Research into GPU accelerated pattern matching for applications in computer security

November 4, 2009

Alexander Gee
age19@student.canterbury.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand

Supervisor: Ray Hunt
Ray.Hunt@canterbury.ac.nz

Abstract

Pattern matching is a fundamental part of many computer programs. Accelerating this process using Graphics Processing Units (GPUs) is thus greatly advantageous for many applications such as intrusion detection systems. This research investigates the implementation of pattern matching on these massively parallel processing units. The second area of research is implementation efficiency and performance advantages introduced by GPU acceleration of this family of algorithms.

Contents

1 Introduction

This paper discusses the utilisation of programmable pipeline Graphics Processing Units (here on referred to as GPUs) for high speed pattern matching. The first question that should be asked is why accelerate pattern matching? Pattern matching is a fundamental part of many computer programs widely used in databasing, search engines and most heavily in computer security. Just some of the major algorithms that use pattern matching include search algorithms, indexing algorithms, genetic engineering algorithms and signature matching algorithms used in computer firewalls and virus scanners. In all these applications pattern matching is responsible for a large percentage of the computational load even though it is mathematically and algorithmically fairly simple. Increasing the speed of pattern matching could therefore improve performance in many areas of computer science. The fundamental questions this paper addresses are can and should GPUs be utilised to improve pattern matching speeds? Being able to offload simple numerical processing to a specialised companion processor obviously is a great boon as this frees more resources on our general purpose processor for other important tasks. However if this offloading reduces performance this may not be desirable.

In the 1980's there began a trend to include Floating Point Units(FPUs) alongside a CPU, this changed the face of computer science. Suddenly it was possible to preform precision mathematics needed for simulations, computer aided design and even accounting in far shorter periods of time. This changed the way people saw and interacted with computers. During the last 5 years there has been seen a similar trend. The inclusion of massively parallel companion processors known as GPUs. GPUs have traditionally been used to render 3D environments for computer gaming and medical science. They have now been harnessed not only to produce photo-realistic virtual environments but as fully functional companion processors capable of complex logic and high throughput. The increase in processing power these devices deliver will likely produce a revolution similar to that of the FPU. Many scientific problems will suddenly become solvable and many more programs will leap into the realm of real time user interaction. In this paper there is a detailed examination of the technical challenges and opportunities these devices offer computer scientists and computer users through their applications in pattern matching. Because of the large volume of field specific terms it is highly recommended that before continuing you read the glossary of terms.

2 Glossary

GPU - Graphics Processing Unit.

GPGPU - General Purpose Graphics Processing Unit. This refers both to GPUs capable of executing general purpose code and the writing of such code.

SIMT - Single Instruction Multiple Thread. SIMT can be compared to the much more common SIMD. However with the extended ability to preform divergent branches.

CUDA - Compute Unified Device Architecture.

Grid - CUDA execution commands process a so called grid of data. A grid consists of a three dimensional array blocks which in turn consist of a three dimensional array of threads. This abstraction makes some geometric operations more intuitive. In string matching these higher dimensions can be ignored.

Warp - A Warp is blocks of threads that execute physically in parallel on a multiprocessor. This definition can be expanded on by the Nvidia Compute PTX ISA 1.2 manual[6] ”Individual threads composing a SIMT warp start together at the same program address . . . A warp executes one common instruction at a time, . . . If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.”

Host - The main computer holding a GPU.

Device - A GPU.

NIDS - Network Intrusion Detection System.

Kernel - A GPU program. Kernels are loaded onto the GPU from the host computer. A kernel is a block of code that will be executed by a group of threads in parallel.

Needle - A string to be found.

Haystack - A large string possibly containing one or more needles.

3 GPU basics

3.1 Traditional GPU programming pipeline

Before the development of specialised frameworks for general purpose graphics processing unit programming all general purpose applications that were to be accelerated on the GPU had to be mapped into a graphical context. These mappings where often over complicated and less than optimal. Several libraries exist which act as wrappers around traditional graphics oriented GPU frameworks to ease the process of developing GPGPU programs using traditional GPU hardware. A good example of this is the Sh shader language[14]. Sh provides an interface to OpenGL[16] that is optimised for the production of non graphical code. This however does not increase the fundamental capacities of OpenGL instead simply making it easier to use. Overheads introduced by less than optimal mappings of the problem domain have in the past often limited execution speed so much that utilising the GPU resulted in slower overall execution. Another limiting factor was the memory architecture of GPUs. In a traditional use nearly all memory transfers a GPU would see ran in one direction moving data from host to device. Often with general purpose code there are greater amounts of data moved in the opposite direction from the device to the host. As earlier GPUs were unoptimised for traffic in this direction they severely limited performance of many algorithms. It was clear that GPUs could be used to solve highly parallel problems but that the available APIs were limiting their usefulness. An in depth investigation into GPGPU is available in [11].

3.1.1 Gather

Gather is a term used to describe the ability to read any memory location from within a GPU program. The name is taken from the fact that such memory accesses are intended to gather data that is needed to preform a calculation. Gather has been a feature of all major GPU programming frameworks as it is relied upon heavily in the field of computer graphics.

3.1.2 Scatter

Scatter is the ability to write the results of a calculation to an arbitrary memory location. This has not been present in traditional GPU programming frameworks as the destination to be written to is always previously known in graphics processing. Without the ability to scatter data many algorithms are greatly limited in implementation and yet more completely unimplementable.

3.2 CUDA

Compute Unified Device Architecture is a programming framework developed by Nvidia that allows their 8000+ series of GPUs to be programmed directly inside standard C programs while utilising the same C syntax. CUDA allows nearly all the functionality available in C. Programs executed on CUDA devices have total freedom to gather and scatter data as needed. They may also perform calculations using any C operators along with pointer arithmetic and most other normal programming functionality.

This is accomplished with the use of several language extensions and a preprocessor. The preprocessor strips out the code intended for execution on the GPU and compiles it separately from the main body of the program. At runtime the code destined to be executed on the CUDA device is transferred into the instruction units of the device by way of a special operating system driver. CUDA thus provides an abstraction layer between the programmer and the GPU specific instructions sets. This allows the production of code that

is portable between various Nvidia graphics devices. Nvidia intends to keep the CUDA programming API over the next generations of its GPUs extending it when needed but maintaining backwards compatibility. It has already been proven that CUDA can offer extreme processing power in many problem domains. For example cryptographic hashing [17][12] , complex graphing [8] and DNA sequencing [13]. However little attention has been paid to GPU acceleration of string matching using CUDA only one paper previously being published [10].

3.3 CUDA Execution

CUDA executes code using a SIMT approach. This is similar to traditional SIMD execution in that the instruction load unit only has to load one instruction for a large number of elements to be processed. In the case of SIMD the elements are data words. In SIMT the elements are threads. This is to say that all threads must be executing the same instruction at the same time. Logically in purely mathematical operations all active threads should wish to execute the same instruction at the same time. However when conditional logic is implemented this is no longer true. Some threads many need to follow one execution path while others follow another. This is called thread divergence. To deal with this CUDA serialises operations. One code path will be executed before the other. In the case of the code in Figure 3.1 a divergent condition is met at line 3. Threads in which the data element at data[tid] is greater than ten will be executed first while threads in which the data element at data[tid] is ten or smaller will be placed in a stall condition. Once the active threads have executed line 4 the stalled threads will be woken and will execute line 6. All threads then wake and continue to execute in parallel. Threads being in a stalled condition waste processing power and reduce the performance of CUDA code. In string matching it is difficult to avoid divergent operations.

```
1 __global__ void divergeKernel(int* data){
2 int tid = (blockIdx.x *blockDim.x ) + threadIdx.x;
3 if(data[tid] > 10)
4 data[tid] = data[tid] + 1;
5 else
6 data[tid] = data[tid] - 1;
7 return
```

Figure 3.1: A kernel that produces thread divergence

3.4 CUDA memory layout

CUDA puts the programmer very close to the metal with regard to memory access. There is no operating system memory management and no memory swapping. A heap is managed using the functions cudaMalloc and cudaFree. Due to the limited stack system of CUDA these functions can only be called from the host computer not from code executing on the GPU. There are several different types of memory exposed, these are visualised in Figure 3.2 and detailed below.

- Registers - CUDA provides 8192 32bit registers for general purpose use. These registers must be divided up between all executing threads. As a CUDA thread device can have 768 threads in the execution state at once this gives us 10 registers per thread if we wish to utilise the device at maximum efficiency. In CUDA revision 1.2 the number of general purpose registers was raised to 16,384 and the maximum executing thread count was raised to 1024. Newer CUDA devices can therefore expend 16 registers per thread while still operating at peak efficiency.
- Global - this memory can be read to and written to from any thread. It is off-chip and contained in several ICs mounted on the GPU board. It is anywhere between 128MB and 4GB in size depending on the device in question.
- Texture - Texture memory is a wrapper around global memory. It is read only on the device but writeable from then host. Texture memory enables an 8KB cache and hardware interpolation on the block

of global memory it wraps. This cache is only effective when reading clustered blocks of memory. Interpolation is performed on dedicated hardware called Raster Operation Processors (ROPs). Due to this dedicated hardware there is little performance penalty involved in multi-sampling operations from texture memory.

- Shared - Shared memory can be read and written by all threads. It is 32KB in size and has a 8KB cache. Shared memory is intended to be a mechanism for message passing and semaphore. The CUDA compiler will reserve sections of this memory for internal use when required.

