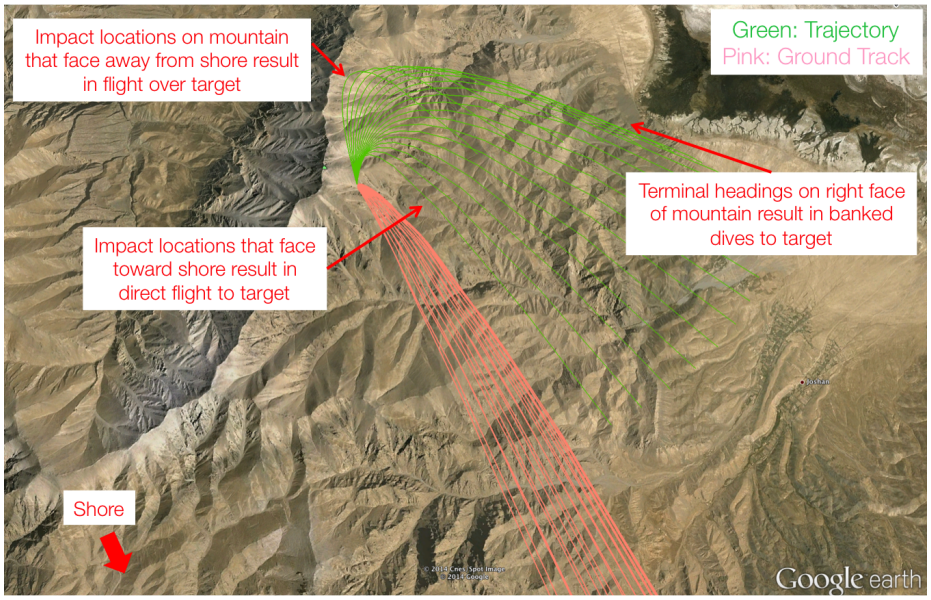

Figure 6.2. Impact geometry constraints – Pre-impact view

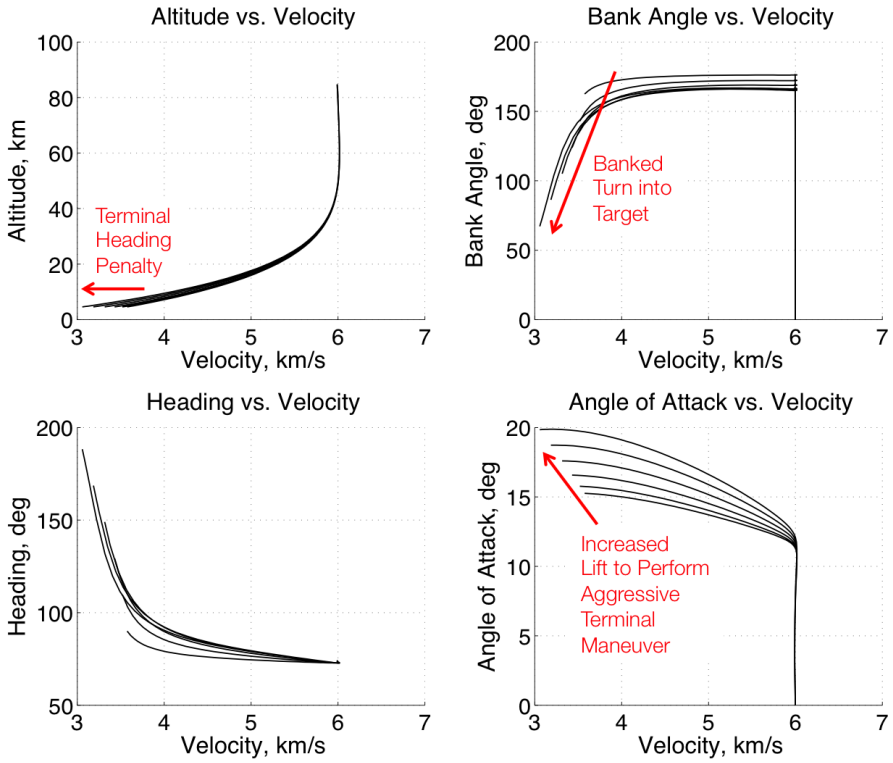

Figure 6.3. Impact geometry constraints – Trajectory and control history

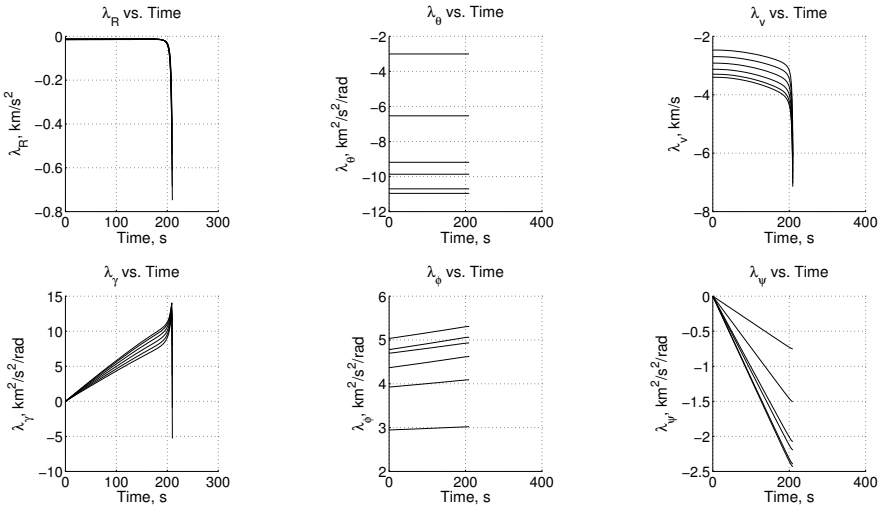

Figure 6.4. Impact geometry constraints – Costates

6.2 Post-Boost Geometry Constraints

In this example, maximum terminal velocity trajectories are constructed for a range of constrained initial headings as shown in Fig. 6.5. The terminal heading is left unconstrained. This essentially means that the weapon is not launched directly towards the target. This is also a pre-cursor to enforcing country overflight constraints where the weapon cannot be launched directly toward the target. The only impact geometry constraint is a terminal flight path angle constraint of -60° that forces a steep approach angle.

Launching the weapon in a direction away from the target necessitates the use of a skip maneuver in order to maximize the terminal velocity, as shown in Fig. 6.6 and Fig. 6.7. The further the initial heading is directed away from the target, the deeper the weapon has to dive into the atmosphere to obtain sufficient lift to perform the aggressive turn maneuver. This will also help loft the vehicle to a higher altitude, which is required to minimize drag and maximize the impact velocity. It can also be seen that there is a penalty in terminal velocity when the vehicle is directed further away

from the target. The costates corresponding to the trajectory are shown in Fig. 6.8. Since the terminal heading is no longer constrained, the costate corresponding to the azimuth, λ_ψ, can be seen to be equal to zero at the terminal point, as required by optimal control theory. This shows that the trajectories obtained are indeed optimal. The final trajectory obtained is extremely different from the simple starting trajectory, and the solver computes these results without requiring any insight into the structure of the final trajectory. The entire process, including computation of all the intermediate trajectories was completed in 50 seconds on the GPU solver.

