University of California

Los Angeles

Accelerating String Matching Applications Using GPUs

A report submitted in partial satisfaction

of the requirements for the degree

Master of Science in Computer Science

by

Stephen J. Oakley

2009

The report of Stephen J. Oakley is approved.

Petros Faloutsos

Adnan Darwiche

Glenn Reinman, Committee Chair

University of California, Los Angeles

2009

To my family . . .

who—among so many other things—

saw to it that I dedicate my heart to my work.

And to my loving girlfriend and editor, Sona Chaudhuri,

who made sure my mug was full and my mind at peace.

Table of Contents

SECTION 1

Introduction

Recent advances in the programmability of discrete graphics cards has lead to a growing trend in the acceleration of compute intensive applications using GPUs. Such applications have demonstrated several orders of magnitude speedups. This work attempts to gain similar performance improvements for two string matching applications.

The first application is a computer virus scanner. The purpose of the application is to scan through all files on a system and search for predefined strings known as virus signatures. Virus signatures are strings of bits which uniquely identify a virus within a file on the system. This signature based detection approach serves as the basis for all industry grade virus scanning software currently available on the market.

The problem of iterating through every file on a system and searching for known virus signatures is computationally intensive because modern virus databases contain several million known signatures. While techniques have been developed to quickly prune the set of candidate signatures which can potentially exist within a given file, even with the fastest known string matching algorithms, virus scanners are still very slow and CPU intensive.

Both of these problems are undesirable in most systems on which virus scanners are deployed. Perhaps the largest market for commercial virus scanners

is that of home and small business consumers. That being said, users will not tolerate significant degredation in system performance while performing scans.

The second application that has been explored is a genetic sequence alignment program. Until recently, genomic sequencing was considered difficult and expensive, and thus prohibited sequencing of large sets of individuals. However, with the advent of short read sequencers this is no longer the case. Short read sequencers produce a set of short reads from random positions with the sample genome. In order to reconstruct the complete genomic sequence, the reads must be aligned, also known as resequenced, to a reference genome from the same species. This is possible because there is known to be less than one percent variation among individuals in the human species. These points of single base pair variations are known as SNPs. The complexity of sequence allignment stems from the fact that the human genome has over three billion base pairs.

The contribution of this work is a new approach to short read alignment. Instead of viewing the problem as mapping a set of reads onto a reference genome, we will frame it in terms of finding all the positions in the genome to which a read can be mapped. This simple, yet powerful transformation allows us to use the very same approach for resequencing as we do for virus scanning. In addition it can significantly reduce the memory overhead associated with hash based indexing as used in [ST], and prevents working on the same piece of the reference genome multiple times.

The basis of the work performed by sequence alignment is very similar to that of the virus scanners despite the fact that the resquencers allow for imperfect matches within some threshold. The most significant difference between the task of the two applications is that the length of the reads being matched by the resequencers typically range in the hundreds to thousands of characters, whereas

the virus signatures are typically under 256 bytes. The final noteworthy distinction is that the reads from the genome are from a very small alphabet containing only four characters, and the virus signatures are from an alphebet of one byte characters.

Throughout the remainder of this work, we explore a set of techniques that attempt to be well suited for both applications while producing significant performance improvements. In section two, we will explore the CUDA compute architecture and corresponding programming considerations. Section three outlines our approach to accelerate these two applications. Section four presents preliminary performance evaluations of the solutions, while section five contains concluding remarks and future works.

SECTION 2

GPU Architecture

The CUDA Unified Compute Architecture is a combination of hardware, drivers and application level libraries that allow developers to design and implement programs that execute on NVidia GPUs (see figure 2.1). The central theme of the model is a single program multiple data (SPMD) approach to processing as described by [NVI08].

