GPU-Accelerated Data Mining with Swarm Intelligence

by

Robin M. Weiss

Honors Thesis

Department of Computer Science

Macalester College

May 2010
Abstract

Swarm intelligence describes the ability of groups of social animals and insects to exhibit highly organized and complex problem-solving behaviors that allow the group as a whole to accomplish tasks which are beyond the capabilities of any one of the constituent individuals. This natural phenomenon is the inspiration for swarm intelligence systems, a class of algorithms that utilizes the emergent patterns of swarms to solve computational problems. Recently, there have been a number of publications regarding the application of swarm intelligence to various data mining problems, yet very few consider multi-threaded, let alone GPU-based implementations. In this paper we adopt the General-Purpose GPU parallel computing model and show how it can be leveraged to increase the accuracy and efficiency of two types of swarm intelligence algorithms for data mining. To illustrate the efficacy of GPU computing for swarm intelligence, we present two swarm intelligence data mining algorithms implemented with CUDA for execution on a GPU device. These algorithms are: (1) AntMinerGPU, an ant colony optimization algorithm for rule-based classification, and (2) ClusterFlockGPU, a bird-flocking algorithm for data clustering. Our results indicate that the AntMinerGPU algorithm is markedly faster than the sequential algorithm on which it is based, and is able to produce classification rules which are competitive with those generated by traditional methods. Additionally, we show that ClusterFlockGPU is competitive with other swarm intelligence and traditional clustering methods, and is not affected by the dimensionality of the data being clustered making it theoretically well-suited for high-dimensional problems.
Acknowledgements

I am extremely grateful for the past four years I have spent as an undergraduate student in the computer science department of Macalester College. The challenges I faced over the past years have allowed me to identify what truly interests me intellectually and as I move forward in my academic career. I would like to begin by thanking my academic advisor Professor Susan Fox for her guidance along the way. Also, many thanks to Professor Elizabeth Shoop for her input and advice throughout the writing process of this paper.

Next, I would like to thank my advisor from the University of Minnesota, Dr. David A. Yuen, for whom I give credit to for introducing me to the world of high-performance and GPU-based computing. Dr. Yuen’s approach to advising offered a mixture of independence and direction which has truly allowed me to grow by leaps and bounds over the past year. My profound thanks are to Dr. Yuen for his guidance and for providing me the resources and contacts I needed to complete this work.

Additionally, I would like to thank Professors Witold Dzwinel and Marcin Kurdziel of AGH University for their input during both the very first, and very last phases of this project. Finally, I would like to thank Professor Michael Steinbach of the University of Minnesota for his critiques of my approach and for assisting me as I took my first steps into the world of data mining and machine learning.
We are drowning in information but starved for knowledge.

John Naisbitt
Contents

Abstract ii

Acknowledgements iii

1 Introduction 1
   1.1 Background and Problem Statement ............................................. 1
   1.2 Organization of This Paper ....................................................... 2

2 Introduction To Swarm Intelligence 4
   2.1 Swarm Intelligence Overview ..................................................... 4
   2.2 Two Types of Swarm Intelligence Systems ....................................... 6
      2.2.1 Ant Colony Optimization Algorithms ........................................ 6
      2.2.2 Flocking Algorithms ......................................................... 9

3 Parallel and GPU Computing 11
   3.1 Parallel Computing Overview .................................................... 12
   3.2 GPU Performance and Hardware Architecture .................................... 14
   3.3 CUDA for GPU Computing ........................................................ 17
   3.4 Swarm Intelligence and GPU Computing ......................................... 18
# Data Mining Tasks, Techniques, and Applications

4.1 Classification

4.1.1 Applications of Rule-Based Classification

4.1.2 Traditional methods for Classification

4.2 Partitional Cluster Analysis

4.2.1 Applications of Data Clustering Algorithms

4.2.2 Traditional Partitional Data Clustering Techniques

# The AntMinerGPU Algorithm

5.1 Ant Colony Optimization

5.1.1 Background and Technical Details

5.1.2 The MAX-MIN Ant System

5.1.3 Ant Colony Optimization and GPGPU

5.2 The AntMinerGPU and AntMiner+ Algorithms

5.2.1 The AntMiner+ Algorithm

5.2.1.1 Transition Rule

5.2.1.2 Heuristic Value

5.2.1.3 Pheromone Updating

5.2.1.4 Early Stopping

5.2.2 The AntMinerGPU Algorithm

5.3 Results and Analysis

5.3.1 Data Sets

5.3.2 Test System

5.3.3 Performance Analysis
5.3.4 Comparison to Traditional Methods . . . . . . . . . . . . . . . . . . . 51
5.3.5 Comparison to AntMiner+ . . . . . . . . . . . . . . . . . . . . . . . . 52
5.4 Next Steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.4.1 Multi-Class Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.4.2 Multiple Colonies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.4.3 Improved Convergence Detector . . . . . . . . . . . . . . . . . . . . . 54
5.4.4 Better Use of Shared Memory . . . . . . . . . . . . . . . . . . . . . . 55
5.4.5 Very Large Data Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

6 The ClusterFlockGPU Algorithm 56

6.1 Swarm Intelligence Algorithms for Data Clustering . . . . . . . . . . . . . . 57
6.1.1 Particle Swarm Optimization for Cluster Analysis . . . . . . . . . . . . . 57
6.1.2 Ant Colony Algorithms for Cluster Analysis . . . . . . . . . . . . . . . 59
6.2 Flock Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
6.3 A Flock Algorithm for Data Cluster Analysis . . . . . . . . . . . . . . . . . . 62
6.3.1 Similarity Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.4 The ClusterFlockGPU Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 65
6.5 Results and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.5.1 Data Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.5.2 Test System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.5.3 Performance Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.5.4 Comparison to Other Methods . . . . . . . . . . . . . . . . . . . . . . . 71
6.6 Next Steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.6.1 Improved Neighbor Detection . . . . . . . . . . . . . . . . . . . . . . . 72

vii
Chapter 1

Introduction

1.1 Background and Problem Statement

Swarm intelligence describes the phenomenon where highly organized and often beneficial global behaviors emerge from the individual actions of a group of decentralized and self-organizing agents. In the past decade, the field of swarm intelligence has become a hot topic in the areas of computer science, collective intelligence, and robotics. To date, swarm intelligence algorithms have been shown to be able to tackle a wide range of hard optimization problems (including the Traveling Salesman, Quadratic Assignment, and Network Routing Problems) and there is a myriad of documented applications of swarm intelligence to computational problems of all sorts [9, 11, 10].

Swarm intelligence algorithms for data mining have been shown to be competitive with traditional techniques [47, 33, 28, 26, 22, 21, 7], and some even provide useful features not found in other methods [13]. However, we find that most of these swarm intelligence data mining algorithms are relatively slow in comparison to traditional ones.
Swarm intelligence systems are characterized by groups of independent agents collectively working to solve some problem and, as such, there is a large amount of implicit parallelism. Interestingly, a survey of the literature reveals that few authors consider parallel, let alone GPU-based implementations to boost the performance of their algorithms. In this paper, we adopt the so-called General-Purpose GPU (GPGPU) computing model and show how it can be applied to swarm intelligence algorithms for data mining to achieve better results and considerable speedups as compared to CPU-based implementations. To this end, we present two GPU-based swarm intelligence data mining algorithms: an ant colony optimization algorithm for rule-based classification (AntMinerGPU), and a flock algorithm for partitional cluster analysis (ClusterFlockGPU).

Our results indicate the AntMinerGPU algorithm produces more accurate classifications than traditional methods on some data sets, and may offer a speedup of 20-50x over the CPU-based AntMiner+ algorithm on which the AntMinerGPU algorithm is based. We also show that the ClusterFlockGPU algorithm is competitive with other swarm intelligence clustering algorithms and the traditional k-means algorithm, and that this algorithm’s running time is unaffected by the dimensionality of the data being clustered. While the focus of this paper is on swarm intelligence algorithms for data mining, we also hope to underscore the fact that swarm intelligence algorithms of all sorts are extremely well-suited for the GPGPU computing model.

1.2 Organization of This Paper

The remainder of this paper is organized as follows. Chapter 2 explores swarm intelligence systems in general, giving a high-level overview of their operations and characteristics. Chap-
ter 3 provides background on the two data mining problems our algorithms are intended to solve: classification and clustering. Chapter 4 gives an overview of the current state of parallel computing, providing background on GPGPU computing with NVIDIA's CUDA toolkit and discussing how this parallel computing model can be utilized to improve the performance of swarm intelligence algorithms. Chapter 5 gives a detailed description of the generic ant colony optimization technique and presents our AntMinerGPU algorithm for rule-based classification. Chapter 6 discusses flock algorithms and presents our ClusterFlockGPU algorithm for partitional data clustering. Finally, Chapter 7 concludes this paper with a summary of concepts presented and possible directions of future research.
Chapter 2

Introduction To Swarm Intelligence

In this chapter we give a brief overview of the history of swarm intelligence systems and describe two classes of swarm intelligence algorithms: ant colony-based and flock-based. Section 2.1 offers an overview of swarm intelligence, explaining the benefits of the approach and the general features of this problem-solving technique. Section 2.2 describes two types of swarm intelligences: that found in colonies of ants, and that found in flocks of birds. These two types of swarm intelligence will be revisited when we present our AntMinerGPU and ClusterFlockGPU algorithms for data mining in Chapters 5 and 6, respectively.

2.1 Swarm Intelligence Overview

Swarm intelligence describes the ability of groups of decentralized and self-organizing agents to exhibit highly organized behaviors. These global behaviors of swarms often allow the swarm as a whole to accomplish tasks which are beyond the capabilities of any one of the constituent individuals. Following the publications of two works, “Swarm Intelligence” [17] and “Swarm Intelligence: From natural to artificial systems” [4], the area of swarm
intelligence became a hot topic in the fields of computer science, collective intelligence, and robotics. Today, the number of successful applications of swarm intelligence continues to grow.

The term “swarm intelligence” was coined in 1989 by Gerardo Beni and Jing Wang in the context of cellular robotic systems [3]. Beni and Wang proposed that groups of simple robotic agents could be programmed to collaboratively solve difficult tasks and described how collectively intelligent behaviors can be exhibited in systems of non-intelligent robots. The authors showed that an artificial system so organized would present three distinct advantages:

1. Agents need not be complex since they will work together to solve problems
2. The overall system is very reliable due to high levels of redundancy
3. Problems are solved in parallel since each agent handles a portion of the problem

These three characteristics are present in nearly all swarm intelligence systems and are what makes swarm intelligence approaches to problem-solving attractive for certain problems. In addition to these general characteristics, nearly all swarm intelligence systems exhibit the following features:

**Homogeneity:** Every member of the swarm follows the same rules and decision making processes.

**Locality:** Actions and decisions of individuals are made on the basis of purely local information and of what agents learn via (direct or indirect) communication with others.

**Randomness:** Swarm members introduce randomness into their decision-making processes in order to explore new solutions.
Positive Feedback: As with Darwinian evolution, “good” solutions that emerge from the actions of swarm agents are identified as having good “fitness” and are reinforced over time.

To further illustrate the characteristics of swarm intelligence, we provide an overview of two classes of swarm intelligence systems in the following section. For a more complete exploration of the patterns of swarm intelligence systems and the theoretical underpinnings thereof, the reader is referred to [17] and [4].

2.2 Two Types of Swarm Intelligence Systems

Nearly all swarm intelligence systems take their inspiration from the behaviors exhibited by social animals or insects in nature. The wide range of behavioral patterns (bird flocking, ant colonies, fish schooling, herding, etc...) found in nature has allowed for many types of swarm intelligence algorithms to be proposed, each well suited for a different type of target problem. For example, the ant colony and flock algorithms presented in Chapters 5 and 6, respectively, differ in terms of how data is represented in the virtual environment and how the actions of swarm agents are interpreted. We give an overview of these contrasting paradigms in swarm intelligence in the following two subsections.

2.2.1 Ant Colony Optimization Algorithms

Perhaps the most well explored swarm intelligence algorithms are ant colony optimization (ACO) methods for discrete optimization. The basic operating principles of these algorithms is given in Figure 2.1.
In ACO, swarm agents make explorations of the solution space of a target problem in an attempt to locate the optimal solution. A solution is defined by the path traveled by an ant agent through solution space. After each iteration, agents evaluate their solution with respect to an objective fitness function. If an agent finds a good solution, it will communicate with others in the swarm to alert them to a potentially rich area of the solution space. By communicating the relative success or failure to find a good solution to others, agents in subsequent iterations are more or less inclined to explore a similar region of the solution space. Over time and with the introduction of random variations in the movements of agents, this repeated exploration of solution space will often lead to a convergence of the swarm as a whole to the optimal solution.

