SHAKER, ALFRED, AUGUST 2017 COMPUTER SCIENCE

COMPARISON OF THE PERFORMANCE OF NVIDIA ACCELERATORS WITH SIMD AND ASSOCIATIVE PROCESSORS ON REAL-TIME APPLICATIONS

Thesis Advisor: Dr. Johnnie W. Baker

Basic tasks for Air Traffic Control will be implemented using NVIDIA's CUDA language on a NVIDIA device and compared to the performance of an Associative SIMD processor doing the same tasks. To do this, we create a simulation of an airfield with constantly moving aircrafts. The 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 Air Traffic Control tasks, so they will give us a good measure of the capabilities of the NVIDIA device. The first task is tracking and correlation of the aircrafts in a 256 nautical mile by 256 nautical mile bounding area on a 2D plane with varying altitudes. This task is executed once each half second during each 8 second major cycle period and uses radar to correlate the exact location of the aircraft and its flight records. During every 8 second cycle, Batcher's algorithm is used to check if any aircraft's projected path has a possibility for collision. If a potential collision is possible within the next 20 minutes, we first locate a collision free path for one of them and then have it switch to this path.

In previous research, the ability of a multicore system to perform basic ATC tasks was investigated. The graph showing its performance increased rapidly as the number of aircraft increased, which is consistent with the general belief that all large real-time systems require exponential time. In contrast, in our earlier research, an associative SIMD system was shown to be able to execute these basic tasks in linear time with a graph that had a very small slope. Additionally, the multicore regularly missed a large number of deadlines while the SIMD system

did not miss a single deadline. Our goal here was to determine whether we could get SIMD-like results using a CUDA implementation of the same real-time system involving basic ATC tasks on a NVIDIA accelerator. Our research shows that our NVIDIA accelerators can provide a SIMD-like implementation of this real-time system. Moreover, using curve-fitting with MATLAB, the graph showing the NVIDIA accelerators performance increases only slightly faster than a linear graph.

COMPARISON OF THE PERFORMANCE OF NVIDIA ACCELERATORS WITH SIMD AND ASSOCIATIVE PROCESSORS ON REAL-TIME APPLICATIONS

A thesis submitted

To Kent State University in partial

Fulfillment of the requirements for the

Degree of Master of Science

by

Alfred Shaker

August, 2017

Thesis written by

Alfred M. Shaker

B.S., Kent State University, 2015

M.S., Kent State University, 2017

Approved by

Dr. Johnnie W. Baker, Advisor

Javed I. Khan, Chair, Department of Computer Science

James L. Blank, Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to sincerely thank Dr. Johnnie W. Baker for providing me with this opportunity and motivating me throughout the process of this research. He is my advisor for this research throughout my time as a graduate student here at Kent State University. The wealth of information at his possession and his many years of experience were a great asset to me throughout the thesis preparation and research work.

I would also like to thank all the people who have supported me in any way during my research, which include Dr. Gokarna Sharma and Dr. Ye Zhao who were also on my committee and I appreciate them taking the time for it, Marcy Curtiss, Azim Shaik, Man (Mike) Yuan, and many others.

I would not be where I am today without my family, so I would like to give a huge “thank you” to my parents Amal Maxaimous and Mikhail Fayez Shaker, and my amazing supporting sisters, Alice and Alexandra. I would not be where I am without their love and support and motivation that they have given me throughout my education.

CHAPTER 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 [1]. 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 Control [2]. ATC is a real-time application that needs to continuously monitor the behavior of the aircrafts while performing various calculations and time-consuming tasks on possibly thousands of aircrafts. Some of these tasks occur during the same half-second interval and the software has to be optimized in a way that ensures all the tasks finish before their deadline. In real-time applications, missing 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 ATC. These deadlines are known 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. The major cycle consists of sixteen of these one-half second intervals and major cycle is repeated infinitely [2]. 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 they do not let the shortcomings of their architecture hinder their performance by being creative and thinking outside the box.

Solutions to this problem have been implemented in a few different ways in the past at Kent State University[1]. ATC has been implemented on multicore systems where aircraft data was stored in memory that all processors in the system could access. This was not very efficient as the nature of dynamic ATC systems proved too complex and difficult for this type of system. In this thesis, we will call a system predictable or deterministic if the time it requires to perform a constant time computation is always the same. Due to being asynchronous, MIMD 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 computational intensive applications like ATC, they typically miss a large number of deadlines. Since 1963, there have been four major attempts to replace the national ATC system that had been developed over the years with one that would meet what were considered to be important required specifications, but after about 10 years, each was abandoned and a new effort at building an ATC system that would meet new and updated specifications was started. The latest attempt at building a new ATC system is NextGen (for Next Generation), which was started around 2003. However, after it failed to meet specifications, the decision was made to implement it anyway in stages between 2012 and 2025. The NextGen system will use GPS to determine the location of aircraft, Current ATC systems are unable to processes most of the radar in real-time and instead use a method called a "secondary radar" which involves having a radar signal hitting an aircraft to prompt the aircraft's transponder to broadcast requested information such as the aircraft's ID and its altitude. This normally eliminates the task of matching the ID of the aircraft with its radar. However, transponders can be turned off or disabled and this normally occurs on aircraft that are engaged in illegal activities. Pilots of aircraft involved in illegal activities currently normally operate in altitudes below 500 feet to avoid as much radar as possible.

However, even pilots skilled at avoiding radar are likely to have their aircraft's position identified by some radar. Also, radar records can be used to study the paths taken by aircraft that crashed, disappeared, or engaged in illegal or questionable activities. It is currently unclear whether totally eliminating radar from the ATC system, as proposed in the NextGen plan, is a desirable option. [1]

Dr. Kenneth Batcher was the chief architect at Goodyear Aerospace for an enchanced SIMD computer called an associative processor or AP during the early 1970's that was explicitly designed for the ATC type applications. This AP included hardware that allow some very useful ATC capabilities such as broadcasting, associative searches, and maximum and minimum reductions to be executed in constant time, A particularly important feature of SIMD architecture is that it is a synchronous system and is deterministic. As shown in [1], this architecture was able to execute even the most intensive ATC tasks in linear time. ATC software was developed for this system handled the basic ATC tasks (e.g., automatic trackimg, conflict detection and resolution, terrain avoidance, and automatic voice advisory) and consisted of only 4017 parallel lines of "assembly code" and about 1600 likes of sequential code for its control unit. The capability of this system to perform basic air traffic control was demonstrated at the International Air Exposition at Dulles Field in Washington DC in 1972. Based on the analysis in [1], this system would have run in linear time. The ASPRO, another AP designed at Goodyear Aerospace, was a second generation STARAN and was used extensively by the Navy for their Air-borne Air Defense Systems applications for over 10 years. [3]

In contrast, it is generally accepted that all ATC software for MIMD systems runs in exponential time. The past and current FAA ATC software systems are extremely large and the research and development of these systems have involved many of the US's large high tech

companies or research groups like Lockheed Martin, Computer Science Corporation, IBM, Mitre, and NASA. Unfortunately, the development of any future MIMD ATC software which runs in polynomial time is not expected.

In this paper, we will be using basic time-consuming ATC tasks to compare the performance of these tasks on NVIDIA accelerators with their performance on the ClearSpeed SIMD and the STARAN associative processor. The main tasks that we will focus on in this paper are the following three: Tracking and Correlation, Collision Avoidance, Collision Resolution. These three tasks are the most time-consuming ATC tasks and therefore give us the best idea of how well our software is performing on the three NVIDIA devices we will use. The implementation developed here is based on the same algorithms for the basic ATC tasks implemented on the STARAN and ClearSpeed systems, 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. We are implementing this code on a GeForce 9800 GT card, which despite being an old and outdated card with Compute Capacity of 1 and working with really old CUDA architecture, is proving to be quite powerful when the program is written optimally. We are also experimenting on two other NVIDIA-CUDA machines, the GTX 880M on a personal laptop, and the Titan X (Pascal) on another dedicated research machine. These are newer NVIDIA graphics cards, and the Titan X has the latest NVIDIA-CUDA architecture, Pascal. The program in total has around 900 lines of code, which include thorough comments documenting the code throughout so that it is easier to come back to and continue working on in the future.

We will be primarily comparing our NVIDIA/CUDA implementation with Mike Yuan's AP implementation. In his dissertation, Yuan used ClearSpeed CSX600 to emulate an AP and implemented all eight key ATC tasks, but we will just be focusing on his implementation and results for the three tasks previously mentioned that we are also implementing. This implementation was done using the Cn (ClearSpeed) language, which like CUDA, is an extension of the familiar C language. The ClearSpeed accelerator used for the computation was a CSX600 card with two chips, each chip consisting of a SIMD system with 96 processing elements connected together by a ring network.

The results we got are very satisfactory, as no deadlines are missed when the program is implemented and executed on all three NVIDIA-CUDA devices. These results show that the implementation on NVIDIA gives us better timings than the CSX600 and the AP timings. However, this comparisons are not normalized to adjust for different clock rates and throughput capacities and are based strictly on computation times. The NVIDIA solution is scalable, and most important, deterministic, as we can execute the program with the same aircraft count multiple times and get the same timings every time. This means that even when the aircraft count is too high to meet deadlines, we can still predict accurately the margin by which the deadline will be missed each time. A graph that shows the increase in the amount of time it takes to execute the tasks as the number of aircrafts increase appears to be a linear or almost linear graph with a very small slope. The results we get here give us confidence in moving forward with implementing the rest of the ATC tasks and creating a fully functional, scalable and dependable ATC system on NVIDIA devices.

In this thesis, the more important background information for this thesis is given in chapter 2. Chapter 3 talks provides a detailed overview of how the NVIDIA program is

constructed and how it works, while Chapter 4 dives deep into the implementation of the algorithms for the tracking and correlation task as well as the collision detection and resolution tasks. In Chapter 5, the results of our experiments on the NVIDIA-CUDA devices are compared to the results of the AP and ClearSpeed implementations. And finally, in Chapter 6, we will provide a summary of the main contributions of this research and of several possible useful future extensions of it.

CHAPTER 2

Background Information

This chapter discusses some of the concepts and terminologies used in this thesis. We will discuss different concepts that will help readers understand this thesis and refer to other published works where further information can be obtained.

2.1 Flynn's Taxonomy and Classifications

Flynn's taxonomy is a classification system used for parallel computers. It is the most widely used scheme and describes what kind of parallelism a device exhibits according to the number of instruction streams and the number of data streams. We denote instruction streams with an "I" and data streams with a "D". If a stream if a single stream, we use a "S" to denote single streams and use "M" to denote multiple streams. There are four possible categories, namely single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction single data (MISD), and multiple instruction multiple data (MIMD). In this paper, we mainly focus on SIMD, but also talk a little about MIMD. The SISD and MISD computers are not discussed further as they are not are they relevant to this paper. [1, 2, 4]

2.2 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. These executions occur simultaneously but the processors

operate asynchronously. This asynchronous execution makes the MIMD computation much 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). MIMD's 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, 2, 4].

2.3 SIMD

SIMD is the style of design found in most early parallel computers. SIMD stores the programs to be executed in the instruction stream, which is a processor also called the control unit. The IS is connected to multiple processing units called processing elements or PEs that execute the IS instructions synchronously, with each PE executing on data from its own memory. These are the main components of a SIMD computer. Each PE is primarily an arithmetic logical unit, or an ALU, which is responsible for performing arithmetic and logical operations. [1, 2, 4]

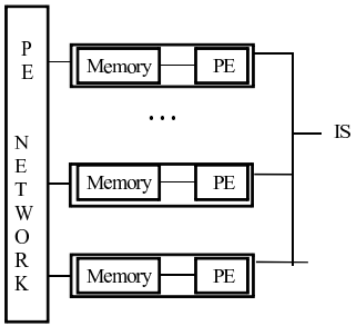

Figure 1: SIMD Model

There are three different types of parallel systems that are sometimes included when discussing SIMD type computers: traditional SIMD discussed above, vector machines, and short SIMDs machines. Traditional SIMD includes the Goodyear Aerospace MPP, the MasPar computers, and Thinking Machines CM_2. 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. Here, we will include only the traditional SIMD computers when we refer to this category. [1, 2, 4]

Most researchers today believe that MIMD computers are more powerful and cost efficient than SIMD computers. However, SIMDs also has some advantages over MIMDs. For one, SIMD's biggest advantage is it's simplicity. Due to there being only one control unit, there is no need for synchronization between machines, which removes a lot of overhead for SIMD machines. SIMD programs are much easier to program and debug. Writing a SIMD program is often very similar to writing a sequential program for the same application. SIMD's synchronous nature also means that the flow control is sequential and it's easy to tell what part of the program

is running at any given time, which makes it easier to debug and optimize. As mentioned earlier, SIMDs have the ability to handle large real-time problems like the real-time ATC problem dramatically simpler and more efficient solution than MIMDs. Here, we will show that the NVIDIA architecture can also provide a similar SIMD-like solution for this problem and that the efficiency of its solutions approximates that of the SIMD solution [1, 2, 4].

2.4 Associative Processor (AP)

An associative processor is an enhanced SIMD architecture that was first used in the STARAN computer, which was built at Goodyear Aerospace during the early 1970's. The chief architect of the STARAN was Dr. Kenneth Batcher. The associative architecture was explicitly designed for the purpose of performing air traffic control. AP has various characteristics that allow it to support ATC much more efficiently than usual SIMD architectures. The SIMD hardware of an associative processor is designed to support constant time operations such as broadcasting, associative searches, maximum/minimum reductions and more. While associative computing was a widely discussed topic much earlier, a formal definition for the associative processing model was first given in [5]. A detailed explanation of these associative properties is given in section 3 of [1]. Additional information can be found in [2, 3, 4].

Goodyear Aerospace built a second generation associative computer called the ASPRO in the late 1970's with 1792 processing elements. Dr. Kenneth Batcher served as an advisor for the design of the ASPRO but was a professor of Computer Science at Kent State during most of the time that the ASPRO was being designed and built. The ASPRO was used extensively by the US Navy primarily in their Northrop Grumman E-2 Hawkeye aircraft for air-borne Air Defense

Systems applications (e.g., aircraft early warning radar surveillance and command and control processing). The ASPRO was still built in 1995 and over 170 systems had already been delivered to the Navy. [5] What distinguishes the STARAN and ASPRO computers from other non-AP SIMD's is they were built so that they could handle the FAA ATC system I/O requirements. They were able to meet these requirements by using a multidimensional access memory (MDA) and a flip network. While much smaller than the omega network, the flip network can simulate the omega network by repeated passes. The flip network can be used as a corner turning network to transfer data in between a slice of memory in a ASPRO/STARAN PE and an outside buffer. The corner turning ability of the flip network and the assignment of one record per processor allows multiple PEs to work together to efficiently transfer a record between one PE and an outside buffer in constant time. The flip network has some additional capabilities as well. Further information with pointers into other references can be found in [1].

2.5 Real-Time Applications

The primary reference for this section is [6]. In this section, we will introduce some real-time definitions that will be useful in understanding the real-time terminology used in this thesis. In order to preserve the precision of these definitions, the statement of most of these definitions are either very similar or else identical to the definition used in this reference textbook. Unfortunately, restating precise definitions "in your own words" tends to introduce a lot of imprecision into these definitions. As a result, the material in this section closely follows the presentation in [6] for the topics covered here.

The correctness of a real-time system depends not only on the results produced but also on the time in which the results are produced. A task is a computation that can be executed by a CPU or thread. At the process level, the main difference between a real-time task and a non-real-time task is that the real-time task has a deadline, which is the maximum time within which it must complete its execution. A deadline is said to be hard if producing the results after its deadline may cause catastrophic consequences on the system under control. A deadline is called firm if producing the result after its deadline is useless to the system, but does not cause any damage. A deadline is called soft if producing the result after its deadline still has some utility for the system, but causes degradation in the performance. The most important property for hard real-time systems is predictability. [6, pg 12-13, 3].

Each execution of a task is called a job. A job is called an instance of a task. Also, jobs are units of work that are scheduled and executed by the system. The arrival time (also called request time or release time) for a task is the time at which a task becomes ready for execution. The computation time is the time required by the processor to execute the task without interruption. The finish time is the time at which a task finishes it execution. The response time is the difference between the finish time and the request time[6, pg 26-27].

An important feature of the real-time tasks is the regularity of their activation. A periodic task consists of an infinite sequence of activations (called jobs) that are regularly activated at a constant rate. If T is a period task and the k^{th} job of a task T is denoted T_k then the activation time for T_k for k>1 is called the activation period of task T. Aperiodic tasks also consist of an infinite sequence of identical jobs (or instances), but their activations do not occus at a constant rate. An aperiodic task where consecutive activations are separated by a minimum inter-arrival time is called a sporadic task. [6, pg 80-82]

The most important property for hard real-time systems is predictability. In safety critical applications, all timings requirements should be guaranteed offline before putting the system into operation. If some tasks cannot be guaranteed within the time constraints, the system must announce this fact in advance, so alternate actions can be planned to handle the exception. The first issue that effects predictability of scheduling is the processor itself. The internal processor features such as instruction prefetch, pipelining, cache memory and direct memory access (or DMA) are a major cause of nondeterminism. While these features improve the processor's average performance. They also introduce non-deterministic features that prevent a precise estimation of worst-case execution times[6, pg 250-254].

2.6 ATC

Air traffic control 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 ATC systems happen 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. The tasks that we focus on in this paper are the three most time consuming and compute intensive tasks: Tracking and Correlation which is executed every half second, Collision Detection which is executed every 8 seconds, and Collision resolution which also is executed every 8 seconds following Collision Detection, when required. The tracking and correlation task 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 Chapters 3 and 4. Due to the nature of this task, the ATC system needs to execute this task once 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 priod. A task cannot execute if the previous task is still being processed. This is another reason we need to make sure that our ATC system meets the required deadlines every period, so that it does not cause the tasks following it to start late and also possibly miss their deadlines. Luckily, these deadlines are known at the time the time the program is created, so for deterministic systems its possible for us to optimize the code so that these deadlines will be met each time they are executed. [1, 2, 4, 7]

CHAPTER 3

CUDA Solution To ATC

This chapter discusses the creation and the execution of the ATC tasks created using CUDA and C. We will discuss the four different kernel functions that setup the flights, generate radar data, handle the tracking and correlation tasks, and the collision detection and resolution tasks throughout the program and when those kernel functions are called. In this chapter, we will also explain the setup and execution of the 8 second simulation period for the aircrafts and how the code manages ensure all deadlines are met and still start executing exactly on time when needed, and not too early or too late. We will also compare our CUDA execution to the AP execution done by Mike Yuan in order to discuss the advantages of each. And finally, we will talk about key CUDA properties and features that make it possible to simulate these key tasks on a parallel system while meeting all deadlines during every iteration.

3.1 Overview of CUDA Solution

3.1.1 Introduction and Key Structures

The program takes advantage of the strengths of the CUDA programming architecture to build the ATC solution. The program is built using the C language with CUDA directives. First thing to note is the drones struct, which is what holds all the necessary information for each aircraft. These are the x and y location positions and their velocities in each of those directions and it's altitude. It has variables that hold information about other aircrafts that are on a potential collision course with this aircraft, and the radar this aircraft matched with in the tracking and correlation task. It also contains the timeTill variable that is used in the collision detection and resolution task and the batx and baty variables that hold the temporary altered x and y values in the same task. The batx and baty variable names are short for Batcher's X and Y variables that are used heavily in the Batcher's conflict detection algorithm. They store the initial X and Y values coming into the kernel function, and are used in calculation of the formulas in Batcher's algorithm, as discussed later in more detail in chapter 4. They also hold the altered X and Y positions when course correction happens so those new values can be tested against the other aircrafts. The other structure that is used is called radar and it holds the generated radar information to use in the tracking and correlation task. That structure has the variables for the radar's x and y positions and the id of the aircraft they match with.

3.1.2 Kernel Functions

The program uses four kernel functions: SetupFlight, GenerateRadarData, TrackDrone, CheckCollisionPath.

