View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

Parallelization of a Six Degree of Freedom Entry Vehicle Trajectory Simulation Using OpenMP and OpenACC

Justin S. Green[*] and R. Anthony Williams[†]
NASA Langley Research Center, Hampton, Virginia, 23681

Julian Gutierrez[‡]
Northeastern University, Boston, Massachusetts, 02115

The art and science of writing parallelized software, using methods such as Open Multi-Processing (OpenMP) and Open Accelerators (OpenACC), is dominated by computer scientists. Engineers and non-computer scientists looking to apply these techniques to their project applications face a steep learning curve, especially when looking to adapt their original single threaded software to run multi-threaded on graphics processing units (GPUs). There are significant changes in mindset that must occur; such as how to manage memory, the organization of instructions, and the use of if statements (also known as branching).

The purpose of this work is twofold: 1) to demonstrate the applicability of parallelized coding methodologies, OpenMP and OpenACC, to tasks outside of the typical large scale matrix mathematics; and 2) to discuss, from an engineer's perspective, the lessons learned from parallelizing software using these computer science techniques. This work applies OpenMP, on both multi-core central processing units (CPUs) and Intel® Xeon Phi™ 7210, and OpenACC on GPUs. These parallelization techniques are used to tackle the simulation of thousands of entry vehicle trajectories through the integration of six degree of freedom (DoF) equations of motion (EoM). The forces and moments acting on the entry vehicle, and used by the EoM, are estimated using multiple models of varying levels of complexity.

Several benchmark comparisons are made on the execution of six DoF trajectory simulation: single thread Intel® Xeon® E5-2670 CPU, multi-thread CPU using OpenMP, multi-thread Xeon Phi™ 7210 using OpenMP, and multi-thread NVIDIA® Tesla® K40 GPU using OpenACC. These benchmarks are run on the Pleiades Supercomputer Cluster at the National Aeronautics and Space Administration (NASA) Ames Research Center (ARC), and a Xeon Phi™ 7210 node at NASA Langley Research Center (LaRC).

Nomenclature

E^P	=	Euler Parameters transformation matrix
$[F_x, F_y, F_z]$	=	vector of total forces acting on the entry vehicle in the body coordinate system
I^B	=	moments and products of inertia matrix of the entry vehicle about its center of mass
m	=	entry vehicle mass
$[M_x, M_y, M_z]$	=	vector of total moment acting on the entry vehicle in the body coordinate system
N_{LC}	=	number of logical cores on the hardware

[*] Doctoral Candidate, Dept. of Mechanical and Aerospace Engineering, University of Virginia, Charlottesville, VA, 22903.
Aerospace Engineer, Atmospheric Flight and Entry Systems Branch, NASA Langley Research Center, Hampton, VA, 23681.
[†] Doctoral Candidate, Dept. of Mathematics and Statistics, Old Dominion University, Norfolk, Va, 23529.
Research Computer Scientist, High Performance Computing Incubator and Atmospheric Flight and Entry Systems Branch, NASA Langley Research Center, Hampton, VA, 23681.
[‡] Pre-Doctoral Candidate, Dept. of Electrical and Computer Engineering, Northeastern University, Boston, MA, *02115*.

N_{ST} = number of concurrently supported threads per processor
N_T = number of threads
$[p, q, r]$ = components of the entry vehicle's rotation rate vector in the body coordinate system
R = Distance from planet center
T^{IB} = rotation matrix transforming a vector from body to planet-fixed coordinate system
tp = software data structure that holds vehicle specific data
$[u, v, w]$ = entry vehicle velocity vector at entry vehicle center of mass in the body coordinate system
$[x, y, z]$ = body coordinate system axes
$[X_P, Y_P, Z_P]$ = planet-fixed coordinate system axes
$[\varepsilon_0, \varepsilon_1, \varepsilon_2, \varepsilon_3]$ = quaternions specifying the body coordinate system in the planet-fixed coordinate system.
Ω = rotation rate matrix in the body coordinate system

I. Introduction

Typical applications for graphics processing units (GPUs) require repeated large matrix mathematics. These applications include, but are not limited to, computer graphics, visual data processing, computational fluid dynamics, and finite element methods. As general purpose graphics processing units become more common, other scientific computing applications are looking to take advantage of their processing power. The authors Slegers, Brown, and Rogers applied GPUs to evaluate a kinematic model of a ram air parafoil system for use by an onboard guidance system[1]. As more non-computer scientists look to apply GPUs to their research, they will be faced with a steep learning curve. Adapting single-threaded software to run on multi-threaded GPUs requires non-trivial changes in mindset, such as: how to manage memory, the organization of instructions, and the use of if statements (also known as branching).

The purposes of this work are: 1) to demonstrate the applicability of directive-based parallelization strategies, Open Multi-Processing (OpenMP) and Open Accelerators (OpenACC), to tasks outside of the typical large scale matrix mathematics; and 2) to discuss, from an engineer's perspective, the lessons learned from parallelizing software using these computer science techniques. This work applies OpenMP on multi-core central processing units (CPUs) and OpenACC on GPUs. These parallelization techniques are used to accelerate the simulation of thousands of entry vehicle trajectories through the integration of six degree of freedom (DoF) equations of motion (EoM). The forces and moments acting on the entry vehicle, and used by the EoM, are estimated using multiple models of varying levels of complexity.

The motivation for this work is to determine the applicability of OpenMP and OpenACC for use in the Program to Optimize Simulated Trajectories – II (POST2). The POST2 software, developed at the NASA Langley Research Center (LaRC), is a generalized rigid body trajectory simulation program, and has "the capability to target and optimize point mass trajectories for multiple powered or unpowered vehicles near an arbitrary rotating, oblate planet"[2]. POST2 has been utilized for many NASA missions from Shuttle, to the Mars Science Laboratory (MSL) and the Space Launch System (SLS). During mission studies and operations, POST2 is utilized for Monte Carlo simulation analyses, which typically simulate 8000 trajectories, although as many as 100,000 trajectories are run. These Monte Carlo simulations are run and rerun throughout the lifetime of a study or mission. Currently, these Monte Carlo simulations are implemented through Message Passing Interface (MPI), which sends a copy of the POST2 executable along with needed simulation data to clusters of CPU nodes. Depending on the sophistication of the trajectory, these Monte Carlo simulations can take on the order of hours to days to complete. The six DoF trajectory simulation used in this research serves as a proxy for POST2. The six DoF trajectory simulation is simpler and less capable than POST2, but it does emulate many of the same software practices and constraints.

Section II of this paper provides details of the hardware used in this research. Section III discusses the coding framework and EoM used by the software to describe entry vehicle's trajectory. Section IV presents the lessons learned for implementing OpenMP and OpenACC. Section V presents the execution time comparisons between the different implementations of the software. Section VI wraps up the research by discussing final conclusions and future work.

