5-2014

Graphics Processing Unit Bloom Filters: Classical and Probabilistic

Joshua Michael Pyle
University of Tennessee - Knoxville, jpyle1@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Joshua Michael Pyle entitled "Graphics Processing Unit Bloom Filters: Classical and Probabilistic." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Qing Cao, Major Professor

We have read this thesis and recommend its acceptance:

Michael Thomason, Michael Berry

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Graphics Processing Unit Bloom Filters: Classical and Probabilistic

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Joshua Michael Pyle
May 2014

Acknowledgements

I would like my to thank my master's committee: Dr. Thomason, Dr. Berry, and Dr Cao. I would also like to thank Dr. Cao and his student, Yanjun Yao, for developing the particular Bloom Filter used in this project. I would like to thank Oak Ridge National Laboratory and my two mentors at the laboratory: Dr. Steed and Dr. Horey, for providing me with ideas and motivation that encouraged my professional growth while I have been a graduate student. I would like to thank my friends in India at GTCI and in Knoxville/ Oak Ridge at ORNL for their support and kindness. I would like to thank my parents for their consistent love and support.

Abstract

Graphics Processing Units (GPUs) have been used to enhance the speed and efficiency of both data structures and algorithms alike. A common data structure used in Computer Science is the Bloom Filter, which is used in many types of applications including databases and security logging. The Bloom Filter is a lossy data structure that uses several hash functions to store keys into a bit array. A novel, new Bloom Filter meant for use in internet traffic detection called the Probabilistic Bloom Filter has recently been developed. In practice, this new Bloom Filter typically makes use of more hash functions than its classical counterpart. Because both of these data structures contain information that can be inserted in independent batch operations, this makes each data structure a prime target to be parallelized on a Graphics Processing Unit. This paper develops a scalable, optimized Graphics Processing Unit implementation of the classical and Probabilistic Bloom Filters. The results of processing the Bloom Filter on the Graphics Processing Unit (GPU) are compared to processing the same Bloom Filter on the Central Processing Unit (CPU). By processing the data structures on Graphics Processing Units, a substantial decrease in processing time was observed and recorded. For most cases, the decrease in time was linearly proportional to the number of keys inserted and the number of hash functions used.

Preface

The project was developed by Joshua Pyle for the Master's of Science degree in Computer Science.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and General Information

1.1 Introduction

As recently as 2012, the amount of data that existed in the entire digital universe was approximately 2.7 zettabytes [18]. In order to efficiently process and extract information from this much data, novel new algorithms and their implementations must be developed. In the last decade, scientists have started relying on Graphics Processing Units to enhance the efficiency of algorithms originally designed for Central Processing Units. The inherit, parallel nature of these machines has the capability to greatly speed up parts of algorithms that can be processed in a parallel manner. By implementing existing algorithms in a parallel architecture, perhaps a significantly larger portion of the digital data available can be more thoroughly processed and better understood. With some industry insiders suggesting that Moore's law may be showing signs of slowing [19], Graphics Processing Units may become more important as time progresses.

1.1 Classical Bloom Filter

In 1970, Burton Bloom developed the idea of a data structure where a form of data compression could be achieved in scenarios where false positives could be safely mitigated and false negatives could be costly [1]. In this paper [2] , Bloom laid out the idea of using a bit vector of length m that could be used to efficiently store n keys whose prior existence in the bit vector could be retrieved in a binary manner. The bit vector Bloom developed is able to support two types of operations: insertions and queries, both

of which make use of k independent hash functions. In a typical implementation, the keys being inserted are stored as strings, and all of the bits in the bit vector are initialized to a value of zero. The steps for inserting an item into the bloom filter, summarized from [1], are shown below.

1) Compute k independent hashes of the key being inserted.

2) Set the value of the bit vector at the index of each calculated hash to 1.

In order to query an item from the bit vector, the following steps are performed [1].

1) Compute k independent hashes of the key being inserted.

2) Determine if each bit at the index of each calculated hash has a value of 1.

At the end of the second step of the query operation, it is determined if each bit at the index of each calculated hash has a value of one. If this statement is true, then the query operation should return that the key has been previously inserted into the bloom filter. If this statement is not true, then the query operation should return that this key has not been previously inserted.

Unfortunately, even if both operations listed above are performed correctly, keys that have not actually been inserted into the bloom filter can fool the query operation into thinking that they have been previously inserted.

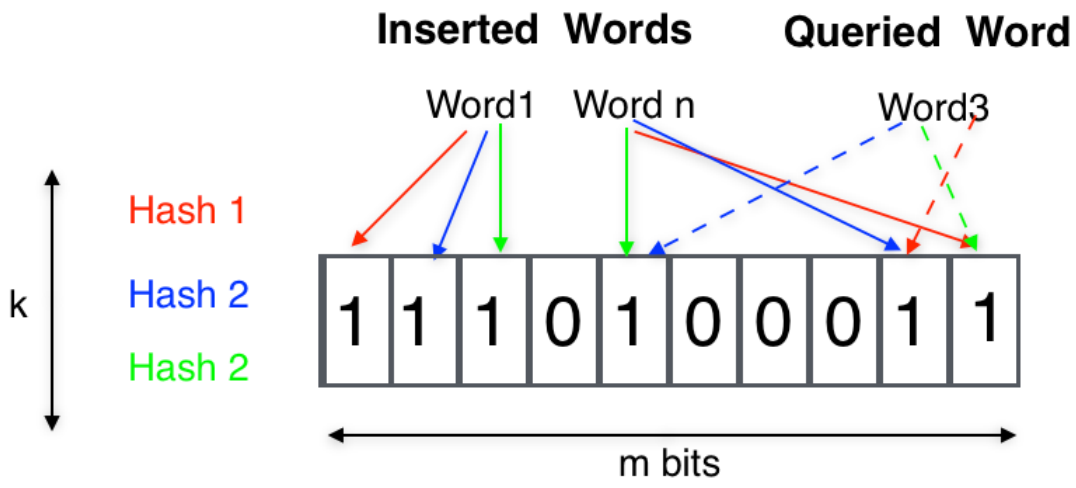

Figure 1. Default Operation of a Bloom Filter with a false positive.

Figure 1 shows a visual representation of a possible scenario of a bloom filter. In this scenario k is three, n is two, and each bit in the bloom filter that corresponds to a calculated hash of an inserted word is set to one. This example also shows that there is a word that is being queried for existence. After the queried word has k hashes computed on it, each computed index value is checked. In this example, each computed index has a value of one, even though the word was never actually inserted into the bloom filter. Because each value is equal to one, the query operation has been fooled into thinking that the item was actually inserted into the bloom filter.

1.2 Probabilistic Bloom Filter

Recently, a variant of the basic Bloom Filter called the Probabilistic Bloom Filter was introduced in [4]. This particular variation was meant to address the issue of efficiently picking out keys that have been inserted magnitudes more often than other keys in the bloom filter. The Probabilistic Bloom filter, as its name implies, utilizes a

random number generator and a predefined threshold called p. The insertion algorithm, described below, decides if a bit in the bit vector should be set to one [4].

1) Calculate k independent hashes of the key being inserted.

2) For each k, generate a uniform random number between 0 and 1.

3) If and only if the value of the random number is less than p, set the value of the bit vector at the calculated hash index to 1.

In the Probabilistic Bloom Filter, the frequency values can be used to determine if a key has been inserted a large number of times. There are three values of interest : frequency (f), minimum frequency (fmin), and maximum frequency (fmax) [4]. Fortunately, all three of these values can be calculated in a straightforward manner by first determining the number of bits in the bit vector set to one for a particular key. The steps to calculate the number of ones for a particular key are described below [4].

1) Calculate k independent hashes of the key being inserted.

2) Count the number of bits set to one across each calculated hash index.

After the count of the number of bits set to one has been retrieved, the three frequency values described above can easily be calculated.

1.3 Compute Unified Device Architecture

A popular manufacturer of Graphics Processing Units is NVIDIA. In order to give users the ability to harness the computing power of these devices, NVIDIA offers the CUDA™ platform , consisting of the architecture and the application programming interface aimed specifically at NVIDIA hardware. The CUDA application programming

interface is SIMT (Single Instruction, Multiple Threads) which means that each thread runs the exact same code, but with different indices [5]. It is important to note that users are not limited to using CUDA; there are other application programming interfaces that exist, most notably OpenCL ™ [16]. OpenCL has the advantage of supporting multiple platforms [16], but OpenCL must be used carefully with vendor specific directives in order to get comparable performance to CUDA on a NVIDIA GPU [17].

1.3.1 Hardware Layout

NVIDIA's architecture offers three main abstractions to programmers: a hierarchy of group threads, shared memories, and barrier synchronizations [5]. First and foremost, it is necessary to understand how the threads are mapped to the application programming interface. Each device contains one grid. The dimensions of the grid can be configured by the programmer; the programmer can choose to use one, two, or three dimensions. Each item inside a grid is referred to as a block, and can be uniquely identified using either one, two, or three numbers depending on the number of dimensions chosen. Finally, each block contains the individual threads that are actually executed. Much like grids, blocks can be organized into one, two, or three dimensions of threads. Each thread can be uniquely identified using one, two, or three numbers, and the identification of the block.

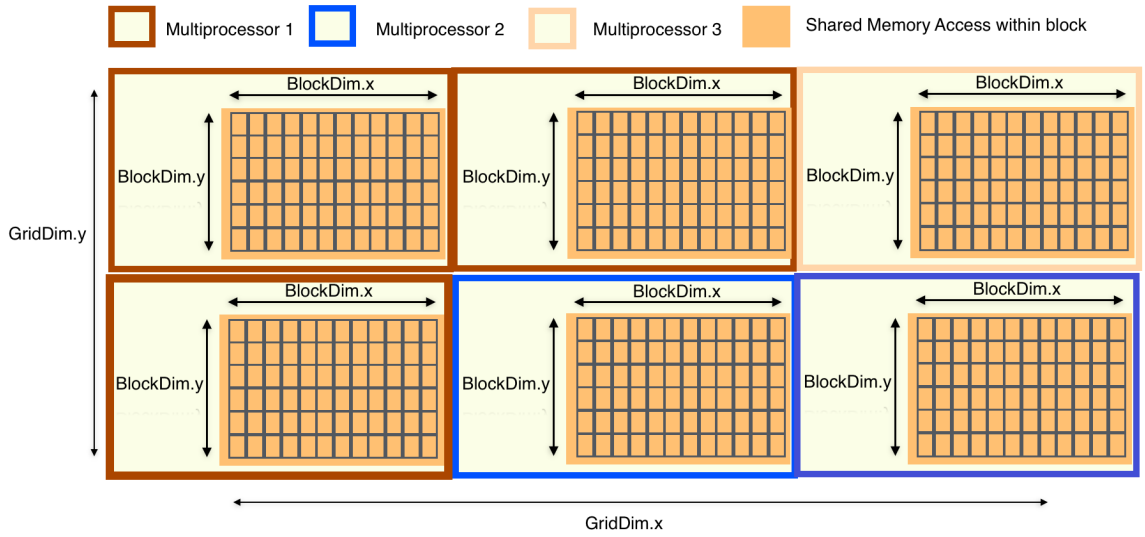

Figure 2. Layout of a GPU

Figure 2 shows a potential layout for the thread hierarchy in CUDA. Essentially, in this example, there is one grid that consists of six unique blocks. The blocks are laid out in a two by three format, with two rows and three columns. Each individual block consists of seventy-two total threads that are laid out in a six by twelve format. Shown in in the figure but discussed in detail in the next sections are multiprocessors and shared memory. Nonetheless, the example given in the figure shows that the threads of each block have a shared memory area, and that different blocks map to different multiprocessors for execution.

1.3.2 Memory Configuration

There are several different types of memory available in the CUDA architecture, each having different scopes and purposes. Perhaps one of the most important ways to to

optimize code running on the CUDA platform is to thoughtfully and carefully use and measure each type of memory appropriately [6].

Table 1: Different types of CUDA memory that are important for optimization. [6]

Memory Type	Scope
Register	1 Thread
Shared	1 Block
Global	1 Grid
Local	1 Thread

Table 1 lists the different types of memories that are available to the CUDA programmer. The quickest type of memory is the register space, which is used to store values of the variables used on the GPU. Typically, there are a set number of registers available per block whose quantity varies depending on the device. Each register belongs to a distinct thread and can not be accessed by other threads [6]. If there are not enough registers to store the different variables, the variables get stored in a slow, local memory.

The second quickest type of memory is the shared memory. It usually functions as a user-controlled cache. Like the register memory, there is a set amount of shared memory per block. However, unlike the register memory, every thread in the block can access the same shared memory [6]. One normal optimization pattern is to store several values from global memory into shared memory for quicker access times [6].

Magnitudes slower than the shared memory is the global memory [6]. As its name implies, global memory can be accessed from any thread inside of any block. Perhaps the

most important feature is that global memory persists through multiple functions calls on the GPU device. The amount of global memory available is usually quite large (around 1024 MBytes). Unfortunately, the access times to global memory are quite slow and should be used as scarcely as possible.

1.3.3 Thread Execution

The manner in which a thread gets executed in the CUDA architecture is straightforward to the programmer. Each whole block gets mapped to a streaming multiprocessor in a way that the programmer has no control over [5]. After a block has been assigned to a multiprocessor, the threads of a block are divided up into groups of thirty-two threads called warps. A warp scheduler will then determine which warps are not waiting for global or shared memory, and which warps are not behind a synchronization barrier [5]. After the warp scheduler selects an eligible warp, it issues instructions to it, and the warp executes using some of the available CUDA cores.

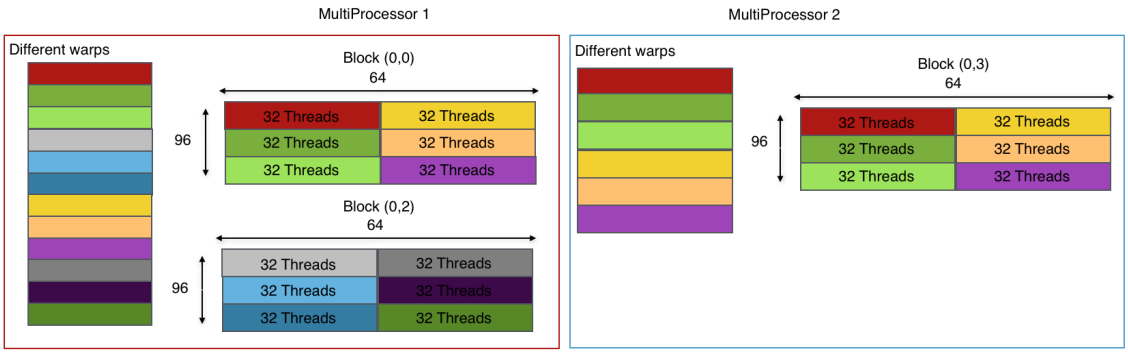

Figure 3. How blocks are mapped to multiprocessors and threads are mapped to warps.

Figure 3 shows a possible assignment of threads to individual warps of a multiprocessor. As the figure shows, the warps are mapped along the rows if the rows are multiple of 32. Each block can execute multiple warps, and each multiprocessor can execute multiple blocks.

Chapter 2

Implementation Overview

2.1 Prior Work

2.1.1 Bloom Filters

There have been two works published to IEEE on the implementation of classical Bloom Filters on Graphics Processing Units. Each paper, while not fully suited for the particular situation covered in this thesis, offers great insights into certain optimizations and programming abstractions that can be made.

The first paper ,written by Costa et al [8], includes support for mass, parallel insertions and queries of data on a bloom filter. Unfortunately, this work does not take into consideration how the memory hierarchy in CUDA could be used to speed up both insertions and queries. The authors rely solely on global memory , which has two main downfalls. Firstly, as stated in the previous section, global memory has a high access time and low bandwidth compared to other types of memory. Secondly, if the accesses to global memory are not done correctly, the local caches on the GPU will not work effectively. However, the authors did stumble across a clever way to allocate memory on the GPU. Instead of allocating the keys into an array of arrays, which produces significant overhead on the CUDA platform, the authors allocate the keys into a single array. The authors then allocate an index table of the keys, including the starting position

and length of each key. By allocating two total arrays instead of an array for each key, the authors managed to speed up the execution time by a large amount.

The second paper, written by Ma et al [9], introduces a classical Bloom Filter designed to speed up a genome processing application. Unlike the first paper by Costa et Al [8], this paper does make explicit use of the CUDA memory hierarchy. Ma et al purposely make use of shared memory in order to speed up accesses to the Bloom Filter used in the application. The authors divide the bit vector into partitions of a set size, copy the bit vector into shared memory on the GPU, and then process each partition separately. Instead of having each thread calculate one hash function, the authors have each thread calculate multiple hash functions in order to cut down on overhead caused by spawning more threads. The authors of this paper did not include an algorithm for an insertion function as their data used in the bloom filter was already stored in a database.

2.1.2 Performance Metrics

Optimizing an application on the Graphics Processing Unit is complicated, involving several different variables and constraints. Fortunately, lots of research has been done on ways to express the efficiency of a GPU application.

Ma et al, in [9], proposed a simple equation to estimate the number of concurrently executing blocks on a GPU. They express shared memory in terms of S, the shared memory used by each thread, and S_B , the maximum amount of shared memory per block. The amount of registers used is expressed in terms of R, the number of registers, and R_T, the number of registers used by the kernel function per thread . The

number of threads requested per block is labeled as T_R. The number of multiprocessors available in the system is labeled as MP. Finally, B_{max} , represents the maximum number of blocks that can be used while T_{maxMP} represents the maximum number of threads available to a multiprocessor. The number of active blocks can then be represented by:

$$B_a = min(\left\lfloor \frac{S}{S_B} \right\rfloor, \left\lfloor \frac{R}{R_T \ x \ T_R} \right\rfloor, \left\lfloor \frac{B_{max}}{MP} \right\rfloor, \left\lfloor \frac{T_{maxMP}}{T_R} \right\rfloor) \qquad (1)$$

In order to attempt to balance the number of blocks used per multiprocessor, the optimal number of blocks chosen should be a multiple of the number of multiprocessors :

$$B_{opt} = \{B_r = i \ x \ B_a \ x \ MP \mid i \in N\} \qquad (2)$$

Much like the optimal number of blocks, the optimal number of threads should be a multiple of the number of threads in one warp:

$$T_{opt} = \{T_r = i \ x \ MP \mid i \in N\} \qquad (3)$$

In [10], the authors create a generalized performance model that can be used to predict and model a GPU application's performance. This equation assumes that T_{opt} has been selected to be a valid value within the constraints of the GPU.

$$Time \propto f_{app}(\overrightarrow{algo}, \overrightarrow{inpt}) \ x \ f_{cache} \ x \ f_{sched} \qquad (4)$$

$$f_{sched} = \frac{\left\lceil \frac{B_r}{B_a \ x \ MP} \right\rceil \ x \ B_a \ x \ MP}{B_r} \qquad (5)$$

$$fcache = min(1, \frac{C}{m}) + (1-min(1, \frac{C}{m}))*G \qquad (6)$$

Equation (4) represents the factors on which execution time depends. The variable f_{cache} represents the cost of cache misses that take occur by randomly accessing memory. The variable f_{sched} determines the cost of scheduling blocks. Equations (5) and (6) show the models for the block scheduling and the cache misses, respectively, if the program's data can not fit into the working space. The variable C represents the size of the cache, and m represents the working size in memory. The value G represents a scalar value of the cost of a low cache hit rate. It is important to note that the Bloom filter contains many random accesses to the bit vector; therefore, many parts of this algorithm are specifically suited for modeling applications like bloom filters.

2.1.3 Memory Accesses

Models in the previous section do not take into account the benefits or costs of laying out memory in an optimal manner. For instance, if the entire working space of an application fits into the cache of an application, and the memory access patterns are not optimized, then the program may not behave in an optimized fashion. Another example of worse than expected performance could include accesses to global memory. If memory access patterns to global memory seem random in mature, then the throughput of the application may suffer by a large amount [5][6][12].

In [12], the authors discuss various types of problems that users can encounter when accessing memory on a CUDA device. Some of these problems are limited to accesses when global memory is used — improper balancing of channel skews, and misuse of access patterns leading to no coalesced accesses of memory. Other problems

are only related to shared memory use, such as shared memory bank conflicts. The authors provide great examples of causation and correlation where unoptimized accesses to global memory mean execution time increases of tens of seconds. Also, examples of this problem are given in both [5] and [6]. The authors in [12] also created a program called CuMapz that would observe and track variables related to these issues. Unfortunately, the source code and binaries do not appear to be publicly available.

2.2 Contributions

This thesis builds upon previous literature to produce a fully functional, optimized, classical Bloom Filter and Probabilistic Bloom Filter. In order to optimize memory accesses and layout, this thesis draws upon extensive knowledge of how the Bloom Filter algorithms work and how memory access problems described in [5],[6], and [12] can be mitigated. The performance models described in [9] and [10] can be used to help speed up execution time by figuring out the best way to allocate GPU resources towards performing the computations.

2.2.1 Scalable Design

The data that can be inserted into a Bloom Filter or queried from a Bloom filter can come in many different lengths and sizes. Some of the keys may be very large in nature, while some of the keys may be smaller. Some groups of keys (referred to as batches), can be in the hundreds of MegaBytes, while other batches can be in the kilobytes. This non-uniform, wide range of domain values is a challenge for the CUDA application programming interface for several reasons. In fact, the CUDA application

programming interface, listed in [5], does not even have a function that will allocate a non-uniform, two-dimensional array of values. However, the biggest problem with a non-uniform input domain is that accessing global memory can occur in irregular patterns, causing unoptimized caching.

Some Bloom Filters, like the Probabilistic Bloom Filter, require more hash functions than other designs. In fact, the Probabilistic Bloom Filter generally uses over a thousand hash functions per word when items are inserted [4]. On the other hand, it is common to see Bloom Filters where the number of hash functions used is less than ten.

This thesis proposes a Bloom Filter design that offers predictable performance across a wide variety of inputs. Regardless of the number of hash functions specified or the size of the input (up to a certain point), the design will scale to accommodate most combinations. The amount of GPU resources scales in a fashion that tries to optimize (1) without too much overhead. Also, the design introduced in this paper seeks to minimize the memory access pattern problems described in [5],[6], and [12] that are prone to occur, especially with a variety of input sizes.

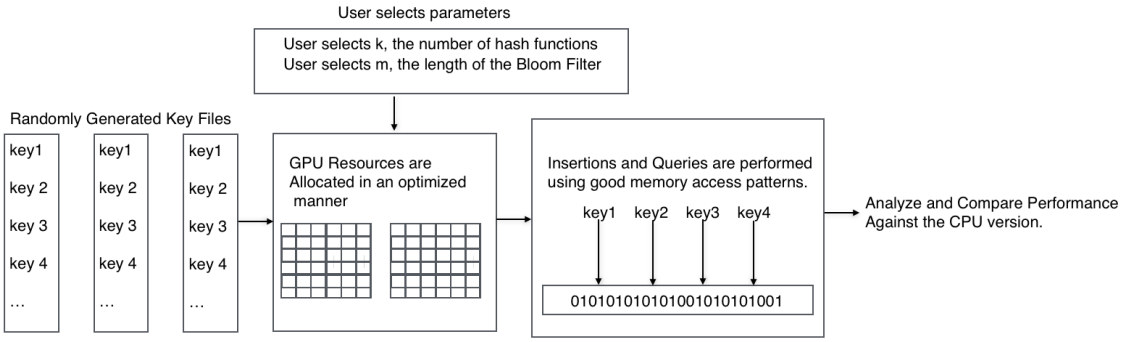

Figure 4: Thesis Proposal diagram.

In order to test the design of the Bloom Filters, several test cases are created. The test cases include data that reflect a variety of input sizes, causing the design to make strategic decisions on how to best allocate resources. For testing purposes, the number of hash functions used was also varied in each test case in order to further put strain on the proposed design. Finally, the results of the GPU Bloom Filters are compared to the results of testing the Bloom Filters on the CPU. Figure 4, shown on the previous page, summarizes the proposal.

