AIAA Guidance, Navigation, and Control Conference
08 - 11 August 2011, Portland, Oregon

AIAA 2011-6660

High-Speed Prediction of Air Traffic for Real-Time Decision Support

Monish D. Tandale*, Sandy Wiraatmadja† and P. K. Menon‡
Optimal Synthesis Inc., Los Altos, CA, 94022-2777

and

Joseph L. Rios§
NASA Ames Research Center, Moffett Field, CA, 94035-1000

The ability to rapidly generate traffic predictions is expected to be central for implementing next-generation air traffic management functionality, both on the ground and aboard aircraft. While high-end computers can be used for this purpose, emerging capabilities of computational hardware such as Graphics Processing Units, together with Cloud Computing concepts can be exploited to realize substantial acceleration of trajectory computations at a modest cost increment. This paper discusses the development of a computational appliance for rapid prediction of aircraft trajectories that combines efficient algorithm and software design with emerging high performance computing architectures. The research effort accelerates trajectory predictions through software profiling and tuning, and implements computationally intensive functions on high performance computing architectures such as computing clusters, multi-threaded programming on multi-core computers and Graphics Processing Units. The fastest of these implementations uses a Graphics Processing Unit, which can perform a system-wide 24-hour trajectory prediction for 35,000 aircraft in less than 2.5 seconds. When compared with the baseline trajectory prediction software, the present approach provides over two orders of magnitude speedup.

Nomenclature

ATM	Air Traffic Management
ASDI	Aircraft Situation Display to Industry
ACES	Airspace Concepts Evaluation System
API	Application Programming Interface
BADA	Base of Aircraft DAta
CARPAT	Computational Appliance for Rapid Prediction of Aircraft Trajectories
CARAT#	Configurable Airspace Research and Analysis Tool – Scriptable
CD&R	Conflict Detection & Resolution
CUBLAS	CUDA Basic Linear Algebra Subprograms

* Research Scientist, 95 First Street, monish@optisyn.com, Senior Member AIAA
† Research Engineer, 95 First Street, sandy@optisyn.com
‡ President & Chief Scientist, 95 First Street, menon@optisyn.com, Fellow AIAA
§ Aerospace Engineer, Systems Modeling and Optimization Branch, Mail Stop 210-15, Joseph.L.Rios@nasa.gov, Member AIAA

CUDA	Compute Unified Device Architecture
FDS	Flight Data Set
FACET	Future ATM Concepts Evaluation Tool
GPU	Graphics Processing Unit
HPC	High Performance Computing
ITWS	Integrated Terminal Weather System
JPDO	Joint Planning and Development Office
MPI	Message Passing Interface
NAS	National Airspace System
NCEP	National Centers for Environmental Prediction
NOAA	National Oceanic and Atmospheric Administration
NextGen	Next Generation Air Transportation System
OpenGL	Open Graphics Library
PAR	Preferred Arrival Route
RAM	Random Access Memory
RUC	Rapid Update Cycle
RMI	Remote Method Invocation
SID	Standard Instrument Departure
STAR	Standard Terminal Approach Route
SMs	Streaming Multiprocessors
TBO	Trajectory Based Operations
TFM	Traffic Flow Management
TRACON	Terminal Radar Approach Control

I. Introduction

An important element of the Next Generation Air Transportation System (NextGen) concept[1] being developed by NASA in partnership with the Joint Planning and Development Office (JPDO), is the Trajectory-Based Operations (TBO) concept. This new concept will dramatically change the manner in which traffic is managed in the national airspace, leading to significant increases in airspace capacity and efficiency. Present air traffic management methodology is based on a fixed airspace structure tied to geographic locations within the NAS, and can be termed as Fixed Airspace Operations. The Trajectory-Based Operations is a paradigm shift from the current approach and uses four-dimensional (4-D) trajectories as the basis for managing the air traffic management (ATM) system. In Trajectory-based operations, all ATM decisions across all time horizons are fundamentally related to 4-D trajectories[1]. These 4-D trajectories are the principal language for information exchange, planning, and analysis, enabling greater use of digital communication and ground-based and airborne automation, and facilitating coordination and collaboration between aircraft operators and air traffic management entities.

Since aircraft trajectory prediction will play a central role in the NextGen, it is important to be able to rapidly generate these predictions, either on the ground or onboard aircraft. While high-end computers can be used for this purpose, cluster computing architectures and emerging high performance computing hardware such as Graphics Processing Units (GPUs)[2] can be exploited to realize substantial acceleration of trajectory computations at a modest cost increment. GPU implementations of computationally demanding algorithms have demonstrated acceleration factors of up to $300\times$ in diverse application areas such as Computational Fluid Dynamics (CFD), computer vision, medical imaging, oil and gas exploration and mathematical finance[3].

The Future ATM Concepts Evaluation Tool (FACET)[4] developed by NASA, equips researchers and service providers with a way to explore, develop and evaluate advanced air transportation concepts before they are field-tested and eventually deployed. FACET is able to simulate a full day's dynamic National Airspace System (NAS) operations by quickly generating and analyzing thousands of aircraft trajectories, using flight schedules, aircraft performance profiles, airspace models, and weather data.

The research presented here investigates the development of a Computational Appliance for Rapid Prediction of Aircraft Trajectories (CARPAT™)[5] that combines the trajectory and airspace modeling features of NASA's FACET software with cluster computing architectures[6] and the emerging computational power of GPUs. Additionally, a recently-developed client-server technology[7-9] that allows access to the FACET functionality over the Internet Protocol (IP) is employed as the interface of the CARPAT system with the users.

The remainder of the paper is organized as follows. An overview of the concept for trajectory prediction accelerations is provided in Section II. Section III discusses the profiling and tuning of the FACET code that was performed to converge on efficient algorithms and subsequent lean code implementation. The cluster implementation of FACET is presented in Section IV and the multi-threaded FACET implementation to exploit multi-core and multi-processor computers is presented in Section V. Section VI describes the attempt at using the GPU as a co-processor for computationally intensive FACET functions and Section VII describes the most successful and promising implementation in which the GPU is used as the primary processor for executing all trajectory propagation functions. Finally, the research summary and conclusions are presented in Section VIII.

II. Conceptual Overview

A conceptual diagram illustrating the operational concept of the computational appliance is given in Figure 1.