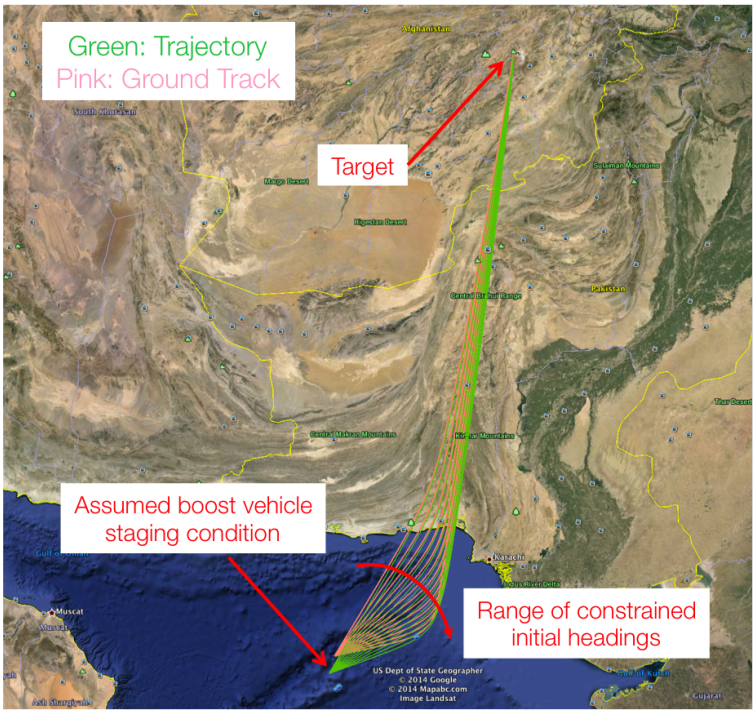

Figure 6.5. Post-boost geometry constraints – Overview

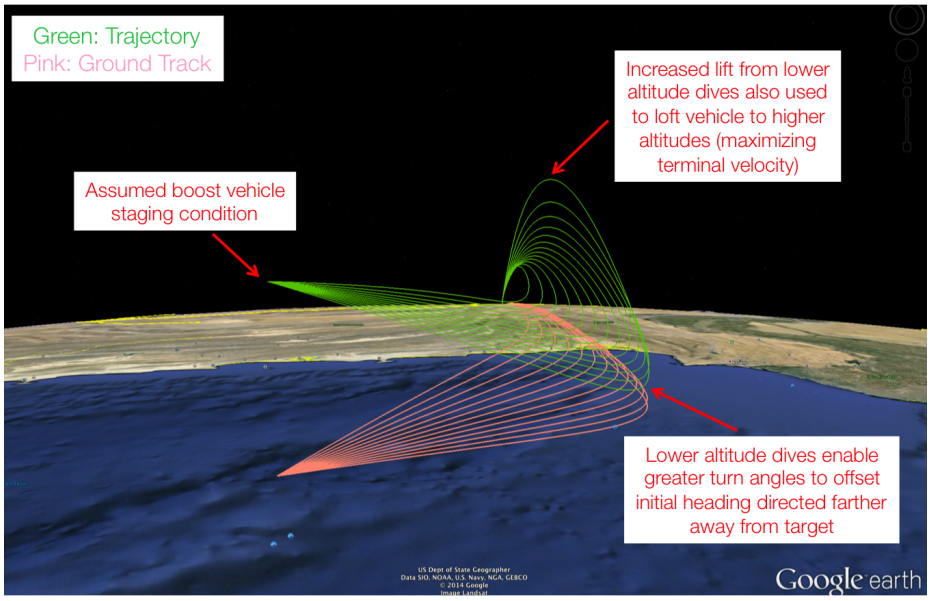

Figure 6.6. Post-boost geometry constraints – Dive and loft maneuver

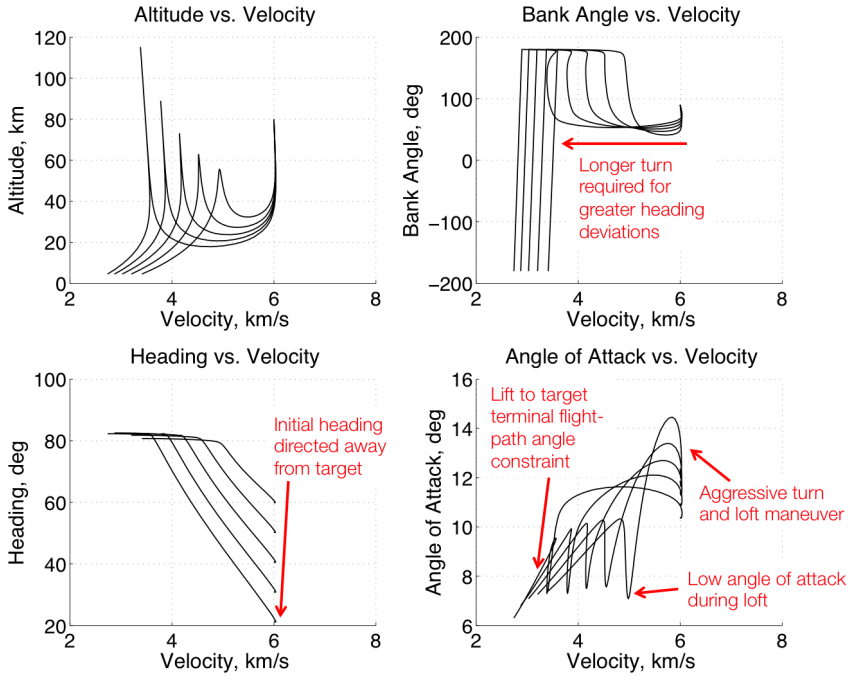

Figure 6.7. Post-boost geometry constraints – Trajectory and control history

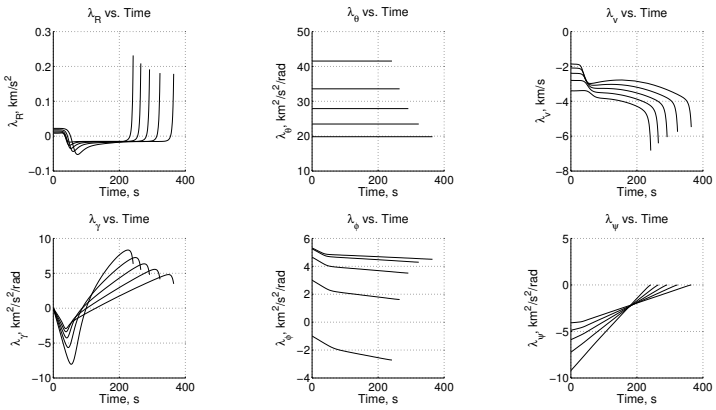

Figure 6.8. Post-boost geometry constraints – Costates

6.3 Stagnation Heat Rate Constraint

While the previous examples demonstrated the computation of trajectories with constraints at the initial and terminal points, this test case shows the construction of trajectories with a path constraint on stagnation heat rate. Path constraints were traditionally considered to be very difficult to implement using indirect methods due to the presence of jump conditions in costates (Fig. 6.11) which are difficult to guess a-priori. The use of continuation allows the constraints to be introduced in an automated manner, without requiring any prior insight into the structure of the solution.