II. Targeted Hardware

Benchmarking of the six DoF trajectory simulation software is conducted on two compute nodes: The National Aeronautics and Space Administration (NASA) Ames Research Center (ARC) Pleiades Supercomputer Cluster and a NASA Langley Research Center (LaRC) node with a Xeon Phi™ 7210 processor (also referred to as Knights Landing or KNL). At Pleiades, the available GPU-enhanced nodes utilize two Intel® Xeon® E5-2670 (Sandy Bridge) host processors connected to an NVIDIA® Tesla® K40 GPU. The NASA LaRC node uses an Intel® Xeon Phi™ 7210 processor. Table 1 provides details of the hardware used in the benchmarking.

Table 1. Hardware specifications[3,4,5,6,7].

	NASA Ames Pleiades Supercomputer		NASA LaRC Node
Hardware	Intel® Xeon® E5-2670 (Sandy Bridge)	NVIDIA® Tesla® K40	Xeon Phi™ 7210
Label Used in Paper	CPU	GPU	KNL
Manufacturer Launch Year	2012	2013	2016
Number of Processor Cores	16 (Two 8-core)	2880	64
Number of Threads Supported Per Core	2	1	4
Processor Speed [GHz]	2.6	0.745	1.3
Total L2 Cache [MB]	40 (20 per 8 cores)	1.536	32
Memory Size [GB]	64 (32 per 8 cores)	12	128
Memory Bandwidth [GB/s]	51.2	288	102

III. Six DoF Trajectory Simulation Software

B. Software Construction

As discussed earlier, the six DoF trajectory simulation software serves as a proxy for studying the applicability of OpenMP and OpenACC for use in POST2. Like POST2, the six DoF trajectory simulation is written in C. However, it is comprised of less than 1500 lines of code, vs the hundreds of thousands to millions of lines of code in POST2. The six DoF trajectory simulation utilizes a four step Runge-Kutta integration scheme to integrate 13 EoM, which define the flight of a single rigid entry vehicle operating at Mars[8]. The software is able to simulate a number of independent trajectories in succession; the number of which is dictated by the user. The software is built to be modular, which is another feature of POST2, to allow models to be inserted and removed as needed. Representative pseudocode is provided in the Appendix to help outline the flow of the simulation software.

The six DoF trajectory software begins by reading a text file of initial states and engine throttle profiles for each trajectory to be simulated. All vehicle specific variables are saved into the data structure, *tp*. Then the software enters the loops that iterate over the number of trajectories and time integration. The Runge-Kutta integration routine calls the main trajectory function, which contains the models needed to estimate the forces and moments acting on the vehicle and the equation of motion model. The equation of motion model computes the 13 state variables used to define the vehicle's position, orientation, and velocities relative to the planet's surface (planet coordinate frame).

C. Equations of Motion

The vehicle is defined relative to a flat non-rotating planet. The kinematic EoM are split into three equations relating the vehicle's position, and four equations defining the orientation (through quaternions) of the vehicle in the planet coordinate frame. They are defined as

$$\begin{bmatrix} \dot{X}_P \\ \dot{Y}_P \\ \dot{Z}_P \end{bmatrix} = [\mathrm{T}^{IB}] \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} \dot{\varepsilon}_0 \\ \dot{\varepsilon}_1 \\ \dot{\varepsilon}_2 \\ \dot{\varepsilon}_3 \end{bmatrix} = \frac{1}{2}[E^P] \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2}$$

where the rotation matrix, T^{IB}, transforms vectors from the body coordinate system to the planet fixed coordinate system and is defined as

$$[T^{\mathrm{IB}}] = \begin{bmatrix} \varepsilon_0^2 + \varepsilon_1^2 - \varepsilon_2^2 - \varepsilon_3^2 & 2(\varepsilon_1\varepsilon_2 - \varepsilon_3\varepsilon_0) & 2(\varepsilon_1\varepsilon_3 + \varepsilon_2\varepsilon_0) \\ 2(\varepsilon_1\varepsilon_2 + \varepsilon_3\varepsilon_0) & \varepsilon_0^2 - \varepsilon_1^2 + \varepsilon_2^2 - \varepsilon_3^2 & 2(\varepsilon_2\varepsilon_3 - \varepsilon_1\varepsilon_0) \\ 2(\varepsilon_1\varepsilon_3 - \varepsilon_2\varepsilon_0) & 2(\varepsilon_2\varepsilon_3 + \varepsilon_1\varepsilon_0) & \varepsilon_0^2 - \varepsilon_1^2 - \varepsilon_2^2 + \varepsilon_3^2 \end{bmatrix} \tag{3}$$

The Euler Parameters transformation matrix, E^P, is

$$[E^P] = \begin{bmatrix} -\varepsilon_1 & -\varepsilon_2 & -\varepsilon_3 \\ \varepsilon_0 & -\varepsilon_3 & \varepsilon_2 \\ \varepsilon_3 & \varepsilon_0 & -\varepsilon_1 \\ -\varepsilon_2 & \varepsilon_1 & \varepsilon_0 \end{bmatrix} \tag{4}$$

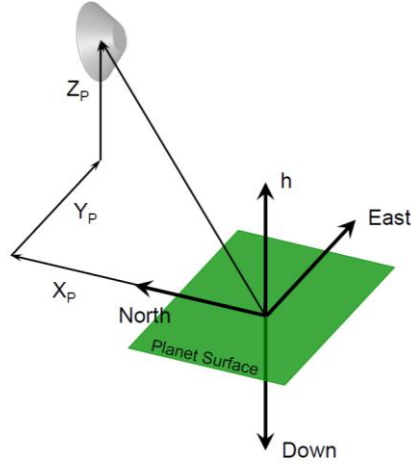

Figure 1. Position of the vehicle's center of mass in the planet coordinate frame. Note Z_P is negative as shown. Image credit: Juan R. Cruz, NASA LaRC.

Six kinetic EoM, defined by Newton's 2nd Law and Euler's equations, express the vehicle's velocity and rotational rates in the vehicle body frame. They are defined as

$$m\left(\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} + [\Omega]\begin{bmatrix} u \\ v \\ w \end{bmatrix}\right) - \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{5}$$

$$[\mathrm{I}^B]\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + [\Omega][\mathrm{I}^B]\begin{bmatrix} p \\ q \\ r \end{bmatrix} - \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{6}$$

where the rotation about the body coordinate system is

$$[\Omega] = \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix} \tag{7}$$

and the moments and products of inertia matrix, I^B, is defined as

$$[\mathrm{I}^B] = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} \tag{8}$$

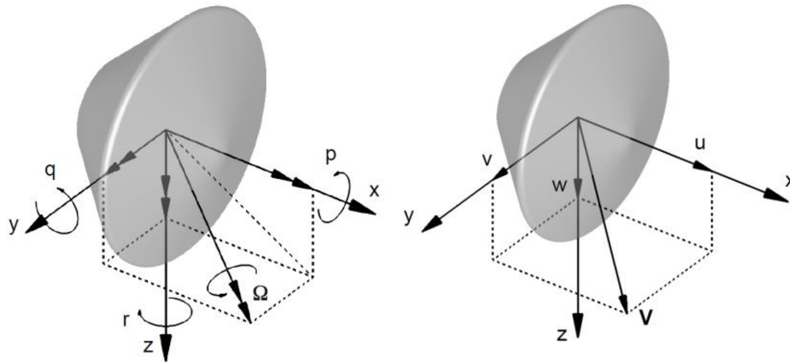

Figure 2. Definition of the vehicle's state variables in the vehicle body frame. Image credit: Juan R. Cruz, NASA LaRC.