- Constant - Constant memory is a 64KB block of memory with an 8KB cache designed to hold operation constants. As the name implies constant memory is read only from the GPU context and accessible from all threads. Only the host may write into constant memory and only between kernel executions.

The most important thing to note is the lack of a cache system comparable to that of a recent x86 CPU. Modern desktop x86 processors have between 512KB and 12MB of fast on-chip cache. Because of this they can execute complex algorithms that deal with large numbers of items without being hampered by memory bandwidth. This is not the case for CUDA programs. Any accesses to CUDA global memory introduce a large latency and result in lower performance. Many programmers utilising CUDA have turned to using shared memory as a user managed cache in an attempt to mitigate this [15].

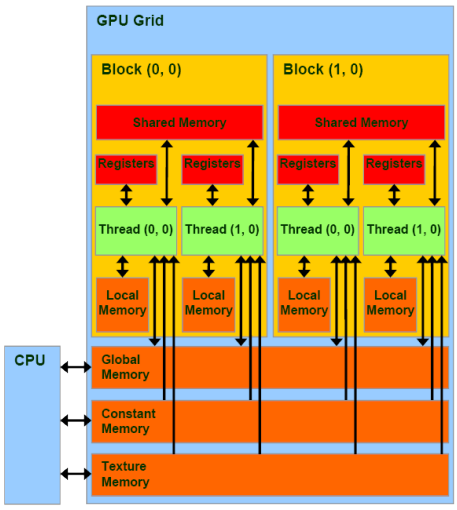


Figure 3.2: Block diagram of CUDA memory layout

The bandwidth from host to device memory imposes a hard performance cap on any solution that can be developed. Host to device speeds can range from 400MB/s to 2GB/s. These transfer rates are significantly faster than the throughput of storage devices and most networks links and as such should not be a limiting factor in tests. In the future it can be expected that the available bandwidth between CPU and GPU will advance faster than storage devices. As such it is unlikely that this will ever become a limiting factor for small systems. In large server architectures data throughput can reach multiple gigabytes per second. In this case multiple devices could be utilised in order to process this volume of traffic.

3.5 The CUDA stack

CUDA has a limited stack architecture which is due to the need to know the stack size at compilation time for memory and register allocation purposes. CUDA unrolls function calls in a manner similar to loop unrolling. This results in a CUDA kernel which encompasses the complete possible map of execution paths. Because of this, recursive algorithms cannot be implemented using function calls as is general practise. This can effect some areas of string matching such as regular expression matching.

3.6 The CUDA heap

The heap implemented in CUDA is limited in a similar manner as the stack. As memory usage must be known by the CUDA driver before a kernel can be executed. Dynamic memory allocation in the GPU memory space is only possible on the host. Allowing kernels to allocate memory from within the GPU would mean that the CUDA driver would not be able to correctly determine positions to place the stack. This means that we must be able to determine the memory usage of any string matching solution developed to run on CUDA before the actual CUDA execution begins.

3.7 Kernel launch overhead

CUDA suffers from overheads involved in copying data and program instructions from system to device memory. All CUDA enabled GPUs are connected to their host system by the PCI express bus. Because of this architecture memory copies between the host and device memory incur a latency overhead. This latency makes small memory transfers overly costly as they significantly degrade the available bandwidth. Bus latency also makes CUDA processing of very small datasets not worthwhile. It became important to find out how much launch overhead was involved and how it scaled as this could potentially cripple CUDA for security applications. If kernel launches occupy a large percentage of the available host processor or GPU time, low latency stream processing such as that required for firewall applications could be hindered. To test the impact of launch overhead a program was produced that executed many, short, busy work kernels. The kernel is shown in Figure 3.3 This kernel accepts a pointer to an integer and a number of additions to preform. This way the the same number of operations can be performed using different numbers of kernel launches. Since CUDA 1.1 there has been support for CUDA atomic operations. These make sure that threads do not enter a race condition while accessing shared data elements, see [5] for more details. The test kernel uses the CUDA atomic operation atomicAdd to ensure that all additions happen without any race conditions.

```
__global__ void addOne(int* data, char numToSum){

    for(int i=0; i < numToSum; i++)
        atomicAdd(data,1);
    return;
}
```

Figure 3.3: Launch overhead test code

Figure 3.4 shows that with each ten fold increase in kernel launches there is a significant but not prohibitive jump in execution time. Minimising the number of launches involved in a matching operation will increase throughput. Unfortunately watchdog timers whose function is to kill unresponsive graphics kernels, are built into all major operating systems. In Windows XP and Windows Vista CUDA kernels are allowed 5 seconds execution time while Windows 7 reduces this execution window to 2 seconds. As a kernel that takes 1 second to execute on a powerful GPU may take several times longer on a less powerful GPU a large margin must be left in order to ensure successful execution is possible on a majority of hardware configurations. Kernel execution time limits can be avoided by installing a GPU dedicated to run CUDA programs. At the current time it is still very uncommon in consumer machines to have a GPU dedicated to executing CUDA programs. This is not an issue for code intended to run on servers in which specific hardware considerations can be required.

3.8 GPU explicit matching vs GPU filtering

There are two possible approaches to performing string matching on GPUs. Either moving the entire process of string matching onto the GPU leaving the CPU completely free or using the GPU as a filtering device to produce a subset of the haystack which we must then search on the CPU. The latter approach provides two key benefits. Firstly there is no need to copy the full text of the needles that are to be searched for onto the device thereby consuming bandwidth and increasing device memory requirements. Secondly the size of the device kernel involved in filtering operations should be significantly lower leading to higher performance. Using the GPU to preform explicit matching introduces some problems. In mission critical operations the lack of ECC bit error correction may not be acceptable. It is possible that calculations preformed on the GPU hardware could be adversely affected by outside sources of information such as radiation sources. In the case of explicit matching the GPU may confirm a match that does not truly exist. In the case of filtering it is possible for the GPU to miss a match but it is not possible to cause a false detection as a CPU check verifies each match.

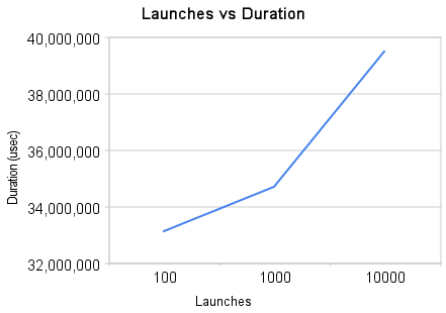


Figure 3.4: Graph of number of launches versus total duration

4 String matching work division

The problem of string matching can be split into parallel segments in several different ways each, having specific advantages and disadvantages.

4.1 Thread per needle

The most obvious way to distribute the workload is to match one needle on each processor against the entire haystack. This has the advantage of being easily extended to work with data streams. A data stream can be buffered into a memory block, then when a set number of bytes have been collected processed, at once by the matching engine. The buffer can be small thus introducing little latency. Unfortunately a thread per needle approach introduces much redundant calculation. This is brought about by the fact that if matching a haystack against two needles ”form” and ”formidable” it is logically obvious that if a position does not contain the first needle it cannot contain the second. Since each needle is being matched in a separate logical thread this information cannot be shared in an efficient manner. Prefix tables could be constructed to help handle this problem.

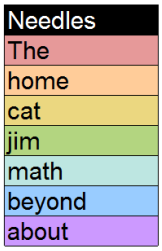


Figure 4.1: A separate thread per needle.

Because all threads iterate over the haystack separately if one thread gains a lead over another memory access patterns will become very poor. One thread may be reading a completely separate part of the haystack to another thread in the CUDA grid. Because of the limited caching mechanisms available most bytes from the haystack will generate a cache miss and have to be read from memory. The cycles lost while waiting for memory operations will significantly impact performance. It is also important to note that some threads may have significantly higher execution times than their peers. If a thread is processing a needle that consists of characters that occur frequently in the haystack it will tend to take more cycles to process and thus finish later. This means the peer threads must wait in a stalled condition for the slower threads to finish executing. This results in a poor utilisation of the available resources.

4.2 Haystack blocking

Figure 4.2: Haystack cut into large contiguous blocks.

A second method of distributing the work load is to split the haystack into blocks and use existing string matching systems such as finite state machines. However to be efficient the haystack must be large. There is little point in parallel workload distribution if the haystack is only a few hundred bytes in size. This means that data needs to be buffered into larger blocks before being processed. Filling these buffers thus introduces some latency and makes this method less suited to streaming data.

In practise blocks must overlap. When dividing a data stream into blocks for matching J-1 (where

J is the length of the longest needle) bytes at the trailing end of the block cannot be fully checked as if a signature exists beginning in one of these positions the matching algorithm will run out of haystack before the match is confirmed. If the size of a block passed to the matching algorithm is small then a higher percentage of the data will need to be passed to the algorithm in the next block. This creates a trade off between solution efficiency and solution latency. As the size of the data blocks is increased the efficiency rises. However additional latency is added as the block requires more time to be filled before being processed.

4.3 Interlaced blocking

Figure 4.3: Haystack interlaced processing.

Block interlacing is a specialisation of the block method as used in [1]. This increases the level of thread cooperation and when tuned can produce more efficient memory access patterns. Unfortunately this means that all threads must complete before a definitive match can be produced. If these threads exist in the same block then this overhead will be hidden as all threads in a CUDA block finish execution simultaneously. Unfortunately the logic required to share relevant data between threads along with the increased frequency of memory accesses associated are detrimental to performance. The logic required to implement an interlaced block approach is difficult to understand and large. This makes interlaced blocks a poor candidate for high performance GPU execution.