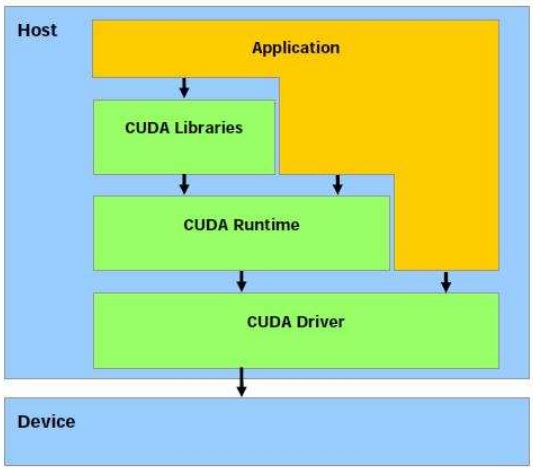

Figure 2.1: CUDA Unified Compute Architecture

NVidia achieves this through their thread hierarchy abstraction. All threads associated with a given kernel are grouped into thread blocks which can be structured with up to three dimensions of threads. Thread blocks are then organized into grids which can also contain up to three dimensions of thread blocks.

A given thread block is executed on a single streaming processor (SP), which is a collection of eight simple, in-order execution cores. Thread blocks within a grid are scheduled by the CUDA runtime across the multiple SPs that make up the GPU.

In addition to the thread hierarchy, CUDA enabled devices have a complex memory hierarchy that developers must be mindful of when they are tuning applications for performance. At the bottom of the hierarchy is the global memory of the GPU. Above the global memory there are per SP constant and texture caches to accelerate memory accesses. Finally, the fastest and smallest memory is the per thread block shared memory which resides within each SP.

However, with CUDA it is not simply enough to be cognizant of the memory hierarchy, but also the memory management subsystem. To achieve optimal performance, it is neccesary that global memory accesses be coalesced so that the memory subsystem can bundle multiple requests into a single larger request. The requirements for memory coalescing vary by device architecture, and thus we refer you to [NVI08] for more information.

A final consideration for performance oriented implementations is that threads within a thread block are executed in a set of warps. Warps are a set of up to 32 threads that have the same instruction counter. During execution, a single warp is executed on the SP at a time, and are switched out on memory stalls or when a maximum instruction count is completed. When the code executes a divergent branch, the different paths are broken into new warps containing the corresponding threads for each trace. This implies that if the code contains many divergent branches the code becomes serialized by the execution of multiple warps resident on each SP.

Processor Cores	112
Graphics Clock (MHz)	600 MHz
Processor Clock (MHz)	1500 MHz
Texture Fill Rate (billion/sec)	33.6
Memory Clock (MHz)	900 MHz
Standard Memory Config	512 MB
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	57.6
Maximum GPU Temperature (in C)	105 C
Maximum Graphics Card Power (W)	105 W

Figure 2.2: Features of NVidia GeForce 8800GT

Form Factor	10.5" x 4.376", Dual Slot
# of Tesla GPUs	1
# of Streaming Processor Cores	240
Frequency of processor cores	1.3 GHz
Single Precision floating point performance (peak)	933
Double Precision floating point performance (peak)	78
Floating Point Precision	IEEE 754 single & double
Total Dedicated Memory	4 GB GDDR3
Memory Speed	800MHz
Memory Interface	512-bit
Memory Bandwidth	102 GB/sec
Max Power Consumption	187.8 W

Figure 2.3: Features of NVidia Tesla C1060

SECTION 3

Design Overview

In this section we will discuss the techniques and innovations used to accelerate the two target applications. We will pay particular attention to the data structures and latency hiding techniques employed. For reference purposes, the implementations of both the virus scanner and the resequencer in C++ contain just over fourteen thousand lines of code.

We will first discuss the techniques employed on the virus scanner as they apply to both applications. The tricks played in the resequencer are simple extensions of those found in the virus scanner.