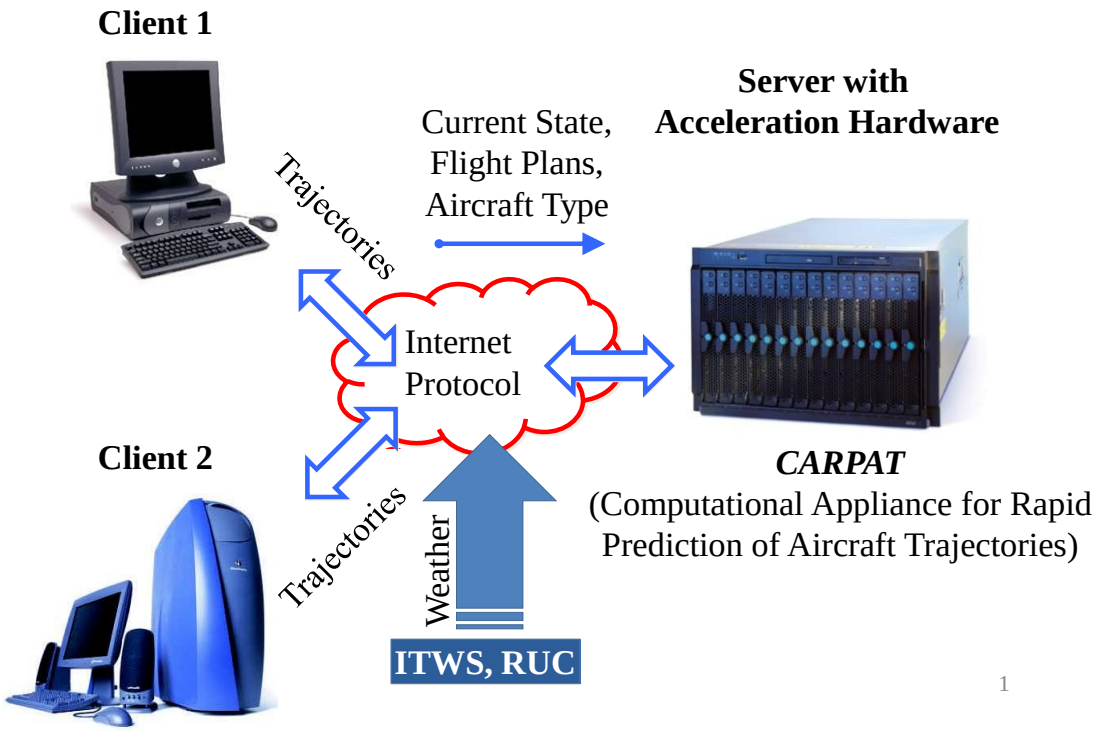


Figure 1. Operational Concept for the Computational Appliance

The computational appliance is configured around one or more servers running the trajectory prediction software, with multiple GPUs providing the acceleration of repeated computations. The server

incorporates fast network hardware to allow rapid access over Internet Protocol. Air traffic management applications running on client machines can send-out the flight plans, aircraft type, and the current states to the computational appliance. Trajectory predictions will be generated by the computational appliance and sent out to the clients over broadband network connection. The computational appliance is configured to periodically download weather data from sources such as Integrated Terminal Weather System (ITWS)[10] or Rapid Update Cycle (RUC)[11] in an automatic manner to ensure that the trajectory predictions take into account the ambient wind fields, thereby increasing the fidelity of the predictions.

Note that the trajectory prediction discussed in this paper is primarily intended for use in demand prediction for solving the national level Traffic Flow Management (TFM) problem. Hence this formulation uses kinematic motion models, together with the aircraft performance data to model the motion of aircraft from one way point to the next, with a propagation time step of 30 seconds. Note that the fidelity of the trajectory predictions obtained with a time step of 30 seconds is sufficient for solving the nation-wide TFM problem. Although not discussed in this paper, a time-scale separation approach can be used to include higher-order dynamics in the trajectory prediction methodology to enable the computation of turn rate and altitude rate, and to enforce load factor or bank angle limits. The trajectories obtained by propagating these higher-order dynamics can be used as the basis for tactical conflict detection and resolution (CD&R) algorithms.

III. Software Profiling & Tuning

Dynamic performance analysis[12] investigates the behavior of a software using information gathered as it executes. The goal of performance analysis is to identify the parts of a program that can be optimized for speed or memory usage. According to Amdahl's law on software optimization[12], the performance can at most be increased in proportion to the number of CPU cycles being used by the part of the code that is being optimized. This allows for identification of those functions and modules that consume a significant fraction of the total CPU execution time, which may need to be optimized.