The process starts with the trajectory from the previous example with a constraint on the initial heading. The path constraint is added by splitting the trajectory at the point of peak heating, introducing a short constrained arc at this location and rapidly reducing this constraint in a series of continuation steps until it reaches the desired value. The effect of enforcing the constraint on the optimal solution is shown in Fig. 6.9 and Fig. 6.10. It can be seen that constraining the heat rate to a lower value causes the vehicle to fly at a higher altitude. This reduces the amount of lift generated and the control authority available, which then leads to delaying the turn towards the target. The trajectory also consists of a penalty in terminal velocity

as the heat rate constraint becomes more strict. The initial angle of attack is also increased to ensure that the stagnation heat rate constraint is satisfied.

The Sutton and Graves [58] convective heating model is used for calculating the stagnation-point heat rate as shown in Eq. (6.1). Since it is a function of radial position and velocity, the corresponding costates will have discontinuities at the entry point of the constrained arc as shown in Fig. 6.11. These jump conditions are also satisfied with high precision to ensure that the trajectories are optimal and of high quality. The entire process is completed on the GPU solver in 51 seconds.

$$\dot{q} = k\sqrt{\frac{\rho}{r_n}}V^3 \tag{6.1}$$

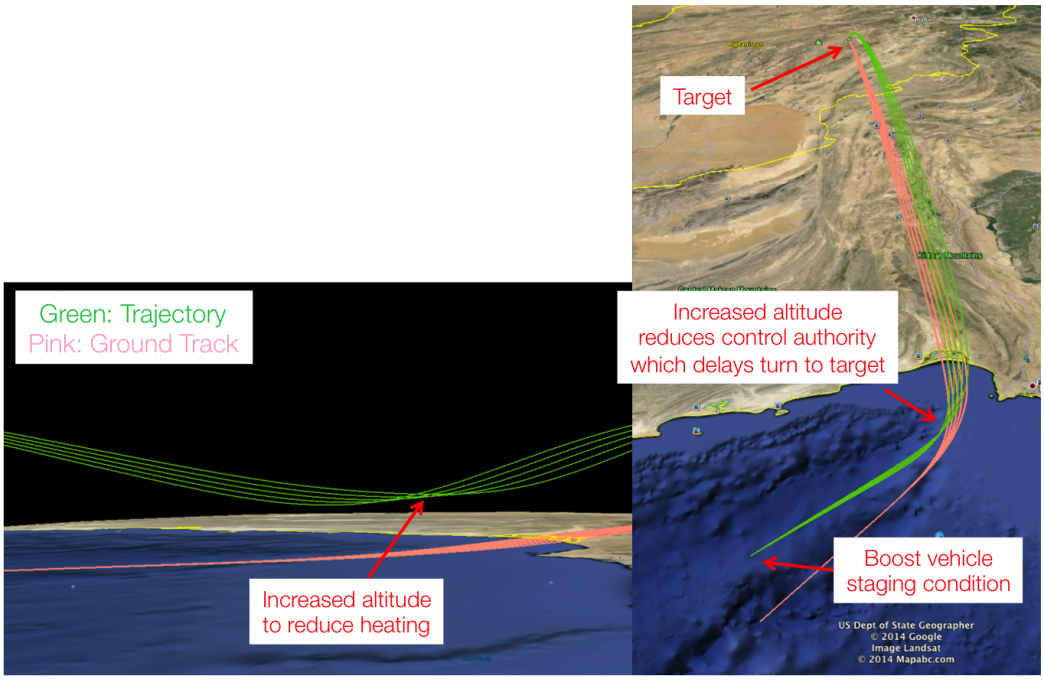

Figure 6.9. Stagnation heat rate – Increased altitude and delayed turn maneuver

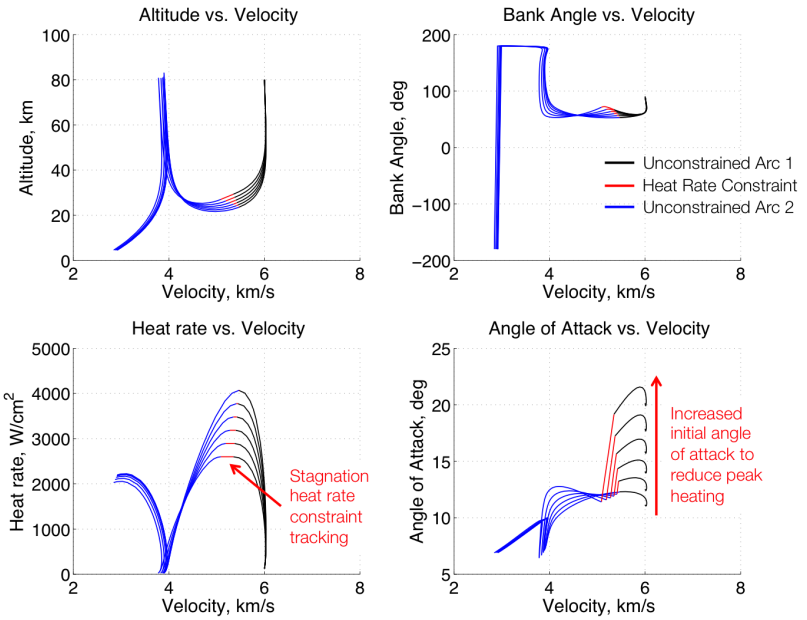

Figure 6.10. Stagnation heat rate – Trajectory and control history

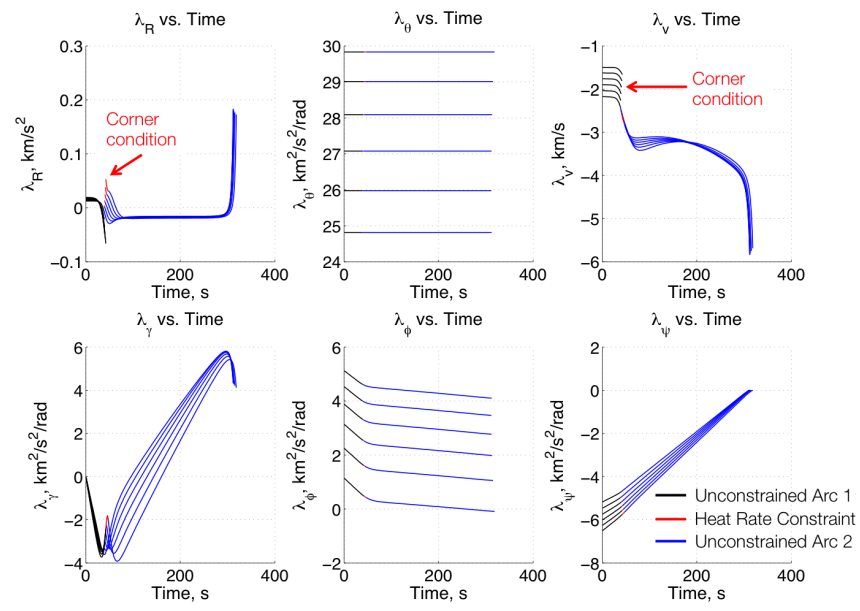

Figure 6.11. Stagnation heat rate – Costates

6.4 Country Overflight Constraint

This scenario involves constructing a maximum terminal velocity trajectory that does not overfly Pakistan. This is implemented by introducing an interior point constraint (or a waypoint) into the trajectory. A path constraint is not suitable in this case since the vehicle (which has limited lift) may not be able to track a complex border geometry. In order to compute this trajectory, first, an optimal trajectory connecting the post-boost staging location and the target is constructed that satisfies the terminal geometry constraints. Then the way point is introduced and the trajectory is "pushed" outside of Pakistan in a rapid continuation process as shown in Figs. 6.12 and 6.13. This process of introducing the waypoint and incrementally changing it, is done in a completely automated manner, transparent to the designer. The presence of a terminal heading constraint forces the vehicle to fly in complicated "corkscrew" trajectory at the end as shown in Fig. 6.12.