The SetupFlight function is called once in the beginning of the program and initializes the aircraft's x and y positions by randomizing them between the values of 0 and 128. After the initial x and y values are calculated randomly, a random number is chosen between 0 and 50 for a temporary variable. If that number is an even number, then the x position is made negative. Another temporary variable is also set to a random number between 0 and 50 and this time if it's odd, the y value becomes negative. This is to set up a virtual airfield of size 128 in the X and Y directions in both the positive and the negative directions. To set up the speed correctly, there are a couple of steps to take. Since we want the speed to be between 30 and 600 nautical miles per hour, we set a speed variable 'S' to a random number between 30 and 600. We also set dx to a random number between 30 and 600. To find a dy value that will match the dx value to get the final velocity S, we use this formula:

S = sqrt[|dx|*|dx| + |dy|*|dy|]

or

dy = sqrt[| S*S -|dx|*|dx| |]

Before we get the square root of the difference between 'S' squared and 'dx' squared, we need to get the absolute value of that number or we run into problems. At this point we have dx and dy as the speed in the x and y direction in nautical miles per hour. To reduce it down to nautical mile per half second iteration, we divide it by 60 minutes x 60 seconds for nautical miles per second, then divide that by 2 to get the speed per half second. After we have dx and dy, we

do the same process with the random temporary variables as with x and y to see if one of them will get randomly set to a negative number. This is so that not all planes are on the same positive slope.

GenerateRadarData takes in the initialized drone values in the beginning of the program and the new x and y values again after every half second iteration to generate new radar data based on that aircraft data. The idea is to simulate radar data that, in a real-life situation, would be received by the program from an outside source, and correlate it with the aircraft's expected location to pinpoint exactly where the aircraft is located, based on it's latest radar reading. There are many reasons that an aircraft's calculated location and its radar location may be different could be due to many reasons such as weather conditions, air traffic controller instructions to the pilot, etc. The path of the aircraft may be temporarily changed to avoid a storm or due to instructions from an air traffic controller to slow down or speed up their arrival at an airport. The wind can slow down or speed up an aircraft or blow it off course. In a real-life situation, this radar data would be in a jumbled-up order and usually would be close to the expected location values for each aircraft, with some randomized noise added. To simulate that, we create a small random number added as "noise" to both the x and y positions of every aircraft to create the radar x and y for that drone. This noise could be a negative or a positive number. One could have a positive number while the other has a negative. Each thread is set up to take one drone and use it to create it's radar data. Once each thread has generate 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 it's 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 it up. 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 is the function that performs the Tracking and Correlation task that is expanded upon in great detail in chapter 4 where we talk about the CUDA implementation on the GT9800 graphics card. The basic idea of this function is that it takes all the current aircraft data and the newly created and jumbled up radar data and tries to match each aircraft's expected position with it's correct radar. It does so by having each thread handle a radar by indexing the radar with the thread's id and iterating all the aircrafts against it. For there to be a correlation between the radars and the aircraft's expected positions, the radar point must fall within the 1 nautical mile by 1 nautical mile square boundary around the aircraft's expected position. If a match is found, the rMatch variable, which is first initialized to 0 for all aircraft, is set from 0 to 1, indicating that the aircraft has matched with just one radar. If multiple radars correlate with the same aircraft, then that aircraft is no longer considered for correlation and will keep it's expected location. If, however, one radar correlates with multiple aircrafts, all aircrafts are unmatched from this radar and the radar is discarded and not used for further correlation checking. If after the first loop through all the drones, there are still radars that have not yet been matched, we have each radar look through the remaining aircrafts that have not been matched up with a radar. This time, however, the aircraft will have a bounding box twice the size of the first one. After this loop if there are still unmatched radars, we double the bounding box again and loop through the

aircrafts to try and find a match. After the third and final loop, unmatched radars are just left unmatched, although the frequency of there even being unmatched radars at this point is very low. Almost 100% of the time, each aircraft's expected position is at least matched with 1 radar, and not very often, unless there is a huge number of aircrafts, is a radar every matched with multiple aircraft's expected positions. This kernel function is called every half second, along with a new set of generated radar data.

CheckCollisionPath is the last kernel function that the program uses and this is the function that performs the collision detection and resolution (or avoidance) task. 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 have them split into different kernel function. This function is also discussed in greater detail in chapter 4 where we discuss the specifics of this implementation. In this function, the thread handling each aircraft uses Batcher's algorithm to check if it is on a collision path by projecting its aircrafts position from 0 to 20 minutes ahead and checking against all other aircraft. An aircraft is considered to be on a collision path if it is within 1000 feet in altitude of the other aircraft and if each aircraft's 3 by 3 bounding boxes intersect. Batcher's algorithm's provides us with the formulas to help us determine whether 2 aircrafts are on a collision course based on their x and y values. These six formulas, described in great detail in chapter 4, are used on the projected location of the aircraft in 20 minutes to calculate some key values based on the x and y positions of the two aircrafts. Using these values we can determine if the projected aircraft is on a collision course with the aircraft it's being compared with and also whether or not the conflict will occur soon. If we do find that we are on a collision course, and the time until collision is critical, we rotate the aircraft 5 degrees and reset the loop to start over and check the new x and y values against all other drones. If we find that

we still are on a collision course that has a critical time until collision, we try rotating 5 degrees in the opposite direction. The angle increased by 5 each consecutive time we need to do collision avoidance on the new x and y values until it reaches a maximum of 30 degrees on each side. An important fact to remember is that the path of the aircraft that is on a collision course remains on its original course until a new path that is collision free is found. If this policy were not followed, the aircraft path might be changed to new path where a collision would occur almost immediately. Usually after checking one or two times, a collision-free path is found. This is because currently our skies are very open. The aircraft with a safer time until collision can be dealt with later if their time until collision becomes critical. Often the potential collision is caused by an aircraft that is turning and disappears as the aircraft continues it's turn. The collision avoidance and resolution function is not called as often as the tracking function, as it is called only once every 8 seconds, this once during the 16 half second iterations in the major ATC cycle.

3.1.3 The Overall Process and the Main Timed Simulation

The first of the four kernel functions to be called 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 kernel function, and all threads initialize an aircraft simultaneously. We copy back the initialized drone data and then we pass it to the GenerateRadar kernel function to set up the initial radar data based on the initial drone data. After we have generated Radar data with the appropriate noise added to them, we copy the variables 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 drones.

We now get to the main part of the program, which involves an 8 second cycle that simulates aircraft flying in a 256 nautical mile by 256 nautical mile (nm) airfield [2]. These 8 second cycles can be repeated periodically using an infinite for-loop or the loop can be repeated a specific number of times for a specific time period. 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 in this real-time system. The tracking and correlation task is initiated by a call to TrackDrone during the initial part of each half-second period. Priod to each TrackDrone in each of the half-second intervals, a call is made to GenerateRadars to create the radar readings of the aircraft that is used in tracking and correlation task. The collision detection and avoidance task is done once at the end of every 16 half-second periods following the execution of the tracking and correlation task using a call to the CheckCollisionPath function. Technically, the call to GenerateRadars prior to calling TrackDrone is not a part of air traffic control but is needed in the simulation of this airfield to create the radar sighting that would come from an outside source in air traffic control. Since the creation of this radar is not part of air traffic control, this activity can occur prior to the start of each half-second time interval. At the end of every loop we also check how much time is left in the half second based on how long the tracking and correlation task and the collision detection and avoidance task took. Whatever time is left, we wait for that long before we execute the next iteration of the loop. This is done to ensure that the activities

scheduled for the next half-second do not start ahead of schedule. We also keep a count of any deadline that is missed in any half-second.

3.2 CUDA Properties and Features

CUDA-NVIDIA can run the ATC tasks efficiently due to the many features of it's memory setup and architecture. The different tiers of memory interact differently and have different access patterns when it comes to the host and the device, making it possible to write code that takes advantage of the high-speed parallelism [8].

3.2.1 Memory Handling

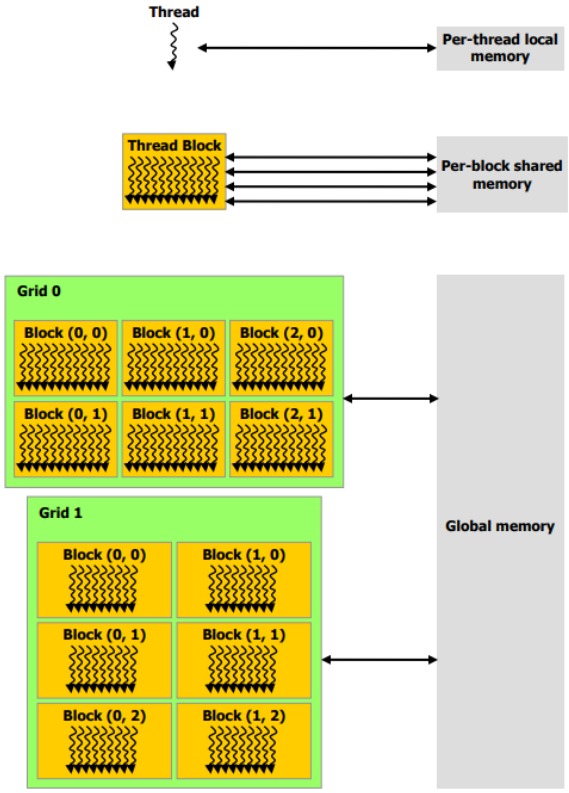

Figure 2: CUDA Memory Handling

The memory in CUDA is set up into a few different segments, each with their own read/write rules. Global memory is accessible by both the host and the device, making it ideal to store data that changes and is persistent throughout the program. This is where we will store most of our data, like the drones and radar data. In the code, we have a struct called drones, that we describe in detail in chapter 4, that has member variables representing different properties of each drone like it's x and y positions and it's velocities. The members of this struct represent the device variables that the kernel functions will use to perform each task. In the host code, we define variables that will hold the host copies of this data, and when we use malloc to allocate memory for it, that gets allocated in the global memory. For the device copies of the variables, we use cudaMalloc to allocate memory for the struct variables. That memory is also allocated in global memory. We now have 2 copies of the struct variables in global memory, one to be used by the host and the other by the CUDA device. If we initialize the drone data in the host we would have to cudaMemcpy to move that over to the device variables, or we can just use our kernel function setupFlight to initialize the drone struct members. If we want to view that data on the host we would have to cudaMemcpy it back from the device to the host. The device variables are passed to the functions as parameters and when they are used in the function, the read/write accesses the global memory and it's speed varies from device to device based on the bandwidth for that device. Luckily, it is fast enough that the overhead from reading and writing is negligible in this case. This memory can also be accessed by every thread in the device, so having multiple blocks access this memory will not be a problem [8].

Shared memory on the other hand, is exclusive to each block, but has no overhead for reading memory and has much higher bandwidth than global memory. That means, threads from one block cannot access the shared memory used by another block. Because our simulation

depends on the use of multiple blocks, we do not make use of shared memory here. If we wanted to use shared memory, we would be confined to a maximum of 1024 threads and be constricted even further due to the small size of shared memory that will limit our use of it. Shared memory is initialized in the kernel function, so the threads in one block will have their own copy of the shared memory variable. Despite the advantage of having shared memory available to handle the aircraft and radars for faster execution, there are a lot of limitations when it comes to using shared memory that we discovered while experimenting with shared memory use in a previous iteration of the code. First, since each block has it's own shared memory, and only threads of that block can read from it, we can do one of two things. Either limit our test size to one block of aircraft, so if the limit on the block is 1024 threads, then that's the limit on the aircrafts we can test. This obviously is not what we want, so the other approach was to have each block grab a segment of the drones and put it into shared memory from global memory and do the tasks on them like collision detection & resolution. But then we run into the issue of having more overhead from transferring memory from global to shared memory, as well as the bigger problem of now not being able to test all drones against all other drones efficiently, as the segmented sets are not compared to other segments in this method. If we did have the segments compare to other segments after they finished computing the task in their own block, that would greatly increase execution time and it will not be efficient at all. After testing and reconstructing the trial solution many times, it appeared best to leave the drones and radar data on the global memory and use multiple blocks with 96 threads each, since we are working with multiples of 96 so there will be none of the threads we assign go unused. This is done because the number of aircrafts that are tested are multiples of 96, and therefore in the code we have 96 aircrafts maximum per block, so the higher the aircraft count goes the more blocks are used, which forces

more SM's to be used which lets us utilize more of the parallelization power of CUDA. Many combinations of block/thread amounts were tested prior to coming to this conclusion based on the running times. [8]

We also can create new variables in the kernel functions, which are unique to each thread and are stored in the registers memory of each thread, which is memory that is thread exclusive and has the fastest memory transfer rates amongst all CUDA memories. This is typically for things like for-loop variables, temp variables on each thread that assist in some calculation, and is nothing that can be copied or transferred outside of the kernel function as the lifetime of this memory is only for that thread's lifetime, which is the duration of the kernel function's execution. [8]

3.2.2 Thread/Block/Grid Parallelism

The way that CUDA works is that it has each thread perform one part of the execution, in this case, each thread works on one aircraft, and they all work simultaneously in groups called warps spread across the streaming multiprocessors on the device. This helps the program run fast and efficient calculations on large data sets without missing deadlines. In the case of the CUDA solution for ATC, we have each thread handle the calculations for one aircraft, and each thread uses it's aircraft as the base when comparing to the other drones/threads. These threads are split into different blocks which allows us to take advantage of more parts of the CUDA device. Being able to compute things this way allows us to have extensive calculations and multiple conditions and loops in each kernel function. [8]

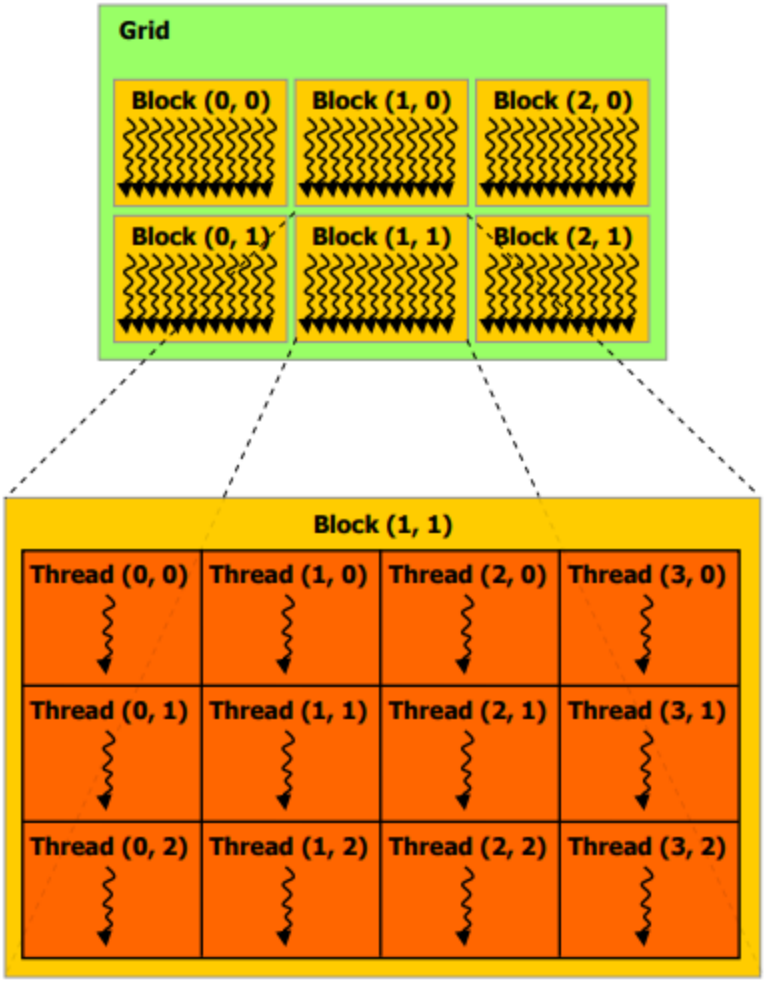

Figure 3: CUDA Thread/Block/Grid architecture

In our code, we index each thread based on the 2d offset, as blocks and threads in our code form a 2d grid that looks like the figure above. To get the right thread index we use the offset and end up with int i = threadIdx.x + blockIdx.x * blockDim.x which will give us the correct index across all the threads in every block. We do this instead of just using threadIdx.x because that represents the thread id of the thread relative to the block, so block 0 will have it's first thread by id 0, and block 1 will also have it's first thread be id 0.

Using this id, we can then have each thread handle the computations for one aircraft by having each thread operate on the i'th element of the aircraft array, so thread 0 will have drone 0,

and thread 1 will have drone 1, and so on. Doing this allows us to use each thread to do as many calculations on that drone as we need to and they would all be repeated for every drone, in parallel. While having each thread acting as a drone, we can also easily iterate against all other drones in the global memory by using a simple for-loop that iterates over the total amount of aircrafts we have, allowing every thread to individually look at every other drone to compare values for tasks like collision detection and resolution. What is interesting is that inversely, we can also have each thread handle the computations for one radar target point by having each thread represent the i'th radar and then iterate against all the aircrafts to try and match up radars to drones. The versatility of this architecture is key in getting a good solution with good timings and allows us to really take advantage of global memory. [8]

The number of active threads that run in parallel is the number of streaming multiprocessor multiplied by the maximum number of concurrent warps, which are 32 threads each. (get actual numbers on both cuda devices). The power that this kind of parallelization gives us is one of the reasons why the CUDA implementation runs so fast compared to the ClearSpeed AP implementation, which only has 96 processors. [8]

CHAPTER 4

CUDA Solution to ATC Implemented on GeForce 9800 GT

This chapter discusses each of the tasks in detail and explains the algorithms and data structures used. These tasks include report correlation and tracking, conflict detection, and conflict resolution. The solution is implemented on a GeForce 9800 GT and a GTX 880M, which means it is compatible on both old and new architecture. 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 information about all the aircrafts in the program. It stores each aircraft's x and y positions; dx and dy, the velocities in the x and y directions per half second, respectively; batx and baty, the x and y values calculated during the Batcher's conflict detection algorithm that store the new trial path for the aircraft; alt, the altitude; col, whether a collision is anticipated; timeTill, time until next collision; colWith, the id of drone this drone is colliding with; and rMatchWith, the radar that this drone has matched with.

Table 1: Aircraft Structure Table.

Variable	Type	Comment
x	float	X position on the (x,y) grid (nautical miles)
y	float	Y position on the (x,y) grid (nautical miles)
batX	float	New x path used when applying batcher's algorithm in CD&R (nautical miles)
batY	float	New y path used when applying batcher's algorithm in CD&R (nautical miles)
dx	float	Change in velocity in X direction (nautical miles/half second)
dy	float	Change in velocity in Y direction (nautical miles/half second)
alt	float	Altitude (in feet)
col	int	0: no collision path; 1: possible collision path
timeTill	float	time until next collision, default at 300. When time found less than default, close call incoming
colWith	int	Id of aircraft colliding with
rMatch	int	0: no radar matched; 1: one radar matched; -1: multiple radars matched

4.1 Initialization

Before we do any of the important tasks, we first initialize the data in a kernel function so that each thread takes a drone to initialize. The x and y values are initialized with random values between -128 and 128, so as to keep within the 256 nautical mile by 256 nautical mile flight simulation square area centered at (0,0). The dx and dy velocities are initialized with values between 30 and 600 nautical miles per hour, which is then halved to get the speed for each half second interval. The altitude is set as a random value between 500 and 6000 feet, and finally the collision is set to 0. [1, 4, 9]

4.2 Radar Correlation and Tracking

In this simulation, Radar Correlation and Tracking is the task that takes a shuffled list of generated radar data based on the last x and y data, as described in Chapter 4, and compares them against each drone to see which drone correlates with which radar, if any. Since our application runs a simulation of these ATC tasks, we have to create radar data that would simulate real radar data that in an actual application, would be coming in to the application from an outside source. Radar data is stored in a struct, just like the drones, and is also stored on global memory. [1, 4, 9]

Table 2: Radar Struct Table.