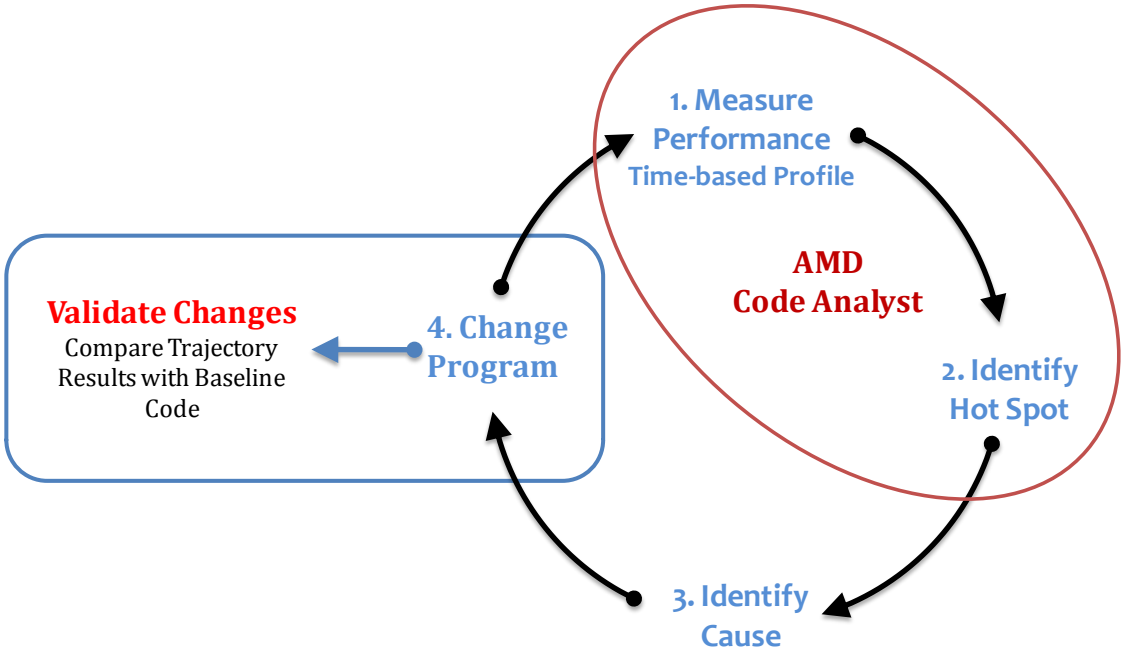


Figure 2. FACET Performance Tuning Cycle

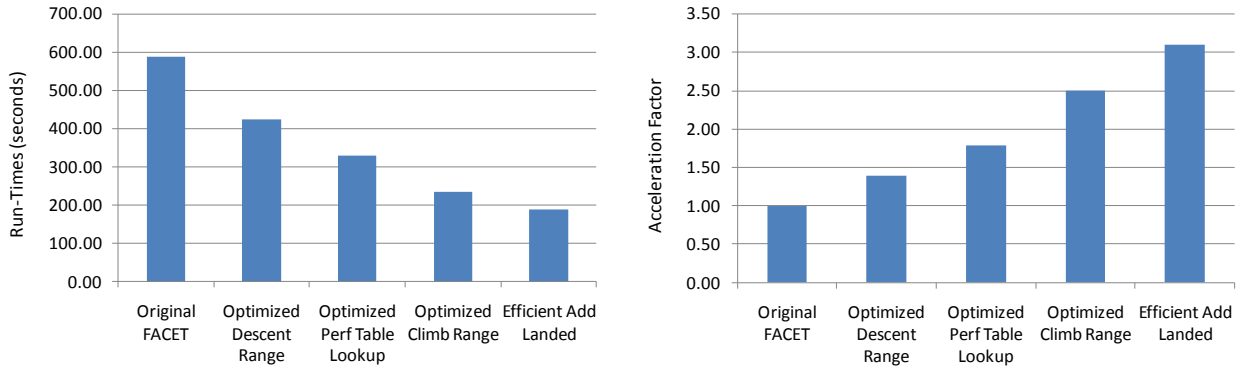
Figure 2 illustrates the performance tuning cycle employed during the course of this research effort. The first step involves running the FACET software for a sample traffic simulation scenario to measure its run-time performance. This is followed by identification of the hot spots that are defined as portions of

the code that consume significant amount of run-time. AMD Code Analyst[13] aids in the first two steps. The third step involves an analysis of the hot spots in the source code to identify potential improvements that can shorten the run-time. The fourth step involves the modification of the program to implement improved code for accelerated performance. These steps can be repeated until the point of diminishing returns is reached.

FACET was profiled using AMD Code Analyst[13] and the hot spots in the code were analyzed to improve run-time performance. The following modifications were made to the FACET code.

1. *Optimized Descent Range*: The descent distance is the downrange the aircraft will travel while descending from the current altitude, and is only a function of the current altitude, given the aircraft type. In the original implementation of FACET, the descent distance was being calculated at every time step even if the altitude remained the same from the previous time step to the current time step. This wasteful computation was eliminated by a modified implementation in which the descent distance is calculated only if the altitude decreased from the previous time step to the current time step, leading to reduced run time.
2. *Optimized Performance Table Lookup*: FACET uses the BADA database to obtain the aircraft performance parameters that are stored in tables as function of the aircraft altitude. In the original FACET implementation, the search for the table index always began from zero and the loop iterated until the current flight level was found. The search was modified to begin from the table index at the previous time step to take advantage of the continuity in aircraft altitude from one time step to the next.
3. *Optimized Climb Range*: The climb distance calculation was also modified in a manner similar to the descent distance calculation.
4. *Efficient implementation of the Add Landed Linked List*: FACET maintains a linked list of all aircraft that landed in the simulation. In the original FACET implementation, a new element was being added to the tail of the linked list that required a complete iteration over the entire list to find the tail. The modified implementation added the new element to the head of the list, thus decreasing the run time from 47.34 seconds to 0.01 seconds for this code segment.

Figure 3 illustrate the results of software profiling and tuning of FACET code. These results show that software profiling and tuning resulted in speedup of 3.1 × over baseline FACET code.



a.) Run-Times for a 24 Hour FACET Run **b.) Acceleration Factor Over Original FACET Code**

Figure 3. Results of FACET Profiling and Software Tuning

IV. Cluster Implementation

Cluster computing techniques can provide an acceleration of trajectory computations by distributing the computational load between multiple processors connected over a high-speed network. The software must be re-written to take advantage of the cluster, and specifically have multiple non-dependent parallel computations involved in its execution. Trajectory propagation of aircraft is an inherently parallel task.

Hence propagation of '*n*' aircraft trajectories on '*m*' cluster nodes can be parallelized by distributing '*n/m*' aircraft on each node.

Compute clusters are commonly programmed using Message Passing Interface (MPI[14]). The most widely used implementations of the MPI are available as C libraries. However, FACET is a mixed C-Java application with a C computational core and a Java interface, which is not amenable to MPI implementation. Hence, an alternate cluster architecture is proposed which uses Java Remote Method Invocation (RMI) instead of MPI. Java Remote Method Invocation[15] enables the programmer to create distributed Java-based applications, in which the methods of remote Java objects can be invoked from other Java Virtual Machines (JVMs) possibly on different hosts. A software package called CARAT# (Configurable Airspace Research and Analysis Tool – Scriptable) was developed under a recent research effort[7,8] that uses RMI to allow remote clients to access FACET functions running on different host computers.

Figure 4 illustrates the cluster computing architecture using CARAT#.

