2015

**Accelerating reconfiguration of network anomaly detection systems**

William C. Edmonds Jr.

Follow this and additional works at: https://louis.uah.edu/uah-theses

**Recommended Citation**

https://louis.uah.edu/uah-theses/125

This Thesis is brought to you for free and open access by the UAH Electronic Theses and Dissertations at LOUIS. It has been accepted for inclusion in Theses by an authorized administrator of LOUIS.
ACCELERATING RECONFIGURATION OF NETWORK ANOMALY DETECTION SYSTEMS

by

WILLIAM C. EDMONDS, JR.

A THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in The Department of Computer Science to The School of Graduate Studies of The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2015
In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

William C. Edmonds, Jr.

11/9/2015 (date)
THESIS APPROVAL FORM

Submitted by William C. Edmonds, Jr. in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in Computer Science.

Dr. Sun-J Kim 11/9/2015
Dr. Letha Etzborn 11/9/2015
Dr. David Coe 11/9/2015
Heggere S. Ranganath 11-9-2015
Dr. Heggere S Ranganath

Dr. Sundar A. C. 11-11-2015
Dr. Sundar Christopher

Dr. David Berkowitz 11-11-15
Dr. David Berkowitz

Committee Chair
Department Chair
College Dean
Graduate Dean
ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree Masters of Science
College/Dept. Science/Computer Science
Computer Science

Name of Candidate William C. Edmonds, Jr.

Title Fast Reoptimization for Improved Network Security

Reducing the time required to keep an anomaly-based network intrusion detection system (NIDS) up-to-date with the continuously changing web server features is of the utmost importance to keep intrusion detection as accurate as possible.

Web applications and content changes frequently on web servers to keep their users interested and returning. This frequent updating poses an interesting problem with regards to anomaly-based network intrusion detection systems (NIDS) as they rely on knowing what normal traffic patterns look like. Retraining and reconfiguration to adapt to new applications or features of a website is highly advised to maintain accurate detection performance. Training an anomaly-based NIDS can be a major strain on computational resources and a time sink. Through the use of cost efficient, off-the-shelf hardware, the training of an anomaly-based NIDS may be offloaded from a web server’s central processing unit (CPU) onto a single instruction, multiple data (SIMD) architecture device and completed hundreds of time faster. Some anomaly-based NIDS algorithms lend themselves to other approaches like a genetic algorithm for accelerated training. An automated search through the input parameter solution space may completed in an efficient manner to reduce the time required for choosing the appropriate input parameters.
In this thesis, the use of graphics processing units (GPU) and a genetic algorithm search heuristic are studied to accelerate the reconfiguration time of two different anomaly and payload-based NIDS. A demonstration of a SIMD implementation is shown on the Payload-based Dispersion (PBD) algorithm, highlighting the data parallel computational design and how to distribute workloads to be processed by individual threads. The SIMD implementation of the PBD algorithm achieves comparable results to other related works, speeding up certain portions of the algorithm from approximately 28 to 55 times. The Lightweight Stateless Payload Inspection (LiSPI) algorithm is selected to show a different use of data parallel computation and a genetic algorithm to achieve fast reconfiguration times. The LiSPI algorithm requires appropriate input parameters to achieve accurate detection performance. The default parameters required to run the algorithm results in a solution space of approximately 43 million combinations, which may grow depending on the parameters provided. A brute force search using the GPU yields optimal results in minutes, where as a serial implementation would require weeks of computation. The genetic algorithm approach also results in acceleration performance similar to the GPU brute force.

Abstract Approval: Committee Chair

Dr. Sun-il Kim

Heggere S. Ranganath

Dr. Heggere S. Ranganath

Department Chair

Graduate Dean

Dr. David Berkowitz
I would like to thank my advisor, Dr. Sun-il Kim, for guiding me through my collegiate career, setting me up for success, and sharing knowledge beyond academia. I would like to thank the committee members, Dr. Letha Etzkorn and Dr. David Coe for their help and guidance; my fellow research students and peers, Chockalingam Karupanchetty, Erik McIntyre, Reece Johnston, and Martin Cox for their input and contribution; and the admin staff Mrs. Betty Nelson and Mrs. Maryann Bierer for their help from day 0 to present.

I am grateful for my family for their support and encouragement through this endeavor; I love you, all.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Symbols</td>
<td>xii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Network Intrusion Detection Approaches and Related Works</td>
<td>5</td>
</tr>
<tr>
<td>2.1.0.1 Signature-based NIDS</td>
<td>5</td>
</tr>
<tr>
<td>2.1.0.2 Anomaly-based NIDS</td>
<td>6</td>
</tr>
<tr>
<td>2.1.0.3 Header and Payload-based NIDS</td>
<td>8</td>
</tr>
<tr>
<td>2.1.0.4 Accelerating NIDS with GPUs</td>
<td>9</td>
</tr>
<tr>
<td>2.1.0.5 The Use of Genetic Algorithms in NIDS</td>
<td>11</td>
</tr>
<tr>
<td>2.2 IDS Models</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Statistics-based Algorithm</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1.1 Bin Counting</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1.2 Bin Statistics</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1.3 Packet Scoring</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1.4 Training Phase</td>
<td>13</td>
</tr>
</tbody>
</table>
2.2.1.5 Detection Phase ........................................ 14

2.2.2 LiSPI Algorithm ............................................. 15

2.2.2.1 Bitmasks ................................................. 15

2.2.2.2 Bin Counting Stage ..................................... 16

2.2.2.3 Quantization and Mapping Stages ........................ 16

2.2.2.4 Training Phase .......................................... 17

2.2.2.5 Detection Phase .......................................... 18

2.2.2.6 Choosing Masks .......................................... 19

3 Acceleration with SIMD ...................................... 21

3.1 nVidia’s CUDA ................................................. 22

3.2 Payload-based Dispersion (PBD) Algorithm .................... 23

3.2.1 Bin Counting .............................................. 23

3.2.2 Bin Statistics .............................................. 24

3.2.3 Packet Scoring ............................................ 26

3.2.4 Evaluation Results ......................................... 28

3.2.4.1 Speedup ............................................... 29

3.2.4.2 Memory Footprint ..................................... 30

4 Alternative Approaches and Solutions ......................... 32

4.1 GPU Solution ................................................... 33

4.1.0.3 Mask Pair Reduction .................................. 33

4.1.0.4 Memoization ............................................. 34

4.1.0.5 Bitmap Generation ...................................... 35
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Feature bitmap: Normal, False Positive, and Attack features.</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Bin counting data structures and block assignment. The shaded portions of the $p$ matrix demonstrates varying lengths of packets.</td>
<td>25</td>
</tr>
<tr>
<td>3.2 This figure shows thread level workloads divided within the $b$ matrix, thread storage in the $k$ matrix, and the resultant vectors.</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Generating the packet anomaly score thresholds are depicted above. Using a similar workload distribution</td>
<td>28</td>
</tr>
<tr>
<td>4.1 Each column in the feature matrix represents a bitmask and each thread is responsible for one pair of bitmasks to produce a corresponding bitmap during the training phase. Each index from each column is mapped using the quantization method($f(P_{i}), f(Q_{i})$) described in Chapter 2.</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Next generation selection: Crossover and Mutation.</td>
<td>40</td>
</tr>
<tr>
<td>4.3 Fitness Score vs. Generations for Web Traffic 1 data set (25k training set, population size 25, 100,000 generations shown).</td>
<td>44</td>
</tr>
<tr>
<td>4.4 Fitness Score vs. Generations for DARPA week 3 data set (20k training set, population size 25, 100,000 generations shown).</td>
<td>45</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Speed up over CPU implementation</td>
<td>30</td>
</tr>
<tr>
<td>3.2 Memory Footprint</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Run times in seconds.</td>
<td>44</td>
</tr>
<tr>
<td>4.2 The GA run times are shown for starting population sizes of 25 and 50, at which an elite solution could be found.</td>
<td>46</td>
</tr>
</tbody>
</table>
## LIST OF SYMBOLS

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Tuning parameter used in PBD method</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Smoothing factor used in PAYL method</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

The state of technology is ever changing and expanding to fit the needs and desires of its consumers. For businesses that provide access to their service or product via the Internet, keeping up with the competition requires providing their service and content with minimal service interruptions. To keep their services fresh, some businesses may opt to add new features to their website or application server to give their customers something new to explore or make their experience better on their website. As in any newly developed software, the newly added features may have hidden, unknown flaws that may leave the system and network vulnerable to an attack. Exploiting undiscovered flaws in software is known as a zero day exploit. Preventing zero day exploits is difficult, but appropriate secure coding habits and experienced review and unit testing may be used as preventative maintenance. It is difficult to guarantee that software is free of zero day exploits especially as new features are continually added and upgraded by not only the original developers, but any third party collaborators.

With regards to systems that connect to a network, anomaly-based network intrusion detection systems (NIDS) may be used to detect zero day attacks by rec-
ognizing anomalies in normal traffic flow. Basically, if network packets contain data that is not normally seen by the network, the network packets may contain code to perform reconnaissance searching for vulnerabilities or actual malicious code for exploitation. In a signature-based NIDS updates to a signature database are made once an attack has been realized and recorded, thus allowing for other systems to be defended against that particular attack. Having anomaly-based and signature-based systems work in tandem can provide a powerful security solution.

Unfortunately, a major bottleneck for anomaly-based NIDS is the training phase of the algorithm. The training phase requires previous traffic that is known to be "normal" to the system and also requires time to train the system. The training algorithms for many anomaly-based NIDS incur a large computational overhead. To optimally tune the system, these algorithms may take several weeks of time on a modern high-end computer. Many modern applications, such those utilized by social media providers or any new and upcoming web service provider, may be updated or have new features added daily. These transitional services may also result in traffic fluctuations, necessitating reconfiguration and tuning of NIDS. Training delays of weeks or even days are unacceptable.