Variable	Type	Comments
rx	float	Radar x position on (x,y) graph. (nautical miles)
ry	float	Radar y position on (x,y) graph. (nautical miles)
rMatchWith	int	Id of aircraft correlated with

The first set of radar data is created using the values initialized by the first kernel function, SetupFlights. The initialized values are the x, y, dx, and dy values. To create the radar data, the GenerateRadarData kernel function takes in those values previously mentioned after they are initialized for the first time in the SetupFlights kernel function. The GenerateRadarData function takes the initialized aircraft data and advances their positions by one half second by adding dx to the x value and dy to the y value. This is known as the "expected position" for the aircraft based on the calculated velocities in the x and y directions for each half second. However, real-life radar data would not be exactly the same as the expected position due to factors such as irregular winds and such, slowing down or speeding up the aircraft or even blowing it slightly off course. To simulate this, a small, randomized value (show calculation?) is added to the x and y values. This way, each aircraft's "expected position" (which is just x + dy and y + dy) will correspond with it's radar position. This created data is copied back from the device after the execution of GenerateRadarData is complete. The radar data array is then split into fourths, with each fourth having it's data set reversed. This is done on the host before passing the data to the kernel function handling this task so that it can emulate real-life situations

where the incoming radar data will not be in the same order as the corresponding aircraft records. This radar data, along with the initialized aircraft data, are passed to TrackDrone. This kernel 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 it's own actual position in the airfield. At the start, 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 drones[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 nautical mile by 1 nautical mile bounding box around the aircraft. We do this by checking if the radar.x is less than aircraft.x + 0.5 and greater than aircraft.x – 0.5, and also if the radar.y is less than aircraft.y + 0.5 and greater than 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, the 'rMatch' variable of that aircraft is changed to 1, which is 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 it's initial value of -1. If the radar has not been matched before, then the rMatchWith variable will be -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 != -1, then any previously matched aircrafts are unmatched from this radar and the 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 it's expected location as it's 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 variables equaling -1, we increase the bounding square around the aircraft by doubling it and looping all the remaining, unmatched aircraft expected positions against the unmatched radars. If after this loop, we are not able to correlate the remaining radars with rMatchWith equaling -1, we double the bounding square around each aircraft expected position again 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. [1, 4, 9]

Once each radar has been matched with an aircraft expected position, or once the aircrafts have been checked two extra times with larger bounding boxes, it's 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 stores, i.e rMatchWith != -1 (unmatched) and rMatchWith != -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 it's actual location and not it's expected position. If we do not satisfy

one or both of these conditions, then the aircraft keeps it's x and y position until the next half second period where we will try to correlate it with a radar again. [1, 4, 9]

Algorithm 1. Algorithm for Tracking and Correlation

1: Radar data is generated on device, copied to host and shuffled then copied to device on global memory

2: Each thread calculates "expected position" for aircraft of the same id 'i'

3: **for** p = 0 -> N (number of aircrafts) **do**

4: Each thread uses radar of same id 'i' with bounding box of 1x1 nautical mile and checks if aircraft 'p' is within bounds

5: If there is an intersection, we check aircraft's rMatch[p] to make sure it's 0 (no radars correlated with this aircraft yet)

6: If rMatch is 0, change to 1. Radar 'i' and aircraft 'p' are now correlated.

7: Record id of aircraft 'p' in radar's rMatchWith[i]

8: If aircraft's rMatch[p] is 1 (previously correlated with radar) then change to -1 and drop correlation with radar. Or if radar's rMatchWith[i] is not -1 (other aircrafts matched with radar) then change to -2 and drop radar

9: **end for**

10: If some radars have rMatchWith[i] as -1 still (no aircraft correlation) double bounding box and repeat 3-9 again for those radars and unmatched planes (with rMatch as 0)

11: Bounding box is doubled again if there are still radars with rMatchWith[i] as -1 and 3-9 is repeated again for those radars and unmatched planes (with rMatch as 0)

12: After third round, correctly correlated aircrafts have their “expected positions” 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

---

4.3 Collision Detection and Resolution

This task is a combination of both collision detection and resolution, because in the case of the algorithm used, they operate in tandem with each other, and are therefore done in one kernel function. This task is performed at the beginning, middle and end of each simulation period, and is a time-consuming task that we will use to compare the systems. In our simulation, collision is only considered to be possible when the aircrafts are on altitudes within 1000 feet of each other. The outermost loop starts at t = 20 and must be less than 21, so it would only run once if we have no collision path within the loop. This is to have an easy way to change the look-ahead time from 20 to anything else like maybe 5 minutes. Inside that loop is the main loop where we iterate against every other aircraft in the structure and within the loop, we check that the current aircraft isn’t comparing to itself and that both aircrafts are within an altitude of 2000 nautical miles of each other. Only then will a conflict be considered, and from there, we apply the formulas of Batcher’s algorithm to check for collisions and then correct any courses where a collision could occur within a critical time span. [1, 4, 9]

4.3.1 Collision Detection

This task checks if an aircraft is on a collision path in the next 20 minutes using Batcher's algorithm. 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. For each aircraft, a 1.5 nautical mile bound is considered for each x and y position to make sure collisions are considered possible within 3 nautical miles of the aircraft. 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 timeTill variable to the value of 300. This variable determines when the next collision will occur for the specific aircraft, and 300 is what is considered a safe number according to Batcher's algorithm which we will use to find out if an aircraft is on a collision path. As seen in the above figure (insert number), 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. We then use the formulas in the figure below the graph for Batcher's algorithm (figure number) 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 nautical miles by and 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 minx, maxX, minY and maxY values. We then use equations 5-6 to determine the timeMin and timeMax. 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 timeMin and timeMax variables, respectively. When we finally have the timeMin and timeMax values, we know that we are on a collision path if timeMin is a smaller value than timeMax. 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 timeMin is less than timeTill, which was initialized at 300 When that happens, we set the timeTill for both aircrafts to the value of timeMin and update the colWith with the appropriate id's for each aircraft. [1, 4, 9]

4.3.2 Collision Resolution

If an aircraft is on a collision path, we check the timeMin value and compare it with our default timeTill value, initialized at 300. If timeMin is less than timeTill, 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 determines from the Batcher's algorithm process, and perform the collision detection task again using these updated batx and baty values. After rotating, we reset the loop by setting the t value to 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 the collision detection task 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 timeMin < timeTill. 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 possible and it works well with a reasonable amount of aircrafts. In practice, collision 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. [1, 4, 9]

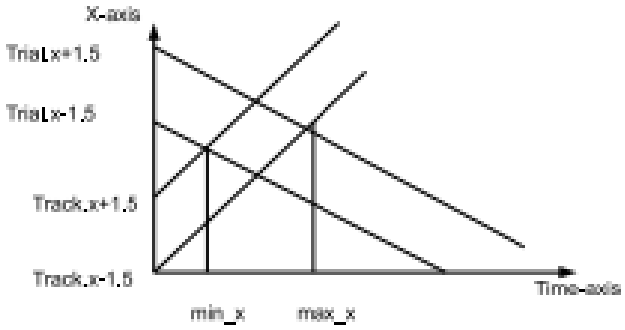

Figure 4: Batcher's Algorithm for X dimension

Algorithm 2. Algorithm for Collision Detection and Resolution

1: Each thread takes the aircraft of the same thread id ‘i’ to iterate it against all aircraft p

2: **for** p = 0 -> N (number of aircraft) in parallel **do**

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 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 3x3 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 is less than time_max, the aircraft ‘i’ is on a collision course with aircraft ‘p’

8: **If** time_min is less than the time_till of the track aircraft ‘i’, then that time_till is updated to be time_min and the collision is considered to be happening soon, so **then**

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 2-7 to do the collision detection task again with the new track path

12: **end if**

13: **end if**

14: if we check against all aircraft ‘p’ with the thread’s aircraft ‘i’ and our chk variable is more than 0, indicating that we have attempted to change course, then we 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

15: **end for**

$$min\_x = \frac{|trial.X_c - track.X_t| - 3}{|trial.V_{xc} - track.V_{xt}|} \quad (1)$$

$$max\_x = \frac{|trial.X_c - track.X_t| + 3}{|trial.V_{xc} - track.V_{xt}|} \quad (2)$$

$$min\_y = \frac{|trial.Y_c - track.Y_t| - 3}{|trial.V_{yc} - track.V_{yt}|} \quad (3)$$

$$max\_y = \frac{|trial.Y_c - track.Y_t| + 3}{|trial.V_{yc} - track.V_{yt}|} \quad (4)$$

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

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

Figure 5: Formulas used for Batcher’s algorithm

CHAPTER 5

Results

This chapter describes the results of comparing the performance of our CUDA devices and the AP device based on important time-consuming tasks that give us a good idea of the performance difference. The tasks are Correlation and Tracking, Collision Detection, and Collision Resolution. 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 show some interesting results and shows us that it is able to perform the tasks in less time than the AP implementation (STARAN), and the ClearSpeed emulation of the AP solution, while also never missing deadlines. The timing results also show us that NVIDIA-CUDA is able to achieve almost linear polynomial curves.

5.1 Experimental Setup

The CUDA solution is set up to perform the three ATC task on a certain number of aircrafts just like the AP solution. This program is the implementation of ATC tasks on an Associative Processor built specifically to perform ATC tasks on the STARAN machine, the first of which was designed by Dr. Kenneth Batcher. The number of aircrafts also dictates the block/thread setup in the CUDA solution. If there are 96 aircrafts, then the setup is 1 block and 96 threads in that block. If it's more, 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. The main tasks that are the most time-consuming and also the most critical are the tracking and correlation task and the collision detection and resolution task. Those are the two tasks that we will look at to compare the timings of our CUDA solution and Mike Yuan's AP solution. (add reference).

We performed the CUDA tests and got their timings on three devices, the GTX880M and 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 recently awarded to our research team by NVIDIA as a grant and has the compute capacity 6.1 and the most recent CUDA architecture on it, Pascal.

5.2 Experimental Results

5.2.1 Tracking and Correlation graphs and timing data

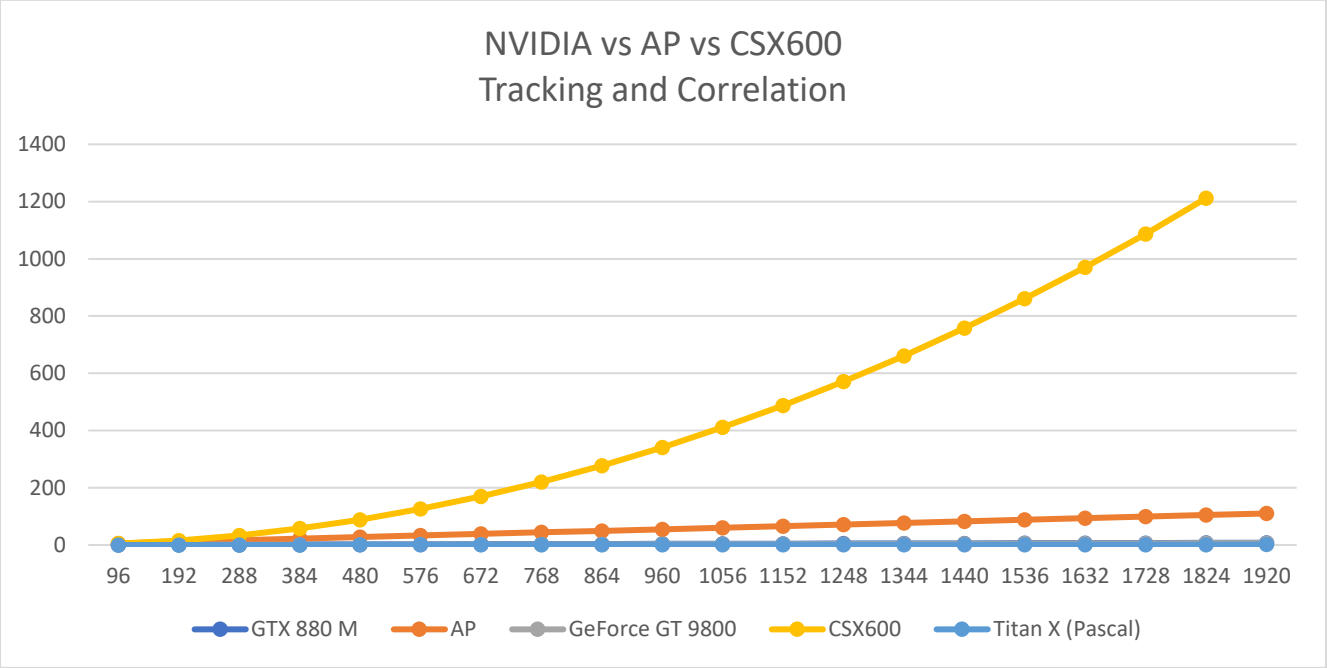

Figure 6: Tracking and Correlation – NVIDIA vs AP vs CSX600

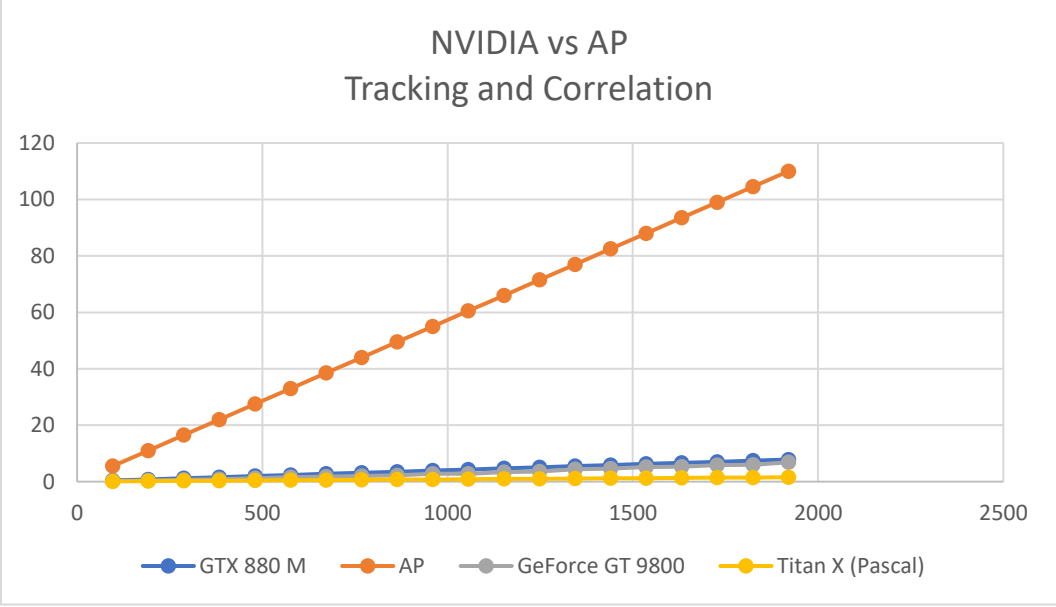

Figure 7: Tracking and Correlation - NVIDIA vs AP

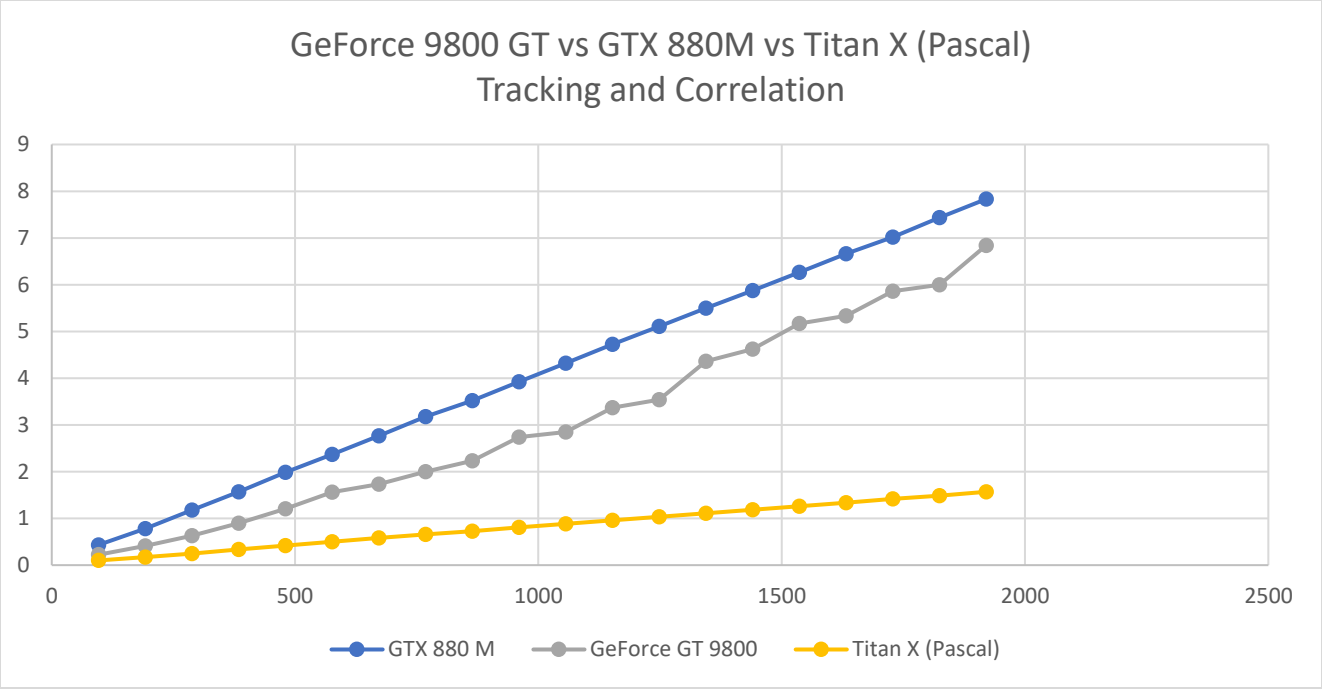

Figure 8: Tracking and Correlation - GeForce 9800 GT vs GTX 880M vs Titan X (Pascal)

In the above graphs, we compare the timings of the tracking and correlation tasks for our five devices. Multiple graphs have been included to better show the timings comparisons, as the CSX600 and AP numbers are so high, we do not get to see the distinction between the GeForce 9800 GT, the GTX 880M, and the Titan X (Pascal) timings except for in the third graph. It is also easier to see the nature of the curves of each machine's timings with the graphs split up in this way. The timing for the NVIDIA CUDA devices are taken as an average of the time it takes each iteration to perform this task, with a total of 64 iterations. This task is performed every 0.5 seconds and in the case of the CUDA devices, never misses a deadline. More about the findings based on these graphs is discussed later in the section discussion about our observation of this data.

Table 3: Tracking and Correlation timing data – NVIDIA, AP, CSX600

Number of Aircrafts	GTX 880 M	AP	GeForce GT 9800	CSX600	Titan X (Pascal)
96	0.432367	5.5	0.223066	4.9	0.100266
192	0.781216	11	0.412273	15.9	0.174547
288	1.178549	16.5	0.632038	33.5	0.249957
384	1.56838	22	0.894232	57.7	0.336558
480	1.988652	27.5	1.205932	88.4	0.417092
576	2.366518	33	1.559513	125.9	0.498639
672	2.767772	38.5	1.729252	169.8	0.579415
768	3.175822	44	2.002404	220.4	0.659704
864	3.521459	49.5	2.233358	277.3	0.726411
960	3.921055	55	2.739625	340.9	0.809999
1056	4.322921	60.5	2.850146	411.1	0.886659
1152	4.721221	66	3.369718	487.8	0.961276
1248	5.108572	71.5	3.543189	571.4	1.036819
1344	5.498272	77	4.362959	660.9	1.109828
1440	5.876674	82.5	4.618795	757.4	1.183798
1536	6.262564	88	5.167105	860.4	1.261841
1632	6.659552	93.5	5.336566	970.1	1.338035
1728	7.0167	99	5.862486	1086.2	1.414719
1824	7.432383	104.5	5.994947	1211.5	1.488991
1920	7.830879	110	6.840152		1.56858

This graph has the raw timing data in milliseconds (ms) for the tracking and correlation task for each of the devices being compared here. [9]