4.4 Thread per position

Another approach to dividing the string matching job into parallel sections is having many threads all executing identical matching algorithms but each processing a different byte offset in the haystack. One advantage of this technique is that threads that execute at the same time will read overlapping portions of the haystack allowing cache mechanisms to operate effectively. This approach however disallows the use of a rolling hash function [3], in hash based search algorithms as each thread operates at only one start position and does not iterate through the haystack. As each thread starts it can check to see if a thread with a smaller offset has found a match. In the case that this is true the thread can be terminated immediately thus avoiding wasteful processing of the haystack at positions after the match.

5 Overview of string matching algorithms

The major families of string matching algorithms along with specific details of their implementations which require special consideration for GPGPU implementation are now discussed.

5.1 Naïve search

pre-calculation time	matching time
no preprocessing	$\Theta((n-m+1)m)$

The naive search is known by many names such as ”brute force string matching” and ”exhaustive matching”. This method scales in a linear manner with haystack length and produces a worst case matching time of n*m. Since there is no need for pre-computation this algorithm is often used when a result of a less precise method needs to be verified. Naïve matching is algorithmically very simple. Because of the algorithmic simplicity resources used per thread will be small leading to efficient utilisation of CUDA hardware.

5.2 Finite state machines

pre-calculation time	matching time
$\Theta(m\|\Sigma\|)$	$\Theta(n)$

Finite state machines operate by moving between a set of states in a highly connected graph that represents the needle space. CUDA has a limited stack and heap system. This is because during kernel executions blocks of global (heap) memory are used to store local variables. Implementing finite state machines is thus somewhat complicated. There are two options for building a state graph for traversal on GPU. The first option is to create a state machine using GPU executed code utilising a user managed heap stored in a preallocated block of memory. The second option is to build the state machine on the host and then copy this memory structure into GPU global memory. The second option leads to smaller simpler kernels. To simplify the operation of moving a finite state machine on to the GPU C structs, which are supported by the CUDA framework, can be used to model the nodes of the graph structure. An in depth investigation into parallel, finite state machine, string matching is presented in [1].

5.3 Boyer-Moore

pre-calculation time	matching time
$\Theta(m+\|\Sigma\|)$	$O(n)$

The Boyer-Moore algorithm is the de-facto standard for string matching. The algorithm first constructs lookup tables for the needles it is required to find. These tables store the optimal jumps that can be performed when a mismatch occurs in the matching process. However, when implemented using the CUDA API the Boyer-Moore algorithm suffers from some problems. The large number of memory lookups and highly branching logic will impact performance. The lack of efficient thread co-operation causes large

amounts of data to be checked more than once or checked superfluously. These redundant operations scale with the number of needles in a linear manner.

5.4 Rabin-Karp

pre-calculation time	matching time
$\Theta(m)$	$\Theta(n+m)$

The Rabin-Karp algorithm is a hash based string match first proposed in [9]. Hashing reduces the amount of logical operations that must be performed at the expense of increasing the number of purely numerical operations. This is a benefit to implementation on GPU as branch divergence can be avoided. The hash used in Rabin-Karp algorithm is of a special variety called a rolling hash [3]. The hash of each needle is computed only once and then stored. The haystack is examined and hashed one byte at a time. This hash once computed is compared against the needle hashes that were determined in the pre-calculation stage. If a needle is present in the text the matching hash will be found. On occasion incorrect hash collisions can be experienced as two different strings may produce the same hashed value. For this reason when a match is detected a brute force confirmation match is then calculated to assure a correct determination. Incorrect hash collisions are very rare and as such the overhead of exhaustive matches being computed superfluously is minimal. Hash based matching appears the most promising for extension onto GPU.

5.5 Bloom filter

pre-calculation time	matching time
$\Theta(m)$	$\Theta(n+m)$

Bloom filters or "k-fold bitstate hashing" are a special application of hashing first proposed in [4]. Bloom filters are used to represent a very large number of possible states, using a much smaller amount of memory, at the cost of potential false positives due to hash collisions. For the problem of string matching false positives are acceptable as a second test can be preformed on positive results. As such the Bloom filter is used to reduce the number of positions in the haystack that have to be exhaustively tested. This was implemented in appendix A for evaluation.

Details of the implementation are now discussed. Before scanning of the haystack commences a hash of each needle that is to be found must be calculated. An array of bits is allocated and bits at the positions produced by hashing the needles are set true. This is a simple Bloom filter utilising a single hash function (k=1). Multiple hash functions can be used to set multiple bits for each needle if false collisions prove frequent. This would however require more instructions to be preformed per byte of data. Because GPUs have large amounts of memory available we can avoid the cost of computing multiple hashes by using a large hash table consisting of between 128MB and 4GB of memory. For the calculations that follow the Bloom filter is assumed to use 134217757 bits. If it is assumed that the output from the hash function that is chosen and the strings that are to be hashed are distributed randomly it follows that the probability of a hash collision is calculated using the standard equation for calculating the probability of a false positive in a Bloom filter:

$q = 1 - (1 - (1/m))^{nk}$

- q - is the probability that a position in the Bloom filter is set true (a hash collision).
- n - is the number of elements that have been added to the Bloom filter.
- m - is the number of bits used to implement a Bloom filter.
- k - is the number of hash functions that as stated before will be set to 1.

If there are 512 needles placed in the Bloom filter then:
$q = 1 - (1 - (1/134217757))^{512} = 0.00000381468917966118088732668137$

If there are 51,200 needles placed in the filter then:
$q = 1 - (1 - (1/134217757))^{51200} = 0.000381396895266373629944624514 92$

This means in practise that large sections of the text can be eliminated as candidate positions. Under perfect conditions, with 512 needles in the Bloom filter exhaustive tests need to be preformed on less than 0.00038% of the possible positions in the haystack. False positive rates will increase as the number of needles that are to be tested for are increased. The impact on performance is small for applications with moderate numbers of needles, for example virus databases and NIDS packet signature databases. In real world conditions higher levels of hash collisions can be expected as many datasets dealt with contain limited alphabets or uneven character distributions. These conditions will affect the ability of the hash function to map the data into a uniformly distributed space. Due to the implementation of the Boolean type in C being stored in a byte instead of a single bit the space needed to store a Bloom filter of 100bits in CUDA is in fact 100bytes. This could be avoided by using a C struct with custom defined offsets in order to pack eight Booleans into a single byte. Bit-bashing approaches often lower performance due to the overhead involved in unpackaging the individual bits. As memory is plentiful on modern GPUs bit-bashing should therefore be avoided in CUDA Bloom filter implementations.

5.6 Precision considerations

GPUs have traditionally used 32bit words in their operations. While the range of 0 to 4,294,967,296 afforded by 32bit words is more than sufficient for most applications the intermediate stages of multiplicative hash calculation can require higher precision. The obvious solution is to use a type with a larger range, such as a long int, however the performance penalty for double precision calculations on CUDA hardware is high. Double precision operations on current generation CUDA devices take approximately five times longer than single precision operations. This limits the size of the hash table that can be worked with using a multiplicative hash. More complex hashes based on bitshifts and random number tables are possible but not covered in this paper.

6 Computer security

Computer security is a fairly recent field. With the widespread adoption of Internet communications it has become necessary to ensure the security of computers from external tampering. This is a largely automated task generally involving scanning the computer against a database of known threats. A more proactive approach can be taken in scanning network traffic during transit for threats. There are therefore two quite separate problems under the name of computer security so called on-line security and off-line security. In the case of on-line security, for example firewall applications, it is important to minimise the amount of latency added to the system by security processing. On-line security can be further broken down into security that is enforced at a personal computer and security that is enforced at a network boundary. Processing the traffic of an entire network requires a large amount of processing power making it an ideal candidate for GPU acceleration. Off-line security also has areas that could benefit from GPU acceleration most notably personal virus scanning. Virus scanning has long been the bane of personal computer users as it can easily bring all but the most powerful systems to a shuddering halt. Most of the computational workload of computer security applications is simple string matching and therefore stands as a good candidate for GPU acceleration. If it is possible to replace dedicated hardware solutions such as those used for network intrusion detection with inexpensive consumer equipment large quantities of money could be saved. If GPU acceleration can be applied to virus scanning on personal computers solutions can be delivered that impact computer users much less than their traditional counterparts. For these reasons utilising GPUs in pattern matching applications could be very beneficial.

6.1 Security data matching specifics

Data matching in computer security is broken down into three major categories. Blacklist matching, hash matching and regular expression matching.

6.1.1 Blacklist matching

This is using string matching to detect keywords or byte patterns that should trigger some kind of security event. Events can be, for example rejecting a packet, alerting an administrator or running a decontamination procedure on a file. Blacklist matching is an optimal candidate for the use of GPU accelerated pattern matching. Black list matching is used primarily in network intrusion detection systems, data loss prevention systems, firewalling and many forms of virus scanning.

6.1.2 Hash matching

Many virus scanners employ hash matching. This is the process of creating a hash of each file on the system and comparing the result with a list of known bad files. Because the entire file is considered this is not a granular approach. A virus that attaches itself onto another file will not be found by this matching method. In most cases a standard cryptographic hash such as MD5 will be utilised for this matching application. These hash functions can be accelerated by GPU utilisation as shown in [17] but are outside the scope of this paper.

6.1.3 Regular expression matching

Regular expressions are a system to allow flexible string queries of a document. Regular expressions consist of strings of characters connected by wildcards. These wildcard positions may match various characters.