As an example of how this approach can be applied to a discrete optimization problem we describe here the general approach of an ACO method for solving the Traveling Salesman Problem (TSP). This problem, as implied by the name, can be conceived of as the problem faced by a salesman who, departing from his home town, wishes to find the shortest possible
route through a number of customer towns, visiting each exactly once and then returning home. Formally, the task presented by the TSP is to find the minimum-cost, Hamiltonian path through a fully-connected weighted graph.

To solve this problem with ACO, a population of ants is generated in the problem graph with each ant starting at a random city (or node). Then, all of the ants incrementally add cities to their tour until all cities have been visited exactly once. When all of the ants have generated a candidate tour, the ant with the shortest tour is allowed to deposit pheromone on the path it took. This increased level of pheromone will cause ants in subsequent generations to follow a similar path, but due to the probabilistic nature of the path finding process of the ants, random variations on this “good” path will be generated (some of which will be better, and some worse). Repeating this process usually leads to the optimal path being found. Figure 2.2 shows how this process proceeds.

Figure 2.2: The progression of pheromone trails laid by ants while solving the Traveling Salesman Problem. Pheromone levels are indicated by edge thickness.
Figure 2.2(a) shows the initial construction graph with no pheromone on edges. After the first group of ants explore the graph, the best path is reinforced with pheromone as shown in Figure 2.2(b). This increased level of pheromone causes ants in the next iteration to explore a similar path. Figure 2.2(c) shows that a better path has been found and reinforced with pheromone. This path is then further reinforced in Figure 2.2(d), and finally taken as the optimal solution in Figure 2.2(e).

In Chapter 5, we show that the task of rule-based classification can be translated into a problem of finding “good” paths through a graph. With this conception of the classification problem, we show how the ability of ant colonies to find good paths through an environment can be used to generate classification rules.

### 2.2.2 Flocking Algorithms

The basis for nearly all flock algorithms is derived from the work of Craig Reynolds and his seminal work “Flocks, Herds, and Schools: A Distributed Behavioral Model” [36]. Here, Reynolds proposed that lifelike computer simulations of bird flocks could be achieved by having each member of the flock adhere to a certain set of “flocking rules.” These rules, computed independently by each flock member, keeps the virtual birds in a cohesive flock (flock centering), moving in the same direction (velocity matching), and prevents collisions with one another or objects in the environment (collision avoidance). Every bird calculates vectors which satisfy these rules and adds them to its current velocity to attain its velocity for the subsequent time step. The general flow of this basic flocking algorithm is given in Figure 2.3.

To be clear, swarm intelligence models based on flocking behaviors interpret the actions
of agents in a different manner than in ACO-style algorithms. With flocking algorithms, swarm agents do not exist in the solution space of the target problem, but instead in their own virtual environment and follow well-defined rules that govern their movements. In these models, agents themselves often represent data objects and solutions to the target problem are defined by patterns in the agents’ arrangements which emerge as they interact with one another and the environment.

A proposed application of this type of swarm intelligence algorithm is for the navigation of groups of unmanned areal vehicles (UAVs). The main idea is that a group of UAVs could use a flocking algorithm to coordinate their movements and maintain cohesion while avoid collisions as they fly to their destination.

We will revisit flocking algorithms in more detail in Chapter 6 when we explore a flocking algorithm that can be used to perform partitional data cluster analysis.
Chapter 3

Parallel and GPU Computing

The two algorithms we present in this paper, AntMinerGPU and CluserFlockGPU, rely on the parallel architecture of GPU devices to achieve their high levels of performance. As such, we feel it is important to provide background on CPU- and GPU-based parallel computing in order to highlight why we adopt a GPU-based approach here. This chapter will give an overview of the recent advances in parallel computing devices and will introduce NVIDIA’s CUDA programming model for General-Purpose GPU (GPGPU) computing. Following this introduction, we will highlight the features of swarm intelligence algorithms that make them well-suited for GPGPU computing and show how this computing model can be used to increase overall performance. Much of the information presented in this chapter is a recapitulation of that found in the introductory sections of David Kirk and Wen-mei Hwu’s book “Programming Massively Parallel Processors: A Hands-On Approach” [19]. The reader is referred to this work for a more information on the concepts presented here.
3.1 Parallel Computing Overview

With the release of each new version, conventional single-core microprocessors have, for the past few decades, steadily brought about performance increases for computer applications. These steady increases in CPU and memory speeds have allowed computer software developers to consistently create applications with better functionality, user interfaces, and overall performance. Today, computer systems built atop a single-core CPU have been able to achieve performance levels on the magnitude of a billion (giga) floating point operations per second (GFLOPS).

During the era of steady improvement of CPU and memory devices, application developers relied heavily on the improvements of computing hardware to increase the speed of their applications and algorithms. Simply put, the same application run on a computer with a faster CPU and faster memory will (most likely) run faster. However, since approximately 2003, the steady increases in CPU clock speeds and transistor counts have slowed. This is due in large part to engineers’ inability to overcome heat dissipation and power consumption issues presented by such high clock rates and transistor densities. The steady increase in computing performance predicted by Moore’s law has essentially come to an end.

In the past 5 years, the market has seen nearly all microprocessor manufactures adopting a design model where multiple processing units, or cores, are used in each chip to increase overall processing power. As manufacturers began adopting parallel architectures in their chips, two major design paradigms emerged: multi-core and many-core. Multi-core processors are characterized by a processing unit containing a relatively small number of “heavy-weight” cores. For example, the Intel Core i7 CPU contains four cores, each a fully functional processor with a wide range of capabilities and a very large instruction set. Generally, multi-core
processors are designed to maximize the performance of sequential code, but also allow for true hardware-level parallelism (albeit on a relatively small scale).

On the other hand, the many-core design paradigm is focused on the optimized execution of parallel code. Many-core processors are characterized by a processing unit comprised of a large number of “light-weight” cores. For example, the NVIDIA GeForce GTX 285 GPU contains a total of 240 light-weight processing cores each with very limited instruction sets and optimized for basic floating point operations.

As shown in Figure 3.1, many-core processors, and specifically GPU devices, lead the way in peak computing performance. It should be noted that the performance levels shown in Figure 3.1 are very rarely achieved by real-world applications, but are speeds theoretically attainable by such devices. An explanation of this large divergence in peak performance is offered in the Section 3.2 when the hardware architecture of modern GPU devices is discussed.

![Figure 3.1: Comparison of GPU and CPU peak GFlop performance](image-url) [29]
With such large amounts of raw computational power theoretically attainable with GPU devices, a growing number of computer scientists have begun porting their algorithms to GPU-based computing systems. In the past 3 years, NVIDIA, one of the largest manufacturers of GPU devices in the market, has been very vocal in promoting the GPU-based approach to parallel computing and maintains a repository of examples on its website ([30]). To date, the CUDA Community Showcase contains over 1000 examples of GPU-based algorithms for a wide range of computing tasks, each reporting considerable speedups (many on the order of 100x) over sequential implementations [30]. With such large amounts of FLOPS achievable with GPU devices, the popularity of GPGPU computing continues to increase. Recently, Oak Ridge National Laboratory announced it will be constructing a new GPU-based supercomputer which is expected to be 10-times more powerful than today’s fastest supercomputer and manufacturers such as Cray have begun including GPU-devices in their next-generation systems.

3.2 GPU Performance and Hardware Architecture

The exponentially increasing performance levels of GPU devices has largely been driven by the video game industry. Interactive 3D video games demand a very high level of data throughput and an absolutely staggering number of floating-point operations per second. Consider that for a SXGA (1280x1024) display, there is a total of \( \sim 1.3 \text{Million} \) individual pixels. With a commonly desired frame rate of 30fps, there is a worst-case scenario of having to compute \( >39 \text{Million} \) pixel-values every second with each value requiring multiple floating point and memory read/write operations. With such high demands for throughput, programmers in the computer graphics community adopted thread-level parallelism as the
dominant paradigm for producing satisfactory results and GPU manufactures followed suit by building hardware that could realize the performance benefits of this paradigm. A very high-level overview of CPU and GPU architecture is given in Figure 3.2.

As shown in the above figure, NVIDIA’s GPU devices are organized as an array of streaming multiprocessors (SMs). These SMs are comprised of a number of stream processors (SPs), each capable of simultaneous execution of many hundreds of threads. Because these SPs are grouped into SMs, the threads executing on the SPs of a single SM are able to cooperate and share instruction cache and control logic, as well as a relatively small amount of on-chip, low-latency memory.

The DRAM on GPU devices provides up to 4GB of storage capacity with very high bandwidth to the SMs (102 GB/sec for the DRAM of the Tesla C1060). It should be
noted however that while this DRAM has comparatively higher bandwidth than system DRAM, latency is also greater. As such, GPU-based applications always attempt to minimize memory operations that access DRAM, instead taking care to intelligently store frequently accessed data in the on-chip “shared memory” of each SM.

Achieving high levels of floating-point performance on a GPU device is possible only when an algorithm employs a massive number of concurrent threads. This way, if a thread stalls due to memory latency, another thread can be scheduled for immediate execution on that ALU while the first thread waits for its memory operation to complete. In this fashion, the very large amount of chip-space dedicated to floating-point calculations is kept saturated with concurrently executing threads and the overall effect is a very high throughput of floating-point operations.

To be clear, GPU devices are designed to be floating-point calculating machines and are thus not ideal for certain types of computing tasks. Complex control and branch structures are better suited for the complex control logic and large instruction sets found in CPU devices. Also, due to the relatively high latency of GPU DRAM as compared to that of CPU memory, applications that require large amounts of memory read/writes are also better suited for CPU devices. Realizing this, the CUDA programming model from NVIDIA is specifically designed to allow for a mixture of GPU and CPU code execution. It should be noted that swarm intelligence algorithms are characterized by the repeated application of the same simple decision making process by multiple independent agents and thus do not require a large amount of complex control structures. Because of this feature, the limited instruction sets of GPU devices will theoretically not impact the performance of swarm intelligence algorithms.

As detailed in Chapter 2, swarm intelligence systems are characterized by groups of
independent agents working asynchronously and collectively to solve a problem. It can easily be envisioned how an asynchronous system such as a swarm can be quite accurately modeled with a GPU-based algorithm by allocating one thread per swarm agent. With this implementation strategy, each agent would be allocated its own private “brain” and virtual swarms would more closely resemble the asynchronous natural swarms on which they are based. This concept is further explored in Section 3.4 as we examine some specific ways in which the GPU computing model can be leveraged to create high-performance swarm intelligence algorithms.

3.3 CUDA for GPU Computing

The idea of using GPU hardware for general purpose computing is a concept that dates back nearly two decades. In the early days, utilizing GPU devices for the execution of non-graphics related algorithms was a very difficult task. Essentially, programmers were forced to use graphics application programming interfaces (APIs) such as OpenGL or Direct3D to gain access to the GPU chip.

The main strategy for doing GPGPU computing was to find clever ways to fit some target algorithm into computer graphics abstractions that are compatible with the graphics API being used. For example, large matrices were translated into images and by casting a general mathematical problem into computer graphics abstractions, data could be loaded into GPU texture memory and standard computer graphics functions (mostly matrix-matrix operations) could be applied.

The process of doing GPGPU computing changed dramatically with the release of NVIDIA’s CUDA toolkit in 2007. In the early 2000s, NVIDIA realized that the interest in using GPU
devices for general purpose computing was growing and wished to capitalize on this emerging market. The goal was to provide a specially extended version of some high-level programming language that would allow programmers to gain direct access to GPU devices without needing in-depth knowledge of computer graphics algorithms and techniques. The result was the CUDA toolkit, a software development kit (SDK) that allows programmers to write parallel code in a specially extended version of the C programming language for parallel execution on most modern NVIDIA GPUs. This SDK allows for a much more generic parallel programming model than was possible in earlier generation GPUs and allows programmers to use common parallel programming abstractions such as parallel threads, barrier synchronization, and atomic operations in GPU-based code.

This paper assumes a familiarity with the CUDA programming model. More information regarding the technical details of CUDA as well as a simple example CUDA program can be found in Appendix A.

In the next section, we describe how the parallel programming abstractions offered by CUDA can be applied to swarm intelligence algorithms. The concepts presented in Section 3.4 will be revisited in greater detail and specificity when we describe our AntMinerGPU and ClusterFlockGPU algorithms in Chapters 5 and 6, respectively.

