

Accelerating Lambert's Problem on the GPU in MATLAB

A Senior Project

presented to

the Faculty of the Aerospace Engineering Department  
California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

by

Nathan Parrish

June, 2012

© 2012 Nathan Parrish

# Accelerating Lambert's Problem on the GPU in MATLAB

Nathan L. Parrish<sup>1</sup>

*California Polytechnic State University, San Luis Obispo CA, 93407*

The challenges and benefits of using the GPU to compute solutions to Lambert's Problem are discussed. Three algorithms (Universal Variables, Gooding's algorithm, and Izzo's algorithm) were adapted for GPU computation directly within MATLAB. The robustness of each algorithm was considered, along with the speed at which it could be computed on each of three computers. All algorithms used were found to be completely robust. Computation time was measured for computation within a for-loop, a parfor-loop, and a call to the MATLAB command 'arrayfun' with gpuArray-type inputs. Then, a Universal Variables Lambert's solver was written in CUDA and compiled for use within MATLAB, where each solution to Lambert's problem was run as a separate thread. The CUDA-based solver was the fastest of all, solving  $1.5 \times 10^6$  solutions per second on a laptop GPU and  $5.0 \times 10^6$  solutions per second on a high-end consumer desktop GPU. The net result is that the Universal Variables algorithm in CUDA runs 76x faster than the same algorithm on the CPU and 3.5x faster than the industry-standard Fortran solver.

## I. Introduction

Lambert's Problem is a classical astrodynamics problem, the solution to which is extremely useful for trajectory design. The problem can be stated as follows: Given two position vectors relative to a central body and the time of flight between them, find the velocity vector at each position. Lambert's problem is commonly solved billions of times for interplanetary trajectory design, so the computational efficiency is extremely important. Traditionally, most computations are performed on the CPU (Central Processing Unit) of a computer. Modern consumer CPUs have 2-8 cores that operate at around 3 GHz. Current higher-end consumer GPUs (Graphics Processing Units) have 300-1000 cores that operate at around 500-1000 MHz. Numerous studies have found that well-suited algorithms can be accelerated by 10-100x by computing on the GPU instead of the CPU<sup>1</sup>.

Lambert's problem cannot be solved analytically; therefore, a solution must be found iteratively. Various algorithms exist for solving Lambert's problem, each with strengths and weaknesses. The main trade is between speed and robustness: The fastest algorithms tend to fail more often, while the more robust algorithms take longer to compute. A brief history on Lambert's Problem is included below:

### A. History of Lambert's Problem

J.H. Lambert (1728-1777) was a German astronomer, physicist, and mathematician. His famous theorem states that the transfer time from Point A to Point B is independent of the orbit's eccentricity and depends only on the sum of the magnitudes of the position vectors, the semimajor axis, and the length of the chord joining Points A and B<sup>2</sup>.

Many mathematicians have developed improved solutions since Lambert first came up with his theorem. Dozens of papers have been published on Lambert's Problem, especially in the last 50 years. Some notable improvements to solving Lambert's problem have come from: Gauss (1809); Lancaster and Blanchard (1969); Bate, Mueller, and White (1971); Kaplan (1976); Battin (1987); Gooding (1990); Prussing and Conway (1993); and Thorne (2004).

### B. Applications of Lambert's Problem

The primary applications of Lambert's problem are orbital determination and trajectory design. This project was developed with trajectory design in mind, where billions of solutions to Lambert's Problem must be solved. Lambert's problem ignores perturbations and assumes impulsive-only burns, but an optimized Lambert's trajectory can be used as the starting point for higher-fidelity models or as the initial guess for an electric-propulsion trajectory<sup>3</sup>. Lambert's problem is also used to generate "porkchop" plots for interplanetary transfers. Compared to higher-fidelity models, analyzing a trajectory with Lambert's Problem is very computationally efficient. However, due to the sheer number of solutions that must be considered for some applications, running a brute force search could still require many days of computer time. By optimizing Lambert's solvers, we can minimize the amount of

---

<sup>1</sup> Undergraduate Student, Aerospace Engineering, 1 Grand Ave, San Luis Obispo, CA, Student Member.

time put into initial mission design. Traditional computing methods perform every action serially – that is, one action cannot commence until the previous one has finished. For trajectory optimization, each Lambert’s solution is totally independent of every other solution, so moving the computation of Lambert’s problem onto the GPU can parallelize this process and significantly speed up the total computation.

### C. History of GPU Development

Over the past 20 years, GPUs have been a rapidly-developing technology. Each generation of GPUs has been able to perform a broader range of calculations faster than the generation before. When 3-dimensional graphics were initially introduced into consumer markets in the early 1990s, GPUs consisted of highly specific hardware that could perform only a limited set of math. Initially, the “graphics pipeline” was hard-wired to first input everything as triangles defined by [4x1] matrices representing either points or vectors; transform these points and vectors to simulate movement; compute the lighting on each triangular surface; and finally, transform the triangles into Cartesian pixel coordinates for display<sup>4</sup>. From their beginning, GPUs were designed to process an enormous amount of data, but since they only have to render a picture 60 times per second, the speed at which they complete a given calculation is not as important as it is for a computer’s central processor<sup>4</sup>.