The forces and moments feeding into the EoM are estimated through several models, which include: gravitational and propulsive (which also require an atmospheric model). The gravity model is a simple $1/R^2$ model. The Mars atmospheric model is an exponential curve fit of a Mars Global Reference Atmosphere Model (Mars-GRAM) 2010 model[9]. The propulsive model takes a time varying throttle profile for multiple forward facing descent engines to estimate the total propulsive forces and moments.

IV. Conversion from Single Threaded to Multi-Threaded

The six DoF trajectory simulation software is designed to simulate multiple independent trajectories per execution, which marks the loop covering those trajectories as *embarrassingly parallel*. Parallelizing this section of code makes each trajectory execute using a separate and unique thread. In parallelizing the software, the most notable changes made were the organization of memory, mitigation of code branching, and the inclusion of the OpenMP and OpenACC calls themselves. As the software is discussed in this section, please refer to the Appendix.

Parallelization over GPUs requires significantly more effort than CPUs and KNLs. Additionally, many of the changes that optimize the operation of the software on GPUs also benefit the CPU and KNL multi-threaded implementations.

A. Open Multi-Processing

Parallelization over CPUs and KNLs using OpenMP is relatively quick and simple to implement. With each compute core able to operate independently, they can easily handle code branching that typically occurs through if statements. In the software, implementing a *#pragma omp parallel for* directive directly above the trajectory for loop indicates to the compiler the for loop to be parallelized. After implementing this call, the next consideration is to ensure that each trajectory thread does not overwrite the data used by another compute core. The data structure *tp* contains all of the pertinent data for a given trajectory. Separating the memory for each thread was performed using the data clause *private(tp)*. These changes along with the compiler flag (*-omp* for pgcc, *–fopenmp* for gcc, or *–qopenmp* for icc) can be implemented quickly and provide a powerful improvement in computational time.

Lastly, it is important to set the number of threads to be used by the OpenMP enabled software. This setting is done by setting the environment variable *OMP_NUM_THREADS*=N_T, where N_T is the number of desired threads. Depending on the application, this number can be set to equal or more/less than the number of logical cores supported by the hardware. The number of logical cores, N_{LC}, is defined here as

$$N_{LC} = N_P N_{ST} \tag{9}$$

where N_P is the number of processors, and N_{ST} is the number of supported threads per processor. There are never more than N_{LC} cores operating concurrently, and thus only N_{LC} threads can be active at any given time. However, depending on the memory access and level of input/output needed, setting $N_T > N_{LC}$ may yield an improved execution time as shown by Bienia et al.[10,§]. Figure 3 investigates the ideal number of threads. The legend is organized as follows: Architecture – Parallelization Strategy – Compiler. For both the CPU and KNL hardware, it was best to set $N_T = N_{LC}$. For the Intel Xeon E5-2670 hardware the optimal number of threads is 32. For the Xeon Phi 7210 the optimal number of threads is 256.

[§] The maximum number of concurrent threads operating is equal to the number of logical cores. If the number of desired threads is set greater than the number of logical cores, then some threads will be paused as others are executing. This behavior may be desireable, if latencies due to memory access and input/output require a significant number of clock cycles.

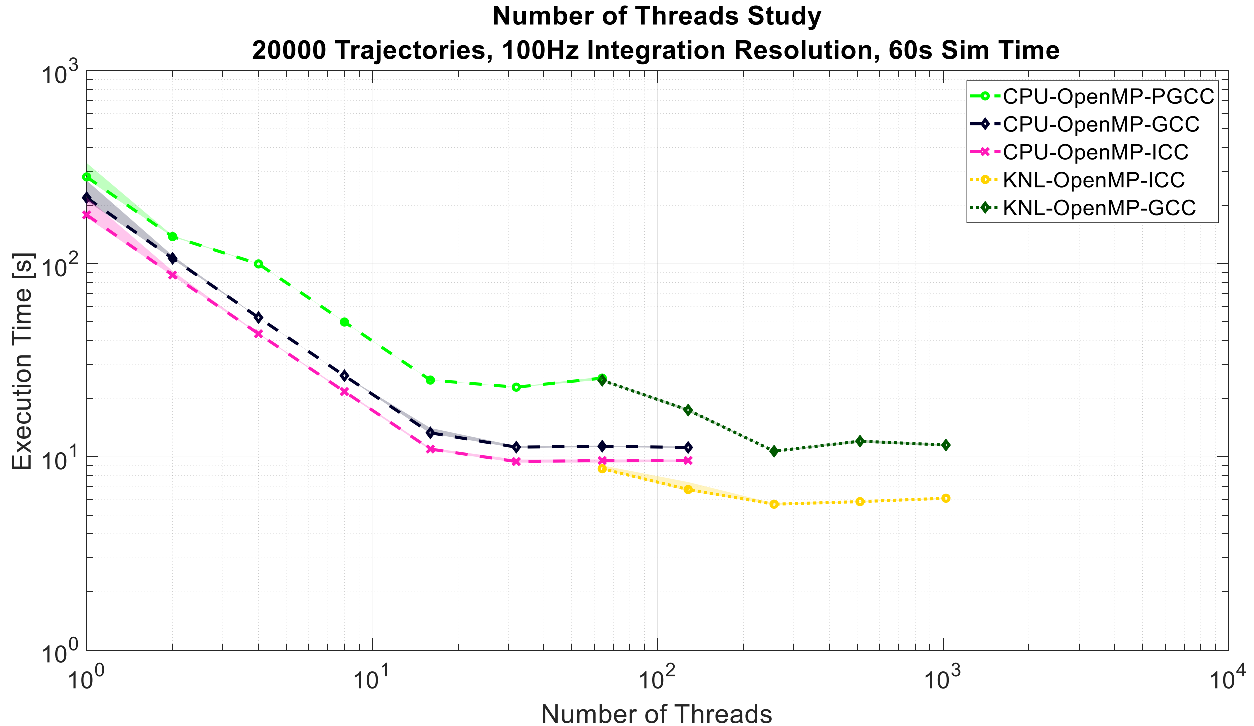

Figure 3: Investigation into the effect the number of threads has on the software execution time. Each mark represents the average obtained through 5 iterations at each testing point, and shaded regions mark the range of results.