3.4 Swarm Intelligence and GPU Computing

This project proposes that swarm intelligence systems are well suited for implementation on GPGPU devices. We now consider the main characteristics of the swarm intelligence algorithms outlined in Chapter 2 with respect to the GPU architecture and programming model outlined in the previous sections.
Ant colony optimization algorithms rely on a large number of explorations of solution space and the law of large numbers to converge the swarm to the optimal solution. Thus, if a larger number of explorations of solution space can be carried out at the same time (in turn necessitating larger swarm populations), more candidate solutions can be generated and there will be a greater chance that the optimal solution will be found (and in less time). Generally, in sequential implementations the population of a swarm has a direct, and often very large, impact on overall running time. This makes extremely large populations of swarm agents infeasible due to the associated complexity. We propose that the massive multi-threading capabilities of GPU devices can be used to achieve much larger populations of swarm agents and in turn a much larger number of solution space explorations without severely degrading running time. This hypothesis is supported by [2] where the authors present a GPU implementation of the MAX-MIN Ant System and report a speedup of over 2x compared to a CPU-based implementation.

To illustrate, we consider a very generalized ACO algorithm and show how it might be parallelized for GPU implementation. Listing 3.2 shows a pseudocode overview of the generalized ACO algorithm.

```
generate N ant agents
until (no improvement in solution quality for X iterations):
  for 1 to N:
    generate a candidate solution;
    evaluate solution quality;
    deposit amount of pheromone proportional to quality on solution;
  done
done
extract final solution
```

Listing 3.2: Generalized sequential ACO algorithm
There is clearly a large amount of implicit parallelism in this algorithm since by definition, all $N$ ant agents have to complete the same tasks each iteration. Because of this, we can easily fit this algorithm into the SPMD execution pattern of CUDA.

A straightforward way of parallelizing the above algorithm for execution on a GPU device would be to allocate one thread per each of the $N$ ant agents. This allows us to essentially “unroll” the for-loop. In such an implementation, each ant agent would generate a candidate solution, evaluate it, and deposit pheromone concurrently. This model leads to a truly asynchronous system where each ant agent has its own thread of control, a system virtually identical to real-world ant colonies. With this implementation strategy, the above pseudocode would be as shown in Listing 3.3. This is the implementation strategy we adopt for our AntMinerGPU algorithm and in Chapter 5 we show the performance benefits of this type of approach.

```plaintext
generate N ant agents
until (no improvement in solution quality for X iterations):
    each of N concurrent threads:
        generates a candidate solution;
        evaluates solution quality;
        deposits amount of pheromone proportional to quality on solution;
    done
done
extract final solution

Listing 3.3: Generalized parallel ACO algorithm with $N = number \ of \ ants$ threads
```

The benefit of the GPU computing model for flocking algorithms is equally as large. To illustrate, consider the very generalized flocking algorithm given in Listing 3.4.
generate N flock members
do until done:
  for each i in N:
    calculate bird i’s flock centering force
    calculate bird i’s velocity matching force
    calculate bird i’s collision avoidance force
    update bird i’s velocity
    update bird i’s position
  done
done

Listing 3.4: Generalized sequential flocking algorithm

As with the ACO algorithm given in Listing 3.2, we can use the multi-threading of a GPU device to “unroll” this algorithm’s for-loop. In this way, each flock member will update its velocity and position simultaneously. By applying the same strategy that was used for the parallel ACO algorithm, this flocking algorithm would become:

generate N flock members
do until done:
  each i in N concurrent threads:
    calculate bird i’s flock centering force
    calculate bird i’s velocity matching force
    calculate birds i’s collision avoidance force
    update bird i’s velocity
    update bird i’s position
  done
done

Listing 3.5: Generalized parallel flocking algorithm

This implementation strategy is adopted for our ClusterFlockGPU algorithm presented in Chapter 6.

In addition to each agent requiring its own thread of control, there are other characteristics of swarm intelligence algorithms also cast them as being well suited for the GPU
architecture. As noted in Section 3.2, memory latency is often the largest bottleneck with GPU-based algorithms. This can be combated by utilizing the smaller but faster on-chip shared memory. Since all agents operate solely on local data, these relatively small chunks of information can be loaded into low-latency shared memory. This accelerates the implementation and mitigates the impact of the long-latency DRAM memory operations. Additionally, during the solution evaluation phase of ACO algorithms, blocks of threads can be linked together to cooperatively evaluate the generated solutions and in turn accelerate this portion of the algorithm as well.
Chapter 4

Data Mining Tasks, Techniques, and Applications

This chapter provides an overview of the two data mining problems that our AntMinerGPU and CluterFlockGPU algorithms are designed to solve. We begin by describing the task of classification in Section 4.1 and then move to data cluster analysis in Section 4.2. These overviews are offered here to provide context for the algorithms that we present in Chapters 5 and 6.

4.1 Classification

The goal of classification is to be able to accurately assign a “class” to a given data point in an automated way and according to some number of the data point’s characteristics. That is, by examining the features of a given datapoint, predict which pre-defined class the data point actually belongs to. While there are many types of classification algorithms (as will be discussed in Section 4.1.2), we focus on so-called “rule-based” classifiers. The goal of
rule-based classification is to produce a set of classification rules that are able to correctly specify the class to which a data point belongs based on its characteristics or features. Our AntMinerGPU algorithm is a so-called “rule-based classifier” and is designed to produce classification rules. The process that AntMinerGPU uses to generate these rules will be examined in detail in Chapter 5. To illustrate how classification rules work, we show in Figure 4.1 a very simple data set and an associated classification rule. By plugging in the values of V1 and V2 from a given data point into the classification rule, the class of the data point can be accurately predicted.

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If (V1 == 1) && (V2 >= 0) && (V2 <= 3)
   class = 1
else
   class = 0

Figure 4.1: A sample data set and an example classification rule for predicting class

Formally, a classification rule \( r_i \) is expressed in the following way:

\[
r_i : (condition) \rightarrow y_i
\]

(4.1)

The left side of a rule, denoted \( condition \), is known as the rule precondition and describes what must be true about the data point being classified in order for the rule to apply. A
rule precondition contains a conjunction of attribute tests in the form of:

\[(A_1 \text{ op } v_1) \land (A_2 \text{ op } v_2) \land \ldots (A_j \text{ op } v_j)\] (4.2)

where \(A_j\) is some attribute (or feature) of the data, \(v_j\) is some value \(A_j\) may possess, and \(op\) is a logical operator \(\in\) \((=, \neq, <, >, \leq, \geq)\). The right side of the rule is the rule consequent and \(y_i\) is the class that \(r_i\) predicts data points satisfying condition to be a member of. A data point is said to be “covered” by some \(r_i\) if its attributes satisfy the precondition of \(r_i\).

Generally, the quality of a classification rule is measured by two values: coverage and confidence. The coverage of a rule is defined as the ratio of data points covered by the rule, to the total number of data points in the set. The confidence of a rule is the ratio of the number of data points covered by the rule and that are of the class predicted by the rule’s consequent, to the total number of data points covered by the rule. Formally, for a data set \(D\) and a classification rule \(r : A \rightarrow y\) coverage and confidence can are defined as follows:

\[Coverage(r) = \frac{|A|}{|D|}\] (4.3)

\[Confidence(r) = \frac{|A \cap y|}{|A|}\] (4.4)

where \(|A|\) denotes the number of data points that satisfy the precondition of \(A\), \(|A \cap y|\) is the number of data points satisfying the precondition of \(A\) and are of class \(y\), and \(|D|\) is the total number of data points in \(D\). Maximizing these two metrics indicates the best possible rule has been found as this indicates the rule can be applied to a large number of data points (coverage) and produces a minimal number of false positives (confidence).
For more information regarding classification, the reader is referred to [42].

4.1.1 Applications of Rule-Based Classification

Rule-based classification is a commonly used knowledge discovery method in a wide range of domains. A short survey of applications is given below.

Medical Diagnostics  In the domain of medical diagnostics, the goal of a classifier is to diagnose a condition based on the features of a patient (test results, genetics, etc...). In the case of genetic disorders, classifiers can be used to extract the key factors that may lead to some condition from the very large amount of genetic information attainable by modern gene profiling technology. For more information see Miller et al. [27] and Tsumoto [44]. A medial diagnostic data set will be used in Chapter 5 for evaluating the performance of our AntMinerGPU algorithm.

Financial Engineering  A very common task in the field of finance and lending is the classification of borrowers based on the likelihood that the borrower will default on a loan. Many financial institutions employ computerized decision support systems that compare the attributes of those requesting loans to those who have been issued loans in the past and in this way are able to classify loanees as safe or risky. Also, investment firms often employ classification systems to determine if a bond presents a risk that the issuing company will fail to make promised payments. For more information see Kim et al. [18], and Bosea et al. [5].

Marketing  Direct marketing is a promotion process where vendors attempt to motivate consumers to place orders. A common example is the direct mailing of catalogs or coupons to consumers a vendor has identified as possibly be interested in purchasing some type of product. In order for this marketing strategy to work, the vendor must be able to accurately segment the consumer base such that customized marketing materials can be delivered to the appropriate segment of consumers. Classification rules are often employed to predict what products a potential customer may be particularly interested in purchasing. For more information see Cheung et al. [6].

4.1.2 Traditional methods for Classification

Here we present a brief survey of other methods for performing classification. For a more in-depth overview of these methods and others, the reader is referred to [14] and [42].
**C4.5** - The C4.5 algorithm was developed by Ross Quinlan and is an extension of Quinlan’s earlier ID3 algorithm. The input for C4.5 is a set of already classified data points which serve as training data. The output is a so-called decision tree, a graph structure that captures decisions and their possible consequences. In the case of classification, a path through the decision tree from root to leaf is dictated by the features of the datapoint to be classified, and the leaf that is reached represents the class to which the datapoint is predicted to belong to.

**k-Nearest Neighbor** - The k-Nearest Neighbor (kNN) algorithm generates classification predictions via a “majority rules” decision making process. With kNN, the data items of a training data set are plotted in the n-dimensional feature space of the data. To classify a new data point, it is likewise plotted in the feature space and its class is predicted by the majority class of that point’s k-nearest neighbors. The kNN algorithm is known to have problems producing good classifications when data points in the training data set contain noisy or irrelevant features.

**Support Vector Machines** - Basic Support Vector Machine (SVM) algorithms for classification are methods for performing binary classification. That is, all data must belong to one of two possible classes. Essentially, SVM methods work by plotting the data points of a training set in the n-dimensional feature space of the data. The algorithm then works to identify a hyperplane that best defines a boundary between the two classes of data in the training set. A new data point can then be classified by plotting it in feature space and examining which side of the boundary it falls on. SVM algorithms have received a good deal of attention recently and have been shown to perform well on many types of data. There are also extensions to the basic SVM algorithm that can allow for multi-class data.

All three of these methods will be revisited in Chapter 5 when we compare their performance to that of our AntMinerGPU algorithm.

### 4.2 Partitional Cluster Analysis

The goal of partitional cluster analysis is to group together data points such that data objects in one group (or “cluster”) are similar to one another, and different from those in other clusters. Partitional data clustering methods attempt to separate (or partition) a collection of data points into clusters whereby data points in one cluster exhibit maximal
similarity to data points in that cluster, while exhibiting minimal similarity to data points in other clusters. This type of data cluster analysis produces so-called “un-nested” clusters. That is, the original data set is separated into non-overlapping subsets where each data item is in exactly one cluster. Our ClusterFlockGPU algorithm presented in Chapter 6 performs this type of clustering.

4.2.1 Applications of Data Clustering Algorithms

We provide here descriptions of two applications of partitional data clustering.

**Information Retrieval**  Document clustering attempts to organize large bodies of text to allow for easier browsing, searching, and retrieval of relevant works. A very large number of algorithms for document clustering have been proposed and this is still a very active area of research. Clustering algorithms similar to those used for document clustering are used heavily in the intelligence community to identify groups of related communiques that can indicate the presence of an organized terrorist network. For more information see Salton [38].

**Marketing and Business**  In market research, survey data can be clustered with results identifying some set of prototypical consumers. This data can then be used to allow for customized advertisements and promotions to be sent to consumers with a high likelihood of purchasing some product. Also, clustering techniques are often used in so-called “recommender systems” that analyze the purchasing patterns of consumers in order to determine other products a consumer is likely to be interested in purchasing. For more information see Wedel et al. [46], and Schafer et al. [39].

4.2.2 Traditional Partitional Data Clustering Techniques

We provide here a brief overview of two commonly used partitional data clustering techniques. For more information, the reader is referred to [14] and [42].
**K-means**  The $k$-means algorithm attempts to locate points, or centroids, in the feature space of an input data set that represent cluster centers. With some given value for $k$, the $k$-means clustering algorithm randomly generates $k$ cluster centroids and assigns each element to the centroid that is closest in feature space. Iteratively, the positions of the $k$ “means”, each representing a cluster center, are relocated to the average location of the elements assigned to that centroid. This continues until some stop criterion is met, normally that an iteration has passed with no re-assignment of any element or relocation of centroid.