3.1 Virus Scanner

We began our work by studying the techniques used by the open source virus scanner ClamAV. The scanner primarily uses the Aho-Corasick string matching algorithm, [AC75]. The prefix tree built by the AC algorithm allows the application to drastically prune the number of signatures that require a full comparison against a given region of a file. We build upon the ideas utilized by ClamAV with several extensions to facilitate asynchronous data transfers between the CPU and GPU and vice versa.

3.1.1 CPU

The CPU portion of the code is divided into the following four sections:

1. **Signature Manager** : The signature manager maintains an Aho-Corasick style prefix tree. The tree is restricted to a configurable depth d. At the leaves of the tree reside a set of signature chunks, which each contain a fixed number of candidate signatures. The chunks are stored in page locked memory so that the pointers to the chunk can be used with cudaMemcpyAsync. Given a prefix, the signature manager uses the AC tree to quickly identify the set of all candidate signature chunks.

2. **File Buffer Manager** : A resource manager for maintaining a set of fixed sized page lock file buffers suitable for use with cudaMemcpyAsync. Each piece of a file which is read by the application is stored in a buffer managed by this component.

3. **File Scanner** : This component iterates over the file system to be scanned, and for every file buffer produces a work unit and submits it to the GPU Scheduler. A work unit consist of a file buffer and the set of candidate chunks to be scanned.

4. **GPU Scheduler** : The GPU scheduler is responsible for scheduling memory copies and compute kernel invocations to the GPU. The scheduler attempts to exploit signature locality on the GPU. That is, it schedules kernels for launch whose set of candidate chunks have the most in common with those already resident in the GPU main memory.

   There are currently two implemented versions of the GPU scheduler. The first is a naive scheduler which simply invokes scans as they arrive in the

work queue. The second uses a window based optimization scheme over the set of candidate IDs for pending work under the constraints of what chunks are resident on the GPU. The optimizing scheduler sorts pending work units in descending order based on how many unique, new signature chunks that would have to be transferred to the GPU.

3.1.2 GPU

Similar to the structure of the CPU, the GPU structures are responsible for maintaining the state of the GPU resources. The revelent components are as follows:

1. **Signature Cache** : The signature cache acts much like a data cache found in modern processor technologies, and actively caches the set of in-use signature chunks. This allows us to avoid having duplicant chunks resident on the GPU. It maintains a set of fixed sized blocks which corresponds to the size of signature chunks. The cache is fully associative, and thus there is no need for complicated addressing schemes. The cache can be explicitly managed via the GPU scheduler. If insertions are unmanaged, the cache defaults to an oldest-first eviction policy. The number of blocks in the cache is optimized to minimize the number of signature chunk transfers while trying to maximize the number of file buffers that can be resident on the GPU. This delicate balance between signature chunks and file buffers allows us to maintain a high level of utilization of GPU computational resources.

2. **GPU Buffer Manager** : Similar to the CPU based file buffer manager, the GPU buffer manager is responsible for maintaining all portions of GPU memory dedicated to fized size buffers. There are two instances of the GPU

buffer manager during an execution of the program. The first is responsible for maintaining all GPU file buffers, and the second handles all result buffers used by the scan kernel.

3. **Scan Kernel** : This component is the scan kernel executed on the GPU which does all the heavy lifting. The kernel scans the entire file buffer for the presence of all candidate signatures identified by the CPU Aho-Corasick tree preprocessing step. The kernel then reports all signature matches found within the buffer and terminates for the given file buffer. Many instances of the scan kernel are active at any given point during execution for a set of distinct file buffers.

   During the early phases of development, we developed a non-branching string matching kernel. This kernel would scan the entire length of a signature, regardless of whether a mismatch was found. This approach was designed around the principle of an algebraic difference metric between the file substring and the virus signature. If the computed metric was zero at the end of the scan, a match between the buffer and a signature was found. The metric used was the sum of the XORs between corresponding characters in the file buffer and the virus signature.

   Despite the fact that the non-branching kernel did indeed significantly reduce thread serialization with the thread blocks, it performed far slower than even the most naive scan kernels. This is a result of the fact that the increase in the amount of comparison performed, registers used and uncoalesced memory accesses produced by each thread significantly reduced the number of concurrent threads executable by the SPs.