Regular expressions are typically used to match short blocks of characters. Having short needles to match reduces the efficiency of hash based string matching as collisions become more likely. Due to this regular expression matching is not a good candidate for GPU acceleration.

7 Performance evaluation

To evaluate the key factors CUDA acceleration of pattern matching CUDA implementations of the naïve search algorithm, Boyer-Moore algorithm and a novel Bloom filter algorithm were produced. Performance tests were conducted on the matching solutions. During these tests several different machines with various GPUs were used to determine the scalability of each solution. Due to the high throughput of these solutions files could not be read from harddisk as needed because the harddisk quickly became a performance bottleneck. To overcome this the test programs read a complete file into system memory before starting the process. The time taken to read the file into memory is not factored into the reported performance. Figure 7.1 shows the key attributes of the GPUs used in testing. A tabulated view of this information is available in appendix B.

7.1 Needles

Needles used in testing are taken from the project Gutenberg edition of the King James Bible. The project Gutenberg Bible is 4.97MB in size and contains 5,213,926 characters. The needle strings were randomly selected by a C# program, the source for which is published in Appendix C. The methodology for choosing the needles was simple. For each needle required a random offset position is generated. The program then navigates to the offset position in a byte stream of the project Gutenberg Bible. A second random integer L is generated where L is of set $\mathbb{R}$ with a range of $512 \leq L \geq 32$. This is taken to be the length of the needle. The L bytes from the offset are read and then stored into the signatures file. The alphabet size of the project Gutenberg Bible was calculated using the tool in Appendix D. In total 89 different characters are used with frequencies roughly as to be expected for English text [2].

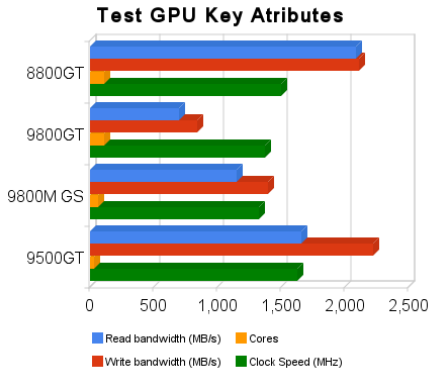


Figure 7.1: Key attributes of test hardware

7.2 Haystack

For testing the implementations a haystack was chosen that did not contain any of the needles selected. The file used was a compressed video stream 740MB in size. This file has an alphabet of 256 characters, the maximum number possible. A full listing of the characters that appear, along with their frequencies, was calculated using the tool provided in Appendix D is available in Appendix E. The haystack was chosen to produce optimal performance in preliminary tests.

7.3 Bloom filter implementation

A Bloom filter implementation proved to be the matching algorithm with the highest throughput; speeds of between 224Mbit/s and 1000Mbit/s were produced during testing. These figures represent a speedup of more than 20 times compared to the same solution being run on CPU as shown in Figure 7.2. This method of matching uses the GPU for parts of the matching operation to which it is best suited, generating hashes of the data at each position and some simple checking. The CPU is used to confirm matches that are detected

on the GPU. This allows the amount of logic implemented in the GPU kernel to be somewhat smaller. This in turn means that the GPU can be utilised in an efficient manner. With more complex algorithms such as the Boyer-Moore algorithm the number of registers available for local variables is exhausted and the choice must be made to either use much slower global memory to store our variables or under utilise the GPU processors to allow enough space for the larger kernel. Using a Bloom filter approach also greatly reduces the number of accesses to main memory. The Bloom filter implementation when scanning large binary files can often process 512KB without the need for any exhaustive tests to be performed. This results in a low CPU utilisation and high throughput. For testing a Bloom filter size of roughly 128MB was chosen. The key reason being making the solution fit on any foreseeable test device. This was judged to be the best compromise between minimising the number of hash collisions and device compatibility. The performance of the Bloom filter solution during testing is shown in Figure 7.2. A tabulated data for these tests is provided in Appendix G.

7.3.1 Effect of cores

The factor that influences the performance of the Bloom filter solution most is the number of CUDA cores available. Each core can process one thread at any one time. This means as we increase the number of cores we increase the number of threads that can be executing at one time. An increase in available cores provides a linear near one to one mapping with each extra core (clocked at 1.5GHz) comes approximately 1MB/s extra performance.

7.3.2 Effect of clock speed

Cores that are clocked faster can execute more instructions per second. This boosts the performance of the matching algorithms. One core clocked operating at 2GHz provides the same throughput as two cores operating at 1GHz. All current CUDA enabled GPUs use clock speeds between 1.242GHz and 1.5Ghz. Utilising Nvidia's performance software it is possible to override the factory set clock speed of CUDA enabled cards. Figure 7.3 shows the throughput of the Bloom filter matching implementation running on the 9800GT GPU at various clock speeds. This graph shows that the relationship between clock speed and throughput in this case appears to be linear.

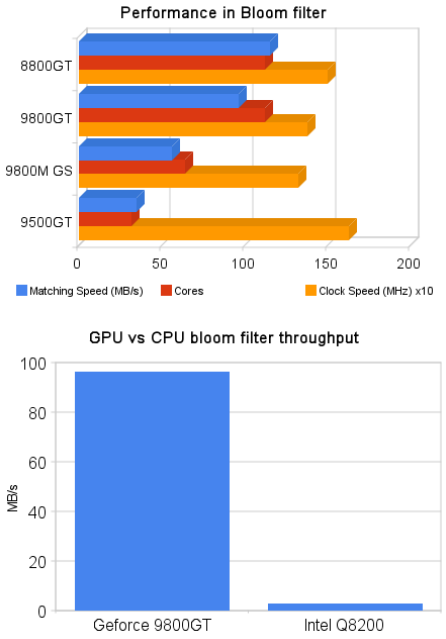
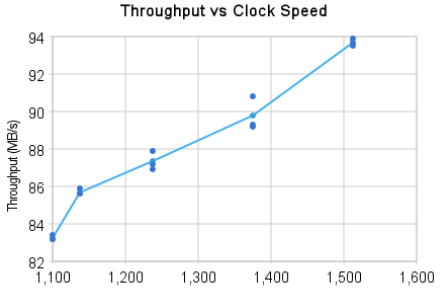


Figure 7.2: Bloom filter performance.

7.3.3 Scaling up

The test results show that on average each core clocked 1MHz can process 700Bytes of the haystack per second. While available test GPUs managed at best 115MB/s, using the data gathered from the testing it is possible to estimate the performance of higher performance CUDA devices. At the time of writing the most powerful GPU produced by Nvidia is the GeForce GTX 295. This has 480 Cores that run at a clock speed of 1.242GHz. The performance can be estimated by the equation: $Bytes/s = Cores * Clockspeed(MHz) * 700$

This leads to an estimated performance of 398MB/s for the GeForce GTX 295. This is significantly higher that the bandwidth available on a 1Gbit/s network link. It is possible to place four of these cards in one computer for a combined throughput of 1592MB/s which is significantly higher than the bandwidth of a 10Gbit/s network link. Nvidia produces a line of "desktop supercomputers"



under the Tesla brand name. These contain four dedicated CUDA GPUs and are used primarily as a replacement for existing supercomputers. One of these machines could, using this solution potentially perform all NIDS operations for a large corporate internet connection.

7.3.4 Performance degradation

Several factors can influence the performance of this implementation. The foremost factor being limited alphabet size. This will cause many more collisions than a larger alphabet and thus more false candidate positions. As such searching in plain text documents is less efficient than searching in binary documents.

7.4 Brute force

A brute force matching algorithm as is shown in Appendix F. This proved to be very slow at matching even when executed on GPU. The maximum speed achieved during testing was 0.884636MB/s which makes this solution unsuitable for any real world application. It does however produce some important data about string matching on CUDA.

It can be demonstrated that this algorithm is highly memory bound by calculating the number of cycles required to process each bit of the haystack at different clock speeds. To do this the following equation is used.

$$c = (1000000/((m/p)/s))/8$$

- c - is the number of processor cycles needed per bit of the haystack.
- m - is the number of Megabytes per second produced in a test.
- p - is the number processors(cores) on the test device.
- s - is the clock speed of the processors in Mhz.

When the brute force solution is executed on a Geforce 9800GT at 1375MHz the throughput achieved is 0.884636MB/s thus the number of cycles per bit can be shown to be:

$$c = (1000000/((0.884636/112)/1375))/8 = 20752$$

When the brute force solution is executed on a Geforce 9800GT at 1100MHz the throughput achieved is 0.871492MB/s thus the number of cycles per bit can be shown to be:

$$c = (1000000/((0.871492/112)/1100))/8 = 16852$$

While in the second case the GPU is operating at a lower speed and producing a lower throughput it is in fact nearly one quarter more efficient. This shows that a large amount of time is being lost as the GPU waits for data to be retrieved from main memory. When the brute force algorithm is executed on a Geforce 9800M GS and compared to the results from execution on the Geforce 9800GT from above the memory dependency becomes even more apparent. As shown in Figure 7.1 The Geforce 9800M GS has nearly twice the memory throughput of the Geforce 9800GT. The Geforce 9800M GS produced a matching throughput of 0.794MB/s it can therefore be shown that the number of cycles per bit is:

$$c = (1000000/((0.794/64)/1325))/8 = 12732$$

Using the data produced by the brute force solution it can be concluded that any solution that wishes to efficiently utilize a GPU for string matching must avoid memory throughput dependency.

7.5 Boyer-Moore

The Boyer-Moore kernel implemented uses the thread per needle approach of work division. Essentially the process consists of running one Boyer-Moore matching algorithm for each needle. This solution produces a speed of around 1.5MB/s. While this throughput is not high it is significantly better than that of the brute force solution however, this algorithm suffers some memory dependency, as shown below utilising the equations presented in the previous section. When the brute force solution is executed on a Geforce 9800GT at 1375MHz the throughput achieved is 1.562143MB/s thus the number of cycles per bit can be shown to be:

$$c = (1000000/((1.562143/112)/1375))/8 = 11752$$

When the brute force solution is executed on a Geforce 9800GT at 1100MHz the throughput achieved is 1.382687MB/s thus the number of cycles per bit can be shown to be:

$$c = (1000000/((1.382687/112)/1100))/8 = 10622$$

The ability of the algorithm to skip over bits where matches cannot occur increases the performance with regard to the naïve solution shown earlier. Two main factors limit the performance of this algorithm. Firstly in cases where there are more processors available than there are needles we will leave part of the CUDA un-utilised. Secondly the intrinsically branching nature of the logic involved in the Boyer-Moore algorithm. The Boyer-Moore kernel produced 1.36968e+07 serialisations per 64KB block of haystack compared to 9.09543e+06 serialisations per 64KB block of haystack for the naïve solution. This means that a large amount of time was spent with threads in a stalled state. Because of this the throughput was significantly lower than the Bloom filter implementation.

8 Conclusions

Using the data presented in this paper it can be concluded that GPU acceleration of pattern matching can greatly increase the throughput of standard computers with respect to virus scanning and network intrusion detection. With the inclusion of a powerful graphics processing unit a desktop PC is quite capable of processing a 1000Mbit/s network link. This should provide significant financial savings over the use of complex dedicated systems. The maintainability and expandability of such a system should also be much greater than that of dedicated hardware. The utilisation of a CUDA enabled GPU leaves the CPU of the host system free for other activities. GPU acceleration could be applied to desktop anti-virus solutions to reduce the performance impact of background scanning operations. When used in a NDIS or firewall situation the offloading of matching work to the GPU could allow other services to be hosted on the same computer thus reducing overall cost further. This research also shows that a further use of GPU string matching may be data ranking algorithms for large search engines and databases. Due to the large amounts of memory and processing power available using GPGPU devices it is possible to analyse very large amounts of data much faster than has previously been possible on consumer hardware. The massive growth in GPU power can be expected to continue into the near future providing greater opportunities for utilisation in this field.

8.1 Future work

A lot more time can be spent tuning algorithms and increasing the performance of string matching on CUDA. Examples include working on more effective memory access patterns and caching mechanisms. Extending the solutions this paper has described to use real streaming packet data and performing a second evaluation would help to further prove the conclusions of this paper. In October of 2009 Nvidia released the specifications of their next generation GPU. This system is called Fermi[7]. Fermi improves on a lot of points that have previously hampered CUDA applications, for example dissimilar kernels can now be scheduled to execute in parallel which allows greater flexibility in application data flow. Along with this cache systems have been radically improved by introducing a large level 2 cache which should significantly increase pattern matching performance. However possibly the most important advance is full C++ support including dynamic memory allocation. More research is required to learn how to best leverage these exciting new architectural improvements to improve the performance of GPU pattern matching.

Bibliography

[1] A Multi-Gb/s Parallel String Matching Engine for Intrusion Detection Systems. *Communications in Computer and Information Science Volume 6*, pages 847–851. Springer Berlin Heidelberg, first edition, 2008.

[2] Todd Feil Abraham Sinkov. *Elementary Cryptanalysis*, pages 16–22. The Mathematical Association of America, 1998. ISBN 0883856220.

[3] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201612445.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7): 422–426, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692.

[5] Nvidia Corporation, 2008. NVIDIA CUDA Compute Unified Device Architecture Reference Manual Version 2.1.

[6] Nvidia Corporation, 2008. Nvidia Compute PTX ISA 1.2 page 9.

[7] Nvidia Corporation, 2009. NVIDIA Fermi Architecture Whitepaper.

[8] P.J. Harish, P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. *LECTURE NOTES IN COMPUTER SCIENCE*, 1(4873):197–208, 2007. ISSN 0302-9743.

[9] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. ISSN 0018-8646.

[10] Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis. String matching on a multi-core gpu using cuda. *Informatics, Panhellenic Conference on*, 0:14–18, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/PCI.2009.47.

[11] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1103900.1103933.

[12] S.A. Manavski. Accelerating large graph algorithms on the gpu using cuda. *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference*, pages 65–68, 2007.

[13] Svetlin Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10+, 2008. ISSN 1471-2105. doi: 10.1186/1471-2105-9-S2-S10. URL `http://dx.doi.org/10.1186/1471-2105-9-S2-S10`.

[14] Stefanus Du Toit Michael D. McCool. *Metaprogramming GPUs with Sh*. AK Peters, 2004. ISBN 1568812299.

[15] Yoshinori Aono Mikael Onsjo, 2009. Online Approximate String Matching with CUDA.

[16] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201604582.

[17] Andrew Zonenberg. Distributed hash cracker: A cross-platform gpu-accelerated password recovery system. Rensselaer Polytechnic Institute, April 2009.

A GPU and CPU bloom filter

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <vector>

#include <cutil_inline.h>

#define BYTES        65536
#define NOTFOUND 100000
#define PRIME_BASE 31
#define PRIME_MOD 134217757
#define MAXSIGLENGTH 32768
#define MAXSIGS            32768

using namespace std;

bool differentLengths[MAXSIGS] = {false};
__constant__ char d_cand[BYTES];


__host__ int hhash(char* s, int sSize)
{
        unsigned int ret = 0;


        for (int i = 0; i < sSize; i++)
        {
                ret = ret*PRIME_BASE + s[i];
                ret %= PRIME_MOD; //don't overflow
        }
        return ret;
}

void hosthashAndCheck( int* found, char* candidate, bool* hashTable){
        for(int i =0 ;i<BYTES-12 ; i++){
                if(hashTable[hhash(&candidate[i],12)]){
                                found[0] = i;
                                break;
                }
        }
```

```
}



__global__ void
hashAndCheck( int* found, char* candidate, bool* hashTable)
{
        int tid = (blockIdx.x *blockDim.x ) + threadIdx.x;


        //HASH##########
        unsigned int ret =0;


        for (int i = 0; i < 12;i++)
        {
                ret = ret*PRIME_BASE + d_cand[ tid + i];
                ret %= PRIME_MOD; //don't overflow

        }

        //##############

        if(hashTable[ret])
                atomicMin(found,tid);


}


struct signature{  //Simple structure to store a string with a length
        char* data;
        short length;
};


int confirm(int position,char* line,struct signature * sigs,int
    numSigs){

        for(int j = 0; j<numSigs ; j++){
                int foundC = 0;
                while( foundC < sigs[j].length && sigs[j].data[foundC]
                    == line[position+foundC])
                        foundC++;
                if(foundC == sigs[j].length)
                        return j;
        }

        return -1;
}

//////////////////////
// Program entry point
//////////////////////
```

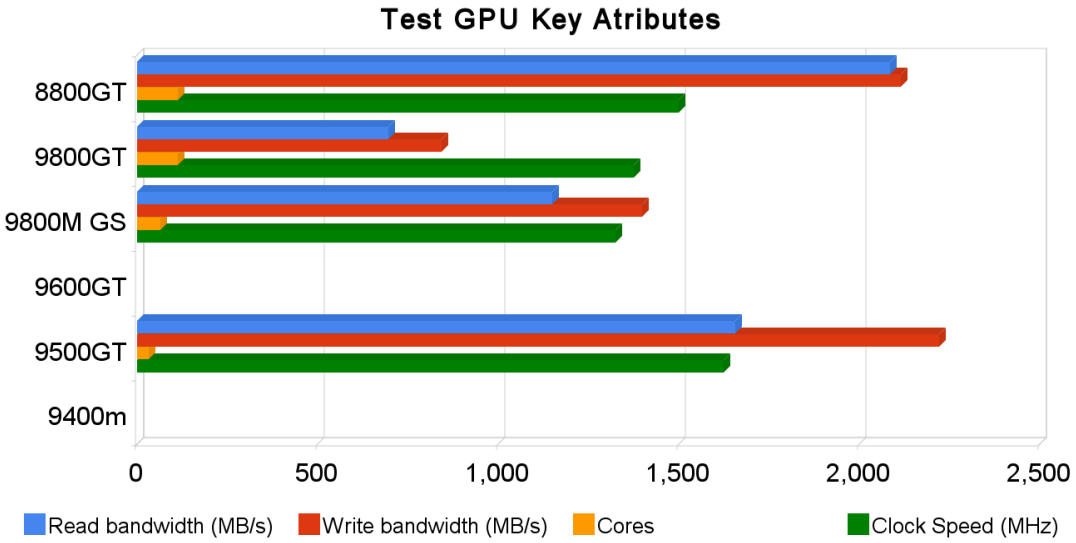
```
int
main( int argc, char** argv)
{
        dim3 threads(512, 1, 1);
        dim3 grid( 128 , 1, 1);

        bool* h_hashTable;

        h_hashTable = (bool*)malloc(PRIME_MOD * sizeof(bool));
        memset(h_hashTable,0,PRIME_MOD * sizeof(bool));


        char* line = (char*) malloc( sizeof(char) * MAXSIGLENGTH);
        int numLoaded=0;
        ifstream myfile("c:\\signatures.txt",ios::binary);


        struct signature *signatures = (struct signature*) malloc(
           MAXSIGS * sizeof(struct signature));

        int hashCollisions =0;
        //######## LOAD THE SIGS ########
        if (myfile.is_open())
        {
                int position =0;
                while (myfile.good() && numLoaded < MAXSIGS)
                {
                        if(myfile.read(line,2).eof())
                                break;

