AAS 09 - 337

FAST SENSITIVITY COMPUTATIONS FOR TRAJECTORY OPTIMIZATION

Nitin Arora[*], Ryan P. Russell[†], and Richard W. Vuduc[‡]

Gradient based trajectory optimization relies on accurate sensitivity information to robustly move a solution towards an optimum. Computational complexity of sensitivity calculations increases exponentially for higher problem dimensions and orders. Hence, the computation of these sensitivities is traditionally a major speed bottleneck in trajectory optimization and targeting algorithms. We propose to use Nvidia's GPU (Graphics Processing Unit) to rapidly calculate the derivatives in a multilayer, parallel, and heterogeneous way while the CPU (Central Processing Unit) sequentially computes the less expensive state equations. The proposed tool computes both the first and second order analytic sensitivities on the GPU with double precision accuracy. For an example trajectory propagation, we demonstrate overlapped computations such that sensitivities are calculated almost for free compared to the conventional CPU implementation.

NOMENCLATURE

t	Time vector
y	State vector
f	Equations of motion for the state
g	Inequality constraint vector
c	Equality constraint vector
X	Nominal state vector
I	Identity matrix
J	Performance index or cost
n	Dimension of state vector
x, y, z	Position vector
u, v, w	Velocity vector
G	Standard gravitational parameter
M	Mass of the body
ϕ^1	First order state transition matrix
ϕ^2	Second order state transition tensor
N_s	Number of sub-trajectories
N_t	Total number of integration steps
SP	Single precision
DP	Double precision
TR	Thread recursion

[*]Graduate Student, Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, 270 Ferst Drive, Atlanta, GA, 30332- 404-483-7015, narora9@mail.gatech.edu

[†]Assistant Professor, Guggenheim School of Aerospace Engineering, Georgia Institute of Technology, 270 Ferst Drive, Atlanta, GA, 30332-0150, 404-385-3342 (voice), 404-894-2760 (fax), ryan.russell@gatech.edu

[‡]Assistant Professor, Computational Science and Engineering Division. Georgia Institute of Technology, Atlanta, Georgia 30332-076, richie@cc.gatech.edu

STM	State transition matirx, $\in \Re_{N\times N}$
STT	State transition tensor, $\in \Re_{N\times N\times N}$
GPU	Graphics Processing Unit
CPU	Central Processing Unit
ODE	Ordinary diffrential equation
$Gflop/s$	Giga floating point operations per second
$nsub$	Total number of points per sub-trajectory (multiple of 16)
$scale$	Scaling parameter for nsub
$s/c\, revs$	Space craft revolutions
$CUDA$	Compute Unified Device Architecture
$NVCC$	NVIDIA C compiler
$GPGPU$	General-purpose computing on graphics processing units

Conventions

i	i^{th} sub-trajectory
j	j^{th} point on a sub-trajectory
T	Transpose
δx	Very small change in x
$\dot{x}$	Complete derivative of x with respect to time
$A*B$	Matrix times tensor, A is a matirx and B is a tensor: $A*B = \Sigma_k$ A(:,:)B(:,:,k)
A^T*B*A	Matrix transpose times tensor times matrix, A is a matirx and B is a tensor: A^T*B*A= Σ_k[A(:,:)^{T}B(k,:,:)A(:,:)]

INTRODUCTION

Mathematical and computational models are used in all areas of science and engineering for performing optimization [1, 2, 3, 4, 5]. Gradient based numerical optimization relies on accurate sensitivity information to robustly move a solution towards an optimum. While there are various subfields in numerical optimization such as Optimal Control [6, 7, 8, 9, 10] and Parameter Optimization [11, 12] , all gradient based continuous methods make use of numerical sensitivities to select new step directions.

Specifically many trajectory optimization algorithms rely heavily on higher order sensitivity information [13, 14 15]. Computational burden of sensitivity calculations increases exponentially with problem complexity and the requirement for higher order derivatives. Therefore, with the existing CPU architecture, it is often not feasible to solve realistic model problems because of the extraordinarily expensive sensitivity calculations. Given a function evaluation computational complexity of O(n), the corresponding first order sensitivities have a computational complexity of O(n^2), and similarly second order sensitivities have a computational complexity of O(n^3). This complexity (and the high costs of large CPU clusters required to overcome) therefore prohibits many classes of high-fidelity optimization problems from being solved.

Parallel sensitivity analysis has been limited to a narrow class of problems [16, 17, 18, 19]. These methods are either not scalable or they perform inefficiently as the current CPU hardware is not able to exploit the massive parallelism present in the underlying problem.

The Nvidia's CUDA (Compute Unified Device Architecture) technology provides a convenient

solution to the above problem. Nvidia's GPU architecture is tailor made to exploit fine grain parallelism and CUDA makes it possible to program the hardware efficiently. With the introduction of double precision capability in the new Nvidia's TESLA C1060 * processor, it is now possible to achieve dramatic speedups if new and innovative algorithms utilize the large amount of parallelism efficiently.

In this paper we propose a new tool, which exploits heterogeneous programming by utilizing the CPU and Nvidia's Graphics Card (GPU) together to achieve substantial speedups for sensitivity computation. The proposed algorithm breaks the CPU derived solution trajectory (or solution path) into numerous smaller blocks and solves the associated sensitivities in a heterogeneous parallel manner on the GPU. These multiple levels of parallelism exploit the fine grained architecture of the GPU.

We perform comparisons with a CPU only simulation on an example Keplerian trajectory. The new CPU plus GPU method is shown to achieve speedups of 233 times and 21 times for first order STM and second order STT sensitivity computations respectively, while maintaining accuracy of at least 13 significant digits. The final speedup for a two body trajectory plus sensitivity propagation over the complete CPU implementation is 4 times (approximately) and 14 times (approximately) for first order STM and second order STT sensitivity evaluations respectively. The tool is general in its design and implementation subject only to user defined equations of motion. The fast sensitivity propagations can therefore be useful to a wide variety of gradient based optimization or targeting problems.