All parameters for the management structures, from chunk size to tree depth, are configurable to allow for performance tuning to individual GPU architectures.

In practice we have found that by minimizing the depth of the prefix tree in the signature manager, the CPU load can be reduced, while increasing the occupancy of the GPU.

3.2 Resequencer

Due to the limited alphabet of the genetic sequences, it is possible to perform what is known as base-pair compression. Base-pair compression is a technique based on the observation that a single base-pair occupies only two bits of memory. As such it is possible to pack up to four base-pairs into a single ASCII character. This technique increases the effective depth of the Aho-Corasick prefix tree by a factor of four per level of the tree.

Since the reads used by the resequencer are much longer than the virus signatures, it is beneficial to have a much deeper prefix tree in practice. This allows the application to quickly reduce the number of candidate reads for a given portion of the genome, and thus allows us to increase the length of the sections of the genomes which are being scanned.

SECTION 4

Results

4.1 Virus Scanner

This section will present the prelimary results obtained by evaluating the performance of the GPU based virus scanner. The baseline for comparison is the ClamAV virus scanner which achieved a maximum scan throughput of roughly 9 MB/sec with a signature set of 250,000 signatures. All signature sets used in the evaluation of the GPU implementation are randomly generated, and thus may not be truly representative of real virus databases. However, it can be argued that since the signatures are the result of a uniform psuedo-random number generator, they are likely to produce conservative performance estimates, since they are likely to produce more candidates per buffer. However, in order to provide a relative comparison against the CPU we implemented both a naive, and a Boyer-Moore,[BM77], string comparison kernels to be run on the CPU.

The results depicted in figure 4.1 illustrate the file throughput of the GPU based scanner when executed over varying size signature databases. The test was performed using a 1MB file buffer, and a scheduling window of length 32. As you can see, performance decreases as signature set increases. This is due to the fact that as the database grows, so does the number of candidate signatures per file buffer. Also of note, is the fact that the performance becomes I/O bound for small signature databases, because the application spends relatively more time

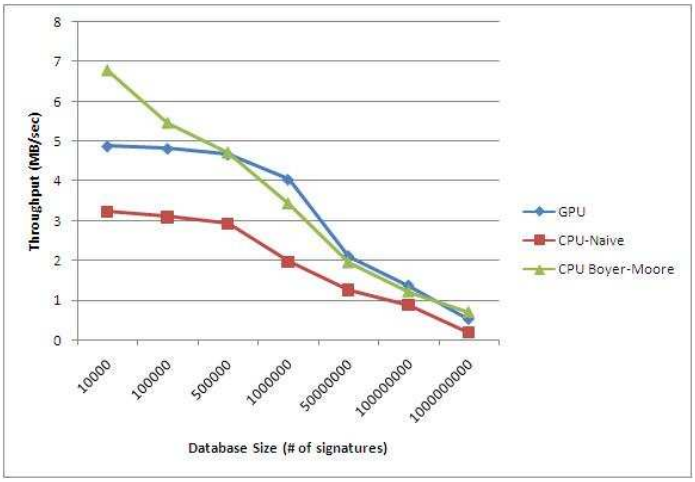

Figure 4.1: Scan throughput as a function of database size.

waiting for file buffers to be transfered over the PCI Express bus.

In comparison to the relative performance of the CPU, we find that the CPU outperforms the GPU based approach for small database sizes. This is due to the fact that the CPU does not incur the transfer latencies of data transfers before a scan. However, for a database around the one million mark, the GPU outperforms the CPU by roughly one half a MB/sec. This does not illustrate the substantial improvements that we hoped for, but shows promise for future improvement.