A subset of the optimal solutions from the family of optimal trajectories are shown in Fig. 6.14, split into pre-waypoint (Unconstrained Arc 1) and post-waypoint (Unconstrained Arc 2) trajectories. The transition point of these two sets of arcs represent the waypoint. It can be seen that the position of the waypoint occurs right after a dive, which is required in order to ensure that the vehicle has enough control authority to make the aggressive turn maneuver. This dive also enables the vehicle to loft up to minimize drag, and therefore maximizing the terminal velocity. These two opposing factors are balanced with high precision to ensure that the vehicle tracks the overflight constraint while also maximizing the terminal velocity. It can also be observed that, as expected, the terminal velocity takes a penalty as the vehicle is forced to fly further out from its target. The corresponding costates can be seen in Fig. 6.15. Since the country overflight constraint is expressed as a function of latitude and longitude, a corner condition in the costates corresponding to these states ($\lambda\theta$ and $\lambda\phi$) can be observed as well. This entire solution process was completed in 85 seconds using the GPU solver.

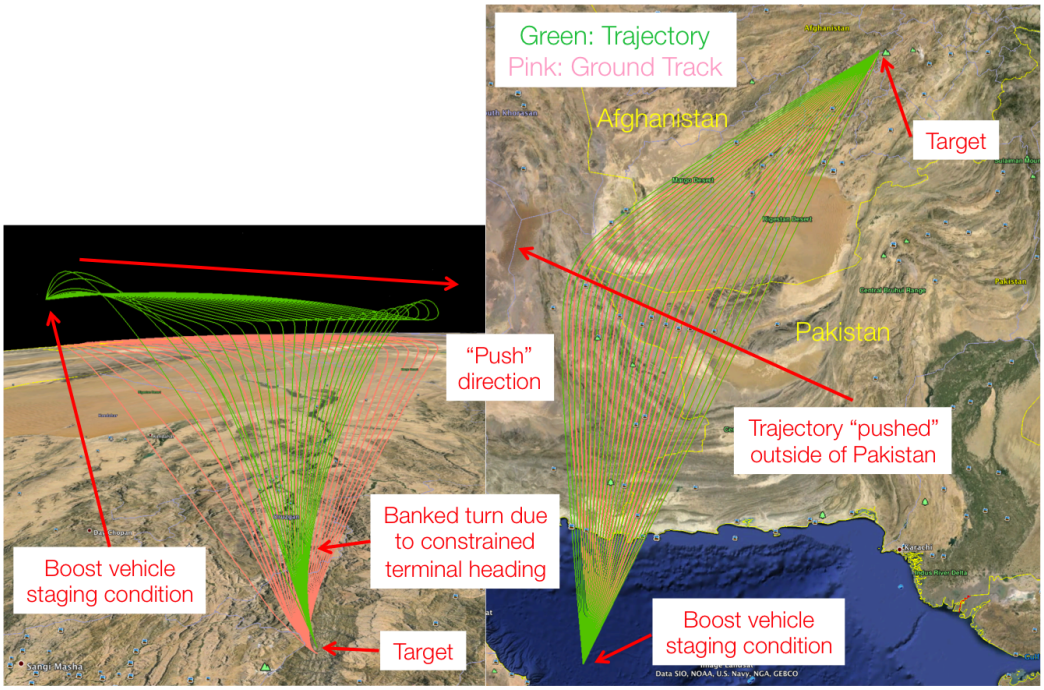

Figure 6.12. Country overflight constraint

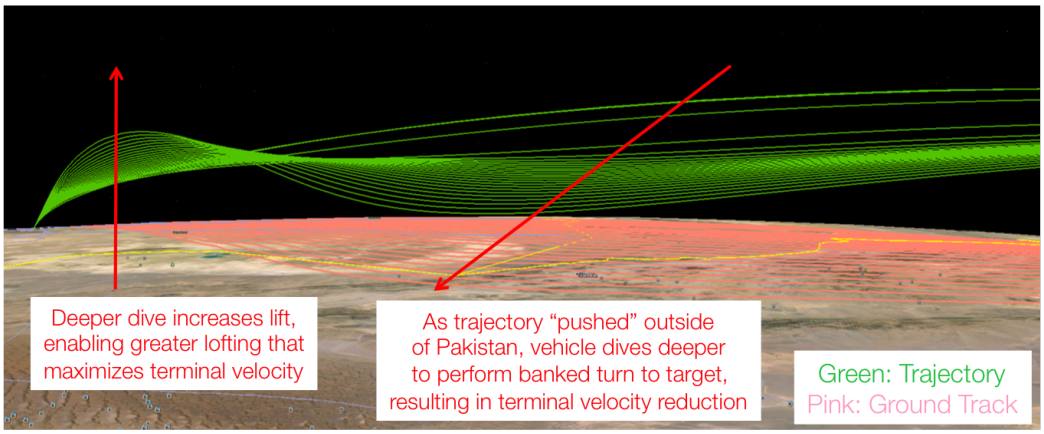

Figure 6.13. Country overflight constraint – Turn and loft maneuver

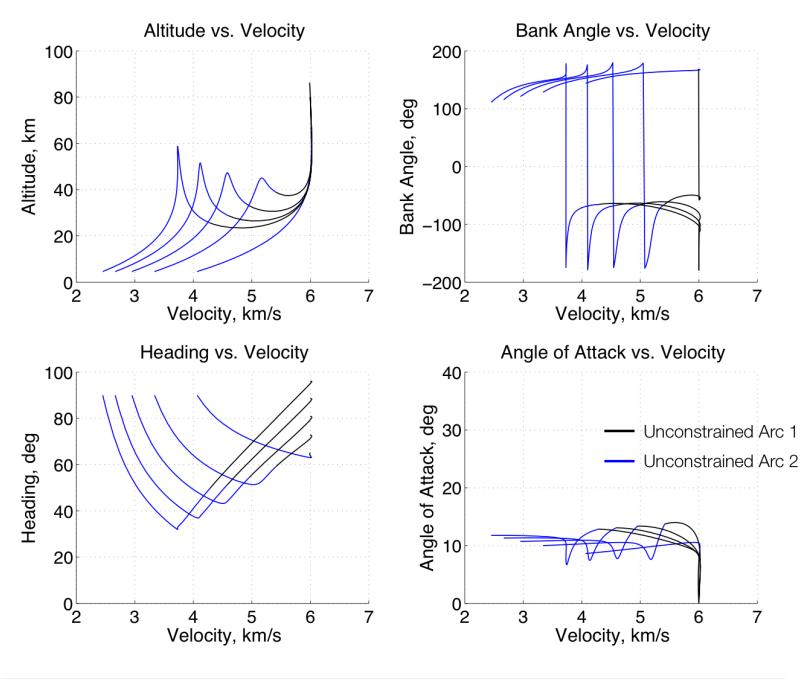

Figure 6.14. Country overflight constraint – Trajectory and control history

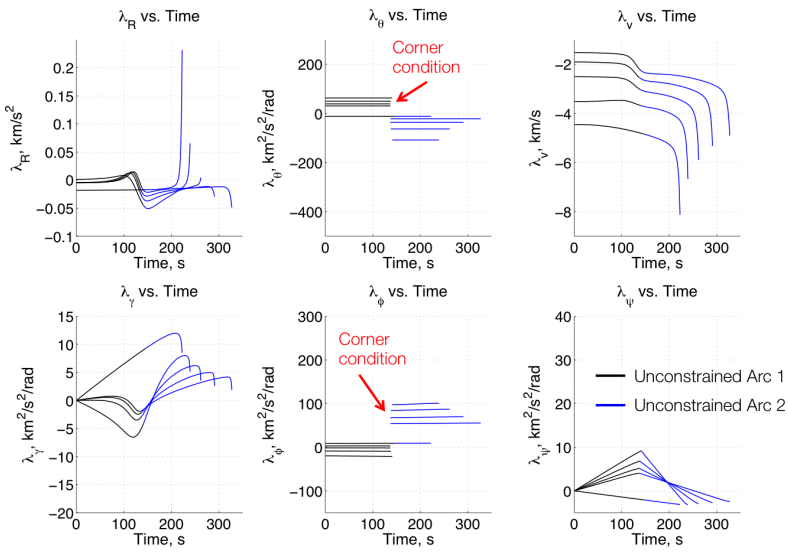

Figure 6.15. Country overflight constraint – Costates

6.5 Simultaneous Constraints

The previous examples illustrated the rapid construction of families of trajectories suitable for a wide range of mission scenarios. The ability to construct such families of trajectories are very useful when conducting trade studies. In contrast, this example shows a single, complex trajectory that satisfies all the required mission constraints. It represents a combination of multiple constraints that results in a very complex optimal trajectory. In this hypothetical example, the weapon is initially directed at a hypothetical target in Pakistan as shown in Fig. 6.16. However, the weapon is re-designated to a different target during the boost-phase, essentially constraining the post-boost geometry to that required for the original target. A maximum velocity trajectory for the new target is constructed, simultaneously satisfying constraints in initial heading and velocity, stagnation heat rate, contested airspace avoidance zones, and impact flight-path angle. The final optimized trajectory and its control history shown in Figures 6.16 and 6.17 and were constructed in 75 seconds using the GPU solver.

The final "S" trajectory requires sufficient control authority at two specific points along the trajectory in order to perform the corresponding turns. These dives and the subsequent lofts can be seen in Fig. 6.17. The two dives are timed precisely so as to have enough lift to perform the maneuvers while still maximizing the velocity at the target. The indirect method solves for the complex control histories (Fig. 6.17) and corner conditions (Fig. 6.18) with high precision. Fig. 6.19 shows a zoomed in view of the costates at the initial part of the trajectory to illustrate the corner conditions at the entry point of the stagnation heat rate constraint. Even in such a complex and highly constrained trajectory, the necessary conditions of optimality are fully satisfied, resulting in a high quality solution.