In 2001, graphics cards started to move toward a programmable pipeline, where a unified grid of processors capable of performing multiple calculations replaced the hard-wired, fixed-purpose units from before. The additional math allowed by this shift made it possible for graphic artists to create new, more realistic visual effects for videogames. The latest step in GPU development is the use of “unified shaders”, which merges the hardware required for processing vertex, geometry, and pixel shaders into “one large grid of data-parallel floating-point processors general enough to run all these shader workloads”<sup>4</sup>. All of these developments and innovations were fueled by the huge market desire for better-looking videogames, but GPU technology has reached a point where it is useful for much more than gaming now. Whereas previously, scientists who wanted to take advantage of the parallel processing capabilities of the GPU had to disguise their science inputs as graphics objects and “trick” the GPU into working for them, we can now code software for the GPU in much the same way as we code standard code for the CPU<sup>1</sup>. The development of the General Purpose GPU (GPGPU) has made every modern GPU a powerful parallel processor that, for some algorithms, can be exploited for tremendous computational speedups.

## II. Coding for GPU in MATLAB

MATLAB has two ways of executing code on the GPU: non-compiled and compiled. Running non-compiled code on the GPU is very simple – in many cases, the same code can be used for CPU or GPU computation. However, this method carries a lot of overhead and becomes extremely slow if computation is not vectorized. Non-compiled code on a GPU ranges from orders of magnitude slower to a few times faster than on a CPU. A much faster option is to use ‘arrayfun’ to call a function. ‘Arrayfun’ executes a function on each element of an input matrix and outputs the results to an output matrix of the same size as the input. When ‘arrayfun’ is used with input data of the class *parallel.gpu.GPUArray*, MATLAB compiles the function into a .PTX (parallel thread execution) file, transfers any function inputs that are stored in the host memory (RAM) to the GPU device memory, and executes the function in parallel on each element of the input matrix (or matrices). Compilation to .PTX occurs the first time the code is called with ‘arrayfun’ in a MATLAB session. Thereafter, compilation only occurs when the function code has changed. Function computation via ‘arrayfun’ on the GPU occurs 1-2 orders of magnitude faster than computation on the CPU, but there are some strict rules to follow in the function code.

First and foremost, only a handful (~100) of built-in MATLAB functions are currently optimized to run on the GPU (as of R2011b). Each new revision of MATLAB has brought more GPU-friendly functions, but the fact remains that code must be written at a much lower level than usual in order to be compiled for GPU.

One of the more frustrating restrictions is the fact that array concatenation is not supported on the GPU. Some simple examples of code that MATLAB cannot compile for the GPU are:

```
a = [1, 3, 5]
b = [4, 6, 8]
c = a(2)*b(3)
b(2) = 10
```

MATLAB cannot understand any of the above lines for GPU compilation. These could be rewritten for the GPU as:

```
a_1 = 1, a_2 = 3, a_3 = 5
b_1 = 4, b_2 = 6, b_3 = 8
c = a_2*b_3
b_2 = 10
```

We can see from this simple example that most functions will quickly devolve into a mess of 1x1 matrices where vectors once stood. For the Lambert's Problem algorithms coded in this project, this was a hassle and an annoyance, but ultimately did not take too long to incorporate into existing algorithms.

Another rule to follow is that no anonymous functions, subfunctions (in the same file), or user-made functions (in another file) can be called from the parent function passed into 'arrayfun'. The whole algorithm must exist entirely in one function in a single .m file, and can only call the MATLAB functions that are GPU-optimized. In a typical MATLAB function, tedious portions of code that are called multiple times tend to be placed in a subfunction and placed at the end of the parent function or in their own .m function file. Doing so allows the programmer to keep his or her work more organized and easily-accessible to others. This practice also lets functions and scripts be written with fewer total lines of code and makes debugging code much easier. To compile GPU code with 'arrayfun', however, MATLAB requires neglecting this habit. Everything must be written as a single function.

Finally, variables stored on the GPU device memory cannot change complexity mid-function. Variables of type *parallel.gpu.GPUArray* have the attributes 'Size' (i.e. [1x1], [3x1]), 'ClassUnderling' (i.e. 'double', 'single', 'logical'), and 'Complexity' (i.e. 'real' or 'complex'). Any variable which is initially declared as 'real' (no imaginary component) must remain 'real', and any variable declared as 'complex' must remain 'complex'. In a strict mathematical definition, a real number is simply the real component of a complex number. Computers, though, know real and complex numbers as two separate types of data. Standard MATLAB code on the CPU can change variable type behind the scenes (frequently without the user ever being aware), but MATLAB code for GPU computation must be more explicitly defined. Also, MATLAB functions such as `sqrt(x)` and `power(x,y)` are only allowed to have complex outputs when the inputs are also complex.

In this project, this limitation was encountered when `sqrt(x)` was used with a negative value of `x`. To get around this problem, every variable that has the potential to ever be complex must be defined as complex at its first declaration. Complex variables with zero imaginary component can be created as follows:

```
var = complex(real_component, imag_component)
```

where `imag_component` equals zero. It was found that while specifying variables' complexity is required on the GPU, doing so slowed down computation on the CPU. A later fix found that complex numbers could be avoided entirely by ensuring that `sqrt(x)` and `power(x,y)` never had negative numbers passed into them. Avoiding complex numbers was effective for this project, but others might require complex numbers.

As will be seen in the Testing Procedures section, following all these rules significantly accelerated code for both CPU and GPU computation.