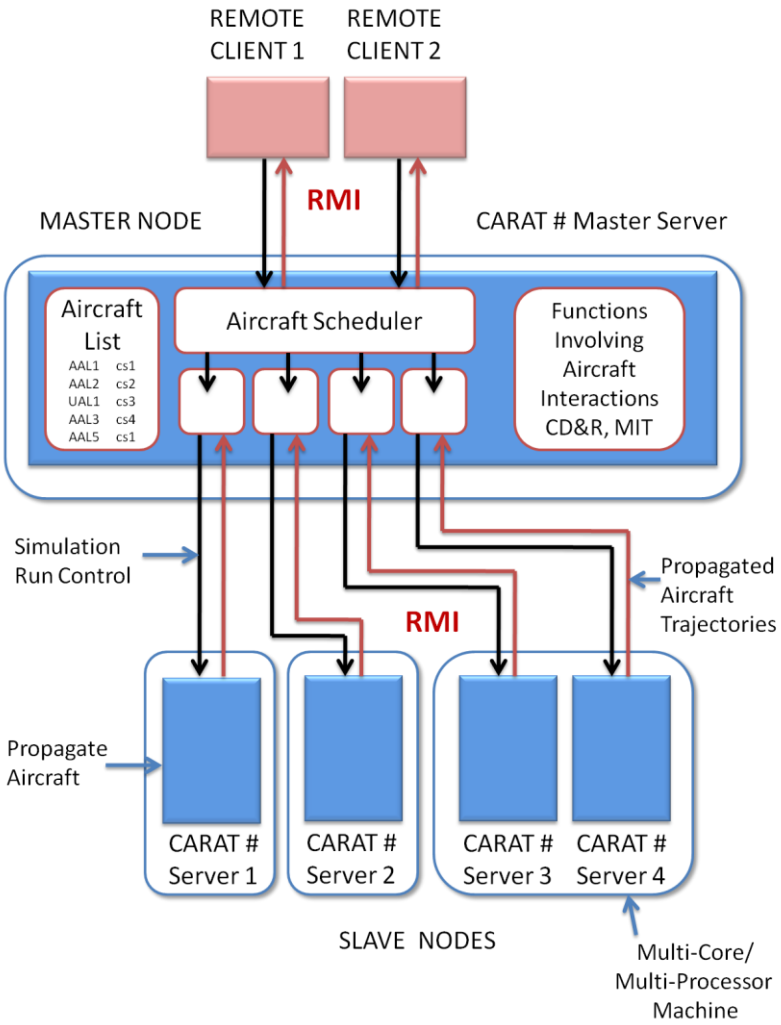


Figure 4. FACET Cluster Implementation using Java Remote Method Invocation (RMI)

Remote clients can invoke functions such as starting a simulation, querying the states of the aircraft, setting the states of an aircraft, and other functions on the master node though RMI. The master node has an aircraft scheduler that dynamically assigns aircraft to each slave node for propagation. The master node also maintains a list of all aircraft and the corresponding slave node that they are assigned to. So

when the remote user queries data for a particular aircraft, the master node just invokes the method on the appropriate slave node and returns the requested data to the user. Since the aircraft are divided and propagated on distributed memory processors, aircraft assigned to a slave node are essentially propagated independent of aircraft assigned to any other node. So any functions that involve interactions between trajectories, such as conflict detection and resolution, can be run on the master node.

The least observed run time for a 24 hour FACET simulation is 110 seconds with a cluster configuration of 3 slave nodes each running 4 independent CARAT# servers. Note that out of the 110 seconds, 32 seconds account for the data transfer time between the master and slave nodes. Thus, the experimental results show that the data transfer times are significant components of the total runtime. This is because some of the machines in the cluster have 100Mbps network cards which slow down the data transfer. With the emerging 10 Gigabit Ethernet standards, the data transfer times are expected to decrease significantly.

The total speedup that was achieved by the cluster implementation over baseline FACET, (including the effects due to software tuning) is $5.34\times$.

V. Multi-Threaded Implementation

Currently available multi-core systems run different threads and processes simultaneously on different cores. A system with N cores is optimally effective when it is presented with N or more concurrent threads. A multi-threaded FACET program can take advantage of the dual cores or quad cores that may be present on the host processor. The key to this implementation is to find functions in FACET that are independent of each other and can be implemented in parallel. During the propagation for a single time-step, the propagation of an aircraft is independent of other aircraft in the airspace, and hence can be executed concurrently in different threads. Any interaction between aircraft can be coded in a separate function and executed after the state propagation of all aircraft at a given time step is complete.

In the original FACET implementation, there exists a loop that runs over the linked list for all aircraft and propagates each aircraft by one time step. The modified implementation divided the aircraft list into N sub-lists for a N-core processor and the propagation loop was executed over each sub-list concurrently in separate threads, which ran on separate cores. The multi-threading was implemented using the `pthread` library[16, 17].

The execution times for a 24 hour simulation using the original single threaded FACET implementation are as follows:

1. Total Execution Time (T): 456 seconds
2. Propagation loop over all aircraft in the aircraft linked list (P): 277 seconds. This is the parallelizable component of the total code.

Thus the time (S) taken for serial code execution is T-P = 456 - 277 = 178 seconds. Note that these execution times were recorded during at an intermediate stage of software profiling and tuning, and all software changes to FACET were not yet implemented.

Using Amdhal's law the best possible performance acceleration using 'n' concurrent threads can be given by

$$Acceleration\ Factor = \frac{T}{S + \frac{P}{n}} \tag{1}$$

The theoretical maximum possible acceleration by parallelization on a quad-core computer is $1.844\times$. Due to practical considerations such as the overhead of starting and terminating threads, and thread contention for shared hardware resources such as the system memory, the acceleration achieved in practice was only $1.57\times$. Note that this acceleration factor is only due to multi threading and does not include the cumulative effect of software profiling and tuning. The cumulative effect of software profiling and tuning, clustering and multi-threading can yield a total speedup of $8.01\times$.

VI. Implementation with GPU as a Co-processor

The highly parallel Graphics Processing Unit (GPU) is rapidly gaining maturity as a powerful engine for computationally demanding applications. Over the past few years, there has been a marked increase in the performance and capabilities of the GPU. It has evolved from a fixed-function processor built around the graphics pipeline into a full-fledged parallel programmable processor. The GPU's rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of complex, computationally-demanding problems to the GPU.

The present research planned to exploit the emerging computational power of GPUs to accelerate compute-intensive portions of the FACET code. As shown in Figure 5, the main FACET code would run on the host processor and the data-parallel, compute-intensive functions of the FACET code would be executed on the GPU coprocessor.