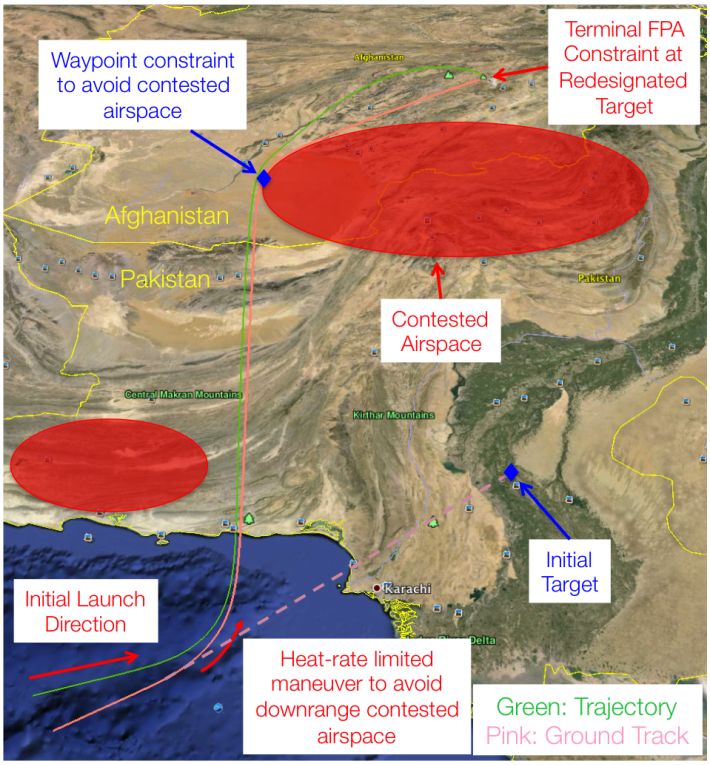

Figure 6.16. Simultaneous constraints example – Final trajectory solution

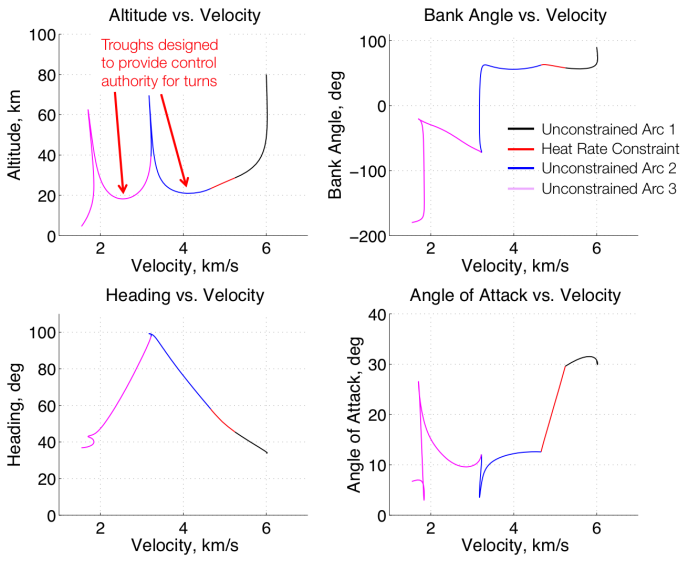

Figure 6.17. Simultaneous constraints example – Trajectory and control history

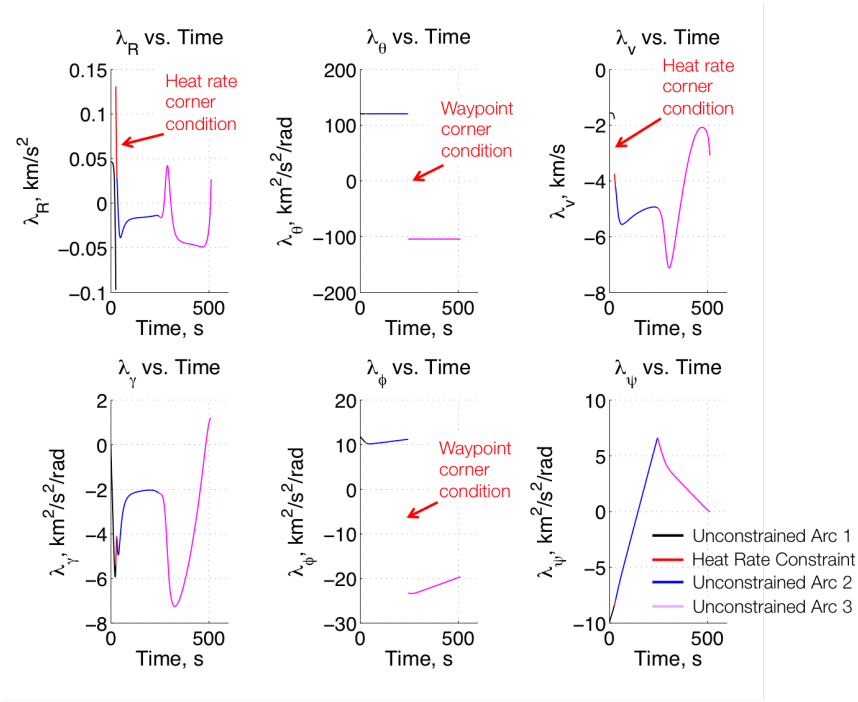

Figure 6.18. Simultaneous constraints example – Costates

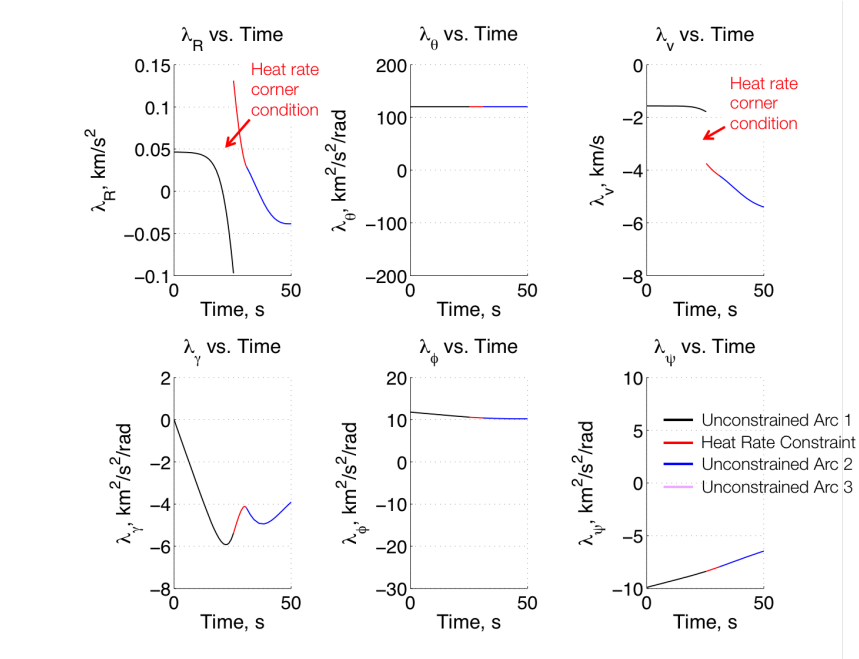

Figure 6.19. Simultaneous constraints example – Costates at initial portion of trajectory

6.6 Benchmarks

All of the example cases were solved using both MATLAB's *bvp4c* solver as well as the GPU-based multiple shooting solver (*bvpgpu*) built by the author. As discussed in Chapters 2 and 4, the multiple shooting method has many features that lends itself to being suitable for implementation on the GPU. The nature of the sensitivity matrix used in the multiple shooting method allows the splitting of the trajectory into smaller segments which can then be computed independently of each other. This is one of the major advantages of this method over *bvp4c*. *bvp4c* uses a collocation method to solve the boundary value problem. It is a purely CPU-based algorithm that does not utilize any kind of parallel processing.

The runtimes for solving the different test scenarios using the two solvers are compared in Fig. 6.20. It can been seen that at present, the GPU accelerated shooting solver offers a 2x-4x speedup over MATLAB's *bvp4c* in the given scenarios.

