Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures

Thomas Antony* and Michael J. Grant†
Purdue University, West Lafayette, Indiana 47907-2045

DOI: 10.2514/1.A33755

A graphics-processing-units-accelerated indirect trajectory optimization methodology that uses the multiple shooting method and continuation is developed using the CUDA platform. The algorithm is designed to exploit the parallelism inherent in the indirect shooting method while maximizing computational efficiency. The resulting rapid optimal control framework enables the construction of high quality optimal trajectories that satisfy constraints and the necessary conditions of optimality. The performance of the framework is highlighted by construction of maximum terminal velocity trajectories for a hypothetical long-range weapon system. Various hypothetical mission scenarios that enforce different combinations of initial, terminal, interior point, and path constraints that demonstrate the rapid construction of complex trajectories are used to compare performance of the graphics-processing-units-accelerated solver to MATLAB®'s bvp4c. Trajectory problems of this kind were previously considered impractical to solve using indirect methods. The graphics-processing-units-accelerated solver is found to be two to four times faster than MATLAB®'s bvp4c for a small-dimensional system, even while running on graphics processing units hardware that is five years behind the state of the art.

Nomenclature

A_{ref}	=	reference area, m^2
C_D	=	coefficient of drag
C_L	=	coefficient of lift
H	=	scale height, m
h	=	altitude, m
L/D	=	lift-to-drag ratio
m	=	mass, kg
q	=	dynamic pressure, kg/m^2
R_E	=	radius of the earth, m
r_n	=	nose radius, m
V	=	velocity, m/s
α	=	angle of attack, rad
β	=	ballistic coefficient
γ	=	flight-path angle, rad
θ	=	latitude, rad
μ	=	standard gravitational parameter, m^3/s^2
ρ_0	=	atmospheric density at sea-level, kg/m^3
σ	=	bank angle, rad
ϕ	=	longitude, rad
Φ	=	state transition matrix
ψ	=	azimuth, rad
ω	=	sidereal rate of Earth, rad/s

I. 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 pseudospectral 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 pseudospectral methods that 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 nonlinear programming (NLP) solver such as SNOPT [9] and are very computationally intensive for large optimization problems [10] even with the exploitation of sparse Jacobian structures.

One factor that these approaches rarely consider is the computing platform on which they are implemented. While Moore's Law [11] 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 has essentially peaked [12]. Computing hardware is now transitioning toward highly parallel architectures rather than powerful monolithic processors [13,14].

Graphics processing units (GPUs) were originally designed to be used as dedicated processors for rendering three-dimensional graphics on computers. Efforts to exploit the GPUs for general-purpose computing applications have been underway since the early 2000s [15,16]. One of the earliest demonstrations of GPU computing was a matrix–matrix multiplication algorithm [17]. The availability of floating point operations on GPU hardware has allowed the implementation of more advanced computational methods on the GPU [18–20]. Over the last decade, the computing power of GPUs has grown exponentially [21,22] as compared to CPUs as shown in Figs. 1 and 2. The modern GPU is a highly capable parallel processor with peak arithmetic and memory bandwidth that substantially outpaces its CPU counterparts [23].

NVIDIA's 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 [24–31].

Direct methods for trajectory optimization, specifically pseudospectral and collocation methods, convert the trajectory optimization problem into a nonlinear programming problem that involves many sequential, iterative operations instead of large, independent parallel operations. Some of the attempts to implement NLP algorithms on GPUs have noted speedups of two to three times the speed of their CPU counterparts for applications in machine learning [32] and radio interferometry [33]. Another recent implementation is that of the L-BFGS-B nonlinear optimization algorithm [34] on a GPU, which

Presented as Paper 2016-0275 at the AIAA Atmospheric Flight Mechanics Conference, San Diego, CA, 4–8 January 2016; received 8 September 2016; revision received 12 March 2017; accepted for publication 8 April 2017; published online 19 June 2017.

*Graduate Research Assistant, School of Aeronautics and Astronautics; tantony@purdue.edu. Student Member AIAA.

†Assistant Professor, School of Aeronautics and Astronautics; mjgrant@purdue.edu. Senior Member AIAA.

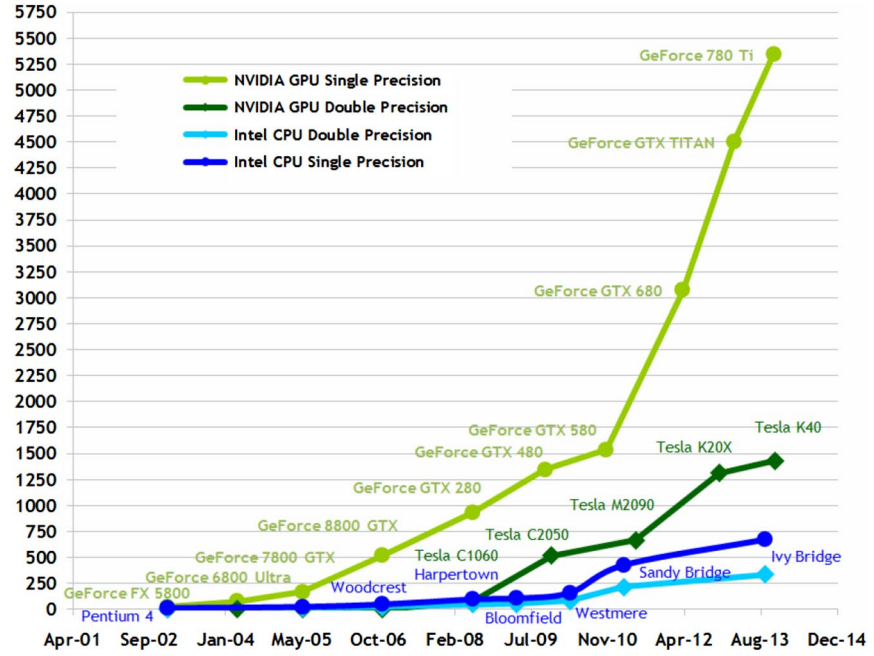


Fig. 1 CPU vs GPU performance trends by year — Theoretical giga floating point operations per second.