5.2.2 Collision Detection & Resolution graphs and timing data

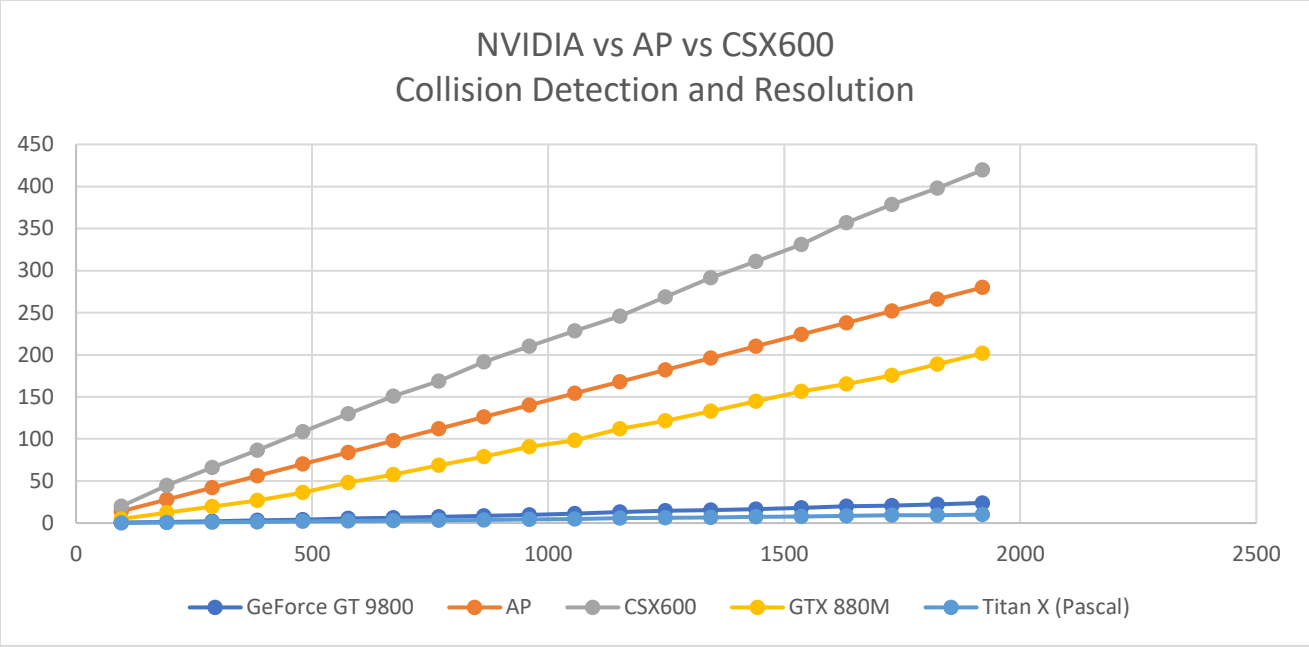

Figure 9: Collision detection and resolution – NVIDIA vs AP vs CSX600

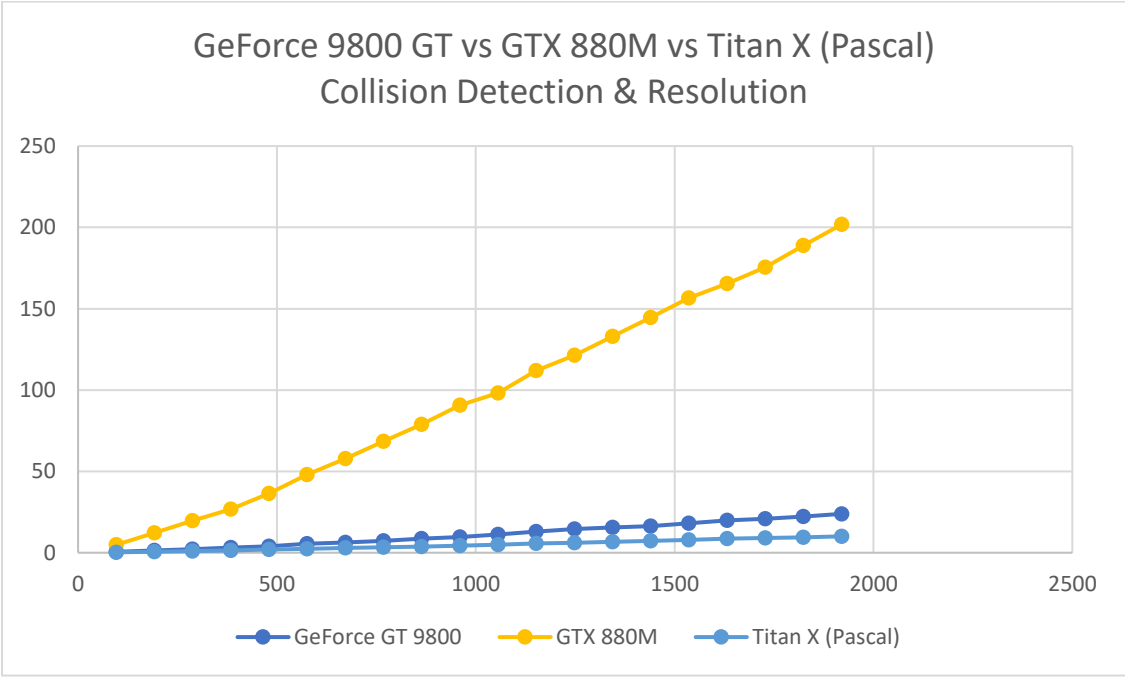

Figure 9: Collision detection and resolution – GeForce 9800 GT vs GTX 880M vs Titan X (Pascal)

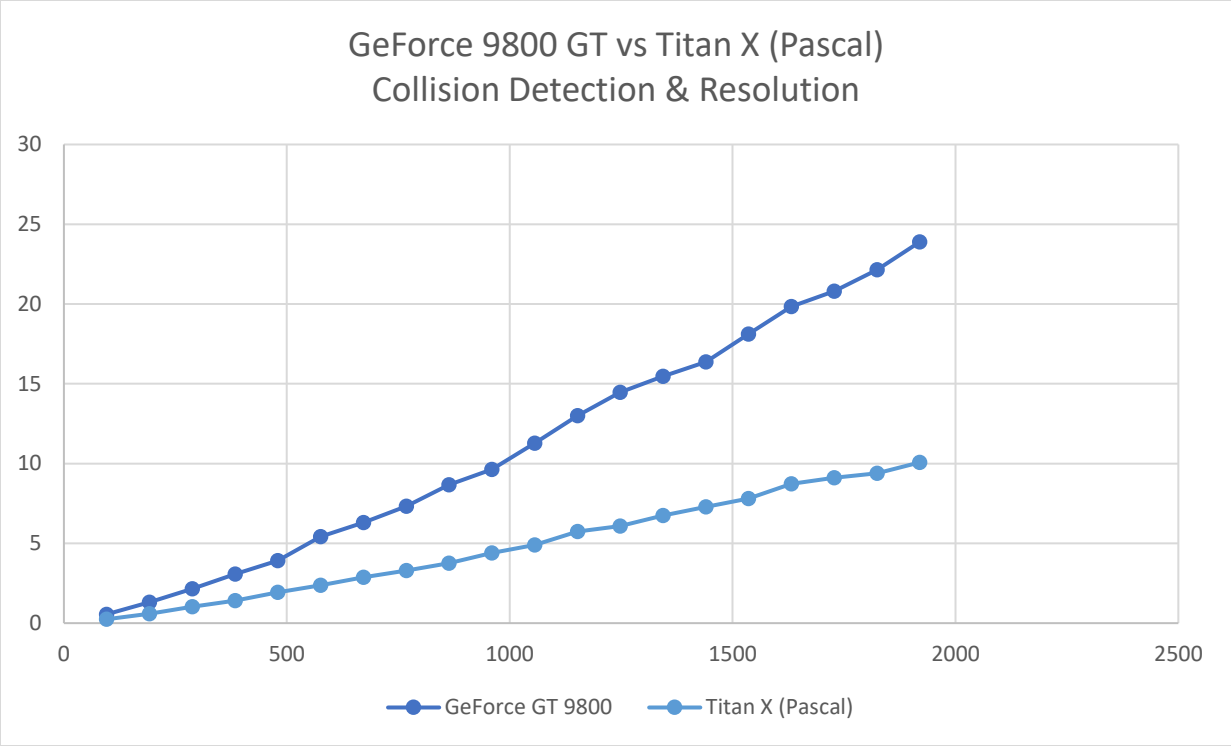

Figure 10: Collision detection and resolution – GeForce 9800 GT vs Titan X (Pascal)

In the graphs above we compare the combined timings for the collision detection and collision resolution tasks of the NVIDIA CUDA implementation and the AP (STARAN) implementation and the ClearSpeed emulation of AP. The timings for the NVIDIA CUDA implementation are taken on the GeForce 9800 GT, the GTX 880M, and the Titan X (Pascal) devices, and are taken as an average of the timings of 4 iterations of this task.

Table 4: Collision detection and resolution timing data – NVIDIA, AP, CSX600

Number of Aircrafts	GeForce GT 9800	AP	CSX600	GTX 880M	Titan X (Pascal)
96	0.555728	14	20	4.852704	0.245624
192	1.30076	28	44.5	12.286863	0.587
288	2.153112	42	65.8	19.681648	1.03136
384	3.065448	56	86.4	26.829536	1.404304
480	3.927008	70	108.5	36.358055	1.928144
576	5.4178	84	129.8	48.063728	2.36324
672	6.29668	98	150.6	57.751953	2.86608
768	7.32976	112	168.5	68.45813	3.30332
864	8.663464	126	191.5	78.915344	3.766048
960	9.625591	140	210.2	90.667419	4.3898
1056	11.273232	154	228.5	98.259506	4.902768
1152	13.002424	168	245.9	111.915611	5.740128
1248	14.468784	182	268.6	121.350845	6.08356
1344	15.476345	196	291.5	132.984772	6.74464
1440	16.365799	210	310.8	144.574982	7.282992
1536	18.124376	224	331.1	156.549316	7.814968
1632	19.836063	238	356.9	165.363739	8.723088
1728	20.804192	252	378.4	175.45755	9.110847
1824	22.148392	266	398.1	188.939377	9.39132
1920	23.893448	280	419.5	201.790222	10.074904

This graph has the raw timing data in milliseconds (ms) for the collision detection and resolution task for each of the devices being compared here. [9]

5.2.3 Tracking and Correlation + Collision Detection and Resolution graph and timing data

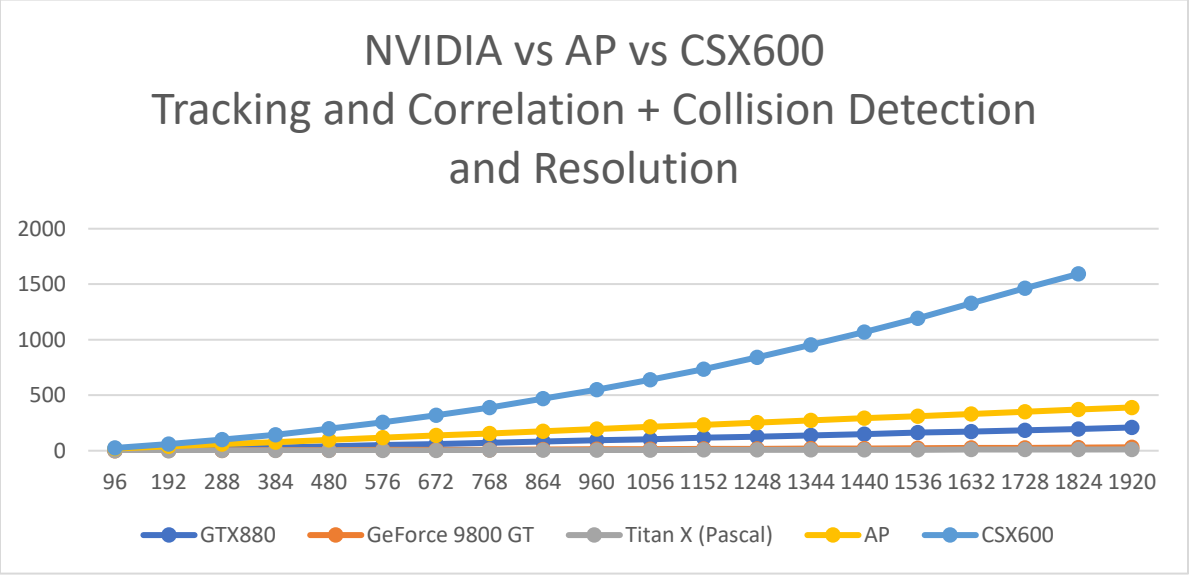

Figure 11: Tracking and correlation + collision detection and resolution: NVIDIA vs AP vs CSX600

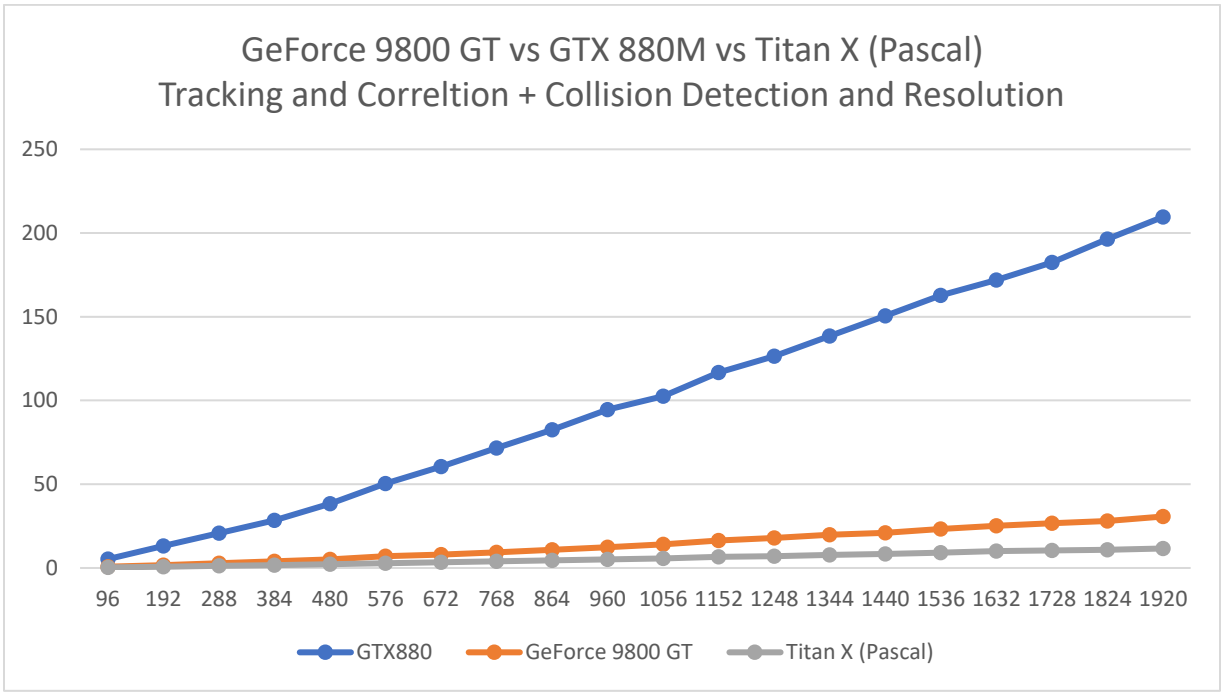

Figure 12: Tracking and correlation + collision detection and resolution: GeForce 9800 GT vs GTX 880M vs Titan X (Pascal)

These graphs show us the performance of both tasks together and compares how each device performs both tasks. We add up the average tracking and correlation runtime with the average collision detection and resolution runtime to get the data being shown here. This helps us see how all three tasks are performing together and how good the curve of that performance is for the application as a whole.

Table 5: Tracking and correlation + collision detection and resolution timing data : NVIDIA vs AP vs CSX600

Number of Aircrafts	GTX880	GeForce 9800 GT	Titan X (Pascal)	AP	CSX600
96	5.285071	0.778794	0.34589	19.5	24.9
192	13.06808	1.713032	0.761546	39	60.4
288	20.860197	2.78515	1.281316	58.5	99.3
384	28.397917	3.95968	1.740861	78	144.1
480	38.346706	5.13294	2.345236	97.5	196.9
576	50.430244	6.977314	2.861879	117	255.7
672	60.519726	8.025931	3.445495	136.5	320.4
768	71.633949	9.332164	3.963024	156	388.9
864	82.436806	10.896822	4.492459	175.5	468.8
960	94.588478	12.365215	5.1998	195	551.1
1056	102.582428	14.123377	5.789427	214.5	639.6
1152	116.636833	16.372143	6.701405	234	733.7
1248	126.459419	18.011972	7.120378	253.5	840
1344	138.483047	19.839304	7.854468	273	952.4
1440	150.45166	20.984594	8.46679	292.5	1068.2
1536	162.811874	23.291481	9.076809	312	1191.5
1632	172.023285	25.17263	10.061123	331.5	1327
1728	182.474243	26.666677	10.525566	351	1464.6
1824	196.371765	28.143339	10.880311	370.5	1592.6
1920	209.621094	30.733601	11.643484	390	

The above table shows us the raw data for the combined average runtime of the tracking and correlation task and the collision detection and resolution task for each device. [9]

5.3 Curve-fitting results of the timing data

Using the software MATLAB's curve-fitting tool, we were able to get more clear information about the nature of the timing data that we have acquired from these devices by examining what kind of timing curve they fit best. The x axis is the number of aircrafts and the y axis is the timing data for the given task on that device at the aircraft count intervals, like the data shown in the previous sections of this chapter. Most of these curves will show that a polynomial 1 (or linear time) curve is what fits them best. For others, a polynomial 2 (or quadratic) will fit them best. From polynomial 3 and onward, the curve seems to fit better, but the terms with the highest exponent start having negative coefficients, which ends up subtracting from the other terms. We show the polynomial 3, or cubic, fit for these timings to demonstrate the previous statement. The "goodness of fit" for a curve is decided based on four values. The first of these values is the SSE, or the Sum of Squares Due to Error, which is a statistic used to measure the total deviation of the points from the fit. The closer the number is to 0, the smaller the random error component, which will make the fit more useful for prediction. The second value is called R-Square, which is the measure of how successful the fit is in explaining the variation of the data. The closer this is to 1, the greater the amount of variance in the data accounted for by the fit, so we would want this value to be as close to 1 as possible, with 1 being the best value. The third value is the degrees of freedom adjusted R-square, which uses the first value we listed, R-Square, and adjusts it based on the residual degrees of freedom. The closer it is to 1, the better the fit. And finally is the root mean squared error, which is the estimate of standard deviation of the random component in the data. A value closer to 0 indicates a better fit [10].

5.3.1 GeForce 9800 GT: Tracking and Correlation

The following graphs show the linear, quadratic, and cubic polynomial curve fittings for the timings of the tracking and correlation task being performed by the GeForce 9800 GT. We can see that the linear fit is good, but the quadratic curve fits it best, with it's coefficient being a very small value, indicating that it is also still close to being a linear fit. The coefficient is small enough to make the linear term more dominate over the domain (i.e., number of aircraft being considered).