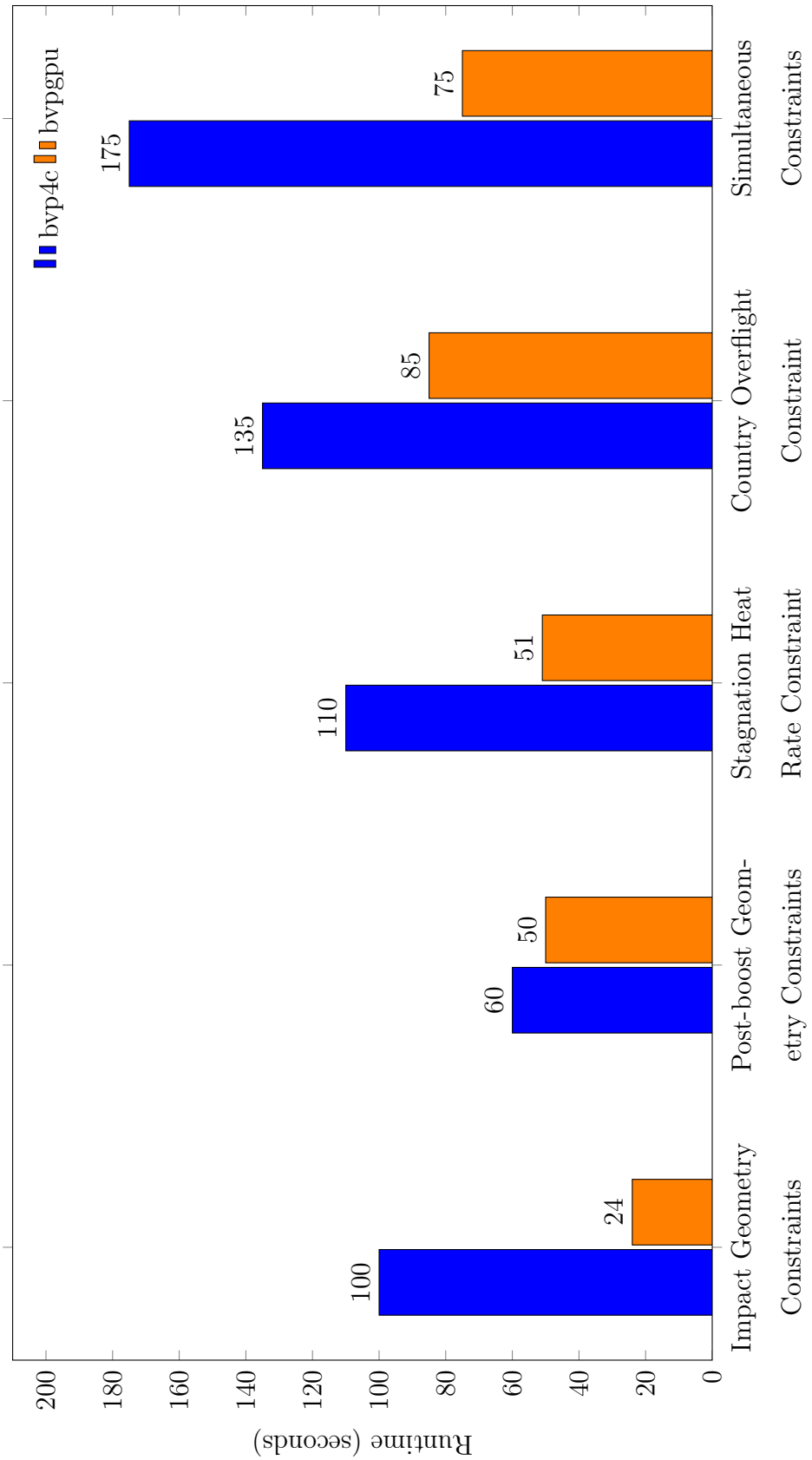

Figure 6.20. Benchmarks – *bvp4c* vs. *bvpgpu*

7. SUMMARY

In this study, a highly parallelized indirect optimization strategy for the rapid design of optimal trajectories is developed. The multiple shooting method is used to develop a custom algorithm that executes very efficiently on a highly parallel GPU computational architecture. It is demonstrated that indirect optimization methods can be used to rapidly solve complex optimization problems by utilizing the GPU-accelerated multiple shooting method. The various algorithmic optimizations that help maximize GPU performance by accounting for GPU processor occupancy, memory access coalescing, and parallel matrix operations are examined. It is seen that the data structures used to store information, as well as the manner in which the problem is structured is crucial to obtaining optimum performance on an GPU. The methodology is demonstrated to provide considerable speedups over using a CPU-based solution method such as MATLAB's *bvp4c* to solve trajectory optimization problems.

The design of maximum terminal velocity trajectories using a hypothetical long range weapon system is used to showcase the different kinds of trajectory problems that can be solved using this methodology. An efficient strategy for constructing an initial guess using an analytic ballistic trajectory is illustrated. Various combinations of initial, terminal, interior point and path constraints are enforced on the trajectory to demonstrate the design of complex hypersonic trajectories. These constraints are enforced in an automated manner, completely transparently to the designer, without any a-priori knowledge about the structure of the final solution. Pontryagin's Minimum Principle is used to evaluate multiple control options at every point along the trajectory to ensure optimality of the final solution. The necessary conditions of optimality, which includes complicated corner conditions in the non-physical costates, are fully satisfied to a high precision in these complex trajectory solutions. The examples

demonstrated that the GPU-accelerated solver is capable of rapidly constructing high quality, optimal hypersonic trajectories using indirect methods that were previously considered impractical.

The current trend in high performance computing technology is towards more parallel architectures, such as multi-core processors and GPUs, rather than monolithic, powerful processors. The work illustrated here demonstrates that indirect methods have characteristics that make them very well suited for implementation on these emerging computational architectures, enabling rapid, automated construction of high quality optimal trajectories.

8. FUTURE WORK

There were specific parts of the shooting algorithm that could be modified (e.g., by splitting the trajectory into segments) to enhance its performance on the GPU. While these modifications helped parallelize the algorithm to a certain extent, the GPU is still not utilized to its full potential. There are definitely other modifications to the algorithm that can be done to improve the performance further and increase GPU processor occupancy. Further segmenting the trajectory into separate arcs beyond what is dictated by optimal control theory may help make the problem more parallel and efficient by allowing parallel propagation of the equations of motion. This will also make the problem less sensitive, allowing it to converge to the final solution in fewer continuation steps.

Solving problems with a large number of states such as multi-vehicle or swarm problems, as well as the use of higher fidelity 6DOF models will probably lead to the saturation of the GPU processors. In this case, it will also be advantageous to modify the solver to make use of multiple GPUs.

The current solver uses a fixed-step RK4 integrator for propagating the trajectories. This comes with the limitation that we are forced to manually pick a step-size which may not be ideal for all parts of the trajectory. Using an adaptive stepping approach will help the solver adapt according to the sensitivity of the problem.