**DBSCAN**  DBSCAN belongs to a class of so-called density-based clustering algorithms. The general approach of DBSCAN is to locate regions of high-density separated by regions of low-density in the feature space of a data set. Regions of high-density are considered to be clusters. The algorithm works by assigning labels (core, border, or noise) to each datapoint. So-called “core” data points are taken to be in the interior of a cluster. “Border” points are those that are not in the core of a cluster, but are within a certain radius of a core point. “Noise” points are defined as those points which are neither core nor border points and are eliminated. Clusters are taken to be all of those points who share a common core (core and border points).

We will revisit these two methods in Chapter 6 in the course of our discussion of our ClusterFlockGPU algorithm.

Having now establish the pertinent background information, we now move on to a detailed discussion of the two swarm intelligence data mining algorithms we have implemented. In Chapter 5 we discuss our AntMinerGPU algorithm for rule-based classification, and in Chapter 6 we describe our ClusterFlockGPU algorithm for partitional cluster analysis.
Chapter 5

The AntMinerGPU Algorithm

In this chapter we explore the basic principles of ant colony optimization (ACO) and describe the AntMinerGPU algorithm for rule-based classification. Section 5.1 presents an overview of ACO algorithms and introduces the MAX-MIN Ant System (MMAS) which will be utilized later on in the AntMinerGPU algorithm. Section 5.2 describes the AntMiner+ algorithm for rule-based classification and of our extension to AntMiner+, the GPU-based AntMinerGPU algorithm. Finally, Section 5.3 presents a listing and analysis of our results. We find that the AntMinerGPU algorithm is competitive with traditional classification algorithms and that it may offer considerable speedups as compared to the CPU-based AntMiner+ algorithm on which it is based.
5.1 Ant Colony Optimization

5.1.1 Background and Technical Details

Ant colony optimization (ACO) is a stochastic optimization strategy inspired by the collaborative path-finding behaviors exhibited by colonies of real ants. In nature, ants are simple organisms, each possessing very limited capabilities and, individually, only able to accomplish the most simple of tasks. Amazingly, colonies of ants are able to collectively solve difficult problems that are far beyond the abilities of any single member of the group. These tasks include the sorting of larva, transportation of heavy objects, and, most notably, finding short (read: optimal) paths between nest and food source.

Real ant colonies are a type of distributed and self-organizing system where the complex “global” behaviors exhibited by the colony as a whole are coordinated by indirect communication between the ants. Specifically, ants communicate with one another by depositing pheromone (a chemical substance produced by each ant) on the ground as they move about. As ants make their random explorations of the environment, they are more likely to follow these pheromone trails. The pheromone on a given trail will intensify as more ants follow it, and decrease in intensity over time by the process of evaporation when ants fail to travel it.

The process of pheromone trail laying/following in real ant colonies is mimicked by virtual ant agents in ACO systems. Generally, in ACO a population of independent ant agents move through an environment (most commonly a graph) that represents the solution space of some target problem. This graph representation of solution space is referred to as a “construction graph.” By generating a path through the construction graph for a target problem, each ant generates a candidate solution to that problem. By repeatedly generating solutions and reinforcing paths which represent good solutions with pheromone, the optimal solution (or
The movement of ants in the construction graph is dictated by a stochastic transition rule based on two pieces of local information: pheromone level and heuristic value. Edges in the construction graph have a level of pheromone associated with them, and each node has a certain heuristic value. The amount of pheromone on an edge is a measure of how many ants have recently traversed the edge and attempts to attract ants to edges which have been identified as components of good solutions. The heuristic value of a node is derived from a priori knowledge about the target problem and attempts to capture the relative importance of a node in candidate solutions.

An ant located on some vertex in the construction graph will select an edge to traverse with a probability given by a transition rule that takes into account the pheromone level on each connected edge and the heuristic value of the vertices to which the edges connect. The decision will favor an edge with a relatively high pheromone level and that connects to a vertex with a relatively high heuristic value. It should be noted that the transition rule only gives the probability of selecting a given “next move.” This probability is used to weight a random selection process where the next move is actually selected. This causes random decisions to be introduced into the system.

Once an ant has completed a tour of the graph, the path traveled by the ant (representative of a candidate solution) is evaluated and an amount of pheromone is deposited on the path relative to the quality of that solution. The updating of pheromone levels on edges in the construction graph is dictated by two factors: reinforcement and evaporation. Reinforcement is the process by which pheromone levels on a given path are increased via the addition of some quantity of pheromone to the edges that path is comprised of. By reinforcing high quality paths, ants making subsequent explorations are more likely to explore similar paths,
introducing randomness along the way, and (hopefully) finding a yet better path. Evapora-
tion is the process whereby pheromone levels on all graph edges are diminished over time. 
Via evaporation, trails that may represent sub-optimal solutions that were reinforced earlier 
in the exploration phase will appear less attractive to the ants and be “forgotten” over time.

As presented in [24], ant systems require that the following be specified:

1. An environment that represents the solution space in a way that allows for the incre-
   mental construction of complete solutions (most commonly a graph)
2. A heuristic function ($\eta$) that specifies the relative importance of solution components
3. A pheromone ($\tau$) updating rule which takes into account reinforcement and evaporation
4. A probabilistic transition rule that is a function of ($\tau$) and ($\eta$)
5. A quantitative fitness function based on the target problem for evaluating the quality 
of candidate solutions
6. A well-defined termination condition

In Section 5.2 we describe how each of these is specified in the context of the AntMin-
erGPU algorithm.

5.1.2 The MAX-MIN Ant System

It has been shown that the performance of simple ant systems as described above is relatively 
poor for large problems [40]. In an effort to make ACO systems suitable for a wider range of 
problems, certain modifications have been proposed and many versions of ACO have been 
offered [11]. The MAX-MIN Ant System (MMAS) proposed by Stutzle et al. has been 
shown in most cases to produce the best results for discrete optimization problems [41]. The 
MMAS differs from the standard model in three ways:
1. The pheromone update rule specifies that after each iteration, only the path representing the iteration-best or global-best solution receives reinforcement.

2. Pheromone levels are limited to a range, \([\tau_{\text{min}}, \tau_{\text{max}}]\), in order to prevent certain routes from becoming overly attractive and others from being unexplored altogether.

3. All edges are initialized to \(\tau_{\text{max}}\) before the first iteration in order to encourage a more complete exploration of solution space in the early phases of the algorithm.

With these modifications, the MMAS has been shown to be able to tackle a wide range of optimization problems (see [41] for a survey of applications), and forms the basis of the AntMiner+ and AntMinerGPU algorithms.

### 5.1.3 Ant Colony Optimization and GPGPU

To reiterate, the general approach of ACO algorithms for solving discrete optimization problems is to use a population of asynchronous, independent agents to thoroughly explore the solution space of the target problem and locate the optimal solution. Ants communicate their relative success or failure to find a good solution via pheromone deposition in this solution space and with yet more explorations, a (near)optimal solution is eventually found.

It is our hypothesis that with a larger population of ant agents than suggested by current literature, higher-quality solutions will be found in less time. Because each ant is a completely independent agent and requires its own memory and control logic (or thread), implementing large ant populations for ACO on a single-core CPU presents a problem with respect to running time. With a sequential implementation, ants are forced to “take turns” on the processor, each generating its tour one after the next. This is a sub-optimal implementation strategy for large ant populations since computational complexity will grow directly with the number of ant agents in the system. We propose that a parallel implementation will reduce the computational complexity of the bulk of the algorithm and thus allow for the efficient
execution of ACO algorithms with very large populations of ant agents. Furthermore, a parallelized implementation where each ant is allocated its own control thread allows the virtual ant colony to truly achieve the asynchronous nature of the real ant colonies on which the model is based. To explore our hypothesis, we present a GPU-based implementation of the AntMiner+ algorithm for rule-based classification in the next section.

5.2 The AntMinerGPU and AntMiner+ Algorithms

The first version of the AntMiner algorithm was proposed by Partinelli et al. in [33] and was shown to be able to produce relatively high-quality classification rules for biomedical data. The algorithm was further developed by Liu et al. in [21] where the authors propose AntMiner2 and AntMiner3. A survey of the literature reveals that the most recent version of the AntMiner algorithm, AntMiner+, produces higher-quality classification rules, and offers a less complex implementation than previous versions. For these reasons, the AntMiner+ version of the AntMiner algorithm has been adopted here. The following subsections contain a recapitulation of the information given in [24] and describe the basic operation of the AntMiner+ algorithm. Following this, we describe AntMinerGPU—the GPU-based version of the AntMiner+ algorithm. For more information regarding the previous version of AntMiner, the reader is referred to [33] and [21].

5.2.1 The AntMiner+ Algorithm

The AntMiner+ classification algorithm is an extension of the MMAS. Given a training data set where the class of each data point is known, AntMiner+ iteratively builds up a set of classification rules that can be used to predict the class of new data points. While [25]
presents an implementation of AntMiner+ able to produce classification rules for datasets with more than 2 classes, the implementation presented here is only concerned with binary classification (i.e. a data points are a member of one of two possible classes). As such, rules generated by AntMiner+ always attempt to identify the predictive characteristics of data points in class 1, using an explicit “else” statement as the final rule to indicate the any data point not covered by a rule must be in class 0.

The basic task of ants in the AntMiner+ algorithm is to probabilistically select a subset of terms \((Variable_i \ op \ Value_k)\) from the set of all possible terms such that once conjoined with the logical AND operator, good classification rules are produced. Figure 5.1 depicts the general form of rules produced by AntMiner+.

\[
\begin{align*}
\text{if} (Variable_1 \ op \ Value_1) \ \text{AND} \ (Variable_2 \ op \ Value_2) \ \text{AND} \ (Variable_i \ op \ Value_k) \\
\text{then Class} = 1 \\
\text{else Class} = 0
\end{align*}
\]

\( \text{where } op \in (=,\leq,\geq) \)  

(5.1)

Figure 5.1: The general form of rules generated by AntMiner+

The AntMiner algorithm uses a simple directed acyclic graph (DAG) as the representation of solution space. Figure 5.2 below depicts the general form of this DAG.

The AntMiner+ construction graph is created by generating one node group for each variable in the training data set. These node groups are shown as vertical rectangles in Figure 5.2. Each group contains nodes representing all possible values the respective variable may possess. As can be seen in Figure 5.2, a distinction is made between nominal and ordinal variables in the construction graph.

Nominal variables, for example V1 in Figure 5.2, are represented by one node group.
The group contains a node for each value the variable may possess, as well as an additional node representing “does not matter.” This “dummy” node can be selected by the ants and represents the possibility that the given variable has no bearing on the class that is predicted by the rule. An ant visiting node \( k \) in the node group representing nominal variable \( i \) will add the term \((\text{Variable}_i = \text{Value}_k)\) to its candidate rule. If node \( k \) is the “dummy” node, no term is added to the classification rule.

The AntMiner+ algorithm also allows for ordinal (or continuous) variables to be used in the data mining process. For these types of variables, we allow for terms which can specify a range of values. That is, we allow for terms such as \((\text{Variable}_i \geq \text{Value}_k)\) and \((\text{Variable}_i \leq \text{Value}_k)\). As such, ordinal variables are represented by two node groups as shown in Figure 5.2 for variable \( V2 \). These two groups represent a lower and upper bound that can be placed on a given ordinal variable in the classification rule.

A given ant will select some term to add to its candidate rule according to a probabilistic selection rule that is a function of the pheromone level on the edge going to that node (term) and heuristic value associated with that node. Detailed descriptions of how pheromone levels,
heuristic values, and probabilities are calculated are given in the following subsections.

The movements of ants are restricted by requiring that each nominal variable appear at most once and each ordinal variable at least twice in the classification rule (accomplished via the direction of edges in the graph). In the case of terms representing ordinal variables, the choice of terms is also restricted such that a rule will not be produced that leads to logically unsatisfiable conditions. Specifically, we wish to avoid the situation where a rule specifies \((\text{Variable}_i \leq x) \text{ AND } (\text{Variable}_i \geq y)\) where \(y > x\). To prevent this case, we simply remove edges from the construction graph that would allow for this type of path.