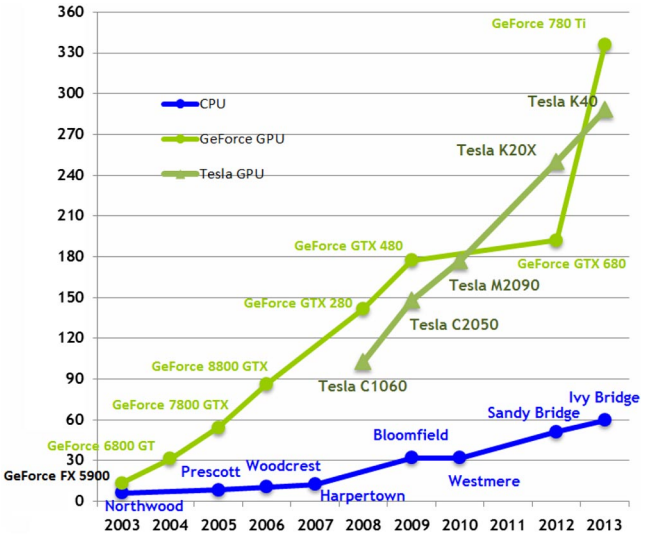


Fig. 2 CPU vs GPU performance trends by year — Theoretical memory bandwidth in GB/s.

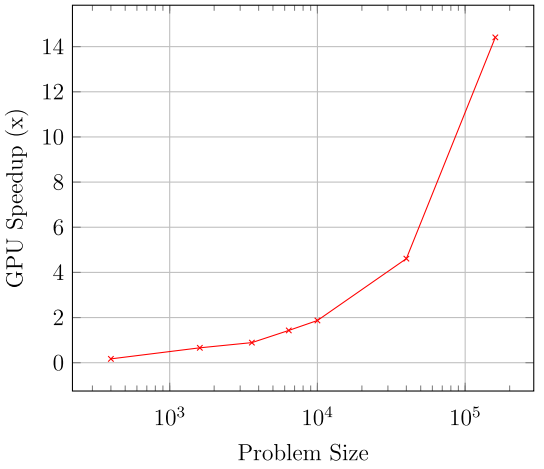


Fig. 3 Speedup of L-BFGS-B algorithm on GPU for the elastic-plastic torsion problem [34].

shows significant speedups, but for only extremely large optimization problems, with dimensions numbering in the millions. Fei et al. [34] 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. 3, the GPU does not start showing any benefits in speed for that algorithm until the NLP problem size reaches approximately 6400 states. Even with a problem with 40,000 states, the speedup can be observed to be only around five times faster.

Direct methods generally guarantee a solution at the expense of computational speed [1]. Most direct pseudospectral solvers use third-party NLP solvers such as SNOPT that have not yet been redesigned to exploit GPUs. As shown in the example in Fig. 3, there may be ways in which parts of an NLP solver, such as the evaluation of constraints or the Jacobian, can exploit the GPU, while the iterations themselves may remain sequential in nature. The NLP problems resulting from the application of pseudospectral methods to trajectory problems usually have dimensions in the high hundreds or low thousands, which, at least in the previous example, does not seem to be enough to leverage the GPU. Even if certain direct methods such as direct shooting can probably leverage GPUs, at the time of writing, there are no GPU-enabled implementations of direct methods that can be used as a benchmark. This makes it difficult to make a fair comparison between a direct method and a GPU-enabled indirect method such as the one described in this work. Therefore, the goal of this study is not a comparison between direct and indirect methods but rather to show that indirect methods have features that make them capable of exploiting these emerging computational architectures to obtain faster performance. MATLAB®'s bvp4c [35] was selected as the CPU solver as its performance is a known quantity and because it is easy to setup. The bvp4c solver has also been optimized by Mathworks over the past decade, and any CPU-based solver that we implement from scratch may be suboptimal in comparison.

Indirect methods use calculus of variations to formulate the optimization problem as a boundary value problem (BVP) [36]. The BVPs can be solved using a suitable numerical solver, and the multiple shooting method is a popular and fast solver [37,38]. As discussed in the upcoming sections, the multiple shooting method has many features that make it suitable for implementation on a highly parallel computing architecture such as the GPU.

While the use of indirect methods for real-world problems is still an open challenge, prior work has shown that the historical

challenges associated with indirect optimization methods can be overcome for certain problems, enabling the rapid construction of high-quality solutions [39]. Some of the known issues such as handling path constraints can be solved to a certain extent using regularization [40]. Other issues such as the use of bang–bang control is being addressed by ongoing work [41]. The goal of this study is to demonstrate that indirect multiple shooting, when it can be applied, is a good candidate for effectively leveraging the GPU for these problems. 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 multiple shooting method can be efficiently executed even while running on GPU hardware that is five years behind the state of the art. As we show in Sec. III.D, our solver is able to get a speedup of two to four times over the CPU-based bvp4c for a low-dimensional problem. This benchmark was performed as validation to show that our implementation is indeed effectively using the GPUs rather than as a comparison showing which is the faster solver. By exploiting GPUs, the upper limit on problem size is probably much higher than that for CPU-based methods, especially since the computational power of GPUs is growing exponentially [22].

Gradient-based algorithms such as the shooting methods depend on the availability of accurate sensitivity information to converge toward 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 $\mathcal{O}(N^2)$ [42]. 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 has shown that it is possible to use the GPU to accelerate the computation of sensitivity information for the dynamic system [42]. Following that work, many GPU-specific optimizations in this investigation were devised to make the multiple shooting algorithm much faster. The algorithm and 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.

A. Overview of NVIDIA Fermi Architecture

The general layout of an Intel CPU and that of an NVIDIA Fermi [43] GPU are shown in Figs. 4 and 5, respectively. In the Intel Nehalem architecture [44] CPU shown in Fig. 4, there are four processing cores and a large cache memory block. In the Fermi GPU

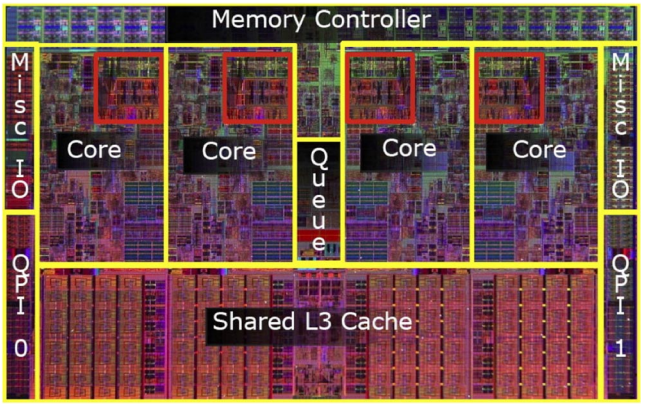


Fig. 4 Intel Core i7 Nehalem architecture [44].