There may be numerical methods for solving boundary value problems, other than the multiple shooting method that are more suited for implementation on the GPU. One example is the Modified Chebyshev-Picard Iteration (MCPI) method [59–61]. In this method, the trajectory is modeled using Chebyshev polynomial orthogonal basis functions and an iteration process based on the Picard-Lindelöf theorem [62] is used to converge towards the solution. Ref. 60 also demonstrates an NVIDIA CUDA implementation of this algorithm, showing considerable speedups over the

CPU variant. Various test cases also show that the convergence properties of the MCPI method are independent of the initial guess. However, the MCPI method was formulated for use with BVPs with specific kinds of boundary conditions. A more generalized form of the method is yet to be formulated. Similar methods have been applied to astrodynamics problems [63–65], but they are yet to be used for hypersonic trajectory optimization. The inherent parallel nature of a generalized MCPI method may make it a good candidate for inclusion in the rapid trajectory optimization framework.

LIST OF REFERENCES

LIST OF REFERENCES

[1] John T. Betts. Survey of numerical methods for trajectory optimization. *Journal of Guidance, Control, and Dynamics*, 21(2):193–207, 1998.

[2] Charles R. Hargraves and Stephen W. Paris. Direct trajectory optimization using nonlinear programming and collocation. *Journal of Guidance, Control, and Dynamics*, 10(4):338–342, 1987.

[3] Albert L. Herman and Bruce A. Conway. Direct optimization using collocation based on high-order gauss-lobatto quadrature rules. *Journal of Guidance, Control, and Dynamics*, 19(3):592–599, 1996.

[4] Robert Bibeau and D. Rubenstein. Trajectory optimization for a fixed-trim reentry vehicle using direct collocation and nonlinear programming. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2000.

[5] Michael J. Grant and Gavin F. Mendeck. Mars science laboratory entry optimization using particle swarm methodology. In *Proceedings of AIAA Atmospheric Flight Mechanics Conference and Exhibit, Hilton Head, South Carolina*, 2007.

[6] J.T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. Advances in Design and Control. Society for Industrial and Applied Mathematics, 2001.

[7] Michael A. Patterson and Anil V. Rao. GPOPS- II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp–Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming. *ACM Transactions on Mathematical Software*, 39(3):1–41, 2013.

[8] I. Michael Ross. A beginners guide to DIDO: a MATLAB application package for solving optimal control problems. 2007.

[9] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM journal on optimization*, 12(4):979–1006, 2002.

[10] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[11] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. *ACM SIGARCH Computer Architecture News*, 28(2):248–259, May 2000.

[12] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[13] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[14] D. Luebke, M. Harris, and N. Govindaraju. GPGPU: general-purpose computation on graphics hardware. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[15] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[16] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55. ACM, 2001.

[17] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101. Eurographics Association, 2003.

[18] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM, 2003.

[19] Jesse D. Hall, Nathan A. Carr, and John C. Hart. Cache and bandwidth aware matrix multiplication on the GPU. 2003.

[20] NVIDIA Corporation. Cuda C Programming Guide, 2014.

[21] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[22] John E. Stone, David J. Hardy, Ivan S. Ufimtsev, and Klaus Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29(2):116–125, 2010.

[23] Dimitri Komatitsch, Gordon Erlebacher, Dominik Göddeke, and David Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229(20):7692–7714, 2010.

[24] In Kyu Park, Nitin Singhal, Man Hee Lee, Sungdae Cho, and Chris W. Kim. Design and performance evaluation of image processing algorithms on GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):91–104, 2011.