In the forthcoming sections we state the general sensitivity formulation, provida a brief introduction of the current NVIDIA GPU and the CUDA programming model, explain the algorithmic implementation, and finally present the example results.

GENERAL SENSITIVITY FORUMULATION

Numerical optimization refers to maximizing or minimizing a continuous function subject to certain constraints and input variables. A general numerical optimization strategy is shown in Fig 1. The black box function can be any continuous function of the input.

A common and general optimization problem involving state equations is mathematically defined as follows

$$\min_{y(t_0)} J(y_f, t_f)\ ,\ \text{subject to} \begin{cases} \dot{y} = f(y,t) \\ c(y_f, t_f) = 0 \\ g(y_f, t_f) < 0 \\ y \in \Re, t \in \Re+ \end{cases}$$

Here J is the performance index which we want to minimize, y is the vector of state (and control) variables, t is time, f(y) represents dynamics of the system, $c(y)$ and $g(y)$ are the equality and inequality constraints (of arbitrary dimension) on the state vector. If J=0 the problem reduces to a targeting or boundary value problem.

To solve the above problem using gradient methods , the sensitivities (derivatives) of the final sate vector with respect to the initial state vector are required. The first order derivatives can be

*http://www.nvidia.com/object/product_tesla_c1060_us.html

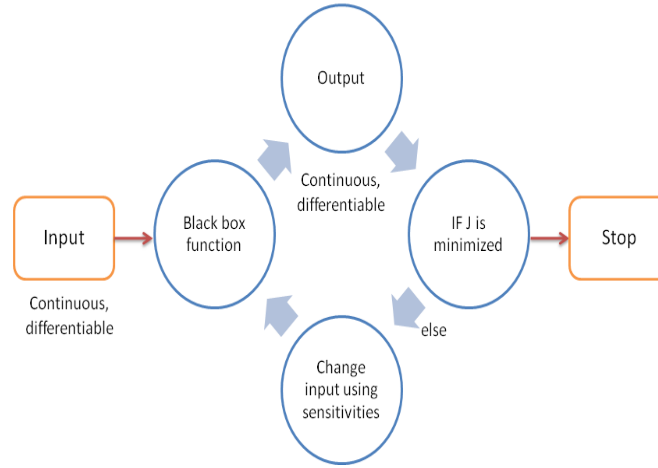

Figure 1. General Solution Strategy

computed using numerical differencing of the function or analytically by direct integration of the so-called state transition matrix (STM) [7, 20]. Many solution techniques (Newtons method for example) require second order derivatives to converge the solution efficiently towards the optimum. The second order state transition tensors (STT) can also be calculated via numerical differencing or direct integration [13, 21, 15]. The STM and STT are used to map derivatives from one time to another on a given continuous trajectory. Please refer to [20, 21, 22] for detailed discussion of STM and STT. The STTs have an exceptionally high computational cost associated with them. Hence, second order derivatives (hessians) are usually only approximated; and full second order derivatives are only used in specialized high-fidelity methods [23, 24]. Many important trajectory optimization problems are highly non-linear in nature making these higher order sensitivitiy computations very attractive in the solution process.

The general taylor series expression for the first order STM and the second order STT about the nominal state (X) is given by Eq. 1

$$\delta X_{j+1} = \phi^1 \delta X_j + \frac{1}{2} \delta X_j^T * \phi^2 * \delta X_j \tag{1}$$

These highly coupled sensitivities (ϕ^1 and ϕ^2) are evaluated alongside the integration of the state vector by solving Eq. 2 and Eq. 3.

$$\dot{\phi^1} = f_x \phi^1 \tag{2}$$

$$\dot{\phi^2} = f_x * \phi^2 + (\phi^1)^T * f_{xx} * \phi^1 \tag{3}$$

subject to initial condition $\phi^1(t_o) = I_{n \times n}$ and $\phi^2(t_o) = 0_{n \times n \times n}$

The complexity of computing the sensitivities in terms of flops (floating point operations per second) is of the order O(n^{p+1}), where n is the dimension of the state vector and p is the order of sensitivity required. In this paper we consider only up to p=2. Consider a typical trajectory problem of dimension 6. The STT and STM are of dimension 6x6 and 6x6x6 respectively. A concurrent

evaluation of the state, STM, and STT therefore requires numerical integration of 6+36+216 coupled equations. Note that the STT dimension can be reduced to $n(n+1)/2$ if symmetry is considered.

Although the successive steps of the state trajectory must be computed in sequential form, the STM and STT can be calculated in parallel once all points of the state are known. We build upon this insight and use the Nvidia GPU hardware with the help of CUDA technology to achieve substantial performance improvement. In the next section we give a brief overview of the Nvidia GPU architecture and the CUDA technology.

NVIDIA GPU ARCHITECTURE AND CUDA

Recent advances in the programmable GPU has lead to the development of a highly parallel and multi-threaded processor with many-cores(nvidia cite). Given the GPU's high computational power and its ability to tap fine grain parallelism, researchers are now mapping non-graphical applications to the hardware with a wide range of success [25, 26, 27, 28, 29]. This field is generally called GPGPU (General Purpose Computing on GPU's) programming. The main breakthrough in GPGPU programming came with the development of Nvidia TESLA architecture (in late 2006 [30]) along with the recent introduction Nvidia CUDA * technology. Before CUDA advanced GPU programming knowledge was required to exploit the hardware effectively and it was still not very efficient. Post CUDA, there has been tremendous growth in wide scale GPGPU programming applications on the TESLA architecture. Most of these applications have witnessed a performance boost of 5 to 500 times, thereby outperforming many mid-range supercomputers with just one graphics processor in most cases. The latest TESLA G200 architecture (the C1060 series) consists of 240 cores and 4 GB of device memory. With the recent addition of double precision floating point arithmetic support to CUDA its possible now to achieve performance increase without sacrificing accuracy. The main task is to design an algorithm which maps well to the GPU and exploits this abundant computing power.