B. Open Accelerators

As discussed earlier in this paper, implementing software onto GPUs requires significantly more effort than for CPUs. Adapting the six DoF trajectory simulation software to run on GPU hardware occurred through two main generations. The first generation focused on the inclusion of the OpenACC *#pragma* statements and the modification of the software's execution. The second generation investigated the software's usage of the GPU and efforts into improving memory organization.

1. Initial Implementation onto the Graphics Processing Unit

The implementation of the OpenACC parallelization strategy for GPU operation is similar to OpenMP in its use of *#pragma* statements. However, in OpenACC these *#pragma* statements tell the CPU when and where in the software to engage the GPU hardware. This engagement necessitates careful handling of data passing to, from, or generated on, the GPU. Much like with OpenMP, OpenACC utilizes a *#pragma acc parallel loop* directive directly above the trajectory for loop, which is parallelized using the GPU hardware. However, unlike OpenMP, OpenACC utilizes *#pragma acc data* clauses, which control the flow of data between the GPU and the CPU. Specifically, the *copyin* statement is used to pass data from the CPU to the GPU, and the *create* statement is used to create the *tp* data structure on the GPU.

Branching within a software will cause significant penalties to execution time. Running on the Pleiades nodes, the OpenACC implementation of the six DoF trajectory simulation that included branching ran approximately 10X slower than the single threaded version. Originally, the atmospheric model was controlled through three nested if statements. The trajectory simulation is focused on a powered descent vehicle, which places the vehicle lower in the Martian atmosphere. This placement allowed for the atmospheric model to be simplified and removed the branching.

2. Improving the Usage of the Graphics Processing Unit

After enabling the six DoF trajectory simulation software to run on the GPU and obtaining all the time integrated data to be returned to the CPU, the software was analyzed using profiler tools. For this work, the NVIDIA Profiler (NVPROF) and the NVIDIA Visual Profiler (NVVP) were utilized. Through these profilers, two main areas for improvement were identified using these profilers: utilization and occupancy of the GPU.

Utilization can be analyzed for the compute and/or the memory bandwidth resources, and is a percentage of the amount of resources used to the total available[11]. Through NVPROF, the six DoF trajectory simulation was found to have a compute utilization of 15%, and a memory utilization of 55%. Of the compute resources used by the simulation

software, the majority was dedicated to memory operations. This result indicates the six DoF trajectory simulation software is limited by the performance of the memory architecture in latency and bandwidth. Based on these results, an attempt was made to improve the memory usage within the software by: decreasing the amount of memory used by the software and restructuring the data to improve memory access (known as memory coalescing).

To decrease the memory burden, it was necessary to refine how the data was structured. This data restructuring involved the removal of redundant or constant variables from the *tp* data structure. Additionally, for the original single threaded version of the six DoF trajectory simulation it was convenient to keep all pertinent data packaged within two separate but similar data structures. These two structures were used for the four step Runge-Kutta integration scheme; the first was used for the outer loop integration, and the second was used for the inner loop integration. Using two data structures created a larger than needed data transfer overhead for the OpenACC version. The inner loop data structure was replaced with a small array to store the current state variables. The removal of unneeded variables and the redundant data structure saved approximately 3.608 KB per trajectory; the current size of the data structure is 3.176 KB. Further data structure improvements involved reorganizing the variables inside. Originally, variables were ordered alphabetically. Changing the order to be based on spatial data locality (variables used in the same functions are put closer together) increased the efficiency of memory access. The overhaul of the data structure improved the execution time by 5%.

To address the local memory overhead concern, significant code redesign was required. Proper memory coalescing from RAM memory is required for efficient memory bandwidth usage. The GPU exposes efficient coalescing when the data used per thread is stored contiguously in a structure of arrays (SOA) instead of an array of structures (AOS). The initial GPU implementation of the software stored data in an AOS, which was stored in local memory. Data stored in this way caused a 99.5% local memory overhead, meaning most memory operations are local memory related, which results in a high L1 cache utilization, creating congestion and possibly thrashing of data in the L1 cache[**]. Additionally, storing data inside of local memory on the GPU did not allow for the full time history of integrated data to be sent back to the CPU; only data at the conclusion of the simulation could be returned. To help reduce the local memory overhead, data was moved into global memory and into a SOA. To do this, one more dimension was added to each variable in the *tp* data structure, which is used to index the trajectories. The added dimension is in the first index of each array to be able to coalesce the reads from global memory efficiently.

The initial GPU implementation of the software called the integration loop inside the parallelized trajectory loop. Moving the time integration loop outside of the parallelized trajectory loop allows data to be copied from the GPU to the CPU at every time step using the pragma *update* clause.

After the above improvements to memory access, compute and memory bandwidth utilization is still an issue for the software. Figure 4, from NVVP, shows the GPU utilization by the six DoF trajectory simulation software, which shows memory bandwidth is still an issue. Table 2 compares the NVIDIA Tesla K40, used in this study, with newer GPU hardware. Given that the six DoF trajectory simulation software is memory bound, the increase in the memory capabilities of the new hardware should improve the compute utilization of the GPU hardware, thus improving the execution time.

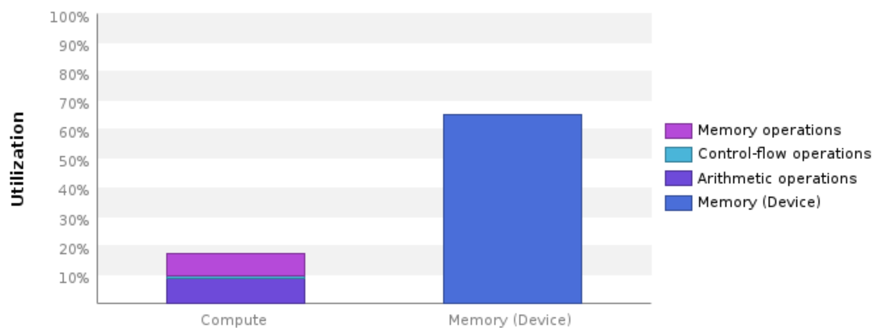

Figure 4: GPU utilization report from the NVIDIA Visual Profiler.

[**] When a level of cache memory is exhausted, the cache will evict older data to lower level cache as new data is read in. If this older data is later required, then it must be read back into the cache memory. Cache thrashing occurs when data is constantly (and possibly indefinitely) exchanged back and forth between different cache memory levels, which results in slower performance[13].

Table 2: GPU hardware comparisons[12].