### III. Coding for GPU in CUDA

#### A. Preparing a computer for CUDA

CUDA (Compute Unified Device Architecture) is an NVIDIA-specific GPU programming language based largely on C. It competes with OpenCL (Open Computing Language) as a parallel computing language. OpenCL is open-source standard developed by the Khronos Group, whose goal is to make the same code run in parallel on all devices<sup>2</sup>. While OpenCL is more platform-independent than CUDA, CUDA enjoys more developer support and a friendlier set of tools to get programmers started. For this reason, CUDA was chosen over OpenCL for this project.

The largest hurdle to overcome in this project was learning the C language. Once armed with a basic understanding of C, making the modifications necessary for CUDA was simple. In order to get a computer set up for CUDA programming, the following steps must be taken:

---

<sup>2</sup> "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems", <http://www.khronos.org/opencl/>

1. Install Microsoft Visual Studio. This provides the backbone for compiling C code.
2. Install the CUDA Toolkit. This adds on to the Microsoft Visual Studio compiling tools, making it possible to compile CUDA code. After installing, Visual Studio must be configured to recognize the CUDA tools.
3. Install/update the NVIDIA CUDA graphics driver. This allows the computer to run compiled CUDA code. For most computers with NVIDIA GPUs, this will already be installed. This is the only software required to run compiled CUDA code.
4. Install the CUDA SDK. This includes many examples of CUDA codes.

### B. Compiling and running CUDA code

Once a computer is set up for CUDA, .cu files may be compiled from a command prompt (Windows) or a terminal (Unix). An example command given for this project is:

```
nvcc filename.cu -m32 -arch=sm_13 -ptx
```

This command is broken down in Table 1, below.

**Table 1. Explanation of a command to compile CUDA code for MATLAB**

| Command or option:  | What it means:   |
|---------------------|--|
| <b>nvcc</b>         | Compile CUDA code  |
| <b>filename.cu</b>  | The file name of the source code   |
| <b>-m32 OR -m64</b> | Specify compilation for 32-bit or 64-bit use                                   |
| <b>-arch=sm_13</b>  | Compile for the compute 1.3 GPU architecture (allows double-precision numbers) |
| <b>-ptx</b>         | Output a .ptx file (required by MATLAB)  |

Running this command creates a .ptx file which can be passed into MATLAB along with the .cu source file to create a kernel object in MATLAB. A kernel is equivalent to a function in MATLAB and is the core code which is distributed to each multiprocessor in the GPU to run in parallel. In MATLAB, a kernel object is created with the command:

```
k = parallel.gpu.CUDAKernel('filename.ptx','filename.cu');
```

and executed with the command:

```
[out1, out2] = feval(k, in1, in2);
```

where *k* is the kernel, *in1* and *in2* are the kernel inputs, and *out1* and *out2* are the kernel outputs. As with using `arrayfun` on the GPU, all the inputs to the kernel must be of the type `parallel.gpu.GPUArray` (data held on the GPU). In MATLAB, data can be sent to the GPU with the `gpuArray()` command and returned from the GPU with the `gather()` command.

GPU programming offers some interesting options for how a kernel should be run. In CUDA, we can set how many blocks of threads we want to use and how many threads within each block. The number of threads within each block is set via a [1x3] vector of indices, while the number of blocks of threads is set via a [1x2] vector of indices. For all of the GPUs used in testing for this project, the max number of threads per block is 512, and the max number of blocks is 65,535. Therefore, the maximum number of threads allowed for any one kernel call is 65,535x512, or about 3.3x10<sup>6</sup>. These different indexing options allow us a lot of flexibility when it comes to identifying each individual thread. Within the CUDA kernel, the index of the current thread is given by a few built-in variables called `threadIdx` (a structure with x, y, and z components to identify the thread within a block) and `blockIdx` (a structure with x and y components to identify the block within the entire grid of blocks)<sup>1</sup>.

### C. Problems encountered

In MATLAB, it was found that the use of CUDA kernels is very sensitive and MATLAB would seem to randomly fail. In testing, sometimes even running the same exact code twice would yield different results. For instance, on Computer 3, *sometimes* the first 192 solutions returned by MATLAB would be an (apparently) random wrong answer, while all the solutions after that would be correct. Other times, MATLAB would time out while waiting for the GPU to return data. Yet other times, MATLAB would time out while waiting for the CUDA kernel to launch in the first place. The error reporting for CUDA/MATLAB integration is generally useless, as evidenced by the following example, which showed up frequently:

```
An unexpected error occurred during CUDA execution. The CUDA error was:  
CUDA_ERROR_UNKNOWN.
```

These issues were extremely frustrating and their source remains a mystery. After much trial and error, it was found that changing the size of the inputs and outputs had a drastic effect on whether MATLAB would crash or not. Again, however, the problems were intermittent. Sometimes, a given size input would work fine, but running a code a second time (without changing anything) would make the graphics driver crash. It is not clear whether these problems lie with MATLAB or with CUDA (or somewhere in the middle), but a workaround was found. By trial and error, some input sizes were found which always returned the same, correct results. After that, input size was not changed any more, and everything ran perfectly.

## IV. Testing Procedures

Three different algorithms for solving Lambert's Problem were adapted for GPU computation. The algorithms used were: Universal Variables (as given by Curtis), Gooding's algorithm, and Izzo's algorithm. Each algorithm was tested on the CPU and GPU of three different machines. The computers used in testing are described in Table 2.