Collecting traffic data that represents the network traffic of the updated system also takes time. There has been work done with regards to artificial training traffic to help get the training phase started until enough real traffic has been collected to better provide a more accurate service from the anomaly-based NIDS [CHOKS]. Retraining the NIDS repetitively does not only come with a possible cost of intermittent service, but also a large computational cost which could affect the performance of the system.
it is trying to protect. Offloading this work onto a separate device makes a significant impact. In the case of web servers, the computational resources (and their associated thread-pools) should be dedicated to server threads that handle incoming requests and provide its services. Instead, the NIDS tuning algorithm would compete for the same resource(s).

In this thesis, a paradigm for offloading and speeding up the computation necessary for frequent reconfiguration of NIDS is presented. Leveraging low-cost Single Instruction, Multiple Data (SIMD) architectures allows for cost-efficient implementations and is largely available in commodity-off-the-shelf products like graphical processing units (GPUs). For example, adding a GPU to a web server or hypervisor frees the computational burden of retraining the NIDS from the CPUs and onto the GPUs. After initial setup of the anomaly-based NIDS, a sliding window of time (i.e. hours, days, weeks) may be used to constantly keep the system trained and up-to-date with the current state of the system. After each retraining, the GPU passes back the new thresholds and parameters required by the detection phase of the NIDS. Depending on the size of the training traffic sets and if the algorithms lend themselves to being performed in parallel, retraining may be accomplished in the matter of minutes versus days or weeks.

Two different anomaly-based NIDS algorithms were chosen for this thesis. The first algorithm is used to demonstrate the speedup efficiencies gained from a SIMD implementation of the original algorithm. The second algorithm may also be implemented in a similar fashion as the first algorithm for use in SIMD architectures, but the algorithm presents other opportunities for further optimizations through the
use of SIMD architectures and search heuristics. These opportunities for further optimizations are investigated in this thesis through the use of a parallel brute force search and a genetic algorithm to find optimal input parameters to reduce computational and system administrator resources.
CHAPTER 2

BACKGROUND

2.1 Network Intrusion Detection Approaches and Related Works

Network intrusion detection systems (NIDS) can be generally classified into two classes: anomaly-based and signature-based. Furthermore, inspection of the packet headers versus payloads are other well known approaches for NIDS. Each approach has its strong and weak points, each should be investigated to see if one or any combination of the approaches provide the amount of security required based on the possible consequential performance trade-offs.

2.1.0.1 Signature-based NIDS

Signature-based NIDS identify attacks or malicious events by use of signatures. Signatures of attacks may be created by system administrators or downloaded from shared, trusted databases of malicious event signatures. Signatures may consist of particular combinations of byte strings present in a network packet that have shown to be malicious in nature. By identifying the common byte string that identifies the malicious attack, the byte string may then be hashed with a hashing algorithm to generate a unique signature. Signatures provide an extremely low false positive rate as
the signatures are unique to the particular malicious event identified, thus allowing for low service interruption due to false alarms. New attacks or new iterations of previously known attacks slip by signature-based NIDS because the key steps to compromising its target have not yet had a signature created for it. If the attack goes completely undetected by the system administrators and any other security systems in place, the system remains compromised and the attacker(s) may continue malicious activities until found. Once attacks have been identified and a signature of the attack is created, updating the signature database as soon as possible is of the utmost importance, as any service providers may have multiple systems that may also be compromised by the same attack.

2.1.0.2 Anomaly-based NIDS

Anomaly-based NIDS detect anomalies in the network traffic based off a sample set of previous network traffic. Anomalies can range from actual attacks on the system, exploit reconnaissance, or benign traffic fluctuations due to new features added to a web server on the network and has become popular via social media. Upon detection of an anomaly, the anomalous packets may be dropped or redirected for further analysis and possible signature creation for signature-based NIDS if found to be malicious. Higher false positive rates compared to signature-based detection is a weakness for anomaly-based approaches, but continuous and appropriately scheduled retraining of the network traffic aids in reducing the number of false alarms. Another limitation for anomaly-based NIDS is the network traffic that is required for training of the system. The network traffic must continuously be stored and if the network
traffic is not sanitized prior to training the system, the NIDS may not detect attacks.
The statistics-based algorithm discussed in this thesis is able to train with a traffic set
containing attacks and is resilient enough to go on to detect those attacks with ap-
propriate parameter tuning [1]. Similar approaches to statistics-based approach can
be found in the literature utilizing byte frequency distribution within the payload
includes [2–4]. Service-specific IDSs have been proposed where the anomaly score
of a request is calculated using three properties: type of the request, length of the
request and payload distribution [3]. In this approach, if the length of the request is
longer than the average length, it is likely to be an attack, as buffer overflow attacks
often contain a large number of null characters. In the case of a model for character
distribution, the ASCII characters are grouped into six categories. Then a single
uniform distribution model of these six groups for all requests of one service over all
possible payloads’ length is computed. Finally, a chi-square test is performed against
this model to calculate the anomaly score of test requests.

A lot of research regarding web logging and parameter inspection can be found
in the literature [5–8]. Investigation of anomalous activity in HTTP requests and
the parameters used to pass values to server-side programs were performed in [9].
The GET requests are recorded in the web server’s access logs and were used to
extract features such as the presence or absence of attributes (parameters), length
of attributes, character distribution in the attributes, the order of attributes, access
frequency, inter-request time delay, and a few others.

Variables that may affect the performance of anomaly-based system can be
due to the fact of content changing on the network, such as web pages, and possibly
the use of data sets that are not pertinent to traffic in the wild. Concept drift is when web pages are frequently modified according to the needs of the user (e.g. fields, links, and icons), which can influence the patterns of the traffic [10]. An evaluation of the anomaly-based intrusion detection methods using public data sets was conducted reviewing their accuracy and ability to reflect real life conditions (traffic in the wild) [11].

2.1.0.3 Header and Payload-based NIDS

Header-based and payload-based inspection are other approaches one may take when designing a NIDS. The trade-off between accuracy, complexity, and detection delay between header-based and payload-based anomaly detectors is discussed in a comparative study in [12]. Header-based approaches look at information like the IP and MAC addresses of the source and destinations along with port information, essentially covering everything at the network level. Black or white lists may then be made in the firewall of a network to efficiently mitigate particular attacks, but attacks coming from within the may still require a header-based NIDS for better coverage of the network [13–15].

Payload-based inspection focuses mainly on the payload of network packets as this is where the application or system exploits will reside. Finding attacks within the payload allow for signature-based systems to update their signature database and anomaly-based systems to tune its parameters to find similar anomalous payloads, but also work with header-based approaches on classifying the headers that were associated with those tainted payloads. [16–19].
2.1.0.4 Accelerating NIDS with GPUs

In the past few years, many works have been published with regards to leveraging cost effective GPUs to accelerate monitoring and analyzing network systems [20–25]. The works found concentrate more on pattern matching within packets in order to secure the network system from known attacks, but to the best of my knowledge, leveraging GPUs for use in anomaly-based detection is not thoroughly investigated. Each paper found and presented here all stressed the benefits of offloading pattern matching processes from the main CPU to the workhorse device, the GPU. The main concern is the throughput when it comes to GPU-based pattern matching NIDS. The work from [26] ported the SNORT IDS to GPU achieving 16 Gbps on their first attempt and 30 Gbps with a best case scenario on their second attempt. Their approach used a string searching algorithm called the Aho-Corsaick algorithm, where a string is searched using a dictionary in a state-machine. The authors modified the algorithm such that on a failure to match, the thread terminates. The packet payloads were concatenated to keep the data within uniform sizes, which makes it more efficient for parallel processing. A thread was assigned to each byte within the packet data structure and traversed the state-machine for each subsequent byte until it either found a success or failure state. All successful matches for that batch are returned to the main memory for classifying the exact match stored in a look-up table.

In [27] a signature-based NIDS focusing on the header was investigated for acceleration with GPUs. The authors looked at fast port and keyword matching to
improve filter performance. The headers are stripped from the payloads and processed by individual threads, allowing thousands of network packet headers to be filtered in parallel.

The authors in [28] were able to achieve a 2.3 Gbps tested on pattern sizes of 4 bytes. The authors were able to implement a design where the pattern matching may happen concurrently with the host portion of the NIDS application. Using concurrency with the host application, the authors were able to hide the cost of some memory operations, but were also able to use faster memory exchanges with locked memory locations which allow for direct access from the device to the host memory. Their contribution showed it was possible to achieve comparable speeds to batched memory approaches with regards to GPU signature matching.

For many anomaly-based NIDS, the major area needing acceleration is during the training phase of the algorithm. With the discussed anomaly-based algorithms in this thesis, the detection phases are actually just as fast, if not faster, on a thread on the CPU due to the heavy memory transfer operation cost on the GPU. Signature-based NIDS do not require a training phase, only a dictionary with all signatures to be searched. Depending on the traffic volume, attempting to run a signature detection algorithm on a live system may severely hinder the quality of service. Further research is required to fully understand the trade-offs between the SIMD NIDS approaches.
2.1.0.5 The Use of Genetic Algorithms in NIDS

2.2 IDS Models

In this section, a brief overview of the different IDS algorithms utilized to illustrate the problem addressed in this thesis is presented.

2.2.1 Statistics-based Algorithm

The statistics-based approach in [1] inspects byte frequencies of network packet payloads to detect anomalies. A histogram consisting of 256 bins (each representing a possible byte value) is created for each packet that passes through the NIDS. Each bin is then compared to a range that has been calculated to be normal. If a bin is out of the normal range, a counter is incremented each time to keep track of the packet’s total anomalies (maximum of 256 anomalies per packet). In this algorithm there are two main phases: training and detection. In the training phase of the algorithm, sanitized traffic sets and attack data sets are used to determine the average and standard deviation of anomaly scores. During the detection phase, if the inspected packet falls out of the calculated range of normal anomaly scores, the packet is flagged as anomalous. The DARPA network packet data set was used in the original literature resulting in 100% attack detection with a false positive rate per packet of 0.5% using only 10 thousand packets for initial training.