The basic operation of the AntMiner+ algorithm is as follows. At the start of every iteration, ants begin in the start node and walk through the construction graph until reaching the end node. As an ant walks, it records the nodes it has visited, each representing a logical term which is added to the ant’s candidate classification rule. Once all ants in a generation reach the end node, the rule described by each ant’s tour is evaluated and the path representing the best rule is reinforced with pheromone. In the next iteration, this pheromone trail will encourage other ants to explore a similar region of the construction graph. When all ants have converged to a given path (classification rule), the rule is extracted and data points in the training set which are covered by the rule are removed. The process then repeats until a specified percentage of training data points have been covered or early stopping occurs (early stopping criteria is described in Section 5.2.1.4). Listing 5.1 presents the pseudocode for the AntMiner+ algorithm as given in [24] and Figure 5.3 gives an example classification rule.
generate construction graph

do until (min. percentage of training data remains OR early stopping)
    initialize heuristic values, pheromones, and edge probabilities
    while (not converged)
        create ants
        let ants walk from start node to end node
        evaporate pheromone from edges
        reinforce path of iteration-best ant
        clamp pheromone levels to [Tmin, Tmax] as specified by the MMAS
        kill ants
        update edge probabilities
    end
    extract rule
    remove data points from training data covered by the extracted rule
end

Listing 5.1: Pseudocode for the AntMiner+ algorithm

![Construction Graph Diagram]

if \( V1 = 1 \) \&\& \( V2 >= 0 \) \&\& \( V2 <= 3 \) 
then Class = 1
else Class = 0

Figure 5.3: An example path through the construction graph and its corresponding classification rule. Here, we use “X” to indicate a “dummy” node.
5.2.1.1 Transition Rule

Equation 5.2 gives the probability $\text{Prob}(n_{i,j}, n_{i+1,k})$ an ant will select to move from node $(n_{i,j})$ (the node representing value $j$ for variable $i$) to some node $(n_{i+1,k})$. The notation $|\text{condition}|$ denotes the number of values fulfilling condition.

$$\text{Prob}(n_{i,j}, n_{i+1,k}) = \frac{\left[\tau(n_{i,j}, n_{i+1,k})\right]^\alpha \cdot \left[\eta(n_{i+1,k})\right]^\beta}{\sum_{l=1}^{|V_{i+1}|} \left[\tau(n_{i,j}, n_{i+1,l})\right]^\alpha \cdot \left[\eta(n_{i+1,l})\right]^\beta} (5.2)$$

This probability rule is a function of pheromone levels on a given edge ($\tau$), and the heuristic value of the node to which a given edge connects ($\eta$). The values $\alpha$ and $\beta$ are constants that are used to weigh the relative influence of these two factors on the ants’ decisions and are supplied as parameters to the algorithm. Experimentation revealed that setting $\alpha$ and $\beta$ both equal to 1 produced the best results in our test cases.

5.2.1.2 Heuristic Value

The heuristic value of a node attempts to capture the importance of that node’s value for its variable in classification rules. The heuristic value of a given node is taken to be the number of data points with $\text{class} = 1$ where the variable of the given node possesses the value which that node represents. This is given in equation 5.3 below. We use the notation $|\text{condition}|$ to refer to the number of elements in the set of all data points that satisfy condition.

$$\eta_{n_{i,k}} = \frac{|(Variable_i = Value_k) \cap (\text{class} = 1)|}{|Variable_i = Value_k|} (5.3)$$

It should be noted that these values are relatively static and only need to be updated when a rule is extracted and data points are removed from the training data set.
5.2.1.3 Pheromone Updating

The updating of pheromone levels in the construction graph is accomplished in two phases: evaporation and then reinforcement. Pheromone evaporation occurs on all edges in the construction graph. To account for evaporation, the pheromone level on an edge in iteration \((t+1)\) is given by:

\[
\tau_{n_{i,j},n_{i+1,j}}(t+1) = \rho \cdot \tau_{n_{i,j},n_{i+1,j}}(t)
\]  

(5.4)

The value \(\rho\) is a so-called evaporation constant and is usually in the range \([0.8, 0.99]\) as described in [41]. We select a relatively high value for \(\rho\), 0.95. In this way, we are able to encourage a more complete exploration of the construction graph.

Pheromone reinforcement in the MMAS only occurs on the path of the “best ant.” The “best ant” can be taken to be either the best ant of the current iteration, or the best ant found since the start of the algorithm. The literature suggests that better results can be achieved by favoring the iteration-best ant over the global-best ant [41] and this strategy is adopted here.

The AntMiner+ algorithm uses the sum of confidence and coverage (similar to those metrics described in Section 4.1) to establish a quantitative measure of the quality of the rules described by the ants. To determine quality, Equation 5.5 below is applied to each ants’ rule and the “best ant” is taken to be the one whose rule gives the largest value of \(\Delta_{best}\).

\[
\Delta_{best} = \underbrace{|\text{rule} \cap \text{class} = 1|}_{\text{confidence}} \frac{1}{|\text{rule}|} + \underbrace{|\text{rule} \cap \text{class} = 1|}_{\text{coverage}} \frac{1}{|\text{Covered} = 0|}
\]  

(5.5)

The confidence and coverage of some rule are computed as the ratio of data points covered
by rule that are of class 1 to the total number of data points covered by rule and the total number of uncovered data points in the training data set, respectively. By combining the measures of confidence and coverage, \( \Delta_{\text{best}} \) will be maximized by a rule that generates few false positives, yet covers many data points.

Once the best ant of a given generation has been determined, the edges comprising its path are reinforced with an amount of pheromone equal to its quality. Pheromone reinforcement for the path of the best ant is given by equation 5.6.

\[
\tau_{n_i,j,n_{i+1,j}}(t+1) = \tau_{n_i,j,n_{i+1,j}}(t) + \Delta_{\text{best}}
\]  

(5.6)

The MMAS also specifies that pheromone levels on edges of the construction graph be limited to the range \([\tau_{\text{min}}, \tau_{\text{max}}]\). As pheromone levels are updated via evaporation and reinforcement, care must be taken to ensure the resultant levels are bounded by this range. Furthermore, the values of \( \tau_{\text{min}} \) and \( \tau_{\text{max}} \) are dynamically updated during execution of the algorithm. Whenever the quality of an iteration-best ant improves the quality of the global-best ant, the values of \( \tau_{\text{max}} \) and \( \tau_{\text{min}} \) are updated by applying equation 5.7 followed by equation 5.8. The overall effect of this step is to increase the values of \( \tau_{\text{max}} \) and \( \tau_{\text{min}} \), and to increase the range of pheromone levels between them.

\[
\tau_{\text{max}} = \frac{1}{1 - \rho} \cdot \Delta_{\text{best}}
\]  

(5.7)

\[
\tau_{\text{min}} = \frac{\tau_{\text{max}}}{a}
\]  

(5.8)

In Equation 5.7, \( \rho \) is the evaporation constant from Equation 5.4, and in Equation 5.8, \( a \) is a parameter. The exact derivation of these equations is given in [11] and [41].
5.2.1.4 Early Stopping

Early stopping is a commonly used technique in machine learning that prevents classification rules from fitting the noise or particularities of training data. The goal of rule-based classification algorithms is to produce a set of rules that can predict the class of new and previously unseen data points. Rules that are extracted from training data and can accurately predict the class of new data points are said to generalize well and are desirable. To avoid extracting rules that do not generalize well, a validation data set can be used to test rules during rule creation.

To accomplish this, training data is split into two subsets, one used for rule creation and the other for validation. Rules are continually created by the classification algorithm until the performance of the rules begins to decrease on the validation data set. In this way, the classification algorithm will stop extracting rules as soon as it detects it is fitting rules to the particularities or noise of the training data set and not extracting general rules.

5.2.2 The AntMinerGPU Algorithm

As discussed in Chapter 3, GPU computing with CUDA requires the CPU to orchestrate GPU kernel calls. While a good deal of the processing in the proposed AntMinerGPU algorithm does occur on the GPU, there is an interplay between CPU and GPU operations. A pseudocode overview of AntMinerGPU is given in Listing 5.2 and Figure 5.4 provides a diagrammatic representation of the control flow of the algorithm. In Listing 5.2, we use the notation (C/G - threadCount) to indicate where the step takes place (CPU or GPU) and how many concurrent threads are used (threadCount).
Listing 5.2: Pseudocode for the AntMinerGPU algorithm

**edgeCount** = total number of edges in construction graph

**nodeCount** = total number of nodes in construction graph

**variableCount** = dimensionality of training data

**dataCount** = total number of data points in training data set
5.3 Results and Analysis

To assess the performance of the AntMinerGPU algorithm, we compare it to the AntMiner+ algorithm and to three traditional classification algorithms via the Weka software package. Weka provides Java implementations of a number of machine learning and data mining algorithms and is a commonly used program in the teaching of the topic. For more information about Weka, the reader is referred to [31].
We begin by assessing the efficiency of AntMinerGPU by running it on test data sets of varying size and dimensionality. We then compare the performance AntMinerGPU to that of the three traditional classification algorithms presented in Chapter 4 (C4.5, k-Nearest Neighbor, and Support Vector Machines) and to the AntMiner+ algorithm on which AntMinerGPU is based.

5.3.1 Data Sets

We use three data sets to assess the performance of the AntMinerGPU algorithm. All data sets were downloaded from the UCI machine learning repository ([16]).

**Wisconsin Breast Cancer (WBC)** - This data set consists of 569 instances corresponding to profiled breast tumor cells. Each instance has 9 numerical attributes and one class designation. The 9 attributes describe various characteristics of the profiled cells. The class attribute designates the data point as “benign” or “malignant.” The task of classification for this data set is to predict whether a previously unseen data point is of the class benign or malignant. For more information, refer to [23].

**Mushroom** - This data set consists of 8124 instances corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each instance has 22 nominal attributes and a class designation. Some instances have missing values for attribute 11 and since the AntMinerGPU algorithm is unable to handle missing data, this attribute has been removed giving a total of 21 attributes in the actual test data set. The class attribute designates each instance as being a mushroom that is ‘definitely edible’, ‘definitely poisonous’, or of ‘unknown edibility’. Because the AntMinerGPU algorithm is intended for binary classification only, we combine the ‘definitely poisonous’ and ‘unknown edibility’ into one class. The task of classification for this data set is to
determine whether a previously unseen data point is ‘poisonous’ or ‘edible’. For more information, refer to [43].

**Contraceptive Method Choice (CMC)** - This data set is derived from the 1987 National Indonesia Contraceptive Prevalence Survey. It consists of 1473 instances, each with 9 attributes and a class designation. Two of the 9 attributes are ordinal, and the remaining nominal. The class attribute is in the set \{1, 2, 3\} indicating no, long-term, and short-term use of contraceptives respectively. Because the AntMinerGPU algorithm is only capable of performing binary classification, data points in class 3 are combined with those in class 2 resulting in a data set with a total of 2 classes. The goal of classification for this data set is to determine if a woman uses contraceptives based on her descriptive characteristics. For more information, refer to [20].

For testing purposes, we generate training and testing data sets by first randomizing the order of data points, and then take the first 66% as training data, and the last 33% a testing data.

### 5.3.2 Test System

Results presented in Section 5.3.3 were gathered on two different computers. Results for the Weka implementations of traditional classifications algorithms were gathered on an Apple MacBook Pro laptop with a 2.53 GHz Intel Core 2 Duo processor and 4GB of DDR3 memory. The AntMinerGPU algorithm was run on a desktop computer system with a 2.4 GHz Intel Core 2 Quad processor and 3 GB of DDR3 memory. The GPU device used was an NVIDIA Tesla C1060. We feel a valid comparison can be made between the results gathered on these two systems as their hardware configurations are similar. Also, because in the case
of the GPU-based algorithm, very little work is being done by the CPU and as such the configuration of the GPU-based computer system (CPU specs, memory type, etc...) has little bearing on the results.

5.3.3 Performance Analysis

Because AntMinerGPU is a stochastic algorithm, results can vary from run to run. For this reason, the metrics reported in Figure 5.5 are averages over 10 runs on each test data set with the all-time-best value shown in parentheses. All runs were completed with a population of 6000 ants, and we require the ants to continue generating rules until 90% of training data of class 1 is covered. These values are held constant across all tests for comparison’s sake. In a real-world application, these values would be set depending on the characteristics of the data set being classified. The testing metrics we give are precision, recall, and accuracy as given by:

\[
\begin{align*}
\text{Precision} & = \frac{tp}{tp + fp} \\
\text{Recall} & = \frac{tp}{tp + fn} \\
\text{Accuracy} & = \frac{tp + tn}{tp + tn + fp + fn}
\end{align*}
\]

where \( tp = \) true positives, \( fp = \) false positives, \( tn = \) true negatives, and \( fn = \) false negatives.

We see that AntMinerGPU is able to generate high-quality classification rules for the WBC and Mushroom data sets. An interesting pattern that we have discovered is that for some data sets, the precision of rules generated by AntMinerGPU are consistently near 100%. The WBC and Mushroom data sets are clear examples of this phenomenon. The
Figure 5.5: Performance of the AntMinerGPU algorithm. Values shown are averages over 10 independent runs. Best performance exhibited over the 10 runs shown in parentheses.

cause of this behavior is in need of further investigation.

Also under consideration is the impact of the size of ant populations on running time and accuracy. In [25], the authors use a population of 1000 ants in their sequential implementation of the AntMiner+ algorithm. To determine the effect of ant population, we run AntMinerGPU with a range of populations, plotting this variable against accuracy of rules generated for the WBC data set as well as the average running time of the algorithm in Figure 5.6.