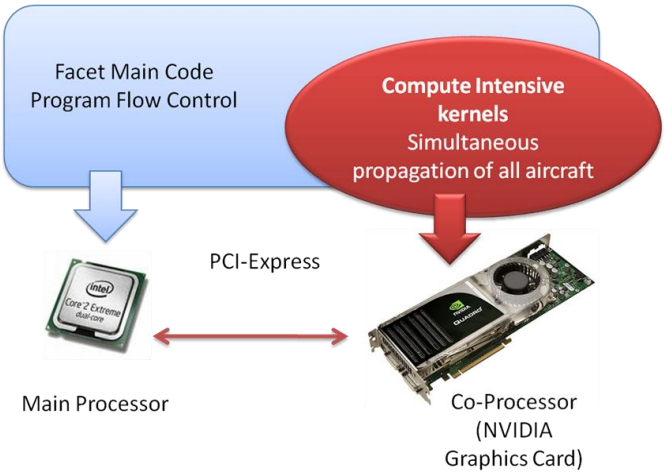


Figure 5. Using GPU as a Co-processor

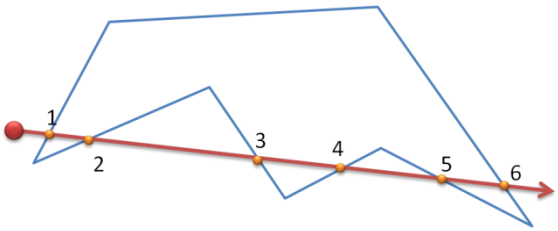


Figure 6. Illustration of the Ray Casting Algorithm

First, the profiling procedure described in Section III was used to identify "hot spots" or time-consuming functions. Among the top few functions was the `sim_inBoundary` function that calculates whether a point is inside a polygon using the Ray Casting Algorithm[18]. In this algorithm, a ray is projected from the point under consideration and the number of intersections of the ray with the edges of the polygon is determined. If the number of intersections is even, then the point is outside the polygon. Conversely, the point is inside for odd number of intersections. This algorithm is parallelizable as the intersection of the ray with every polygon edge can be performed independently. Hence, this algorithm was implemented on the GPU. However, since the number of edges in one polygon is less than one hundred, the time taken for transferring the polygon data from the host to the GPU was significantly larger than the time saved by parallel implementation of the ray casting algorithm on the GPU. Thus the memory access latency cannot be hidden by the parallel computation. Consequently, the GPU was found to take much longer to evaluate the `sim_inBoundary` function. A 24-hour NAS simulation took almost 26 minutes with `sim_inBoundary` implemented on the GPU. This was 10 × slower than FACET with `sim_inBoundary` evaluated on the CPU, at that stage of FACET development.

The conclusion from this exercise is that the function implemented on the GPU must be a parallelizable function with significant run time for a single call and must involve minimum amount of data transfer between the host and the GPU. The most obvious parallelizable function that consumes significant run time is the propagation of every aircraft in the simulation for one time step. The function is parallelizable as the propagation of every aircraft is independent of the other, for a single time step. Any

function that involves aircraft interaction such as conflict detection and resolution can be run after every trajectory propagation step.

The obvious next step is to implement the propagation of all aircraft in FACET for a single time step, in parallel on the GPU. However, FACET is a mixed C and Java application and cannot be readily adapted for implementation on the GPU in its original form. Moreover, FACET runs many other analysis functions that are not central to trajectory propagation. Hence, to facilitate trajectory propagation on the GPU, the core trajectory propagation functions in FACET were re-coded completely in the C language, to create a C Trajectory Predictor (TP) application. Further details on the TP software are described in the following section.

VII. Implementation with GPU as the Primary Processor

This section discusses the implementation in which the GPU was used a primary processor, performing all the trajectory prediction functions, rather than as a co-processor used to accelerate only a small chunk of computationally intensive functions.

A. Trajectory Predictor (TP)

As mentioned earlier, the core trajectory propagation functions in FACET were recoded exclusively in C, to create a C application named Trajectory Predictor. The features of the Trajectory Predictor are:

1. TP can accept flight data input required for the simulation in both the ACES Flight Data Set File (FDS) and the FACET Tracks File (TRX) formats. TP parses flight plans published in standard format defined by FAA.
2. TP incorporates the BADA (Base of Aircraft Data) Database for obtaining the aircraft type specific performance parameters such as preferred climb/descent rates, preferred cruise velocities, altitude ceilings, etc. Note that BADA covers 294 Aircraft Types which covers 80% of Air Traffic in the NAS.
3. TP incorporates NAS data such as airports, named waypoints, airways, Preferred Arrival Route (PAR), Standard Instrument Departure (SID), Standard Terminal Approach Route (STAR) and sector boundary data.
4. TP uses a 3-D hash map to rapidly identify the current sector for every aircraft at every time step in the simulation.
5. TP returns the following flight trajectory data at every 30 second interval: time since the start of simulation, flight mode (0 – preflight, 1-climb, 2-cruise, 3-descent, 4-landed), latitude, longitude (degrees), altitude, true air speed, altitude rate, heading angle, flight path angle and sector index. Note that the size of the entire trajectory data for ~35,000 flights in a 24 hr NAS simulation can be as large as 500 MB.
6. The effect of the wind field is incorporated in the trajectory prediction. TP uses the wind field data from RUC weather forecast (13 km grid) published by NOAA/NCEP.

1. Trajectory Prediction Equations

As in other trajectory prediction software, the TP employs the Euler's integration method, with the following equations[4] to propagate a flight:

Calculation of great-circle heading between current point and the next target waypoint of the flight plan:

$$\psi_k = \tan^{-1} \frac{\sin(\tau_w - \tau_k)\cos(\lambda_w)}{\sin(\lambda_w)\cos(\lambda_k) - \sin(\lambda_k)\cos(\lambda_w)\cos(\tau_w - \tau_k)} \tag{2}$$

Latitude Propagation:

$$\lambda_{k+1} = \lambda_k + \frac{V_{G_k}\cos(\psi_k)}{(R_E + h_k)}\Delta t \tag{3}$$

Longitude Propagation:

$$\tau_{k+1} = \tau_k + \frac{V_{G_k} \sin(\psi_k)}{(R_E + h_k)\cos(\lambda_k)} \Delta t \tag{4}$$

Altitude Propagation:

$$h_{k+1} = h_k + \dot{h_k} \Delta t \tag{5}$$

Here, ψ_k is the great-circle heading between current point (λ_k, τ_k) and the next target waypoint (λ_w, τ_w) of the flight plan and (λ_k, τ_k, h_k) are the latitude-longitude-altitude coordinates of aircraft at the current time step. V_{G_k} is the ground speed derived from the preferred true airspeed at the flight altitude for specific aircraft type derived from the BADA tables and the RUC (Rapid Update Cycle) wind speed data resolved along the flight path. R_E denotes radius of the Earth and $\dot{h}_k$ is the preferred climb/descent rate for the specific aircraft type obtained from the BADA data. Note that the propagation time step Δt chosen for the trajectory prediction is 30 seconds.

2. Multi-Threaded Version of Trajectory Predictor

Much like the multi-threaded version of FACET described in Section V, a multi-threaded version of TP was also developed. This code automatically detects the number of cores on the computer. The aircraft list is then split into multiple lists and each aircraft list runs in parallel on different cores. Table 1 displays the run times for a 24-hour, 35,000 flight NAS simulation for the multi-threaded TP implementation executed on a quad core computer. The speedup achieved by multi-threaded TP is $2.68\times$.