Reconfiguring the algorithm to perform more accurately is accomplished with scalar coefficients multiplied to the bin range allotted thresholds. A separate scalar
Each phase consists of multiple stages: bin counting, bin statistics computation, and packet scoring. Descriptions of each phase are discussed below.

### 2.2.1.1 Bin Counting

When a packet enters the NIDS, its headers are stripped leaving only the payload of 1460 bytes. As previously mentioned, a histogram is kept for each packet and each byte has a representative bin where the frequency of its occurrence is kept, which is termed bin counting. Once every byte has been counted, bin statistics computations may be applied to the packet’s histogram in the bin statistics stage.

### 2.2.1.2 Bin Statistics

Bin statistics computation encompasses taking the average and standard deviation per bin of every payload histogram within the chosen network traffic training set. The standard deviation is used for the thresholds around each bin average for bin counting in the detection phase. The standard deviations are applied to the computed average for each bin and two vector arrays are generated representing the upper threshold values for each bin and the lower threshold values for each bin.

### 2.2.1.3 Packet Scoring

After the bin statistics have been performed, packet scoring checks for any bins that are above or below its respective bin range for each packet. Any byte
frequencies that lie beyond the calculated thresholds are summed and termed the packet’s anomaly score. Anomaly scores may then have its own threshold calculated by averaging all anomaly scores of a given set of network packets and calculating the standard deviation. The customization of threshold values, as mentioned, may then be set by coefficient multipliers on a bin level and an anomaly score level in order to achieve better false positive and true positive performance.

2.2.1.4 Training Phase

The training phase stages consist of bin counting, bin statistics computation, and packet scoring. The training phase requires three sets of network traffic data: a sanitized training set, a sanitized testing set, and an attack data set. The training set provides the initial threshold vectors for each bin and anomaly score, the test set provides performance with regards to true negative and false positive anomalies, and the attack data set provides true positive and false negative anomalies.

First, bin counting and bin statistics are performed on the training data set with given scalar multipliers of 1.0 for both the bins and the anomaly scores. Thus, for each bin a threshold is generated and an overall threshold for anomaly scores is generated using the average and standard deviation for all the packets in the training data set.

Next, bin counting is performed on the testing data set to check against the generated thresholds per bin from the training data set. The anomaly scores are calculated per packet and then compared to the anomaly score threshold. If the individual test packet’s anomaly score is beyond the set anomaly threshold, the packet
is flagged as anomalous. Ideally, none of the test data set is flagged as anomalous, but packet size and infrequent, legitimate content within the payload may cause false positives.

Finally, the attack data set is put through the same motions as the test data set, except the final desired result would be to find that the majority of the packets contained in the attack data set is flagged as anomalous. Finding all packets within the attack data set to be anomalous is unrealistic as attacks need to have some legitimacy to them in order to at the least communicate and pass data to its target.

For better performance in lowering false positive rates and false negative rates, adjusting the multiplier coefficients may be required. Repeating the bin statistics and packet scoring stages is then necessary to gauge how well the tuning parameters performed with regards to false positive and true positives.

### 2.2.1.5 Detection Phase

The detection phase has precisely the same stages used to determine true positive and false positive performance except live network traffic packets passing through the NIDS are inspected. As packets flow through the NIDS, each packets goes through the bin counting and anomaly score phase, which are flagged as normal or anomalous. The system administrator then has the option to drop any packets that are flagged as anomalous or use inspection methods such as honeypots to observe the behavior of the traffic.
2.2.2 LiSPI Algorithm

The Lightweight Stateless Payload Inspection (LiSPI) algorithm is an anomaly-based NIDS described in [29]. This algorithm looks at specific combinations of byte frequencies and uses a quantization method to generate a 2-D bitmap which represents the shape, or features, of normal traffic data. Once the system is trained, packets passing through the NIDS will have its payload inspected, quantized, and compared to the pre-generated bitmap. If a packet is mapped to a position on the bitmap that has not been marked during training, then the packet is deemed anomalous and is dropped or further inspected. Using the DARPA network packet data sets in the original literature, a 100% per attack detection rate was achieved at a very low positive of 0.027% false positive rate per packet. Similar results were achieved during the experiments in this thesis.

Similarly to the previously described algorithm, there is a training and a detection phase. Both phases have similar stages consisting of: bin counting, quantization, and mapping. Differences occur at the end of each phase, which are described below. First, a description of how the LiSPI algorithm accomplishes inspection of various byte frequency combinations.

2.2.2.1 Bitmasks

The LiSPI algorithm organizes packet payloads into byte histograms and is able to inspect various combinations of bins through the use of bitmasks. Bitmasks have three possible values: 0, 1, or X, where X represents "don't cares" (1's or 0's).
These $8n$-bit masks are able to represent $n$-gram bins, but for the purposes of this thesis, we experiment with 1-gram bins and 8-bit sized masks. Given a 256 bin histogram for all 256 unique bytes that may occur in a packet, the bitmask patterns represent the bins of interest. For example the bitmask 00000010 represents the bin representing the byte value 2, while a bitmask of 000000X0 represents the bins that represent the two byte values 0 and 2. The sum of the bins represented by the given bitmasks are quantized and used as indexes for mapping onto a 2-D bitmap. For every $X$, the number of bins inspected goes up by a power of 2. In total there are $3^{8n}$ combinations, choosing the masks is the interesting problem this paper explores through the use of parallel processing and a genetic algorithm.

2.2.2.2 Bin Counting Stage

Bin counting is precisely the same process as mentioned in the previous algorithm. Each payload byte is tallied into a histogram of 256 bins and is stored into a matrix termed $b$ matrix. After this histogram is created, the sum of the bins that are represented by the bitmask chosen is stored as the bin sum for that packet.

2.2.2.3 Quantization and Mapping Stages

Once the bin sum is calculated for a mask, a quantization measure takes place to determine its particular axis coordinate on the 2-D bitmap. A single mask generates a single coordinate, thus two masks are required to be used on each payload packet. Various quantization methods may be used to produce the coordinates as mentioned in the original paper [29]. A simple example would be to use the modulo operator,
modding the bin sum with the size of one of the dimensions of the bitmap. A more specific description of the quantization used in this paper is explained below in the training phase section.

2.2.2.4 Training Phase

The training phase requires the same three data sets as the PBD algorithm, a training data set, a testing data set for false positive performance, and an attack data set for true positive performance. For the sake of this example, any arbitrary pair of masks may be used for the training phase and the 2-D bitmap is an 16 by 16 matrix initialized with zeroes. First, the training set goes through the bin counting process and outputs two bin sums, one for each respective mask. Then, the bin sums are quantized to generate the coordinate pair that maps to a position on the 2-D bitmap where the value is set to 1. In this thesis, the quantization formula is as such: the ratio of the current inspected packet’s bin sum and the max bin sum found in the training data set multiplied by the ratio \( \frac{16}{1} \). The process is repeated for every packet in the training set. Once all training packets have been mapped, the final 2-D bitmap is stored and represents the shape of normal traffic data using the given mask pair.

Next, the testing data set is processed and quantized in the same fashion as the training data set using the same mask pair, but instead of storing 1’s at the generated coordinates for each packet, if the position on the bitmap contains a 1 the packet is considered normal, while a 0 indicates an anomalous packet. False positive performance may then be determined by the ratio of anomalous packets to the number of packets in the testing data set. True positive performance may be calculated the
same way by performing the same steps as described above on the attack data set, and taking the ratio of anomalous packets to the total number of packets in the attack data set.

In the Figure 2.1, a feature bitmap is shown filled in after the training, testing, and attack data sets have been mapped. Gray, shaded boxes represent normal traffic data mapped by the training session using the training data set. Thus, a gray box indicates a 1 on the bitmap, while white boxes indicate a 0 or anomalous packet. Any packets from the testing data set that is mapped to a gray box is regarded as a true negative, as testing packets should be. If a testing data set packet is mapped to a white square, it is considered a false positive (indicated in the figure by a circle with a white background). If an attack data set packet is mapped to a white square, it is correctly indicated as anomalous (depicted in the figure as an X with a white background). An attack data set packet mapped to a gray square shows a false negative result (an X with a gray background in the figure) meaning an attack packet has passed itself as a normal packet.

Reduction of the false positives and false negatives are discussed later in this chapter.

2.2.2.5 Detection Phase

The detection phase is performed the same way during the false positive and true positive testing except on live traffic data passing through the NIDS. As a packet enters the NIDS, its histogram is created for its payload and the bin sums are calculated for the set mask pair. The bin sums are processed using the quantization
method described and are mapped to the stored 2-D bitmap created during the training phase. Given a 0 or 1 at the mapped location determines whether or not the packet is classified as normal or anomalous, respectively.