Figure 5.6: The effect of population on running time and accuracy of the AntMinerGPU algorithm. Graph (a) shows the relationship between ant population sizes and the accuracy of extracted rules, and graph (b) relates ant population size to overall running time
Interestingly, accuracy appears to plateau at approximately 1500 ants with only minor improvements being gained by yet larger populations. Running time shows a roughly linear relation to population.

The results presented in [25] indicate that the dominating factor for running time of the AntMiner+ algorithm is the size of the data set being classified. We explore here the impact of data size as well as dimensionality on running time. We use the Mushroom data set in both of these tests. To achieve data sets of varying size, we simply truncate the original data set by varying amounts. Different degrees of dimensionality are achieved by either removing or duplicating some number of variables from the original data set. Results are shown in Figure 5.7.

\begin{figure}[h]
\centering
\begin{subfigure}{0.45\textwidth}
\centering
\includegraphics[width=\textwidth]{data_size_vs_runtime.png}
\caption{(a) Data Size vs. Running Time}
\end{subfigure}
\begin{subfigure}{0.45\textwidth}
\centering
\includegraphics[width=\textwidth]{dimensionality_vs_runtime.png}
\caption{(b) Dimensionality vs. Running Time}
\end{subfigure}
\caption{The impact of (a) data set size and (b) dimensionality on overall running time of AntMinerGPU}
\end{figure}

These results indicate the AntMinerGPU algorithm has similar time complexity to the AntMiner+ algorithm with regard to the size of the data set being classified. Figure 5.7(a) indicates that AntMinerGPU scales in a roughly linear fashion with the size of the input data set. Figure 5.7(b) reveals that dimensionality has a similar effect on running time and
we find the overall running time of AntMinerGPU to scale linearly with dimensionality.

5.3.4 Comparison to Traditional Methods

As mentioned, we use the Weka software package to gather data regarding the performance of traditional classification algorithms. We use the following algorithms for our comparison, and use the default parameters Weka provides for each algorithm.

**k-Nearest Neighbor (NNge)** - This is a nearest-neighbor-like algorithm that uses “non-nested generalized exemplars.” This algorithm is described in detail in [37].

**Support Vector Machine (SMO)** - This algorithm is an implementation of John Platt’s Sequential Minimal Optimization (SMO) algorithm for Support Vector Machines. This algorithm is described in detail in [34].

**C4.5 (J48)** - This algorithm generates a pruned C4.5 decision tree. This algorithm is described in detail in [35].

![Table](image)

Figure 5.8: Comparison of the accuracy given by the AntMinerGPU algorithm and three traditional classification algorithms. AntMinerGPU accuracies are average values over 10 runs. We also give the best performance exhibited by AntMinerGPU over the 10 runs in parentheses. The accuracy of the method giving the best results is shown in bold.

We find that in general, the results of AntMinerGPU are competitive with those of the traditional algorithms we compare it to. To be noted is that the best performance of
AntMinerGPU over 10 runs is extremely competitive with traditional methods if not better. We propose a method to take advantage of this in Section 5.4.

### 5.3.5 Comparison to AntMiner+

We compare the accuracy of AntMiner+ and AntMinerGPU by applying both to the WBC data set. For this data set, AntMinerGPU generates, on average, rules with 96.5% accuracy with an all-time-best of 98.5% accuracy. AntMiner+ generates, on average, rules with 96.4% accuracy [1]. In terms of this metric, we conclude that AntMinerGPU performs only marginally better than AntMiner+. We believe this to be because the AntMiner+ algorithm uses a large population of ants (1000) regardless of the impact this has on its running time.

In [25], the authors run the AntMiner+ algorithm on 12 data sets ranging in size from 150 to 1728 instances, and from 4 to 19 dimensions. They report the average running time to be 1776 seconds. We ran AntMinerGPU on 3 data sets ranging in size from 683 to 8124 instances, and from 9 to 21 dimensions. Our average running time was 21.3 seconds. While the accuracy of rules produced by the AntMinerGPU algorithm are only marginally better than those given by AntMiner+, we do see a marked improvement in running time. We note these results and comparison values in Figure 5.9.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># of Sets</th>
<th>Size (avg)</th>
<th>Dimensionality (avg)</th>
<th>Average Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AntMiner+</td>
<td>12</td>
<td>min: 150 max: 1728 (763)</td>
<td>min: 4 max: 19 (8.6)</td>
<td>1776</td>
</tr>
<tr>
<td>AntMinerGPU</td>
<td>3</td>
<td>min: 683 max: 8124 (3428)</td>
<td>min: 9 max: 21 (13)</td>
<td>21.3</td>
</tr>
</tbody>
</table>

Figure 5.9: Comparison of the overall average running time of the AntMiner+ and AntMinerGPU algorithms. Comparison values are taken from [25]
It should of course be noted this is not an exact comparison of performance as the two algorithms were run on different data sets, on different hardware, and were both implemented in different programming languages. Because of this, we very cautiously estimate that AntMinerGPU provides a 20-50x speedup over the AntMiner+ algorithm and acknowledge the more work is needed to accurately assess the speedup of the AntMinerGPU algorithm over AntMiner+.

5.4 Next Steps

We present here a few improvements to AntMinerGPU that could increase the performance and usability of the algorithm.

5.4.1 Multi-Class Data

A very major limitation of the AntMinerGPU algorithm is its inability to classify data containing more than 2 classes. In [25] the authors present a version of the AntMiner+ algorithm that is capable of multi-class classification. To achieve this functionality, an additional node group can be added to the construction graph which contains nodes representing the classes contained in the training data set. In this way, ants construct rules and then pick a class node which specifies which class the candidate rule classifies. Including this functionality would require minimal modification of the AntMinerGPU code and is a feature we wish to see implemented.
5.4.2 Multiple Colonies

We note in figure 5.8 that the all-time-best performance of the AntMinerGPU algorithm over 10 runs exceeds the average performance and can often exceed the accuracy of the traditional methods we compare to. To take advantage of this, we propose that multiple instances of AntMinerGPU may be run concurrently, perhaps on a multi-GPU computer system, and the instance with the best performance be taken as the final solution. Because all ACO systems, AntMinerGPU included, rely heavily on parameter settings (such as the evaporation constant, pheromone level limits, etc...), we also suppose that the multiple instances of AntMinerGPU could be run with varying parameter values.

5.4.3 Improved Convergence Detector

As noted in the description of the AntMinerGPU algorithm, we use the passing of 20 iterations with no improvement of the global-best solution generated by the ants to indicate that the best solution has been found. In [24] that authors advocate a termination condition that is triggered when a path in the construction graph from start node to end node has pheromone level $\tau_{max}$ on all of its edges and all other edges in graph have $\tau_{min}$ pheromone. We found that the time associated with analyzing the construction graph as is needed to evaluate this termination condition to be substantial for larger construction graphs. Also, experimentation revealed that after 20 consecutive iterations with no improvement of the global-best solution, it was rare that one was ever found. We propose that a hybrid of these termination conditions could be used to effectively and efficiently determine convergence. Further investigation regarding the effect of termination condition on rule accuracy and algorithm running time needs to be carried out.
5.4.4 Better Use of Shared Memory

We believe that better use could be made of specialized GPU memories to further improve the running time of AntMinerGPU. We find that overall running time of the algorithm scales directly with the size of the input data set and dimensionality. In the current version, AntMinerGPU stores training data in long-latency global memory. Because this data is stored in a 1-D array in row major form, we are able to mitigate the impact of global memory latency by binding the training data to a texture object and capitalizing on the the performance benefits of caching. There is a possibility that yet better performance could be achieved by more intelligently utilizing shared-memory in order to keep training data points in on-chip memory to greatly reduce the need for threads to access data points stored in global memory.

5.4.5 Very Large Data Sets

The limiting factor for analyzing very large scale data sets with GPGPU devices is the limited amount of memory these devices contain. In order to store much larger training data sets in GPU memory, we propose that a pre-processing step could be used to translate the values of the input data set into short integer values. This step would generate a look-up table that would be stored in CPU memory and could be used to translate between these short integer “value codes” and the actual values that they represent. By translating input data to short integer form, twice the amount of data can be loaded onto the GPU as compared to using integer or float data types. This step would of course introduce more running time as pre- and post-processing is needed, but would also allow for larger data sets to be classified.
Chapter 6

The ClusterFlockGPU Algorithm

This chapter explores swarm intelligence algorithms for the data clustering problem and introduces the ClusterFlockGPU algorithm. Section 6.1 will present two examples of swarm intelligence algorithms for data cluster analysis. While the focus of this chapter is a flock algorithm, we give a brief overview of other swarm intelligence approaches to the data clustering problem and will compare the reported effectiveness of these algorithms to our ClusterFlockGPU algorithm in Section 6.5. Section 6.2 explores the fundamentals of flock algorithms, and Section 6.3 presents the basis for a flock-based clustering algorithm. Section 6.4 gives a description of the ClusterFlockGPU algorithm and GPU-based implementation. Finally, Section 6.5 gives results and analysis and Section 6.6 discusses future work.
6.1 Swarm Intelligence Algorithms for Data Clustering

6.1.1 Particle Swarm Optimization for Cluster Analysis

The particle swarm optimization (PSO) technique is a very well explored area in swarm intelligence and has been shown to generate good results for the data clustering problem [26]. We explore the nature of PSO clustering algorithms here as we will revisit this method in our results section when we compare the performance of ClusterFlockGPU to a PSO-based clustering algorithms.

In PSO algorithms, a swarm refers to a collection of potential solutions to some target problem and each solution is referred to as a particle. In the case of data clustering, solution space represents all possible clusterings of data points and each particle represents one such configuration.

At the beginning of the general PSO clustering algorithm, some number of particles are created in solution space with each maintaining three pieces of private information:

1. Position in n-dimensional solution space
2. Velocity in n-dimensional solution space
3. The position and quality of this particle’s personal-best solution

In the initialization of the particles, (1) and (2) are initialized to random values, and (3) is initialized to NULL.

During every iteration, each particle updates its position as dictated by its velocity. When each particle attains a new position in solution space, each evaluates the quality of its respective solution (position). If this quality exceeds the quality of the best-so-far solution
a given particle has found, it will update this value and corresponding position in its private
memory.

The social aspect of the PSO algorithm comes at the end of each iteration when all
particles compare the quality of the solution each has found. This exchange of information
allows each particle to know the location of the best solution found by the swarm as a whole.
Each particle then adds to its velocity a vector pointing to the global-best solution, as well
as a vector pointing toward its personal-best solution. In this way, all particles are drawn
toward regions of solution space that either itself and/or others have found good solutions in.
To encourage a more complete exploration of solution space, random perturbations are also
introduced into the velocities of the particles. This process of exploration and communication
is repeated for some maximum number of iterations and eventually particles converge to the
(near)optimal position in solution space.

An overview of the generic PSO clustering algorithm is given below in Listing 6.1. For a
more complete description of PSO algorithms in general, as well as those used for clustering,
the reader is referred to [47], [17], and [26].

initialize N particles in solution space
until done:
  for each i in N:
    evaluate quality of the solution represented by i’s position
    if quality is better than i’s personal-best solution
      i’s personal-best position = i’s position
    communicate the quality of i’s solution with all other particles
    velocity += rand1*<vector pointing to global-best solution> +
                 rand2*<vector pointing to personal-best solution>
  done
done
final clustering = global-best solution of particle swarm

Listing 6.1: Pseudocode of the generalized PSO clustering algorithm where rand1 and
rand2 are random values
In Section 6.5 we will use the results of the PSO clustering algorithm given in [26] as comparison values to evaluate the performance of ClusterFlockGPU.

6.1.2 Ant Colony Algorithms for Cluster Analysis

As we will also be comparing the performance of ClusterFlockGPU to an ant colony approach to clustering, we provide some background here regarding the operation of ant-based clustering algorithms.

The first study relating the behaviors of ant colonies to the data clustering problem appears in [8]. Here, Deneubourg et al. examine the brood sorting behaviors of ant colonies and note that as ants make random movements, picking up and dropping larva along the way, distinct piles of similar larva are created. This clear metaphor for partitional clustering was further advanced by Lumer and Faieta in [22], where the authors present an actual clustering algorithm inspired by the brood sorting behavior described in [8].

While many variations on ant colony clustering algorithms exist, the basic idea is as follows. First, a two dimension virtual environment is created as a workspace for the ants. Then, a collection of objects are generated, one for each data point in the input data set, and are randomly scattered throughout the environment. Some number of virtual ants are then created and are allowed to wander randomly through the environment. At each step, an ant can either pickup or drop a data object at its current location given that there is a data object at the ant’s location or that the ant is carrying a data object, respectively. The probability that an ant will pick up some data object \( i \) is given by the formula:

\[
P_{pick}(i) = \left( \frac{k_p}{k_p + f(i)} \right)^2 \tag{6.1}
\]
where $k_p$ is a constant and $f(i)$ is an estimation of the number of objects close to $i$’s location and their relative similarity to $i$.