Table 1. Run Times for a 24-Hour NAS Simulation for Multi-Threaded Trajectory Predictor Implementation

Num Threads	Propagation Time (s)	Acceleration Factor
1	66.31	1
2	37.57	1.76
3	29.82	2.22
4	24.76	2.68

3. Comparison between CARPAT Trajectory Predictor Output with FACET and ACES Trajectory Outputs

This section compares the trajectory output generated by the CARPAT Trajectory Predictor with the trajectories generated by FACET and ACES. The primary objective is to demonstrate that the significant speedup of the trajectory propagation in TP is achieved without a loss of fidelity.

Figure 7 shows the comparison between the time histories of latitude, longitude and altitude for trajectories generated by FACET and TP. It may be observed that the trajectories match closely.

Figure 8 shows that the time histories of latitude, longitude and altitude as predicted by TP are offset in time with the time histories generated by ACES. This is because ACES does not use aircraft preferred unimpeded climb schedules below 10,000 ft, but uses empirical TRACON models to calculate TRACON transit time. ACES uses aircraft preferred climb and descent schedules above 10,000 ft. Figure 9 shows that the climb trajectories predicted by Trajectory Predictor and ACES match closely above 10,000 feet. Figure 10 shows that the climb scheduled executed by both Trajectory Predictor and ACES match closely with the BADA climb schedule above 10,000 feet.

The excellent match between the trajectories generated by Trajectory Predictor, FACET and ACES demonstrate that that the fidelity of the trajectory predictions has not been compromised in order to achieve the speedup on the GPU.

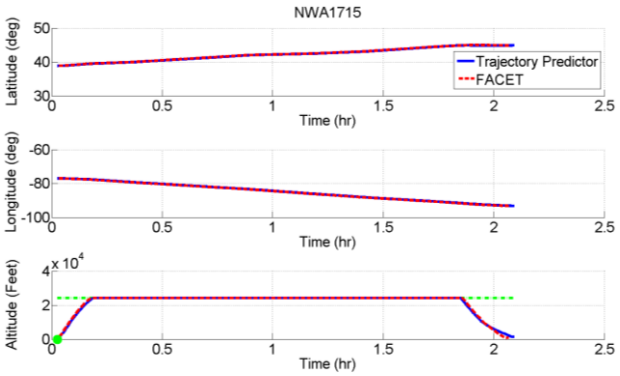


Figure 7. Trajectory Comparison with FACET

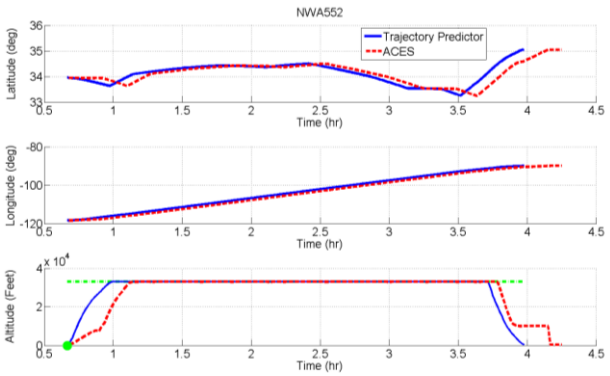


Figure 8. Trajectory Comparison with ACES

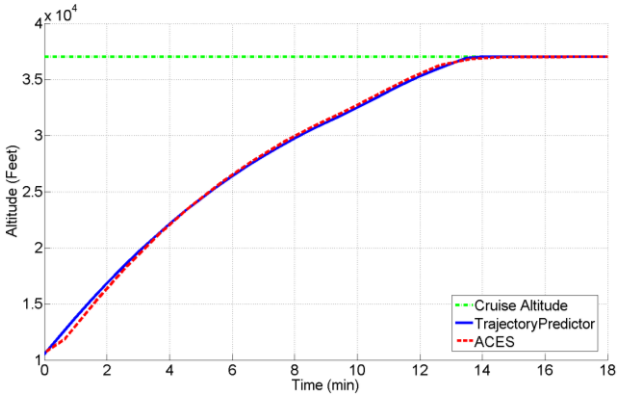


Figure 9. Climb Trajectory Comparison with ACES: Above 10,000 ft

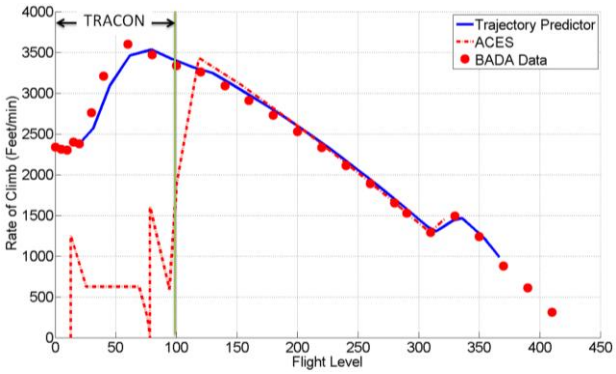


Figure 10. Comparison of Climb Schedules between TP and ACES

B. GPU Implementation

After the development of Trajectory Predictor code in C was completed, this code was ported to run on the GPU using CUDA. This section describes the porting of the C code to the GPU in detail.

1. Compute Unified Device Architecture (CUDA)

Until very recently, the only means available for accessing the power of the GPU was by formulating the compute problem as an equivalent graphics rendering problem, and then coding it in OpenGL. This involves a fairly complex mapping process. In addition to OpenGL syntax, expertise in graphics rendering techniques is essential to establish the correspondence between the graphics rendering processes and the kernel of the problem being solved. Since this involves extensive code rewrite, the process is extremely error prone.

Recently, NVIDIA released Compute Unified Device Architecture (CUDA)[19] which is a set of software tools for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API. The CUDA software stack is composed of several layers: a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries for Fast Fourier Transforms and Basic Linear Algebra Subprograms (CUFFT and CUBLAS, respectively). The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance. The CUDA API comprises an extension to the C programming language, and has a shallow learning curve. The CUDA toolkit also includes the 'nvcc[20]' (NVIDIA C compiler), which can compile targets for both host CPU and GPU code. CUDA also has a hardware thread manager that can automatically handle threading, without requiring a programmer to explicitly write threaded code. CUDA has three key abstractions that provide parallelism:

1. A block of threads that can execute the parallel parts of the applications as kernels - functions that are called from the host that run simultaneously on the device.
2. On-chip shared memories that can be accessed by all threads within a block, thus providing faster data read/write.
3. Barrier synchronization that will block all threads from running until all previous CUDA calls have been completed.