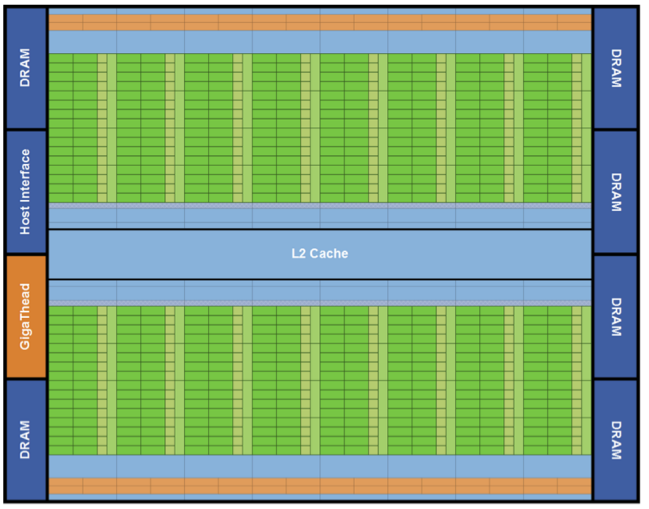


Fig. 5 NVIDIA Fermi architecture [43] (DRAM, dynamic random access memory).

(Fig. 5), each of the long vertical green rectangles represents a streaming multiprocessor (SM) that contains up to 32 individual processing cores or streaming processors for a total of up to 512 processing cores per GPU (Fig. 6). The newer Kepler [45] and Maxwell [46] 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 that can be expressed as data-parallel computations [21], with a high ratio of arithmetic operations to memory operations.

B. Overview of NVIDIA Compute Unified Device Architecture Framework

The 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. CUDA programs have been able to give speedups of 5–500 times faster than their CPU counterparts in many applications ranging from molecular simulation [27] to modeling of aircraft traffic [28]. Structuring 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 (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 is 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 in which all of its threads are executed in batches. 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 Sec. II.

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 codes, with the former running on the CPU and the latter running on the GPU. The two parts of the code are

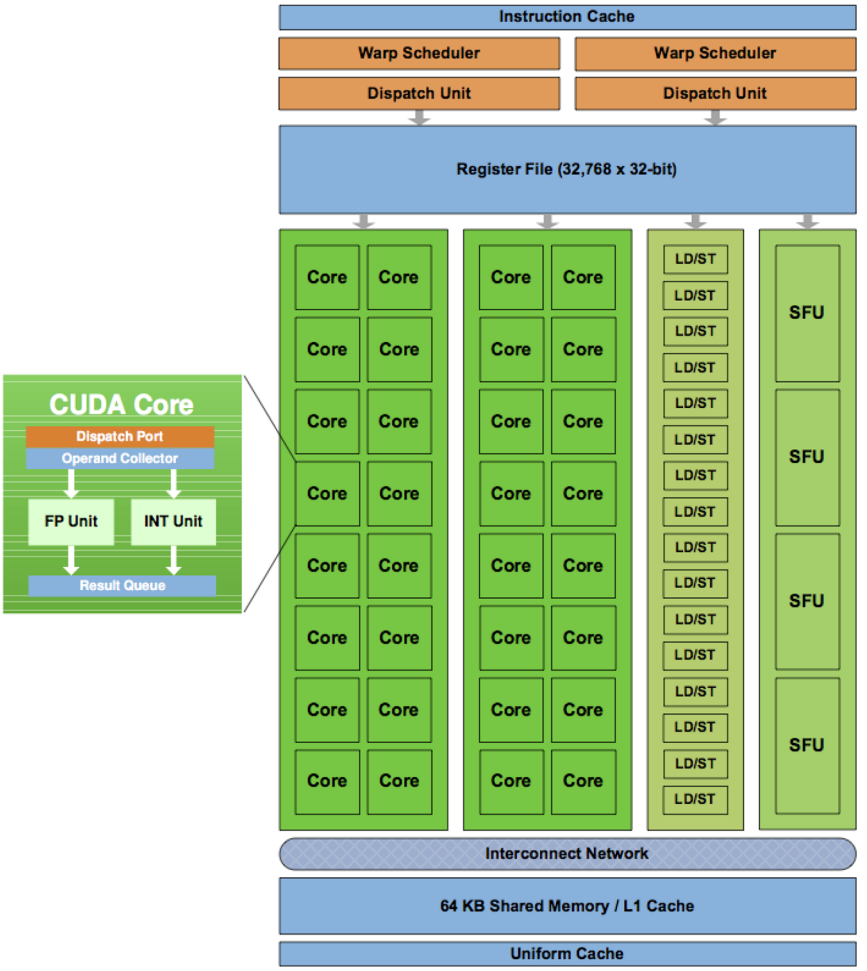


Fig. 6 Fermi SM [43] (LD/ST, load/store units).

compiled separately and then combined. In CUDA, there can be no GPU device code that runs by itself. There is always some host code that launches the kernels and then collects the results for postprocessing. NVCC can also generate device code that can be loaded by CUDA at run time. This is very useful in cases in which problem-specific information (such as the dynamic equations) has to be changed at run time.

C. GPU-Accelerated Solver Implementation

The GPU-accelerated optimal control solver implements a MATLAB® interface. The host code that controls the overall process and launches the GPU kernel is precompiled into a MATLAB® mex library. The solver uses Mathematica [47] to derive the necessary conditions of optimality and then generates and compiles the CUDA C code transparently to the designer. This produces a standalone device code in a binary format that can be loaded at run time by CUDA. This process is to be executed only once per problem. Once the CUDA binaries are available, changing problem parameters does not require recompilation. Figure 7 outlines this process.

It is to be noted that Mathematica is used only for performing symbolic manipulations, such as taking derivatives, solving the control law equation, etc. It is found to be more robust than MATLAB®'s Symbolic Toolbox at solving nonlinear systems of

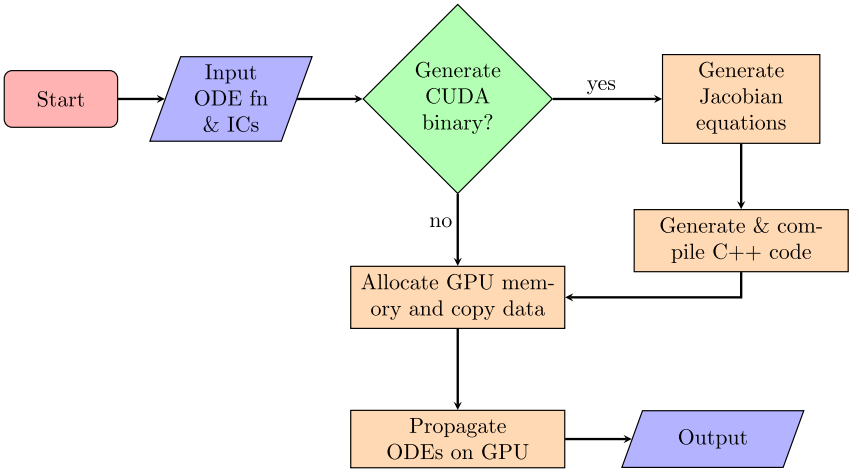


Fig. 7 Flowchart of GPU accelerator implementation (ODE, ordinary differential equations; IC, initial conditions).

equations to derive the control law. The equations thus derived are then used to create the MATLAB® and CUDA code files for the problems. Mathematica is not being used for any numerical computation.