CUDA (Compute Unified Device Architecture)

The CUDA computing architecture is a C-like programing language with keywords for labeling data-parallel functions (kernels), and their associated data structures. Kernels generally execute a large number of threads (on the order of tens of thousands) in parallel. A thread is basically a fork which results from concurrent execution of computation on the GPU. Typically, in the GPU programming model, thousands of threads perform the same set of operations over a different set of data. It is worth noting that CUDA threads are computationally lighter than the threads on the CPU and hence they need very few cycles to generate and schedule.

The NVIDIA C Compiler (NVCC) is responsible for compiling the CUDA code. The part of the code which runs on the GPU is called the device code and the part of the code running on the CPU is called the host code. The host and device codes can be compiled using different compilers and linked at runtime. For our implementation, we compile the host code with Intel Fortran compiler and link it with the device code compiled with NVCC.

The GPU execution starts with the host invoking a kernel function, where a large number of threads are spawned. All threads which run on a kernel are collectively called a grid block. This gird block is further divided into smaller units called thread blocks. Each thread block can have at most 512 threads, which can communicate and synchronize among each other via shared memory (up-to

* http://www.nvidia.com/object/cuda_what_is.html

16 Kb). When all threads of a kernel complete their execution, the corresponding grid terminates and the execution continues on the host code until another kernel is invoked.

The main points which have to be kept in mind while designing a CUDA algorithm are

- designing a fine-grained parallel algorithm with sufficient amount of independent thread blocks to hide global memory latency (time to access GPU's main memory)
- using shared memory for data reuse within a thread block (shared memory is 300 times faster than global memory)
- coalesced and conflict free memory access between multiple memory abstractions (device memory, shared memory, register memory)
- minimizing and/or hiding CPU-GPU memory transfers (PCI Bus transfers) as they are slow and hence directly affect the performance
- optimizing register usage (total 16384 registers available on G200) which restricts the number of threads and thread blocks which can be deployed simultaneously
- concurrent execution (overlapping work between CPU and GPU)

All of these topics make it challenging to develop algorithms which map effectively to the GPU. Often non-intuitive techniques are developed to map conventionally serial algorithm to the GPU *. Once a algorithm is developed, achieving high performance (5x-50x) is commonly possible. Very high performance boost (up to 100x or more) are possible only if the algorithm maps efficiently to the GPU hardware and a sufficient amount optimization has been performed. Hence algorithm development is the major activity for consideration when programing in CUDA. Fig 2 gives a overview of the CUDA programming model.

The next section disscusses the implementation of the fast sensitivity calculation tool.

HETROGENOUS IMPLEMENTATION

At any given point on the solution trajectory, the STMs and STTs (any order) are a function of the state vector and time at that point. Hence, these sensitivities can be evaluated in parallel once we obtain the state information for the whole trajectory.

Given two STMs which map the partial derivatives between times t_i to t_{i+1} and between times t_{i+1} to t_{i+2}, then the equivalent STM mapping between times t_i to t_{i+2} is given by the chain rule in Eq 4:

$$\phi^1(t_{i+2}, t_i) = \phi^1(t_{i+2}, t_{i+1})\phi^1(t_{i+1}, t_i) \quad (4)$$

For a second order STT, the mapping expression from one time to another is more complicated and is calculated in Eq 5.

$$\phi^2(i+2, i) = \phi^1(t_{i+2}, t_{i+1}) * \phi^2(t_{i+1}, t_i) + (\phi^1(t_{i+1}, t_i))^T * \phi^2(t_{i+2}, t_{i+1}) * \phi^1(t_{i+1}, t_i) \quad (5)$$

Herein lies the motivation to compute STMs and STTs in parallel. Given N_t number of integration steps required for the state trajectory, the STM and STT from point i to $i+1$ for all $i = 1..N_t - 1$

* http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#scan

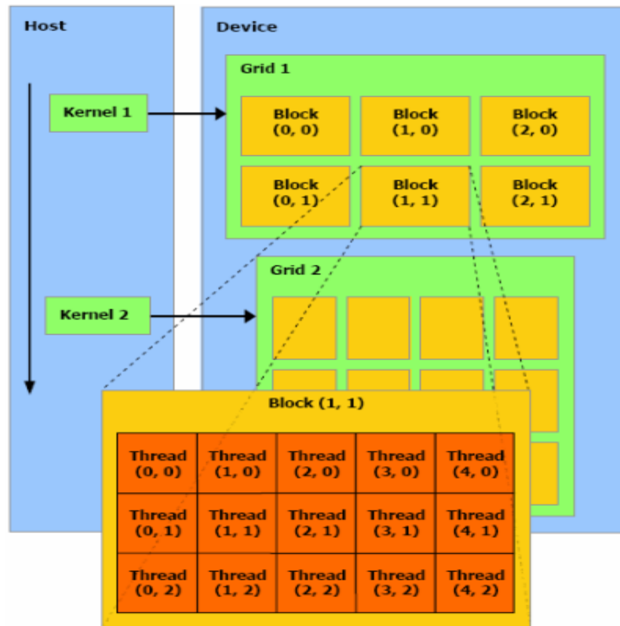

Figure 2. CUDA programming model [figure taken from 31]

can be calculated in parallel with initial conditions I and 0 respectively. The final state sensitivities with respect to the initial state are then calculated with the recursive evaluations of Eq. 4 and Eq. 5. We call this final step the reduction or reduce step.