CUDA code also imposes the following restrictions:

1. Dynamic memory allocation inside a structure is not allowed.
2. Host functions (executing on the CPU) cannot be called from the device functions (executing on the GPU).
3. Recursive functions are not allowed.
4. Linked lists cannot be used. The data access in linked lists is performed sequentially and they are not suited for parallel access needed in a parallel implementation on the GPU.
5. Only integer and single-precision floating-point numbers are supported. Note that Tesla and new Fermi architecture supports double precision float.

Note that due to the presence of the above factors in FACET C code, especially the heavy use of linked lists, the FACET C code required substantial modification for GPU implementation. Hence, the trajectory propagation code of FACET was completely recoded to create Trajectory Predictor.

2. GPU Hardware

All the numerical computations reported in this paper were generated on a desktop computer with NVIDA Tesla C 1060[21] high-performance graphics card (See Figure 11). All timing results for GPU implementation given in this paper were obtained by executing the TP CUDA code on this workstation.

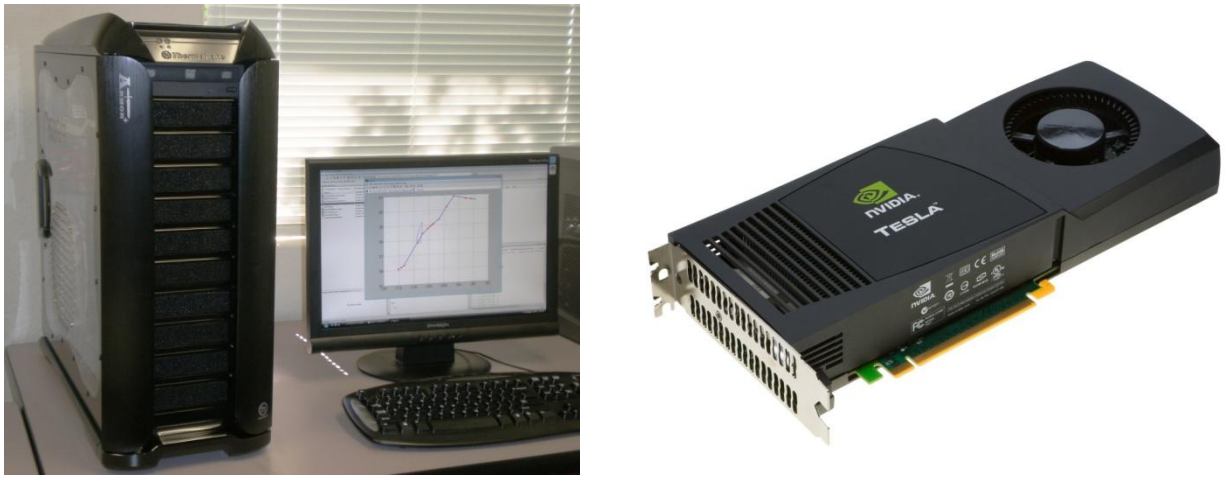


Figure 11. Desktop WorkStation with Tesla C 1060 High Performance Computing Graphics Card used in the Present Research

3. Features of CUDA Implementation of Trajectory Prediction[22]

1. *Trajectory data streaming from GPU to host memory*: As mentioned elsewhere in this paper, a large amount of data (~500 MB for 35,000 aircraft in a 24 hr NAS simulation) must be transferred from the GPU to the host memory. The CUDA implementation takes advantage of data streaming to enable concurrent trajectory propagation and the transfer of data sets, and thus achieves overall application speedup. To perform data streaming, the aircraft list is divided into two sub-lists. When the propagation of sub-list 1 is in progress, the trajectory data for aircraft in sub-list 2 is streamed back to the host, and vice versa. Data streaming is made possible by the asynchronous memory transfer capabilities of the CUDA implementation. Asynchronous memory

transfer enables a non-blocking transfer, where control is returned back to the host immediately and the kernel can be executed simultaneously. Asynchronous memory transfer is performed as DMA (Direct Memory Access) without host CPU involvement and hence has higher bandwidth. DMA requires that the host memory has to reside in page-locked memory (physical address cannot be changed by the operating system), so the amount of host memory in the trajectory buffer must be kept low.

2. *Memory coalescing*: In the original C implementation of TP, the aircraft data is stored in an array of structures, where each aircraft structure stores the various data elements for a single aircraft. In this case, data for a single aircraft field, aircraft altitude for example, do not reside contiguously in memory. If the data fields for various aircraft reside continuously in memory such that the i^{th} thread access the i^{th} element in the data array, the memory accesses for multiple threads can be coalesced into a single access thus making the memory access faster. The memory coalescing implications of the aircraft data choices are illustrated in Figure 12. To facilitate coalescing, individual arrays for each aircraft data field were created in the modified implementation.

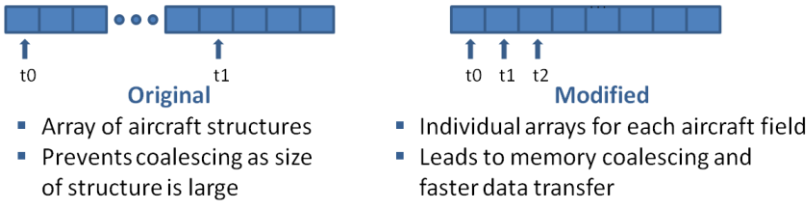


Figure 12. Memory Coalescing Implications of Aircraft Data Structure Choices

3. *Minimizing branching in TP code*: The NVIDIA GPUs are built on a scalable array of multithreaded Streaming Multiprocessors (SMs). To manage hundreds of threads running several different programs, the multiprocessor employs a new architecture called SIMT (Single Instruction Multiple Thread). The multiprocessor SIMT unit creates, manages, schedules and executes threads in groups of 32 parallel threads called *warps*. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If the threads of a warp diverge via a data dependent conditional branch, the warp serially executes each branch path taken. Thus, data dependent branching in the code increases the proportion of serial computations and the run times increase. While porting the TP code to the GPU, special attention was given to reducing or completely eliminating the `if-else` statements and `while` loops whenever possible.

Apart from the above considerations, various design parameters such as number of threads per block and number of registers allocated per thread, etc. were tweaked to achieve optimum performance.