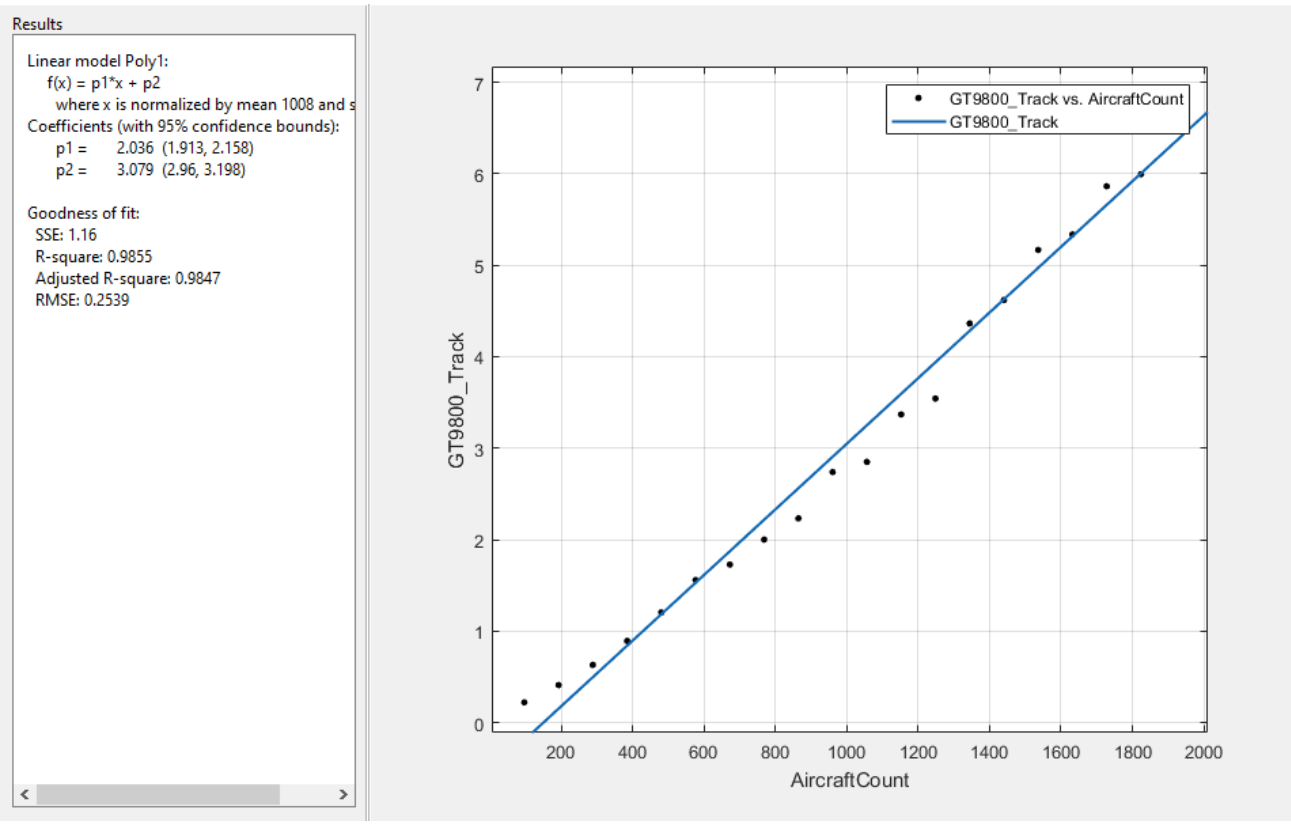

Figure 13: Tracking and correlation linear curve fitting for GeForce 9800 GT

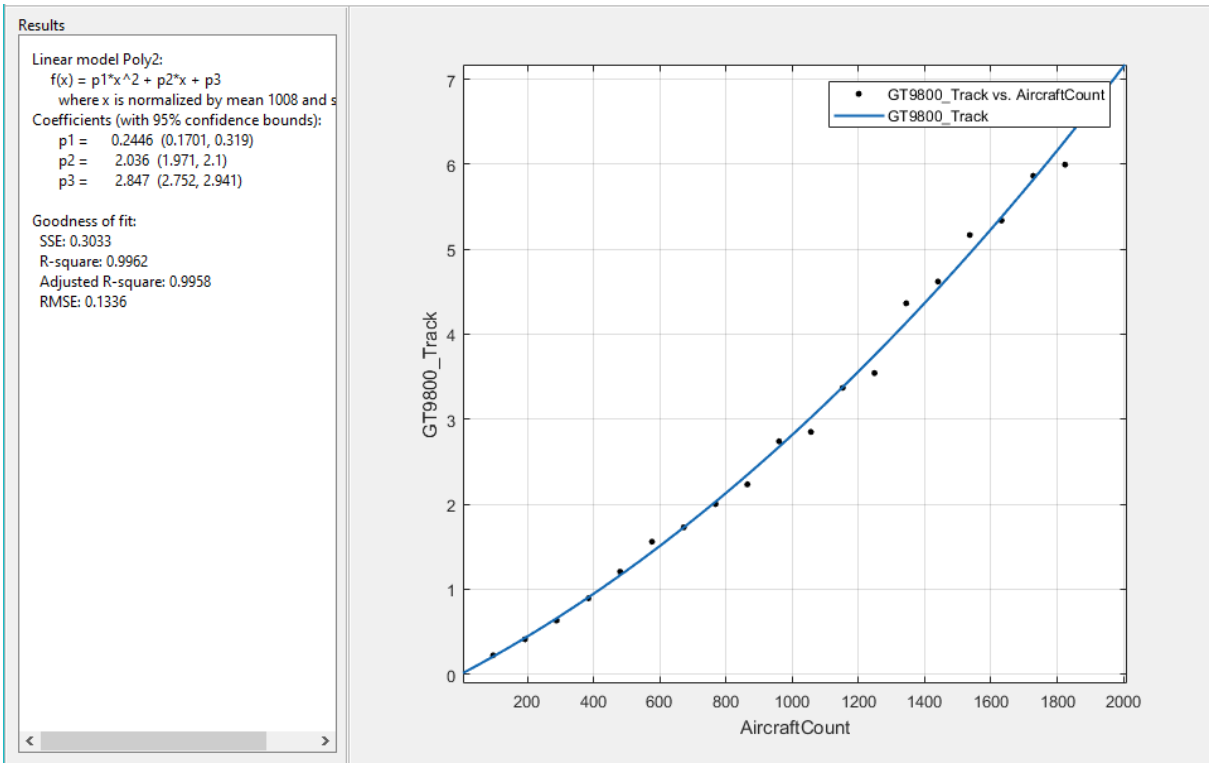

Figure 14: Tracking and correlation quadratic curve fitting for GeForce 9800 GT

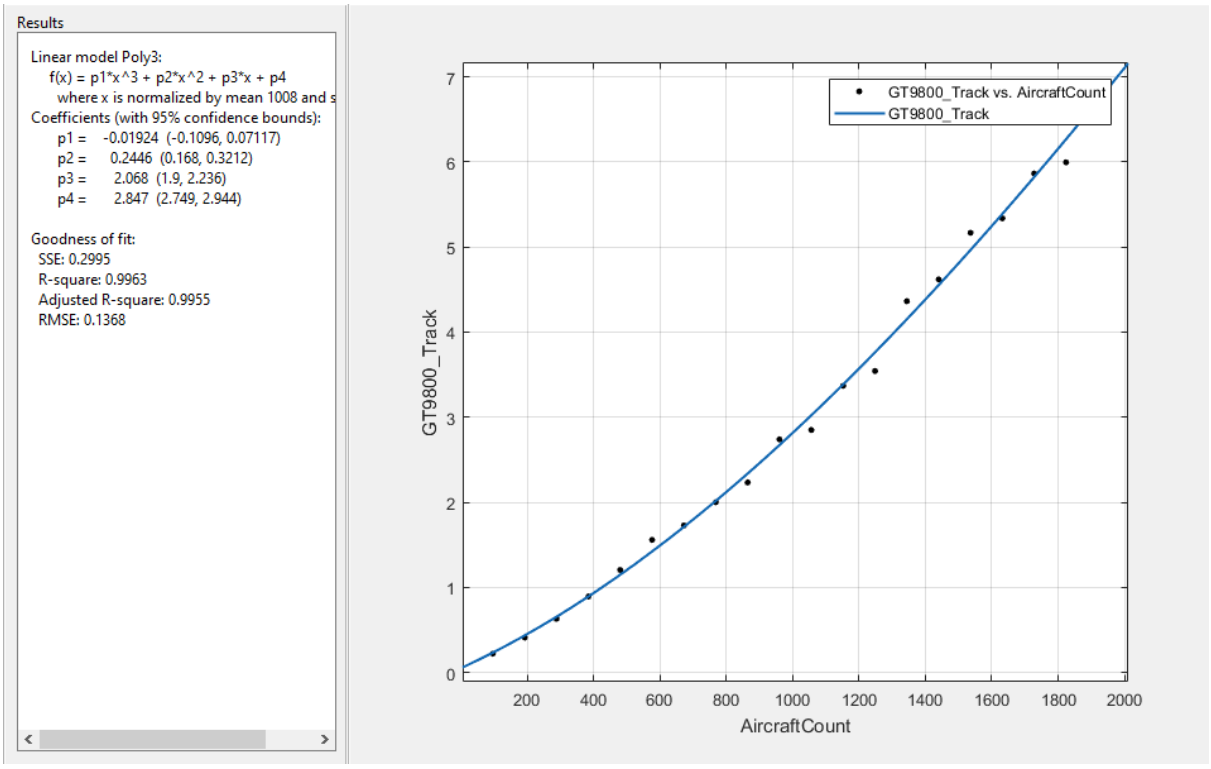

Figure 15: Tracking and correlation cubic curve fitting for GeForce 9800 GT

5.3.2 GeForce 9800 GT: Collision Detection and Resolution

These following graphs are the curve-fitting results of the collision detection and resolution task on the GeForce 9800 GT. These results show a good linear fit that looks to fit a quadratic curve better. However, due to the quadratic coefficient being so small and the linear being much higher, this curve is considered to be more linear.

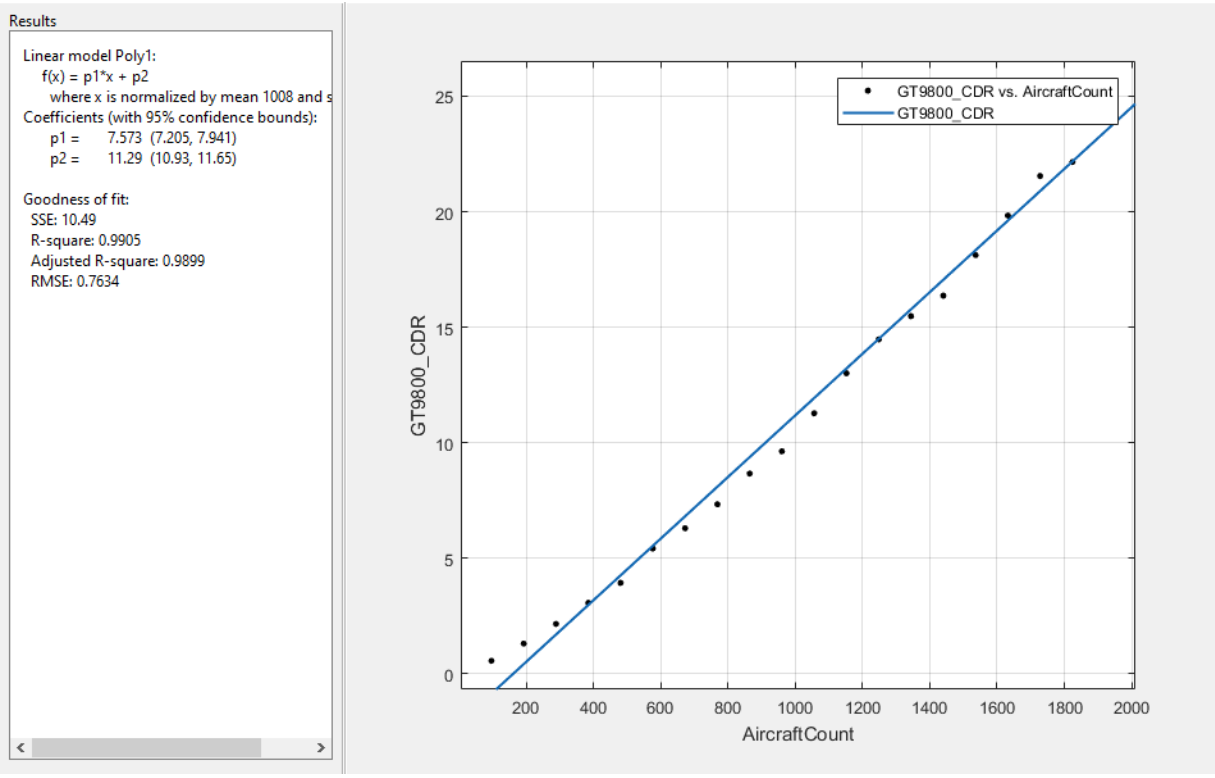

Figure 16: Collision detection and resolution linear curve fitting for GeForce 9800 GT

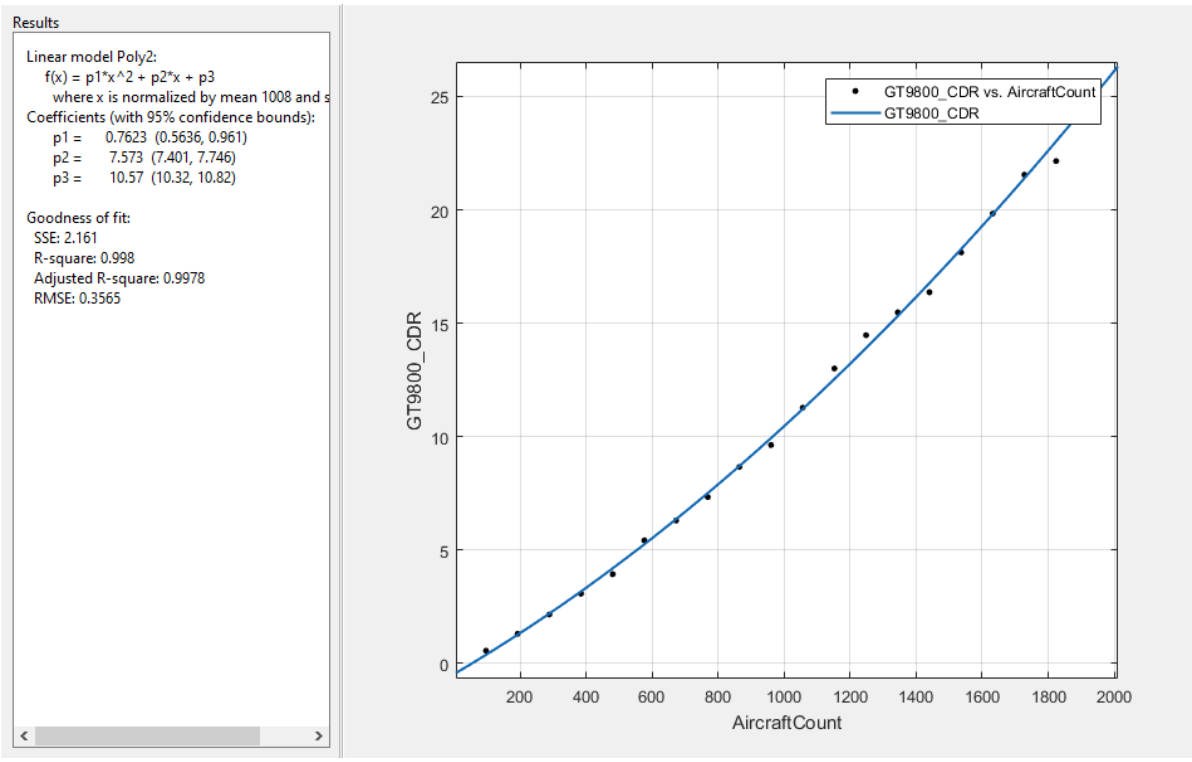

Figure 17: Collision detection and resolution quadratic curve fitting for GeForce 9800 GT

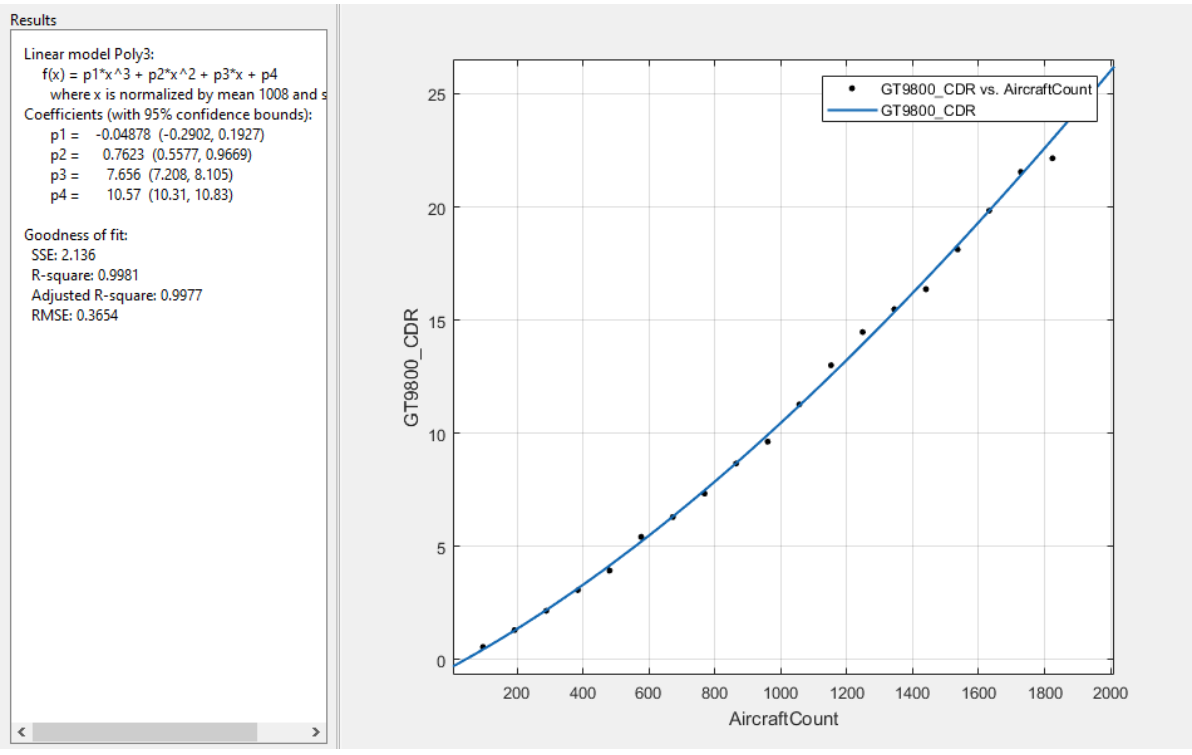

Figure 18: Collision detection and resolution cubic curve fitting for GeForce 9800 GT

5.3.3 GeForce 9800 GT: Tracking and Correlation + Collision Detection and Resolution

These graphs below show the combined results of all three of our tasks, which gives us a good overview of how the device performs on iterations where both of these tasks are called and have to meet that half second deadline. The curve fitting results show a good linear fit, which is helped by correlation and tracking fit, which was found to be exactly linear. However, it does show a better quadratic fit, with a much lower SSE value. A cubic fit makes it so the coefficient is negative, so we do not consider it.

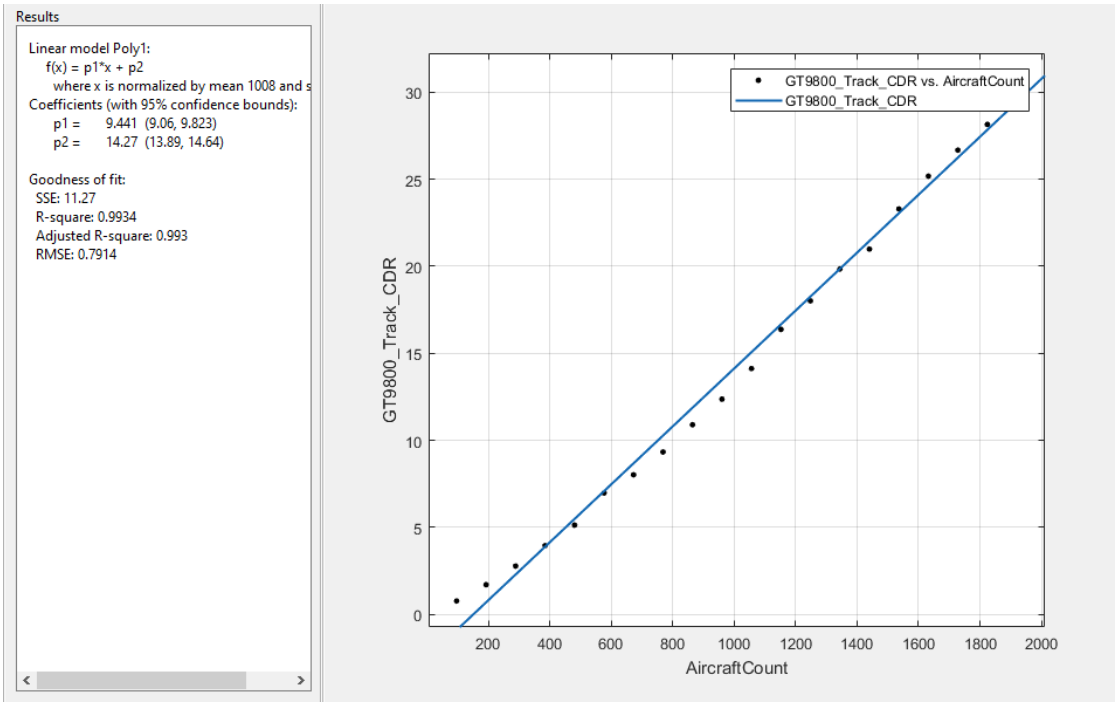

Figure 19: Tracking and correlation + collision detection and resolution linear curve fitting for GeForce 9800 GT

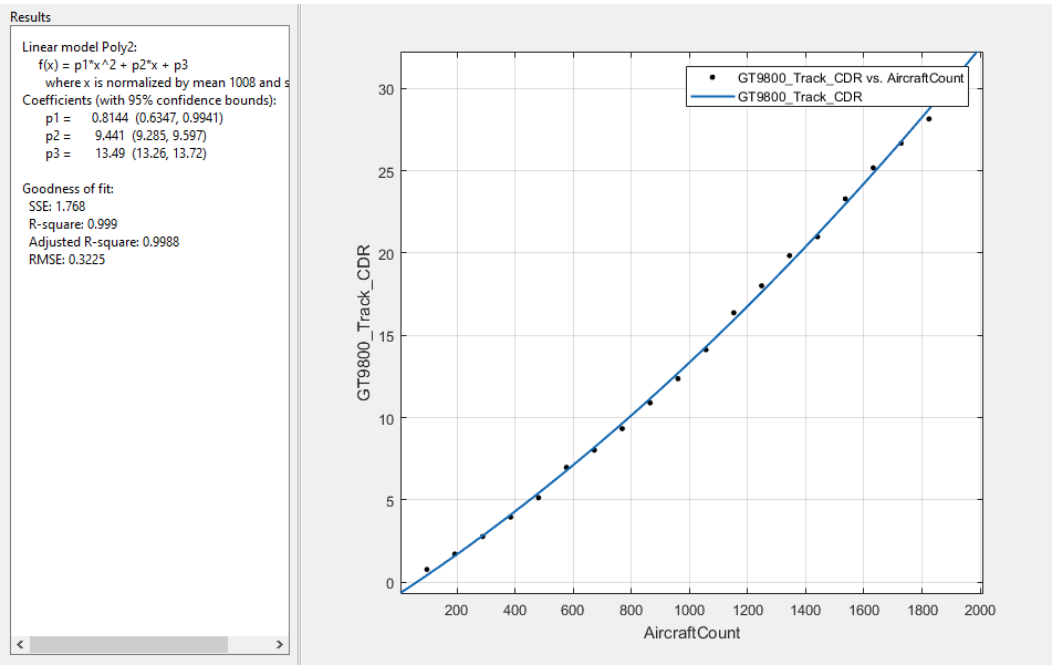

Figure 20: Tracking and correlation + collision detection and resolution quadratic curve fitting for GeForce 9800 GT

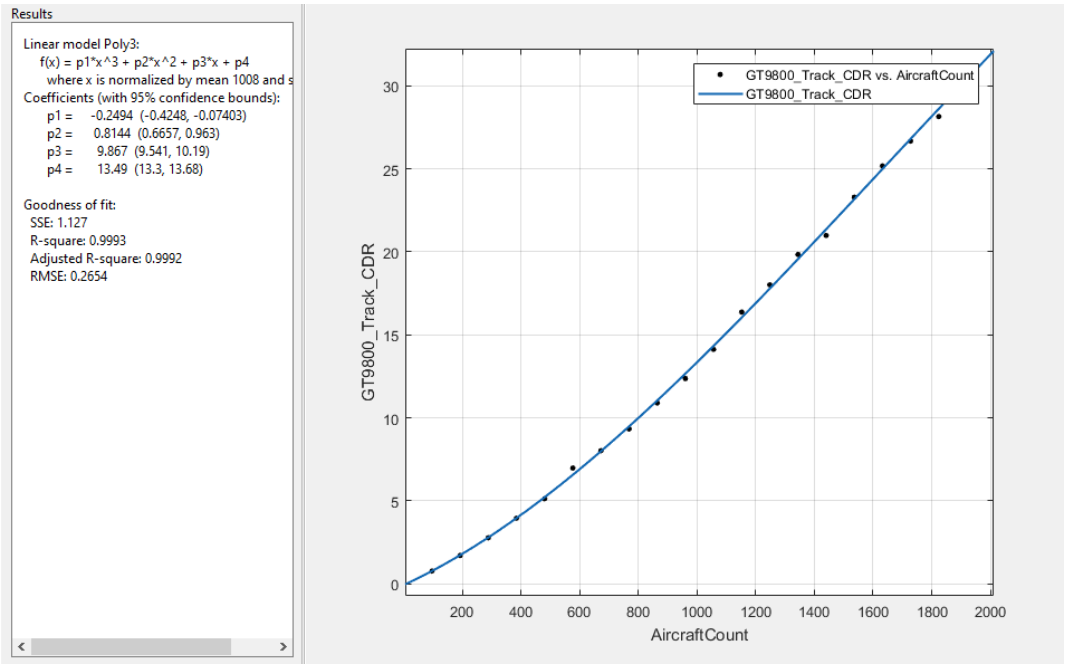

Figure 21: Tracking and correlation + collision detection and resolution cubic curve fitting for GeForce 9800 GT

5.3.4 GTX 880M: Tracking and Correlation

These graphs show the curve fitting results of the tracking and correlation task timings on the GTX 880M device. These results show a linear fit for the timing results, which further proves the SIMD-like nature of NVIDIA-CUDA devices. A quadratic fit would not be better, as it has a negative coefficient.

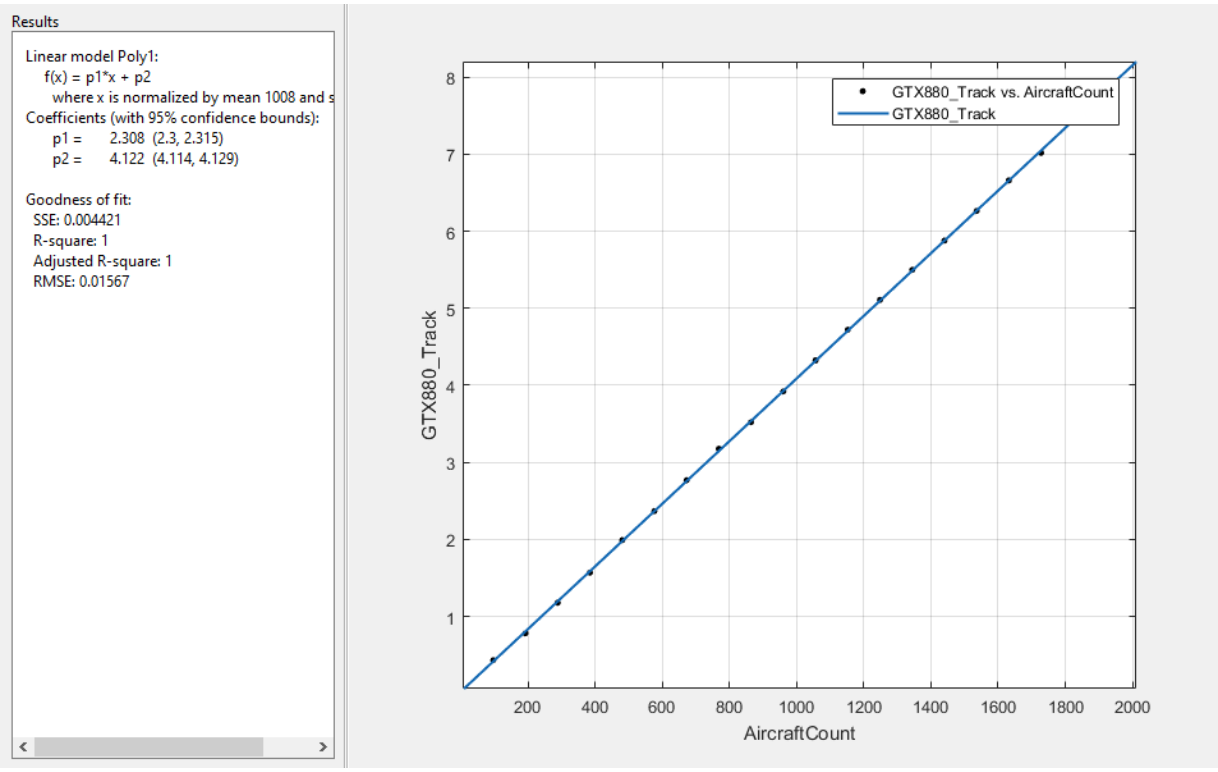

Figure 22: Tracking and correlation linear curve fitting for GTX 880M

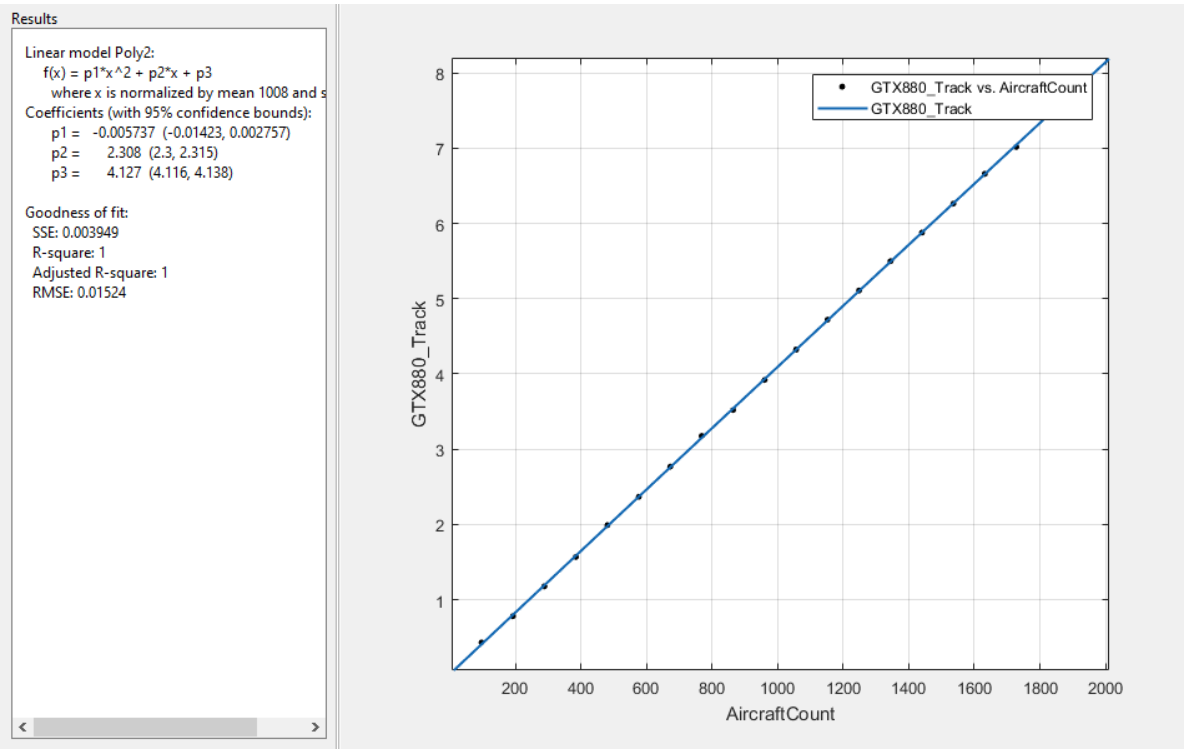

Figure 23: Tracking and correlation quadratic curve fitting for GTX 880M

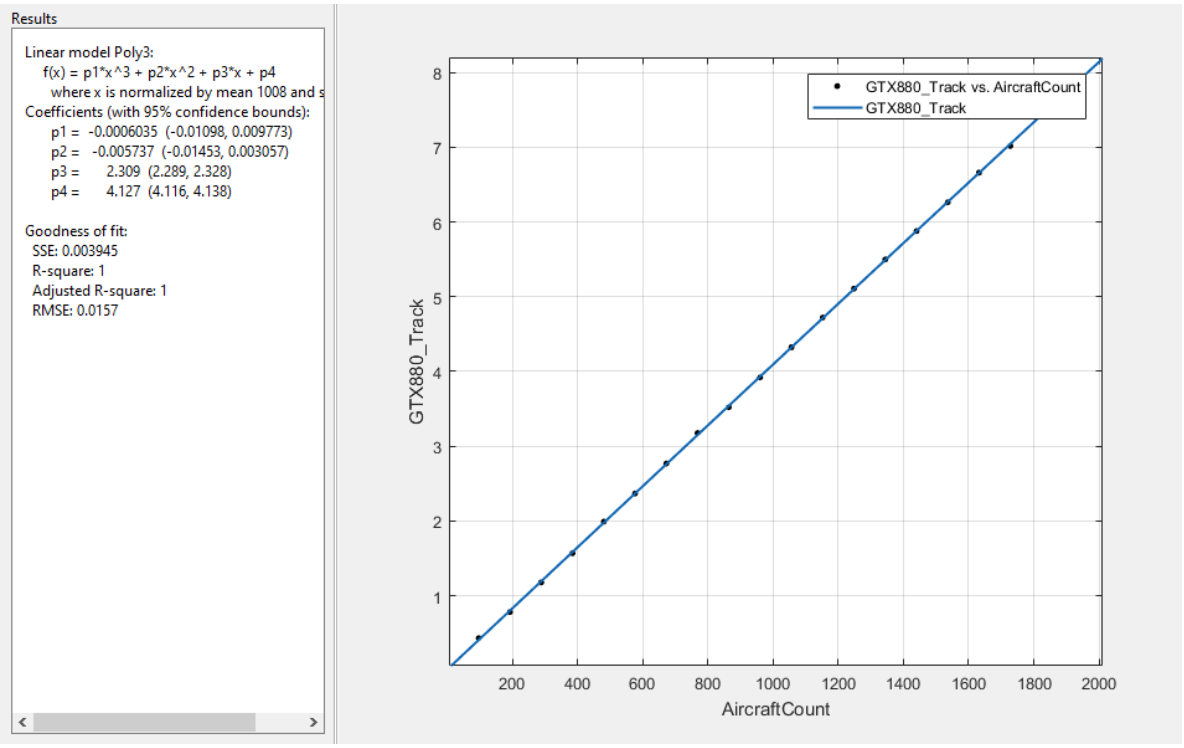

Figure 24: Tracking and correlation cubic curve fitting for GTX 880M

5.3.5 GTX 880M: Collision Detection and Resolution

The graphs below show the curve fitting results of the collision detection and resolution task timing results on the GTX 880M device. The results here show a good linear fit, although with a high SSE value. The quadratic fit has a much lower SSE value and seems to have a better fit. The cubic curve again has a negative coefficient, so we do not consider it.

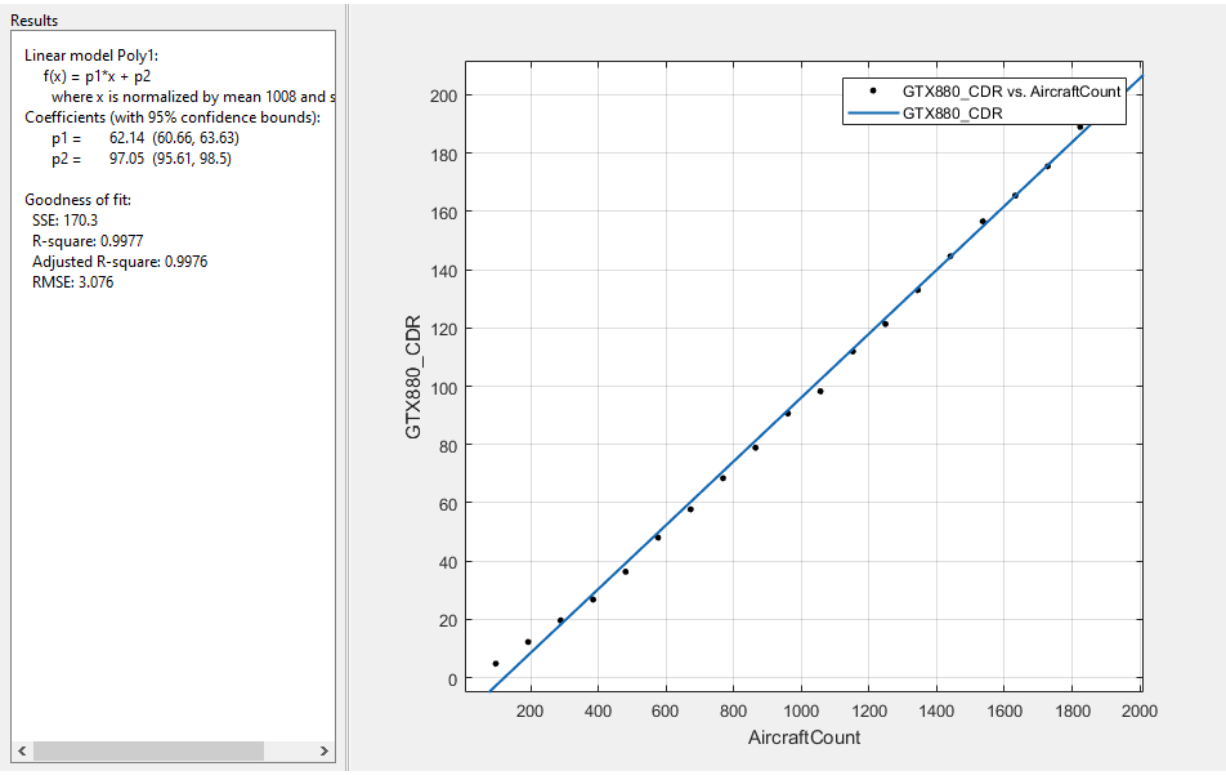

Figure 25: Collision detection and resolution linear curve fitting for GTX 880M

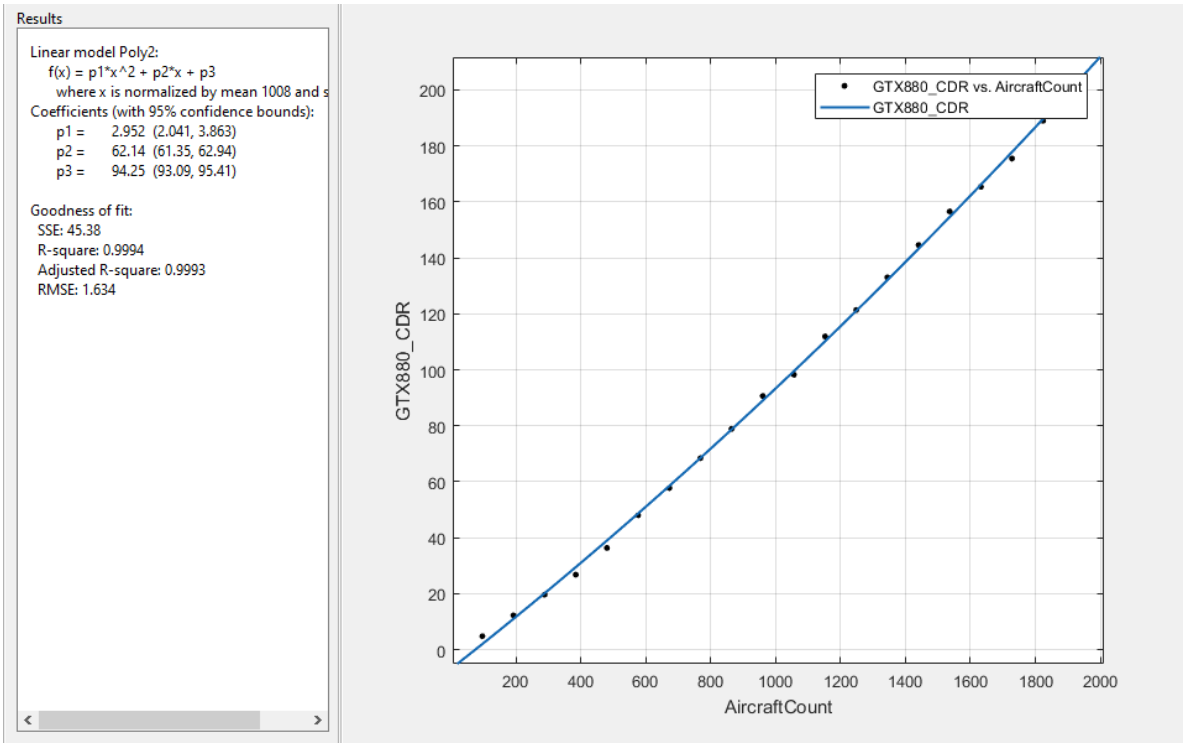

Figure 26: Collision detection and resolution quadratic curve fitting for GTX 880M

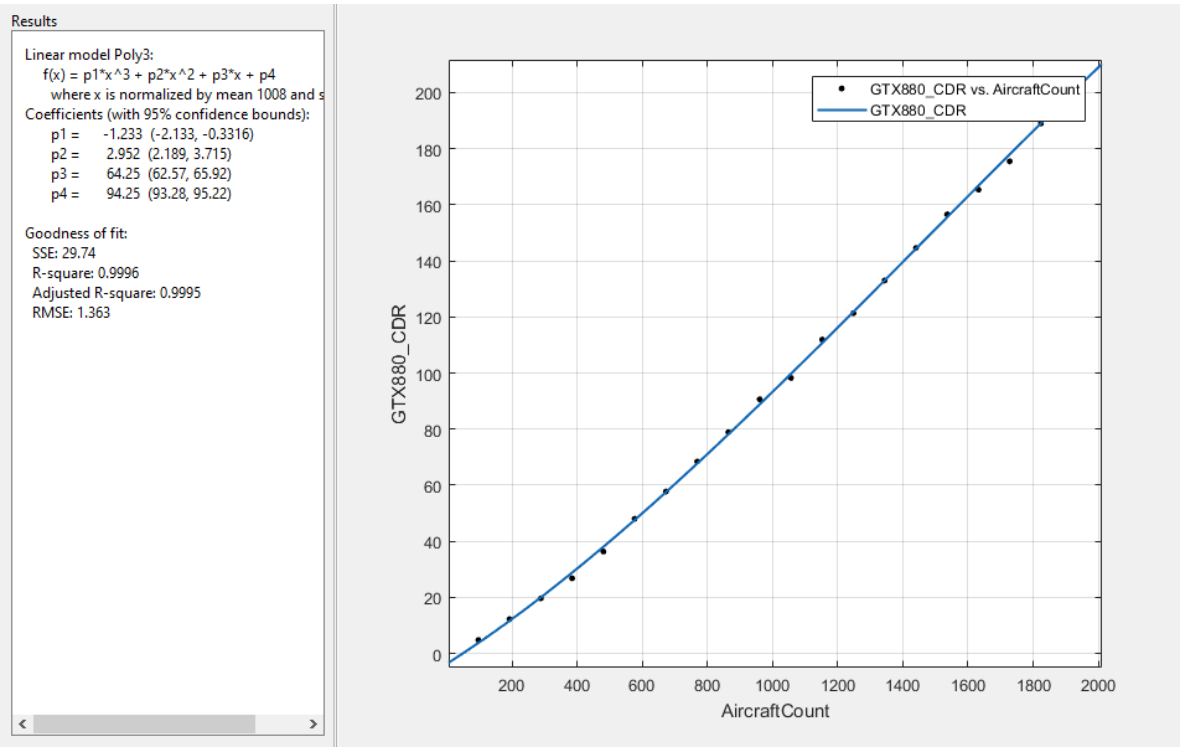

Figure 27: Collision detection and resolution cubic curve fitting for GTX 880M

5.3.6 GTX 880M: Tracking and Correlation + Collision Detection and Resolution

The graphs below show the curve fitting results for the combined timing results of the tacking and correlation task and the collision detection and resolution tasks. The quadratic curve of the collision detection and resolution timings make the curve fitting result here closer to quadratic, according to the "goodness of fit values".

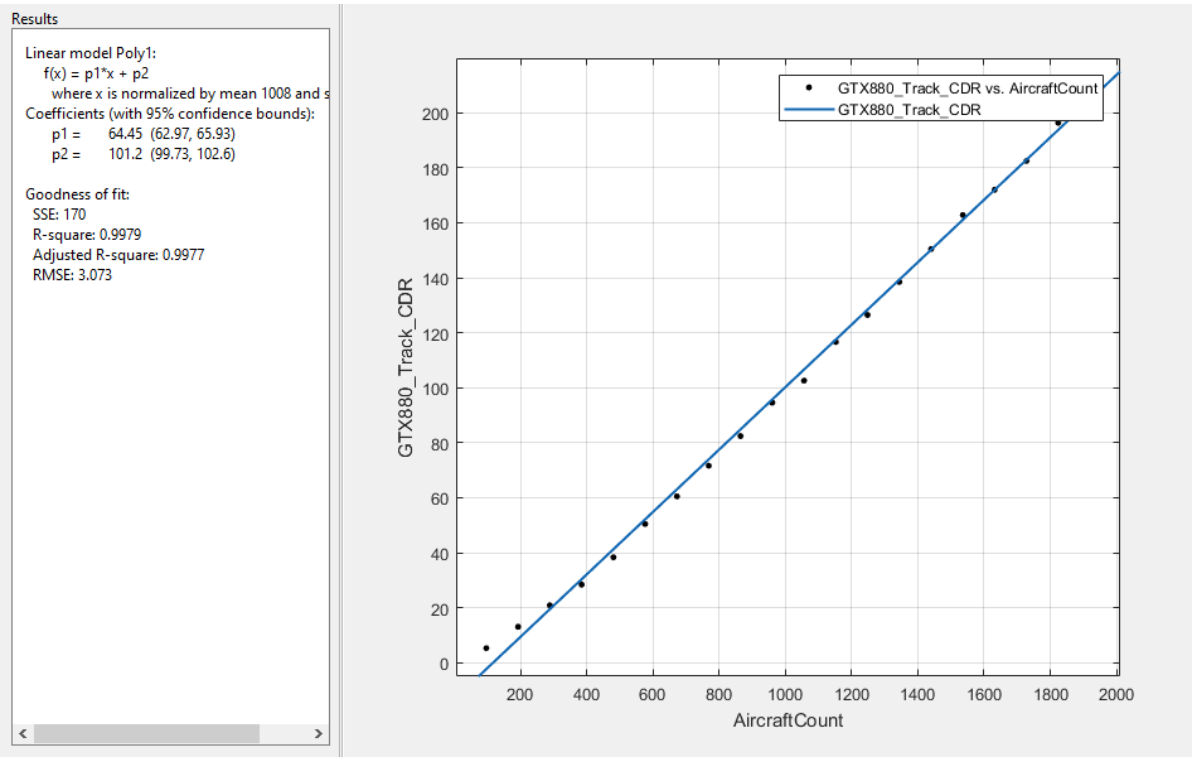

Figure 28: Tracking and correlation + collision detection and resolution linear curve fitting for GTX 880M

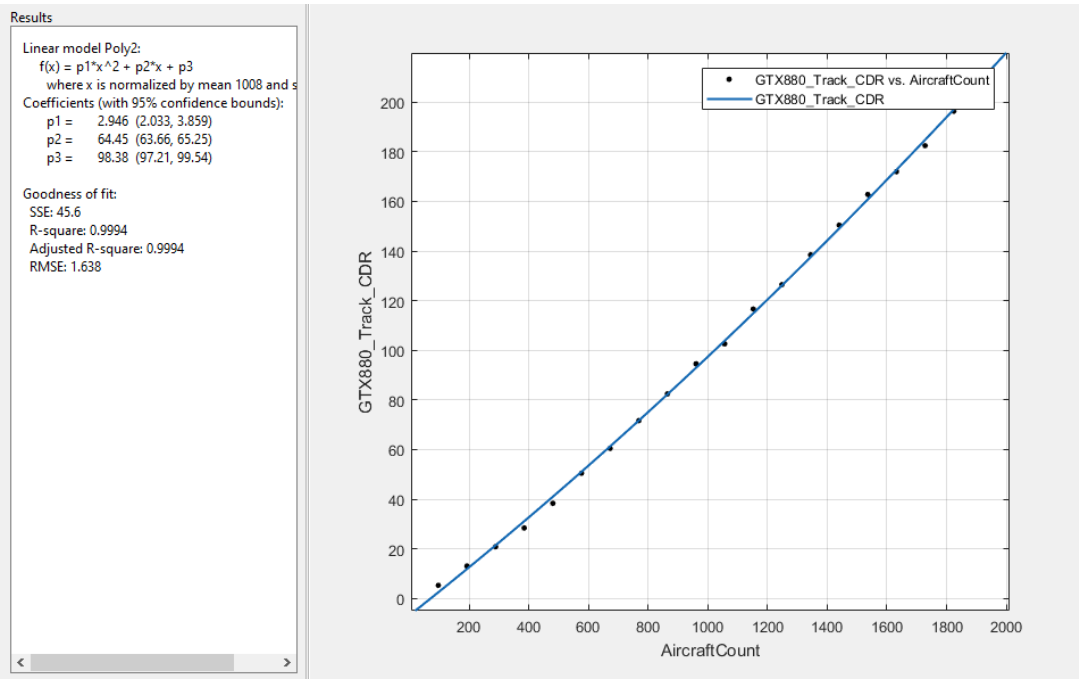

Figure 29: Tracking and correlation + collision detection and resolution quadratic curve fitting for GTX 880M

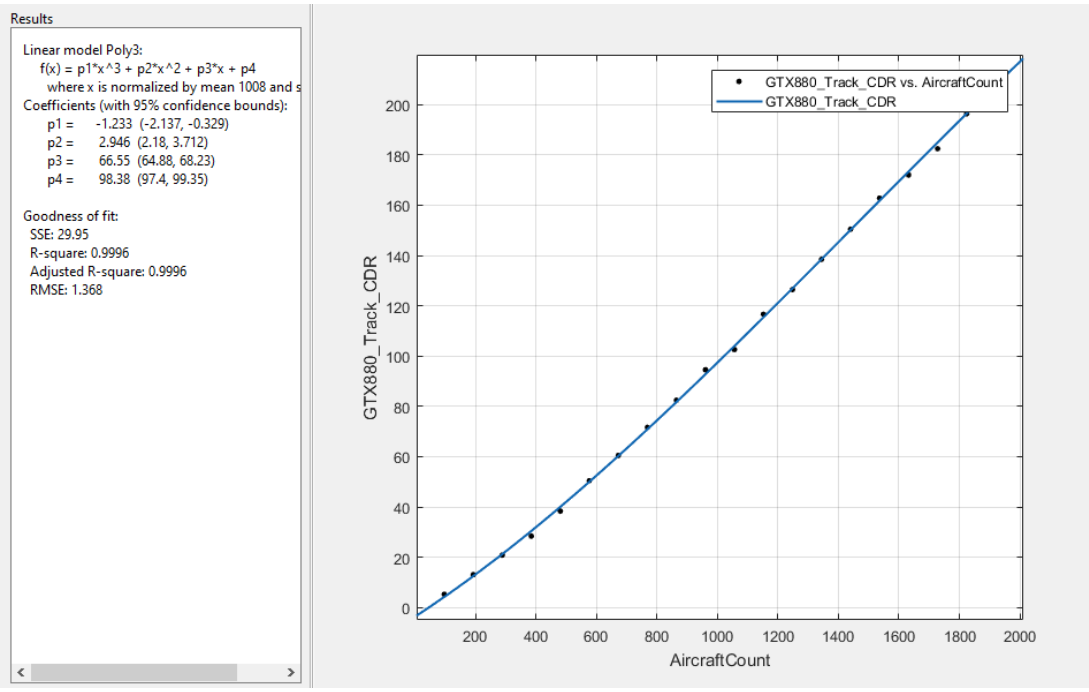

Figure 30: Tracking and correlation + collision detection and resolution cubic curve fitting for GTX 880M

5.3.7 Titan X (Pascal): Tracking and Correlation

The graphs below show the curve fitting results of the tracking and correlation task timing results for the Titan X (Pascal) device. The results here show a good linear fit, while the quadratic and cubic fits have negative coefficients. With these results, we have shown that we can get linear timings and SIMD-like behavior from all three of our NVIDIA-CUDA devices that we are using.

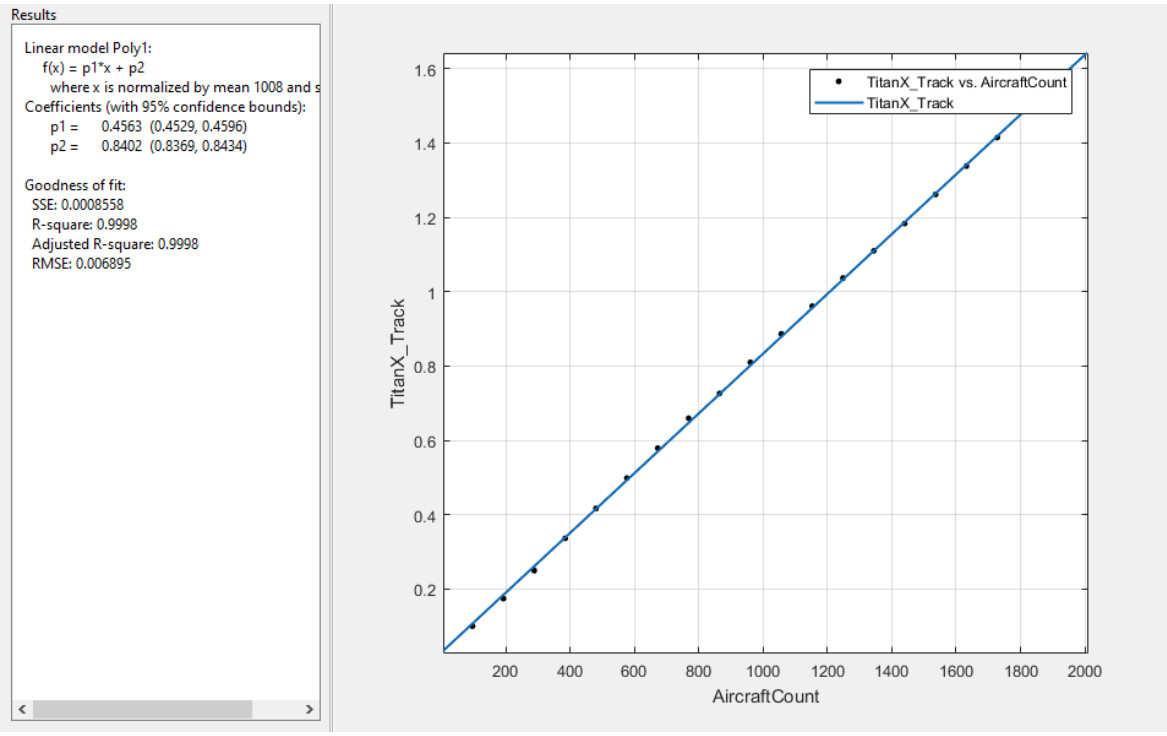

Figure 31: Tracking and correlation linear curve fitting for Titan X (Pascal)

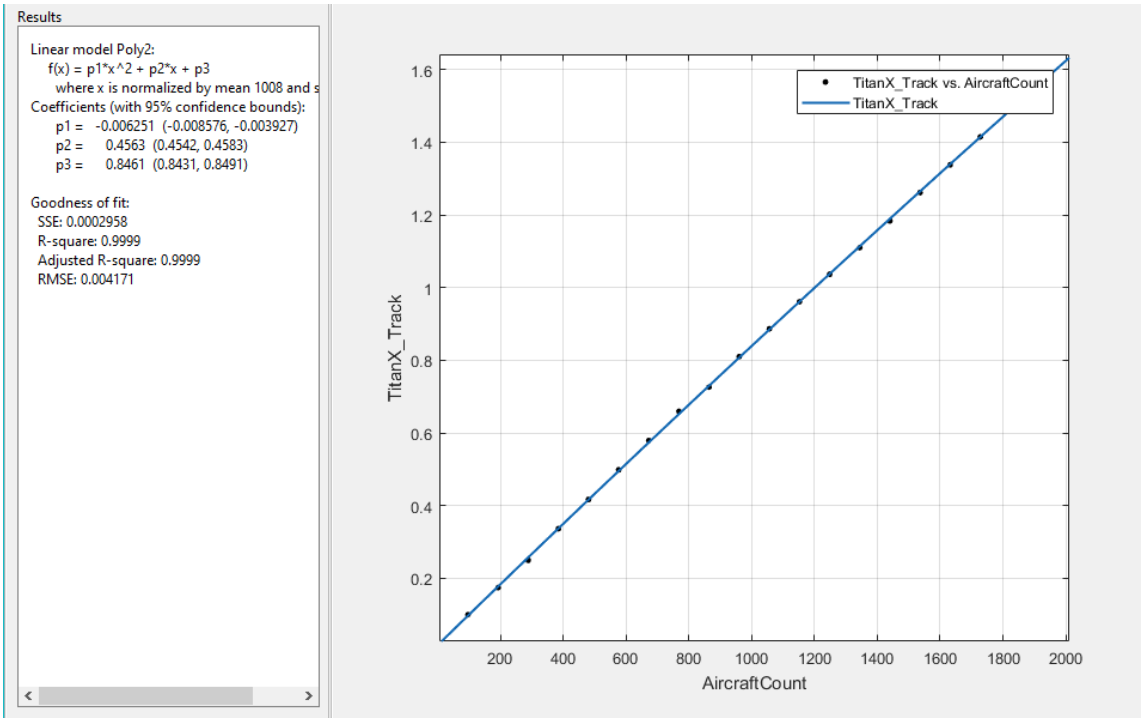

Figure 32: Tracking and correlation quadratic curve fitting for Titan X (Pascal)

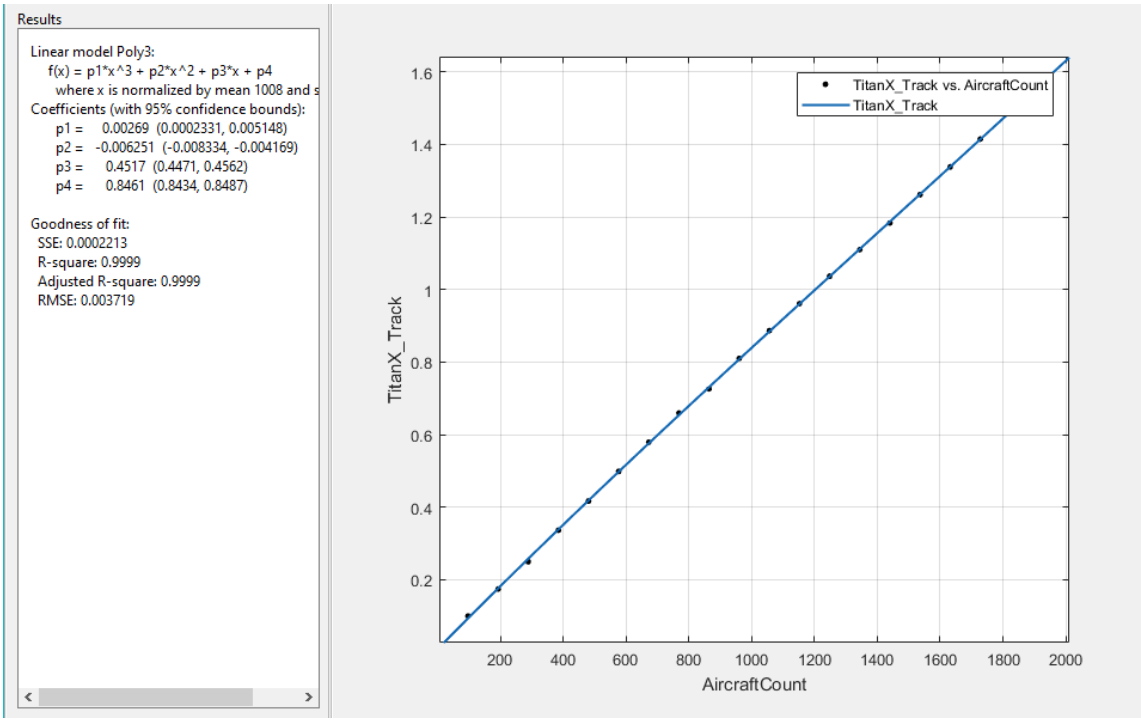

Figure 33: Tracking and correlation cubic curve fitting for Titan X (Pascal)

5.3.8 Titan X (Pascal): Collision Detection and Resolution

The graphs below show the curve fitting results of the collision detection and resolution task timings on the Titan X (Pascal). These results show a very good linear fit. While the quadratic fit has better "goodness of fit" values, and it's coefficient is not negative, the coefficient is still small enough to where this can be considered very close to linear by making the linear term dominate over the domain.