The next section discusses this solution strategy in detail.

Solution Strategy

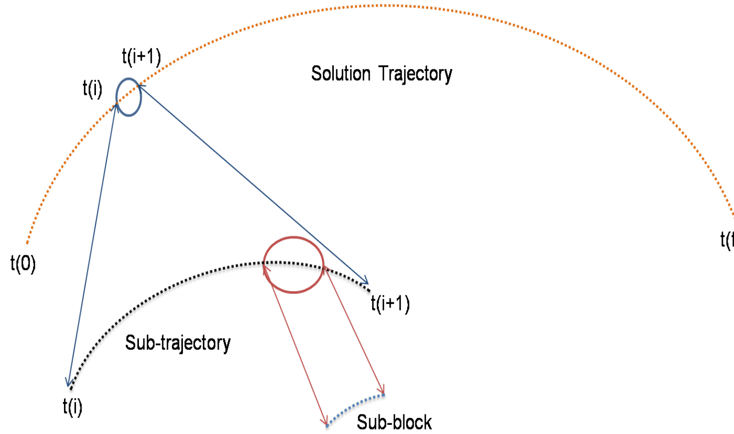

Figure 3. Solution Strategy

We start by breaking the CPU generated sequential trajectory into multiple sub-trajectories with each sub-trajectory consisiting of $nsub$ number of integration steps. Figure 3 shows the basic solution strategy which further breaks a sub-trajectory into various blocks, with each block containing a

certain number of points. As only state information is required to compute the sensitivities between two points in a particular block, this structure is a perfect candidate for explicit parallelism. The sensitivities within each block are mapped to a GPU's thread block and multiple blocks are joined together to form a GPU grid block. The whole grid block is then evaluated in parallel. The final sensitivity matrix is calculated via the chain matrix reduction on the CPU. Further, these computations are repeated for each sub-trajectory which gives rise to multiple levels of parallelism and permits concurrent execution.

To calculate the sensitivities a GPU kernel is invoked as soon as we have the CPU integrated state at each point on a sub-trajectory. So while the GPU is evaluating the sensitivities for the sub-trajectory (i) the CPU advances with the state integration for the sub-trajectory $(i + 1)$. This enables overlapping the GPU sensitivity computation with the CPU state integration. Typically, for our first order STM computation the GPU finishes before the CPU has finished integrating the next sub-trajectory. This results in an almost complete computation overlap between the two hardwares, except for the last sub-trajectory computation on the GPU. This heterogeneous computation strategy along with an intelligent memory copy operation exploits the GPU architecture efficiently. The basic execution strategy is the same for both the first and second order STM and STT evaluation.

Next we elaborate on the specifics of the first and second order implementation.

First order STM implementation

The evaluation of the first order STM from one point to the next is divided into 4 kernel calls. The first kernel is responsible for calculating the initial function evaluation and initializing the global memory for each thread. The global memory holds the state information and the step size taken by the integrator at each point. The next two kernels execute 12 times (sequentially) corresponding to the 12 function evaluations required per time step in the ODE 78 Dormand Prince integrator [32] (implemented on the GPU). After execution of kernels 1,2 and 3 we obtain the final state transition matrices between subsequent points on the current sub-trajectory being evaluated.

The kernel 4 is then invoked to locally reduce the STMs in each thread block to a single STM. This operation uses a thread recursion (TR) algorithm. To facilitate faster parallel computations, threads of a particular thread block load matrices in a special order. The TR algorithm for a fictitious thread block size of 8 is as follows

1: **// Suppose we have in total 8 threads hence 8 first order state transition matrices to multiply in descending order**
2: **// load matrices to each thread in the following order**
3: $thread1 = Mat7$, $thread2 = Mat3$, $thread3 = Mat5$, $thread4 = Mat1$
4: $thread5 = Mat6$, $thread6 = Mat4$, $thread7 = Mat0$
5: **// this loading can be automated using bitwise operations; next we do recursive multiplication, each matirx multiplication is done is parallel at each iteration and uses shared memory to enable data reuse**
6: **// 1st iteration**
7: $thread1 : M76 = M7 * M6$, $thread2 : M32 = M3 * M2$, $thread3 : M54 = M5 * M4;$, $thread4 : M10 = M1 * M0;$
8: $threadssynchronize;$
9: **// 2nd iteration**
10: $thread1 : M7654 = M76 * M54$, $thread2 : M3210 = M32 * M10$
11: $threadssynchronize;$
12: **// 3rd iteration**
13: $thread1 : M = M7654 * M3210$
14: **// the final thread then writes the results back into the global memory of the GPU**
15: **for** $threadID == 1$ **do**
16: **// other threads in a thread block just wait**
17: $globalMem(threadID) = M$
18: **end for**