The dynamic equations of the system are propagated on the CPU using the compiled MATLAB® code, and only the sensitivity information is propagated on the GPU. It is done in this manner because the number of dynamic equations (usually around 6–24 for reentry problems) does not constitute a big enough problem to justify the overheads of using the GPU. Benchmarks show that a hybrid method in which the states are propagated on the CPU and the sensitivities on the GPU give the best performance with current hardware. The solver is tested on an NVIDIA Tesla M2090 GPU (Fermi architecture) with an Intel Xeon E5-2670 2.60 GHz processor CPU.

II. GPU Shooting 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. Although 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. To optimize the code for maximum performance on the GPU, it is necessary to understand how the threads are scheduled and executed by CUDA.

A. 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 them, 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 $\Phi_1, \Phi_2, \ldots, \Phi_P$ are the STMs of the individual trajectory segments, the STM for the whole trajectory Φ can be computed as shown in Eq. (1):

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

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 fourth-order Runge–Kutta (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 most 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 set up 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. The particular test cases we run for this study only need 512 time steps to obtain reasonable results. For more sensitive problems, using more steps may be required. A better approach for hypersensitive problems might be to use adaptive RKF45 integration within each segment.

GPU occupancy is further magnified when the multiple shooting method is used. In this case, the trajectory is separated into multiple arcs, with the possibility of some of them following different dynamic paths (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 CPU-based propagation uses the same number of time steps as the GPU propagator. 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.

Modern GPUs can have in excess of 3800 CUDA cores [48]. Splitting the trajectory into this many segments might make sense when solving hypersensitive problems that will allow maximum GPU occupancy while using small time steps. However, on such high-end GPUs, there is probably an upper limit beyond which floating point errors (due to small time steps) cause diminishing returns. GPUs with more cores would be more useful for larger problems with more states and costates even if the number of segments remains relatively low.

B. 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. 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 a minimum number of transactions, resulting in a 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 Fig. 8. The diagram represents the storage of the elements of the STM for a trajectory with P segments and N dynamic equations. Φ_k represents element (i, j) of the STM for trajectory segment k. The data are 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 $\Phi_{1,1}$ for all the trajectory segments, followed by the element $\Phi_{2,1}$, and so on. The first thread block propagates the equations for element $\Phi_{1,1}$ for all the segments and accesses the contiguous memory block from Φ_1 to Φ_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.

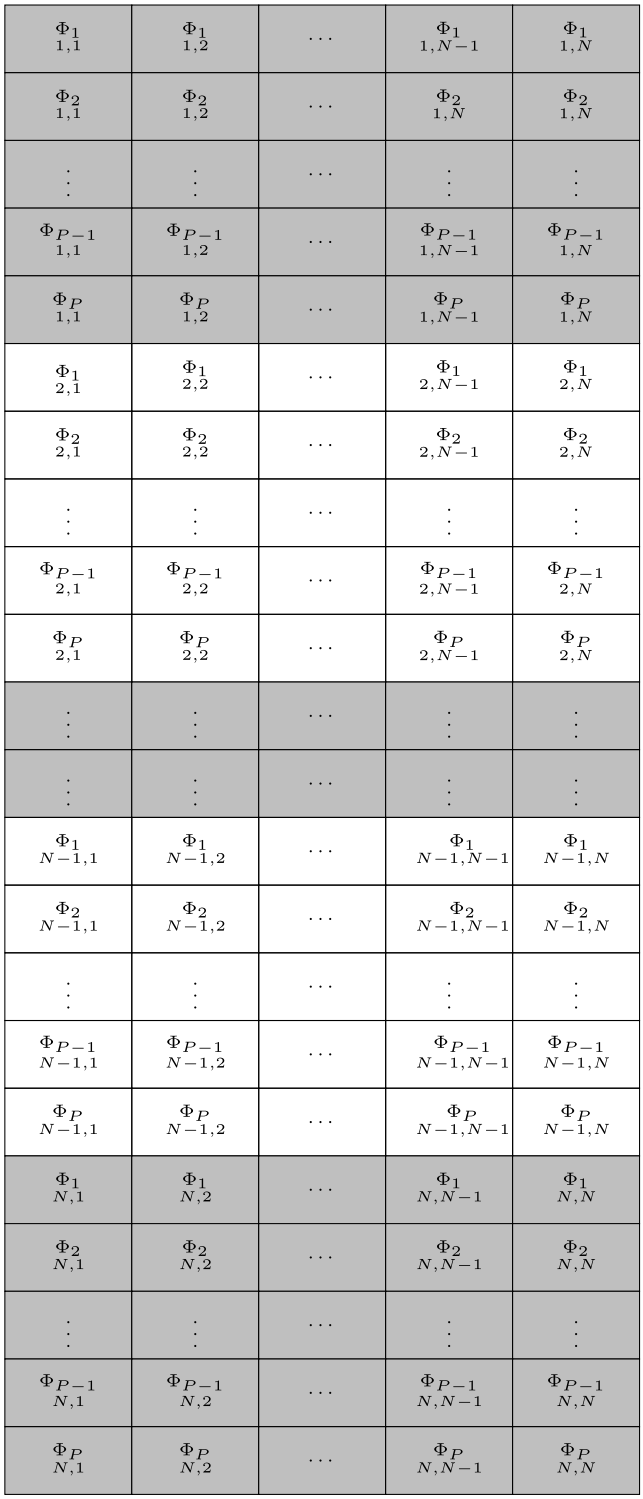


Fig. 8 Combined STM data structure in GPU memory.