4. GPU Implementation Results

The results of the GPU implementation are summarized in Table 2.

Table 2. GPU Acceleration Results

	Run Time (s)	Speed Up		
		w.r.t FACET	w.r.t CPU	w.r.t Multi-Threaded
FACET	586.86			
Trajectory Predictor on CPU	46.56	12.60		
Trajectory Predictor on CPU (Multi-Threaded)	24.54	23.91	1.90	
Trajectory Predictor on GPU	**2.42**	**242.50**	**19.24**	**10.14**

Note that the run time for a 24 hour simulation of the NAS using the original baseline FACET implementation was 586.86 seconds. An identical simulation generating the trajectory prediction data can be executed by the CARPAT Trajectory Predictor in 2.42 seconds giving an acceleration factor of 242.50 ×. When compared with the optimized, non-threaded Trajectory Predictor C code running on a CPU, the GPU gives 19.24 × faster performance. Even when the multi-threading opportunities offered by quad-core CPUs are considered, the GPU still provides 10.14 × faster performance.

VIII. Summary and Concluding Remarks

The objective of the research presented in this paper was to develop the prototype for a computational appliance for rapid prediction of aircraft trajectories that leverages recent developments in High Performance Computing technologies. The development strategy was to develop efficient algorithms for fast trajectory propagation and implement the lean, optimized code on emerging high performance computing architectures. The following HPC technologies were explored:

1. Cluster Computing
2. Multi-Threading for Multi-Processor/Multi Core CPUs
3. General Purpose Computing using Graphics Processing Units

The most successful and promising implementation turned out to be the Trajectory Predictor on the GPU, which can perform a 24 hour trajectory prediction of 35,000 aircraft the in the NAS in less than 2.5 seconds.

The results of the present research are summarized in the following.

1. FACET was profiled using the AMD Code Analyst and the hot-spots in the code were analyzed to improve run-time performance. The modifications to FACET code resulted in an acceleration of 3.10 × over original FACET.
2. A Linux cluster with commodity microprocessors connected over the Ethernet protocol was set up. A cluster implementation of FACET was developed using Java RMI, to run on this cluster. The resultant acceleration was 5.34 × over the baseline FACET.
3. A multi-threaded FACET implementation was developed using the `pthreads` library to accelerate performance on multi-processor/multi-core processors. The speedup of 8.01 × was achieved over the original FACET implementation.
4. Trajectory Predictor code was developed exclusively in C, modeled after the core trajectory prediction functions in FACET. Trajectory Predictor was implemented in CUDA for execution on an NVIDIA Tesla C1060 High Performance Graphics Processing Unit. CUDA implementation of the trajectory predictor code was able to perform a 24 hour simulation of the National Airspace System in less than 2.5 seconds which is 240 × faster than a 24 hour simulation in the baseline FACET software.
5. The trajectory predictions generated by CARPAT trajectory predictor were validated by comparisons with trajectories generated from FACET and ACES.

Acknowledgments

This research was supported under NASA Ames Research Center Contracts NNX07CA13P, NNX08CA02C, and NNA10DC12C. We are grateful to Dr. Shon Grabbe and other research scientists at NASA Ames for their encouragement and support of the present work.

References

[1]"Concept of Operations for the Next Generation Air Transportation System", Joint Planning and Development Office, Version 0.2, July 2006.

[2]GPU: http://en.wikipedia.org/wiki/Gpu

[3] CUDA Community Showcase: http://www.nvidia.com/object/cuda_apps_flash_new.html#

[4]Bilimoria, K. D., Sridhar, B., Chatterji, G. B., Sheth, G., and Grabbe, S., "FACET: Future ATM Concepts Evaluation Tool," *3rd USA/Europe Air Traffic Management R&D Seminar*, Naples, Italy, June 2000.

[5] Tandale, M. D., Menon, P. K., "Rapid Prediction of Air Traffic for Trajectory Based Operations," *8th Aviation Technology, Integration and Operations Conference*, Anchorage AK, 14-19 September, 2008.

[6]Computer Cluster: http://en.wikipedia.org/wiki/Computer_cluster

[7]Menon, P. K., Diaz, G. M., Tandale, M. D., and Kwan, J., "CARAT# - A Rapid Prototyping Software for Developing Next-Generation Air Traffic Management Algorithms," *Final Report Prepared under NASA Contract No. NNA05BE64C*, Vol. I, Optimal Synthesis Inc, Palo Alto, CA, November 21, 2006.

[8]Menon, P. K., Diaz, G. M., Vaddi, S. S., and Grabbe, S. R., "A Rapid Prototyping Environment for En Route Air Traffic Management Research", *AIAA Guidance, Navigation and Control Conference*, August 15-18 2005, San Francisco, CA.

[9]Menon P. K., Cheng V. H. L., Kwan J. S., Lin V., Peng W., W. Krueger W., Manikonda V., "Open-Source Based Software Systems for Linking Disparate Software Components," *Final report prepared under NASA NRA Contract No. NNL08AA12B/NNL08AB71T*, Aug 2009.

[10]Integrated Terminal Weather System: /http://www.raytheon.com/capabilities/products/itws/

[11]RUC Main Website: http://ruc.noaa.gov/

[12]Wikipedia Article on Software Profiling: http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29

[13]AMD CodeAnalyst Performance Analyzer for Windows and Linux programs on AMD processors: http://developer.amd.com/cawin.jsp http://developer.amd.com/calinux.jsp

[14]Message Passing Interface (MPI) http://www-unix.mcs.anl.gov/mpi/

[15]JAVA Remote Method Invocation (RMI): http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[16]Wikipedia article on POSIX threads: http://en.wikipedia.org/wiki/POSIX_Threads

[17]POSIX Threads Programming tutorial at the Lawrence Livermore National Laboratory website: https://computing.llnl.gov/tutorials/pthreads/

[18]Wikipedia article on the Ray Casting Algorithm: http://en.wikipedia.org/wiki/Point_in_polygon

[19]NVIDIA: CUDA http://www.NVIDIA.com/object/cuda_home.html#

[20]The CUDA Compiler Driver NVCC: http://moss.csc.ncsu.edu/~mueller/cluster/NVIDIA/2.0/nvcc_2.0.pdf

[21]NVIDIA Tesla C1060 High Performance Computing Graphics Card: http://www.NVIDIA.com/object/product_tesla_c1060_us.html

[22]NVIDIA CUDA Programming Guide 2.0: http://developer.download.NVIDIA.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf