Performance Comparison of NVIDIA accelerators with SIMD, Associative, and Multi-core Processors for Air Traffic Management

Alfred Shaker
Kent State University
Kent, Ohio
ashaker@kent.edu

Johnnie W. Baker
Kent State University
Kent, Ohio
jwbaker@kent.edu

Gokarna Sharma
Kent State University
Kent, Ohio
sharma@cs.kent.edu

Mike Yuan
Hefei University of Technology
Hefei, China
myuan2012@126.com

ABSTRACT

Basic tasks for Air Traffic Management (ATM) will be implemented using NVIDIA's CUDA language on an NVIDIA device and compared to the performance of SIMD, associative, and multi-core processors doing the same tasks; CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. To do this, we create a simulation of an airfield with constantly moving aircrafts. The basic tasks that will be used in the evaluation are: *tracking and correlation*, *collision detection*, and *collision resolution*. These are the most compute-intensive of the ATM tasks, so they will give us a good measure of the capabilities of the NVIDIA device. In our previous research, it was shown through a performance comparison between an associative SIMD processor and a multi-core processor that the associative SIMD processor can execute these tasks in linear time without missing a single deadline, whereas the multi-core processor required significantly more (exponential to be precise) time. Additionally, the multi-core processor regularly missed a large number of deadlines. Our goal in this paper is to determine whether we could get SIMD-like results using a CUDA implementation of the basic ATM tasks on NVIDIA accelerators. This will allow the novel use of increasingly popular NVIDIA accelerators for ATM tasks in place of a SIMD processor that was found to be a good fit for these tasks in the previous research. Our ATM implementation and evaluation on three different NVIDIA accelerators shows that all three NVIDIA accelerators can provide a SIMD-like implementation for ATM. Curve-fitting with MATLAB shows that the performance of NVIDIA accelerators increases only slightly faster than a linear graph.

ICPP '18 Comp, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6523-9/18/08…$15.00
https://doi.org/10.1145/3229710.3229757

CCS CONCEPTS

• **Theory of computation → Parallel algorithms**; • **Computing methodologies → Parallel algorithms**; **Parallel programming languages**; • **Computer systems organization → Parallel architectures**;

KEYWORDS

Air traffic management; Air traffic control; Parallel computing; NVIDIA accelerators; CUDA programming language; SIMD; Multiprocessors; Associative processors; Collision detection and resolution

ACM Reference Format:
Alfred Shaker, Johnnie W. Baker, Gokarna Sharma, and Mike Yuan. 2018. Performance Comparison of NVIDIA accelerators with SIMD, Associative, and Multi-core Processors for Air Traffic Management. In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3229710.3229757

1 INTRODUCTION

Large real-time applications rely on powerful and versatile architectures that support large scale parallelization to ensure that deadlines are met during execution. Such applications are also a great way to measure the power and throughput of parallel architectures. We will be looking at perhaps the most popular and widely discussed real-time application, *Air Traffic Management* (ATM). ATM is a real-time application that needs to continuously monitor the behavior of the aircrafts while performing various calculations and time-consuming tasks on typically many thousands of aircrafts. In real-time applications, missing certain types of deadlines can be catastrophic in real-life situations. These types of deadlines are called *hard* deadlines and apply to many of the deadlines that occur in ATM [2, 4]. These deadlines are known for ATM when developing the software, as in this case, we know one or more tasks have to meet the deadline that occurs at the end of each of the one-half second intervals during which they execute. Developers have to use the features of the architecture that they are working on to fully optimize the software and make sure they are taking advantage of the strengths of this architecture. They also need to make sure that

they do not let the shortcomings of their architecture hinder their performance.

Solutions to ATM have been implemented in a few different ways in the past [11, 13]. It has been implemented on multi-core systems where aircraft data was stored in shared memory that all processors in the system could access. This was found to be not very efficient as the nature of ATM systems proved too complex and difficult for multi-core systems (see [13], Sections 5, 9, 10). We call a system *predictable* (or deterministic) if the time it requires to perform a specific constant time computation is always the same [2]. Due to being asynchronous, multi-core computations are not predictable and the time they require to perform a constant time computation can vary widely. Due to this, they could not be guaranteed to meet deadlines. In compute-intensive applications like ATM, as shown by the previous research [12, 13], they consistently miss a large number of deadlines.

ATM has also been implemented on an enhanced *single instruction multiple data* (SIMD) computer, called an *associative processor* (AP), that was explicitly designed for the ATM-type applications. This AP included hardware that allow some very useful ATM capabilities such as broadcasting, associative searches, and maximum and minimum reductions to be executed in constant time. A particularly important feature of a SIMD architecture is that it is a synchronous system and is deterministic. Note that APs, with their additional hardware, still retain these properties. As shown in [12, 13], this architecture was able to execute even the most compute-intensive ATM tasks in linear time. ATM software developed for this system handled the basic ATM tasks with only 4017 lines of assembly code and about 1600 lines of sequential code for its control unit (see [13], Section 8.2.7).

In contrast, it is generally accepted that all ATM software for multi-core systems runs in exponential time [10]. The past and current FAA ATM software systems are extremely large and the research and development of these systems have involved many of the US's large high tech companies and research groups like Lockheed Martin, Computer Science Corporation, IBM, MITRE, and NASA [4]. Identification of hardware that can supports software with a substantially lower complexity for ATM could have a major impact on a wide range of other applications as well.

1.1 Contributions

In this paper, we will be using basic compute-intensive ATM tasks to compare the performance of these tasks on NVIDIA accelerators with their performance on the ClearSpeed SIMD, the STARAN AP, and the Intel Xeon multi-core processor. The main ATM tasks that we will focus on in this paper are the following three: (i) *Tracking and Correlation*, (ii) *Collision Avoidance*, and (iii) *Collision Resolution* [4, 11, 13]. These three tasks are the most time-consuming ATM tasks and therefore give us the best idea of how well our software is performing on the three NVIDIA devices we will use: (i) GeForce 9800 GT, (ii) GTX 880M, and (iii) Titan X (Pascal). Despite being an old card with Compute Capacity of 1 and working with really old CUDA architecture, GeForce 9800 GT is still able to handle this application very efficiently. The remaining two are newer cards from NVIDIA with relatively small and large Compute Capacity, with the GTX 880M running on a personal laptop and the Titan X (Pascal) running on a dedicated research computer. The Titan X has a Pascal architecture.

The implementation developed in this paper is based on the same algorithms for the basic ATM tasks implemented on the ClearSpeed SIMD and STARAN AP systems [12, 13], but are executed very differently, due to the difference in the architecture of the device on which they are running. For the NVIDIA implementation, we use the CUDA architecture and language to parallelize a C program that has been created from scratch and re-written many times to achieve the optimal performance on the device [5].

We will be primarily comparing our NVIDIA implementation with an AP implementation, and the multi-core implementation from [12, 13]. In both [12, 13], the ClearSpeed CSX600 accelerator was used to emulate an AP. The implementation was done using the Cn (ClearSpeed) language, which like CUDA, is an extension of the familiar C language. The ClearSpeed CSX600 accelerator has two chips, each chip consisting of a SIMD system with 96 processing elements connected together by a ring network.

It was shown in [12, 13] that the associative SIMD processor can execute all three basic ATM tasks in linear time without missing a single deadline, whereas the multi-core curve increases rapidly and possibly exponentially in what is essentially certain to be an exponential curve [10]. Additionally, the multi-core processor regularly missed a large number of deadlines. Our research on three different NVIDIA accelerators indicates that all three NVIDIA accelerators can provide a SIMD-like implementation for ATM. Curve-fitting with MATLAB shows that the performance of NVIDIA accelerators is slightly faster than a linear graph.

Recently, Thompson *et al.* [11] provided a parallel CUDA implementation for airspace deconfliction. *Airspace deconfliction* here means, as in our case, the collision detection and collision avoidance tasks. Specifically, they compared parallel and sequential implementations of their deconfliction algorithm, which ensures that the aircraft avoids collision with terrain; this is called *terrain avoidance* in [12, 13]. However, they do not consider collisions between aircrafts. This is different from the collision detection approach we use in this paper, which handles aircraft-to-aircraft collisions. Furthermore, the algorithm that we use is different from their algorithm. In summary, we have the following four contributions:

(1) We implemented the algorithms to handle three basic compute-intensive ATM tasks in NVIDIA-CUDA.
(2) We created a simulation of an airfield with constantly moving aircrafts for experimentation.
(3) We compared the results obtained in three different NVIDIA accelerators GeForce 9800 GT, GTX 880M, and Titan X (Pascal) to the SIMD, AP, and multi-core processors. The multi-core processor used is a 16-core Intel Xeon processor.
(4) The results show that NVIDIA accelerators can provide a SIMD-like linear execution for ATM; we advocate this as our main contribution. This finding might have implications on other real-time applications on the usability of a SIMD-like implementation through NVIDIA accelerators.

1.2 Roadmap

We proceed as follows. We provide brief background on different systems (AP, SIMD, Multi-core) used in our research in Section 2.

The ATM problem is discussed in Section 3. In Section 4, we provide an overview of a NVIDIA CUDA implementation and then provide details in Section 5. We provide experimental results in Section 6 and conclude in Section 7 with a short discussion.

2 OVERVIEW OF SIMD, AP, AND MULTI-CORES

2.1 SIMD

SIMD is the style of design found in most early parallel computers. The SIMD programs are stored in a processor called the control unit. The instruction stream (IS) is a network that connects the control unit to multiple processing units called processing elements (PEs). The control unit and IS are also collectively called an IS. The PEs execute the IS instructions synchronously, with each PE executing on data from its own memory (Fig. 1). These are the main components of a SIMD computer. Each PE is primarily an arithmetic logical unit (ALU), which is responsible for performing arithmetic and logical operations [1, 7, 12, 13].

There are three different types of parallel systems that are sometimes included when discussing SIMD computers: traditional SIMD discussed above, vector machines, and short SIMD machines. Traditional SIMD includes the Goodyear Aerospace MPP, the MasPar computers, and Thinking Machines CM2. Vector machines involve the use of pipelined processors. And finally, short SIMD machines evolved from desktop machines as they grew in processing power and were able to support real-time applications like video-gaming and video processing. The NVIDIA architecture belongs in this category. In this paper, we will include only the traditional SIMD when we refer to this category [13].

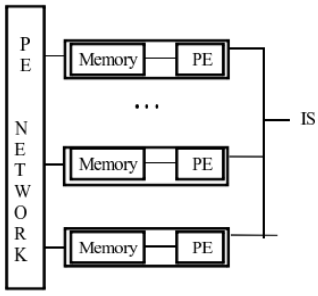

Figure 1: An illustration of a SIMD architecture

2.2 AP

An associative processor is an enhanced SIMD architecture that was first used in the STARAN computer built at Goodyear Aerospace during the early 1970's. The associative architecture was explicitly designed for the purpose of performing ATM. AP has various characteristics that allow it to support ATM much more efficiently than usual SIMD architectures. The SIMD hardware of an AP is designed to support constant time operations such as broadcasts, associative searches, maximum/minimum reductions and more. While associative computing was a widely discussed topic much earlier, a formal definition for the AP model was first given in [6, 7]. A detailed explanation of these associative properties is given in [13]. Additional information can be found in [1, 6, 7, 12].

2.3 MIMD

MIMD is generally considered to be the most important class of parallel computers, as it includes most computers being built. With MIMD computers, each processor executes one instruction stream on one data stream (Fig. 2). These executions occur simultaneously but the processors operate asynchronously. This asynchronous execution makes the MIMD computation generally considered to be more efficient, as it is not necessary for all processors to complete a step at a time before going on to the next step. Since the instructions they are executing are different and require different amounts of time, requiring the execution of each instruction to be synchronous is usually both inefficient and unnecessarily restrictive. There are two key methods of communication that MIMD computer utilize, namely by the use of shared memory (called multiprocessors) and by the use of message passing (called multicomputers). MIMDs are considered less restricted and more important than SIMD computers to most researchers. But despite that, a lot of different NP-hard problems occur with MIMD execution of real-time applications. Some of those include, but are not restricted to, load balancing, race conditions, non-determinism, dynamic scheduling, etc. Typically, when evaluating the performance of a MIMD application, only average case performance is considered since the worst cases that can occurs are usually much worse than the average performance [1, 12, 13].

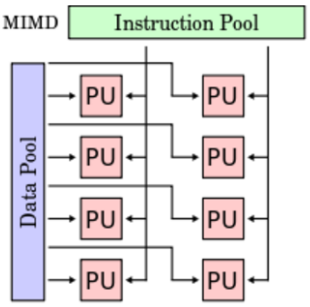

Figure 2: An illustration of a MIMD architecture

3 AIR TRAFFIC MANAGEMENT SYSTEM

An ATM system is a real-time system that continuously monitors and manages thousands of aircrafts moving in an airfield while processing large volumes of data and computations for all those aircrafts. The tasks of the ATM systems occur periodically at different time intervals, based on the task, and must meet deadlines in order to avoid catastrophic consequences. The data maintained for all the aircraft being monitored constitutes a dynamic database due to the fact that this data is continuously being accessed and changed at a very rapid rate. There are some minor variations in the period length and the frequency of the execution of the ATM tasks. The times used here are those used by Goodyear Aerospace in their ATM software for the STARAN. This software was used in their 1972 demonstration to FAA of the STARAN performing ATM at the Dulles Airfield using real radar. For more information and a video, see [13]. The tasks that we focus on are the following three most compute-intensive ATM tasks:

- (**Task 1**) Tracking and Correlation which is executed every half second,
- (**Task 2**) Collision Detection which is executed every 8 seconds, and

- (**Task 3**) Collision Resolution which is executed every 8 seconds as part of **Task 2**'s function, if required.

Task 1 is the main cause for rapid and continuous data changes in the aircraft records, as it tracks the flight movement of the aircrafts and matches them with their appropriate radars, as discussed in more detail in Sections 4 and 5. We consider the aircrafts in a 256 nautical mile by 256 nautical mile bounding area on a 2D plane with varying altitudes. Due to the nature of this task, the ATM system needs to execute this task once for each aircraft being tracked during every half-second period and this execution needs to be optimized so that this task is always completed prior to the end of each half-second period. A task cannot execute if the previous task is still being processed. It is essential that the execution of the earlier tasks in a period be completed sufficiently early for the execution of the remaining tasks in that period to be able to complete their execution before the end of that period. Remaining tasks that may not have time to complete their execution before the end of the period must be skipped (may be aborted in some implementations) so that the execution of the tasks in the next period can start on time. Luckily, these deadlines are known at the time the program is created, so for deterministic architectures its possible for us to optimize the code so that these deadlines will be met each time they are executed [1, 12, 13].

4 IMPLEMENTATION OVERVIEW FOR ATM IN NVIDIA CUDA

The NVIDIA-CUDA program we develop centers around two main kernel functions that together perform the 3 main ATM tasks that we will be using to evaluate the performance of NVIDIA devices. These two functions are: *TrackDrone* and *CheckCollisionPath*. The first funcation performs **Task 1** on all aircrafts simultaneously, matching the correct radar points with their corresponding expected locations for each aircraft. The latter function combines **Task 2** and **Task 3** into one kernel function that projects the aircraft location 20 minutes ahead, and using that data and Batcher's algorithm [13], determines if an aircraft is on a collision path with another aircraft. If the aircraft is found to be on a collision path with another aircraft and the time to collision becomes critical, then a new proposed path is selected by rotating a few degrees left or right from the current path. Before the aircraft is instructed to follow this new path, it must be checked against all other aircraft flights to make sure it is collision-free where no critical collisions could occur. The reason for having both functions in the same kernel is that it cuts overhead for memory and data transfer because we don't have to get information from one kernel function and transfer it back to the host from the device, then feed that into a totally different function to perform **Task 3**, then go back to **Task 2** and check that new path. This way, we were able to optimize it better and have it run more smoothly in one kernel function. Just like the tracking function, this one performs the tasks on all aircraft simultaneously and even has variables to check if an aircraft has already been found to be on a dangerous collision path, so that two threads don't try to manipulate the same aircraft and cause a memory error.

The program has what is called a *major cycle* that is split into 16 *periods*, each representing a half second of the 8 second major cycle. **Task 1** is performed every period, so every half second, while **Task 2** and **Task 3** are performed once in the 16th period of every major cycle. The program also has a function that is only performed once at the start of the program to set up the needed aircraft data, and a function to generate the simulated radar data before every period based on the aircraft data either obtained from the initial setup or from the previous period, depending on where we are in the process of this simulation.

4.1 Kernel Functions

The NVIDIA-CUDA program uses *SetupFlight*, *GenerateRadarData*, *TrackDrone*, and *CheckCollisionPath* functions. We describe them separately below.

SetupFlight: This function is called once in the beginning of the program to create an aircraft at its initial (x, y) location. Random values are selected between 0 and 128 for both x and y. Next, a random number is chosen between 0 and 50 for a temporary variable. If this number is even, then the value of x will be negative. A second random number is selected between 0 and 50 and if it is odd, the value of y will be negative. The purpose of the *SetupFlight* kernel is to create a virtual airfield with the aircraft positions (x, y) satisfying $-125 \leq x, y \leq 125$. Note that many planes will disappear from this 125 grid in a fairly short period of time, especially if flying at a faster speed. To avoid this, when an aircraft exits this grid at location (x, y), then it is programmed that another aircraft with the same speed and direction of flight is re-entered into the grid at the location $(-x, -y)$.

In order to assign each aircraft a velocity, we generate a random speed S, where $30 \leq S \leq 600$. The direction of this flight is determined by selecting a random value Δx between 30 and 600. The value dx is the distance in nautical miles the aircraft travels parallel to the x-axis each hour and has magnitude $|dx| = \Delta x$. We randomly determine the sign of dx using the procedure used earlier for selecting the aircraft's location. The value dy is the distance in nautical miles that the aircraft travels each hour parallel to the y-axis and has magnitude $|dy| = \sqrt{(S^2 - |dx|^2)}$. Whether dy is positive or negative is again determined randomly. The velocity of the aircraft is $v = dx + dy$. The signed values of dx and dy allow aircraft to fly in all directions within the simulated airspace. Each period in our 8 second cycle is one-half a second. Therefore, dx is converted from nautical miles per hour to nautical miles per period by dividing it by 7200. Likewise, dy is converted to nautical miles per period by dividing its original value by 7200. The altitude that the aircraft is traveling at will also be selected randomly.

GenerateRadarData: The purpose of this function is to simulate the aircraft locations that are obtained from radar towers during each half-second period. While most aircraft in the US are within the range of 2 to 6 radars, a radar report may not be obtained for some aircraft during some periods. We simplify this situation by assuming that at most one radar is received for each aircraft each period.

The expected location of a plane each period is $(x + dx, y + dy)$, where (x, y) is its location in the preceding period. On the other hand, the radar for an aircraft gives its exact location. There are many reasons why an aircraft's calculated position rarely agrees with its true position. For example, the prevailing wind may slow

down or speed up flight speed of the aircraft, or may push it to the right or left of its path. To simulate the radar position of an aircraft, we add a small random noise to the both coordinates of its expected position. This noise in either coordinate can be either positive or negative and this is randomly determined. To prevent the list of simulated radar locations from having the same order as the aircraft matching them, we reorder items in the radar list.

While ATM is moving towards replacing much of the use of radar with GPS, radar will still play a critical role in ATM for some time. Aircraft can turn off transponder systems and essentially disappear if radar is not being processed. This occurs regularly with aircraft being used for smuggling drug, human trafficking, and other illegal activities. Currently, radar is used as a backup for transponders, which are called secondary radar and provides aircraft location information obtained directly from the aircraft. All radar in the USA is saved and can be used to retrace the flight of aircraft that has disappeared over large uninhabited areas including oceans.

Current air traffic control systems are unable to process most of the radar received, due to the computational complexity involved with processing radar in addition to performing the other ATM tasks [4]. However, this makes the processing of all radar as a part of ATM an ideal tool to use in testing the ability of different architectures to handle real-time computations.

Each thread is set up to take one aircraft and use it to create its radar data. Once each thread has generated the appropriate radar data, we copy the data back to the host and the radar data array is split into fourths and each fourth is reversed in the host so as to mix up the array so that our tracking and correlation function will have some work to do trying to find the right radar. Otherwise, if they are left in the order they are created, then the tracking and correlation function would just take each of its own threads and be able to easily match them to the radar as $radar[0]$ will match with $drone[0]$. But we want to simulate the real life situation where $radar[0]$ does not necessarily match with $drone[0]$ so we jumble up the radar reports. This function is called once after *SetupFlight* outside the task cycle (the loop calling *TrackDrone* every half second and *CheckCollisionPath* every 8 seconds). After that, this function is called every half second at the end of the task cycle to create the new radar data for the next time the cycle starts up and *TrackDrone* is called.

TrackDrone: This function performs **Task 1** that is expanded upon in great detail in Section VI. This kernel function is called every half second, along with a new set of generated radar data.

CheckCollisionPath: This is the last kernel function that the program uses and this is the function that performs **Task 2** and **Task 3**. This is the most involved function in the program as it does quite a few things that all work well together and are better to have in one function rather than split them into different kernel functions. Functions for **Task 2** and **Task 3** are not called as often as the function for **Task 1**, as it is called only once during the every 8 second major cycle during one of the 16 half-second periods.

4.2 The Overall Process and the Main Timed Simulation

The first of the four kernel functions to be called in our NVIDIA-CUDA program is *SetupFlight*. Right after we declare and allocate the memories for the host and device variables and copy the needed variables to the device, we pass the drone data structure members that need to be initialized to *SetupFlight* function, and all threads initialize an aircraft simultaneously. We copy back the initialized drone data and then we pass it to the *GenerateRadarData* function to set up the initial radar data based on the initial drone data. Using initial position of an aircraft as its expected position, the radar position is obtained by adding noise to this position. Next, the variables are copied back to the host where we split the array into fourths and reverse the pairs in each fourth. This is done to jumble up the value pairs so that the tracking and correlation function can be simulated like a real-life situation where the radars are not in the same order as the aircraft.

We now get to the main part of the NVIDIA-CUDA program, which involves an 8 second cycle that simulates aircraft flying in a 256 nautical mile by 256 nautical mile (nm) airfield [1, 12, 13]. These 8 second cycles can be repeated periodically using an infinite for-loop or a loop that is repeated a specific number of times. The 8 second cycle is divided into 16 half-second intervals and specific air traffic control tasks are scheduled to be executed each half-second. Since this simulation is a real-time system, each of the tasks scheduled each half-second must be completed before the end of that half-second interval. This 16 half-second cycle is called the major cycle.

Task 1 is initiated by a call to *TrackDrone* during the initial part of each half-second period. Prior to each *TrackDrone* in each of the half-second intervals, a call is made to *GenerateRadarData* to create the radar readings of the aircraft that is used in **Task 1**. **Task 2** and **Task 3** are done once at the end of every 16 half-second periods following the execution of **Task 1** using a call to the *CheckCollisionPath* function. Technically, the call to *GenerateRadarData* prior to calling *TrackDrone* is not a part of ATM but is needed in the simulation of this airfield to create the radar sighting that would come from an outside source in ATM. Since the creation of this radar is not part of ATM, this activity can occur prior to the start of each half-second time interval. After the last task scheduled each half-second period, we also check how much time is left in that period. Whatever time is left, we wait that long before executing the next period. This ensures that the tasks scheduled for the next period do not start ahead of schedule. It also ensures that any activities schedule for the next iteration of the loop do not start ahead of schedule. We also keep a count of any deadline that is missed in any half-second.

5 CUDA IMPLEMENTATION DETAILS FOR ATM

We now discuss **Tasks 1-3** in detail and explain the algorithms and data structures used in our NVIDIA-CUDA implementation. The solution is implemented on a GeForce 9800 GT, a GTX 880M, and a Titan X (Pascal) which means it is compatible on both old and new NVIDIA architectures. There is a difference in execution time but

the code is the same. The program uses global memory and is not restricted by shared memory size, which is what makes it compatible on the old and new architecture. A data structure is stored in global memory, called *drone*, that stores the flight record information for all aircraft in the program. It store each aircraft's x and y position; the velocities dx and dy in the x and y direction each half-second; $batx$ and $baty$, the x and y values stored during the conflict detection algorithm that store the new trial path for the aircraft; alt, the altitude; col, weather a collision is anticipated; $time\_till$, the time until the next collision; $colWidth$, the id of the aircraft that this aircraft is colliding with; and $rMatchWith$, the radar that this aircraft has matched with. We use the conflict detection algorithm of Batcher [13]; the pseudocode is given in Algorithm 2.

5.1 Task 1: Radar Correlation and Tracking

In this simulation, **Task 1** takes a shuffled list of generated radar data, as described in Section 4, and compares them against each aircraft to see which aircraft correlates with which radar, if any. The pseudocode for **Task 1** is given in Algorithm 1. Since our application runs a simulation of these ATM tasks, we have to create radar data that simulates real radar data that, in an actual application, would come into the application from an outside source. Radar data, like aircraft records, is stored in a struct in global memory [12, 13].

The first set of radar data is created using the values initialized by *SetupFlight*. This radar data, along with the initialized aircraft data, are passed to *TrackDrone*. This function computes the "expected location" of the aircrafts using their dx and dy values per half-second intervals and then uses the newly generated radar data to correlate them to the "expected locations" of the aircrafts. The goal is to have each aircraft's expected position correlate with one radar, and that aircraft then gets the position of the radar as its own actual position in the airfield. Each thread handles one aircraft each, using the index of the thread (with respect to the grid) as the index for the aircrafts, and initializes data for each aircraft and radar at the index of the thread's id. Each aircraft's expected position is calculated, the $rMatch$ for each aircraft is initialized to 0, and the $rMatchedWith$ for each radar is initialized to -1. Each thread then switches to handling one radar point each by taking the radar data at the same index of the thread and using it when looping through all the aircrafts to see if any aircraft will match this radar.

To be more specific, if i represents the index of the thread and p is the index of the for loop, each thread will have a $radar[i]$ that will be compared against all $drone[p]$. To check if the expected location of an aircraft and a radar correlate, we check to see if the radar is inside the 1×1 nautical mile bounding box around the aircraft. We do this by checking if $aircraft.x - 0.5 < radar.x < aircraft.x + 0.5$, and also if $aircraft.y - 0.5 < radar.y < aircraft.y + 0.5$. Both of these conditions need to be met for correlation to be possible. If a radar correlates with the expected location of an aircraft, $rMatch \leftarrow 1$ ($rMatch = 0$ by default). The $rMatchWith$ variable in the radar struct is then checked to see if it holds the id of another aircraft it has matched with before, or if it has its initial value of -1. If the radar has not been matched before, then the $rMatchWith = -1$, so we set it to the id of the aircraft we matched with. Otherwise, if the $rMatchWith$ variable is the id of another aircraft, i.e, $rMatchWith \neq -1$, then any previously matched aircrafts

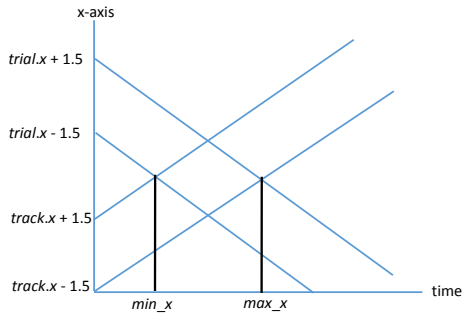

Figure 3: An illustration of the *time-x* track graph

are unmatched from this radar and $rMatchWith$ variable is set to -2, indicating that this radar has now been discarded. If, on the other hand, multiple radars correlate to the same aircraft, then that aircraft's $rMatch$ is set to -1, indicating that the aircraft is no longer being considered for correlation and will keep its expected location as its x and y values. If, by the end of the first loop we are not able to correlate all the radar points with their respective aircraft expected positions, i.e., we still have radars with their $rMatchWith = -1$, we increase the bounding square around the aircraft by doubling it and looping all the remaining, unmatched aircraft's expected positions against the unmatched radars. If after this loop, we are not able to correlate the remaining radars with $rMatchWith$ equaling -1, we again double the bounding square around each aircraft expected position and loop all the remaining, unmatched aircraft expected positions against each unmatched radar. No more loops are done after this, and all remaining unmatched radars are left unmatched, and aircrafts that have not correlated with a radar will keep their expected positions as their x and y positions [12, 13].

Once each radar has been matched with an aircraft's expected position, or once the aircrafts have been checked two extra times with larger bounding boxes, its time to check which aircrafts get their locations updated. We have each thread handle one radar again, and this time we check the radar's $rMatchWith$ variable to see if we have an id of a matched aircraft stored, i.e., $rMatchWith \neq -1$ (unmatched) and $rMatchWith \neq -2$ (discarded). If we find that $rMatchWith$ has a valid aircraft id, we check if that aircraft's $rMatch$ value is equal to 1, meaning that it has matched with only 1 drone and has not been discarded or left unmatched. If we satisfy both those conditions, we then assign the radar's rx and ry values to the aircraft's x and y values, indicating that the aircraft is now at its actual location and not at its expected position. If we do not satisfy one or both of these conditions, then the aircraft keeps its x and y position until the next half-second period where we will try to correlate it with a radar again [12, 13].

5.2 Task 2: Collision Detection

This task checks if an aircraft is on a collision path within the next 20 minutes using an algorithm due to Kenneth E. Batcher. The pseudocode for **Task 2** is given in Algorithm 2. Each thread handles one aircraft by indexing the aircrafts using the id of the thread and uses a for-loop to iterate over the entire aircraft array to compare itself to all the other aircrafts in the global memory. As shown on the *time-x* graph (Fig. 3), an error band about each aircraft is created

by both adding and subtracting 1.5 nautical miles to each aircraft's location (t, x). A similar error band is created on the *time-y* graph. For each aircraft, we initialize *colWith*, the id of the aircraft we are colliding with in the future, to -1. We also set an initial *time_till* variable to the value of 300. This variable determines when the next collision will occur for the specific aircraft, and 300 is considered a safe number for a future potential future crash time. As seen in the above Fig. 3, Batcher's algorithm determines if two aircrafts are on a collision course. For our simulation, the "track" aircraft is the one each thread is handling, indexed with the thread's id, and the trial is each aircraft we compare against when looping against all other aircrafts in the array. We project the x and y values 20 minutes ahead using their respective dx and dy values.

$$min\_x = \frac{|trial.x - track.x| - 3}{|trial.\Delta x - track.\Delta x|} \quad (1)$$

$$max\_x = \frac{|trial.x - track.x| + 3}{|trial.\Delta x - track.\Delta x|} \quad (2)$$

$$min\_y = \frac{|trial.y - track.y| - 3}{|trial.\Delta y - track.\Delta y|} \quad (3)$$

$$max\_y = \frac{|trial.y - track.y| + 3}{|trial.\Delta y - track.\Delta y|} \quad (4)$$

$$time\_min = \max\{min\_x, min\_y\} \quad (5)$$

$$time\_max = \min\{max\_x, max\_y\} \quad (6)$$

We then use Equations 1–6 to determine some key values that will let us know if the aircrafts are on a collision course. The constant value 3 being added and subtracted in different formulas is the total bounding box that we are using for each aircraft. Having a value of 3 in Equations 1–4 means that the bounding area is 1.5 × 1.5 nautical miles, meaning we add 1.5 to x for example for the upper bound, and subtract 1.5 from x for the lower bound. In Equations 1–4 we use the projected locations of the trial and track aircrafts to find the *min_x*, *max_x*, *min_y*, and *max_y* values[1]. We then use Equations 5 and 6 to determine *time_min* and *time_max*. As we can see in the graph, we are trying to calculate *min_x* and *max_x* values, which in our code translate to the *time_min* and *time_max* variables, respectively. When we finally have the *time_min* and *time_max* values, we know that we are on a collision path if $time\_min < time\_max$. We then check if the collision is within a critical time frame, as collisions that are further away can be resolved over time by the planes turning and moving naturally on their paths. A critical time frame is determined to be anything less than 300 when using Batcher's algorithm to detect collisions. So our next step is to check whether $time\_min < time\_till$, which was initialized at 300. When that happens, we set the $time\_till \leftarrow time\_min$ for both aircrafts and update *colWith* with the appropriate id's for each aircraft [12, 13].

5.3 Task 3: Collision Resolution

The pseudocode for **Task 3** is given in Algorithm 2. If an aircraft is on a collision path, we check the *time_min* value and compare it with our default *time_till* value, initialized at 300. If $time\_min < time\_till$, then the time until the next collision is critical and the aircraft's path needs to be altered. We rotate the aircraft by an angle of 5 degrees and save the updated x and y values in new $batx$ and $baty$ variables indicating that these values are determined from the Batcher's algorithm process, and perform **Task 2** again using these updated $batx$ and $baty$ values. After rotating, we reset the loop by setting $t = 19$ so that it can increment to 20 and be at the start of the loop again, and finally break out of the loop to start over again for us to use these new values for the aircraft to start checking against all other aircrafts from the beginning again. If the aircraft is still on a collision course, then we rotate the aircraft 5 degrees in the opposite direction and perform **Task 2** on those new $batx$ and $baty$ values again. We continue to alter the paths in each direction and incrementing the angle by 5 degrees each time, to a maximum of 30, if we keep having collision courses where the $time\_min < time\_till$. Eventually, we will get on a path that is acceptable and without any upcoming conflicts. We can assume that the far away conflicts will resolve naturally or we can wait for them to become critical before we try to change directions and solve those collision courses.

Collisions will be more inevitable with more aircrafts on such a small field, and sometimes the path could fix itself based on the movement of the plane to collide with. Theoretically, complete collision avoidance is not possible in some situations, but our implementation does a good job of avoiding and resolving as much as

Algorithm 1: Tracking and Correlation

1 Radar data is generated on device, copied to host and shuffled, and then copied to device on global memory;
2 Thread i calculates "expected position" for aircraft with id i;
3 **For** $p = 0$ to N (number of aircrafts) **do**
4 Thread i uses radar of id i with bounding box of 1×1 nautical mile and checks if aircraft p is within bounds;
5 **If** *there is an intersection* **then** check aircraft's $rMatch[p]$ to make sure it's 0 (no radars correlated with this aircraft yet);
6 **If** $rMatch == 0$ **then** $rMatch \leftarrow 1$ (radar i and aircraft p are now correlated);
7 Record id of aircraft p in radar's $rMatchWith[i]$;
8 **If** $rMatch[p] == 1$ (previously correlated with radar) **then** $rMatch[p] \leftarrow -1$ and drop correlation with radar;
9 **If** $rMatchWith[i] \neq -1$ (other aircrafts matched with radar) **then** $rMatchWith[i] \leftarrow -2$ and drop radar;
10 **If** *some radars have* $rMatchWith[i] == -1$ (no aircraft correlation) **then** double the bounding box and repeat **Lines 3–9** for those radars and unmatched planes (with $rMatch == 0$);
11 **If** *some radars still have* $rMatchWith[i] == -1$ **then** double again the bounding box and repeat **Lines 3–9** for those radars and unmatched planes (with $rMatch == 0$);
12 Correctly correlated aircrafts have their "expected position" x and y change to "actual location" which is the radar rx and ry while uncorrelated aircrafts keep their expected locations as their x and y;

[1] In Equations 1–4, $trial.x$ and $trial.y$ are the x and y coordinates of the plane that is checking against for possible collisions whereas $track.x$ and $track.y$ are the x and y coordinates of the plane that we are tracking; Δx and Δy are the velocities of those planes in x- and y-axis, respectively.

Algorithm 2: Collision Detection and Resolution

1 Thread i takes the aircraft of id i to iterate it against all aircraft p;
2 **For** $p = 0$ to N (number of aircraft) **do in parallel**
3 **If** *an aircraft p is not the same id as the thread's aircraft i and they are both within 1000 feet of each other* **then**
4 Project both aircraft position 20 minutes ahead;
5 Calculate min_x, max_x, min_y and max_y valuea using Equations 1–4 where $trial$ refers to aircraft p being iterated against and $track$ is aircraft i that each thread holds (-3 and +3 on the equations indicate 1.5 nm boundary being added to both aircraft on x and y positions to create a 3×3 nm bounding box);
6 Find largest minimum time $time\_min$ and smallest maximum time $time\_max$ for the x and y dimensions using Equations 5 and 6;
7 **If** $time\_min < time\_max$ **then** the aircraft i is on a collision course with aircraft p;
8 **If** $time\_min < time\_till$ *of the track aircraft i* **then** $time\_till \leftarrow time\_min$ and the collision is considered to be happening soon;
9 Increment the chk variable, indicating that a course correction is being made and change the col variable for both trial and track aircrafts to be 1, as well as the $colWith$ of the $trial$ and $track$ to be each other's id's to indicate that they are both colliding with each other;
10 Rotate the track aircraft's x and y by 5 degrees each time this task is called, alternating between positive and negative, and increasing up to 30 degrees on each side;
11 Repeat **Lines 2–7** to do the collision detection task again with the new track path;
12 **If** *all aircraft p with the thread i's aircraft i are checked and* $chk > 0$ (indicating that we have attempted to change course) **then** give the aircraft x and y the new path x and y (that is collision-free) and reset the collision variables to not show collision for this aircraft;

possible and it works well with a reasonable amount of aircrafts. In practice, collisions are rare and any left unresolved after the collision detection and resolution algorithm that were urgent would be avoided by changing the altitude of the aircrafts [12, 13].

6 EXPERIMENTAL RESULTS

We describe here the results of comparing the performance of three NVIDIA devices with a SIMD, AP, and Multi-core device running **Tasks 1–3**. We look at the difference in performance based on the graph curves and the total execution time for each task based on the number of aircrafts for that specific test. The NVIDIA-CUDA results are interesting and they show that NVIDIA devices are able to perform the tasks in less time than the AP (STARAN), the ClearSpeed emulation of the AP, and the multi-core (Intel Xeon 16 cores) while also never missing deadlines. The timing results also show us that NVIDIA-CUDA is able to achieve almost linear polynomial curves (see Figs. 4-9).

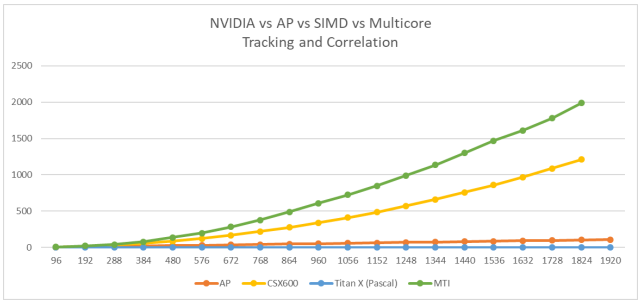

Figure 4: Comparing Task 1 timings in all platforms

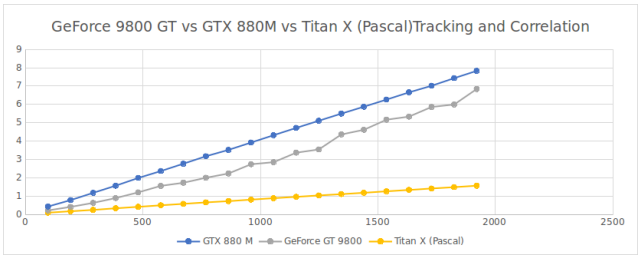

Figure 5: Comparing Task 1 timings in all NVIDIA cards

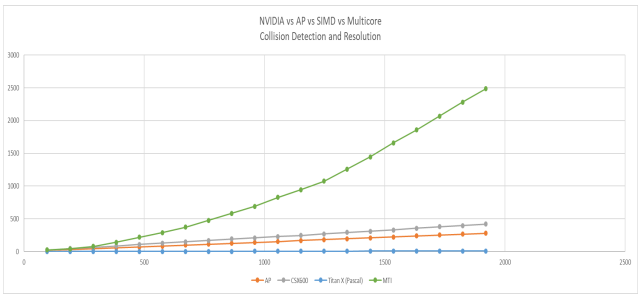

Figure 6: Comparing Tasks 2 and 3 timings in all platforms

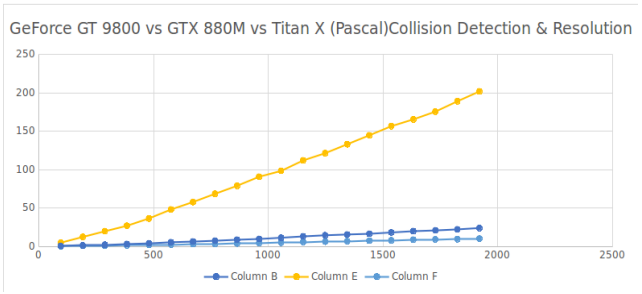

Figure 7: Comparing Tasks 2 and 3 timings in all NVIDIA cards

6.1 Experimental Setup

The CUDA solution is set up to perform **Tasks 1–3** on an arbitrary number of aircrafts just like the AP solution. This program is the implementation of ATM tasks on an AP built specifically to perform

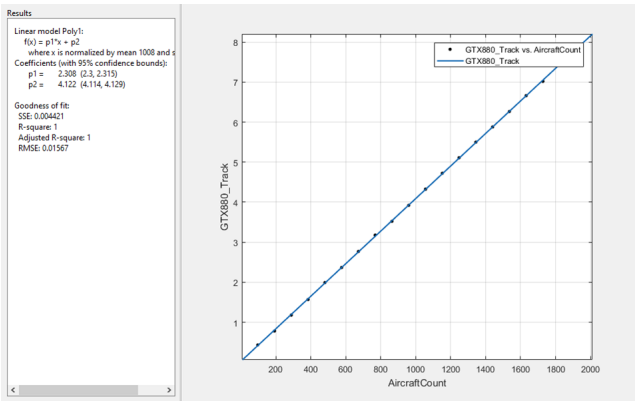

Figure 8: Near linear Curve for Task 1 timings on the GTX880M card

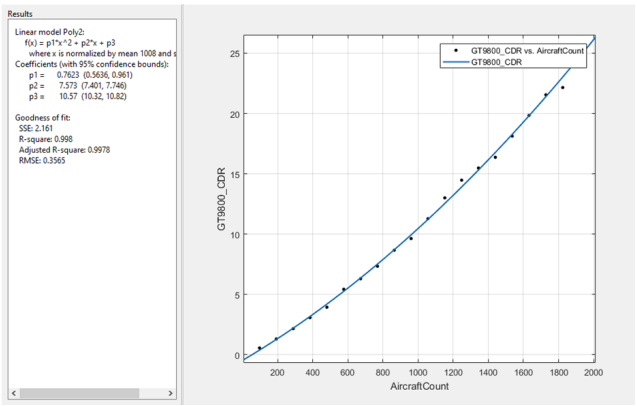

Figure 9: Quadratic (low coefficient) curve for Tasks 2 and 3 timings on GT9800 NVIDIA card

ATM tasks on the STARAN machine. The number of aircrafts also dictates the block/thread setup in the CUDA solution. If there are 96 aircrafts, then the setup used here is 1 block and 96 threads in that block. For more aircraft, the limit on threads per block remains 96 but the blocks increase as the number of aircrafts increases. The tasks are each individually timed and their timings are taken as an average of all iterations of the task. We performed the CUDA tests and got their timings on three devices, the GTX880M, the GeForce 9800 GT, and the Titan X (Pascal). The GTX880M is a card on a personal laptop, with the compute capacity of 3.0, while the 9800 GT is the main CUDA research card in a Linux server with the compute capacity 1.0. The Titan X (Pascal) is a card with the compute capacity 6.1 and a current CUDA architecture on it, Pascal.

6.2 Results and Observations

The timing results that we acquired by implementing **Tasks 1–3** on each of our NVIDIA-CUDA devices show that these NVIDIA-CUDA executions are obtaining SIMD-like timing. That is the curve for its timing results were linear or very close, like quadratic with a very small quadratic coefficient. With the curve fitting software discussed earlier, we were able to find a fit on each NVIDIA device that was linear, with the others being quadratic that are close to being linear due to a very small coefficient for the quadratic term. We were able to optimize the program to make sure that no deadlines were missed while experimenting with the code. Many previous sets of results were collected and evaluated before the final ones reported were collected, due to continuous optimization of the program based on those timing results.

First, looking at each of the different graphs that show timing comparisons between the devices, we see that the NVIDIA-CUDA devices never miss a deadline, nor do they come close to it. All three NVIDIA-CUDA devices are running these tasks much faster than all the AP (STARAN), ClearSpeed (emulation of STARAN AP on CSX600 ClearSpeed), and Xeon implementations. We were also able to determine that the runtime is deterministic for NVIDIA-CUDA as each time we ran the program on any of the three machines, we would get the exact same timings again and again for each machine respectively. There are sometimes small, almost negligible, variations when a task has to do extra work due to, for example, more collision courses being discovered that that require the path of an aircraft be altered during that iteration. The timings are predictable because of this deterministic nature, and therefore allow for scheduling and give us confidence that we will never miss a deadline under certain, pre-programmed conditions such as the maximum number of aircrafts.

If we do get to a point where one of the machines starts missing deadlines consistently, we can count on the deterministic nature of NVIDIA-CUDA to to be able to reschedule in a way that avoids missing those deadlines. The graphs also show us how the NVIDIA-CUDA timing results have a linear looking curve, much like the linear AP timings, but with a lower slope. It's also very apparent from the way these results look that the NVIDIA-CUDA timings do not show signs of having even a quadratic curve that approximates the NVIDIA-CUDA curve unless the quadratic coefficient is extremely slow. We cannot tell what the nature of the extended curves are from just looking at these graphs over a restricted domain, but they show us that the NVIDIA-CUDA timings are much faster and have the potential to be comparable to the AP (STARAN) in terms of efficiency when running ATM.

We further examined the nature of the curves of our timing results by using the program MATLAB [3] that has software for curve-fitting that shows detailed results with four values that indicate a curve's "goodness of fit". When we observe these timing results using the MATLAB curve fitting software, we can see that a lot of these have a linear or almost linear curve, while others have a more quadratic curve. The curve for the GeForce 9800 GT's performance (Fig. 9) with collision detection and resolution shows a curve that seems to fit quadratic better than linear based on the "goodness of fit" numbers. However, the quadratic coefficient is very small compared to the linear coefficient, which means that this curve is closer to linear than quadratic. The GTX 880M has a linear curve for its tracking and correlation timings (Fig. 8) as shown by its "goodness of fit" values. We also see similar, linear curves on the Titan X (Pascal) tracking and correlation timing results. Every one of the three devices was able to produce a linear or near linear fit. Similar results were observed for all tasks in the experiments. The runtime will differ according to the bandwidth and other hardware

specifications, but as we can see here with these results, they will still produce similar curves for their results.

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this paper, we were also able to show that NVIDIA accelerators can handle the most time-consuming of the ATM tasks without ever missing a deadline or starting too soon on major or minor cycles outperforming other three SIMD, AP, and multi-core architectures. Multiple computations using the same NVIDIA-CUDA architecture have repeatedly shown that constant time tasks require a fixed amount of computation time. This demonstrates that this architecture is deterministic. For ATM, having a deterministic architecture means that we know exactly how long the running time will be for each major cycle in the program when given a fixed number of aircraft to work with. Additionally, given a maximum number of aircraft to monitor and control and a deterministic architecture with sufficient throughput capacity, we can create ATM software that under the assumed conditions will meet every deadline. When using a deterministic architecture, there may be multiple situations that can occur which require additional computation, such as having more potential collisions. These situations can be handled by allowing a little additional computation. From our experimentation, we have found that the variation in time needed to handle various special situations to be no larger than 5 times the usual amount of time required, but these situations seldom occur. The graphs we obtained for our execution time appeared to be linear or near linear. Upon closer inspection using curve-fitting tools, we can see that we have almost linear curves for our NVIDIA-CUDA timings for the number of aircraft tested.

Lastly, this work shows that deterministic architectures are able to support predictable real-time systems. By using a deterministic architecture with an adequate throughput capacity, a system designer can guarantee that under normal conditions that every deadline will be met. Real-time professionals were able to make adjustments to the operating system of sequential systems so that all deadlines could be assured of being met under normal conditions. With parallel computers (which were always MIMD), the running time of code was typically unpredictable, preventing this from being possible. Getting them to at least consider SIMDs and other deterministic parallel architectures has been extremely difficult since MIMDs were the dominant architectures.

7.2 Future Work

We would like to build upon this work in the future. For example, we would like to implement all basic ATM tasks and create a more complete ATM system that can be tested on NVIDIA-CUDA machines to determine if it is still viable and will not miss deadlines or change the curves of the execution graph significantly.

It is unfair to compare the performance of the various performances of these architectures based on their running time. The clock cycle times and the size of these different systems vary widely. For example, the ClearSpeed system has only 96 processors, which is a small fraction of the number of threads that are available on most NVIDIA devices. The number of CUDA cores and other CUDA specifications such as the number of Streaming Multiprocessors for each of our CUDA devices is a factor when it comes to evaluating the performance of our application on the various NVIDIA devices, so that is another thing we need to take into account when comparing the efficiency and throughput of these systems. One possible way of trying to compare the performances of these systems more fairly would be to obtain or determine the maximum throughput capacity (or a good estimate of it) of as many of these systems as possible. This information can be used to normalize the graphs of the various systems, with the execution results of each system being normalized to have the same throughput capacity. These graphs should be useful in comparing the efficiency of the ATM computations of these various systems.

Recently, there is a renewed interest in exploring SIMDization through increasingly wide vector units on commodity processors and accelerators (such as Intel's Xeon Phi) [8, 9]. We would like to build up on this work and implement the basic ATM tasks (and others) studied in this paper in these commodity processors that provide efficient, vector-based parallel computation.

A longer term future research focus is to expanding the implementation of basic ATM tasks on NVIDIA systems to being able to provide ATM for small groups of aircraft. This work could be used to provide a mobile ATM center in remote areas where sufficient number of UASs or drones were being used. An alternate extension of this work could be to develop the ability to control a group of drones (called swarms) working on application in remote areas. This work would also be of interest to the USAF, since they are in the process of replacing fighter planes with swarms of drones.

ACKNOWLEDGEMENT

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

REFERENCES

[1] Johnnie W. Baker. 2018. www.cs.kent.edu/~jbaker/PRTS-F14
[2] Giorgio C. Buttazzo. 2011. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* (3rd ed.). Springer.
[3] MathWorks. 2018. https://www.mathworks.com/help/curvefit/evaluating-goodness-of-fit.html?requestedDomain=true
[4] M. Nolan. 2010. *Fundamentals of Air Traffic Control.* Cengage Learning.
[5] NVIDIA. 2018. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
[6] Jerry L. Potter. 1991. *Associative Computing: A Programming Paradigm for Massively Parallel Computers.* Perseus Publishing.
[7] Jerry L. Potter, Johnnie W. Baker, Stephen L. Scott, Arvind K. Bansal, Chokchai Leangsuksun, and Chandra R. Asthagiri. 1994. ASC: An Associative-Computing Paradigm. *IEEE Computer* 27, 11 (1994), 19–25.
[8] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient Execution of Recursive Programs on Commodity Vector Hardware. In *PLDI*. 509–520.
[9] Bin Ren, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2017. Exploiting Vector and Multicore Parallelism for Recursive, Data- and Task-Parallel Programs. In *PPoPP*. 117–130.
[10] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. 1995. Implications of Classical Scheduling Results for Real-Time Systems. *Computer* 28, 6 (June 1995), 16–25. https://doi.org/10.1109/2.386982
[11] Elizabeth Thompson, Nathan Clem, David A. Peter, John Bryan, Barry I. Peterson, and Dave Holbrook. 2015. Parallel Cuda Implementation of Conflict Detection for Application to Airspace Deconfliction. *J. Supercomput.* 71, 10 (2015), 3787–3810.
[12] M. Yuan. 2012. *A SIMD Approach to Large-scale Real-time System Air Traffic Control Using Associative Processor and Consequences for Parallel Computing.* Kent State University. https://books.google.com/books?id=3WxSAQAACAAJ
[13] Man (Mike) Yuan, Johnnie W. Baker, and Will C. Meilander. 2013. Comparisons of Air Traffic Control Implementations on an Associative Processor with a MIMD and Consequences for Parallel Computing. *J. Parallel Distrib. Comput.* 73, 2 (2013), 256–272.