	NVIDIA® Tesla® K40	NVIDIA® Tesla® P100	NVIDIA® Tesla® V100
Manufacturer Launch Year	2013	2016	2017
Memory Size [GB]	12	16	16
Memory Bandwidth [GB]	288	732	900
Number of Single Precision Cores	2880	3584	5120
Number of Double Precision Cores	960	1792	2560

Occupancy is the percentage of active warps (groups of 32 threads on the Tesla K40 architecture) to the maximum number of active warps supported by the GPU[11]. The number of active warps depends on the available resources the hardware can provide, and the amount required to run the software. One of these resources being the amount of registers used within a program. The amount of registers used by a kernel correlates directly to the local variable allocations and the complexity of the function itself. On the Tesla K40 architecture, the maximum number of registers is 65536 per streaming multiprocessor (SM) and 255 per thread[6]. These registers must be divided up amongst all threads being used by the parallelized program. Increasing the number of registers used per thread decreases the need for the threads to access L1 and L2 cache memory, which increases the speed of the program. However, this comes at the cost of decreasing the number of concurrent threads that could be run on the GPU. Therefore, a balance must be struck between the number of registers used per thread and the complexity of the program. Given the architecture of the Tesla K40 GPU and six DoF trajectory simulation software, the pgcc compiler chose a default register count of 124 per thread (per trajectory), limiting the occupancy of the GPU to 25%.

To target the occupancy issue, a balance must be made between the maximum number of registers the kernel can use with the register spills caused by each function within the software. GPUs obtain their high speedups by hiding the latency of the execution for each thread with overlapping execution. As an example, this happens when a warp is loading values from memory, the GPU suspends those threads and executes a new warp in the meantime, and hides the latency from the memory reads with execution of other threads. Increasing the number of trajectories increases the utilization. Once a thread block is assigned to a SM, all of its warps exist in the SM until they exit the kernel. Thus, a block is not launched until there are sufficient registers for all warps of the block, and until there is enough free shared memory for the block. Decreasing the number of registers used per thread, increases the occupancy of the GPU (number of warps that can be active in an SM). However, given the complexity of the software, this will cause an increase in register spills to local memory, which can hinder performance. Determining this value is key to striking a compromise between local memory usage due to spills and increasing the occupancy by reducing the number of registers per thread. Figure 5 shows the performance achieved from running 10 experiments per register value and averaging those results. Limiting the maximum register usage to 80 resulted in the best performance.

Table 3 categorizes the modifications made to the six DoF trajectory simulation software and their corresponding improvements. The memory modifications made for the GPU/OpenACC implementation, to improve its utilization and occupancy, also provided benefits to the CPU/OpenMP implementations. A final improvement for both the OpenACC and OpenMP implementation was to include function inlining at compile time. Function inlining reduces the function call overhead on the GPU by replacing the function calls with the lines of code of the function itself within the main program. Function inlining provided a significant decrease in the GPU implementation’s execution time, while providing a modest improvement for the CPU implementations.

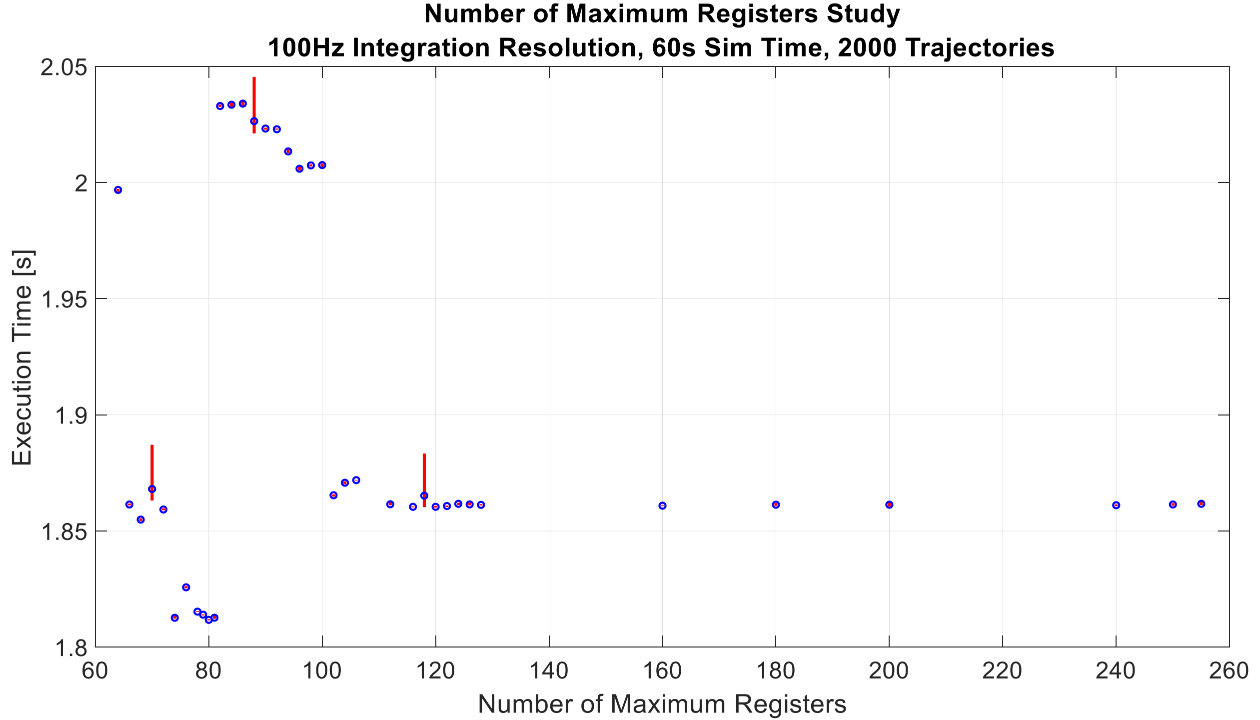

Figure 5: Investigations into the maximum register specification made at compile time. Each mark represents the average obtained through 10 iterations at each testing point, and the red bars indicate the range of results.

Table 3: Six DoF trajectory simulation execution time for running 20000 trajectories. Each trajectory is integrated at 100 Hz, and simulates a 60 s long trajectory.