Table 2. Specs of the computers used for testing

|                                | Computer 1                    | Computer 2                | Computer 3             |
|--------------------------------|-------------------------------|---------------------------|------------------------|
| <b>Central processor (CPU)</b> | Intel Pentium Dual Core E6600 | Intel Core i7 960         | Intel Core i5-2450M    |
| <b>CPU cores</b>               | 2                             | 4                         | 2                      |
| <b>RAM</b>                     | 3.5 GB                        | 8 GB                      | 8 GB                   |
| <b>Graphics card (GPU)</b>     | NVIDIA GeForce GT 430         | NVIDIA GeForce GTX 560 Ti | NVIDIA GeForce GT 540M |
| <b>GPU cores</b>               | 96                            | 384                       | 96                     |
| <b>Graphics memory</b>         | 1 GB                          | 1 GB                      | 1 GB                   |
| <b>Operating system</b>        | Windows 7, 32-bit             | Fedora 14, 64-bit         | Windows 7, 64-bit      |
| <b>MATLAB version</b>          | R2012a                        | R2011b                    | R2012a                 |

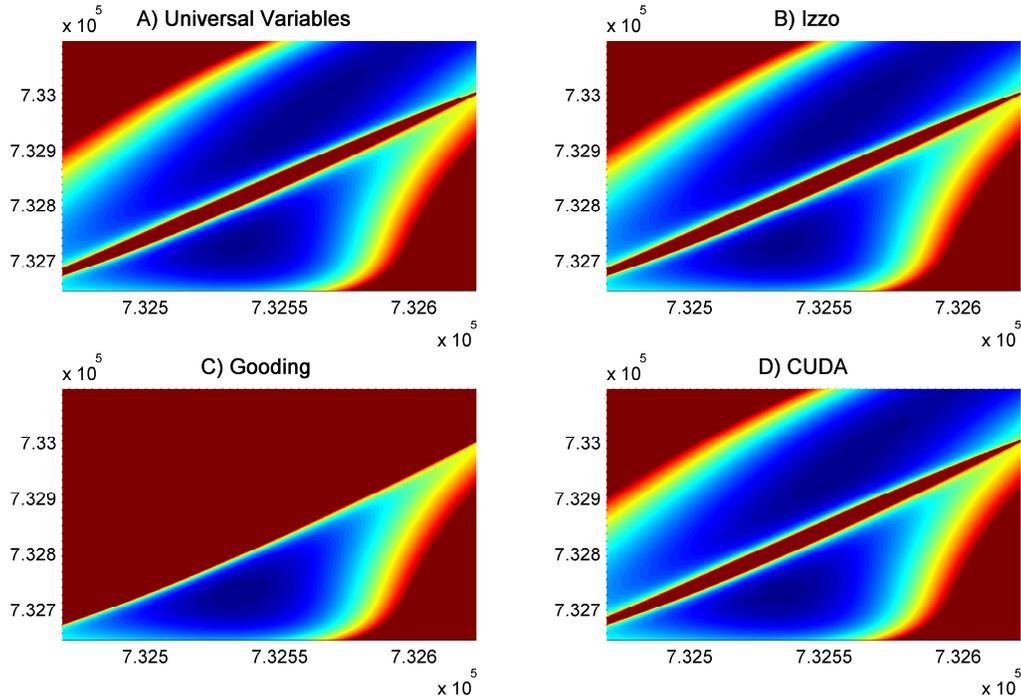
### A. Testing for Speed

The algorithms were tested by each performing the same set of calculations for a transfer from Jupiter to Uranus at various times (such as might be done when designing an interplanetary gravity-assist trajectory). The starting and ending position vectors were calculated once by interpolating ephemeris data (downloaded from JPL's Horizons tool) for each of the planets at one million discreet time steps. Various-length subsets of these position vectors were input into the Lambert's Problem solvers, and the time to calculate each subset was measured five times and averaged. The number of Lambert's Problem solutions requested was varied between 1 and  $10^6$  to examine the effect that input size has on computational efficiency.

### B. Testing for Accuracy and Robustness

The results from each of the algorithms were compared to the results of a publicly-available Lambert's Problem solver from the European Space Agency that is known to be good. A porkchop plot was generated by each

Lambert’s solver to prove that it was robust over a wide variety of transit options. The porkchop plots in Figure 1 are for the 2005 Earth-to-Mars opportunity.



**Figure 1. Porkchop plots generated by four methods**

In the plots, the x-axis represents the possible launch dates, the y-axis represents the possible arrival dates, and the colors represent the C3 required to leave Earth. Here, the values themselves are not important – only that the four methods agree with each other. As evidenced by Figure 1, all the methods except for Gooding’s return the exact same porkchop plot. The one generated with Gooding’s algorithm lacks the upper half because it was not set up to calculate “long-way” trajectories for this project. Thus, only the “short-way” trajectories return a C3 lower than the cutoff point for this plot. Besides that difference, all four porkchop plots are identical (within some very tight tolerance).