2.2.2.6 Choosing Masks

Reducing the false positives and false negatives may then be required. Re-performing the training phase with different combinations of mask pairs, changing the quantization method, or even changing the size of the bitmap to allow for a larger resolution representation of the feature space could result in better accuracy. Although the actual computational requirements of this algorithm may be made very

Figure 2.1: Feature bitmap: Normal, False Positive, and Attack features.
efficient and fast, choosing an effective mask pair is difficult. Extensive reviewing of
the network traffic data may be required along with many trial-and-error iterations for
choosing a good mask pair, but leveraging parallel processing and genetic algorithms
are shown in this thesis to provide substantial speed-up of the training phase and
finding effective mask pairs with high accuracy.
CHAPTER 3

ACCELERATION WITH SIMD

There are certain characteristics in algorithms that make them better suited for parallel computation on SIMD architectures. Investigating the functions in the NIDS algorithms from the previous chapter, some of the functions have loops that perform the same set of instructions for every given packet and the results for each loop are independent of any previous or subsequent results. This characteristic of independent loop operations allows the algorithm to be expressed in a data-parallel computational (DPC) manner, meaning each packet may be processed in parallel, running the same set of instructions and input parameters, providing the same results as if processed in a sequential manner. Another characteristic found in the described NIDS algorithms is that they require minimal memory operations after initialization and may operate within a near constant memory footprint. These characteristics are present in the discussed NIDS algorithms giving the opportunity for acceleration through the use of GPUs.
3.1 nVidia’s CUDA

In order to implement the NIDS algorithms into a data-parallel computable (DPC) manner, understanding the architecture of the GPU is necessary. The particular nVidia GPU used in this thesis has streaming multiprocessors (SM) that contain 32 CUDA cores. This design allows for each (SM) to have 32 threads for true parallel operations. These 32 threads are called a warp. The number of threads allowed to be queued for concurrent operation is not set by the number of cores, but rather what is called the block size. Blocks are logical organizations of threads, while a grid is a logical organization of blocks. Each block acts independently and is distributed amongst the SM for processing. Each nVidia product has particular specifications on the grid size (number of allowed blocks) and block size (number allowed threads per block). Using the block design allows for a scalable approach as nVidia continues to design and manufacture products with an ever increasing number of SMs. Depending on the kernel, more threads may be created to mask memory operations, but queueing up too many threads may cause deterioration in performance.

Accessing individual threads is made easy by the CUDA framework. Each block and thread has its own index which may be accessed by global constants, threadIdx and blockIdx, within the global memory of the GPU. Accessing any particular thread may be done so in a similar fashion to that of a dynamic array allocated in C++. Every block has the same number of threads, thus accessing the third block (at 10 threads per block), seventh thread is equivalent to an index of position 26 (block index starts at 0).
The shape and size of data structures to be processed should be taken into account to allow for mutually exclusive operations on the data to avoid the requirement of memory sharing management. Uniformly sized data to fit in \( n \) by \( m \) matrices are highly desirable as each thread may work on certain rows or columns of data in a uniform manner without concern of data integrity loss due to memory sharing issues. With a DPC data structure, each thread runs the same kernel (GPU function) and is responsible for a certain amount of data elements in the DPC data structure, which is termed its workload.

In the sections below, descriptions of speed-up methods are presented for the training phases for both NIDS algorithms described in the previous chapter.

3.2 Payload-based Dispersion (PBD) Algorithm

The requirement to reconfigure this algorithm multiple times is one of many reasons to look for solutions for accelerating the training process. This following section describes the approach for implementing the PBD algorithm for SIMD processing.

3.2.1 Bin Counting

As mentioned in the previous chapter, this stage creates histograms of size 256 for each packet payload where each bin is the frequency of each byte present in the payload. The data structures used in this stage are matrices containing \( N \) rows and 256 columns, where the rows represent individual packets and the columns represent each bin in the byte frequency histogram. This histogram matrix is termed
To create the $b$ matrix, the payloads for the given set of network packets are stored in a fixed size matrix, termed $p$ matrix, of $N$ packets by 1500 in order to contain all appropriate contents for any given TCP/IP payload size. An additional auxiliary vector is required holding the payload sizes for each packet termed $ps$ vector. The $p$ matrix has row independent data, thus the rows should be divided amongst the threads. In order to better use GPU architecture, the number of packets were divided by the number of threads $k$ to determine the number of blocks to be used to allow for more efficient usage of the CUDA cores.

In Figure 3.1 a depiction of workloads for each thread are shown. Each block has $k$ number of threads and each thread is assigned its own packet. The threads know the length of each packet via the $ps$ vector and traverses the row in the $p$ matrix appropriately updating the $b$ matrix with the byte frequencies.

### 3.2.2 Bin Statistics

The $b$ matrix from the bin counting stage will remain in GPU memory as the bin statistics stage requires it. The $p$ matrix may be removed to free up space as it is no longer required. The $p$ matrix is also not required for reconfiguration unless additional packets are added or removed from the current set. Revisiting the bin statistics stage, the average and standard deviation for each bin is calculated from the $b$ matrix. A thread results matrix, termed $k$ matrix, is allocated to store the results of each thread. Each thread is assigned multiple rows (packets) to process in a fashion similar to a parallel reduction method. Given a matrix with 10,000 rows and spawning 1000 threads, each thread would have a workload of 10 rows.
Figure 3.1: Bin counting data structures and block assignment. The shaded portions of the \( p \) matrix demonstrates varying lengths of packets.

each. Implementing the parallel reduction method, each column (bin) was assigned a single block of threads to keep bin-related data in the same allotted memory for each streaming processor. Then \( k \) threads were chosen to be spawned per block. To calculate the workload \( w \) for each thread, the number of packets (rows) \( N \) was integer divided by the \( k \) threads. To account for the scenario when \( N \) packets are not evenly divisible by \( k \) threads, \( w \) is increased by 1 meaning there is now a lesser amount of required threads \( r \). To calculate the required number of threads, \( n \) is integer divided by the increased workload \( w \). Finally, another check to see if \( r \) evenly divides into the new \( w \) using the modulo operator and if there is a remainder, \( w \) is stored into \( W \) representing normal workload and \( w \) is then replaced with the remaining workload.
calculated by taking the difference of \( n \) and the product of \( r \) and \( W \). This approach may also be used in the previous stage, bin counting.

\[
w = \frac{n}{k}
\]

IF \((n \mod k == 0)\)

\[
W = w
\]

\[
r = \frac{n}{w}
\]

ELSE

\[
w = w + 1
\]

\[
W = w
\]

\[
r = \frac{n}{w}
\]

\[
w = n - r \times W
\]

Figure 3.2 shows the \( b \) matrix divided into columns by each block. Within each column are divisions of workloads for each thread. The shaded portion of the \( b \) matrix and \( k \) matrix depicts the association of threads and its allocated area for storing its results. The resultant vectors from the bin statistics stage are bin sums, averages, and variances. The key vectors produces are upper and lower threshold vectors which are computed by taking the sum and difference, respectively, of the bin averages and bin variances.

3.2.3 Packet Scoring

In this final stage, the auxiliary \( ps \) vector (packet size vector), \( b \) matrix, and upper and lower threshold vectors are required, all of which are already stored in the
GPU’s memory. This stage counts the number of bins for each packet that exceed the upper and lower thresholds, resulting in packet anomaly scores. The same workload distribution implementation as in the bin counting stage: each thread only processes one packet. Each thread goes through each bin of its assigned packet in the $b$ matrix, compares the value to the over and under value for that particular bin, and stores the anomaly score tally in a resultant vector. Then the overall anomaly score average and variance is calculated using the parallel reduction method across a single vector as depicted in the Figure Figure 3.3. Each thread is assigned a certain section of the resultant anomaly score vector and stores the sum of its section into the first index it is assigned. Finally a single thread is used to compute the average and variance of all the results.

Figure 3.2: This figure shows thread level workloads divided within the $b$ matrix, thread storage in the $k$ matrix, and the resultant vectors.
3.2.4 Evaluation Results

This section discusses the test methods and performance of the parallelized PBD algorithm. The parallel implementation was tested on a lab machine running Ubuntu 14.04 Linux version 12.04.2, 4GHz AMD FX-8350 CPU with 32GB of RAM, and an nVidia GTX 780 with 3GB of on-board memory. The network data sets were
comprised from HTTP web traffic captured from a web server located at the university and were properly sanitized of any malformed or duplicate packets. The training data set used came from the first one hundred thousand packets in chronological order. The remaining sixty thousand packets were used for the testing data set. The attack data set originated from the DARPA attack data sets, which had 2487 packets. The focus of this thesis is not the true and false positive performance of the algorithms, as they were tested for this metric in the original literature. To reiterate, the speedup performance of the PBD algorithm through the use of a SIMD implementation is the focus of this chapter. All results provided are within 1% with a 99% confidence.

### 3.2.4.1 Speedup

There is a balancing act when it comes to choosing the number of threads to use per block. The goal is to choose the amount of threads that will mask the cost of the memory operations required by the code. The term occupancy is used to describe the ratio of active warps and max possible warps in a streaming multi-processor (SM). The block size, or number of threads per block, and the amount of registers required by threads affects the occupancy the most. An occupancy of 1 means the max number of active warps are running on the SM and is a seemingly desirable goal. But during testing, various thread counts were used ranging from 1 thread to 1024 threads. An occupancy of approximately 0.33 to 0.5 yielded the fastest results amongst the thread counts tested. For this particular implementation’s code, 32 threads per block was chosen as it gave the best speedup results, consistently.
The speedup from the serial implementation is show in Table 3.1 for each of the stages along with the expected number of iterations for each phase. The bin counting stage is only required to convert a batch of packets from the \( p \) matrix to the \( b \) matrix, thus bin counting is not required for subsequent retraining or tuning of the algorithm. Bin statistics also only requires one iteration, as the averages and standard deviations only need to be calculated once. Finally, the anomaly scoring stage is the stage that may require upwards to 2,500 iterations to find the appropriate threshold scalars as mentioned in [1]. The speedups presented are comparable to other works related to this approach.

**Table 3.1**: Speed up over CPU implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Speed up</th>
<th>Occurrence per each new packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin counting</td>
<td>8.5</td>
<td>N/A</td>
</tr>
<tr>
<td>Bin stat.</td>
<td>55.6</td>
<td>( 1 \times )</td>
</tr>
<tr>
<td>Packet scoring</td>
<td>28.7</td>
<td>up to ( 2500 \times )</td>
</tr>
</tbody>
</table>

### 3.2.4.2 Memory Footprint

The memory breakdown is shown in table Table 3.2 for all of the required data structures in the GPU implementation. GPU devices have shown a growth in on-board memory allowing for further offloading onto these devices with regards to data storage. By having larger amounts of onboard memory, training packets may reside on the device reducing the costly data transfer times. For the GTX 780, an estimate of over 1 million packets may reside on the device and still be able to run the PBD algorithm.
### Table 3.2: Memory Footprint

<table>
<thead>
<tr>
<th>Name</th>
<th>Footprint (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix ( p )</td>
<td>( n \times 1500 )</td>
</tr>
<tr>
<td>matrix ( b )</td>
<td>( n \times 256 \times 4 )</td>
</tr>
<tr>
<td>BinStats</td>
<td>((n \times 1028) + (k \times 3072) + 9216)</td>
</tr>
<tr>
<td>PacketScoring</td>
<td>((n \times 8) + (k \times 8) + 5120)</td>
</tr>
<tr>
<td>Total</td>
<td>((n \times 2536) + (k \times 3080) + 9216)</td>
</tr>
</tbody>
</table>
CHAPTER 4

ALTERNATIVE APPROACHES AND SOLUTIONS

Mask selection is the key to utilizing LiSPI effectively for a given network. Because of the variability across different traffic patterns, it is not feasible to pre-generate mask pairs that work across different systems. Therefore, each deployment instance requires evaluation of different potential mask pairs. Evaluating each mask pair requires the training and testing for effectiveness on data sets consisting of tens to hundreds of thousands of packets. For example, on a fast, modern computer (tested on a 4GHz AMD FX-8350 CPU), searching through the entire problem space required roughly 4 weeks for a training set consisting of only 20,000 packets. If online updates are desirable, this wait time cost is too high. During the wait time, the traffic characteristics of the network may actually change before finding the best performing mask pair. Therefore, instead of a brute-force approach, some form of analysis of the traffic is required to try and hypothesize which sets of bit-patterns may work best on the given training set. In [29], various sets of features were tested to cover byte ranges that included all possible values [0-255] and splitting the byte value range into 8 ranges [0,31],[23,63],...,[224,255]. A number of such heuristics approaches were attempted, all of which required hand-tuning on a trial and error basis.
Fortunately, the LiSPI algorithm is lightweight enough to be run in parallel using GPUs and also has an attribute that makes the algorithm well suited to use a genetic algorithm to search the solution space for strongly performing mask pairs. In the following sections are descriptions of how a GPU is used to brute force search for the optimal mask pair solution and also a genetic algorithm design ran on a CPU to search for strong mask pairs within the solution space.

4.1 GPU Solution

The LiSPI algorithm is able to be sped up using the same approach as the PBD algorithm, but another opportunity is available for further parallelization of the algorithm in order to assure the best mask pairs are found given the bitmask and bitmap sizes. A parallel brute force approach to finding the optimal mask pairs may be implemented allowing for massive amounts of possible mask pairs to be computed hundreds of times faster than a sequential brute force calculation.

4.1.0.3 Mask Pair Reduction

Recapping the LiSPI background from Chapter 2, an 8-bit length mask was chosen for this thesis, yielding over 43 million combinations of 8-bit mask pairs ($3^{16} = 43,046,721$). Looking further into the mask pairs, some redundancy was found as about half of the mask pairs have a mirrored matching mask pair. For example, the mask pair (XXXX0000, 1111XXXX) mirrors (1111XXXX, XXXX0000). Removal of one of these pairs are justified since the resulting bitmap of these two mask pairs are simply transposed versions of one another with the same attack detection and false
positive totals. This reductions drops the number of mask pairs down to \((3^{16} - 3^{8} \times (3^{8} - 1)/2)) 21,526,641 combinations. The reduced set of all unique mask pairs are saved in what is termed the combination matrix which holds the masks’ byte values and their respective attack detection rates and false positive rates, which are later processed.

4.1.0.4 Memoization

A data structure termed feature matrix is used to store data with regards to each individual mask. This matrix holds precomputed quantized bin sums for each given mask. Recalling the bin counting and quantization stage from the LiSPI algorithm, each traffic data set required each packet’s histogram to have its bins summed together based on the given bitmask. By precomputing all \(3^8\) possible bitmasks and storing the quantized result in the feature matrix, the feature matrix becomes a lookup table for the bitmap generation step. The final dimensions of the feature matrix is \(N\) packets by \(3^8\), where \(N\) is the number of packets for each respective traffic data set used in the training phase.

The creation of the feature matrix may be accomplished in a parallel fashion using the GPU as it is in a data parallel computational organization. Each column in the feature matrix represents a single mask, while each row represents a single packet. To parallelize the process of filling out the matrix, each thread is assigned a single column (mask) and stores the resultant bitmap index from the bin counting and quantization stage into the appropriate feature matrix index for every packet in
the given data set. A feature matrix for each the training, testing, and attack data set are created and stored for the bitmap creation.

There were some limitations with regards to the size of the feature matrices exceeding the available memory on the graphics card used in this thesis. For clarity, the graphics card’s on-board memory will be denoted as device memory and the CPU’s main memory will be referred to as host memory. This limitation required transferring the feature tables from the device to the host’s memory to free up space for further calculations. The device memory was able to hold approximately 160 thousand packet histograms and still be able to complete the algorithm.

4.1.0.5 Bitmap Generation

After completion of the feature table generation, bitmaps are generated for each unique mask pair which will be used in measuring false positive and true positive performance. The training set’s feature table is used to generate the 16x16 bitmaps for all ~21 million mask pair combinations, subject to the memory available on the GPU. Thus, in our case, the bitmaps were generated one million at a time, which are then subsequently dumped into a bitmap matrix in the host memory upon completion.

For bitmap generation, each thread is designated a mask pair, predetermined in the combination matrix previously mentioned, which designates the appropriate columns in the feature table. Going through each packet in the feature table in the designated columns, the threads use the feature pairs in each row to map 1’s to the appropriate bitmap in the bitmap matrix. Upon completion of all one million mask combinations, the bitmaps are transferred to the host’s main memory and the
device zeroes out its data structure in its memory for the next million bitmaps to be produced. This process continues until all bitmaps are generated for all unique mask pair combinations.

In Figure 4.1, the responsibilities for each thread is shown. Two columns are assigned to each thread, $P$ and $Q$. Each thread uses the pair of values on each row of its assigned columns to map to a 16x16 bitmap. Notice $Thread_0$ has identical columns for its mask pairs, while $Thread_1$ has its Q mask on the next column to the right. This pattern is repeated for all combinations of columns in the feature matrix for one million queued threads. The bitmaps are stored in a larger matrix and moved to host memory upon completion its completion for every one million threads.

Once all bitmaps are generated, the feature tables of the testing data set and attack data set are loaded into the device memory. The work load is entirely identical to the bitmap generation except for mapping the feature pairs to the generated bitmaps. Each thread is designated a unique mask pair and iterates down the columns of the feature table to enable a look-up on the bitmap for the mask pair in question. As mentioned in the previous section, a 1 value indicates normal traffic and a 0 indicates a false positive for the testing data set, while a 1 indicates a false negative and a 0 indicates a true positive for the attack data set.

False positives and true positives for each mask pair are tallied at the end and stored within the combination matrix. Similar to the bitmap generation process, one million results (computing the true and false positives) are queued at a time to stay within the GPU’s memory limits. This process continues until all true positive and false positive rates are calculated for every unique mask pair combination.
Figure 4.1: Each column in the feature matrix represents a bitmask and each thread is responsible for one pair of bitmasks to produce a corresponding bitmap during the training phase. Each index from each column is mapped using the quantization method \( f(P_i), f(Q_i) \) described in Chapter 2.

4.2 Genetic Algorithm

Genetic algorithms may also be used to optimize current IDS approaches or be the heart of the algorithm identifying intrusions while being competitive with other efficient approaches such as parallel approaches [30, 31]. In this section a genetic algorithm is described to search the solution space of mask pairs as another approach to selecting highly performing LiSPI mask pairs.
Genetic algorithms are used to search a large solution space of a given problem based on some set of possible input parameters. The set of input parameters are typically encoded into a bit string using various methods for data representation, which is often compared to an organism's chromosomes. An organism, in this case, is one possible solution and its genes are the bits within the encoded bit string. The term organism and solution are used interchangeably through the rest of this paper. By initializing a predetermined population size with randomly generated bit strings, the evolutionary process of genetics is simulated. The fitter organisms have a higher probability of surviving and passing on its genes. Each solution within a population must have its fitness measured using a fitness score function. With regards to this paper's intrusion detection approach, a particular solution's false positive and true positive scores are used to measure the fitness with a heavier weight given to false positives rates. The fitness function is based on the LiSPI algorithm, where the true positive and false positives values are used to calculate the fitness score of an organism.

Survival of organisms or their traits between each generation are decided by using the roulette wheel algorithm explained later in this section. During the selection process, two solutions are chosen at a time. First, a crossover may be performed, which allows for genes, or parameters, to be swapped between the solution pair, producing two new solutions, for the next generation. Finally, each gene, or bit in the bit string, may be mutated to mitigate being caught in local maxima of the solution space. The selection, crossover, and mutation process is repeated until the next generation's population is full. The population size remains constant. Upon filling
the next generation, the entire process is repeated from the fitness score calculation for each solution in the new generation. A stop condition is required to end the genetic algorithm.

The goal of the genetic algorithm, as used in this context, is to automatically find viable solutions quickly. The LiSPI bit mask problem space is relatively small compared to some of the canonical optimization problems with extremely large solution spaces, but still large enough to challenge other simple heuristics or human-guided selection.

4.2.1 Initialization

The bit masks from the LiSPI algorithm lend itself well to genetic algorithms as they represent the different byte value combinations. A mask pair is treated as an organism. A random number generator is used to select masks to create the initial population. Each mask is paired with another making sure to have only unique pairs of masks. For instance, mask pairs (m1, m2) and (m2, m1) are the same for LiSPI. The population sizes of 25 and 50 are reported (larger sets were also tested, but did not prove to be beneficial).

4.2.2 Determining Fitness

Upon filling the initial generation’s population, the fitness score is determined for each individual organism (mask pair). This score is based on the true positive rate (TPR, attack packets detected) and the false positive rate (FPR, normal traffic flagged as anomalies). The fitness function used is as follows: \[ \text{Fitness} = \alpha(TPR) + \]
(1 − α)(1−FPR) where α is a scalar weight ranging from [0,1]. In the experiments below, alpha is set to 0.15, thus weighing the false positive rates more heavily.