Hardware/ Parallelization Scheme/Number of Logical Cores	Compiler	Adjustments Made To Software	Max Execution Time [s]	Min Execution Time [s]	Mean Execution Time [s]
GPU/OpenACC/ 2880	PGCC	Original Implementation	17.925889	17.91272	17.9195
		After Memory Reorganization & for Loop Switch	14.22608	14.21557	14.2202
		Including Function Inlining	8.96356	8.946091	8.95077
CPU/OpenMP/32	GCC	Original Implementation	18.567092	18.3187	18.3867
		After Memory Reorganization	17.945576	17.5486	17.7444
		Including Function Inlining	11.863747	11.37236	11.6888
	ICC	Original Implementation	15.663124	15.16546	15.3323
		After Memory Reorganization	15.007072	14.63887	14.8136
		Including Function Inlining	9.520547	9.172192	9.29136
	PGCC	Original Implementation	25.413	24.04681	24.4751
		After Memory Reorganization	23.219827	22.94037	23.0076
		Including Function Inlining	15.419285	14.86854	15.0825

V. Comparisons

Multi-threaded comparisons are made across several computational architecture, and across three compilers: pgcc (ver. 17.1-0), developed by The Portland Group, Inc.; gcc (ver. 6.2.0), the GNU compiler; and icc (ver. 18.0.0), developed by Intel®. All comparisons execute the same lines of code with minor changes, which included: the

differences in selected hardware, the three compilers, and the compiler flags that activate OpenMP vs OpenACC. The largest difference between the OpenMP and OpenACC implementation is the order of execution of the trajectory and time loops, which was discussed in Section IV.B. Although the order of the loops changed, the core functionality of the software remains the same. Table 4 lists the compiler flags used for each compiler and hardware configuration. At the time of publication, the gcc compiler on the Pleiades Supercomputer did not support OpenACC, which limits the OpenACC/GPU study to just the pgcc compiler. Additionally, the version of pgcc on Pleiades was not applied to the KNL hardware, because it did not have KNL specific hardware targeting and was limited to a maximum of 64 threads for OpenMP.

Table 4. Compiler flags used in comparison analysis.

		pgcc	gcc	icc
Optimization	Enable	-fast -Minline -Mipa=fast,inline	-Ofast -flto -ffat-lto-objects	-fast -ffat-lto-objects
OpenMP	Enable	-mp	-fopenmp	-qopenmp
	CPU Hardware Targeting	-	-march=native	-xhost
	KNL Hardware Targeting	-	-march=knl	-xmic-avx512
OpenACC	Enable	-acc	-	-
	GPU Hardware Targeting	-ta=tesla:fastmath, cc35, maxregcount:80	-	-

Figure 6 and Figure 7 provide comparisons across scaling the trajectory simulation. All comparisons simulate 60 seconds of a powered decent vehicle trajectory as it decelerates to land on the Martian surface. In both figures, the legends are organized as follows: Number of Logical Cores – Architecture – Parallelization Strategy – Compiler. For the GPU hardware execution, results showing variable definitions as all doubles (double precision) or all floats (single precision) is also noted.

All concerns regarding memory accesses, occupancy, and local memory usage of the simulation software are factors that can be mitigated by changing the data type from doubles to floats. The main reasons for this are: access to more single-precision units per SM compared to double-precision (see Table 2); register usage decreases (increasing occupancy to almost 40%); global memory transactions reduce by half, thus increasing the memory throughput; and less register spills, which reduces local memory requests. For all studied trajectories, the accuracy difference between doubles and floats was $<$ 3.0%, which was a tolerable difference for this application. Additionally, the CPU and KNL hardware implementations did not show significant performance gains when using all float data types versus all doubles.

Figure 6 compares the scaling across the number of simulated trajectories. The linear relationship observed in the CPU and KNL results is due to the trajectory simulation problem being embarrassingly parallel. The GPU performance shows an approximate 1.3-1.8X speed of execution improvement of the all float implementation over the all double implementation. The GPU lines remain flat until the 4000-8000 simulated trajectories range, which is due to the low number of simulated trajectories not utilizing the full 2880 cores available on the GPU. It is also why the CPU and KNL results outperform the GPUs in this range. For the higher range of trajectories investigated (> 8000), it's notable that the GPU performance is not significantly improved over the OpenMP enabled CPU implementation using the icc compiler, and is similar to the KNL implementations. Two factors play into this result. One, the KNL hardware is three years newer than the GPU hardware. Two, the nature of the trajectory simulation problem itself is not well suited for GPUs. Although branching (due to if statements) has been mitigated, it is not possible to fully remove them. Also, the scale of trajectories investigated ended at 20000, which does not fully leverage the capability of the GPU (typically scale to the millions and larger). Typical POST2 Monte Carlo simulations of vehicle trajectories are in the range of 8000 trajectories.

Trends across the different compilers are noticeable in Figure 6 as well. In looking at the OpenMP results, it is not surprising that the icc compiler out performs the gcc and pgcc compilers, since Intel hardware is used. It is notable that the pgcc compiler used 1.35-1.65X more execution time than the icc and gcc compilers. Lastly, the gcc compiled version executing on 32 threads CPU versus the 256 threads on KNL achieved similar execution times, even though the KNL uses 8 times the number of threads.

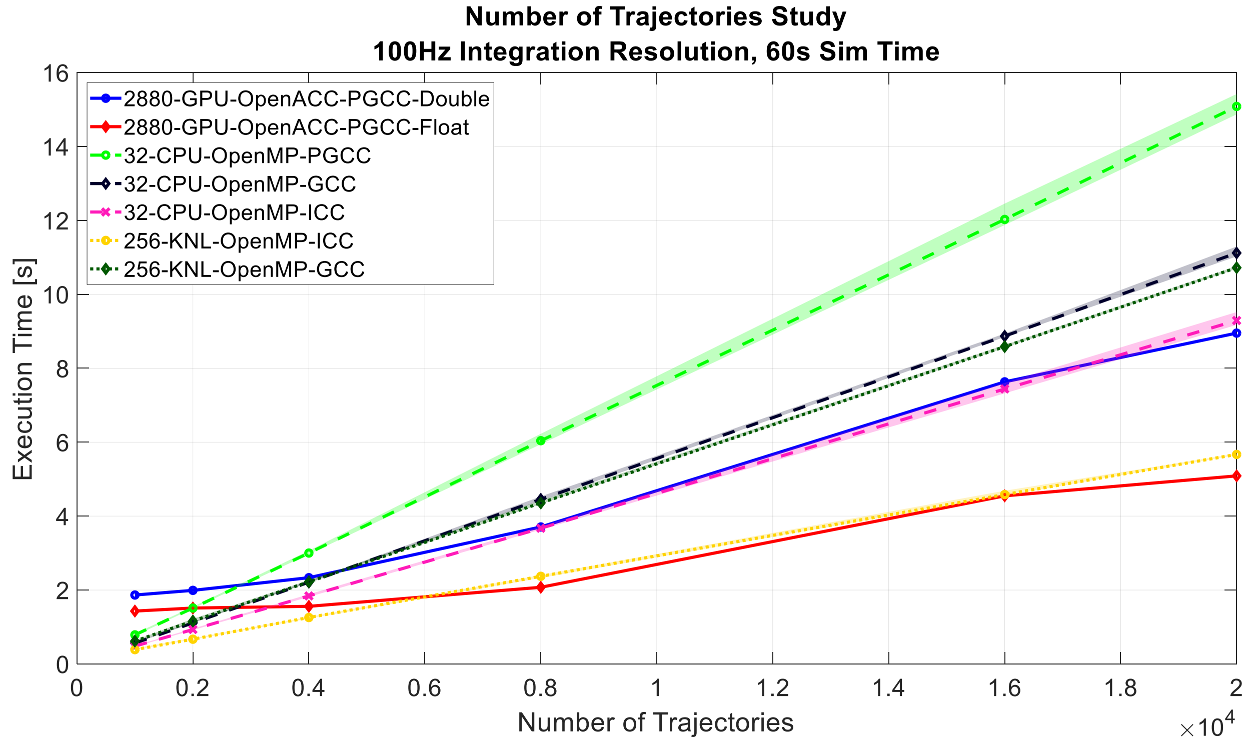

Figure 6: Comparisons of compilers and hardware with scaling the number of trajectories. Each mark represents the average obtained through 10 iterations at each testing point, and shaded regions mark the range of results.