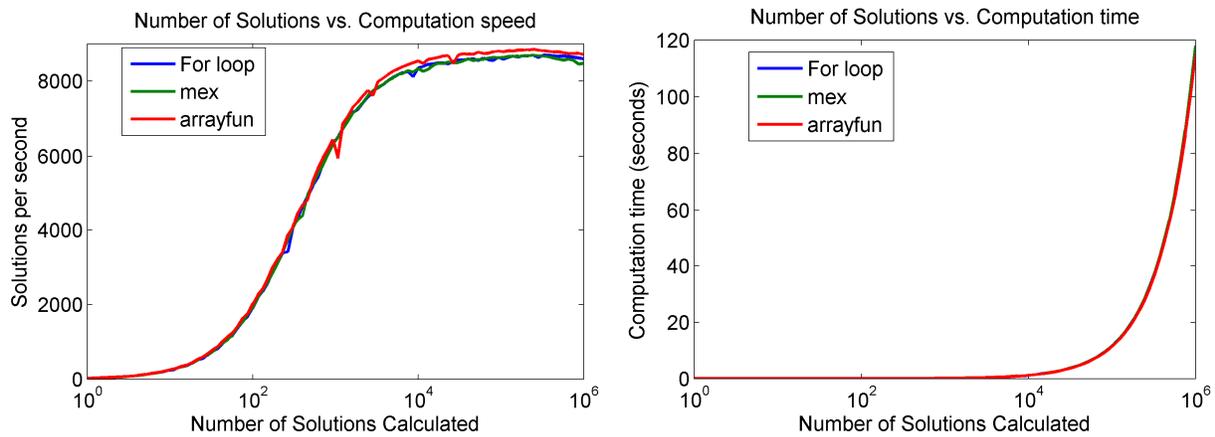
## V. Comparing the Algorithms in MATLAB

### A. Universal Variables

The Universal Variables method works for all types of transfer orbits. It uses the universal variable to relate energy and angular momentum, thus simplifying many equations. The Universal Variables method can be used for a variety of algorithms. The one used here is from Curtis<sup>2</sup> and uses Newton-Raphson iterations to solve for the value of an intermediary variable which is a function of both the semimajor axis and the universal variable.

One significant disadvantage of the Universal Variables algorithm is that it does not allow for multiple-revolution solutions (where the spacecraft completes at least 1 complete revolution around the central body before arriving at the specified position). Other formulations of the Universal Variables method can compute multi-rev solutions<sup>5</sup>, but those were not considered in this project.

The Universal Variables algorithm was tested on CPU using three methods: for loop, arrayfun, and mex. The for-loop method was the simplest to code and involved simply looping through the inputs. To use arrayfun, the algorithm was called on the CPU the same way as it is for GPU computation. To run the code as a mex function, MATLAB’s *codegen* tool was used to generate a C++ compiled version of the code. Creating a mex file from a typical MATLAB function requires adjusting the original code to follow certain rules. However, MATLAB’s rules for making code compatible with GPU compilation are extremely similar to the rules for CPU compilation, so no change was necessary to make the code mex-compatible. The results of the comparison between for-loop, arrayfun, and mex are shown in Figure 2.



**Figure 2. Comparing for-loop, arrayfun, and mex on CPU for Universal Variables algorithm**

As can be seen from the above figure, all three computation methods on the CPU were almost exactly the same. Calculation via arrayfun has a slight performance advantage, but not much. For most applications, the speed gain from using arrayfun is not worth the extra programming.

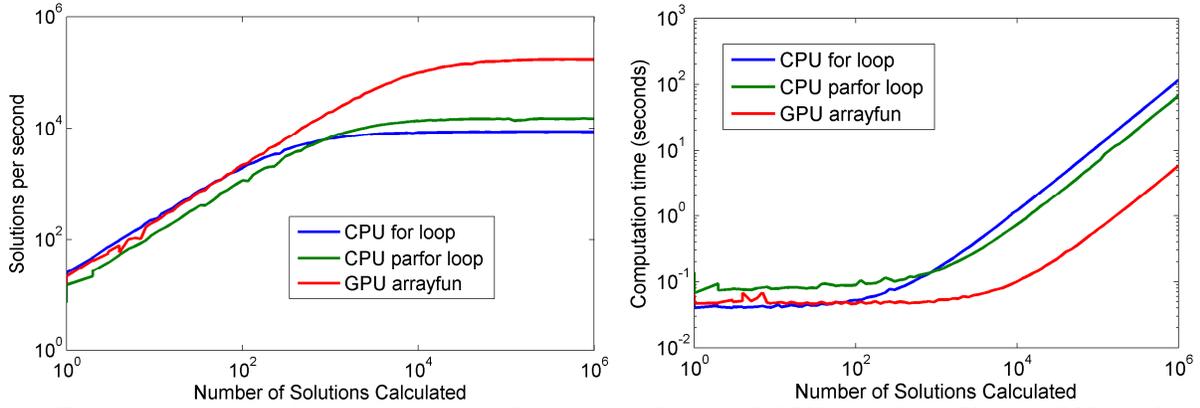
The Universal Variables algorithm was then tested in two additional ways: within a ‘parfor’ loop (on the CPU) and with ‘arrayfun’ on the GPU. Both of these computation methods were tested on each computer (described previously). It was found that, for Computer 1, using a ‘parfor’ loop resulted in a max speed gain of 1.7x. The same was seen for Computer 3. On Computer 2, the ‘parfor’ loop gave a max speed gain of 3.5X. Since Computer 1 has two cores, its parfor-loop performance would ideally be exactly 2x its for-loop performance. Computer 2 has four cores, so we would expect its parfor-loop performance to be exactly 4x its for-loop performance. These speed advantages are not quite reached, however, because there is data overhead and not all the cores are completely active all the time during computation.

The speedup attained from GPU computation is much more substantial. On Computer 1, GPU performance was 20x faster than for-loop performance. On Computer 2, the performance was 75x faster. These results are summarized in Table 3. Important to note is that the speedup of a given algorithm is hugely dependent on the architecture of both the CPU and the GPU installed in a computer. One computer might perform better with one algorithm, while another might run a different algorithm faster.