2.3 Limitations and Problems

There are some problems that can occur when trying to build an application on the Graphics Processing Unit. One of the biggest problems is that the GPU is usually purposed for more specific tasks such as graphics rendering. If any other application attempts to spend too much time on the GPU, then the operating system will usually kick the application off of the GPU. If the GPU is operating in a shared system and other users are launching applications on the GPU, then the performance of the application may also deteriorate. In order to bypass some of these problems, this paper uses a GPU that performs no Graphics Processing and is not on a shared system.

Chapter 3

Graphics Processing Unit Implementation

3.1 Allocation of Resources

As discussed in the previous section, the best way to optimize and speed up a CUDA application is to allocate and access the resources in a clever manner. This section discusses the preprocessing required to ensure that the algorithm can run in an optimal fashion.

3.1.1 Allocation of Bit Vectors

As shown in section one of this paper, each Bloom Filter contains a bit vector that contains m bits. In some Bloom Filter implementations, the bit vector is represented as character array that contains either a zero or a one. In other implementations, each character in a Bloom Filter's character array contains a field of eight bits that can be individually addressed via bit shifts. This second way of representing the bit vector as a character array is prone to race conditions in applications where multiple threads try to write to the same index in the character array. A race condition happens when two different threads are modifying the overall value of the same index in the character array. Of course, this situation can be mitigated using locks or atomic functions, but the use of such features causes slower performance. By using the first method described, the probability of this race condition happening is much smaller than the second method due to having less indices map to the same index of the character array. For this reason, the authors of [8] decided to treat each specific index of the character array as a bit.

Likewise, in this thesis, it is decided that each index of the character array should be treated as a bit. This implementation does not address individual bits.

The bit vector used by the Bloom Filter has certain requirements. First and foremost, the bit vector should be able to persist the entire length of the application. An application should be able to insert multiple sets of data and should also be able to query multiple sets of data. On top of having to be persistent, the memory used by the bit vector is usually quite large as m is typically a large value for Bloom Filters.

The only type of memory in the CUDA architecture that supports persistency and large swaths of memory is global memory. Even though global memory is slower than other types of memory, it is necessary to store the bit vector in its entirety in global memory. The bit vector representing the overall Bloom Filter is stored in global memory.

3.1.2 Allocation of Keys

The values that can be inserted into a Bloom Filter, referred to as keys, are usually stored in files containing thousands of keys. In order for the Graphics Processing Unit to be able to process each key from a file, the keys must be loaded into a memory space that is large enough to accommodate each key from a file. The only memory space available in the CUDA architecture for large swaths of data is the global memory. However, in the design proposed in this paper, global memory is just a temporary spot where the keys stay before they can properly be processed.

Unfortunately, the length of each key can vary tremendously. The only support CUDA has for allocating arrays in a two dimensional manner requires that each key be

the same length [5]. In order to get around this issue, an approach similar to the what was used in [8] is used . In [8], the authors decide to concatenate each key into a large single array of keys. The authors then decide to create an index table consisting of the starting index and length of each key in the array. The index table is also stored in global memory. However, in this thesis, the keys are concatenated into a single, large array separated by commas. In order to cut down on the amount of global memory space used, only the starting index of each word is stored.

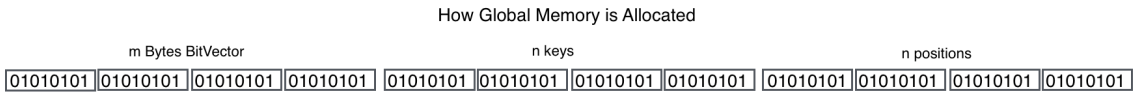

Figure 5: How Global Memory is allocated

Figure 5 summarizes how global memory is allocated. The three sections are located in a single, linear manner.

3.1.2 Allocation of Threads and Blocks

In order to allocate the blocks and threads of a CUDA device in an optimal manner, many parameters and values must be kept in mind. For instance, it is important that the number of blocks per multiprocessor is set to a maximum, so that more threads can execute concurrently. This can be done by trying to to optimize equation (3) and equation (1). It is also important that the number of threads allocated be a multiple of the warp size, equation (3). If these parameters are not optimized, it typically implies that there are threads sitting idle that could be executing. For accessing shared and global memory, it may be important to know which threads map to which warp Ids.

In an ideal situation, in order to optimize equation (1) and to satisfy the requirements of equations (2)(3), the following steps should be taken.

1) The amount of shared memory allocated to each block is equal to the max number of shared memory divided by the maximum number of blocks per multiprocessor. In this case, that value is eight [5].

2) Ensure that the number of threads per block are no more than the number of threads per multiprocessor divided by eight. Again, eight is chosen because that is the maximum number of blocks per multiprocessor [5].

3) Ensure that the number of registers used across eight blocks is less than the maximum number of registers per multiprocessor.

Unfortunately, most situations of a GPU Bloom Filter fall outside the spectrum of an ideal situation. For instance, if the number of keys being inserted into the Bloom Filter is quite large, then problems can arise quite quickly. As stated in section one , each device contains one grid. Unfortunately, each grid does not have an infinite number of blocks available. In the ideal situation, the number of threads per block is scaled down so that more blocks may be executed at the same time on a multiprocessor. By scaling down the number of threads being used per block, the number of blocks spawned is increased. Often times, if there are a large number of keys being inserted at one time into a Bloom Filter, the number of blocks spawned will actually be more than CUDA can handle, causing an error. In order to handle the typical situation, the following steps are performed.

1) Allocate the maximum amount of shared memory to each block. The amount of shared memory is being allocated is not of concern here because only one block will be able to launch at a time.

2) Allocate as many threads as possible to each block.

The steps listed above do not provide an optimal solution; rather, the steps listed above describe a general solution for any use case. Because this paper looks into scalable solutions, this method is chosen.

In order to allocate the threads and blocks used, the following steps are taken.

1) Figure out the maximum number of keys that can be processed by a block by dividing the maximum number of threads per block by the number of hash functions requested.

2) For each block, the number of columns should be the number of hash functions rounded to the nearest value of 32.

3) For each block, the number of rows should be equal to the the maximum number of threads per block divided by the number of hash functions rounded to the nearest value of 32.

4) Allocate the number of blocks needed. Start by filling up the first column of the grid, and add more columns as necessary.

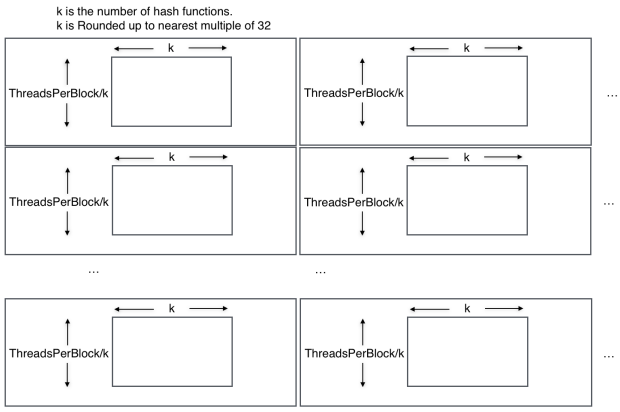

Figure 6: How Threads and Blocks are Allocated.

Figure 6 shows a description of how blocks and threads are implemented. The number of columns of each block matches the number of hash functions, padded to the nearest value of 32. The number of rows is equal to the maximum number of threads per block divided by the number of hash functions being used, padded to the nearest value of 32.

3.2 Algorithm Implementation

At this point in the process, all the variables needed have been allocated to global memory, and all the blocks and threads that are needed have been allocated. Inside global memory are the keys that will be inserted stored along with an index table used to figure out where each key is located at. Also inside global memory is the character array that represents the Bloom Filter's bit vector. Now, with all the resources allocated, the actual implementation of the algorithm can be performed.

3.2.1 Implementation

As discussed in the previous two sections, the goal of this thesis is to perform the Bloom Filter operations in a single batch mode. Fortunately, with all of the blocks properly allocated, performing the different operations becomes quite simple. The steps listed below and on the next page describe how the insert and query algorithms work.

In order to insert items into the Bloom Filter, the following logic is used:

1) Each word gets assigned to a row of a block.

2) Each column of each row gets assigned to a hash function of a word.

3) The first thread of every row looks up the indices of the word it is responsible for.

4) Based on the index of the word calculated, the first thread of every row stores the word it is responsible for into the fast shared memory.

5) Each column of every row calculates the hash of the word it is responsible for. In this step, each column consults the shared memory, and not the slow global memory for each word access.

6) Each column of every row writes a value of one to the index of the Bloom Filter that has been calculated.

In order to insert items into the Probabilistic Bloom Filter, the same steps listed above can be used, with one minor modification. In step six, a random value should be calculated. If the random value is less than the probability specified by the user, then the current thread should write a value of one to the index of the Bloom Filter that has been

calculated. If the random value is greater than or equal to the threshold probability specified by the user, then the Bloom Filter should not be written to. While this may not seem like a major performance hit on the surface, generating the random number used can be quite difficult. Fortunately, the CUDA api provides a high quality random number generate that can be used.

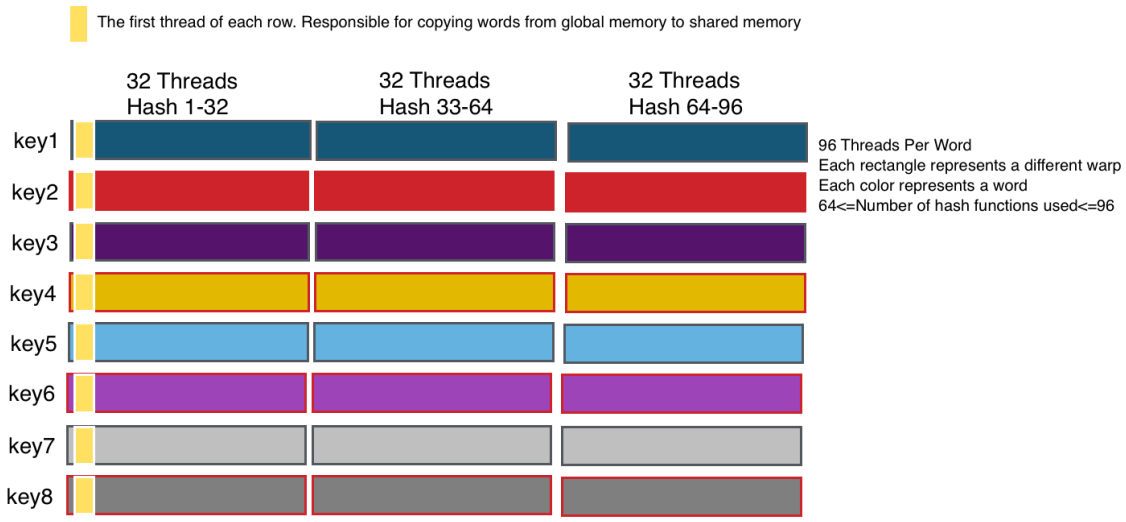

Figure 7: Representation of Keys and Hash Functions for a particular insertion

Figure 7 shows how the words and hash functions are allocated throughout a thread block for both implementations. Each block in a grid will have threads in a similar structure as to what is shown here. If a particular block in a grid does not have any words to process, the block will just return without doing anything.

Now, the only operation left to implement is the query operation. The following steps can be used to query a Bloom Filter.

1) After allocating the words and index table, in global memory allocate a character array initiated with all ones for each key. This is the results array.

2) Each word gets assigned to a row of a block.

3) Each column of each row gets assigned to a hash function of a word.

4) Based on the index of the word calculated, the first thread of every row stores the word it is responsible for from global memory to shared memory.

5) Each column of each row calculates the hash of the word it is responsible for.

6) If any column of each row calculates a hash index of the Bloom Filter that does not have a value of one, the index of the array from step (1) is set to 0.

7) Now, each key that exists in the Bloom Filter has a value of one in the results array. If any index of the results array has a value of zero, then that item was not found in the Bloom Filter.

It should be noted that each of these situations eliminates a race condition problem because each thread only writes a value if it must. When multiple threads have to write a value, each thread will write the same value to the same memory location. However, the Probabilistic Bloom Filter query is a little bit different than the previous situations.

The Probabilistic Bloom Filter's query algorithm looks really similar to the basic Bloom Filter query algorithm. Unfortunately, the performance is a little bit worse than the regular Bloom Filter's query algorithm due to the need of an atomic addition function.

Essentially, the only changes needed to the regular Bloom Filter's query function to transform it into a Probabilistic query are described below.

1) Allocate a result array of integers, not characters. The length of the array should be equal to the number of keys being being queried.

2) Instead of writing a value of zero to the results array if and only if the hash calculated index has a value of zero, sum up the value of each calculated hash index. This summation will consists of ones and zeros. In order to make sure that there are no race conditions, the addition function should be the CUDA atomic add function. While this atomic function may slow down performance, it is the only way to correctly add up items across multiple threads. After the summation been calculated, the summation should be written to the index of the results array for the current key. Thus, each key will have a count of the number of ones across each hash function, and each of the three values of interest (f,fmin,fmax) can easily be calculated on the host side.

One item of interest that was glossed over in previous paragraphs was the hashing technique that was used. The same hashing technique was used for both Probabilistic and Classical Bloom filters, and for both insertion and query operations. There are a few ways to perform hashing on a Graphics Processing Unit, and either way makes use of the fact that each thread can be uniquely identified with an integer. Some hash functions, such as the Murmur hash function, take in a seed that is an integer. One easy way to generate several different Murmur hashes of the same word is to use different seeds. In the Graphics Processing Unit environment, each of the hashes using different seeds can be

performed at the same time. Another option is to use the technique discussed in [3]. The method discussed in three takes two distinct, unique hash functions, and a third function. The third function is a simple function that is usually linear, exponential, or squared. The following page has a diagram that explains how each hash function is computed.

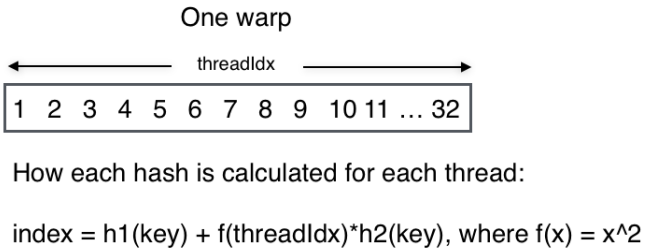

Figure 8: Hashing Technique

Figure 8, shown above, summarizes how hashing is done for this Bloom Filter using the method described in [3]. Each thread calculates two simple hash functions and then calculates the results of a third function based on the thread identification number. The results of the three functions are used to calculate the index that should be selected in the Bloom Filter's bit vector.

3.3 Memory Access Patterns

The previous sections discuss how the algorithms are implemented and how the resources of each Graphics Processing Unit are allocated. This section discusses the logic behind the way the memory allocations were made in order to avoid the problems listed in [5][6][12].

3.3.1 Global Memory

As stated in previous sections, the random access patterns of Bloom Filters to its bit vector implies that many accesses to global memory can not be optimized. However, there are some accesses to global memory that are not random in nature. For instance, in this thesis the keys are copied from global memory to shared memory. The individual bytes that make up each word are located sequentially in global memory. Unfortunately, each key is a random length, and thus it is impossible to have the global memory accesses be optimized by requesting only information in the same cache line. If the words were stored in a format of 128 bytes, the length of a cache line in CUDA 2.x, then fewer overall memory accesses would be needed to load the information [5][6]. However, there is no guarantee that the words will be aligned to those cache lines, so a thread may need to read from multiple cache lines when loading a word from global memory.

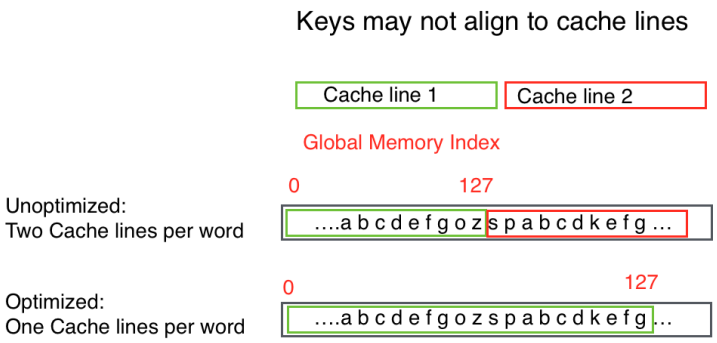

Figure 9: Why some global memory accesses are not optimized.

Figure 9 explains the case of an unoptimized access to global memory. In the figure, the first example has a word that spans across multiple multiple cache lines that are aligned by 128 bytes. In this case, each warp will have to make two memory accesses

instead of just one. However, in the second case, the word fits perfectly in just one cache line. Because the word is aligned to the cache line, only one memory access has to be made.

3.3.2 Shared Memory

Unlike global memory, the accesses to shared memory can be optimized. One of the biggest problems with Shared memory is that each byte is assigned to a shared memory bank. Each of these banks can be accessed by individual threads in a warp. If two threads inside of a particular warp ask a particular memory bank for two different memory addresses simultaneously, a bank conflict occurs [5][6][12]. In this particular situation, the memory requests will be processed in a serial manner. However, if multiple threads all request the same memory address from the same bank, then the value of the memory address gets broadcast to all threads who requested the memory. This does not slow down the access to shared memory.

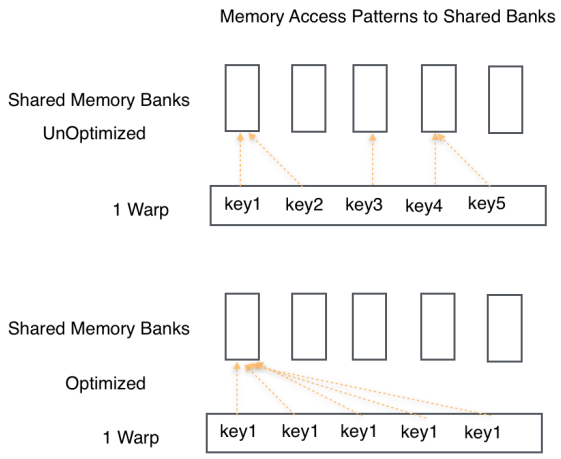

Figure 10: Shared Memory Access Patterns

Figure 10 shows an example of two shared memory access cases. The first case is not optimized because multiple threads in the same warp access the same memory bank looking for different memory values. That is, the threads that are processing key 1 and key 2 both try to load information from the first memory bank. Because one thread is looking for key 1 and the other thread is looking for key 2, a bank conflict occurs. In the second example, all threads in the warp access the same memory bank looking for the key 1. Because each thread accesses the same memory bank looking for the same value (key 1), the memory bank can broadcast the requested value out to each thread. This is the optimized case.

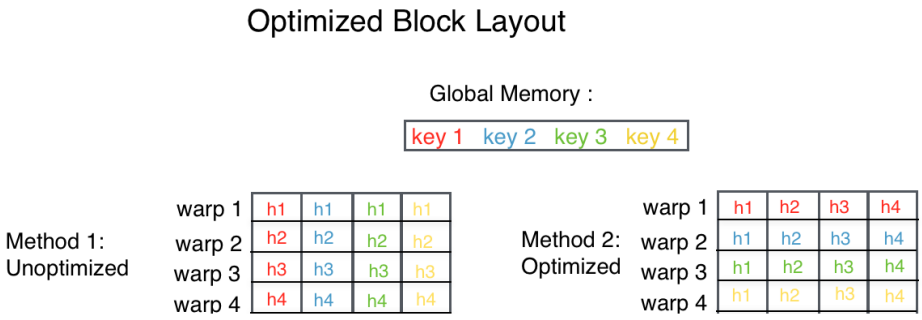

Figure 11: Optimized Block Layout for shared memory

Figure 11 shows two types of ways to allocate hash functions and keys to a particular block. As discussed in the previous section, this thesis uses one key per row with multiple hash functions. However, the alternative, shown as Method 1, does not allow for optimized Shared Memory Accesses. This is because there are multiple keys per warp, and that means that each key may be stored in a different memory bank.

3.3.2 Thread Divergence

There is also a second benefit to using Method 2 from Figure 11. All warps try to execute in lock step. Unfortunately, a program that computes hash functions on a key usually contains multiple branches. The branches happen inside of a for loop while the hash is being computed. Sometimes, one thread of a warp may go down a different branch of a program than another thread of the same warp. If two threads are on different execution paths, then the warp must execute each thread separately in a serial fashion. However, if each thread of a warp is processing the same key, then each thread will be on the same execution path. If a warp contains multiple keys that it is processing then there is a large chance that the warp will experience some form of divergence. By assigning each key to a particular row, thus mapping each key to one warp, the issue of divergence was eliminated.

3.4 Scalability of the system

One requirement of the system was that it had to be able to support a large number of hash functions. A major problem with supporting a large number of operations is that there are a limited number of threads inside of a block. For instance, if a user wishes to create a Bloom Filter that hashes each key over 1024 times, then the computations will spread over multiple blocks.

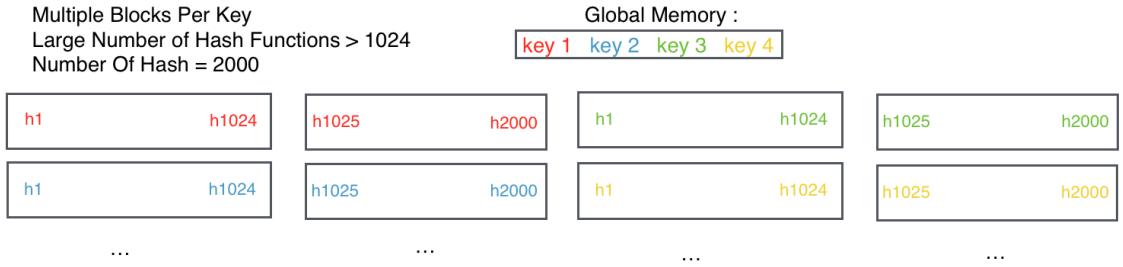

Figure 12: How the hashes of keys can span multiple blocks.

Figure 12 shows how this thesis computes a large number of hash functions on a particular key. In order to make each hash function individual index the particular key, it was chosen to have the blocks expand in a column wise direction. It is important to note that the maximum number of threads per block is around 1024 for many devices [5][6]. However, that number does vary.

Chapter 4

Results

4.1 Experiments

In order to verify the performance of the design, several different test cases are created. The tests are created to stretch the design by using a large number of hash functions and large amounts of global memory. By ensuring that each test case has different numbers of keys and hash functions, the Bloom Filter has to allocate different grid and block sizes for each particular test.

To make sure that the Graphics Processing Unit can efficiently be used to process a Bloom Filter, the performance of both the Classical and Probabilistic Bloom Filters on the GPU are compared to CPU versions of both algorithms. The comparison is done to prove that the Graphics Processing Unit can more efficiently process a Bloom Filter than a Central Processing Unit of similar stock and price. Obviously, if one of the processing units is more powerful than the other, then the competition is not fair. The meaning of stock and price in this thesis implies that the Graphics Processing Unit and CPU are the default units in a stock, Dell computer.

In each test, all of the keys used are generated in the same manner. Each key consists of random letters and numbers. The length of each key is also a randomly generated number between 1 and 150 units in length. For testing, the user specifies the number of batch files containing keys and how many bytes each batch file should consist of. Each batch file is inserted into the Bloom Filter in different operations.

The Probabilistic Bloom Filter is meant to pick out values that have been inserted more often than other keys. To simulate this behavior, the user can specify the number of times to insert a particular batch into the Probabilistic Bloom Filter.

In order to test the performance in terms of false positives and false negatives, a query operation is also implemented. The user has the option of generating several key batches that were not inserted into the Bloom Filter in order to test the false positive rate.

In order to check the correctness of the implementation of the algorithms, both the Bloom Filter and Probabilistic Bloom Filter are written to a file after execution. Both of the Graphics Processing Unit versions are checked against their respective CPU implementations. In addition to the basic Bit Vector in the Bloom Filter, the program also outputs the results of the query and timing information.

4.2 Results

The Graphics Processing Unit that was chosen to perform the tests was a modest Nvidia Geforce GT 620. The webpage listed in [14] lists the resources and computing power of the device. Oddly, the webpages resources contradict what the DeviceQuery program provided by Nvidia gives. Nonetheless, many of the values may seem quite small and really underscore the complexity of allocating resources on a Graphics Processing Unit.

There are some values that play a role in performance that were not discussed in earlier sections. One of these values is the number of CUDA Cores available on the entire

system. Each CUDA Core functions as an ALU or a FLU [5][6]. Also important is the amount of shared memory per block and the maximum number of threads per multiprocessor.

Table 2: Stats of the GPU used.

Resource Type	Value
CUDA Cores	48
Maximum Number of Threads Per Block	1024
Maximum Number of Threads Per MP	1536
Amount of Global Memory	1024 MBytes
Shared Memory Per Block	49152 Bytes

Table 2, shown above, lists the stats of the GeForce GT620 used. As the table shows, the most scarce and valuable resource is the amount of shared memory available per block. Because each key is stored into Shared Memory in a particular block, it is important to keep in mind that total size of the keys processed in each block is limited to 49,152 bytes.

4.2.1 Bloom Filter

The first set of tests was performed on the classical Bloom Filter. The number of hash functions was varied over a wide range of values. The expected behavior was that the execution time of the Bloom Filter would vary linearly with respect to the number of hash functions used. However, due to the spawning of blocks and some requests taking certain values, it was assumed that the increase may not always be consistent.

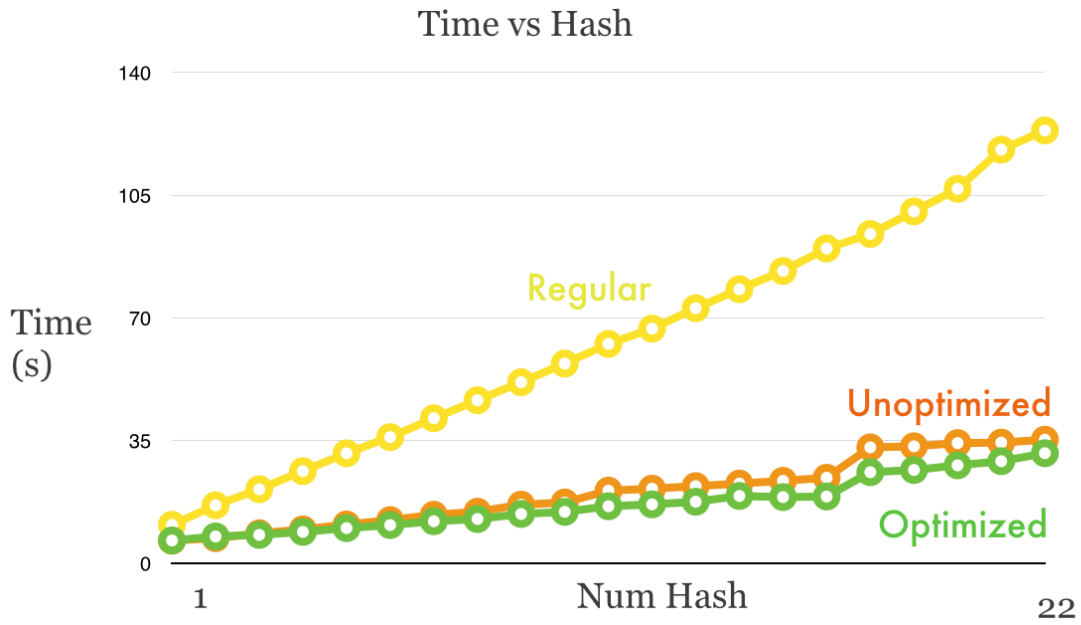

Figure 13: Run times for different implementations of Bloom Filters

Figure 13 shows the results of a classical Bloom Filter on different devices using different methods. The yellow line shown in the figure represents the results of running a Bloom Filter on a CPU unit. The run times of the program are quite predictable in that they increase linearly. The orange line represents running the Bloom Filter on the Graphics Processing Unit in a manner where accesses to shared memory are not optimized. The green line represents running the Bloom Filter on the Graphics Processing Unit in a manner where accesses to shared memory are optimized. As the figure shows, the optimized version is a few seconds quicker than the unoptimized version. As described in the implementation section, this is due to a factor of thread divergence and shared memory access times. Both Graphics Processing Unit implementations are much quicker than the Central Processing Unit implementations. This is proof that the Central

Processing Unit is not as efficient as a comparable Graphics Processing Unit for processing a Bloom Filter.

Next, the amount of data being processed was varied while the number of hashed functions used was kept constant. Again, the expected results were a linear increase in the execution time.

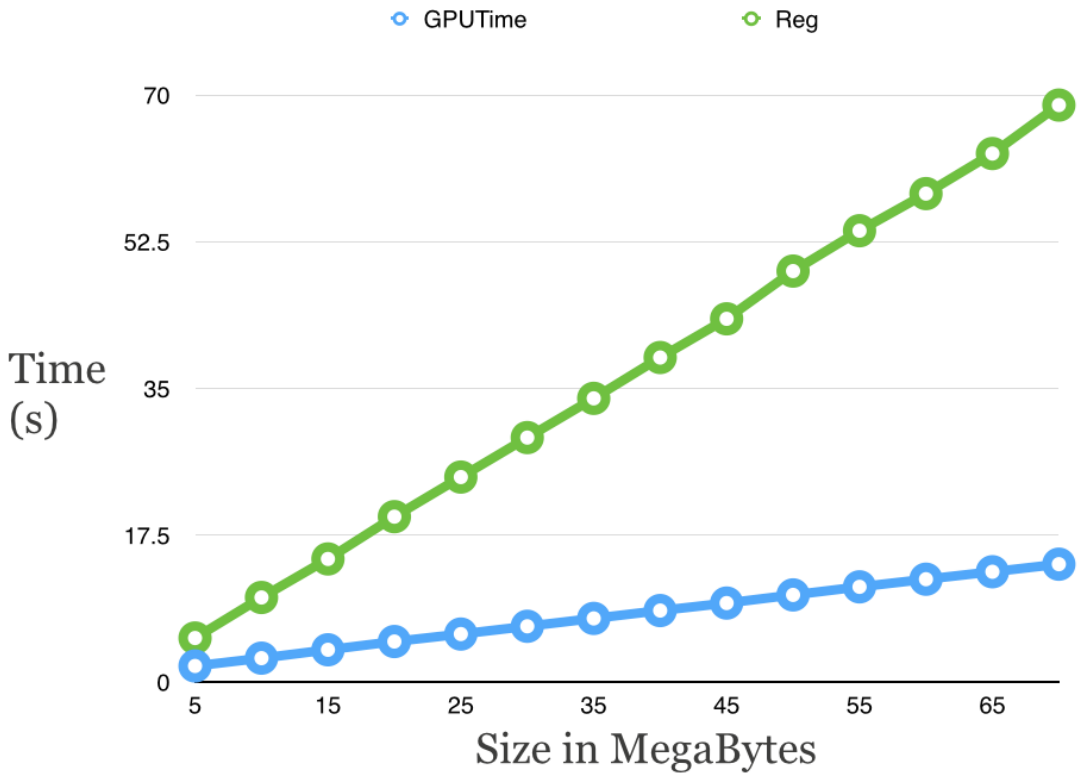

Figure 14: Hash Function constant, number of bytes varied.

Figure 14 shows the results of the experiment described in the previous paragraph. The results prove that the Bloom Filter design can handle a wide range of input sizes for the keys. The results also show that the Graphics Processing Unit Implementation is much quicker than the Central Processing Unit implementation.

4.2.2 Probabilistic Bloom Filter

Finally, the Graphics Processing Unit implementation of the Probabilistic Bloom Filter was tested against the Central Processing Unit version. One of the problems with the Probabilistic Bloom Filter is that random number generation takes a lot of overhead on a Graphics Processing Unit Device. Because of this, the size of the batches must be smaller so that the overhead can be accounted for.

In order to test the Probabilistic Bloom Filter, a test was created where the number of hash functions was varied and the amount of data inserted was kept constant. The number of hash functions used for the tests were quite large in order to reflect the typical use case of a Probabilistic Bloom Filter.

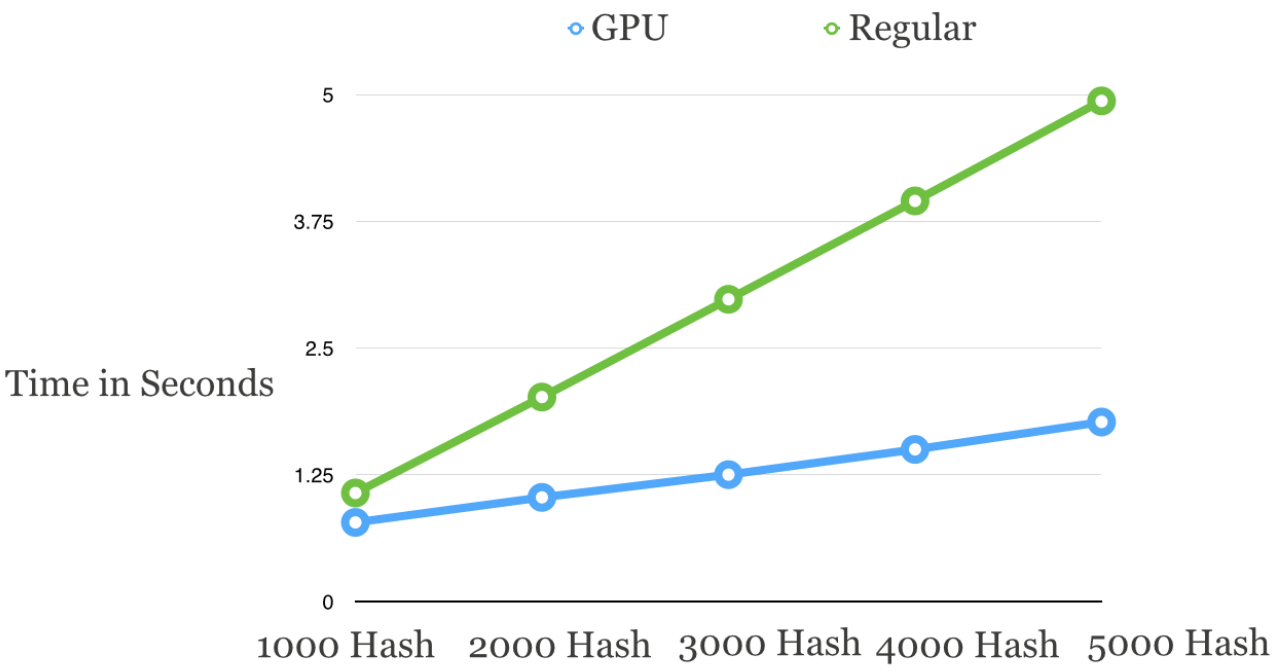

Figure 15: Probabilistic Bloom Filter Results.

Figure 15, shown on the previous page, shows the results of testing the Probabilistic Bloom Filter over a wide range of hash functions. The end result is that the Probabilistic Bloom Filter is almost twice as fast for applications using 3000 hash functions or more. For a small number of hash functions, like 1000, the execution times for the Central Processing Unit and the Graphics processing unit are quite similar. This is due to the overhead of generating a large number of random numbers simultaneously on a Graphics Processing Unit device.

Chapter 5

Conclusions and Recommendations

This thesis has implemented two different versions of a Bloom Filter on a Nvidia based Graphics Processing Unit. By learning from the lessons of other papers submitted to IEEE on the subject manner, and by following the recommendations of papers and Nvidia on the subject of performance, this thesis has created a guideline to implementing an optimized Bloom Filter on Graphics Processing Units. The results of this thesis show that the Central Processing Unit implementation is much slower than the Graphics Processing unit for most input parameter combinations. The results of this thesis also compare an unoptimized implementation on the Graphics Processing Unit to an optimized version described in this thesis. The optimized version performs much better than the unoptimized version that experiences thread divergence and bank conflicts.

There is a lot of work that can be done to improve both the design and the results. Perhaps the biggest issue with this thesis is the random number generation for the Probabilistic Bloom Filter. The overhead spawned by the random number generators could possibly outweigh the benefits of using the Graphics Processing Unit to process the Probabilistic Bloom Filter. There needs to be a memory efficient way to generate a large number of random numbers simultaneously. Once a memory efficient way to generate random numbers on the Graphics Processing Unit is developed, then the execution time should become very close to the execution time experienced by the regular Bloom Filter.

List of References

[1] S. Tarkoma, C.E. Rothenberg, and E. Lagerspetz, “Theory and Practice of Bloom Filters for Distributed Systems,” IEEE Comm. Surveys and Tutorials, vol. 14, no. 1, pp. 131-155, first quarter 2012.

[2] Bloom, B. (1970). “Space/Time trade-offs in hash coding with allowable errors”, Communications of the ACM, vol 13, no. 7, pp 422-426

[3] A. Kirsch and M. MitzenMacher, “Less hashing, same peformance: building a better Bloom Filter,” in ESA’06: Proc. 14th Annual European Symposium on Algorithms. London, UK: Springer-Verlag, 2006, pp. 456-467

[4] Yao, Y., Sisi, X., Jilong, L., Qing, C., & Michael Berry, “Identifying frequent flows in large traffic sets through Probabilistic Bloom Filters.” (Unpublished doctoral dissertation).

[5] C. Nvidia, NVIDIA CUDA Programming Guide (2013) [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[6] C. Nvidia, NVIDIA CUDA Best Practices Guide (2013) [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[7] Marsaglia, G.; Zaman, A. (1991). "A new class of random number generators". *Annals of Applied Probability* **1** (3): 462–480. doi:10.1214/aoap/1177005878.

[8] L.Costa, S. Al-Kiswany, and M. Ripeanu,“GPU Support for Batch Oriented Workloads,” in IEEE 28th International Performance Computing and Communications Conference (IPCCC). IEEE, 2009, pp. 231–238.

[9] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin, “Bloom filter performance on graphics engines,” in Proc. of Int’l Conf. on Parallel Processing, 2011, pp. 522–531.

[10] L. Ma and R. D. Chamberlain, “A performance model for memory bandwidth constrained applications on graphics engines,” in Proc. of Int’l Conf. on Application-specific Systems, Architectures and Processors, 2012.

[11] L. Ma ,R. D. Chamberlain, and K. Agrawal “A memory access model for highly-threaded many-core architectures ” in Proc. Parallel and Distributed Systems (ICPADS) , 2012.

[12] Y. Kim and A. Shrivastava. “CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA.” in Proc. ACM Design Automation Conference (DAC), 2011, pp. 128-133.

[13] Q. Xie, T. Huang, Z. Zhou, Liang Xia, Y. Zhu, and J. Jiang. “An accurate power model for GPU processors”. in Computing and Convergence Technology (ICCCT), 2012 7th International Conference on, 2012, pp 1141-1146.

[14] NVIDIA. GT Specification. GT620 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gt640/specifications

[15] Al Mouhamed M., ul Hassan Khan A.. “Exploration of Automatic Optimization for CUDA Programming[C]”. in IEEE Conference Publications: Parallel Distributed and Grid Computing (PDGC). pp. 55-60, (2012)

[16] The OpenCL Specification 2.0, Khronos OpenCL Working Group.

[17] J. Fang, A. Varbanescu, and H. Sips, “A comprehensive performance comparison of cuda and opencl,” in Parallel Processing (ICPP), 2011 International Conference on, sept. 2011, pp. 216 –225.

[18] F. Gens, “IDC Predictions 2012: Competing for 2020”. IDC Home: The Premier Global Market Intelligence Firm. Available: http://cdn.idc.com/research/Predictions12/Main/downloads/IDCTOP10Predictions2012.pdf

[19] I. Paul. “The End Of Moore’s Law is on the Horizon,says AMD”. PCWorld. Available: http://www.pcworld.com/article/2032913/the-end-of-moores-law-is-on-the-horizon-says-amd.html

Vita

Joshua Pyle was born to Michael and Karen Pyle in Donelson, Tennessee. Joshua developed an interest in Computer Science while studying for a bachelor's degree in Electrical Engineering from Tennessee Technological University.