C. 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 Basic Linear Algebra Subroutines 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 application program interface (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. (1), the new batched matrix multiplication API is found to be useful for making the process even faster. To do this, the process in Eq. (1) is transformed into a series of parallel steps called matrix reduction, an example of which is shown in Eq. (2):

$$\left.\begin{aligned}\Phi &= (\Phi_{512}\cdot\Phi_{511})\cdot(\Phi_{510}\cdot\Phi_{509})\cdot\ldots\cdot(\Phi_4\cdot\Phi_3)\cdot(\Phi_2\cdot\Phi_1)\\ &= (\Phi_{512\cdot511}\cdot\Phi_{510\cdot509})\cdot\ldots\cdot(\Phi_{4\cdot3}\cdot\Phi_{2\cdot1})\\ &= \Phi_{512\cdot511\cdot510\cdot509}\cdot\ldots\cdot\Phi_{4\cdot3\cdot2\cdot1}\\ &\vdots\\ &= \Phi_{512\cdot511\cdot\ldots\cdot2\cdot1}\end{aligned}\right\}\log_2 512 = 9\ \text{steps} \quad (2)$$

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. (2) is 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.

To summarize, the following GPU-specific optimizations are used to help the algorithm exploit the GPU architecture:

1) Split the trajectory into a number of segments (ideally a power of 2 close to the number of CUDA cores) to maximize GPU occupancy.

2) Structure the problem such that all the threads in a multiprocessor are integrating the same equation for different segments, thereby avoiding the issue of thread divergence.

3) Combine the state transition matrices for different trajectory segments into one large matrix, structured to allow memory access coalescing.

4) Parallel matrix reduction for computing the final STM by multiplying all the STMs from the individual trajectory segments.

III. Test Scenarios and Benchmarks

Trajectory optimization of a hypothetical unpowered long range weapon is used to demonstrate the benefits of the GPU-accelerated optimal control solver. In this, a hypothetical U.S. warship is located near the Gulf of Oman. A hypothetical high-value target is located 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 staging condition with a certain altitude, latitude, longitude, and velocity. A terminal position is specified in terms of altitude, latitude, and longitude. These common postboost staging/impact conditions are specified in Table 1.

A vehicle-centric polar coordinate system and three-degrees-of-freedom dynamic model [49] are used to develop the equations of motion for the problem as shown in Eqs. (4–9). 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 2.

Table 1 Postboost staging and impact conditions

State	Staging	Impact
Altitude h, m	80,000	4570
Velocity V, m/s	6000	free
Flight-path angle γ, deg	free	−60
Latitude θ, rad	23.14	33.66
Longitude ϕ, rad	64.07	67.63

Table 2 Environment parameters

Parameter	Value
μ, m^3/s^2	3.986×10^{14}
R_E, m	6.3781×10^{6}
ρ_0, kg/m^3	1.2
H, m	7500 m
ω, rad/s	7.29211585×10^{-5}

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. (3–11):

$$\text{Min } J = -V(t_f)^2 \quad (3)$$

Subject to:

$$\dot{r} = V\sin(\gamma) \quad (4)$$

$$\dot{\theta} = \frac{V\cos(\gamma)\cos(\psi)}{r\cos(\phi)} \quad (5)$$

$$\dot{\phi} = \frac{V\cos(\gamma)\sin(\psi)}{r} \quad (6)$$

$$\dot{V} = -\frac{qC_DA_{\text{ref}}}{m} - \frac{\mu\sin(\gamma)}{r^2} + \omega^2 r\cos(\phi)[\sin(\gamma)\cos(\phi) - \cos(\gamma)\sin(\phi)\sin(\psi)] \quad (7)$$

$$\dot{\gamma} = \frac{qC_LA_{\text{ref}}\cos(\sigma)}{mV} - \frac{\mu\cos(\gamma)}{Vr^2} + \frac{V\cos(\gamma)}{r} + 2\omega\cos(\phi)\cos(\psi) + \omega^2 r\cos(\phi)[\cos(\gamma)\cos(\phi) + \sin(\gamma)\sin(\phi)\sin(\psi)] \quad (8)$$

$$\dot{\psi} = \frac{qC_LA_{\text{ref}}\sin(\sigma)}{mV\cos(\gamma)} - \frac{V\cos(\gamma)\cos(\psi)\tan(\phi)}{r} + 2\omega[\tan(\gamma)\cos(\phi)\sin(\psi) - \sin(\phi)] - \frac{\omega^2 r\sin(\phi)\cos(\phi)\cos(\psi)}{V\cos(\gamma)} \quad (9)$$

where

$$q = (1/2)\rho V^2$$
$$\rho = \rho_0\exp[-(r - R_E)/H] \quad (10)$$

$$C_L = C_{L1}\alpha + C_{L0}$$
$$C_D = C_{D2}\alpha^2 + C_{D1}\alpha + C_{D0} \quad (11)$$

The hypersonic trajectory optimization problem defined previously is used to construct a number of hypothetical mission scenarios to demonstrate the application of the GPU-accelerated indirect solver for generating high-quality optimal trajectories. Different combinations of path constraints (such as heat rate) and/or interior point constraints (e.g., country overflight constraints) are enforced along the trajectory depending on the scenario being analyzed. These conditions represent fictitious but relevant mission

scenarios. Several such mission scenarios are analyzed, and the performance of the GPU solver is benchmarked against MATLAB®'s bvp4c solver. One such mission scenario that involves a combination of initial and terminal point constraints, interior point constraints, and path constraints is described in this section.

A. Construction of Initial Guess

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 postboost 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 [50] trajectory solution 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. (12), where V_0 is the postboost 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 deg prevents singularities in the equations of motion:

$$V = V_0 \exp[Ce^{(-h/H)}]$$
$$\text{where } C = \frac{\rho_0 H}{2\beta \sin(\gamma)},\ H \text{ is the atmospheric scale height}$$
$$\beta \text{ is the ballistic coefficient, and } \gamma \text{ is the flight path angle} \tag{12}$$

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. (13) 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:

$$\lambda_{v,f} = -2v_f \tag{13}$$

B. 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 Fig. 9. At the end of this process, a maximum terminal velocity trajectory connecting the postboost staging location and the targeted impact location is obtained.

C. Simultaneous Constraints Scenario

This example showcases a trajectory optimization problem with 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. 10a. However, the weapon is redesignated to a different target during the boost phase, essentially constraining the postboost 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 are shown in Figs. 10a and 10b and are constructed in 75 s 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. 10b. The two dives are timed precisely to have enough lift to perform the maneuvers while still maximizing the velocity at the target. The presence of path constraints and interior point constraints creates corner conditions in costates at the junction points between the arcs, which have to be solved simultaneously along with the boundary values. The indirect method solves for the complex control histories (Fig. 10b) and corner conditions (Fig. 10c) with high precision. Even in such a complex and highly constrained trajectory, the necessary conditions of optimality are fully satisfied, resulting in a high-quality solution.