**Table 3. Speedup of Universal Variables algorithm**

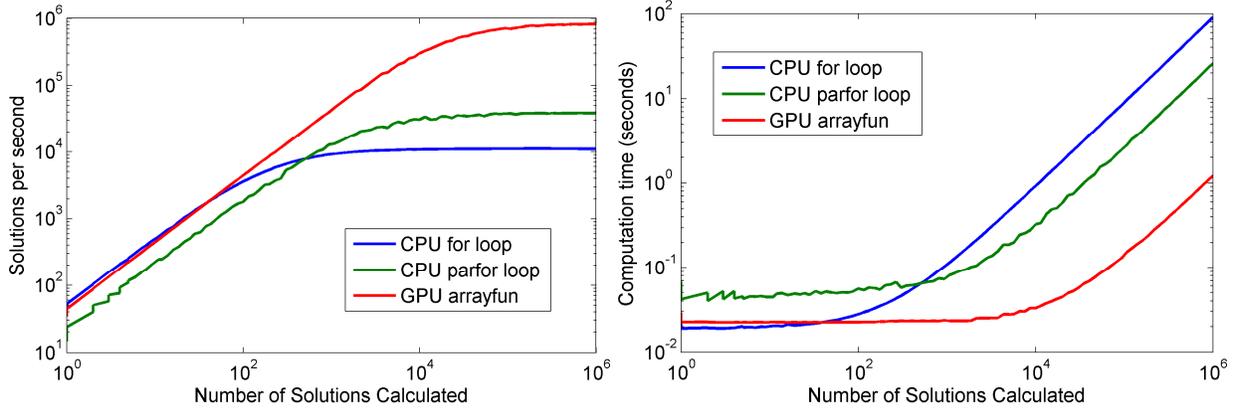
|                     | Computer 1 speedup | Computer 2 speedup | Computer 3 speedup |
|---------------------|--------------------|--------------------|--------------------|
| <b>for-loop</b>     | 1x                 | 1x                 | 1x                 |
| <b>parfor-loop</b>  | 1.7x               | 3.5x               | 1.7x               |
| <b>GPU arrayfun</b> | 20x                | 75x                | 12x                |

These speedups are not constant for all sizes of input matrices, though. As Figure 3 shows, the simple CPU for-loop is even faster than parfor and GPU arrayfun computation when the number of solutions calculated is ~100 or less. Interesting to note is that at some point, each computation method’s performance plateaus. On Computer 1, this point was at about 1,000 solutions for the for-loop, 10,000 for the parfor-loop, and 100,000 for the GPU arrayfun.



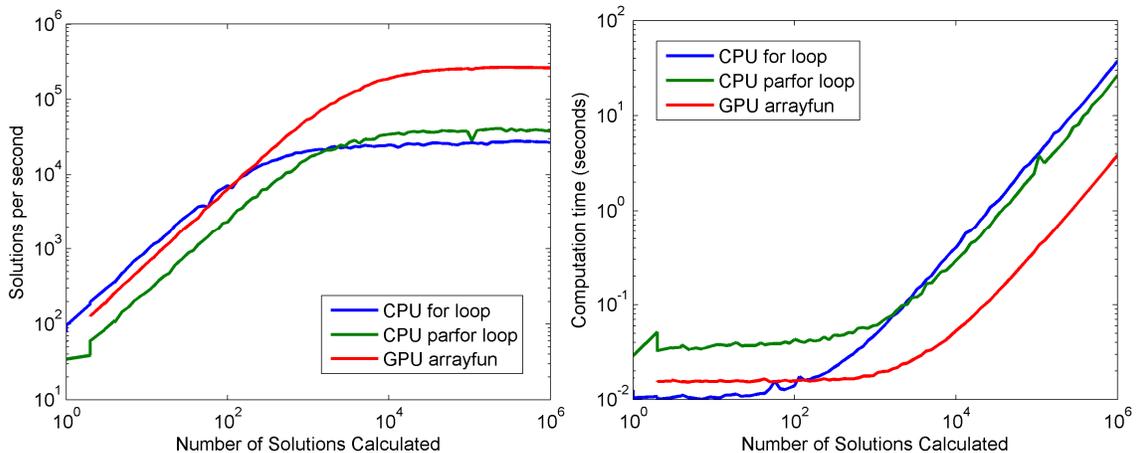
**Figure 3. Computer 1 performance: for-loop, parfor-loop, and GPU arrayfun – Universal Variables**

Figure 4 (below) shows similar trends for the Universal Variables algorithm on Computer 2.



**Figure 4. Computer 2 performance: for-loop, parfor-loop, and GPU arrayfun – Universal Variables**

Computer 2 is clearly a faster computer, but we see the same trend. For-loop computation is best if  $\sim 100$  or fewer solutions are being calculated; otherwise, parfor or GPU arrayfun computation is much faster.



**Figure 5. Computer 3 performance: for-loop, parfor-loop, and GPU arrayfun – Universal Variables**

Interesting to note is the fact that GPU arrayfun computation takes almost exactly the same amount of time for a single solution as it does for 1,000 solutions. This is likely because it takes at least 1,000 calculations being passed into the GPU before all of the hundreds of cores are used efficiently.

We can see that the same trends are true across all three computers. The main difference between computers is the scaling of the vertical axis – the proportional differences are roughly the same across computers. The trends from Figures 3, 4, and 5 are repeated more-or-less for the other two algorithms, so equivalent figures are not printed for Gooding’s algorithm and Izzo’s algorithm.