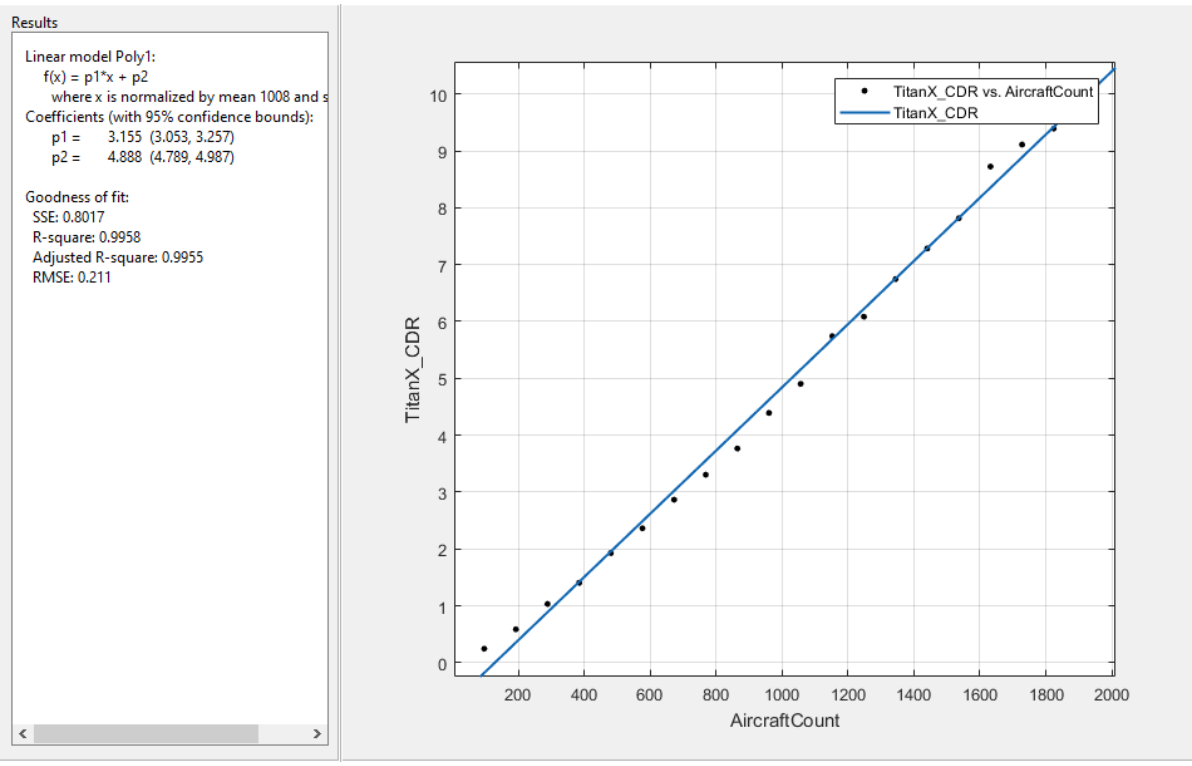

Figure 34: Collision detection and resolution linear curve fitting for Titan X (Pascal)

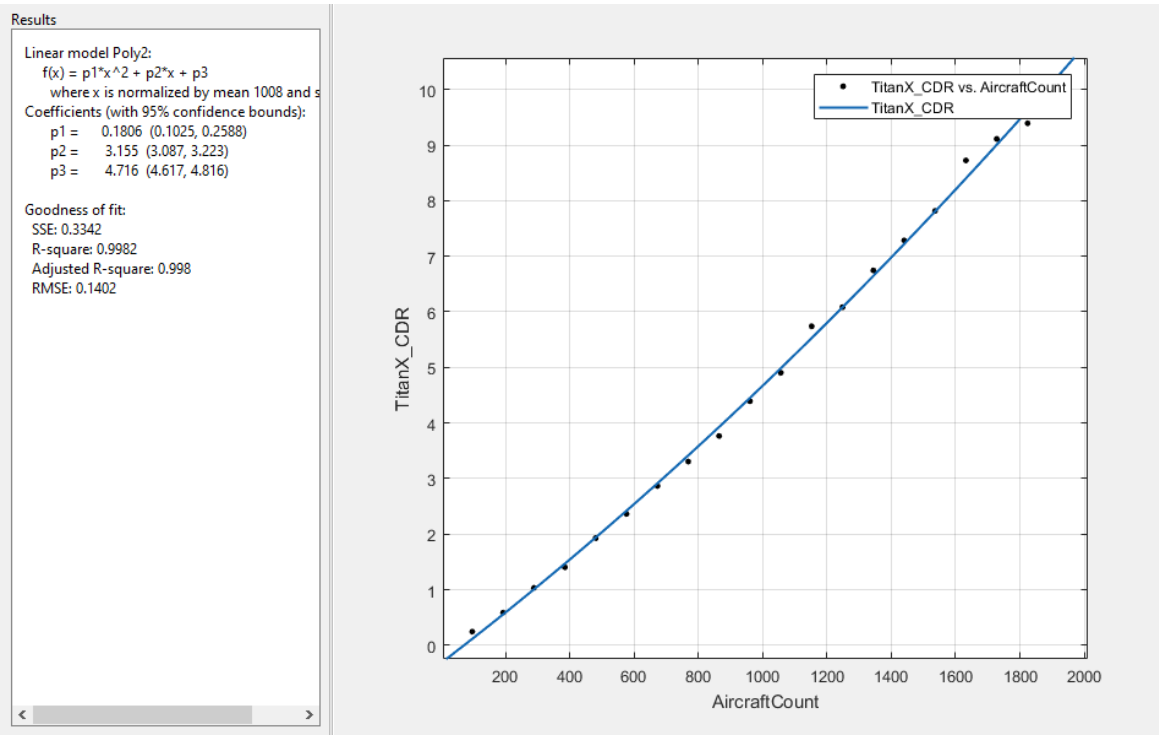

Figure 35: Collision detection and resolution quadratic curve fitting for Titan X (Pascal)

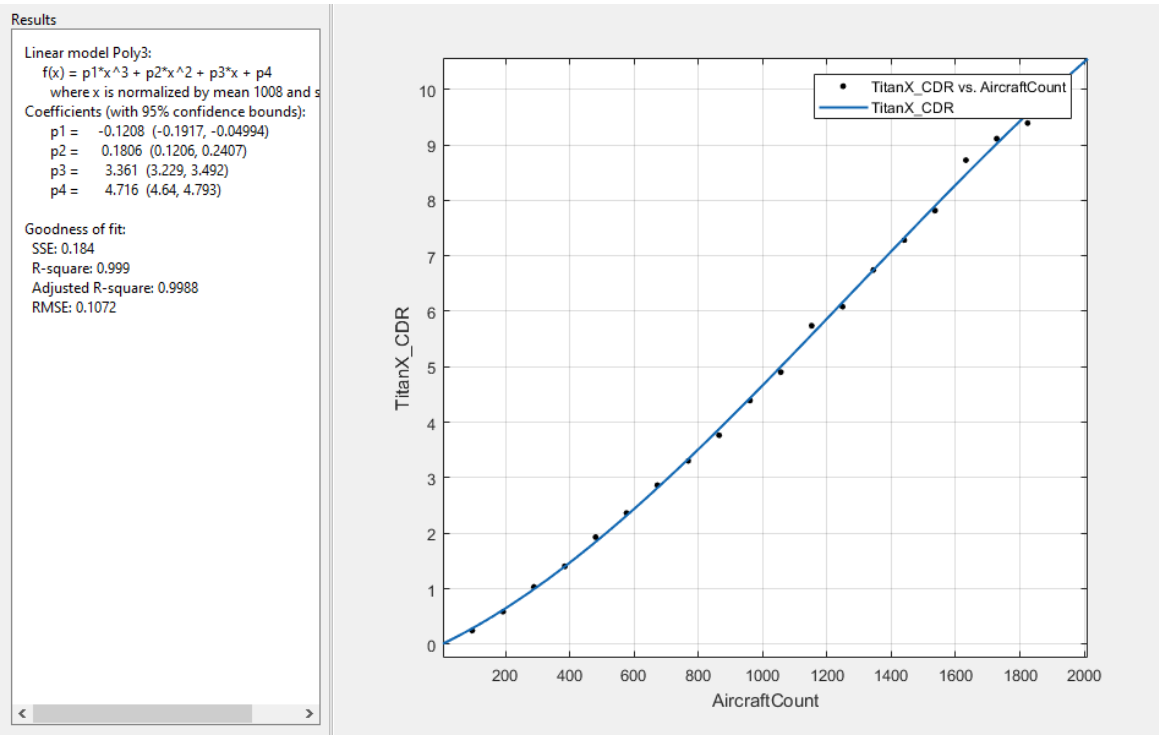

Figure 36: Collision detection and resolution cubic curve fitting for Titan X (Pascal)

5.3.9 Titan X (Pascal): Tracking and Correlation + Collision Detection and Resolution

The graphs below show the curve fitting results for the combined timings of the tracking and correlation task and the collision detection and resolution tasks. Due to the fit for the tracking and correlation task being linear, and the fit for the collision detection and resolution being quadratic but still very close to linear due to its small coefficient, we get results that show a very good linear curve, with very good "goodness of fit" values. However, the quadratic curve here again is shown to be a better fit according to the "goodness of fit" values, but the coefficient here again is small enough to consider this a more linear curve.

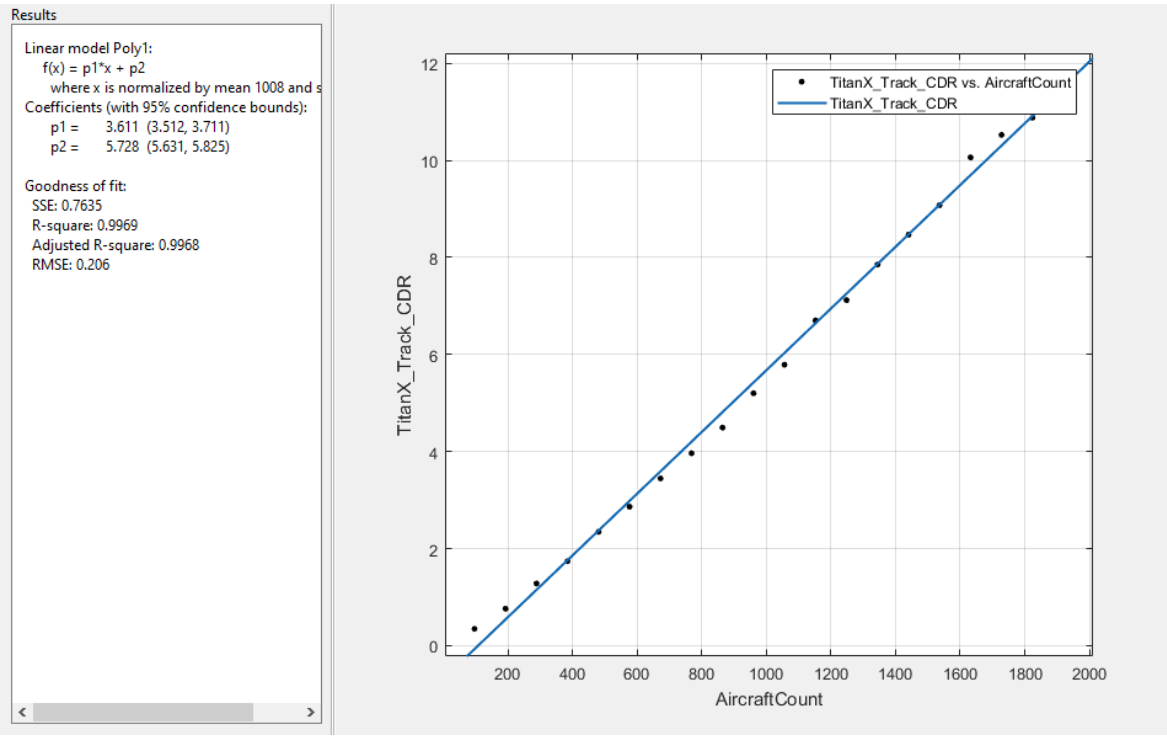

Figure 37: Tracking and correlation + collision detection and resolution linear curve fitting for Titan X (Pascal)

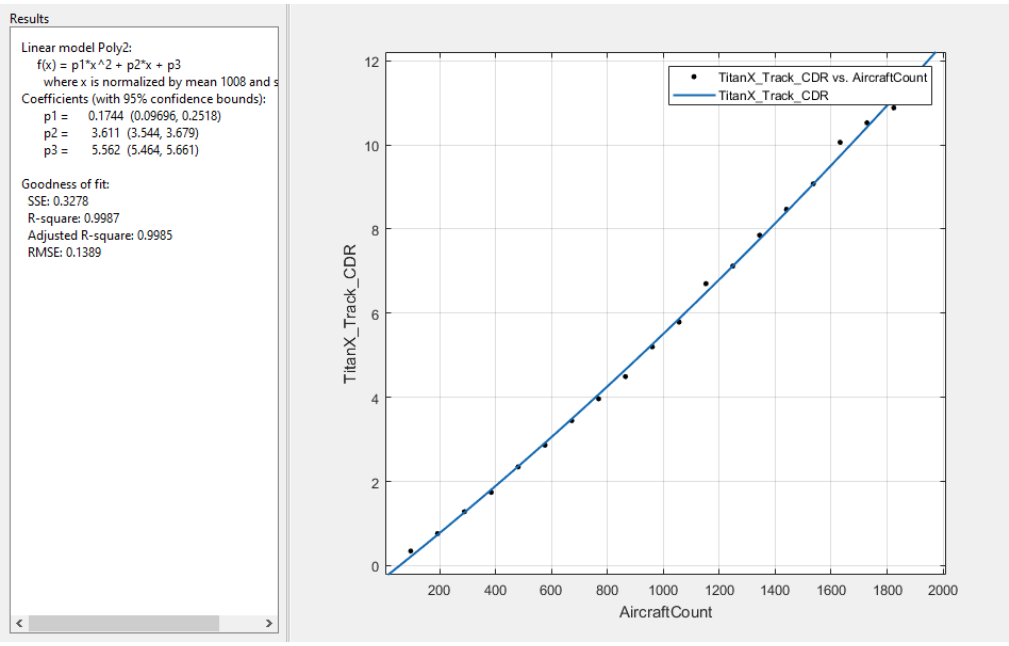

Figure 38: Tracking and correlation + collision detection and resolution quadratic curve fitting for Titan X (Pascal)

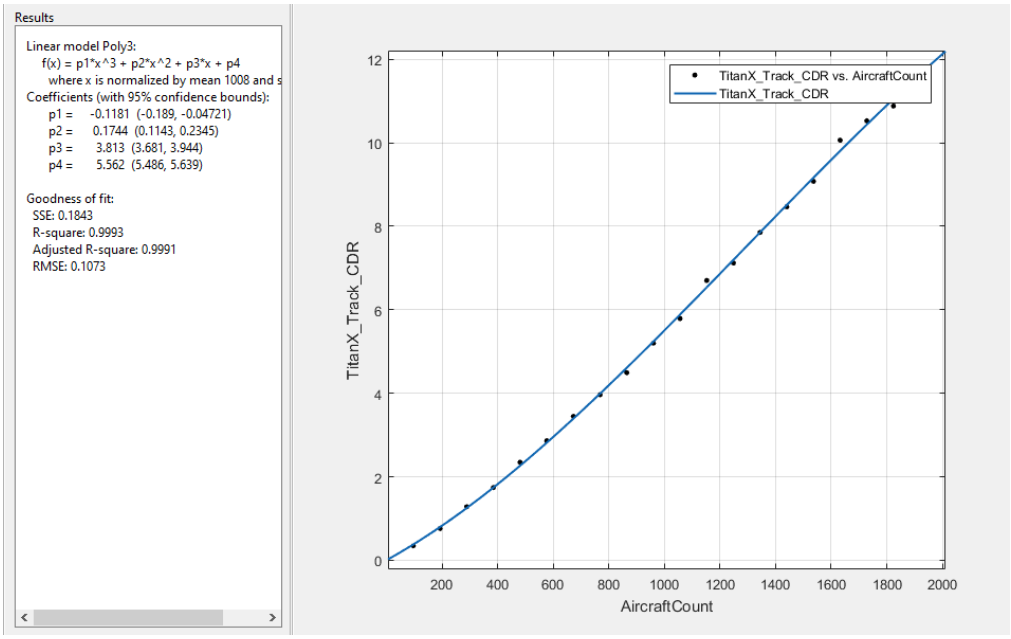

Figure 39: Tracking and correlation + collision detection and resolution cubic curve fitting for Titan X (Pascal)

5.4 Observations

The timing results that we acquired by implementing the three most compute-intensive air traffic control tasks on each of our NVIDIA-CUDA devices show that these NVIDIA-CUDA executions are obtaining SIMD-like timing. This means that the curve for its timing results were linear or very close, like quadratic. With the curve fitting data discussed earlier, we were able to find a fit on each machine that was linear, with the others being quadratic that lean more toward linear due to a small enough coefficient. 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 presented in this paper, due to continuous optimization of the program based on those timing results.

First, looking at each of the different graphs showing 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 machines that we are experimenting with are running these tasks much faster than both the AP (STARAN) and ClearSpeed (emulation of STARAN AP on CSX600 ClearSpeed) 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 need to 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 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 involves a large coefficient for the quadratic term. We cannot tell exactly 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 quicker and have the potential to be comparable to the AP (STARAN) in terms of efficiency when running real-time application such as ATC.

We further examined the nature of the curves of our timing results by using the program MATLAB that has a tool for curve-fitting that shows detailed results with four values that indicate a curve's "goodness of fit", as talked about earlier in this chapter. When we observe these timing results in the curve fitting tool, we can see that a lot of these have a linear curve, while others have a more quadratic curve. The curve for the GeForce 9800 GT's performance (figure number) 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 more linear than quadratic. The GTX 880M has a linear curve for its tracking and correlation timings (figure number) 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, showing that, due to the nature of the CUDA architecture, running the same code on different NVIDIA-CUDA devices ensures the same efficiency of the program across the devices. The runtime will differ according to the bandwidth and other

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

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

From the work done in this thesis, we are able to establish several things. We were able to develop a small real-time application that was around 900 lines of code (sequential + parallel) that consisted of three important compute-intensive basic ATC tasks. With that real-time application, we were also able to show that NVIDIA-CUDA can handle the most time-consuming of the ATC tasks without ever missing a deadline or starting too soon on major cycles. 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 the air traffic control application, 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 air traffic control 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 time to handle special situations like these. From our experimentation, we have found that the variation in time needed to handle various special situations like these to be no

larger than 5%, 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 linear and almost linear polynomial curves for our NVIDIA-CUDA timings for the number of aircraft tested.

This research shows that deterministic architecture 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.

6.2 Future Work

The results we have obtained from these experiments show that we can implement these ATC tasks on NVIDIA-CUDA devices without missing deadlines in almost linear time. We would like to build upon this work in the future. For example, we would like to implement all basic ATC tasks and create a more complete ATC 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 amount 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 of as many of these systems as possible. Obtaining or estimating the throughput of earlier systems such as the STARAN and ASPRO may be difficult. 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. Additionally, this information can be used to compare the efficiency with which each system utilizes it's throughput capacity in its computation of the basic ATC tasks. This comparison should provide more information about which of the computation methods are most efficient (e.g., NVIDIA-CUDA, ClearSpeed, and if necessary information can be obtained, the AP, i.e., STARAN or ASPRO). The ClearSpeed system can be used to emulate an AP and these execution results may provide additional information about the efficiency of an APs computation when compared to SIMDs and NVIDIA-CUDA systems. A multicore system executing earlier OpenMP code used by Mike Yuan in his dissertation could also be included as it will provide another useful comparison.

Another thing we are considering is to implement an OpenGL interpolation with CUDA to be able to provide some visualization capabilities. These could be used to support cockpit displays for pilots and a wider area of display for the air traffic control centers. Additionally, this may provide a more comprehensive visualization of the aircrafts in an airfield and provide a better understanding of how the algorithms are working in terms of controlling UASs or drones in terms of keeping the aircrafts steady and avoiding conflict. I believe this will help us develop a more efficient ATC application and further enhance the accuracy of our algorithms being able to visually observe the behavior of the aircrafts and find out if we need to improve on certain tasks or algorithms in the future.

We also intend on continuing to research the best curve fitting for the timing results that we get from the NVIDIA-CUDA implementation and see how the curve compares to the other implementations in more depth and detail. We will investigate it more extensively and see what kind of optimizations we can do to get better curves so that we end up having the most efficient algorithms possible. This would involve doing more in-depth experimentation with a much larger maximum number of aircraft, e.g., 32,000. This would give us a much larger set of data from which we can get a more accurate curve-fitting estimate for the execution graph. We also want to use this curve fitting data to provide further information about SIMD and MIMD execution efficiencies regarding real-time applications.

A longer term possible future research focus is to expand the implementation of basic ATC tasks on NVIDIA systems to being able to provide ATC for small groups of aircraft. This could be tested using the Kent State Air Traffic Control Laboratory in the College of Applied Engineering, Sustainability, and Technology (CEAST). This work could be used to provide a mobile ATC center in remote areas where sufficient number of UASs or drones were being used to need ATC control. 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. To pursue this research, we would probably need to partner with one or more faculty whose expertise was in aviation and drones. Again, the Air Traffic Control Laboratory in CEAST would be an important resource in this project.

References

1. M. Yuan, J. Baker, W. Meilander, "Comparison of air traffic control on an associative processor with a MIMD and consequences for parallel computing," J. Parallel Distrib. Comput. Vol 73, pp 256-272, 2013.

2. J. Baker, Parallel & Real-Time Systems Slides, 2014, Available at: www.cs.kent.edu/~jbaker/PRTS-Sp13/ | www.cs.kent.edu/~jbaker/PRTS-F14

3. J.Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, C. Asthagiri, "Asc: An associative-computing paradigm," Computer, vol. 27, no. 11, pp. 19–25, 1994.

4. M. Yuan, "A SIMD approach to large-scale real-time system air traffic control using associative processor and consequences for parallel computing," Ph.D. Thesis, Department of Computer Science, Kent State University, 2012.

5. B. Parhami, "SIMD Machines: Do they have a Significant Future?," Report on a Panel Discussion at Frontiers '95, Available: https://www.ece.ucsb.edu/~parhami/pubs_folder/parh95-can-simd-future.pdf

6. G. Buttazzo, "Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications" Third Edition, Springer, 2011.

7. M. Nolan, "Fundamentals of Air Traffic Control", Fifth Edition, International Code Council, 2011

8. "NVIDIA CUDA C Programming Guide", Version 4.2, 2012, Available at: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

9. M. Yuan, Unpublished document containing additional information about his dissertation notes and his code, 2012.

10. Mathworks,“Evaluating Goodness of Fit”, 2017, Available at: https://www.mathworks.com/help/curvefit/evaluating-goodness-of-fit.html