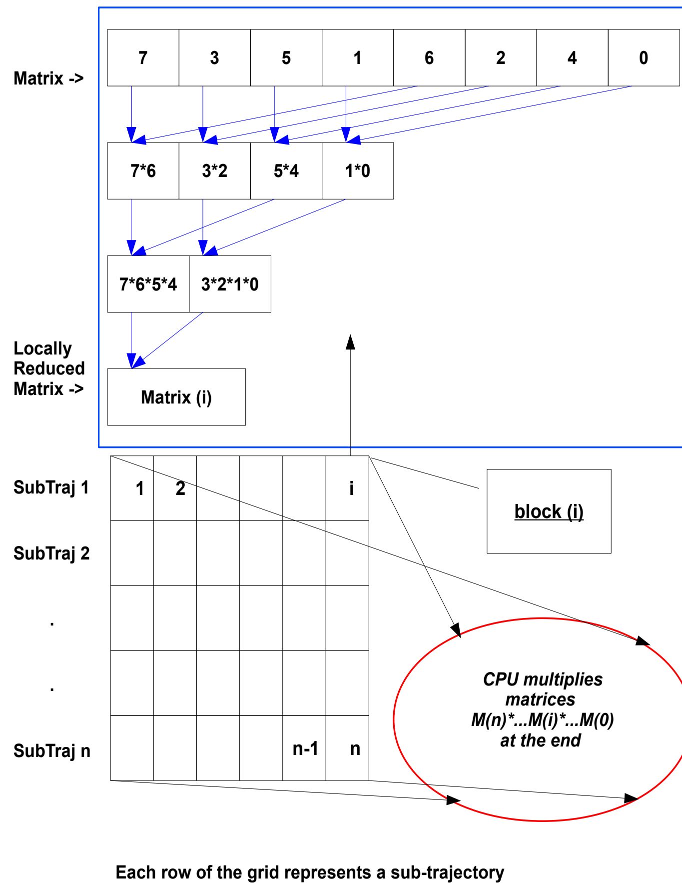

Figure 4. TR algorithm and final CPU reduction

Though the above algorithm is shown for 8 threads per thread block, we use 256 threads per thread block in our actual code. After kernel 4 is finished we are left with limited number of state transition matrices which have to be multiplied (in decreasing order) to get the final state transition matrix for the sub-trajectory. The above procedure except for the final CPU reduction step is repeated for various sub-trajectories. The final reduction step is performed all together for each sub-trajectory on the CPU after the state integration.

Using the TR algorithm we are able to sginificantly reduce to the number of matrices which multiply in the final reduction step on the CPU. This strategy also imparts numerical stability to the STM evaluation as we are always multipliying matrices which are of the same order (approximately).

Figure 4 shows the TR algorithm and the final CPU reduction operation.

First order STM + Second order STT implementation

The second order implementation for each sub-trajectory is accomplished by 6 kernel calls and one final complete CPU reduction. As in the previous case, kernel 1 performs the initial function evaluation and global memory intalization for each thread. Kernels 2 to 6 are called 12 times

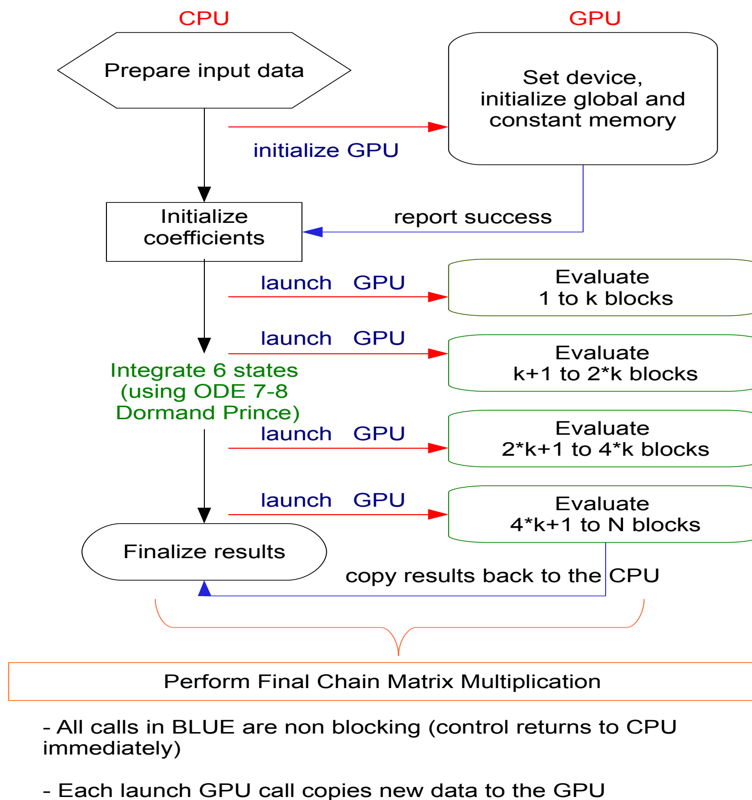

Figure 5. General hetrogenous algorithm for sensitivity computation

sequentially, each performing a part of a single function evaluation for one step of the integrator. Specifically, kernels 4 and 5 carry out required the matrix tensor products needed for the second order STT integration. After all kernels are finished we obtain the full first order STM and second order STT between subsequent points on the sub-trajectory. This operation is repeated for all the sub-trajectories, followed by a complete reduction to obtain the final sensitivities on the CPU.

User interface

We basically replicate the capability of a general integrator where the user provides a set of routines which perform the function evaluation. These routines are programmed in C language as CUDA currently is compatible only with C. The user has full control over the parameters which directly affect the performance of the tool, like the number of points in a sub-trajectory ($nsub$), number of thread blocks, number of sub-trajectories (N_s), etc.

The user routines (both for first order STM and second order STT) should be optimized for minimizing global memory transfers and avoiding shared bank conflicts (avoiding threads to read from shared memory in a random fashion) even at the expense of doing more floating point operations (flops). This is often the case for a GPU kernel, as global memory operations are generally more expensive (up-to 300 times slower) then floating point operations on the GPU due to lack of cache memory.

By default the number of points per sub-trajectory ($nsub$) is set to $30720/scale$. The parameter

$nsub$ has to be a multiple of 16 to achieve high global memory performance on the GPU. By default, the scale parameter has a value of 4 for the first order STM computation and 2 for the second order STT computation. The code automatically handles the last sub-trajectory branch evaluation by launching empty threads on the GPU, if N_t (number of steps taken by the CPU integrator) for the complete trajectory is not a multiple of $nsub$.

Figure 5 depicts the general heterogeneous algorithm.

RESULTS

In this section we evaluate the performance of our new tool against an optimized CPU implementation. A 2-body near earth propagation is used as the test trajectory. We evaluate performance for both the first order STM and second order STT implementation. For the 2-body case the order of integration of the state vector (y) is 6. Hence the order of integration for the STM computation is 42 (36+6) and the order for the STM plus STT computation its 258 (6 + 36 + 216). We are aware of the symmetry present in the second order STT computation but we currently choose to avoid the added complication in the GPU implementation.

Table 1 states the initial conditions for the propagated trajectory.

Table 1. Initial condition (body-fixed frame) for trajectory integration

Orbital Parameter	Value
Semi-major axis (a)	8300 (km)
Eccentricity (e)	0.49
Inclination (i)	35 (degrees)
Argument of periapsis (ω)	9 (degrees)
Longitude of ascending node (Ω)	20 (degrees)
True anomaly at epoch (ν)	0 (degrees)

For the current computation, the $scale$ parameter is set to a default value of 4 for the first order STM computation and 2 for the second order STT computation for all the results. Hence the number of points per sub-trajectory is 7680 and 15360, respectively.

Table 2. Test hardware specifications

Component type	Component
CPU	2 x Intel Core 2 Duo E6550 @ 2.33 Ghz
Operating system	Linux X86_64
GPU	Tesla C1060
Memory	4 GB

Test Hardware

Table 2 gives the specifications of the test hardware.

Table 3. Maximum theoretical performance comparison

Criteria	CPU	GPU
Max SP Gflop/s	24	933
Max DP Gflop/s	12	78

Table 3 gives the theorectical performance of the CPU and GPU used for this example.

The CPU code is compiled with the Intel Fortran compiler version 11.0 with optimization level set to "*-fast*". This enables auto vectorization and inter-procedural optimization. These optimizations result in a 2 times improvement in performance over the un-optimized CPU code. Apart from compiler optimization the CPU code is tuned for high performance Fortran 90.

The CUDA code is compiled using the NVCC compiler version 2.0 . All computations are carried out using a ODE 78 Dormand Prince integrator [32] set to unitless tolerance of 1E-14. For consistency and importance to the astrodynamics community, IEEE compliant double precision arithmetic has been used for all the results presented.

Table 4. Performance table for first order sensitivity computation

Tof (days)	State only (CPU)	State + STM (CPU)	State + STM (GPU plus CPU)
4.25	0.10	0.38	0.12
17.00	0.39	1.52	0.41
25.00	0.58	2.29	0.60
68.00	1.54	6.11	1.58
100.00	2.32	9.15	2.35
136.00	3.09	12.19	3.13
200.00	4.54	17.86	4.60

First order STM computation

Table 4 gives the performance of our tool compared to the corresponding CPU implemenation for first order STM plus state computation.

We define speedup by Eq. 6

$$speedup = \frac{(CPU\ time\ for\ integrating\ sensitivities\ along\ with\ the\ state)}{(CPU\ time\ for\ integrating\ the\ state + GPU\ time\ for\ integrating\ sensitivities)} \tag{6}$$

This speedup is always less then the theoretical maximum speedup, defined by Eq. 7

$$speedup_{max} = \frac{(CPU\ time\ for\ integrating\ sensitivities\ along\ with\ the\ state)}{(CPU\ time\ for\ integrating\ only\ the\ state)} \tag{7}$$

Figure 6 shows the speedup we achieve over the full CPU implementation with respect to time of flight. As we are able to almost completely hide the first order STM calculations on the GPU, the final speedup

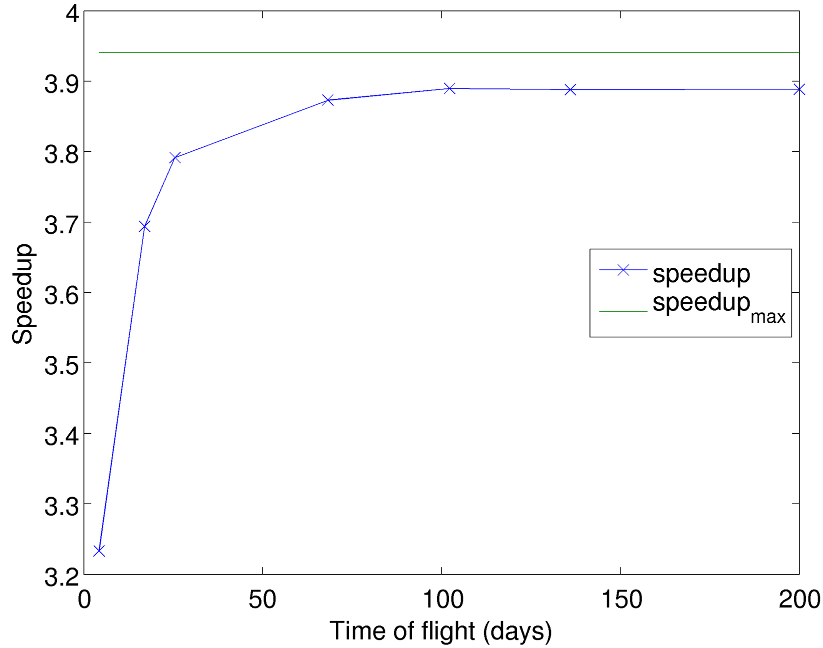

Figure 6. Speedup for complete STM computation

approaches the theoretical maximum value (Fig 6). In terms of speed, our GPU implemenation of the STM plus state is almost as fast as the the CPU implementation of the states only. Therefore, we approximately achieve the conventionally diffcicult to compute STM calculations for free. We further note that, for a computationally more expensive STM calculation (e.g higher dimension state and complicated force models) the theoretical speedup limit will be much higher and so will be our speedup over the CPU implementation.