D. Benchmarks

The test scenario described previously incorporates initial and terminal boundary conditions, a waypoint constraint, and a path constraint (on the stagnation point heat rate). Four other examples in which each of these constraints are applied individually are also analyzed, and the performance is benchmarked. All of the scenarios are solved using both MATLAB®'s bvp4c solver as well as the GPU-based multiple shooting solver (bvpgpu) built by the authors. As discussed in Sec. II, 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. The bvp4c solver uses a collocation method to solve the boundary value problem. It is a purely CPU-based algorithm that does not use any kind of parallel processing. It is to be noted that, while bvp4c uses collocation (a method used by many direct solvers), in this case, it is being used to solve a boundary value problem resulting from an indirect method rather than for direct optimization. As mentioned in the Introduction, the goal of this benchmark is to show that our GPU implementation is effective in using the GPU rather than to show that it is better than direct methods.

The run times for solving the different test scenarios using the two solvers are compared in Fig. 11. The first two examples in the benchmark contain only initial and terminal boundary conditions and have no in-flight constraints. The third and fourth test cases introduce a stagnation point heat rate constraint and a waypoint constraint into the trajectory, respectively. It can been seen that at present the GPU-accelerated shooting solver offers a speedup of two to four times faster than MATLAB®'s bvp4c in the given scenarios. It is to be

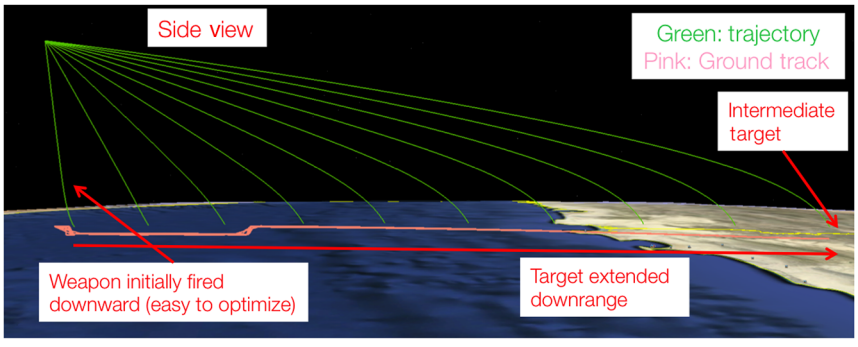
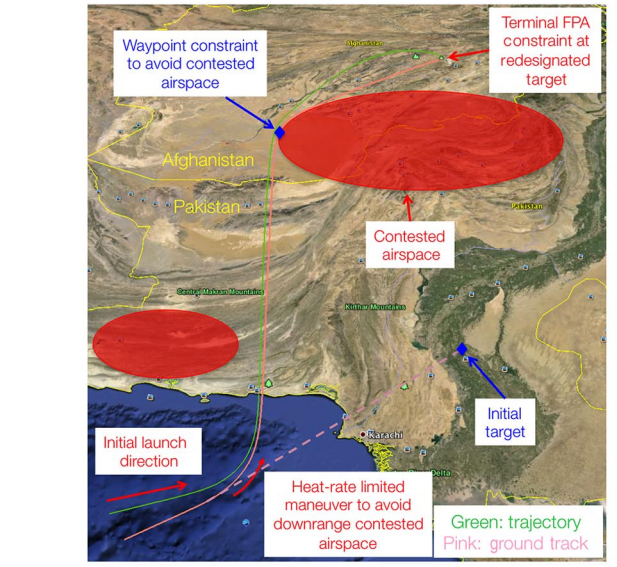
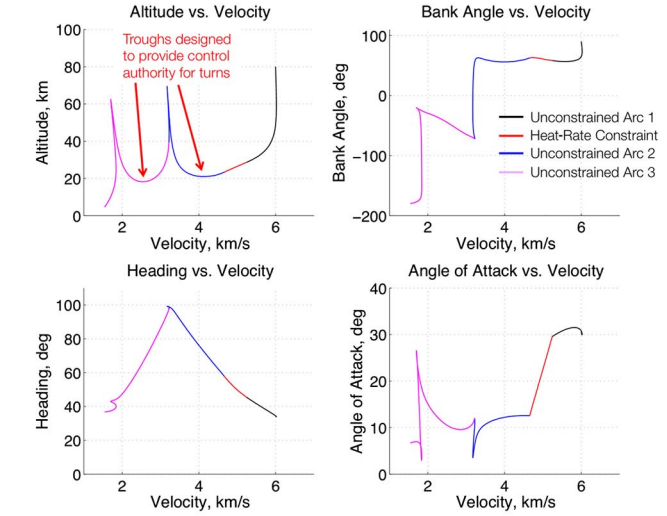


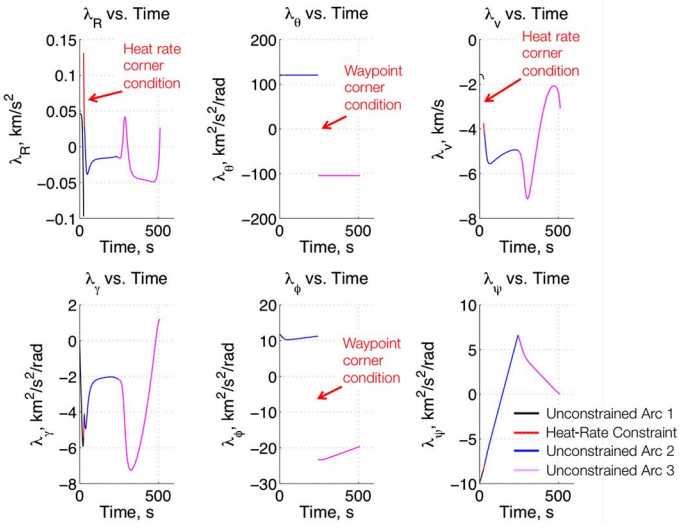
Fig. 9 Side view of initial part of continuation process.



a) Final trajectory solution



b) Trajectory and control history



c) Costates

Fig. 10 Simultaneous constraints example (FPA, flight path angle).

noted that this speedup is obtained for a problem with a relatively low dimensionality of 12 (states and costates). The goal of these benchmarks is to show that it is possible to design a multiple shooting algorithm in such a way as to exploit these emerging highly parallel architectures. If there were a hypothetical direct solver that uses the NLP algorithm similar to the one in [34], the dimensionality of the problem would probably have to grow significantly before it shows a speedup similar to what we achieve here. However, since there are no

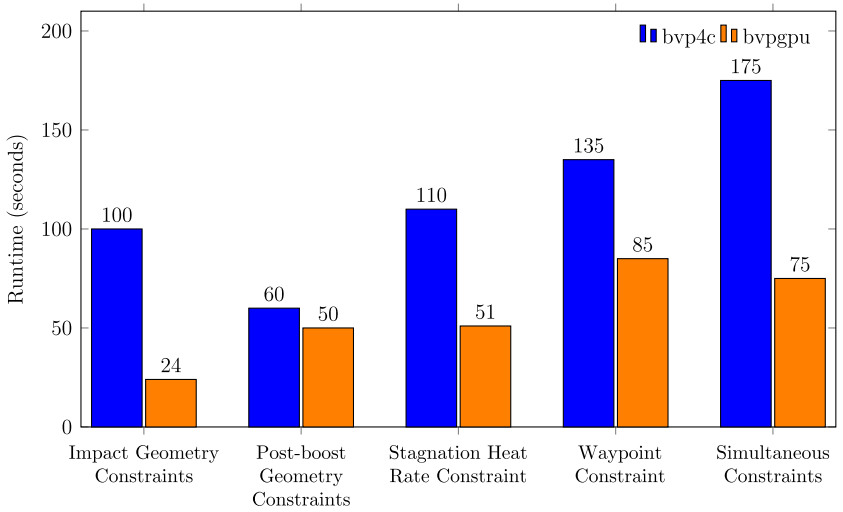


Fig. 11 Benchmarks: bvp4c vs bvpgpu.

direct solvers that use this (or other) GPU-enabled algorithms, there is no way to make an actual one-to-one comparison.

IV. Conclusions

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 using 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 are crucial to obtaining optimum performance on a GPU. The performance of the solver is compared against MATLAB®'s bvp4c to validate that it is in fact effectively using the GPU. The speedup obtained was for a relatively small problem, and GPU performance benefits are expected to improve as the problem gets bigger.

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. Various combinations of initial, terminal, interior point, and path constraints are enforced on the trajectory to demonstrate the design of complex hypersonic trajectories. 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 toward more parallel architectures, such as multicore 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.

Acknowledgment

This research was funded by Draper Laboratories through the University Research and Development Program.

References

[1] Betts, J. T., "Survey of Numerical Methods for Trajectory Optimization," *Journal of Guidance, Control, and Dynamics*, Vol. 21, No. 2, 1998, pp. 193–207. doi:10.2514/2.4231

[2] Hargraves, C. R., and Paris, S. W., "Direct Trajectory Optimization Using Nonlinear Programming and Collocation," *Journal of Guidance, Control, and Dynamics*, Vol. 10, No. 4, 1987, pp. 338–342. doi:10.2514/3.20223

[3] Herman, A. L., and Conway, B. A., "Direct Optimization Using Collocation Based on High-Order Gauss-Lobatto Quadrature Rules," *Journal of Guidance, Control, and Dynamics*, Vol. 19, No. 3, 1996, pp. 592–599. doi:10.2514/3.21662

[4] Bibeau, R., and Rubenstein, D., "Trajectory Optimization for a Fixed-Trim Reentry Vehicle Using Direct Collocation and Nonlinear Programming," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, AIAA Paper 2000-4262, 2000.

[5] Grant, M. J., and Mendeck, G. F., "Mars Science Laboratory Entry Optimization Using Particle Swarm Methodology," *Proceedings of AIAA Atmospheric Flight Mechanics Conference and Exhibit*, AIAA Paper 2007-6393, 2007.

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

[7] Patterson, M. A., and Rao, A. V., "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*, Vol. 39, No. 3, 2013, pp. 1–36. doi:10.1145/2450153

[8] Ross, I. M., "A Beginner's Guide to DIDO: A MATLAB Application Package for Solving Optimal Control Problems," Technical Rept. TR-711, Carmel, CA, Nov. 2007.

[9] Gill, P. E., Murray, W., and Saunders, M. A., "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization," *SIAM Journal on Optimization*, Vol. 12, No. 4, 2002, pp. 979–1006. doi:10.1137/S1052623499350013

[10] Saunders, B. R., "Optimal Trajectory Design Under Uncertainty," Ph.D. Thesis, Massachusetts Inst. of Technology, Cambridge, MA, 2012.

[11] Moore, G. E., "Cramming More Components onto Integrated Circuits," *Electronics*, Vol. 38, No. 8, April 1965, pp. 114–117.

[12] Agarwal, V., Hrishikesh, M. S., Keckler, S. W., and Burger, D., "Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures," *ACM SIGARCH Computer Architecture News*, Vol. 28, No. 2, May 2000, pp. 248–259. doi:10.1145/342001

[13] Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D., "GPUs and the Future of Parallel Computing," *IEEE Micro*, Vol. 31, No. 5, 2011, pp. 7–17. doi:10.1109/MM.2011.89

[14] Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O., "State-of-the-Art in Heterogeneous Computing," *Scientific Programming*, Vol. 18, No. 1, 2010, pp. 1–33. doi:10.1155/2010/540159

[15] Luebke, D., Harris, M., and Govindaraju, N., "GPGPU: General-Purpose Computation on Graphics Hardware," *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Association for Computing Machinery (ACM), 2006, Paper 208, http://dl.acm.org/citation.cfm?id=1188672.

[16] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J., "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, Vol. 26, Wiley Online Library, Hoboken, NJ, 2007, pp. 80–113, http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2007.01012.x/abstract.

[17] Larsen, E. S., and McAllister, D., "Fast Matrix Multiplies Using Graphics Hardware," *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, Association for Computing Machinery, New York, 2001, pp. 55–55.

[18] Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A., "Simulation of Cloud Dynamics on Graphics Hardware," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Assoc., Aire-la-Ville, Switzerland, 2003, pp. 92–101.

[19] Krüger, J., and Westermann, R., "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," *ACM Transactions on Graphics (TOG)*, Vol. 22, ACM, 2003, pp. 908–916.

[20] Hall, J. D., Carr, N. A., and Hart, J. C., "Cache and Bandwidth Aware Matrix Multiplication on the GPU," Univ. of Illinois Technical Rept. UIUCDCS-R-2003-2328, Urbana-Champaign, 2003.

[21] *Cuda C Programming Guide*, NVIDIA Corp., 2014.

[22] Brodtkorb, A. R., Hagen, T. R., and Sætra, M. L., "GPU Programming Strategies and Trends in GPU Computing," *Journal of Parallel and Distributed Computing*, Vol. 73, No. 1, Jan. 2013, pp. 4–13, http://dl.acm.org/citation.cfm?id=2399603.

[23] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C., "GPU Computing," *Proceedings of the IEEE*, Vol. 96, No. 5, 2008, pp. 879–899. doi:10.1109/JPROC.2008.917757

[24] Stone, J. E., Hardy, D. J., Ufimtsev, I. S., and Schulten, K., "GPU-Accelerated Molecular Modeling Coming of Age," *Journal of Molecular Graphics and Modelling*, Vol. 29, No. 2, 2010, pp. 116–125. doi:10.1016/j.jmgm.2010.06.010

[25] Komatitsch, D., Erlebacher, G., Göddeke, D., and Michéa, D., "High-Order Finite-Element Seismic Wave Propagation Modeling with MPI on a Large GPU Cluster," *Journal of Computational Physics*, Vol. 229, No. 20, 2010, pp. 7692–7714. doi:10.1016/j.jcp.2010.06.024

[26] Park, I. K., Singhal, N., Lee, M. H., Cho, S., and Kim, C. W., "Design and Performance Evaluation of Image Processing Algorithms on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 1, 2011, pp. 91–104. doi:10.1109/TPDS.2010.115

[27] Zhu, Y., Liu, H., and Li, Z., "GALAMOST: GPU-Accelerated Large-Scale Molecular Simulation Toolkit," *Journal of Computational Chemistry*, Vol. 34, No. 25, 2013, pp. 2197–2211. doi:10.1002/jcc.23365

[28] Menon, P., Tandale, M., and Wiraatmadja, S., "Faster Simulations of the National Airspace System," *NVIDIA GPU Technology Conference*, NVIDIA Corporation, 2010, Paper 2214, http://www.nvidia.com/content/GTC-2010/pdfs/2214_GTC2010.pdf.

[29] Lopez-Diaz, L., Aurelio, D., Torres, L., Martinez, E., Hernandez-Lopez, M., Gomez, J., Alejos, O., Carpentieri, M., Finocchio, G., and Consolo, G., "Micromagnetic Simulations Using Graphics Processing Units," *Journal of Physics D: Applied Physics*, Vol. 45, No. 32, 2012, Paper 323001. doi:10.1088/0022-3727/45/32/323001

[30] Cuevas, C., Berjón, D., Morán, F., and García, N., "Moving Object Detection for Real-Time Augmented Reality Applications in a GPGPU," *IEEE Transactions on Consumer Electronics*, Vol. 58, No. 1, 2012, pp. 117–125. doi:10.1109/TCE.2012.6170063

[31] Amada, T., Imura, M., Yasumuro, Y., Manabe, Y., and Chihara, K., "Particle-Based Fluid Simulation on GPU," *ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH*, ACM SIGGRAPH Paper C-36, 2004, http://www.cs.unc.edu/Events/Conferences/GP2/proc.pdf.

[32] Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Le, Q. V., and Ng, A. Y., "On Optimization Methods for Deep Learning," *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, Omnipress, Madison, WI, 2011, pp. 265–272.

[33] Yatawatta, S., Kazemi, S., and Zaroubi, S., "GPU Accelerated Nonlinear Optimization in Radio Interferometric Calibration," *Innovative Parallel Computing (InPar), 2012*, IEEE, Piscataway, NJ, 2012, pp. 1–6.

[34] Fei, Y., Rong, G., Wang, B., and Wang, W., "Parallel L-BFGS-B Algorithm on GPU," *Computers and Graphics*, Vol. 40, May 2014, pp. 1–9. doi:10.1016/j.cag.2014.01.002

[35] Shampine, L. F., Reichelt, M. W., and Kierzenka, J., "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c," 2000, https://www.mathworks.com/bvp_tutorial.

[36] Bryson, A., and Ho, Y., *Applied Optimal Control — Optimization, Estimation, and Control*, Taylor and Francis, Philadelphia, 1975, pp. 42–127, Chaps. 2, 3.

[37] Morrison, D. D., Riley, J. D., and Zancanaro, J. F., "Multiple Shooting Method for Two-Point Boundary Value Problems," *Communications of the ACM*, Vol. 5, No. 12, Dec. 1962, pp. 613–614. doi:10.1145/355580.369128

[38] Stoer, J., and Bulirsch, R., *Introduction to Numerical Analysis*, Vol. 12, Springer–Verlag, Berlin, 2002, pp. 557–581.

[39] Grant, M. J., and Braun, R. D., "Rapid Indirect Trajectory Optimization for Conceptual Design of Hypersonic Missions," *Journal of Spacecraft and Rockets*, Vol. 52, Special Section on Numerical Simulation of Hypersonic Flows, 2015, pp. 177–182. doi:10.2514/1.A32949

[40] Graichen, K., Kugi, A., Petit, N., and Chaplais, F., "Handling Constraints in Optimal Control with Saturation Functions and System Extension," *Systems and Control Letters*, Vol. 59, No. 11, 2010, pp. 671–679. doi:10.1016/j.sysconle.2010.08.003

[41] Mall, K., and Grant, M. J., "Epsilon-Trig Regularization Method for Bang-Bang Optimal Control Problems," *AIAA Atmospheric Flight Mechanics Conference*, AIAA Paper 2016-3238, 2016, http://arc.aiaa.org/doi/abs/10.2514/6.2016-3238.

[42] Arora, N., Russell, R. P., and Vuduc, R. W., "Fast Sensitivity Computations for Trajectory Optimization," *Advances in the Astronautical Sciences*, Vol. 135, No. 1, 2009, pp. 545–560.

[43] *NVIDIA's Next Generation CUDA Compute Architecture: FERMI*, NVIDIA Corp., 2009, http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[44] Gunther, S., and Singhal, R., "Next Generation Intel Microarchitecture (nehalem) Family: Architectural Insights and Power Management," *Intel Developer Forum*, Vol. 165, 2008, http://inf-server.inf.uth.gr/courses/CE431/resources/Nehalem.pdf.

[45] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, NVIDIA Corp., 2012, https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[46] *NVIDIA GeForce GTX 750Ti: Featuring First-Generation Maxwell GPU Technology*, NVIDIA Corp., 2014, http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.

[47] Wolfram Research, Inc., "Mathematica," Version 9.0, Wolfram Research, Inc., Champaign, Illinois, 2012.

[48] "NVIDIA Tesla P100 the Most Advanced Datacenter Accelerator Ever Built," 2016, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[49] Vinh, N. X., Busemann, A., and Culp, R. D., *Hypersonic and Planetary Entry Flight Mechanics*, Univ. of Michigan Press, Ann Arbor, MI, 1980.

[50] Allen, H. J., and Eggers, A. J., "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, https://ntrs.nasa.gov/search.jsp?R=19930091020.

D. A. Spencer
Associate Editor