Figure 4.2 illustrates the effect the choice of file buffer size has on the maximum throughput of the scanner. The test was run using a database of one million signatures and the naive scheduler. As you can see, when small file buffers are used, the computational occupancy of the GPU drops off and execution time becomes dominated by tranfer latencies. We have found that the optimal file buffer size is one megabyte. Using a one megabyte file buffer, and using the CUDA profiler, we were able to see that the device occupancy was 100 percent.

It is important to note that as buffer size increases, the number of candidate

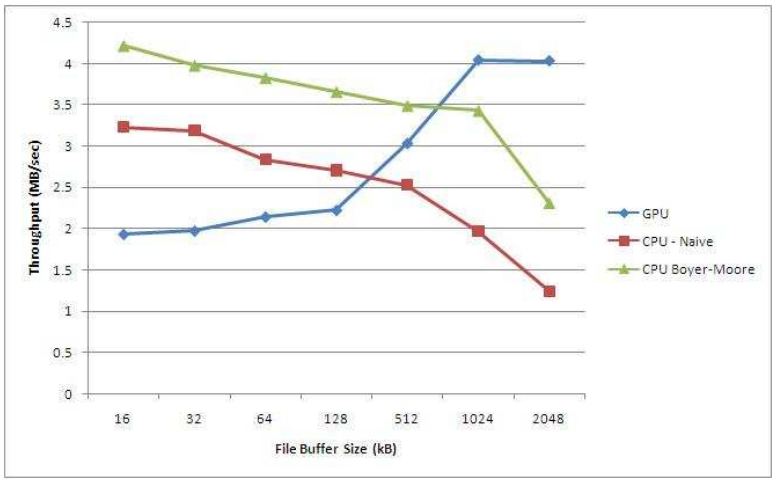

Figure 4.2: Scan throughput as a function of file buffer size.

signature chunks to be compared drastically increases. It is from this data that we best see the power of the high thread concurrency of the GPU. As the buffer sizes increase, the performance exhibited by the CPU rapidly degrades, whereas the GPU performance improves.

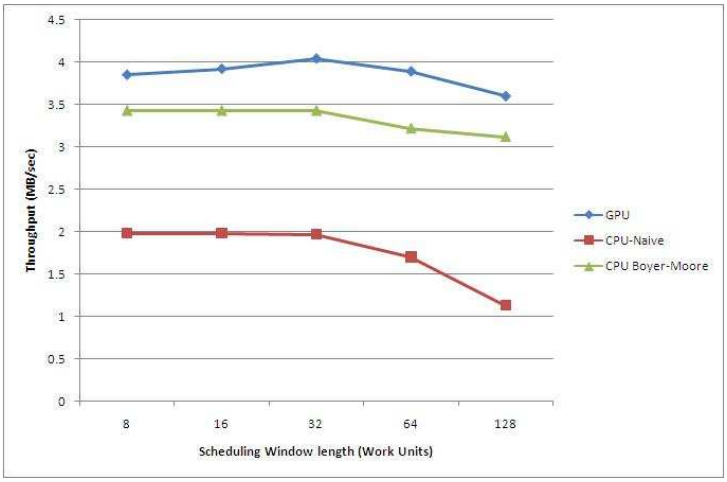

Figure 4.3: Scan throughput as a function of the length of the scheduling window.

Finally, figure 4.3 illustrates the effect of the length of the scheduling window on the performance of the GPU. The test was performed over a database of one

million signatures, and a file buffer size of 1MB. The maximum throughput was obtained using a window of length 32. With smaller windows, the scheduler was not able to accurately predict the signature chunk usage to best exploit locality, and with longer windows, the time to schedule work units began to dominate execution times. It should be noted that the performance of the CPU based approach also degrades as window size increases.

4.2 Resequencer

In order to evaluate the resequencer, it is necessary to examine not only the design parameters considered for the virus scanner, but also the impact that the length of the read has on the throughput. Our baseline for comparison is the MUMmer sequence aligner which can achieve a throughput of 5 mega base-pairs per sec (MBP/sec) for exact match alignment (i.e. no SNPs). The following will evaluate our GPU based approach, with exact matches, over the same genome and set of reads as used by MUMmer. It is important to note that the evaluation of the scheduling window and buffer size closely mimic the results obtained by the virus scanner, and thus are not replicated here.