First order STM + Second order STT computation

Table 5. Performance table for first order sensitivity computation

Tof (days)	State only (CPU)	State + STM + STT (CPU)	State + STM + STT (GPU plus CPU)
4.25	0.10	3.58	0.32
17.00	0.39	14.29	1.15
25.00	0.59	21.63	1.76
68.00	1.53	56.10	4.54
100.00	2.31	85.00	6.77
136.00	3.06	112.50	8.43
200.00	4.48	167.70	11.94

Table 5 gives the performance of our tool compared to the corresponding CPU implemenation for the second order STT, the first order STM, and state computation.

Figure 7 shows the speedup over the CPU implementation with increasing time of flight. We can see that the speedup is more impressive than the first order STM implementation because the dimension of the STT

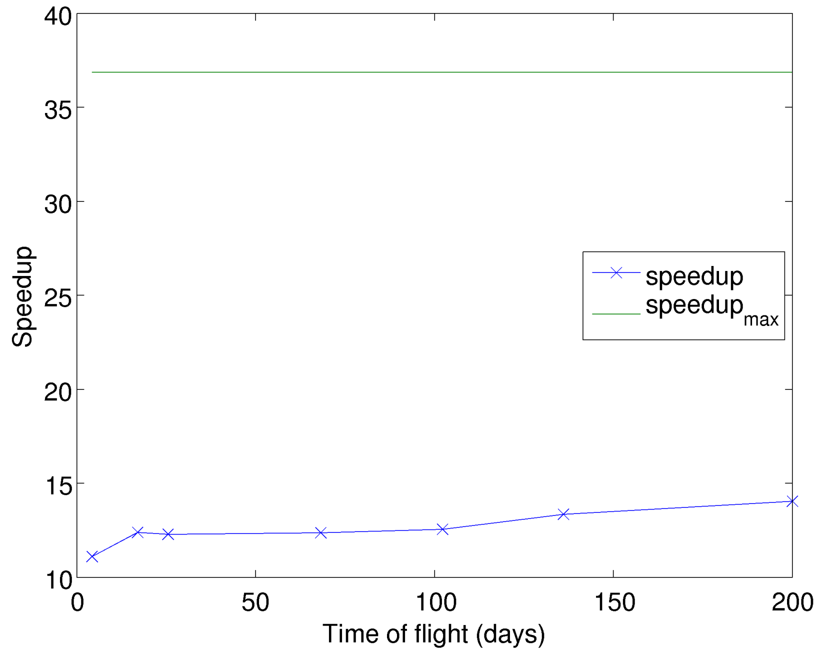

Figure 7. Speedup for complete STM plus STT computation

is 6 times larger. Due to the final complete reduction being done on the CPU (as opposed to the first order STM case) we are not able to efficiently overlap the computations between the GPU and CPU. Still we are able to achieve an order of magnitude speed improvement over the CPU implementation. As the GPU favors computation over memory operation, we expect this speedup value to be higher for more computationally expensive STT evaluations.

It is worthwhile to note that the GPU performs 8 times (approximately) faster in single precision mode than in double precision mode.

Numerical acuuracy

The ODE 78 integration on the GPU is accurate up to 14 digits when compared to the CPU integration. This has been achieved by designing numerically stable algorithms and by not using fast intrinsic math functions on the GPU. The final first order STM and second order STT have relative errors of 1E-13 (approximately) for smaller propagations (>=100 s/c revs). For longer propagations (>=1000 s/c revs) the final relative error in the computation is 1E-11 (approximately).

It is well known that after large number of space craft revolutions both the STM and STT become very large. On the CPU computing these sensitivities leads to rounding errors as large matrices are multiplied by small matrices at each successful integration step. While on the GPU, as the matrices are reduced in thread blocks in parallel, they are always of approximately the same order during multiplication. Even when they are finally reduced on the CPU they have a more stable rounding error behavior, as the matrices are again of the same order approximately. This explains the increase in relative error as the number of revolutions of the spacecraft increases. The STM from the GPU are therefore closer to the truth than those computed on the CPU for long propagations.

Conclusion

We propose a new tool capable of computing first and second order sensitivities in parallel for gradient based numerical optimization. The proposed tool implements a heterogeneous algorithm, utilizing both the CPU and GPU concurrently to achieve substantial performance increase. We are able to compute the first order sensitivities for almost no extra computational cost than compared to integrating just the states on the CPU. As for the second order sensitivities, we are able to compute them at up to 14 times faster for a 2-body trajectory integration example. The error in the sensitivities calculated by our tool when compared with their CPU counterparts is 1E-13 approximately. Further, our implementation is numerically more stable for long trajectory propagations compared to an equivalent CPU implementation. The software is general in its design and can applied to any gradient based optimization problem which requires fast and accurate sensitivities.

As part of future work, implementing automatic sensitivity calculation using numeric differencing and other approaches on the GPU is a promising area to work towards. Enabling real time sensitivity generation on the GPU using ephemeris data is also being considered.

Given the performance, accuracy and generality of our tool, it is well suited for application to a wide range of numerical optimization problems. Problems which are intractable due to their high computational complexity and/or dimension can now be attempted more readily without the burden of slow sensitivity calculations.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the Nvidia CUDA community forums for their valuable inputs.

REFERENCES

[1] S. F. P. Saramgo and S. V. JR, “Optimization of the trajectory planning of robot manipulators taking into account the dynamics of the system,” *Mechanism and machine theory*, 1998, pp. 883–894.

[2] D. E. Clough and F. W. Ramirez, “Mathematical modeling and optimization of the dehydrogenation of ethylbenzene to form styrene,” *American Institute of Chemical Engineers Journal*, Vol. 22, 2004.