### B. Gooding

Gooding’s procedure also works for all types of transfer orbits. An advantage to Gooding’s algorithm is that it can consider single- or multiple-revolution solutions. It is based on Lancaster’s approach and uses Halley’s cubic iteration process to iterate on the parameter at its heart. It chooses initial estimates for the variable to iterate such that the algorithm always converges to very tight tolerances within three iterations<sup>6</sup>.

The MATLAB code used for the Gooding method was translated line-by-line from three Fortran subroutines that Gooding published<sup>6</sup>. In Fortran, the algorithm can process roughly  $3.5 \times 10^5$  solutions per second (on the CPU). Its MATLAB equivalent runs at a max of  $\sim 25,000$  solutions per second. Further optimization of the MATLAB code would undoubtedly speed it up, but it will never perform as well as it does in Fortran. The main reason for this is that Fortran is a compiled language, so fewer instructions end up getting sent to the CPU.

### C. Izzo

Izzo’s algorithm describes the generic solution of the boundary condition problem through the use of an intermediary variable that is a function of the semimajor axis. The method works for all transfer orbit types and for single or multiple revolutions<sup>7</sup>. It claims to be able to solve all possible geometries with the same amount of computation. This algorithm is more recent than the Universal Variables or Gooding algorithms, so it has not been tested or verified as extensively.

The Izzo algorithm is made publicly available through the European Space Agency in a MATLAB function. That function was optimized by Oldenhuis and posted to Mathworks Central, where it was accessed for this project<sup>8</sup>.

### D. Summary of Results

Figure 6 (below) shows the performance, on the CPU, of each of the algorithms that were tested. Gooding’s and Izzo’s algorithms are both much faster than Universal Variables, but Izzo’s has a slight advantage at higher numbers of solutions.

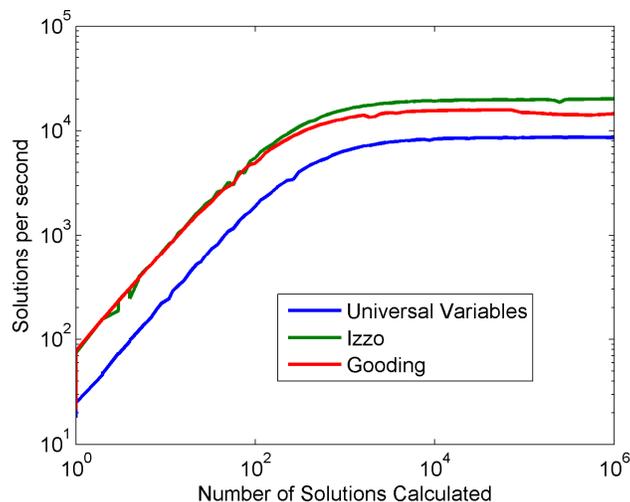
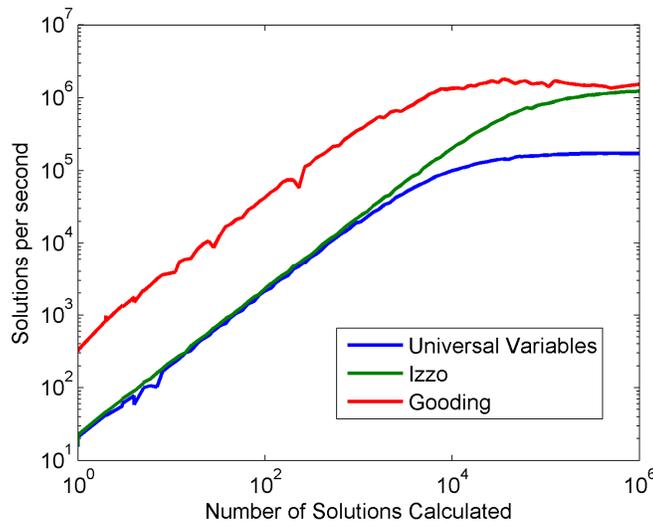


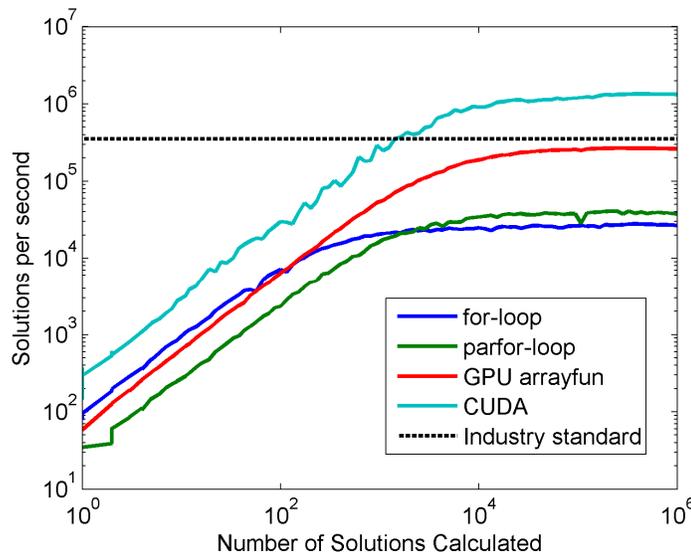
Figure 6. Comparison of all tested methods on CPU (Computer 1)

Figure 7 (below) compares the algorithms’ performance once again, but this time on the GPU (using MATLAB’s arrayfun). We can see that Gooding’s method is substantially faster than either Izzo or Universal Variables for low numbers of solutions. However, the performance of Gooding’s method levels off sooner than the others, so it ends up being about the same speed as Izzo’s method for large sets of calculations.