Likewise, an ant will drop a data item $i$ at its current location with a probability $P_{\text{drop}}$ that correlates to the presence of other data items near the ant’s position that are relatively similar to $i$. Formally, we have:

$$P_{\text{drop}}(i) = \begin{cases} 
2f(i) & \text{if } f(i) < k_d \\
1 & \text{otherwise.}
\end{cases} \quad (6.2)$$

where $f(i)$ is as in Equation 6.1 and $k_d$ is a constant. Note that $f(i) \in [0, 1]$ and $k_d$ is set such that if $k_d << f(i)$ then an ant will almost certainly pick up a data item in a low density region. Over time, this repeated process of picking up and dropping data objects leads to piles of data objects being formed with each pile being strongly representative of a cluster in the input data set.

In Section 6.5, we use the results of the Ant-Based clustering algorithm given in [15] as comparison values to evaluate the performance of our ClusterFlockGPU algorithm.

### 6.2 Flock Algorithms

The term “flock algorithm” refers to any algorithm that mimics the collective behaviors exhibited by flocks of birds, schools of fish, or herds of animals in nature. The basis for nearly all flock algorithms is derived from the work of Craig Reynolds and his seminal work “Flocks, Herds, and Schools: A Distributed Behavioral Model” [36]. In [36], Reynolds proposes that the global behaviors of a flock of birds can be achieved in a virtual environment
by having each bird agent (or “boid”) adhere to three simple rules as it flies. These three rules are:

1. **Collision Avoidance**: Each boid avoids colliding with nearby flockmates by pulling away from any nearby boid that is less than some threshold distance away.

2. **Velocity Matching**: Each boid modifies its velocity to match the velocity of its nearby flockmates.

3. **Flock Centering**: Boids modify their velocity to stay close to nearby flockmates—essentially attempting to move toward the centroid of its \( k \) nearest flockmates.

The overall flocking algorithm can be summarized as follows:

```plaintext
create N boid agents and initialized with random position and velocity
for each boid:
    vAvoid = collision avoidance force
    vMatching = velocity matching force
    vCentering = flock centering force
    boid.velocity += (c1 * vAvoid) + (c2 * vMatching) + (c3 * vCentering)
    boid.position += boid.position + boid.velocity
done
```

Listing 6.2: Pseudocode of the generalized flocking algorithm

In the above algorithm, \( c1, c2, \) and \( c3 \) are constants that weight the relative influence of the three components of each boids’ velocity updating. Although not depicted above, also note that in most implementations the magnitude of each boids’ velocity is limited to some maximum to prevent chaotic, out of control behavior.

The basic boid flocking algorithm was initially developed by Reynolds for the purpose of computer graphics applications but has since been applied in a wide range of domains. These applications range from rendering life-like scenes of bat swarms and penguin flocks
in the Tim Burton movie Batman Returns, to experimental navigational and coordination systems for flocks of unmanned aerial vehicles (UAVs) [32]. We present here an adaptation of the basic boid flocking algorithm for performing partitional data cluster analysis.

6.3 A Flock Algorithm for Data Cluster Analysis

The ClusterFlockGPU algorithm is a flock algorithm for partitional data cluster analysis that is a synthesis of the flock-based clustering algorithms proposed in [7] and [45]. All three of these algorithms capitalize on the “birds of a feather flock together” effect to accomplish the task of partitional clustering. The main idea is that if a flock of boids is generated where each boid represents a data point from an input data set, then the natural flocking behaviors of the boids will eventually lead to the formation of sub-flocks such that boids in a given sub-flock are similar to one another, and dissimilar to those in all other sub-flocks. The primary advantage of this approach is that the number of clusters to be found does not need to be known at runtime. This value simply appears from the distributed actions of the flock agents.

Like the methods presented in [7] and [45], the ClusterFlockGPU algorithm uses a modified version of Reynolds’ algorithm to achieve a partitional clustering effect. With each boid representing a data point from the input data set, we repeatedly apply the following rules to determine the motion of the boids in a 2-dimensional virtual environment.

1. **Avoidance:** Each boid attempts to move away from the most dissimilar boid in its vicinity with the magnitude of the force proportional to how dissimilar the two boids are.

2. **Velocity Matching:** Each boid modifies its velocity to match the velocity of the boid which it is most similar to.
3. **Flock Centering**: Each boid moves toward the centroid of its $k$ most similar other boids with a force proportional to the similarity between this boid and the most similar of its $k$ most similar other boids.

A pseudocode overview of the sequential ClusterFlockGPU algorithm is given in Listing 6.3. We will examine the GPU-based implementation in section 6.4.

```plaintext
parse input data set and generate one boid per data point
calculate pair-wise similarity between all boids
determine each boids $k$ most similar boids
while (maximum number of iterations not met)
  for each boid:
    calculate avoidance force ($v_{Avoid}$)
    calculate velocity matching force ($v_{Matching}$)
    calculate flock centering force ($v_{Centering}$)
    boid.velocity += $c_1 * (v_{Avoid} * (1 - sim(boid, leastSimilarNearbyBoid))$
    boid.velocity += $c_2 * v_{Matching}$
    boid.velocity += $c_3 * (v_{Centering} * sim(boid, mostSimilarBoid))$
    if needed, limit boid.velocity to some maximum magnitude
    boid.position = boid.position + boid.velocity
  done
if current iteration > Omega
  shrink environment
  decrease maximum velocity
done
extract clusters
```

Listing 6.3: Pseudocode of a sequential version of the ClusterFlockGPU algorithm where $c_1$, $c_2$, and $c_3$ are constants that weight the relative influence of each flocking rule on boids' velocity.

The ClusterFlockGPU algorithm works by first generating a population of boids, each representing one of the input data points. We then calculate the similarity between each pair of boids by applying some similarity metric. A brief overview of similarity metrics is offered in section 6.3.1. Suffice it to say that there is some function $similarity(boidA, boidB)$ that generates a real value in the range $[0, 1]$ that signifies the similarity between the data
represented by \textit{boidA} and \textit{boidB} (0 signifying \textit{boidA} and \textit{boidB} completely different, and 1 indicating \textit{boidA} is identical to \textit{boidB}).

For some predefined number of iterations, the flocking rules presented above are applied to each boid. When the number of completed iterations is greater than the parameter \(\Omega\), the environment of the boids is shrunk by a small amount each iteration and the boids’ maximum velocity is incrementally reduced each iteration. This is a unique feature of the ClusterFlock-GPU algorithm and we have found it to be helpful in avoiding the situation where multiple clusters of what should be one form. By slowly shrinking the boids environment, more boids are brought into close proximity of one another allowing for similar sub-flocks to merge more easily and causing dissimilar boids to be repelled from one another.

When the maximum number of iterations has been reached (our termination condition), boids will have separated into discernible sub-flocks which can be interpreted as clusters. We then use a simple agglomerative algorithm to extract cluster membership from the positions of boids in their virtual environment. Starting with a randomly chosen boid not already assigned to a cluster, we group all boids whose euclidean distance from the selected boid is less than some threshold distance into a cluster. This process continues by selecting another unassigned boid and repeating the grouping process until all boids have been assigned to a cluster.

\subsection{Similarity Metrics}

The purpose of a similarity metric is to produce a numerical representation of the similarity between two data points. A large number of similarity metrics have been proposed, all with their strengths and weaknesses. In fact, the selection of a similarity metric can be the most
difficult aspect of performing data cluster analysis as the selected metric must be suited for the size and dimensionality of the data set being clustered. We present below a brief overview of the three similarity metrics that the ClusterFlockGPU algorithm can use.

**Euclidean**
Also called “as the crow flies” or 2-norm, this distance measure is given by the Pythagorean formula. It reports the distance between two data points in $n$-dimensional feature space which is then normalized to the range $[0, 1]$. Euclidean similarity is very simple to implement and for certain data sets offers an acceptable representation of similarity between data points. Euclidean distance does not work well on high-dimensional data sets and also does not capture the “shape” of data points, only their magnitude.

**Cosine**
Cosine similarity is very commonly used in document clustering applications. Essentially, cosine similarity gives the cosine of the angle between two data vectors. That is, with data points conceived of as vectors in $n$-dimensional feature space, cosine similarity of two data points will equal 1 if the angle between them is 0 (indicating vectors are identical), and 0 if the angle is 90 degrees (indicating no shared features). Cosine similarity is a good choice for data sets where the magnitude of data points’ features are not important or not wanted when considering similarity.

**Pearson’s Correlation**
The correlation between two data objects is a measure of the linear relationship between the features of the objects. Pearson’s correlation gives a comparison between two data objects that takes into account the “shapes” of the data points. Correlation metrics have built-in normalization and are often used when the shape of data points is more important than the magnitude.

### 6.4 The ClusterFlockGPU Algorithm

Listing 6.4 gives a pseudocode overview of the GPU implementation of the ClusterFlockGPU algorithm. Once again, we use the notation $(C/G - threadCount)$ to indicate where the step takes place (CPU or GPU) and how many concurrent threads are used. Figure 6.1 depicts the control flow of ClusterFlockGPU diagrammatically.
(C) parse input data set
(C) generate one boid per data point,
(C) initialize all boids with random position and velocity
(C) copy boids to GPU
(C) copy input data set to GPU
(G - numBoids^2) calculate pair-wise similarity between all boids
(G - numBoids) determine each boids k most similar boids
   while (maximum number of iterations not met)
      (G - numBoids) find each boid’s most dissimilar neighbor
      (G - numBoids) apply flocking rules and update boid positions
      (C) if current iteration # > Omega
         (C) shrink environment
         (C) decrease maximum velocity
   done
(C) extract clusters

Listing 6.4: Pseudocode of the GPU-based version of the ClusterFlockGPU algorithm
6.5 Results and Analysis

To assess the performance of the ClusterFlockGPU algorithm, we compare it to the PSO clustering algorithm given in [26], the ant colony clustering algorithm given in [15], and the Data Swarm Clustering (DSC) algorithm given in [45] (one of the algorithms on which ClusterFlockGPU is based). We also compare the results of ClusterFlockGPU to the simple
6.5.1 Data Sets

Because [26], [15], and [45] all provide results for the performance of their respective algorithms on the Iris and Wisconsin Breast Cancer data sets, these are the data sets adopted for testing here. Both data sets were downloaded from the UCI Machine Learning Repository ([16]).

1. **Iris** - Fisher’s Iris data set is perhaps one of the most well known in machine learning and pattern recognition literature. The data set is comprised of 150 instances corresponding to three species of iris flowers. Each instance has 4 real valued attributes and one class attribute. The class attribute designates each instance as being of the species Setosa, Versicolour or Virginica. For more information refer to [12].

2. **Wisconsin Breast Cancer (WBC)** - This data set consists of 699 instances corresponding to profiled breast tumor cells. Each instance has 9 numerical attributes and one class designation. The 9 attributes describe various characteristics of the profiled cells. The class attribute designates benign or malignant to the data point. For more information refer to [23].

6.5.2 Test System

Results presented in Section 6.5.3 were gathered on two different computers. Results for the Weka implementations of traditional classifications algorithms were gathered on an Apple MacBook Pro laptop with a 2.53 GHz Intel Core 2 Duo processor and 4GB of DDR3 memory. The AntMinerGPU algorithm was run on a desktop computer system with a 2.4 GHz Intel Core 2 Quad processor and 3 GB of DDR3 memory. The GPU device used was an NVIDIA Tesla C1060. We feel a valid comparison can be made between the results gathered on these two systems as their hardware configurations are similar. Also, because in the case
of the GPU-based algorithm, very little work is being done by the CPU and as such the configuration of the GPU-based computer system (CPU specs, memory type, etc...) has little bearing on the results.

6.5.3 Performance Analysis

We begin our analysis of the performance of ClusterFlockGPU by assessing its running time on data sets of varying sizes and dimensionality. To this end, we run the algorithm on differently sized synthetic data sets of randomly generated values. Running ClusterFlockGPU on these random data sets serves two purposes. First, it allows us to generate data sets of arbitrary size such that we can compare the effect of data size and dimensionality on performance. Second, if the algorithm is performing a true clustering, no distinct clusters should emerge from the random data as we assume no strong groupings of data points to be present. Our results for this experiment are shown in Figures 6.2 and 6.3.