As you can see, the GPU implementation's performance increases as the number of reads and the read length increases. The only exception to this is the case with reads of length 1024 base-pairs and 60 million reads. The performance degraded due to the fact that the CPU began to run out of memory, and thus could no longer allocate new resources for work units. The increase in performance with the increase in problem size is due to the fact that the PCI Express bus has long latencies and high bandwidth. This means that for larger datasets, the memory transfers remain roughly constant in time, but far more work is being performed per kernel invocation. As shown in the graph for long reads, and a large number

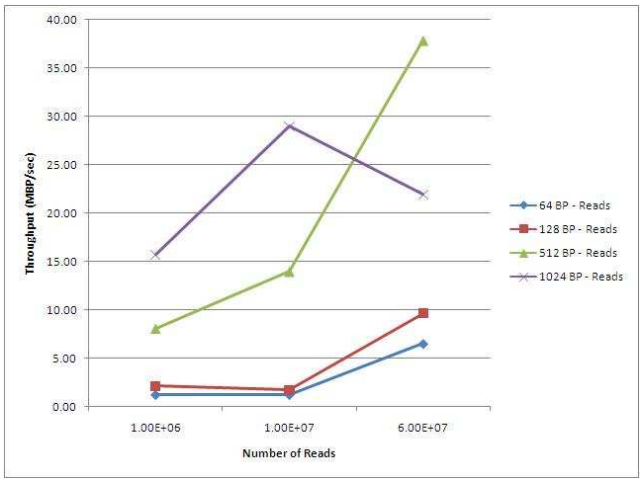

Figure 4.4: Resequence throughput as a function of read length.

of reads, the GPU significantly outperforms the CPU based MUMmer sequence aligner.

In summary, the virus scanner produced approximately half the throughput of the ClamAV system; however, it used a larger signature set. When contrasted with the purely CPU based implementation of our approach, the GPU demonstrates potential performance benefits beyond what can be achieved with the CPU alone. One the other hand, with the resequencer, the GPU produces significant improvements over the performance of the CPU. This is due to the fact that the computational granularity of the resequencer is much coarser than that of the virus scanner. Additionally, we were able to illustrate the drastic impact that parameter tuning has on striking the balance between CPU and GPU and between transfer latencies and device occupancy.

SECTION 5

Conclusion

In conclusion, GPUs provide a powerful acceleration platform given that the application is compute intensive. Some of the results presented suggest that virus scanning may not exhibit the characteristics needed for several orders of magnitude speedups, but it still has the potential to gain considerable advantages over a purely CPU based approach. Our system needs to be evaluated to discover the performance bottlenecks, and to ensure that we are using the GPU to the maximum of its capabilities.

In the future we will explore what performance benefits can be gained by preprocessing reads and virus signatures to ensure that they are in prefix sorted order. This small change would allow for better memory utilization and even allow for improved locality of candidate strings on the GPU. Additionally, we will experiment with using GPU texture caches to combat the limitations of the problem structure to produce coalesced reads from the GPU main memory. Another area of research is the use of thread blocked shared memory to decrease memory latencies experienced by the individual threads. The last, and possibly most important area of research is the evaluation of the system against commercial virus databases.

References

[AC75] Alfred V. Aho and Margaret J. Corasick. "Efficient string matching: an aid to bibliographic search." *Commun. ACM*, **18**(6):333–340, June 1975.

[BM77] Robert S. Boyer and J. Strother Moore. "A fast string searching algorithm." *Commun. ACM*, **20**(10):762–772, 1977.

[NVI08] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[ST] Michael C. Schatz and Cole Trapnell. "Fast Exact String Matching on the GPU." Technical report.