4.2.3 Selecting the Next Generation

After all organisms have their fitness score calculated, a probability is calculated for each organism. This value is used to determine if an organism should be crossed with another organism and pass its traits onto the next generation. This probability for an individual organism is calculated by dividing its fitness score by the sum of the entire current generation’s fitness scores. Then the organisms are sorted by each organism’s probability in ascending order.
Organisms are chosen two at a time from the current generation using the roulette wheel algorithm. Any organism may be chosen multiple times to possibly cross with another during the selection phase, but the resulting offspring must be unique to the next generation’s population. In our implementation, the top 20% of the organisms move onto the next generation by form of elitism in order to have a higher probability of the elites passing on their strong traits; the top 20% of each generation are termed *elites*.

Crossover is decided after the selection of two organisms have been made via the roulette wheel algorithm. A random number generator is used to determine if there will be a crossover between the two organisms with a probability to cross set at 70%. This process allows the chance for newer traits to be produced in the next generation, while still maintaining the current generation’s traits.

If the decision is made to cross, a randomly chosen position in the bit mask pair is set as a pivot to indicate where to swap the subsets of mask pairs with the other organism. For example, see Figure Figure 4.2. The crossover pivot point was randomly chosen at the third element in the organism and the subsequent elements are swapped. The result is a pair of new offspring organisms with traits inherited from the parents.

Whether or not the selected organisms crossed, a 5% probability to mutate a single gene in the chromosome is given individually for each gene. This means multiple mutations may occur in the same organism. In the example shown in Figure Figure 4.2, the last bit of the first offspring was randomly chosen to switch to a
0 or 1. The 0 and 1 have equal probability and 0 was randomly chosen. The same occurs with the fifth element of the second offspring.

After crossing and mutation has been completed, another check is made to ensure only unique organisms (mask pairs) make it to the next generation. Once the next generation is filled to the specified population size, the fitness score is calculated for the entire new generation and the entire process is repeated.

### 4.2.4 Stopping Conditions

Stop conditions are required to end the genetic algorithm. Stop conditions may vary, and multiple stop conditions may be used together. In this paper, the stopping condition is determined by false positive rates below 0.5% and true positive rates above 98%. Using the fitness function described above, this provides a fitness score threshold of 0.99275.

### 4.3 Evaluation Results

For all implementations, serial, parallel, and genetic algorithm, two sets of web traffic were used for training and testing. The first set was captured at one of the web servers at the university and is referred to as WEB traffic, and the second set is from the DARPA week 3 web traffic data set. The DARPA set has been criticized as being outdated or artificial, but still serve as an interesting reference point. Note that our results are not dependent on the use of DARPA data. Both traffic sets were sanitized for any malformed packets prior to testing. Each traffic set was then divided into training and testing sets. Each training subset starts at the beginning of
the captured packets and the testing subsets are from the remaining portions of each traffic set. Testing sets are used for measuring false positive rates. The DARPA week three set is split in a similar fashion; three training sets are made with ranges 20, 60, and 100 thousand packets. The final 60 thousand packets are used from the end of the week 3 DARAP data set. For the WEB traffic set, there are fewer total packets. The first 25,222 packets were used for training and the final 13,306 packets were used for testing false positives. To determine true positives, or attack detection rates, an attack data set was constructed using the attack data from DARPA in addition to newer attacks such as Code Red II and Nimda. Except for the single individual run figures for the genetic algorithms, all reported values are within 1% of the numbers shown with 99% confidence in this section.

Figures 4.3 and Figure 4.4 show the top elite’s fitness score over the number of generations along with the mean fitness score for each generation for WEB (training set size 25,222) and DARPA (training set size 20K). Note that the y-axes for both figures do not begin at 0 in order to give a better view of what is happening with the data. For these particular figures, there are no data points below each bottom-limit of the y-axis. The top fitness score for both figures show areas of plateauing and large jumps, indicating crossovers and mutations are allowing for exploration of the solution space in a relatively short number of generations. Figure Figure 4.4 shows that a fitness score that meets the stopping condition is found at the twelfth generation with 0.015% false positive and 99.88% true positive rates, and a fitness score of 0.9988. In Figure Figure 4.4, a fitness score that meets the stopping condition
Figure 4.3: Fitness Score vs. Generations for Web Traffic 1 data set (25k training set, population size 25, 100,000 generations shown).

can be seen at generation 139 with 0.08% false positive and 98.15% true positive rates, and a fitness score of 0.9966.

Table 4.1: Run times in seconds.

<table>
<thead>
<tr>
<th>Data Set Size</th>
<th>LiSPI</th>
<th>GPU-LiSPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>20k</td>
<td>2,223,766</td>
<td>760</td>
</tr>
<tr>
<td>60k</td>
<td>3,310,474</td>
<td>1,112</td>
</tr>
<tr>
<td>100k</td>
<td>4,397,053</td>
<td>1,502</td>
</tr>
</tbody>
</table>

Table 4.1 shows the run times in seconds for brute force calculation of all unique mask pair combinations of the LiSPI algorithm using a CPU (AMD FX-8350 and 32GB RAM) and a GPU (Nvidia GTX 780 with 3GB GPU memory).
Figure 4.4: Fitness Score vs. Generations for DARPA week 3 data set (20k training set, population size 25, 100,000 generations shown).

The first column of the table represents the training set sizes. The testing set and attack set sizes were fixed at 60K and 2,485 respectively. The second column represents the run times for a CPU implementation of the LiSPI algorithm using the C programming language. The third column represents run times for the GPU-based parallel implementation described in the previous section. Speed-up values on average were 2943x. From the GPU runs, the optimal solutions for WEB is 98.31% TPR and 0.045% FPR and 99.88% TPR and 0.015% FPR for DARPA. It should be noted that in previous tests, LiSPI was shown to perform well in terms of having a small performance penalty. Tested on a desktop-grade processor, the detection stage yielded average latency value of 0.0045ms. As a reference, the latency for the PAYL
algorithm [2], which is one of the simplest, and therefore fastest payload based algorithms, is 0.0074ms (with size normalization). The corresponding true and false positive rates for PAYL are 98.99% and 0.19% respectively (DARPA). With the false positive rate, LiSPI outperforms it by an order of magnitude while being faster. Other approaches that provide discriminating capabilities similar to LiSPI were shown to have significant degradation in latency figures (up to a factor of 10). As previously mentioned, LiSPI’s drawback was with finding a good mask pair, which made it difficult for network operators to deploy a security component with LiSPI. Our results show that the training can be automated, effectively mitigating the problem.

Table 4.2: The GA run times are shown for starting population sizes of 25 and 50, at which an elite solution could be found.

<table>
<thead>
<tr>
<th>Population Size</th>
<th>25</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Set</td>
<td>Best (sec)</td>
<td>Worst (sec)</td>
</tr>
<tr>
<td>WEB</td>
<td>52</td>
<td>1238</td>
</tr>
<tr>
<td>DARPA 20k</td>
<td>3</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 4.2 shows the estimated run time to finding the first elite organism out of ten (random) sample runs with the genetic algorithm. Both the best and worst runs are shown, respectively, in each row. The genetic algorithm implementation shows run times that are promising in both best and worst of the sample runs. Population size 25 for the WEB traffic, the genetic algorithm hit the stop-condition at generations 20 and 479 for best and worst runs; for the DARPA 20k traffic set, the genetic algorithm hit its stop-condition at generations 1 and 33 for its best and worst recorded runs. Population size 50 for the WEB traffic, the genetic algorithm hit the stop-condition at generations 10 and 341 for best and worst runs; for the DARPA 20k
traffic set, the genetic algorithm hit its stop-condition at generations 1 and 17 for its best and worst recorded runs.
CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Keeping intrusion detection systems up-to-date with constantly changing technology is key to successful security systems. There are many network intrusion detection systems and algorithms available, but no single solution is the end all, be all approach for every network; multiple algorithms should be investigated to improve overall security. Anomaly-based NIDS require historical network traffic to build a template of what is deemed normal and safe to compare new incoming traffic against. Building this template may consume more computational resources than is available for time periods unacceptable to keep the network secure. This thesis has shown that off the shelf GPUs are able to alleviate this heavy workload and provide an up-to-date template of normal traffic while speeding up the processing time by up to 400 times. Further optimization techniques such as genetic algorithms may also be used to help reduce the tuning of the algorithms to build a better, more accurate, template, which in turn yields better true positive and false positive performance.