                        short lengthToRead = (((unsigned char)line[1])
                            << 8) + ((unsigned char)line[0]);

                        myfile.read(line,lengthToRead );

                        signatures[numLoaded].length = lengthToRead;
                        signatures[numLoaded].data = (char*)malloc(
                            sizeof(char)* lengthToRead );

                        memcpy(signatures[numLoaded].data,line,
                            lengthToRead);
                        differentLengths[lengthToRead] = true;
                        numLoaded++;
                }
                myfile.close();
        }else{
                printf("No Signatures File\n");
                exit(0);
        }



        short* h_lengthTable = (short*)malloc(sizeof(short)*MAXSIGS);
        int realLengths = 0;
```

```
//###### PRODUCE SIG LENGTHS ARRAY #####


for(int i=0;i<MAXSIGS;i++){
        if(differentLengths[i])
                h_lengthTable[realLengths++]=i;
}


//###### PUT HASHES IN TABLE ##########
for(int i =0 ; i<numLoaded ; i++){

        int hash =  hhash(signatures[i].data,12);
        if(h_hashTable[hash])
                hashCollisions++;
        h_hashTable[hash] = true;
}
//#################

printf("Hash Collisions: %d\n",hashCollisions);

//CPU
//char* candidate = (char*)malloc(BYTES);

//GPU
char* candidate;

int size = 0;
FILE *f = fopen("c:\\input.txt", "rb");

if (f == NULL)
{
        printf("Input File Not Found\n");
        exit(1);
}

fseek(f,0,SEEK_END);
int     fileLength = ftell(f);

fseek(f,0,SEEK_SET);
char* fileBuffer = (char*)malloc(fileLength*sizeof(char));

if(fread(fileBuffer,1,fileLength,f)!=fileLength)
{
        printf("Input File Load Error\n");
        exit(1);
}

printf("File Loaded\n");

int currentOffset =0;


cudaSetDevice( cutGetMaxGflopsDeviceId() );
```

```
unsigned int timer = 0;
cutilCheckError( cutCreateTimer( &timer));
cutilCheckError( cutStartTimer( timer));


size = BYTES;

bool* d_hashTable;
cutilSafeCall( cudaMalloc( (void**) &d_hashTable, PRIME_MOD *
    sizeof(bool)));
cutilSafeCall( cudaMemcpy( d_hashTable, h_hashTable, PRIME_MOD
     * sizeof(bool), cudaMemcpyHostToDevice) );

int* d_found;
cutilSafeCall( cudaMalloc( (void**) &d_found, sizeof(int)));

int* found = (int*)malloc(sizeof(int));
*found= NOTFOUND;

cudaMemcpy( d_found, found,sizeof(int), cudaMemcpyHostToDevice
    );

bool* h_odata = (bool*) malloc( sizeof(bool)*BYTES);

long totalRead = 0;
int numFound=1;

while(size==BYTES){


        if((fileLength-currentOffset)> BYTES)
                size= BYTES;
        else
                size = (fileLength-currentOffset);

        candidate = &fileBuffer[currentOffset];

        currentOffset+=BYTES;

        /* //CPU TEST
        memcpy(candidate,&fileBuffer[currentOffset],size);
        hosthashAndCheck(found, candidate, h_hashTable);
        */

        //GPU TEST
        cudaMemcpyToSymbol( d_cand, candidate, size * sizeof(
            char), 0, cudaMemcpyHostToDevice);
        hashAndCheck<<< grid, threads>>>(d_found, d_cand,
            d_hashTable);


        cudaThreadSynchronize();
```

```
                //GPU TEST
                cutilSafeCall( cudaMemcpy(found, d_found, sizeof(int),
                    cudaMemcpyDeviceToHost) );

                if(*found!=NOTFOUND){

                        int sigNum=confirm(*found,candidate,signatures
                            , numLoaded);
                        if(sigNum != -1){
                                numFound++;
                        }

                        currentOffset -=  ((BYTES-found[0])-1);

                }else{

                        currentOffset -= h_lengthTable[realLengths-1];

                }
                *found=NOTFOUND;

                cudaMemcpy( d_found, found,sizeof(int),
                    cudaMemcpyHostToDevice);

        }

        printf("\n\nNumber found:%d\n",--numFound);

        printf( "Processing time: %f (ms)\n", cutGetTimerValue( timer)
            );
        printf( "MB/s: %f\n", ((((double)fileLength / (
            cutGetTimerValue( timer) / 1000))/1024)/1024) );
        cutilCheckError( cutDeleteTimer( timer));


        free(fileBuffer);

        cudaFree(d_hashTable);
        cudaThreadExit();

        cutilExit(argc, argv);
}
```

B GPU Key Data



GPU	Matching Speed (MB/s)	Cores	Clock Speed (MHz)
9400m			
9500GT	34.8	32	1625
9600GT			
9800M GS	55.9	64	1325
9800GT	96.5	112	1375
8800GT	115	112	1500

Alphabet Calc

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace AlphabetCalc
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream file = new FileStream("input.txt", FileMode.
               Open, FileAccess.Read);
            int[] counts = new int[257];

            while (file.CanRead)
            {
                int theByte = file.ReadByte();
                if (theByte < 0)
                    break;
                counts[theByte]++;
            }

            int alphabetSize=0;

            for (int i = 0; i < counts.Length; i++)
            {
                if (counts[i] != 0)
                {
                    Console.Write((char)i + "␣");
                    Console.WriteLine(counts[i]);
                    alphabetSize++;
                }
            }

            Console.WriteLine("Alphabet␣Size:" + alphabetSize );

        }
    }
}
```

D Signature builder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace SigBuild
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream file = new FileStream("input.txt", FileMode.
                Open, FileAccess.Read);
            FileStream fileOut = new FileStream("signatures.txt",
                FileMode.Create, FileAccess.Write);

            byte[] buffy = new byte[int.Parse(args[2])];
            Random random = new Random();

            for (int i = 0; i < int.Parse( args[0]); i++) {
                short sigLength=0;
                while (sigLength < int.Parse(args[1]))
                    sigLength = (short)random.Next(int.Parse(args[2]))
                        ;
                file.Seek(random.Next(), SeekOrigin.Begin);
                while (file.Read(buffy, 0, sigLength) != sigLength)
                {
                    file.Seek(random.Next(), SeekOrigin.Begin);
                }

                fileOut.Write(System.BitConverter.GetBytes(sigLength),
                    0, 2);
                fileOut.Write(buffy, 0, sigLength);

            }
            file.Close();
            fileOut.Close();

        }
    }
}
```

E Distribution of alphabet in haystack

This is a listing of each character and the number of times it occured in the test haystack. If the character has a displayable mapping in the ASCII character set it is represented by that character if not it is replaced with ?. If the character is undisplayable only the count for the character appears.

```
  3284199
? 2986584
? 2896749
? 3118674
? 2897301
? 2946930
? 3139198
 2893710
 2992466
          2885248

 3010642
? 2795635
? 3083136
 2959386
? 2764846
 3198016
? 2948543
? 2934993
? 3002487
? 2727603
 3016193
 2933543
? 2814425
? 2936792
? 2954311
? 2913823
? 2807946
? 2979024
? 2886925
? 2722944
? 3117463
? 3264192
  2896183
! 2913613
" 2949020
```

\# 2941019
$ 2993971
% 2981321
\& 2795734
’ 2891656
(2912938
) 3014302
* 3088502
+ 2961476
, 2909269
- 2870291
. 2972928
/ 3063484
0 3082520
1 2820742
2 2918471
3 2874584
4 2807854
5 2776617
6 2979568
7 2930615
8 3013033
9 2859265
: 2927039
; 2758888
< 3171712
= 3059690
> 3259399
? 3169241
@ 2908803
A 2843601
B 2879010
C 2895797
D 2936147
E 2918245
F 2856730
G 3009721
H 2967638
I 2955821
J 3008723
K 2957598
L 2911721
M 2856795
N 2955249
O 3108394
P 2951457
Q 2959451
R 2936114
S 3062238
T 3108241
U 3265097
V 2972525
W 3191458
X 2861618

Y 2866192
Z 2918901
[2934834
\ 3018704
] 2936868
^ 3100832
_ 3162007
‘ 2933757
a 2916034
b 2867573
c 2868833
d 2959086
e 2939212
f 2814810
g 2983399
h 2743648
i 2888054
j 2968886
k 2811952
l 2943805
m 2996842
n 2915983
o 3071039
p 3078620
q 2871551
r 2954237
s 2964034
t 2924401
u 2871581
v 2796344
w 3008904
x 3180447
y 3053673
z 3113504
{ 3198674
| 3092795
} 3331331
˜ 3091363
 3302656
? 2994280
? 3015346
? 2970456
? 2921947
? 2979489
? 2860493
? 2910681
? 3063912
? 2892541
? 2851616
? 2942365
? 2966202
? 2796548
? 2830244
? 2852082

? 3192019
? 2867558
? 2956834
? 2964433
? 2961692
? 2956475
? 3111694
? 2930244
? 3096108
? 2820229
? 2888381
? 2773707
? 2929764
? 2979506
? 2960075
? 3084505
? 3154289
2851335
2853700
2889584
2914456
2931572
2984880
2948832
3166492
2997644
2981806
3271874
3199390
2842077
2967216
2969999
3186157
2835658
2889644
2902762
2922897
2811549
2993326
2999454
3046588
2923332
3060246
2869232
3023390
3076786
3261203
3156834
3221303
3046459
3052748
2961505
3079907
2807469

2986554
2752795
3036993
2849433
2979417
3004262
3081000
2805550
2852378
2995101
3158158
2758637
2876073
2982616
3088226
2870973
3211325
2820543
2967942
2871100
2975041
2904969
3086982
2977578
2954738
3239095
3224524
3200977
3133333
2954836
2954367
2944518
3146194
2844342
3147757
2885926
3180008
3123570
2999355
2903523
3004956
3024126
3398233
3251620
3047793
3128153
3029918
3123077
3242359
3103413
3436849
3112192
3106694
3254789