**Figure 7. Comparison of all tested methods on GPU (using MATLAB arrayfun, Computer 1)**

Following the complete testing of all three algorithms on both CPU and GPU, the Universal Variables method was chosen as the one algorithm to code in CUDA. Although the Universal Variables method is not as fast as the other two algorithms looked at in this project, it is much simpler to understand and much easier to code. The purpose of this project was to prove a concept – future work could include moving Gooding’s or Izzo’s algorithm to CUDA for the ultimate speed boost. As evidenced by Figure 8, CUDA does have a significant speed boost over the MATLAB GPU code compiled with arrayfun.



**Figure 8. Comparison of the Universal Variables algorithm across all computation methods (Computer 3)**

Figure 8 summarizes the success of this project and hints at the implications of GPU computing as a whole. The four colored lines are all using the Universal Variables algorithm, but each with a different computation method. The cyan line represents the results of the algorithm coded in CUDA C and is clearly the fastest by far. The black dotted line represents the peak speed of the industry standard Lambert’s solver – Gooding’s algorithm running in Fortran. We see in Figure 8 that the CUDA implementation of Universal Variables is 3.5x faster than Gooding’s method on Fortran. Even a less-efficient algorithm, when accelerated with GPU computation, can be faster than the speediest CPU algorithm.

The maximum speedup of each algorithm is summarized in the following table:

**Table 4. Speedup of each algorithm, relative to for-loop speed of each algorithm (Computer 3)**

|                     | Universal Variables | Izzo | Gooding |
|---------------------|---------------------|------|---------|
| <b>for-loop</b>     | 1x                  | 1x   | 1x      |
| <b>parfor-loop</b>  | 1.7x                | 1.7x | 1.7x    |
| <b>GPU arrayfun</b> | 12x                 | 78x  | 139x    |
| <b>Fortran</b>      | N/A                 | N/A  | 26x     |
| <b>CUDA</b>         | 76x                 | N/A  | N/A     |

By comparing the speedup seen by each algorithm for the different computing methods, we can see that Gooding’s algorithm benefited most from GPU acceleration. This suggests that the algorithm is more compilation-friendly than other methods. The Gooding method would be the ideal choice for the translation to CUDA, but it was rejected because it is more difficult to code than Universal Variables.

## VI. Conclusion

Performing calculations on the GPU from within MATLAB proved to be a fairly straightforward process. Certain limitations must be taken into account, but multiple algorithms were accelerated by 1-2 orders of magnitude with relatively little work.

By coding the Universal Variables algorithm in CUDA C, we can get an even better feel for the potential of GPU computing. The Universal Variables algorithm was not nearly as fast as Gooding’s method when used with arrayfun in MATLAB. When coded in CUDA, however, it was 3.5x faster than even Gooding’s Fortran Lambert’s solver. In fact, GPU computation time begins to become negligible compared to the time required to transfer data to and from the GPU memory.

Future work would include coding Gooding’s method in CUDA, as that should be the fastest algorithm, period. Also, with more research into the low-level details of GPU programming, the CUDA code for the Universal Variables algorithm could undoubtedly be made more efficient. Looking at the bigger picture, the more code that is moved onto the GPU, the better. By performing more calculations on the GPU, data transfer between the device and host memory (the stumbling block for many GPU applications) can be minimized. Greater and greater portions of trajectory optimization algorithms can be parallelized and accelerated on the GPU.

Although there are still many kinks to work out with MATLAB’s use of the GPU, the end results are very promising. If, in three months worth of weekends, a college student can both learn CUDA and apply it to solve a well-studied problem fast enough to compete with the best CPU algorithms (refined over years by experienced programmers), then the world is clearly ready for a massively parallel computing future.

### References:

<sup>1</sup> Sanders, J. and Kandrot, E., *CUDA by Example: An Introduction*, Addison-Wesley Professional, Upper Saddle River, NJ, 2010.

<sup>2</sup> Curtis, H. D., *Orbital Mechanics for Engineering Students*, Elsevier, Burlington, MA, 2010.

<sup>3</sup> Landau, D. and Strange, N., “Near-Earth Asteroids Accessible to Human Exploration with High-Power Electric Propulsion”. Preprint AAS 11-446

<sup>4</sup> Luebke, D. and Humphreys, G., “How GPUs Work”. *IEEE Computer*, 40(2):96–100, 2007.

<sup>5</sup> Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, Space Technology Library, Microcosm Press, Hawthorne, CA, 2007.

<sup>6</sup> Gooding, R.H. "A procedure for the solution of Lambert's orbital boundary-value problem". *Celestial Mechanics and Dynamical Astronomy*, Vol. 48, No. 2, 1990, pp.145–165.

<sup>7</sup> Izzo, D., “Global optimisation and space pruning for spacecraft trajectory design”. *Spacecraft Trajectory Optimization* (Conway Ed.), Cambridge University Press, 2009.

<sup>8</sup> Oldenhuis, R., “Robust solver for Lambert’s orbital-boundary value problem”, *Mathworks Central*, URL: <http://www.mathworks.com/matlabcentral/fileexchange/26348-robust-solver-for-lamberts-orbital-boundary-value-problem> [cited 6 June 2012].