[25] Y.L. Zhu, H. Liu, and Z.W. Li. GALAMOST : GPU-accelerated large-scale molecular simulation toolkit. *Journal of Computational Chemistry*, 34(25):2197–2211, 2013.

[26] P.K. Menon, Monish Tandale, and Sandy Wiraatmadja. Faster Simulations of the National Airspace System. In *NVIDIA GPU Technology Conference*, San Jose,CA, 2010.

[27] L. Lopez-Diaz, D. Aurelio, L. Torres, E. Martinez, M.A. Hernandez-Lopez, J. Gomez, O. Alejos, M. Carpentieri, G. Finocchio, and G. Consolo. Micromagnetic simulations using graphics processing units. *Journal of Physics D: Applied Physics*, 45(32):323001, 2012.

[28] Carlos Cuevas, Daniel Berjón, Francisco Morán, and Narciso García. Moving object detection for real-time augmented reality applications in a GPGPU. *Consumer Electronics, IEEE Transactions on*, 58(1):117–125, 2012.

[29] Takashi Amada, Masataka Imura, Yoshihiro Yasumuro, Yoshitsugu Manabe, and Kunihiro Chihara. Particle-based fluid simulation on GPU. In *ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH*, 2004.

[30] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V. Le, and Andrew Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272, 2011.

[31] Sarod Yatawatta, Sanaz Kazemi, and Saleem Zaroubi. GPU accelerated nonlinear optimization in radio interferometric calibration. In *Innovative Parallel Computing (InPar), 2012*, pages 1–6. IEEE, 2012.

[32] Yun Fei, Guodong Rong, Bin Wang, and Wenping Wang. Parallel L-BFGS-B algorithm on GPU. *Computers & Graphics*, 40:1–9, 2014.

[33] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the cuda architecture. *Information Sciences*, 181(20):4642–4657, 2011.

[34] You Zhou and Ying Tan. Gpu-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 1493–1500. IEEE, 2009.

[35] Hao Chen, Nicholas S. Flann, and Daniel W. Watson. Parallel genetic simulated annealing: a massively parallel simd algorithm. *Parallel and Distributed Systems, IEEE Transactions on*, 9(2):126–136, 1998.

[36] Sifa Zhang and Zhenming He. Implementation of parallel genetic algorithm based on CUDA. In *Advances in Computation and Intelligence*, pages 24–30. Springer, 2009.

[37] Petr Pospíchal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the CUDA architecture. In *Applications of Evolutionary Computation*, pages 442–451. Springer, 2010.

[38] Michael J. Grant, Ian G. Clark, and Robert D. Braun. Rapid Simultaneous Hypersonic Aerodynamic and Trajectory Optimization Using Variational Methods. In *AIAA Atmospheric Flight Mechanics Conference and Exhibit, Portland, OR*, pages 8–11, 2011.

[39] A.E. Bryson and Y.C. Ho. *Applied Optimal Control – Optimization, Estimation, and Control*. Taylor & Francis, 1975.

[40] David D. Morrison, James D. Riley, and John F. Zancanaro. Multiple Shooting Method for Two-point Boundary Value Problems. *Communications of the ACM*, 5(12):613–614, Dec 1962.

[41] Josef Stoer and Roland Bulirsch. *Introduction to numerical analysis*, volume 12. Springer, 2002.

[42] Michael J. Grant. *Rapid Simultaneous Hypersonic Aerodynamic and Trajectory Optimization for Conceptual Design*. PhD thesis, Georgia Institute of Technology, 2012.

[43] Josef Stoer, Roland Bulirsch, R. Bartels, Walter Gautschi, and C. Witzgall. *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1993.

[44] Lawrence F. Shampine, Jacek Kierzenka, and Mark W. Reichelt. Solving boundary value problems for ordinary differential equations in MATLAB with bvp4c. 2000.

[45] Nitin Arora, Ryan P. Russell, and Richard W. Vuduc. Fast Sensitivity Computations for Trajectory Optimization. *Advances in the Astronautical Sciences*, 135(1):545–560, 2009.

[46] James Ferguson. A Brief Survey of the History of the Calculus of Variations and its Applications. *arXiv preprint math/0402357*, 2004.

[47] MIGUEL DE ICAZA HERRERA. Galileo, bernoulli, leibniz and newton around the brachistochrone problem. *Rev Mexicana Fis*, 40(3):459–475, 1994.

[48] Cornelius Lanczos. *The variational principles of mechanics*, volume 4. Courier Dover Publications, 1970.

[49] James M. Longuski, José J. Guzmán, and John E. Prussing. *Optimal Control with Aerospace Applications*. Springer, 2014.

[50] Derek F. Lawden. *Optimal trajectories for space navigation*. Butterworths, 1963.

[51] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture : FERMI, 2009.

[52] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture : Kepler GK110, 2012.

[53] NVIDIA Corporation. NVIDIA GeForce GTX 750Ti : Featuring First-Generation Maxwell GPU Technology, 2014.

[54] S Gunther and R Singhal. Next generation Intel microarchitecture (nehalem) family: Architectural insights and power management. In *Intel Developer Forum*, volume 165, 2008.

[55] Wolfram Research, Inc. Mathematica, 2012.

[56] Nguyen X. Vinh, A. Busemann, and R. D. Culp. *Hypersonic and Planetary Entry Flight Mechanics*. The University of Michigan Press, Ann Arbor, 1980.

[57] H. Julian Allen and Alfred J. Eggers. *A study of the motion and aerodynamic heating of missiles entering the earth's atmosphere at high supersonic speeds*. National Advisory Committee for Aeronautics, 1957.

[58] Kenneth Sutton and Randolph A. Graves Jr. A general stagnation-point convective heating equation for arbitrary gas mixtures. Technical report, National Aeronautics and Space Administration, 1971.

[59] Terry Feagin and Paul Nacozy. Matrix formulation of the Picard method for parallel computation. *Celestial mechanics*, 29(2):107–115, 1983.

[60] Xiaoli Bai. *Modified Chebyshev-Picard iteration methods for solution of initial value and boundary value problems*. PhD thesis, Texas A&M University, 2010.

[61] A. Probe, Brent Macomber, Donghoon Kim, R. Woollands, and John L. Junkins. Terminal Convergence Approximation Modified Chebyshev Picard Iteration for Efficient Numerical Integration of Orbital Trajectories. Advanced Maui Optical and Space Surveillance Technologies Conference, Maui, Hawaii, 2014.

[62] Earl A. Coddington and Norman Levinson. *Theory of ordinary differential equations*. Tata McGraw-Hill Education, 1955.

[63] Navid Nakhjiri and Benjamin Villac. Modified Picard Integrator for Spaceflight Mechanics. *Journal of Guidance, Control, and Dynamics*, 37(5):1625–1637, 2014.

[64] Xiaoli Bai and John L. Junkins. Modified Chebyshev-Picard iteration methods for station-keeping of translunar halo orbits. *Mathematical Problems in Engineering*, 2012, 2012.

[65] Robyn Woollands, Ahmad Bani Younes, Brent Macomber, Austin Probe, Donghoon Kim, and John L. Junkins. Validation of Accuracy and Efficiency of Long-Arc Orbit Propagation Using the Method of Manufactured Solutions and the Round-Trip-Closure Method. 2014.