```
 3397830
 3118463
 3268128
 3296010
 3683456
Alphabet Size:256
```

F GPU brute force

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <cutil_inline.h>

#define BYTES        65536
#define NOTFOUND 100000
#define MAXSIGLENGTH 32768
#define MAXSIGS          32768

using namespace std;

bool differentLengths[MAXSIGS] = {false};
__constant__ char d_cand[BYTES];

struct signature{  //Simple structure to store a string with a length
        char data[512];
        short length;
};


__global__ void
check( int* dfound, char* candidate, struct signature* sigs, int
    numSigs)
{

        int tid = (blockIdx.x *blockDim.x ) + threadIdx.x;
        if(dfound[0]<tid)
                return;

        for(int j = 0; j<numSigs ; j++){
                short foundC = 0;
                if((sigs[j].length+tid)<BYTES){
                        int length = sigs[j].length;
                        while( foundC < length && tid+foundC <BYTES ){
                                int databyte = sigs[j].data[foundC];
                                int compbyte = d_cand[tid+foundC];
                                if(databyte!=compbyte)
                                        break;
```

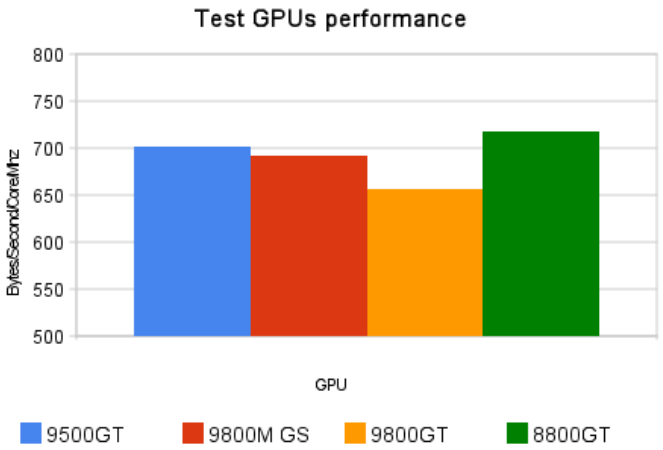
```
                                foundC++;

                        }
                        if(foundC == sigs[j].length)
                                atomicMin(dfound,tid);
                }
        }
        return;
}

int confirm(int position,char* line,struct signature * sigs,int
    numSigs){

        for(int j = 0; j<numSigs ; j++){
                int foundC = 0;
                while( foundC < sigs[j].length && sigs[j].data[foundC]
                    == line[position+foundC])
                        foundC++;
                if(foundC == sigs[j].length)
                        return j;
        }
        return -1;
}

////////////////////////////////////
// Program entry point
////////////////////////////////////
int
main( int argc, char** argv)
{
        dim3 threads(512, 1, 1);
        dim3 grid( 128 , 1, 1);

        bool* h_hashTable;

        h_hashTable = (bool*)malloc(PRIME_MOD * sizeof(bool));
        memset(h_hashTable,0,PRIME_MOD * sizeof(bool));


        char* line = (char*) malloc( sizeof(char) * MAXSIGLENGTH);
        int numLoaded=0;
        ifstream myfile("c:\\signatures.txt",ios::binary);


        struct signature *signatures = (struct signature*) malloc(
            MAXSIGS * sizeof(struct signature));

        int hashCollisions =0;
        //########## LOAD THE SIGS #########
        if (myfile.is_open())
        {
                int position =0;
                while (myfile.good() && numLoaded < MAXSIGS)
                {
```

```
                if(myfile.read(line,2).eof())
                        break;

                short lengthToRead = (((unsigned char)line[1])
                    << 8) + ((unsigned char)line[0]);

                myfile.read(line,lengthToRead );

                signatures[numLoaded].length = lengthToRead;
                memcpy(signatures[numLoaded].data,line,
                    lengthToRead);
                differentLengths[lengthToRead] = true;
                numLoaded++;
        }
        myfile.close();
}else{
        printf("No Signatures File\n");
        exit(0);
}



short* h_lengthTable = (short*)malloc(sizeof(short)*MAXSIGS);
int realLengths = 0;
//###### PRODUCE SIG LENGTHS ARRAY ######


for(int i=0;i<MAXSIGS;i++){
        if(differentLengths[i])
                h_lengthTable[realLengths++]=i;
}


char* candidate;

int size = 0;
FILE *f = fopen("c:\\input.txt", "rb");

if (f == NULL)
{
        printf("Input File Not Found\n");
        exit(1);
}

fseek(f,0,SEEK_END);
int     fileLength = ftell(f);

fseek(f,0,SEEK_SET);
char* fileBuffer = (char*)malloc(fileLength*sizeof(char));

if(fread(fileBuffer,1,fileLength,f)!=fileLength)
{
        printf("Input File Load Error\n");
        exit(1);
```

```
}

printf("File Loaded\n");

int currentOffset =0;

cudaSetDevice( cutGetMaxGflopsDeviceId() );

unsigned int timer = 0;
cutilCheckError( cutCreateTimer( &timer));
cutilCheckError( cutStartTimer( timer));

size = BYTES;

int* d_found;
cutilSafeCall( cudaMalloc( (void**) &d_found, sizeof(int)));

int* found = (int*)malloc(sizeof(int));
*found= NOTFOUND;

cudaMemcpy( d_found, found,sizeof(int), cudaMemcpyHostToDevice
    );

        struct signature* d_Sigs;
cutilSafeCall( cudaMalloc( (void**) &d_Sigs, numLoaded *
    sizeof(struct signature)));
cutilSafeCall( cudaMemcpy( d_Sigs, signatures, numLoaded *
    sizeof(struct signature), cudaMemcpyHostToDevice) );

bool* h_odata = (bool*) malloc( sizeof(bool)*BYTES);

long totalRead = 0;
int numFound=1;

while(size==BYTES){

        if((fileLength-currentOffset)> BYTES)
                size= BYTES;
        else
                size = (fileLength-currentOffset);

        candidate = &fileBuffer[currentOffset];
        currentOffset+=BYTES;

        cudaMemcpyToSymbol( d_cand, candidate, size * sizeof(
            char), 0, cudaMemcpyHostToDevice);

        check<<< grid, threads>>>(d_found, d_cand, d_Sigs,
            numLoaded);

        cudaThreadSynchronize();

        cutilSafeCall( cudaMemcpy(found, d_found, sizeof(int),
             cudaMemcpyDeviceToHost) );
```

```
                cudaGetLastError();

                if(*found!=NOTFOUND){

                        int sigNum=confirm(*found,candidate,signatures
                            ,numLoaded);
                        if(sigNum != -1){
                                numFound++;
                        }

                        currentOffset -= ((BYTES-found[0])-1);

                }else{

                        currentOffset -= h_lengthTable[realLengths-1];

                }
                        printf("%d\n",currentOffset);
                *found=NOTFOUND;

                cudaMemcpy( d_found, found,sizeof(int),
                    cudaMemcpyHostToDevice);

        }

        printf("\n\nNumber found:%d\n",--numFound);
        printf( "Processing time: %f (ms)\n", cutGetTimerValue( timer)
            );
        printf( "MB/s: %f\n", ((((double)fileLength / (
            cutGetTimerValue( timer) / 1000))/1024)/1024) );
        cutilCheckError( cutDeleteTimer( timer));

        free(fileBuffer);

        cudaFree(d_Sigs);
        cudaThreadExit();
        cutilExit(argc, argv);
}
```

Bloom filter test data



GPU	Matching Speed (MB/s)	Cores	Clock Speed (MHz) / 10	Clock Speed (MHz)	MB/s per core	GPU	Bytes per core per million cylces	Average Cycles per bit
9500GT	34.8	32	162.5	1625	1.0875	9500GT	701.739323076923	178.128823466685
9800M GS	55.9	64	132.5	1325	0.8734375	9800M GS	691.219320754717	180.839852484906
9800GT	96.5	112	137.5	1375	0.86160714285714	9800GT	657.062233766234	190.240731511091
8800GT	115	112	150	1500	1.02678571428571	8800GT	717.775238095238	174.149222995924

H Boyer-Moore Implementation

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <cutil_inline.h>

#define BYTES        65536
#define NOTFOUND 100000
#define MAXSIGLENGTH 32768
#define MAXSIGS          32768

using namespace std;

bool differentLengths[MAXSIGS] = {false};
__constant__ char d_cand[BYTES];

struct signature{  //Simple structure to store a string with a length
        char data[512];
        short length;
};