Figure 7 compares the integration frequency, which directly relates to the number of integration time steps taken. When simulating 2000 trajectories, the all double implementation on the GPU requires the most time to execute and the all float version has a similar execution time as the OpenMP with the pgcc compiler. As in Figure 6, this demonstrates how the low number of simulated trajectories doesn't fully utilize the capabilities of the GPU. When the number of simulated trajectories is higher, such as shown in the bottom of Figure 7 with 20000 trajectories, the GPUs execution times are more favorable when compared to the OpenMP enabled CPU implementations. In looking at the 20000 simulated trajectories, the OpenACC enabled GPU implementation using all float variable definitions outperforms the OpenMP enabled KNL implementation with the icc compiler by a range of 0.6 – 47.9 s. However, the KNL version retains the double precision accuracy.

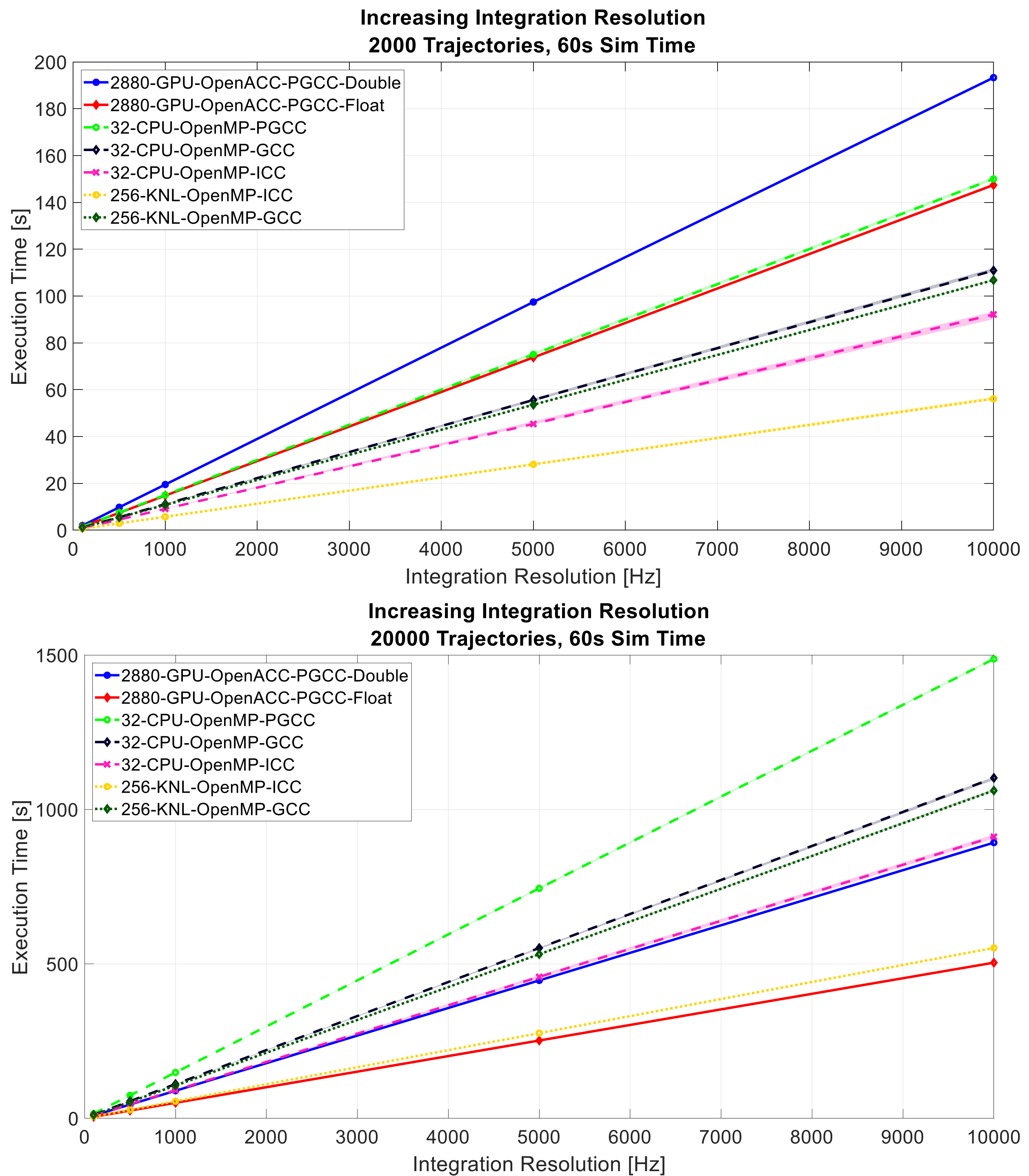

Figure 7: Comparisons of compilers and hardware with scaling the integration frequency. The top plot simulates 2000 trajectories, and the bottom plot simulates 20000 trajectories. Each mark represents the average obtained through 10 iterations at each testing point, and shaded regions mark the range of results. The shaded region are small compared to the scale on the y-axis.

VI. Conclusions and Future Work

The objective of this paper was to apply the OpenMP and OpenACC strategies to a six DoF trajectory simulation problem and enable it to run in parallel on CPU, KNL, and GPU hardware. It was also to provide a lessons learned for parallelizing the software, from an engineer's perspective. Three compilers were investigated in this work in an effort to broadly study the effects of parallelizing the software. Four conclusions are drawn from this research: 1) The six DoF trajectory simulation software studied is memory bound, which limits the amount of parallelism it can have on the GPU hardware. 2) The implementation of the OpenMP and OpenACC *#pragma* statements to parallelize software is straight forward, and requires a low level of effort by the programmer. 3) Getting the GPU implemented software to run well, however, requires a significant effort in managing memory and is non-trivial. 4) The OpenMP strategy on KNL hardware provides a significant execution time speed up with minimal effort.

Results from this research found that the GPU hardware typically underperformed compared to the KNL hardware. It should be noted that the GPU hardware used in this study is approximately three years older than the KNL hardware, which was due to the resources available at the time of writing. Future work will investigate performance of the six DoF simulation software on the NVIDIA Tesla P100 and NVIDIA Tesla V100 hardware. The increased memory capabilities of these hardware will play a large roll in decreasing the memory bounded nature of the software, thus increasing its speed of execution. Other areas of research to explore are implementing other parallelization strategies, such as Compute Unified Device Architecture (CUDA), optimizing the KNL implementation using its vectorization capabilities, simulating a larger number of trajectories, and increasing the simulation complexity by adding additional models, such as aerodynamics.

Appendix

Provided below is a pseudocode example of the six DoF trajectory simulation software.

Main Function - CPU

```
main{
  struct tp // integration loop data structure

  // read input data from file

  // Start Timer

  // Loop over the number of trajectories
  #pragma omp for private(tp) nowait
  for(number of trajectories){

    // Initialize data structures

    // Integrate trajectories
    for(number of integration steps){
      runge_kutta(tp); // 4 step Runge-Kutta routine

      // Save current time step to datalog structure
    }
  }
  // End Timer

  // Store output to files
}
```

Main Function - GPU

```
main{
  struct tp // integration loop data structure

  // read input data from file
```

```
  // Start Timer

  // Loop over the number of trajectories
  #pragma acc data create(tp)
  {
    #pragma acc parallel loop independent gang vector
    for(number of trajectories){

      // Initialize data structures

    }

    // Integrate trajectories
    #pragma acc loop seq
    for(number of integration steps){

      // Loop over the number of trajectories
      #pragma acc parallel loop independent gang vector
      for(number of trajectories){

        runge_kutta(tp); // 4 step Runge-Kutta routine
      }

      // Save current time step to datalog structure
      #pragma acc update host(tp)

    }
  }
  // End Timer

  // Store output to files
}
```

Runge-Kutta Function

```
runge_kutta(tp){
  // Save off state
  ysave = tp->y;

  // Obtain outer loop rates
  tp->dydt = trajectory_funct(time,tp->y);

  // Perform inner loop integration
    // Compute first step
    k1 = dt*tp->dydt;
    tp->y = ysave + k1/2;

    // Compute second step
    tp->dydt = trajectory_funct(time+dt/2,tp->y);
    k2 = dt*tp->dydt;
    tp->y = ysave + k2/2;
```

```
      // Compute third step
      tp->dydt = trajectory_funct(time+dt/2,tp->y);
      k3 = dt*tp->dydt;
      tp->y = ysave + k3;

      // Compute fourth step
      tp->dydt = trajectory_funct(time,tp->y);
      k4 = dt*tp->dydt;

   // Perform outer loop integration
   tp->y = ysave + k1/6 + k2/3 + k3/3 + k4/6;
}
```

Trajectory Simulation Function

```
trajectory_funct(time,temp){
   // Gravity model
   grav_model(temp->altitude,temp->force_grav);

   // Atmospheric model
   atmo_model(temp->altitude,temp->pres);

   // Throttle and propulsion models
   prop_model();

   // Sum forces and moments

   // Solve EoM to obtain rates
   eom_solver(temp);
}
```

Acknowledgments

The authors would like to thank Dr. Robert Lindberg, University of Virginia; Dr. James Hoffman, Analytic Mechanics Associates; and Dr. Scott Striepe, NASA LaRC, for their guidance and support. We would also like to thank Dr. Juan R. Cruz, NASA LaRC, for his white paper on entry vehicle equations of motion. The authors would like to thank the NASA LaRC High Performance Computing Incubator (HPCI) for the resources and funding it provided.

Resources supporting this work were provided by the NASA High-End Computing (HEC) Program through the NASA Advanced Supercomputing (NAS) Division at Ames Research Center. The 2017 ORNL Hackathon at NASA was a collaboration between and used resources of both the National Aeronautics and Space Administration and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory. Oak Ridge National Laboratory is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Slegers, N., Brown, A., Rogers, J., "Experimental investigation of stochastic parafoil guidance using a graphics processing unit," Control Engineering Practice, Vol. 36, Mar. 2015, pp.27-38.

[2] Striepe, S. A., Powell, R. W., Desai, P. N., Queen, E. M., Way, D. W., Prince, J. L., Cianciolo, A. M., Davis, J. L., Litton, D. K., Maddock, R. M., Shidner, J. D., Winski, R. G., O'Keefe, S. A., Bowes, A. G., Aguirre, J. T., Garrison, C. A., Hoffman, J. A., Olds, A. D., Dutta, S., Zumwalt, C. H., White, J. P., Brauer, G. L., Marsh, S. M., Lugo, R. A., Green, J. S., "Program To Optimize Simulated Trajectories II (POST2): Utilization Manual," Vol. 2, Ver. 4.0.0.r1173, July 2017.

[3] Dunbar, J., "Pleiades Supercomputer," High-End Computing Capbility [online], https://www.nas.nasa.gov/hecc/resources/pleiades.html [retrieved 5 October 2017].

[4] "Intel® Xeon Phi™ Processor 7210," *Intel* [online], https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors/7210.html [retrieved 5 October 2017].

[5] "Intel® Xeon® Processor E5-2670," *Intel* [online], https://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI [retrieved 18 October 2017].

[6] "Tesla K40 GPU Accelerator: Board Specification," NVIDIA, BD-06902-001_v05, Nov. 2013.

[7] "NVIDIA's Next Generation CUDA™ Computer Architecture: Kepler™ GK110," NVIDIA, Whitepaper V1.0, 2012.

[8] Press, W. H., Teukolsky, S. A., Vellerling, W. T., Flannery, B. P., Numerical Recipies in C The Art of Scientific Computing, 2nd ed., Cambridge University Press, New Deli, 1992, Chap. 16.

[9] Justh, H. L., "Mars Global Reference Atmospheric Model 2010 Version: Users Guide," NASA/TM-2014-217499, 2014.

[10] Bienia, C., Kumar, S., Jaswinder, P. S., Kai, L., "The PARSEC Benchmark Suite: Characterization and Architectural Implicatiolns," 2008 *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, Toronto, ON, Candada, 2008, pp. 72-81.

[11] "Profiler User's Guide," NVIDIA, DU-05982-001_v9.1, Mar. 2018.

[12] "NVIDIA Tesla V100 GPU Architecture," NVIDIA, WP-08608-001_v1.1, Aug. 2017.

[13] Seshardi, V., Mutlu, O., Kozuch, M. A., Mowry, T. C., "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," *21st International Conference on Parallel Architectures and Compilation Techniques (PACT),* IEEE, Minneapolis, 2012, pp. 355-366.