[3] G. J. Whiffen and J. A. Sims, “Application of a novel optimal control algorithm to low-thrust trajectory optimization,” *Proceeding of the 11th Annual AAS/AIAA Space Flight Mechanics Meeting*, 2001, pp. 1524–1540.

[4] W. Feehery and P. Barton, “Dynamic optimization with state variable path constraints,” *Computers and Chemical Engineering*, Vol. 22, No. 9, 1998, pp. 1241–1256.

[5] S. Sayan and A. Kiraci, “A Numerical Optimization Algorithm for Identification of Policy Options to Rehabilitate a Publicly Managed, Pay-As-You-Go Based Pension System,” Computing in Economics and Finance 1999 932, Society for Computational Economics, Mar. 1999.

[6] A. E. Bryson and Y.-C. Ho, “Applied Optimal Control,” 2006.

[7] R. P. Russell, “Primer Vector Theory Applied to Global Low-Thrust Trade Studies,” *Journal of the Guidance,Control and Dynamics*, Vol. 30, 2007, pp. 460–472.

[8] R. P. Russell, *Global Search and Optimization for Free-Return Earth-Mars Cyclers*. 2004.

[9] H. Shen and P. Tsiotras, “Optimal Two-Impulse Rendezvous Between Two Circular Orbits Using Multiple-Revolution Lambert’s Solutions,” *Journal of Guidance, Control, and Dynamics*, Vol. 26, 2003, pp. 50–61.

[10] P. Rao, B. Sutter, and P. Hong, “Six-Degree-of-Freedom Trajectory Targeting and Optimization for Titan Launch Vehicles,” *Journal of Spacecraft and Rockets*, Vol. 34, No. 3, 1997, pp. 341–346.

[11] D. G. Hull, “Numerical Derivatives for Parameter Optimization,” *Journal of Guidance, Control, and Dynamics*, Vol. 2, 1979, pp. 158–160.

[12] P. E. Gill, W. Murray, and M. A. Saunders, “SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization,” *Society for Industrial and Applied Mathematics*, Vol. 47, 2005, pp. 99–131.

[13] A. E. Petropoulos and R. P. Russell, “Low-Thrust Transfers using Primer Vector Theory and a Second-Order Penalty Method,” August 2008.

[14] J. T. Betts, “Survey of Numerical Methods for Trajectory Optimization,” *Journal of Guidance, Control, and Dynamics*, Vol. 21, 1998, p. 193.

[15] R. S. Park and D. J. Scheeres, “Nonlinear Mapping of Gaussian Statistics: Theory and Applications to Spacecraft Trajectory Design,” *Journal of Guidance, Control and Dynamics*, Vol. 29, 2006, pp. 1367–1375.

[16] C. Bischof, L. Green, K. Haigler, and J. T.L. Knauff, "Parallel Calculation of Sensitivity Derivatives for Aircraft Design using Automatic Differentiation," *5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization Conference*, 1994.

[17] R. Byrd, R. Schnabel, and G. Shultz, "Parallel quasi-Newton methods for unconstrained optimization," *Journal of Spacecraft and Rockets*, Vol. 42, No. 1-3, 1988, pp. 273–306.

[18] R. Biedron, J. Samareh, and L. Green, "Parallel Computation Of Sensitivity Derivatives With Application to Aerodynamic Optimization of a Wing," *Computational Aerosciences Workshop (NASA Ames Research Center)*, 1998.

[19] D. Conforti, L. Luca, L. Grandinetti, and R. Musmanno, "A parallel implementation of automatic differentiation for partially separable functions using PVM," *Parallel Computing*, No. 22, 1996, pp. 643–656.

[20] R. H. Battin, "An Introduction to the Mathemat-ics and Methods of Astrodynamics," *AIAA Education Series*, 1999.

[21] R. S. Park and D. J. Scheeres, "Nonlinear Semi-Analytic Methods for Trajectory Estimation," *Journal of Guidance, Control and Dynamics*, Vol. 30, 2007, pp. 1668–1676.

[22] P. Sengupta, S. R. Vadali, and K. T. Alfriend, "Second-order state transition for relative motion near perturbed, elliptic orbits," *Celestial Mechanics and Dynamical Astronomy*, Vol. 97, 2006, pp. 101–129.

[23] G. Lantoine and R. P. Russell, "A Hybrid Differential Dynamic Programming Algorithm for Robust Low-Thrust Optimization," *AAS/AIAA Astrodynamics Specialist Conference and Exhibit*, 2008.

[24] P. Patel and D. J. Scheeres, "A Non-Linear Optimization Algorithm," *AAS*, 2008.

[25] R. G. Belleman, J. Bedorf, and S. F. P. Zwart, "High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA," *New Astronomy*, 2008.

[26] N. Arora, A. Shringarpure, and V. R.W, "Direct N-body Kernels for Multicore Platforms," *International conference on parallel processing*, 2009.

[27] M. Januszewski and M. Kostur, "Accelerating numerical solution of Stochastic Differential Equations with CUDA," *Computational Physics*, 2009.

[28] D. Robilliard, V. Marion, and C. Fonlupt, "High performance genetic programming on GPU," *International Conference on Autonomic Computing*, 2009.

[29] I. S. Ufimtsev and T. J. Martnez, "Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation," *Journal of Chemical Theory and Computation*, Vol. 4, 2008, pp. 222–231.

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro IEEE*, 2008.

[31] NVIDIA, "NVIDIA CUDA Programming Guide 2.0," *Documentation*, 2008.

[32] P. J. Prince and J. R. Dormand, "High order embedded RungeKutta formulae," *Journal of Comput. Appl. Math*, 1981.