Future work entails implementing a version of the genetic algorithm that uses the GPU to generate fitness scores for multiple sets of populations in parallel for
the LiSPI algorithm. We predict a speedup that is larger than the brute force GPU implementation and require less memory and memoization. Further tuning of the fitness score, population sizes, and a live sliding window to constantly keep the LiSPI NIDS up-to-date on a live system is also an interesting future research problem.
__global__ void generateBMaps (unsigned int cBos, unsigned int pktCount, unsigned int *scoreMat, char *bMap, unsigned int *wCombo) {
  unsigned int k, m;
  unsigned int idx1;
  char y = 16;
  idx1 = (blockDim.x * blockIdx.x + threadIdx.x);
  if (blockIdx.x == (gridDim.x - 1)) {
    // I'm the last block
    m = cBos % blockDim.x;
    if (m != 0) {
      if (threadIdx.x < m) {
        // Training
        for (k = 0; k < pktCount; k++) {
          
        }
      }
    }
  }
}


bMap[ idx1*256 + (scoreMat[k*6561 + wCombo]
    [idx1*5 + 1]|*y + scoreMat[k*6561 +
    wCombo[idx1*5]]) ] = 1;
}

// Training
for (k = 0; k < pktCount; k++){
    bMap[ idx1*256 + (scoreMat[k*6561 + wCombo[
        idx1*5 + 1]|*y + scoreMat[k*6561 + wCombo[
        idx1*5]]) ] = 1;
}

else{
    // Training
    for (k = 0; k < pktCount; k++){
        bMap[ idx1*256 + (scoreMat[k*6561 + wCombo[
            idx1*5 + 1]|*y + scoreMat[k*6561 + wCombo[
            idx1*5]]) ] = 1;
    }
}
else{
    // Training
    for (k = 0; k < pktCount; k++){
        bMap[ idx1*256 + (scoreMat[k*6561 + wCombo[idx1*5
            + 1]|*y + scoreMat[k*6561 + wCombo[idx1*5]]) ]
            = 1;
    }
}
```c
__syncthreads();

__global__ void trainSGPU (unsigned int cBos, unsigned int pktCount_FP, unsigned int pktCount_ATTACK, unsigned int *scoreMat_FP, unsigned int *scoreMat_ATTACK, unsigned int *wCombo, char *bMap)
{
  // char FP indicates whether or not it is true positive
  // or false positive; 1 = FP and 2 = TP
  // wCombo is 4*21 million... [mask1][mask2][FP][TP]
  unsigned int k, m;
  unsigned int idx1;
  char y;
  char *Attackarray;

  y = 16;
  idx1 = (blockDim.x * blockIdx.x + threadIdx.x);

  if (blockIdx.x == (gridDim.x - 1))
  {
    // I'm the last block
    m = cBos % blockDim.x;
    if (m != 0) {
      if (threadIdx.x < m) {
        Attackarray = (char*)malloc(20);
      }
  ```
memset(Attackarray, 0, 20);

// False positives
for (k = 0; k < pktCount_FP; k++){
    if (bMap[idx1*256 + (scoreMat_FP[k*6561 + wCombo[idx1*5 + 1]]*y + scoreMat_FP[k*6561 + wCombo[idx1*5]]]) == 0){
        wCombo[idx1*5 + 2] ++;
    }
}

// True Positives
for (k = 0; k < pktCount_ATTACK; k++)
{
    if (bMap[idx1*256 + (scoreMat_ATTACK[k*6561 + wCombo[idx1*5 + 1]]*y + scoreMat_ATTACK[k*6561 + wCombo[idx1*5]])] == 0){
        if (k == 0)
            Attackarray[0] = 1;
        else if (k >= 1 && k < 14)
            Attackarray[1] = 1;
        // else if (k == 14)
        // Attackarray[2] = 1;
}
else if (k >= 14 && k < 510)
Attackarray[2] = 1;
else if (k >= 510 && k < 512)
Attackarray[3] = 1;
else if (k == 512)
Attackarray[4] = 1;
else if (k >= 513 && k < 1167)
Attackarray[5] = 1;
else if (k >= 1167 && k < 1628)
Attackarray[6] = 1;
else if (k >= 1628 && k < 2120)
Attackarray[7] = 1;
else if (k >= 2120 && k < 2130)
Attackarray[8] = 1;
else if (k >= 2130 && k < 2132)
Attackarray[9] = 1;

// else if (k == 2132)
else if (k == 2132)
Attackarray[10] = 1;
else if (k >= 2133 && k < 2143)
else if (k >= 2143 && k < 2189)
Attackarray[12] = 1;
else if (k == 2189)
Attackarray[13] = 1;
else if (k >= 2190 && k < 2430)
Attackarray[14] = 1;
// else if (k == 2431)
// Attackarray[17] = 1;
else if (k >= 2430 && k < 2434)
Attackarray[15] = 1;
else if (k >= 2434 && k < 2437) //
    code red ii
Attackarray[16] = 1;
else if (k >= 2437 && k < 2467)
Attackarray[17] = 1;
else if (k >= 2467 && k < 2469)
Attackarray[18] = 1;
else if (k >= 2469 && k < 2485)
Attackarray[19] = 1;
wCombo[idx1*5 + 3] ++;
}
}
for (k = 0; k < 20; k++){

    // stores how many attacks were found
    wCombo[idx1*5 + 4] += Attackarray[k];
}
free(Attackarray);

}

else
{

    Attackarray = (char*)malloc(20);;
    memset(Attackarray,0,20);

    // False positives
    for (k = 0; k < pktCount_FP; k++){
        if (bMap[idx1*256 + (scoreMat_FP[k*6561 + \\
            wCombo[idx1*5 + 1]*y + scoreMat_FP[k*6561 \\
            + wCombo[idx1*5]])] == 0)
        {
            wCombo[idx1*5 + 2] ++;
        }
    }

    // True Positives
    for (k = 0; k < pktCount_ATTACK; k++)
if (bMap[idx1 * 256 + (scoreMat_ATTACK[k * 6561 \\
+ wCombo[idx1 * 5 + 1]] * y + scoreMat_ATTACK[k * 6561 + wCombo[idx1 * 5]])] == 0)
{

  // ATTACK IS DETECTED

  if (k == 0)
    Attackarray[0] = 1;
  else if (k >= 1 && k < 14)
    Attackarray[1] = 1;

  // else if (k == 14)
  //  Attackarray[2] = 1;
  else if (k >= 14 && k < 510)
    Attackarray[2] = 1;
  else if (k >= 510 && k < 512)
    Attackarray[3] = 1;
  else if (k == 512)
    Attackarray[4] = 1;
  else if (k >= 513 && k < 1167)
    Attackarray[5] = 1;
  else if (k >= 1167 && k < 1628)
    Attackarray[6] = 1;
}
else if (k >= 1628 && k < 2120)
Attackarray[7] = 1;
else if (k >= 2120 && k < 2130)
Attackarray[8] = 1;
else if (k >= 2130 && k < 2132)
Attackarray[9] = 1;
// else if (k == 2132)
else if (k == 2132)
Attackarray[10] = 1;
else if (k >= 2133 && k < 2143)
else if (k >= 2143 && k < 2189)
Attackarray[12] = 1;
else if (k == 2189)
Attackarray[13] = 1;
else if (k >= 2190 && k < 2430)
Attackarray[14] = 1;
// else if (k == 2431)
// Attackarray[17] = 1;
else if (k >= 2430 && k < 2434)
Attackarray[15] = 1;
else if (k >= 2434 && k < 2437) // code

    red ii

Attackarray[16] = 1;

else if (k >= 2437 && k < 2467)

Attackarray[17] = 1;

else if (k >= 2467 && k < 2469)

Attackarray[18] = 1;

else if (k >= 2469 && k < 2485)

Attackarray[19] = 1;

wCombo[idx1*5 + 3] ++;

}

for(k = 0; k < 20; k++){

    // stores how many attacks were found

    wCombo[idx1*5 + 4] += Attackarray[k];

}

free(Attackarray);

}

}

else

{

Attackarray = (char*)malloc(20);

}
memset(Attackarray, 0, 20);

// False positives
for (k = 0; k < pktCount_FP; k++)
{
    if (bMap[idx1*256 + (scoreMat_FP[k*6561 + wCombo[idx1*5 + 1]]*y + scoreMat_FP[k*6561 + wCombo[idx1*5]])] == 0)
    {
        wCombo[idx1*5 + 2] ++;
    }
}

// True Positives
for (k = 0; k < pktCount_ATTACK; k++)
{
    if (bMap[idx1*256 + (scoreMat_ATTACK[k*6561 + wCombo[idx1*5 + 1]]*y + scoreMat_ATTACK[k*6561 + wCombo[idx1*5]])] == 0)
    {
        if (k == 0)
            Attackarray[0] = 1;
        else if (k >= 1 && k < 14)
            Attackarray[1] = 1;
        // else if (k == 14)
// Attackarray[2] = 1;

else if (k >= 14 && k < 510)
Attackarray[2] = 1;

else if (k >= 510 && k < 512)
Attackarray[3] = 1;

else if (k == 512)
Attackarray[4] = 1;

else if (k >= 513 && k < 1167)
Attackarray[5] = 1;

else if (k >= 1167 && k < 1628)
Attackarray[6] = 1;

else if (k >= 1628 && k < 2120)
Attackarray[7] = 1;

else if (k >= 2120 && k < 2130)
Attackarray[8] = 1;

else if (k >= 2130 && k < 2132)
Attackarray[9] = 1;

// else if (k == 2132)


else if (k == 2132)
Attackarray[10] = 1;

else if (k >= 2133 && k < 2143)
else if (k >= 2143 && k < 2189)
Attackarray[12] = 1;
else if (k == 2189)
Attackarray[13] = 1;
else if (k >= 2190 && k < 2430)
Attackarray[14] = 1;
// else if (k == 2431)
// Attackarray[17] = 1;
else if (k >= 2430 && k < 2434)
Attackarray[15] = 1;
else if (k >= 2434 && k < 2437) // code red
Attackarray[16] = 1;
else if (k >= 2437 && k < 2467)
Attackarray[17] = 1;
else if (k >= 2467 && k < 2469)
Attackarray[18] = 1;
else if (k >= 2469 && k < 2485)
Attackarray[19] = 1;

wCombo[idx1*5 + 3] ++;
}
for (k = 0; k < 20; k++] {
    // stores how many attacks were found
    wCombo[idx1*5 + 4] += Attackarray[k];
}
free(Attackarray);
__syncthreads();
__global__ void GPUgenerateScores (unsigned int cBos,
unsigned int pktCount, unsigned char *gpu_masks, unsigned
int *dev_matB, unsigned int *score, unsigned int *
dev_maxScores){
    // cBos = the size of Workload... since the GPU does not
    // have unlimited Memory, we had to split up the work
    // prior to calling foo1... for now, we'll try loading in
    // the whole thing.
    unsigned int k,m;  // k = counter; m = size of last bit of
    // the work load, which is less than the # of threads
    // available
    unsigned int idx1; // This is how you will reference
    threads....
// blockDim, blockIdx, threadIdx are all built-in variables in CUDA library... this allows you to find reference index for each thread.