Figure 6.2: Performance analysis of the ClusterFlockGPU algorithm. Graph (a) illustrates the impact of data size on overall running time, while graph (b) shows the effect of dimensionality on overall running time
Figure 6.2(a) shows that running time for ClusterFlockGPU is constant with respect to dimensionality. We also see that the running time of the ClusterFlockGPU algorithm scales in a roughly linear fashion with the size of the input data set. A study of the source code reveals that the running time of the algorithm is dominated by the time spent identifying the nearest neighbors of each boid as needed for calculating the avoidance force. This step is completed by having every boid calculate its distance to every other boid and those within some threshold distance are taken to be “nearby.” This step has complexity $O(n^2)$ when performed sequentially but is carried out in roughly linear time in our implementation because we use $n$ concurrent threads. This is an admittedly naive implementation strategy and will not scale well when data sets grow beyond 15360 elements (15360 being the maximum number of concurrently executing threads on the Tesla C1060 GPU). We will discuss in Section 6.6 a new implementation strategy that may be used to reduce the complexity of this step of the algorithm.

Figure 6.3 shows the number of clusters that are formed when ClusterFlockGPU is run on random synthetic data sets of varying size. We find that the number of clusters is large in each case showing that no strong clusters are found in the data sets.

<table>
<thead>
<tr>
<th>Synthetic Random Data Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instances:</strong></td>
</tr>
<tr>
<td># of Clusters</td>
</tr>
</tbody>
</table>

Figure 6.3: The number of clusters founds by ClusterFlockGPU in synthetic data sets of randomly generated data.
6.5.4 Comparison to Other Methods

To compare the performance of ClusterFlockGPU to the other algorithms under consideration we use the F-Measure, a commonly used measure of accuracy in the machine learning and information retrieval literature. F-Measure can be defined as follows. For some data set \( X \), let \( C \) denote the set of computed clusters, \( C_i \subseteq X \), and \( T \) the set of labels \( t \) of the known classes of the dataset. Furthermore, let \( N_{ti} \) be the number of data items of class \( t \) within cluster \( C_i \), \( N_i \) the size of cluster \( C_i \), and \( N_t \) the number of data points known to be of class \( t \).

F-Measure reports the sum of the the harmonic mean of precision and recall of each of the generated clusters as shown in equations 6.3 and 6.4. F-Measure considers the purity of each cluster with the precision measure \( \text{Prec}(t, C_i) = \frac{N_{ti}}{N_i} \), and also how much of the data of class \( t \) is contained within cluster \( C_i \) with \( \text{Rec}(t, C_i) = \frac{N_{ti}}{N_t} \).

\[
F_M(t, C_i) = \frac{2 \cdot \text{Prec}(t, C_i) \cdot \text{Rec}(t, C_i)}{\text{Prec}(t, C_i) + \text{Rec}(t, C_i)} \tag{6.3}
\]

\[
F\text{Measure}(C) = \sum_{t \in T} \frac{N_i}{|X|} \max(c \in C_i) \{F_M(t, C_i)\} \tag{6.4}
\]

Because ClusterFlockGPU is a stochastic algorithm, results can vary from run to run. For this reason, the values presented in Figures 6.4 and 6.5 are averages over 10 independent runs on the given data set. Comparison values for PSO, Ant-Based, and DSC clustering algorithms are taken from [26], [15], and [45] respectively.

ClusterFlockGPU proves to be a very competitive method for clustering the Iris data set. We see however that on the WBC data set, the ClusterFlockGPU suffers from the same
6.6 Next Steps

We present here two improvements to the ClusterFlockGPU algorithm.

6.6.1 Improved Neighbor Detection

The largest bottleneck of the ClusterFlockGPU algorithm is the neighbor detection method that performs an “all vs. all” comparison of boids to determine which boids are “nearby”
each other. In order to improve the running time of the neighbor detection phase, it may be possible to borrow a commonly used technique from molecular dynamics (MD) algorithms.

Computational MD algorithms are characterized by computing a very large number of pair-wise interactions between particles. As such, there is a need to maintain a list of each particle’s neighbors. In systems with a very large number of particles, doing an “all vs. all” neighbor detection approach like that used in CluserFlockGPU is unacceptable with regard to running time.

To accelerate the neighbor detection process, MD algorithms such as that described in [1] use a classic trick that we propose here for use in ClusterFlockGPU. The basic idea is to define two values, \( r_{max} \) and \( r_{cut} \) such that \( r_{cut} \) is the threshold distance that specifies if two particles are neighbors and \( r_{max} > r_{cut} \). With these values set, each particle maintains a list of other particles that are less than \( r_{max} \) distance away and only needs to determine if members of this subset are within \( r_{cut} \) and are thus “nearby.” The list of particles within the \( r_{max} \) threshold only needs to be updated when any particle has moved more than \( \frac{1}{2}(r_{max} - r_{cut}) \) distance. By employing a neighbor detection scheme such as this, we believe the overall running time of the CluserFlockGPU algorithm could be accelerated.

### 6.6.2 Improved Cluster Extraction

ClusterFlockGPU uses a simple agglomerative algorithm is used to extract cluster membership from the boids’ final positions in their virtual environment. Starting with a randomly selected boid, all other boids whose euclidean distance from the selected boid is less than some threshold distance are grouped into a cluster. Figure 6.6 shows a clear example of how this cluster extraction method produce erroneous results.
In this case, we can clearly see that three distinct clusters have formed. However, in the case of two clusters in the top of the environment, the distance separating the two clusters is less than the threshold distance for cluster extraction ($T_c$). Because of this, depending on which boid is selected as a cluster center, some of the red boids may be grouped with the blue boids or vice versa. One possible solution for this problem would be to post-process the results of CluserFlockGPU with a DBSCAN-type algorithm. By inputting the final positions of boids in 2-D space to a density-based clustering algorithm, cluster extraction could quickly and accurately be performed.
Chapter 7

Conclusion

7.1 Summary

In this work we explored the features of swarm intelligence systems that make them well-suited for GPU-based computing. Two swarm intelligence data mining algorithms were implemented with CUDA to show the efficacy of General-Purpose GPU (GPGPU) computing for two types of swarm intelligences: ant colonies and flocks. In the case of the AntMinerGPU algorithm presented in Chapter 5, we saw a marked improvement in overall running time as compared to the CPU-based AntMiner+ algorithm on which it is based. This indicates that GPGPU computing may be able to provide for faster and more accurate ant colony optimization algorithms for a range of other problems. While we are unable to determine if the ClusterFlockGPU algorithm described in Chapter 6 runs faster than the Data Swarm Clustering algorithm on which it is based, we do show that it is able to produce slightly better clusterings.
7.2 Future Research

One very promising area of research in the intersection of GPGPU computing and swarm intelligence is in GPU-based particle swarm optimization (PSO) algorithms. Because these systems are quite similar to ant colony optimization systems in that they leverage repeated and probabilistic explorations of solution space by swarm agents, the running time of PSO algorithms are often directly related to the number of swarm agents used. There has not been much research carried out to determine the effect of using populations orders of magnitude larger (millions or billions) than currently proposed. It is possible that GPGPU computing could be leveraged to create efficient PSO algorithms with massive numbers of particles and the effect of very large populations on solution quality could be determined.

Another possible direction of research that could improve the efficiency and scalability of GPU-based computing is to use an API like the Message Passing Interface (MPI) to link together multiple GPU-equipped compute nodes. In this way, multiple GPU devices could be simultaneously leveraged to carry out computations too large for a single GPU. Alternatively, and as noted in section 5.4.2, it may be advantageous to run multiple instances of swarm intelligence algorithms, each on its own GPU, and then select the system which generates the best results as the final solution. In this way, parallel evolution of multiple swarm intelligence systems could be carried out and the best of all instances taken as the final solution.

7.3 Final Remarks

General purpose computing on GPU devices provides for massive amounts of parallel computing capabilities on inexpensive, or so-called “commodity” computer systems. A goal of
this project was to highlight the benefits of GPGPU computing and in doing so, illustrate the possibility of performing massively parallel computations on inexpensive computer systems. In doing so, we hoped to show that high performance computing can be made available to a broader user-based than is currently available due to the prohibitive cost of traditional high performance computer systems. As the GPGPU computing revolution that begun just 3 years ago continues to gain momentum, it is this author’s opinion that the GPGPU model for parallel computing will continue to become more and more prevalent in a wide range of computing applications.

The prospect of using GPGPU devices as the compute platform for swarm intelligence applications is indeed quite promising. By utilizing the massive multi-threading capabilities of GPGPU devices, swarm intelligence algorithms can feasibly achieve very large populations of agents. Furthermore, by allocating one or more threads per agent in a swarm intelligence system, implementations that more closely mimic the asynchronous nature of the systems on which these algorithms are inspired by can be achieved. The implications of being able to very accurately simulate large distributed systems are far reaching.
Appendix A

CUDA Technical Details and Example

We present here a basic example CUDA program in order to highlight the key features of this programming model.

In general, most CUDA programs follow the same basic control flow:

1. The host (CPU) initializes an array with processing data.
2. The array is copied to the global memory of the GPU device.
3. The host invokes a GPU “kernel function” and some number of threads concurrently execute the kernel function on the many cores of the GPU device.
4. The host copies the results generated by the kernel back to system memory.

The CUDA programming model exhibits a Single Program Multiple Data (SPMD) execution pattern. Each thread executes the same code, but based on its unique ID number may have a completely different control flow than other concurrent threads executing the same kernel. In this way, threads can be assigned different tasks based on their ID number.

To illustrate these concepts, a very simple CUDA program is shown in Listing 3.1. The purpose of this program is to square every element in an array of integers, writing the result
#include <stdio.h>
#include <cuda.h>

// GPU Kernel Function
__global__ void squareArray(int *array_d){
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    array_d[index] = array_d[index] * array_d[index];
}

int main(void){
    // Pointers
    int *array_d; // Pointer into device memory for the array
    int *array_h; // Pointer into host memory for the array

    // Number of elements we want in the array
    int N = 1024;

    // The size of the array in memory
    size_t arraySize = N * sizeof(int);

    // Allocate space in host memory for the array
    array_h = (int *)malloc(arraySize);

    // Initialize the array with values from 0 to N
    for (int i = 0; i < N; i++){
        array_h[i] = i;
    }

    // Allocate space in GPU memory for the array
    cudaMalloc((void **) &array_d, arraySize);

    // Copy the array from CPU memory to GPU memory
    cudaMemcpy(array_d, array_h, arraySize, cudaMemcpyHostToDevice);

    // Call the GPU kernel function
    dim3 dimGrid(2,1); // use a 2 x 1 grid
    dim3 dimBlock(512,1); // with 512 x 1 blocks
    squareArray<<<dimGrid, dimBlock>>>(array_d);
Listing 3.1: A simple CUDA program for squaring elements of an array

This program begins by initializing two pointers. We use `array_d` as a pointer into GPU device memory and `array_h` as a pointer into host, or CPU memory. In order to give the kernel function data to process, we begin by initializing `array_h` with values 0 to $N$. This is accomplished on lines 15 through 27 where we allocate memory, and write sequential values. On line 30, the `cudaMalloc()` function is called and `arraySize` memory locations are allocated in GPU memory with the first location being pointed to by `array_d`. Line 33 copies the data stored in `array_h` to the location in GPU memory pointed to by `array_d`. Now that the data to be processed is copied to the GPU device, the GPU kernel function is invoked on line 38. The details of this call requires a discussion of CUDA thread organization.

Thread organization is particularly important in CUDA development. As depicted in Figure A.1, GPU threads are organized into 1-, 2-, 3-, or 4-dimensional “blocks,” and blocks are organized into a 1- or 2-dimensional “grid.” With this organizational structure, each thread can be identified by its unique location in this overall matrix of threads. This location can be computed by the thread via the use of three built-in variables: `blockIdx`, `blockDim`, and `threadIdx`. `blockIdx` contains the index of a thread’s block within the grid in the x and
y directions. `blockDim` contains the size of blocks in the x, y, z, and w directions. And `threadIdx` contains the index of a thread within the block in the x, y, z, and w directions.

Figure A.1: Thread organization in CUDA [29]

In the case of our example code, we use two structures, `dimGrid` and `dimBlock` to request a particular arrangement of threads. On line 36, `dimGrid` specifies we wish to have a one dimensional grid of blocks (2 x 1), and on line 37, `dimBlock` specifies that the blocks within the grid should be 512 x 1 threads. This arrangement of threads is a good match for our problem as we are operating on a one-dimensional array of 1024 values. It should be noted that blocks can be one, two, three, or four dimensional, and the dimension is usually chosen to be that which best fits the specifics of the given problem.

When `squareArray()` is invoked on line 38, 1024 concurrent threads are created on the GPU and each executes the logic defined on lines 4 through 7. Here, each thread uses the built-in variables to identify itself within the overall grid. This allows each thread to determine which item in the array it is responsible for. With this index now computed, each
thread squares that value and writes it back to the original location. After the invocation of squareArray, the results of the kernel function (which were written back into \textit{array.d}) are copied to CPU memory on line 41. At this point, the results could be printed to the screen or operated on further. Finally, we call \textit{free()} and \textit{cudaFree()} on lines 48 and 49 to free the memory locations that were used.
Bibliography