__host__ void preBmBc(char *x, int m, int bmBc[]) { // int bmBc[]) {
   int i;

   for (i = 0; i < 256; ++i)
      bmBc[i] = m;
   for (i = 0; i < m - 1; ++i)
      bmBc[x[i]] = m - i - 1;
}


__host__ void suffixes(char *x, int m, int *suff) {
   int f, g, i;

   suff[m - 1] = m;
   g = m - 1;
```

```
   for (i = m - 2; i >= 0; --i) {
      if (i > g && suff[i + m - 1 - f] < i - g)
         suff[i] = suff[i + m - 1 - f];
      else {
         if (i < g)
            g = i;
         f = i;
         while (g >= 0 && x[g] == x[g + m - 1 - f])
            --g;
         suff[i] = f - g;
      }
   }
}

__host__ void preBmGs(char *x, int m, int bmGs[]) { //int bmGs[]) {
   int i, j, suff[512];

   suffixes(x, m, suff);

   for (i = 0; i < m; ++i)
      bmGs[i] = m;
   j = 0;
   for (i = m - 1; i >= 0; --i)
      if (suff[i] == i + 1)
         for (; j < m - 1 - i; ++j)
            if (bmGs[j] == m)
               bmGs[j] = m - 1 - i;
   for (i = 0; i <= m - 2; ++i)
      bmGs[m - 1 - suff[i]] = m - 1 - i;
}



__host__ void precompKernel( struct signature* sigs, int* tableHeap,
    int numSigs){


        for(int i=0;i<numSigs;i++){

                int m = sigs[i].length;


                int* bmGs = &tableHeap[i*1024];
                int* bmBc = &tableHeap[(i*1024)+512]; //[256];

                // Preprocessing
                preBmGs(sigs[i].data, m, bmGs);
                preBmBc(sigs[i].data, m, bmBc);

        }

}
```

```
__global__ void
testKernel(int* dfound, int numSignatures, struct signature* sigs, int
    * tableHeap)
{




        // access thread id
        unsigned int tid = (blockIdx.x *blockDim.x ) + threadIdx.x;

        if(tid >= numSignatures || dfound[0] < NOTFOUND)
                return;


        char* x= sigs[tid].data;
        int m = sigs[tid].length;
        int n = BYTES;
        char* y = d_cand;

        int i, j;


        int* bmGs = &tableHeap[tid*1024];
        int* bmBc = &tableHeap[(tid*1024)+512];

        //preBmGs(sigs[tid].data, m, bmGs);
        //preBmBc(sigs[tid].data, m, bmBc);


        // Searching
        j = 0;
        while (j <= n - m) {
          for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
          if (i < 0) {
                  atomicMin(dfound,j);
                  return;
          }
          else{
                if(bmGs[i] > bmBc[y[i + j]] - m + 1 + i)
                        j+= bmGs[i];
                else
                        j+=bmBc[y[i + j]] - m + 1 + i;

           }

        }
}
```

```
__global__ void
check( int* dfound, char* candidate, struct signature* sigs, int
    numSigs)
{

        int tid = (blockIdx.x *blockDim.x ) + threadIdx.x;
        if(dfound[0]<tid)
                return;

        for(int j = 0; j<numSigs ; j++){
                short foundC = 0;
                if((sigs[j].length+tid)<BYTES){
                        int length = sigs[j].length;
                        while( foundC < length && tid+foundC <BYTES ){
                                int databyte = sigs[j].data[foundC];
                                int compbyte = d_cand[tid+foundC];
                                if(databyte!=compbyte)
                                        break;
                                foundC++;

                        }
                        if(foundC == sigs[j].length)
                                atomicMin(dfound,tid);
                }
        }
        return;
}

int confirm(int position,char* line,struct signature * sigs,int
    numSigs){

        for(int j = 0; j<numSigs ; j++){
                int foundC = 0;
                while( foundC < sigs[j].length && sigs[j].data[foundC]
                    == line[position+foundC])
                        foundC++;
                if(foundC == sigs[j].length)
                        return j;
        }
        return -1;
}
```

```
////////////////////////////
// Program entry point
////////////////////////////
int
main( int argc, char** argv)
{

        dim3 grid( 1 , 1, 1);

        bool* h_hashTable;


        char* line = (char*) malloc( sizeof(char) * MAXSIGLENGTH);
        int numLoaded=0;
        ifstream myfile("c:\\signatures.txt",ios::binary);


        struct signature *signatures = (struct signature*) malloc(
           MAXSIGS * sizeof(struct signature));

        int hashCollisions =0;
        //###### LOAD THE SIGS #######
        if (myfile.is_open())
        {
                int position =0;
                while (myfile.good() && numLoaded < MAXSIGS)
                {
                        if(myfile.read(line,2).eof())
                                break;

                        short lengthToRead = (((unsigned char)line[1])
                            << 8) + ((unsigned char)line[0]);

                        myfile.read(line,lengthToRead );

                        signatures[numLoaded].length = lengthToRead;
                        memcpy(signatures[numLoaded].data,line,
                            lengthToRead);
                        differentLengths[lengthToRead] = true;
                        numLoaded++;
                }
                myfile.close();
        }else{
                printf("No Signatures File\n");
                exit(0);
        }

        dim3 threads(numLoaded, 1, 1);


        short* h_lengthTable = (short*)malloc(sizeof(short)*MAXSIGS);
        int realLengths = 0;
        //##### PRODUCE SIG LENGTHS ARRAY #####
```

```
for(int i=0;i<MAXSIGS;i++){
        if(differentLengths[i])
                h_lengthTable[realLengths++]=i;
}


char* candidate;

int size = 0;
FILE *f = fopen("c:\\input.txt", "rb");

if (f == NULL)
{
        printf("Input File Not Found\n");
        exit(1);
}

fseek(f,0,SEEK_END);
int     fileLength = ftell(f);

fseek(f,0,SEEK_SET);
char* fileBuffer = (char*)malloc(fileLength*sizeof(char));

if(fread(fileBuffer,1,fileLength,f)!=fileLength)
{
        printf("Input File Load Error\n");
        exit(1);
}

printf("File Loaded\n");

int currentOffset =0;

cudaSetDevice( cutGetMaxGflopsDeviceId() );

unsigned int timer = 0;
cutilCheckError( cutCreateTimer( &timer));
cutilCheckError( cutStartTimer( timer));

size = BYTES;

int* d_found;
cutilSafeCall( cudaMalloc( (void**) &d_found, sizeof(int)));

int* found = (int*)malloc(sizeof(int));
*found= NOTFOUND;

cudaMemcpy( d_found, found,sizeof(int), cudaMemcpyHostToDevice
    );

        struct signature* d_Sigs;
```

```
    cutilSafeCall( cudaMalloc( (void**) &d_Sigs, numLoaded *
        sizeof(struct signature)));
    cutilSafeCall( cudaMemcpy( d_Sigs, signatures, numLoaded *
        sizeof(struct signature), cudaMemcpyHostToDevice) );



    //#### MAKE TABLES ####

    int *tableHeap = (int*)malloc(sizeof(int)*1024*numLoaded);
memset(tableHeap,0,sizeof(int)*1024*numLoaded);

    precompKernel( signatures, tableHeap, numLoaded);

    int* d_tableHeap;
    cutilSafeCall( cudaMalloc( (void**) &d_tableHeap,sizeof(int)
        *1024*numLoaded));

    cutilSafeCall( cudaMemcpy( d_tableHeap,tableHeap,sizeof(int)
        *1024*numLoaded, cudaMemcpyHostToDevice) );

    //######################

    bool* h_odata = (bool*) malloc( sizeof(bool)*BYTES);

    long totalRead = 0;
    int numFound=1;

    while(size==BYTES){

            if((fileLength-currentOffset)> BYTES)
                    size= BYTES;
            else
                    size = (fileLength-currentOffset);

            candidate = &fileBuffer[currentOffset];
            currentOffset+=BYTES;

            cudaMemcpyToSymbol( d_cand, candidate, size * sizeof(
                char), 0, cudaMemcpyHostToDevice);

            //check<<< grid, threads>>>(d_found, d_cand, d_Sigs,
                numLoaded);
            testKernel<<< grid, threads>>>(d_found, numLoaded,
                d_Sigs, d_tableHeap);

            cudaThreadSynchronize();

            cutilSafeCall( cudaMemcpy(found, d_found, sizeof(int),
                 cudaMemcpyDeviceToHost) );
            cudaGetLastError();

            if(*found!=NOTFOUND){
```

```
                        int sigNum=confirm(*found,candidate,signatures
                            ,numLoaded);
                        if(sigNum != -1){
                                numFound++;
                        }

                        currentOffset -= ((BYTES-found[0])-1);

                }else{

                        currentOffset -= h_lengthTable[realLengths-1];

                }
                        printf("%d\n",currentOffset);
                *found=NOTFOUND;

                cudaMemcpy( d_found, found,sizeof(int),
                    cudaMemcpyHostToDevice);

        }

        printf("\n\nNumber found:%d\n",--numFound);
        printf( "Processing time: %f (ms)\n", cutGetTimerValue( timer)
            );
        printf( "MB/s: %f\n", ((((double)fileLength / (
            cutGetTimerValue( timer) / 1000))/1024)/1024) );
        cutilCheckError( cutDeleteTimer( timer));

        free(fileBuffer);

        cudaFree(d_Sigs);
        cudaThreadExit();

        cutilExit(argc, argv);
}
```

A tree has be planted to offset the power used during the production of this paper.