idx1 = (blockDim.x * blockIdx.x + threadIdx.x);

if (blockIdx.x == (gridDim.x - 1))
{
    // I’m the last block
    m = cBos % blockDim.x;
    if (m != 0)
    {
        // Threads didn’t evenly divide into work load
        if (threadIdx.x < m)
        {
            // Threads that have work to do.. the rest
            // will synch at the bottom of this function
            // Generate Prescores

            int i, j;

            unsigned int pCount;

            for (i = 0; i < pktCount; i++){
                pCount = 0;

                for (j = 0; j < 256; j++){
```c
if ((j&gpu_masks[idx1*2+1]) ==
gpu_masks[idx1*2]){
    pCount+= dev_matB[i*256+j];
}
}

if(pCount > dev_maxScores[idx1]){
    dev_maxScores[idx1] = pCount;
}
}

// Generates Scores
for (i = 0; i < pktCount; i++){
    pCount = 0;
    for (j = 0; j < 256; j++){
        if ((j&gpu_masks[idx1*2+1]) ==
            gpu_masks[idx1*2]){
            pCount+= dev_matB[i*256+j];
        }
    }
    if(dev_maxScores[idx1] != 0){
        score[i*(6561) + idx1] = pCount*16/(dev_maxScores[idx1]*1.15);
        if(score[i*(6561) + idx1] > 15)
```
printf("%u-%u-%un", score[i*(6561) + idx1], pCount, dev_maxScores[idx1]);
}
else{

    score[i*(6561) + idx1] = 0;
}
}

}

else
{

    // Threads matched perfectly with the work load size

    // Generate Prescores

    int i, j;

    unsigned int pCount;

    for (i = 0; i < pktCount; i++){
        pCount = 0;
        for (j = 0; j < 256; j++){
            if (((j&gpu_masks[idx1*2+1])==gpu_masks[idx1*2]){

        }
pCount+= dev_matB[i*256+j];
}
}
if(pCount > dev_maxScores[idx1]){  
dev_maxScores[idx1] = pCount;
}

// Generates Scores
for (i = 0; i < pktCount; i++){
    pCount = 0;
    for (j = 0; j < 256; j++){
        if (((j&gpu_masks[idx1*2+1])==gpu_masks[idx1*2]){
            pCount+= dev_matB[i*256+j];
        }
    }
    if(dev_maxScores[idx1] != 0){
        score[i*(6561) + idx1] = pCount*16/(dev_maxScores[idx1]*1.15);
        if(score[i*(6561) + idx1]>15)
            printf("%u-%u-%u", score[i*(6561) + idx1], pCount, dev_maxScores[idx1]);
    }
else{
    score[i*(6561) + idx1] = 0;
}
}
}
}
}
}

else
{

    // Every other block of threads, besides last block
    // Generate Prescores

    int i, j;
    unsigned int pCount;

    for (i = 0; i < pktCount; i++){
        pCount = 0;
        for (j = 0; j < 256; j++){
            if (((j&gpu_masks[idx1*2+1]) == gpu_masks[idx1*2])
                 *2)){
                pCount+= dev_matB[i*256+j];
            }
        }
        if(pCount > dev_maxScores[idx1]){
        
    
}
dev_maxScores[idx1] = pCount;

}
}

// Generates Scores

for (i = 0; i < pktCount; i++){
    pCount = 0;
    for (j = 0; j < 256; j++){
        if ((j&gpu_masks[idx1*2+1])==gpu_masks[idx1*2]){
            pCount+= dev_matB[i*256+j];
        }
    }
}

if(dev_maxScores[idx1] != 0){
    score[i*(6561) + idx1] = pCount*16/(dev_maxScores[idx1]*1.15);
    if(score[i*(6561) + idx1]>15)
        printf("%u-%u-%un",score[i*(6561) + idx1], pCount,dev_maxScores[idx1]);
}
else{
    score[i*(6561) + idx1] =0;
}
}]

}]

// Make all threads wait for each other to finish before exiting function... exiting too early can screw things up

__syncthreads();

__global__ void GPUgeneratePreScores(unsigned int cBos,
                                        unsigned int pktCount, unsigned char *gpu_masks, unsigned int *dev_matB, unsigned int *score, unsigned int *dev_maxScores){

    // cBos = the size of Workload... since the GPU does not have unlimited Memory, we had to split up the work prior to calling foo1... for now, we'll try loading in the whole thing.

    unsigned int k,m; // k = counter; m = size of last bit of the work load, which is less than the # of threads available

    unsigned int idx1; // This is how you will reference threads....
// blockDim, blockIdx, threadIdx are all built-in
variables in CUDA library... this allows you to find
reference index for each thread.

idx1 = (blockDim.x * blockIdx.x + threadIdx.x);

if (blockIdx.x == (gridDim.x - 1))
{
    // I'm the last block
    m = cBos % blockDim.x;
    if (m != 0)
    {
        // Threads didn't evenly divide into work load
        if (threadIdx.x < m)
        {
            // Threads that have work to do.. the rest
            // will synch at the bottom of this function

            // Generate Prescores

            int i, j;

            unsigned int pCount;

            for (i = 0; i < pktCount; i++){
                pCount = 0;
                for (j = 0; j < 256; j++){
if ((j&gpu_masks[idx1*2+1]) ==
        gpus_masks[idx1*2]){
    pCount+= dev_matB[i*256+j];
}
}
if(pCount > dev_maxScores[idx1]){
    dev_maxScores[idx1] = pCount;
}
}
else{
    // Threads matched perfectly with the work load size
    // Generate Prescores
    int i, j;
    unsigned int pCount;
    for (i = 0; i < pktCount; i++){
        pCount = 0;
        for (j = 0; j < 256; j++){
if ((j&gpu_masks[idx1*2+1])==gpu_masks[idx1*2]){
    pCount+= dev_matB[i*256+j];
}

if(pCount > dev_maxScores[idx1]){
    dev_maxScores[idx1] = pCount;
}

else {

    // Every other block of threads, besides last block
    // Generate Prescores
    int i, j;
    unsigned int pCount;
    for (i = 0; i < pktCount; i++){
        pCount = 0;
        for (j = 0; j < 256; j++){
            if (((j&gpu_masks[idx1*2+1])==gpu_masks[idx1*2])){

            }
pCount+= dev_matB[i * 256 + j];

}

}

if (pCount > dev_maxScores[idx1]) {
    dev_maxScores[idx1] = pCount;
}

}

}

// Make all threads wait for each other to finish before exiting function... exiting too early can screw things up
__syncthreads();

__global__ void GPUgenerateCurScores (unsigned int cBos, unsigned int pktCount, unsigned char *gpu_masks, unsigned int *dev_matB, unsigned int *score, unsigned int *dev_maxScores) {
    // cBos = the size of Workload... since the GPU does not have unlimited Memory, we had to split up the work prior to calling foo1... for now, we'll try loading in the whole thing.
unsigned int k, m; // k = counter; m = size of last bit of the work load, which is less than the # of threads available

unsigned int idx1; // This is how you will reference threads....

// blockDim, blockIdx, threadIdx are all built-in variables in CUDA library... this allows you to find reference index for each thread.

idx1 = (blockDim.x * blockIdx.x + threadIdx.x);

if (blockIdx.x == (gridDim.x - 1))
{
    // I'm the last block
    m = cBos % blockDim.x;
    if (m != 0)
    {
        // Threads didn't evenly divide into work load
        if (threadIdx.x < m)
        {
            // Threads that have work to do... the rest will sync at the bottom of this function

            // Generates Scores

            int i, j;
        }
unsigned int pCount;

for (i = 0; i < pktCount; i++){
    pCount = 0;
    for (j = 0; j < 256; j++){
        if ((j & gpu_masks[idx1*2+1]) == gpu_masks[idx1*2]){
            pCount += dev_matB[i*256+j];
        }
    }
    if(dev_maxScores[idx1] != 0){
        score[i*(6561) + idx1] = pCount*16/(dev_maxScores[idx1]*1.15);
        if(score[i*(6561) + idx1] > 15)
            printf("%u-%u-%u\n", score[i*(6561) + idx1], pCount, dev_maxScores[idx1]);
    }
    else{
        score[i*(6561) + idx1] = 0;
    }
}
}
else
{

    // Threads matched perfectly with the work load size

    // Generates Scores

    int i , j;

    unsigned int pCount;

    for ( i = 0; i < pktCount; i++){
        pCount = 0;
        for ( j = 0; j < 256; j++){
            if ((j&gpu_masks[idx1*2+1])==gpu_masks[idx1*2]){
                pCount+= dev_matB[i*256+j];
            }
        }
        if(dev_maxScores[idx1] != 0){
            score[i*(6561) + idx1] = pCount*16/(dev_maxScores[idx1]*1.15);
            if(score[i*(6561) + idx1]>15)
                printf("%u/uni2423-%u/uni2423-%u",
                    score[i*(6561) + idx1], pCount, dev_maxScores[idx1]);
        }
    }
}
else{
    score[i*(6561) + idx1] = 0;
}
}
}

else{

    // Every other block of threads, besides last block
    // Generates Scores

    int i, j;
    unsigned int pCount;
    for (i = 0; i < pktCount; i++){
        pCount = 0;
        for (j = 0; j < 256; j++){
            if (((j&gpu_masks[idx1*2+1])==gpu_masks[idx1*2+2]){
                pCount+= dev_matB[i*256+j];
            }
        }
    }
    if(dev_maxScores[idx1] != 0){

score[i*(6561) + idx1] = pCount*16/(dev_maxScores[idx1]*1.15);
if(score[i*(6561) + idx1]>15)
    printf("%u-%u-%un", score[i*(6561) + idx1], pCount, dev_maxScores[idx1]);
else {
    score[i*(6561) + idx1] = 0;
}
}

// Make all threads wait for each other to finish before exiting function... exiting too early can screw things up
__syncthreads();
}
CHAPTER 7
REFERENCES


